

Contents

ACMX2 Programming Guide	38
LostSideDead - ACMX2 Technical Reference	38
acidcam-gpu + ACMX2 - Complete Technical Reference	38
Program Screenshot	39
Project Origin and Purpose	39
Why CUDA Kernels and GLSL Shaders?	40
How Each Technology Is Used	40
ACMX2 Qt Application - Orchestration Layer	40
How the Qt Interface Works	41
Standalone CLI - acidcam-gpu/app/main_cv.cu	41
Startup: Device and Camera Discovery	41
AnimationState The Per-Frame Parameter Evolution Engine	41
updateAndDraw() The Frame Processing Function	42
Output Encoding via MXWrite	42
The Public API ac-gpu.hpp	42
Key Types and Definitions	42
The DynamicFrameBuffer Class	42
The launch_filter() Function	43
FilterParams The Per-Frame Parameter Bundle	43
The Unified Kernel unifiedFilterKernel	44
Thread Layout and Pixel Assignment	44
The Filter Chain Loop	44
Device Helper Functions	45
The Filter Catalog 905 Effects Across All Categories	45
Temporal / Trail Effects	45
Geometric Distortion	46
Color Manipulation	46
Bitwise / XOR Operations	46
Glitch and Noise	46
Scan / Line Effects	46
Block / Square Effects	47
Wave / Oscillation Effects	47
Mirror / Reflection	47
Pixel Read / Strobe / Blend Collections	47
Advanced / Cinematic Effects	48
The Two-Stage GPU Pipeline CUDA Compute + GLSL Shaders	48
Stage 1: CUDA Compute Kernels Massively Parallel Pixel Processing	48
Stage 2: GLSL Shaders Full-Frame Post-Processing	48
The Transfer: Zero-Copy GPU Interop	49
Filter and Shader Stacking Two Independent Composable Chains	49
CUDA Filter Chain (Stacking Inside the Kernel)	49
GLSL Shader Pass Stack (Ping-Pong Framebuffer Passes)	49
Combined Stacking: The Full Pipeline	50
Why Stacking Produces Infinite Visuals	50
In-Depth: Why Order Dominates Output	50
Build System, Dependencies, and Distribution	51
Required Dependencies	51
Build Order	51
Container Distribution (Podman)	51
Development Environment	51
Source Code Map by Project Part	52
ACMX2 Core	52
ACMX2 Media + Audio + 3D	52

ACMX2 Interface + Tools	52
acidcam-gpu Library	52
Ops + Deployment	52
Expanded Meaning of Each Code Map Item	52
End-to-End Runtime Flow	53
Detailed Runtime Inner Workings	53
Project Architecture Goals and Design Tradeoffs	53
1) Composability First	53
2) Deterministic Runtime Paths	54
3) Throughput-Oriented Execution	54
4) Operator Visibility	54
Engineering Implications	54
Build, Deployment, and Operational Model	54
Build Surface	54
Operational Surface	54
Project Summary	54
 AcidCam GPU - Effects Reference	 55
 Part I - CUDA Filters	 55
Unified Filter Kernel	55
Core Processing (0-99)	101
Filter #0 - SelfAlphaBlend	101
Filter #1 - MedianBlend	102
Filter #2 - MedianBlendXor	103
Filter #3 - SquareBlockResize	104
Filter #4 - SelfScaleRefined	104
Filter #5 - StrangeGlitch	105
Filter #6 - MatrixOutline	105
Filter #7 - AuraTrails	106
Filter #8 - MirrorReverseColor	106
Filter #9 - SquareShrink	107
Filter #10 - MotionGhostTrails	107
Filter #11 - StretchColMatrix8	107
Filter #12 - StretchColMatrix16	108
Filter #13 - StretchColMatrix32	108
Filter #14 - GradientFlashColor	108
Filter #15 - HorizontalGlitch	108
Filter #16 - VerticalGlitch	109
Filter #17 - WaveTrails	109
Filter #18 - PixelInterlace	109
Filter #19 - ColorWaveTrails	110
Filter #20 - ParticleSlide	110
Filter #21 - DiagPixelated	111
Filter #22 - DiagPixelatedResize	111
Filter #23 - RGBShiftTrails	112
Filter #24 - PictureShiftDown	112
Filter #25 - PictureShiftRight	113
Filter #26 - PictureShiftVariable	113
Filter #27 - StretchR_Right	113
Filter #28 - StretchG_Right	113
Filter #29 - StretchB_Right	114
Filter #30 - StretchR_Down	114
Filter #31 - StretchG_Down	114
Filter #32 - StretchB_Down	114
Filter #33 - Distorted_LinesY	115
Filter #34 - Distorted_LinesX	115

Filter #35 - TripHSV	115
Filter #36 - XorSumStrobe	117
Filter #37 - DetectEdges	117
Filter #38 - SobelNorm	117
Filter #39 - LineInLineOut	118
Filter #40 - LineInLineOut4_Increase	119
Filter #41 - LineInLineOut_ReverseIncrease	119
Filter #42 - LineInLineOut_ReverseIncrease2	119
Filter #43 - LineInLineOut_InvertedY	120
Filter #44 - LineInLineOut_ReverseInvertedY	120
Filter #45 - LineInLineOut_Vertical	121
Filter #46 - LineInLineOut_VerticalIncrease	121
Filter #47 - LineInLineOut_IncreaseImage	121
Filter #48 - SquareByRow	122
Filter #49 - SquareByRowRev	122
Filter #50 - SquareByRow2	123
Filter #51 - DivideByValue	124
Filter #52 - ColorCollectionSubtleStrobe	124
Filter #53 - CollectionRandom	125
Filter #54 - CollectionAlphaXor	125
Filter #55 - ColorCollection64X	126
Filter #56 - ColorCollectionSwitch	126
Filter #57 - ColorCollectionRGB_Index	127
Filter #58 - ColorCollectionGhostTrails	127
Filter #59 - ColorCollectionScale	128
Filter #60 - ColorCollectionReverseStrobe	128
Filter #61 - ColorCollectionXorPixel	129
Filter #62 - BlendWithSource25	130
Filter #63 - BlendWithSource50	130
Filter #64 - BlendWithSource75	130
Filter #65 - BlendWithSource100	131
Filter #66 - ColorCollectionXorOffsetFlash	131
Filter #67 - ColorCollectionMatrixGhost	132
Filter #68 - MildStrobe	132
Filter #69 - ReduceBy50	133
Filter #70 - ColorPositionAverageXor	133
Filter #71 - ColorPositionXor	134
Filter #72 - GrayStrobe	134
Filter #73 - ColorStrobeXor	135
Filter #74 - ColorGhost	135
Filter #75 - BlurredOutXor	136
Filter #76 - DizzyFilter	136
Filter #77 - Buzzed	137
Filter #78 - BuzzedDark	137
Filter #79 - AllRed	138
Filter #80 - AllGreen	138
Filter #81 - AllBlue	138
Filter #82 - NegativeStrobe	139
Filter #83 - XorAddMul	139
Filter #84 - HorizontalLines	139
Filter #85 - StrobeRedGreenBlue	140
Filter #86 - Pulse	140
Filter #87 - DiamondPattern	141
Filter #88 - Bitwise_XOR	141
Filter #89 - Bitwise_AND	142
Filter #90 - Bitwise_OR	142
Filter #91 - BlendSwitch	142

Filter #92 - LineRGB	143
Filter #93 - PixelRGB	143
Filter #94 - InvertedScanlines	144
Filter #95 - ScanSwitch	144
Filter #96 - ScanAlphaSwitch	145
Filter #97 - RGBFlash	145
Filter #98 - DiagonalLines	146
Filter #99 - Darken	146
Color & Temporal (100-199)	146
Filter #100 - SelfXorBlend	146
Filter #101 - SelfXorDoubleFlash	147
Filter #102 - SelfOrDoubleFlash	147
Filter #103 - BlendRowCurvedSqrt	148
Filter #104 - XorAlpha	148
Filter #105 - RandomXorBlend	148
Filter #106 - AndStrobe	149
Filter #107 - AndStrobeScale	149
Filter #108 - AndPixelStrobe	149
Filter #109 - AndOrXorStrobe	150
Filter #110 - FadeInAndOut	150
Filter #111 - BrightStrobe	151
Filter #112 - DarkStrobe	151
Filter #113 - RandomXorOpposite	151
Filter #114 - GradientRainbow	152
Filter #115 - cossinMultiply	152
Filter #116 - colorAccumulate1	153
Filter #117 - colorAccumulate2	153
Filter #118 - WeakBlend	153
Filter #119 - StrobeEffect	154
Filter #120 - Blend3	154
Filter #121 - NegParadox	155
Filter #122 - ThoughtMode	155
Filter #123 - Tri	156
Filter #124 - Distort	156
Filter #125 - colorAccumulate3	156
Filter #126 - filter8	157
Filter #127 - filter3	157
Filter #128 - rainbowBlend	157
Filter #129 - pixelScale	158
Filter #130 - GradientSelf	158
Filter #131 - GradientSelfVertical	159
Filter #132 - GradientDown	159
Filter #133 - GraidentHorizontal	159
Filter #134 - Inter	160
Filter #135 - BlendedScanLines	160
Filter #136 - GradientStripes	160
Filter #137 - XorSine	161
Filter #138 - Circular	161
Filter #139 - RandomPixels	162
Filter #140 - DarkRandomPixels	162
Filter #141 - Bars	162
Filter #142 - NegativeByRow	163
Filter #143 - XorScale	163
Filter #144 - SelfAlphaRGB	164
Filter #145 - BitwiseXorStrobe	164
Filter #146 - OrStrobe	165
Filter #147 - DivideAndIncH	165

Filter #148 - DivideAndIncW	166
Filter #149 - RandomIncrease	166
Filter #150 - SelfAlphaScaleBlend	167
Filter #151 - FadeBars	167
Filter #152 - StrobeXor	168
Filter #153 - Blank	168
Filter #154 - ColorVariableBlend	168
Filter #155 - ColorXorBlend	169
Filter #156 - ColorAddBlend	169
Filter #157 - SurroundingPixels	170
Filter #158 - SurroundingPixelsAlpha	170
Filter #159 - DarkModBlend	170
Filter #160 - IncreaseDecreaseGamma	171
Filter #161 - BlendChannelXor	171
Filter #162 - IncDifference	172
Filter #163 - IncDifferenceAlpha	172
Filter #164 - MirrorXorAlpha	172
Filter #165 - IntertwinedMirror	173
Filter #166 - ColorFadeFilter	174
Filter #167 - ColorChannelMoveUpAndDown	174
Filter #168 - MedianStrobe	175
Filter #169 - RGBBlend	175
Filter #170 - BGRBlend	176
Filter #171 - FlipAlphaBlend	176
Filter #172 - RandomFlipFilter	177
Filter #173 - SelfScaleByFrame	177
Filter #174 - AlphaBlendMirror	177
Filter #175 - TwistedVision	178
Filter #176 - TruncateColor	178
Filter #177 - TruncateVariable	179
Filter #178 - TruncateVariableScale	179
Filter #179 - XorFade	179
Filter #180 - SineValue	180
Filter #181 - FadeRtoGtoB	180
Filter #182 - FadeRandomChannel	181
Filter #183 - VariableLines	181
Filter #184 - VariableLinesVertical	181
Filter #185 - RowMedianBlend	182
Filter #186 - MirrorReverseColorBlend	182
Filter #187 - PsychoticVision	183
Filter #188 - PixelGlitch	183
Filter #189 - StaticGlitch	183
Filter #190 - WavePattern	184
Filter #191 - WavePatternXor	184
Filter #192 - DiagonalXor	185
Filter #193 - RGBShiftBlend	185
Filter #194 - ChannelShuffle	186
Filter #195 - ChannelShuffleRand	186
Filter #196 - PixelCounter	186
Filter #197 - PixelCounterXor	187
Filter #198 - RowColorBlend	187
Filter #199 - ColumnColorBlend	187
Spatial & Pattern (200-299)	188
Filter #200 - CheckerboardXor	188
Filter #201 - CheckerboardBlend	188
Filter #202 - SineWaveDistort	189
Filter #203 - CosineWaveDistort	189

Filter #204 - SinCosBlend	189
Filter #205 - PixelReverseXor	190
Filter #206 - LinesAcrossX	190
Filter #207 - XorLineX	191
Filter #208 - AlphaComponentIncrease	191
Filter #209 - ExpandContract	191
Filter #210 - LongLines	192
Filter #211 - TearRight	192
Filter #212 - TearDown	193
Filter #213 - DistortionByRow	193
Filter #214 - DistortionByCol	194
Filter #215 - AlternateAlpha	194
Filter #216 - DiagSquareRGB	194
Filter #217 - ShiftPixelsRGB	195
Filter #218 - ColorWaveTrailsRGB	195
Filter #219 - ProperTrails	196
Filter #220 - XorLag	196
Filter #221 - PixelateBlend	196
Filter #222 - DiagPixel	197
Filter #223 - DiagPixelY	197
Filter #224 - ExpandLeftRight	198
Filter #225 - DiagSquare	198
Filter #226 - HorizontalColorOffset	198
Filter #227 - PrevFrameNotEqual	199
Filter #228 - BlackLines	199
Filter #229 - DizzyMode	200
Filter #230 - GhostShift	200
Filter #231 - RGBSplitFilter	201
Filter #232 - PixelateRect	201
Filter #233 - CollectionXor4	201
Filter #234 - RectangleSpin	202
Filter #235 - RectanglePlotXY	202
Filter #236 - ShiftLinesDown	203
Filter #237 - PictureStretch	203
Filter #238 - PictureStretchPieces	204
Filter #239 - VisualSnow	204
Filter #240 - VisualSnowX2	205
Filter #241 - LineGlitch	205
Filter #242 - SlitReverse64	205
Filter #243 - SlitReverse64_Increase	206
Filter #244 - SlitStretch	206
Filter #245 - LineLeftRight	207
Filter #246 - LineLeftRightResize	207
Filter #247 - RGBLineTrails	207
Filter #248 - RGBCollectionBlend	208
Filter #249 - RGBCollectionIncrease	208
Filter #250 - RGBLongTrails	209
Filter #251 - FadeRGB_Speed	209
Filter #252 - RGBStrobeTrails	209
Filter #253 - BoxGlitch	210
Filter #254 - VerticalPictureDistort	210
Filter #255 - ShortTrail	211
Filter #256 - DiagInward	211
Filter #257 - DiagSquareInward	211
Filter #258 - DiagSquareInwardResize	212
Filter #259 - PictureShiftDownRight	212
Filter #260 - FlipPictureShift	213

Filter #261 - RGBWideTrails	213
Filter #262 - LineInLineOut_Increase	214
Filter #263 - LineInLineOut2_Increase	214
Filter #264 - LineInLineOut3_Increase	214
Filter #265 - SquareByRow2Plus	215
Filter #266 - FrameSep	215
Filter #267 - FrameSep2	216
Filter #268 - FrameStopStart	216
Filter #269 - OutOfOrder	217
Filter #270 - TrackingDown	217
Filter #271 - TrackingDownBlend	217
Filter #272 - TrackingRev	218
Filter #273 - TrackingMirror	218
Filter #274 - BlockPixels	219
Filter #275 - FrameChop	219
Filter #276 - YLineDown	219
Filter #277 - YLineDownBlend	220
Filter #278 - SquareDiff1	220
Filter #279 - LineAcrossX	221
Filter #280 - ColorGlitch	221
Filter #281 - PixelShiftUp	221
Filter #282 - PixelShiftDown	222
Filter #283 - PixelShiftLeft	222
Filter #284 - PixelShiftRight	223
Filter #285 - PixelShiftDiagonal	223
Filter #286 - WaveBlend	223
Filter #287 - WaveBlendX2	224
Filter #288 - SineWaveBlend	224
Filter #289 - CosineWaveBlend	224
Filter #290 - SpiralWave	225
Filter #291 - RadialBlur	225
Filter #292 - ZoomBlur	226
Filter #293 - RotateBlend	226
Filter #294 - MirrorWave	227
Filter #295 - MirrorWaveX	227
Filter #296 - MirrorWaveY	227
Filter #297 - PixelDrift	228
Filter #298 - PixelDriftX	228
Filter #299 - PixelDriftY	229
Advanced Blend (300-399)	229
Filter #300 - ColorPulse	229
Filter #301 - ColorPulseRGB	229
Filter #302 - ColorPulseXor	230
Filter #303 - GlitchBlock	230
Filter #304 - GlitchBlockXor	231
Filter #305 - GlitchLine	231
Filter #306 - GlitchLineX	231
Filter #307 - NoiseBlend	232
Filter #308 - NoiseBlendX2	232
Filter #309 - NoiseXor	233
Filter #310 - ChannelShift	233
Filter #311 - ChannelShiftX	233
Filter #312 - ChannelRotate	234
Filter #313 - DiagonalStretch	234
Filter #314 - DiagonalStretchX	235
Filter #315 - DiagonalMirror	235
Filter #316 - SquareWave	235

Filter #317 - SquareWaveX	236
Filter #318 - SquareWaveBlend	236
Filter #319 - TriangleWave	237
Filter #320 - TriangleWaveBlend	237
Filter #321 - SawtoothWave	237
Filter #322 - SawtoothWaveBlend	238
Filter #323 - PulseWave	238
Filter #324 - PulseWaveBlend	239
Filter #325 - StepWave	239
Filter #326 - StepWaveBlend	239
Filter #327 - RippleEffect	240
Filter #328 - RippleEffectX2	240
Filter #329 - ShockWave	241
Filter #330 - ShockWaveBlend	242
Filter #331 - TwistEffect	242
Filter #332 - TwistEffectBlend	243
Filter #333 - FishEye	243
Filter #334 - FishEyeBlend	244
Filter #335 - Kaleidoscope	244
Filter #336 - KaleidoscopeBlend	245
Filter #337 - TunnelEffect	245
Filter #338 - TunnelEffectBlend	246
Filter #339 - VortexEffect	246
Filter #340 - VortexEffectBlend	247
Filter #341 - ColorDrift	248
Filter #342 - ColorDriftX	248
Filter #343 - RGBShift	248
Filter #344 - RGBShiftX	249
Filter #345 - ChromaticAberration	249
Filter #346 - ChromaticAberrationX	249
Filter #347 - Posterize	250
Filter #348 - PosterizeBlend	250
Filter #349 - Solarize	251
Filter #350 - SolarizeBlend	251
Filter #351 - GammaBright	251
Filter #352 - GammaDark	252
Filter #353 - ContrastBoost	252
Filter #354 - ContrastReduce	252
Filter #355 - EdgeGlowBlend	253
Filter #356 - FrameBlendMulti	253
Filter #357 - FrameBlendMultiX	254
Filter #358 - AcidTrailsBlend	254
Filter #359 - AcidGlitchX	255
Filter #360 - AlphaXorBlend	255
Filter #361 - AlphaXorBlendDouble	256
Filter #362 - AndOrXorStrobeScale	256
Filter #363 - AveragePixelsXorBlend	256
Filter #364 - BitwiseRotateBlend	257
Filter #365 - BitwiseRotateDiffBlend	257
Filter #366 - BitwiseXorScaleBlend	258
Filter #367 - BlackAndWhiteStrobe	258
Filter #368 - BlendAlphaXorScale	258
Filter #369 - BlendBurredXor	259
Filter #370 - BlendCombinedXor	259
Filter #371 - BlendIncreaseRGB	260
Filter #372 - BlendThreeXor	260
Filter #373 - BlurDistortionBlend	260

Filter #374 - ColorAccumulate	261
Filter #375 - ColorAccumulateBlend	261
Filter #376 - ColorAccumulateXor	261
Filter #377 - ColorChannelBlend	262
Filter #378 - ColorChannelXor	262
Filter #379 - ColorCollectionEnergy	262
Filter #380 - ColorCollectionWave	263
Filter #381 - ColorFadeXor	263
Filter #382 - ColorIntensityBlend	264
Filter #383 - ColorIntensityXor	264
Filter #384 - ColorMoveBlend	264
Filter #385 - ColorPixelBlend	265
Filter #386 - ColorPixelXor	265
Filter #387 - ColorScaleBlend	265
Filter #388 - ColorWaveXor	266
Filter #389 - CosineMultiplyBlend	266
Filter #390 - DarkModBlendXor	267
Filter #391 - DifferenceBlend	267
Filter #392 - DifferenceXorBlend	267
Filter #393 - DistortBlend	268
Filter #394 - DiamondPatternBlend	268
Filter #395 - FadeBlendXor	268
Filter #396 - FlashBlendXor	269
Filter #397 - GhostTrailsBlend	269
Filter #398 - AddInvert	270
Filter #399 - AlphaBlendSimple	270
Extended Effects (400-499)	271
Filter #400 - AlphaBlendDoubleX	271
Filter #401 - AlphaStrobeBlendX	271
Filter #402 - BitwiseAndBlend	271
Filter #403 - BitwiseXorAverage	272
Filter #404 - BitwiseXorBlendX	272
Filter #405 - BlackStrobe	272
Filter #406 - BlendAlphaXorX	273
Filter #407 - BlendCombinedValuesX	273
Filter #408 - BlendFor360	274
Filter #409 - BlendForward16	274
Filter #410 - BlendForward32	274
Filter #411 - BlendFromXtoY	275
Filter #412 - BlendIncreaseX	275
Filter #413 - BlendRedGreenBlue	275
Filter #414 - BlendWithColorX	276
Filter #415 - BlendAngle	276
Filter #416 - BlockScale	276
Filter #417 - BlockStrobe	277
Filter #418 - BlockXor	277
Filter #419 - BlockyTrails16	278
Filter #420 - BlockyTrails32	278
Filter #421 - BlurDistortionX	278
Filter #422 - CannyStrobe	279
Filter #423 - ColorFadeSlow	279
Filter #424 - ColorFibonacci	280
Filter #425 - CurtainEffect	280
Filter #426 - DarkColorFibonacci	280
Filter #427 - DarkColorsBlend	281
Filter #428 - EnergizeBlend	281
Filter #429 - AverageLines	282

Filter #430 - AverageLinesBlendX	282
Filter #431 - BlendRowAlpha	282
Filter #432 - BlendInOut	283
Filter #433 - ColorFlashIncreaseX	283
Filter #434 - ColorIncreaseInOut	284
Filter #435 - ColorLinesX	284
Filter #436 - ColorMoveDownX	284
Filter #437 - ColorOrderSwapX	285
Filter #438 - ColorPulseAlphaX	285
Filter #439 - ColorRowShiftX	286
Filter #440 - ColorShiftXorX	286
Filter #441 - CopyXorAlphaX	287
Filter #442 - CycleShiftRGBX	287
Filter #443 - DarkNegateX	287
Filter #444 - DarkSelfAlphaX	288
Filter #445 - DiagonalGlitch	288
Filter #446 - DigitalHaze	288
Filter #447 - DoubleXorBlend	289
Filter #448 - EchoBlend	289
Filter #449 - ElectricEdge	290
Filter #450 - FlashColorStrobe	290
Filter #451 - FrameDiffXor	291
Filter #452 - GhostMirror	291
Filter #453 - GlitchSort	291
Filter #454 - HeatWave	292
Filter #455 - InterlaceBlend	292
Filter #456 - InvertStrobe	293
Filter #457 - KaleidoBlend	293
Filter #458 - LightStrobe	293
Filter #459 - LineGlitchX	294
Filter #460 - MosaicBlend	294
Filter #461 - NegatePulse	295
Filter #462 - OffsetGhost	295
Filter #463 - PixelateWave	295
Filter #464 - QuantizeBlend	296
Filter #465 - RandomLines	296
Filter #466 - RippleDisplace	297
Filter #467 - RotateShift	297
Filter #468 - SaturationGlow	297
Filter #469 - ScaleToCenter	298
Filter #470 - ShadowMirror	298
Filter #471 - ShiftChannels	299
Filter #472 - SliceGlitch	299
Filter #473 - SobelGlow	299
Filter #474 - SpectralShift	300
Filter #475 - SpiralTrail	300
Filter #476 - SquareTrails	301
Filter #477 - StrobeNegate	301
Filter #478 - ThermalBlend	302
Filter #479 - TintShift	302
Filter #480 - TrailEcho	302
Filter #481 - TransitionBlend	303
Filter #482 - TwistWarp	303
Filter #483 - VerticalShift	304
Filter #484 - VortexBlend	304
Filter #485 - WeavePattern	305
Filter #486 - WhiteBurst	305

Filter #487 - WiggleDisplace	305
Filter #488 - XorPulseX	306
Filter #489 - YellowShift	306
Filter #490 - ZigzagGlitch	307
Filter #491 - AlphaModulate	307
Filter #492 - BlockSwap	307
Filter #493 - ColorResonance	308
Filter #494 - DepthGlitch	308
Filter #495 - EchoShift	309
Filter #496 - FractalNoise	309
Filter #497 - GradientRotate	309
Filter #498 - HarmonicShift	310
Filter #499 - AcidWarp	310
Visual FX A (500-599)	311
Filter #500 - BlendDiagonal	311
Filter #501 - ChromaFlash	311
Filter #502 - CircleWave	311
Filter #503 - ColorCrush	312
Filter #504 - CrosshatchBlend	312
Filter #505 - CyberGlitch	312
Filter #506 - DarkPulse	313
Filter #507 - DiamondPatternX	313
Filter #508 - DigitalRain	314
Filter #509 - DisplaceX	314
Filter #510 - DriftBlend	314
Filter #511 - EdgePulse	315
Filter #512 - FlameEffect	315
Filter #513 - FlickerShift	316
Filter #514 - GhostLayer	316
Filter #515 - GlitchBlockX	316
Filter #516 - GlowPulse	317
Filter #517 - GridDistort	317
Filter #518 - HexPattern	318
Filter #519 - HueRotate	318
Filter #520 - InterweaveX	318
Filter #521 - JitterBlend	319
Filter #522 - KaleidoScope4	319
Filter #523 - LaserScan	320
Filter #524 - LightLeak	320
Filter #525 - MeltDown	320
Filter #526 - MirrorDiag	321
Filter #527 - NeonGlow	321
Filter #528 - NoiseBlendX	322
Filter #529 - PixelDrift	322
Filter #530 - PlasmaWave	322
Filter #531 - PrismSplit	323
Filter #532 - PulseRadial	323
Filter #533 - RainbowStrobe	324
Filter #534 - RefractionX	324
Filter #535 - ScanlineX	324
Filter #536 - ShatterEffect	325
Filter #537 - StaticNoise	325
Filter #538 - TunnelVision	326
Filter #539 - AberrationPulse	326
Filter #540 - AquaWave	326
Filter #541 - BinaryFlash	327
Filter #542 - BloomGlow	327

Filter #543 - CellularNoise	328
Filter #544 - ChromaShift2	328
Filter #545 - ColorBands	328
Filter #546 - ColorVortex	329
Filter #547 - CrystalMosaic	329
Filter #548 - CubicDistort	330
Filter #549 - DepthFade	330
Filter #550 - DiscoFlash	330
Filter #551 - DitherBlend	331
Filter #552 - DoubleVision	331
Filter #553 - DreamHaze	332
Filter #554 - ElectricStorm	332
Filter #555 - EmbossShift	332
Filter #556 - FiberOptic	333
Filter #557 - FilmGrain	333
Filter #558 - FireWorks	333
Filter #559 - FluidMotion	334
Filter #560 - FogRoll	334
Filter #561 - GlassRefract	335
Filter #562 - GlowTrails	335
Filter #563 - GridPulse	336
Filter #564 - HalftoneBlend	336
Filter #565 - HeatDistort	336
Filter #566 - HoloGlitch	337
Filter #567 - InfraredView	337
Filter #568 - LavaLamp	338
Filter #569 - LensFlare	338
Filter #570 - LightningBolt	339
Filter #571 - LiquidMetal	339
Filter #572 - MatrixCode	339
Filter #573 - MirrorKaleid	340
Filter #574 - NightVision	340
Filter #575 - OilSlick	341
Filter #576 - ParticleField	341
Filter #577 - PinwheelSpin	341
Filter #578 - PixelStorm	342
Filter #579 - PlaidPattern	342
Filter #580 - PolarInvert	343
Filter #581 - PolychromeTint	343
Filter #582 - PopArtDots	344
Filter #583 - PrismaticEdge	344
Filter #584 - PulseWarp	345
Filter #585 - QuantumNoise	345
Filter #586 - QuiltBlend	345
Filter #587 - RadarSweep	346
Filter #588 - RaindropRipple	346
Filter #589 - RasterBars	347
Filter #590 - RetroTube	347
Filter #591 - RingWave	348
Filter #592 - RippleTank	348
Filter #593 - RotatingPrism	348
Filter #594 - SandStorm	349
Filter #595 - SaturationPulse	349
Filter #596 - ScatterPixel	350
Filter #597 - ShadowPlay	350
Filter #598 - ShimmerGlass	350
Filter #599 - SilhouetteBlend	351

Visual FX B (600-699)	351
Filter #600 - SketchOutline	351
Filter #601 - SliceShift	352
Filter #602 - SmearMotion	352
Filter #603 - SmokeWisp	353
Filter #604 - SnowDrift	353
Filter #605 - SolarFlare	354
Filter #606 - SparkShower	354
Filter #607 - SpectrumWave	354
Filter #608 - SpiralZoom	355
Filter #609 - SplitMirror	355
Filter #610 - StarBurst	356
Filter #611 - StaticPulse	356
Filter #612 - StencilCut	357
Filter #613 - StippleShade	357
Filter #614 - StormCloud	357
Filter #615 - StreakBlur	358
Filter #616 - StrobeEdge	358
Filter #617 - SubpixelShift	359
Filter #618 - SwimDistort	359
Filter #619 - TangentWarp	360
Filter #620 - TapeGlitch	360
Filter #621 - TechnoGrid	360
Filter #622 - TeleportPixel	361
Filter #623 - TemporalBlur	361
Filter #624 - TerraFracture	362
Filter #625 - TextureWave	362
Filter #626 - ThresholdPulse	362
Filter #627 - TidalWave	363
Filter #628 - TintCycle	363
Filter #629 - TraceEdge	364
Filter #630 - TriangleMosaic	364
Filter #631 - TripleSplit	365
Filter #632 - TurbulentFlow	365
Filter #633 - UnderwaterCaustic	366
Filter #634 - UnsharpPulse	366
Filter #635 - VaporTrail	366
Filter #636 - VectorField	367
Filter #637 - VelocityBlur	367
Filter #638 - VerticalMelt	368
Filter #639 - VHSTracking	368
Filter #640 - VibrantPop	369
Filter #641 - VignetteFlash	369
Filter #642 - VoronoiShatter	370
Filter #643 - WarpSpeed	370
Filter #644 - WaterColor	370
Filter #645 - WaveCollapse	371
Filter #646 - WebPattern	371
Filter #647 - WhirlpoolSpin	372
Filter #648 - WindBlast	372
Filter #649 - WireframePulse	373
Filter #650 - XRayFlash	373
Filter #651 - ZebraStripe	374
Filter #652 - ZenRipple	374
Filter #653 - ZigzagWave	374
Filter #654 - ZoomPulse	375
Filter #655 - ZoneTint	375

Filter #656 - AcidDrip	376
Filter #657 - AuroraWave	376
Filter #658 - BandPass	376
Filter #659 - BilinearStretch	377
Filter #660 - BleedThrough	377
Filter #661 - BlockShatter	378
Filter #662 - BlurMask	378
Filter #663 - BokehBlur	379
Filter #664 - BounceWave	379
Filter #665 - BrokenGlass	380
Filter #666 - BubbleWarp	380
Filter #667 - CRTCurvature	381
Filter #668 - CascadeBlend	381
Filter #669 - CelShade	381
Filter #670 - ChainReaction	382
Filter #671 - ChannelDelay	382
Filter #672 - ChromaBleed	383
Filter #673 - CircuitTrace	383
Filter #674 - ClockWipe	384
Filter #675 - CloudShadow	384
Filter #676 - ColorBurn	384
Filter #677 - ColorHalves	385
Filter #678 - ComicDots	385
Filter #679 - ConcentricPulse	386
Filter #680 - CopperTone	386
Filter #681 - CornerStretch	386
Filter #682 - CosmicDust	387
Filter #683 - CrossBlur	387
Filter #684 - CrossProcess	388
Filter #685 - CrystalEdge	388
Filter #686 - CubeRotate	389
Filter #687 - CurtainReveal	389
Filter #688 - CyberPunk	389
Filter #689 - DataCorrupt	390
Filter #690 - DebrisField	390
Filter #691 - DeepFry	391
Filter #692 - DesyncRGB	391
Filter #693 - DiagonalWipe	392
Filter #694 - DigitalArtifact	392
Filter #695 - DimensionRift	392
Filter #696 - DotCrawl	393
Filter #697 - DualTone	393
Filter #698 - EchoFade	394
Filter #699 - EdgeMelt	394
Visual FX C (700-735)	395
Filter #700 - ElasticWarp	395
Filter #701 - EmberGlow	395
Filter #702 - EntropyShift	396
Filter #703 - ErosionBlend	396
Filter #704 - ExplosionBurst	397
Filter #705 - FacetMirror	397
Filter #706 - FadeStreak	398
Filter #707 - FeatherEdge	398
Filter #708 - FlashFreeze	398
Filter #709 - FlipMirror	399
Filter #710 - FloatDrift	399
Filter #711 - FlowField	400

Filter #712 - FoldWarp	400
Filter #713 - FragmentScatter	400
Filter #714 - FrequencyPulse	401
Filter #715 - FrostBite	401
Filter #716 - FuseBlend	402
Filter #717 - GalaxySpiral	402
Filter #718 - GelWobble	403
Filter #719 - GhostEcho	403
Filter #720 - GlassShatter	403
Filter #721 - GlimmerPulse	404
Filter #722 - GlitchMosaic	404
Filter #723 - GlowEdge	405
Filter #724 - GradientMelt	405
Filter #725 - GrainStorm	406
Filter #726 - GravityPull	406
Filter #727 - GridWarp	406
Filter #728 - HazeLayer	407
Filter #729 - HeatRipple	407
Filter #730 - HexagonBlur	408
Filter #731 - HighContrast	408
Filter #732 - HologramScan	408
Filter #733 - HorizonBend	409
Filter #734 - HotSpot	409
Filter #735 - HueWobble	410
Glitch Series (736-857)	410
Filter #736 - glitch_alpha_diamond1	410
Filter #737 - glitch_alpha_diamond2	411
Filter #738 - glitch_alpha_diamond3	411
Filter #739 - glitch_alpha_diamond4	411
Filter #740 - glitch_alpha_diamond5	412
Filter #741 - glitch_alphatrails	412
Filter #742 - glitch_color_fade_ex1	413
Filter #743 - glitch_color_fade_ex2	413
Filter #744 - glitch_color_shift	414
Filter #745 - glitch_colorxor01	414
Filter #746 - glitch_colorxor02	415
Filter #747 - glitch_colorxor03	415
Filter #748 - glitch_colorxor04	415
Filter #749 - glitch_colorxor05	416
Filter #750 - glitch_colorxor06	416
Filter #751 - glitch_colorxor07	416
Filter #752 - glitch_colorxor08	417
Filter #753 - glitch_colorxor09	417
Filter #754 - glitch_colorxor10	418
Filter #755 - glitch_colorxor11	418
Filter #756 - glitch_colorxor12	418
Filter #757 - glitch_colorxor13	419
Filter #758 - glitch_colorxor14	419
Filter #759 - glitch_colorxor15	419
Filter #760 - glitch_colorxor16	420
Filter #761 - glitch_colorxor17	420
Filter #762 - glitch_colorxor18	420
Filter #763 - glitch_colorxor19	421
Filter #764 - glitch_colorxor20	421
Filter #765 - glitch_distort_picture	422
Filter #766 - glitch_echo	422
Filter #767 - glitch_fast_monocolortrails	423

Filter #768 - glitch_frame_pixels	423
Filter #769 - glitch_frame_resize_stretch	423
Filter #770 - glitch_frame_skip	424
Filter #771 - glitch_frame_skip_resize	424
Filter #772 - glitch_frame_skip_shadow	425
Filter #773 - glitch_frame_skip_shadow_x2	425
Filter #774 - glitch_frame_slide01	426
Filter #775 - glitch_frame_slide02	426
Filter #776 - glitch_frame_slide03	427
Filter #777 - glitch_frame_slide04	427
Filter #778 - glitch_frame_square_col	427
Filter #779 - glitch_frame_stutter_filter	428
Filter #780 - glitch_glitch_x1	428
Filter #781 - glitch_gpt1	429
Filter #782 - glitch_gpt2	429
Filter #783 - glitch_gpt3	430
Filter #784 - glitch_invertflash	430
Filter #785 - glitch_line_across_side01	431
Filter #786 - glitch_line_across_side02	431
Filter #787 - glitch_line_across_side03	432
Filter #788 - glitch_line_across_side04	432
Filter #789 - glitch_line_across_side05	433
Filter #790 - glitch_line_across_side06	433
Filter #791 - glitch_line_collection01	434
Filter #792 - glitch_line_collection02	434
Filter #793 - glitch_line_collection03	434
Filter #794 - glitch_line_collection04	435
Filter #795 - glitch_line_collection05	435
Filter #796 - glitch_line_collection06	436
Filter #797 - glitch_line_collection07	436
Filter #798 - glitch_line_offset_inout	437
Filter #799 - glitch_line_offset_inout2	437
Filter #800 - glitch_line_offset_inout3	437
Filter #801 - glitch_newrandblend	438
Filter #802 - glitch_newstretchlines	438
Filter #803 - glitch_newvarlines	439
Filter #804 - glitch_outoforder	439
Filter #805 - glitch_pic_adjust1	440
Filter #806 - glitch_pic_adjust2	440
Filter #807 - glitch_picture_jump01	440
Filter #808 - glitch_picture_jump02	441
Filter #809 - glitch_picture_jump03	441
Filter #810 - glitch_picture_jump04	442
Filter #811 - glitch_picture_jump05	442
Filter #812 - glitch_picture_jump06	442
Filter #813 - glitch_picture_jump07	443
Filter #814 - glitch_picture_jump08	443
Filter #815 - glitch_picture_jump09	444
Filter #816 - glitch_picture_jump10	444
Filter #817 - glitch_picture_jump11	445
Filter #818 - glitch_picture_jump12	445
Filter #819 - glitch_picture_jump13	446
Filter #820 - glitch_picture_jump14	446
Filter #821 - glitch_picture_jump15	446
Filter #822 - glitch_picture_jump16	447
Filter #823 - glitch_picture_jump17	447
Filter #824 - glitch_picture_jump18	448

Filter #825 - glitch_picture_jump19	448
Filter #826 - glitch_picture_jump20	449
Filter #827 - glitch_picture_jump21	449
Filter #828 - glitch_picture_jump22	449
Filter #829 - glitch_picture_jump23	450
Filter #830 - glitch_picture_jump24	450
Filter #831 - glitch_picture_jump25	451
Filter #832 - glitch_picture_jump26	451
Filter #833 - glitch_picture_jump27	452
Filter #834 - glitch_picture_jump28	452
Filter #835 - glitch_picture_jump29	453
Filter #836 - glitch_picture_jump30	453
Filter #837 - glitch_picture_jump31	453
Filter #838 - glitch_rect_size	454
Filter #839 - glitch_rsquare2	454
Filter #840 - glitch_rsquare3	455
Filter #841 - glitch_rsquare4	455
Filter #842 - glitch_rsquare5	456
Filter #843 - glitch_slice_frame	456
Filter #844 - glitch_slice_frame_w	457
Filter #845 - glitch_square_block_v2a	457
Filter #846 - glitch_square_block_v2b	458
Filter #847 - glitch_square_block_v2c	458
Filter #848 - glitch_square_block_v2d	459
Filter #849 - glitch_square_block_v2e	459
Filter #850 - glitch_square_block_v2f	460
Filter #851 - glitch_square_block_v2g	460
Filter #852 - glitch_square_block_v2h	461
Filter #853 - glitch_square_xor	461
Filter #854 - glitch_store_frame10	462
Filter #855 - glitch_stuckframe	462
Filter #856 - glitch_stutter_long	462
Filter #857 - glitch_stutter_sbrv	463
AC Glitch Library (858-904)	464
Filter #858 - acgl_glitch_AlphaBlendFive	464
Filter #859 - acgl_glitch_AlphaBlendTri	464
Filter #860 - acgl_glitch_AlphaBlendExpand	465
Filter #861 - acgl_glitch_BarsCol	465
Filter #862 - acgl_glitch_BarsColAlpha	466
Filter #863 - acgl_glitch_BarsHoriz	466
Filter #864 - acgl_glitch_BarsHorizAlpha	466
Filter #865 - acgl_glitch_BlackSquare	467
Filter #866 - acgl_glitch_BlendStuck	467
Filter #867 - acgl_glitch_ColorDistort	468
Filter #868 - acgl_glitch_ColorRect	468
Filter #869 - acgl_glitch_ColorOnOff	469
Filter #870 - acgl_glitch_DEM	469
Filter #871 - acgl_glitch_FrameMirror	470
Filter #872 - acgl_glitch_FramePix	470
Filter #873 - acgl_glitch_FrameReverse	471
Filter #874 - acgl_glitch_FrameReverse2	471
Filter #875 - acgl_glitch_FrameReverseNoBlend	472
Filter #876 - acgl_glitch_FrameSepBand	472
Filter #877 - acgl_glitch_FrameSwap	472
Filter #878 - acgl_glitch_FrameXBlend	473
Filter #879 - acgl_glitch_FrameXBlendXor	473
Filter #880 - acgl_glitch_FrameYBlend	474

Filter #881 - acgl_glitch_FrameYBlendXor	474
Filter #882 - acgl_glitch_AddMulXor	474
Filter #883 - acgl_glitch_AddXor	475
Filter #884 - acgl_glitch_ColorBarsRand	475
Filter #885 - acgl_glitch_ColorBarsX	476
Filter #886 - acgl_glitch_ColorBarsY	477
Filter #887 - acgl_glitch_ColorShiftY	477
Filter #888 - acgl_glitch_LineCollectionRGB	478
Filter #889 - acgl_glitch_FrameX2	478
Filter #890 - acgl_glitch_NewBars	479
Filter #891 - acgl_glitch_NewBars2	479
Filter #892 - acgl_glitch_NewBlendLines	480
Filter #893 - acgl_glitch_NewLines	480
Filter #894 - acgl_glitch_NewOne	481
Filter #895 - acgl_glitch_StrobeCycle	481
Filter #896 - acgl_glitch_StuckFrame2	482
Filter #897 - acgl_glitch_StuckLine	482
Filter #898 - acgl_glitch_StuckRow	483
Filter #899 - acgl_glitch_StuckRowLine	483
Filter #900 - acgl_glitch_SepBlocks	483
Filter #901 - acgl_glitch_Plug1	484
Filter #902 - acgl_glitch_OppositeDir	484
Filter #903 - acgl_glitch_OffStuck	485
Filter #904 - acgl_glitch_NewVarBlendLines	485

Part II - GLSL Shaders 486

Vertex Shader	486

Other Effects (225 shaders)	486

Shader #13 - addup cos	486
Shader #26 - alpha diamond	487
Shader #27 - alpha sin os	488
Shader #32 - apart	488
Shader #42 - aura3	489
Shader #43 - aura4	489
Shader #53 - balloon	490
Shader #60 - bi cycle	491
Shader #63 - blackhole	491
Shader #64 - blank	492
Shader #67 - blue	493
Shader #71 - blue strobe	494
Shader #76 - bowl by time	494
Shader #78 - breathe	495
Shader #79 - bright	496
Shader #81 - brighten	497
Shader #82 - brighten rev	497
Shader #93 - cane	498
Shader #94 - cd	498
Shader #98 - cmod	499
Shader #139 - comb3	500
Shader #144 - comp quality	501
Shader #155 - cos	502
Shader #156 - cos sin	502
Shader #161 - crt	503
Shader #162 - crystal 2	503
Shader #163 - crystal 3	504
Shader #164 - crystal 4	505
Shader #165 - crystal	506

Shader #166 - crystalball	506
Shader #176 - cyclone	507
Shader #181 - dark	507
Shader #183 - dark psych	508
Shader #189 - dark square	509
Shader #190 - darkb	510
Shader #191 - darkg	510
Shader #192 - darkr	511
Shader #193 - darkred	511
Shader #209 - delic	512
Shader #211 - diagsquare	513
Shader #212 - diagsquarecircle	513
Shader #217 - disk	514
Shader #232 - dmdc	514
Shader #233 - drain	515
Shader #241 - drain reset	516
Shader #324 - empty1	516
Shader #333 - fat blue	517
Shader #334 - fat green	517
Shader #335 - fat red	518
Shader #337 - fat slow	518
Shader #338 - fat	518
Shader #345 - focus	519
Shader #354 - frac mir	519
Shader #458 - g ufo	520
Shader #461 - gaura	521
Shader #473 - gem pong	522
Shader #478 - gem frac	523
Shader #489 - genergy	523
Shader #490 - geo pi	524
Shader #491 - geometric	524
Shader #492 - geometric2	525
Shader #493 - geometric3	526
Shader #494 - geometric4	526
Shader #495 - geometric5	527
Shader #496 - gfs	527
Shader #498 - ggrad	528
Shader #499 - gkale	528
Shader #526 - gmir2	529
Shader #545 - green	530
Shader #546 - green aura	531
Shader #550 - green move	532
Shader #551 - green strobe	532
Shader #552 - greenblue	533
Shader #564 - gvortex	534
Shader #572 - heartthrob	534
Shader #573 - heartthrob2	535
Shader #575 - heat	536
Shader #590 - index purple	536
Shader #592 - inflate	537
Shader #595 - kale	537
Shader #596 - kale2	538
Shader #597 - kale3	538
Shader #598 - kale4	539
Shader #609 - lineinlineout	540
Shader #623 - magic 2	540
Shader #624 - magic	541

Shader #626 - matrix	541
Shader #632 - mb	541
Shader #633 - minimize	542
Shader #634 - mirdmd	543
Shader #635 - mirdmd1	543
Shader #636 - mirdmdr	544
Shader #640 - mirdmdt	544
Shader #716 - mix diag	545
Shader #717 - move	545
Shader #718 - moveGL	546
Shader #720 - negative	546
Shader #728 - nofilter	546
Shader #729 - nothing	547
Shader #738 - old film	547
Shader #739 - onoffsep	548
Shader #740 - open shadow	548
Shader #741 - open shadow jump	549
Shader #742 - open shadow multi	549
Shader #743 - open shadow multi x2	550
Shader #744 - open up	550
Shader #745 - optxtime	551
Shader #746 - optxtime cos	552
Shader #747 - optxtime tex	553
Shader #748 - page turn	553
Shader #749 - page turn2	554
Shader #750 - page turn3	554
Shader #751 - particle 1	555
Shader #752 - pastel	556
Shader #753 - pebble	556
Shader #754 - pebble fast	557
Shader #760 - pinch	557
Shader #762 - plasma	558
Shader #763 - plasma2	559
Shader #764 - plasma3	560
Shader #765 - plasma prism	561
Shader #768 - pong ataan ex	561
Shader #769 - pong atan	562
Shader #770 - pong atan2	563
Shader #771 - pong atan3	563
Shader #772 - pong pi.mp4	564
Shader #773 - pong tex	565
Shader #774 - pool	565
Shader #776 - prisim glass triangle	565
Shader #777 - prism	566
Shader #778 - prism glass	567
Shader #780 - prism glass wall	568
Shader #782 - prism quad	569
Shader #783 - prism quad by time	570
Shader #784 - prism quad by time fast	571
Shader #785 - prism quad strong	572
Shader #786 - psych	572
Shader #793 - pull	573
Shader #794 - pull out	573
Shader #796 - purple fade	574
Shader #798 - purple strobe	574
Shader #800 - radial	575
Shader #853 - react12	576

Shader #854 - react13	576
Shader #855 - react14	576
Shader #856 - react15	577
Shader #861 - react2	578
Shader #862 - react3	578
Shader #863 - react4	579
Shader #866 - react7	579
Shader #868 - react9	579
Shader #869 - recep 2	580
Shader #870 - recep	580
Shader #871 - rect1	581
Shader #872 - rect spiral	581
Shader #873 - red	582
Shader #874 - red strobe	583
Shader #876 - reverse	583
Shader #877 - reverse alpha	584
Shader #878 - reverse vert	585
Shader #880 - reverse xy	585
Shader #911 - sbrv	586
Shader #913 - scramble 2	586
Shader #914 - scramble 3	587
Shader #915 - scramble	587
Shader #923 - shake	587
Shader #924 - sheet	588
Shader #925 - shift	588
Shader #926 - shift dark	589
Shader #927 - shrink texture	590
Shader #928 - shrink texture 4	591
Shader #929 - sin	591
Shader #930 - sin delic	592
Shader #933 - sine grad	593
Shader #934 - sine grad2	593
Shader #935 - sine grad3	594
Shader #936 - sine grad4	594
Shader #937 - sine grad5	595
Shader #942 - sine time	596
Shader #943 - sing	596
Shader #944 - single glittch	597
Shader #946 - skinny	598
Shader #947 - slither	598
Shader #948 - slither vert	599
Shader #953 - snake	599
Shader #954 - snake dir	600
Shader #955 - snake updown	600
Shader #956 - snes	601
Shader #957 - source strobe	601
Shader #959 - spiral aura	602
Shader #969 - spiral square	602
Shader #972 - splash x	603
Shader #973 - splash y	603
Shader #974 - splash	604
Shader #978 - star	604
Shader #979 - star5	605
Shader #980 - stardust	606
Shader #981 - starX	606
Shader #982 - starX2	607
Shader #983 - starX3	608

Shader #984 - starX4	608
Shader #985 - strobe	609
Shader #987 - strobe in out	609
Shader #989 - strobe x1	610
Shader #991 - surround	611
Shader #999 - tan	612
Shader #1002 - tear	613
Shader #1003 - tearing	613
Shader #1004 - tearing max	613
Shader #1005 - tearing single	614
Shader #1006 - tearing spiral	614
Shader #1012 - time frac	615
Shader #1016 - tornado	615
Shader #1017 - tremor1	616
Shader #1018 - tremor2	617
Shader #1019 - tremor3	617
Shader #1020 - tremor4	618
Shader #1025 - tv show	618
Shader #1028 - twirl tex	619
Shader #1059 - weirdlines	619
Shader #1060 - whirlx	620
Shader #1064 - wrap around	621
Shader #1066 - xcordstrobe	621
Shader #1089 - yin yang	622
Shader #1090 - zigzag	623
Color Manipulation (118 shaders)	623
Shader #25 - all colors	623
Shader #39 - audiocolor	625
Shader #40 - aura	626
Shader #61 - bit rainbow	627
Shader #80 - bright rainbow	627
Shader #97 - chue	628
Shader #105 - color f diag	629
Shader #106 - color f	629
Shader #107 - color f2	630
Shader #108 - color max	630
Shader #109 - color mod dark	632
Shader #110 - color mod	632
Shader #111 - color o	633
Shader #112 - color effect	633
Shader #114 - color flux	634
Shader #115 - color flux whirl	635
Shader #116 - color g	636
Shader #118 - color grad1	637
Shader #119 - color grad rainbow	637
Shader #120 - color increase	638
Shader #121 - color l	639
Shader #122 - color pool	640
Shader #123 - color swirl o1	641
Shader #124 - color swirl beautiful	641
Shader #125 - color swirl c	642
Shader #126 - color swirl g	643
Shader #128 - color swirl sin	643
Shader #130 - color timeval	644
Shader #131 - color w	645
Shader #132 - color w2	646
Shader #135 - colorsheet	647

Shader #154	- coord color	647
Shader #185	- dark rainbow	648
Shader #186	- dark rainbow2	648
Shader #187	- dark rainbow limit	649
Shader #188	- dark rainbow rubber soul	650
Shader #201	- deepcolor	650
Shader #210	- dhue	651
Shader #240	- drain rainbow	652
Shader #261	- echo alpha three rgb	653
Shader #267	- echo color	654
Shader #268	- echo color1	654
Shader #269	- echo color2	655
Shader #270	- echo color3	656
Shader #271	- echo color4	656
Shader #272	- echo color5	657
Shader #275	- echo div4 rainbow	657
Shader #283	- echo mix colors	658
Shader #284	- echo mix colors rev	659
Shader #287	- echo rainbow spin	659
Shader #288	- echo rainbow swirl	660
Shader #293	- echo rgb	661
Shader #294	- echo rgb1	661
Shader #297	- echo s rgb	662
Shader #325	- es1	663
Shader #336	- fat rgb	664
Shader #466	- gem color spiral	664
Shader #467	- gem color spsiral	665
Shader #470	- gem hue frac	666
Shader #486	- gem rainbow metal	667
Shader #487	- gem rainbow metal audio	668
Shader #502	- gkalei	669
Shader #539	- gradient	670
Shader #540	- gradient color	670
Shader #541	- gradient position	671
Shader #542	- gradient shift	672
Shader #543	- grainbow	673
Shader #544	- grayscale	673
Shader #548	- green gradient	674
Shader #549	- green gradient strobe	675
Shader #577	- huei af2	676
Shader #621	- list rainbow enter	677
Shader #637	- mirdmdrcolor	678
Shader #719	- moving gradient	678
Shader #730	- o1 hue	679
Shader #758	- picolor	681
Shader #766	- plasma rainbow	681
Shader #789	- psych wave rainbow	683
Shader #804	- rainbow touch	684
Shader #805	- rainbow.banding	685
Shader #807	- rainbow bright	685
Shader #808	- rainbow cd	686
Shader #809	- rainbow cd curtain	687
Shader #810	- rainbow cd spin	689
Shader #811	- rainbow cd strobe	690
Shader #812	- rainbow color1	691
Shader #813	- rainbow color2	691
Shader #814	- rainbow color3	692

Shader #815 - rainbow color swirl	692
Shader #816 - rainbow expand	693
Shader #822 - rainbow ink	694
Shader #823 - rainbow ink t	695
Shader #824 - rainbow left	695
Shader #828 - rainbow pi01	696
Shader #829 - rainbow pi neoin	697
Shader #830 - rainbow prisim	697
Shader #831 - rainbow rip prism	698
Shader #832 - rainbow spiral	699
Shader #833 - rainbow spiral loose	701
Shader #834 - rainbow spiral rev	702
Shader #835 - rainbow spiral rev2	704
Shader #836 - rainbow swirl limit	705
Shader #837 - rainbow swirl limit discard	706
Shader #838 - rainbow top	707
Shader #883 - rgb control	707
Shader #886 - RGBColorTrails cache	708
Shader #887 - rgbt	709
Shader #888 - rgbvar inc	709
Shader #889 - rhue	710
Shader #890 - rightintwo	711
Shader #910 - sac geo	712
Shader #920 - sepia	713
Shader #931 - sine1	713
Shader #932 - sine2	714
Shader #938 - sine grad6	714
Shader #939 - sine grad7	715
Shader #977 - srainbow	716
Shader #986 - strobe colors	717
Cursor & Mouse (117 shaders)	718
Shader #22 - air full mouse	718
Shader #23 - air mouse	718
Shader #33 - apart mouse	719
Shader #38 - audio mouse	720
Shader #41 - aura2	720
Shader #56 - bend2mouse	723
Shader #84 - brot zoom mouse	724
Shader #85 - bubble 2	725
Shader #86 - bubble move	725
Shader #87 - bubble zoom mouse	726
Shader #88 - bubble	727
Shader #89 - bubble amp	727
Shader #90 - bubbleGL	728
Shader #91 - BubbleMouse	728
Shader #99 - code flux mouse	729
Shader #100 - code rev3	730
Shader #127 - color swirl o1 mouse	730
Shader #129 - color swirl sin mouse	731
Shader #133 - color w mouse	732
Shader #137 - comb3 frac mouse	733
Shader #138 - comb3 frac mouse2	734
Shader #140 - comb3 geo mouse	735
Shader #141 - comb3 mouse	736
Shader #143 - comp mouse	738
Shader #170 - CursorMove	738
Shader #171 - CursorWarp2	739

Shader #178 - cyclone zoom mouse	740
Shader #214 - diamond pattern 1	741
Shader #239 - drain mouse	742
Shader #245 - dseek	742
Shader #295 - echo rgb mouse	743
Shader #318 - elastic2	744
Shader #319 - elastic mouse	745
Shader #340 - fish pos mouse	746
Shader #342 - fisheye mouse	747
Shader #351 - frac wrap	748
Shader #353 - frac cosine	751
Shader #355 - frac mirror mouse zoom	754
Shader #356 - frac mouse z	756
Shader #357 - frac mouse z2	758
Shader #359 - frac shader01	760
Shader #360 - frac shader01 dark	762
Shader #363 - frac shader02 dmd	766
Shader #364 - frac shader02 dmd2	768
Shader #366 - frac shader02 dmd3	771
Shader #367 - frac shader02 dmd4	774
Shader #368 - frac shader02 dmd5	777
Shader #369 - frac shader02 dmd6i	780
Shader #370 - frac shader02 dmd6i1	783
Shader #376 - frac shader02 dmd6i bubble	786
Shader #378 - frac shader02 dmd6i wrap	789
Shader #380 - frac shader02 dmd7i	792
Shader #381 - frac shader02 dmd8i	795
Shader #382 - frac shader02 dmd9i	798
Shader #396 - frac shader02 octo	801
Shader #401 - frac shader03 size	804
Shader #403 - frac shader03 wormhole	807
Shader #404 - frac shader03 wormhole2	810
Shader #405 - frac shader03 wormhole3	814
Shader #406 - frac shader03 wormhole4	817
Shader #413 - frac shader05	820
Shader #414 - frac shader diamond	822
Shader #415 - frac shader dmd4 glass	824
Shader #420 - frac star1	827
Shader #421 - frac texture	829
Shader #430 - frac1 shader04 tri	831
Shader #456 - g drum by mouse	834
Shader #459 - g ufo 3d	835
Shader #460 - g ufo 3d v2	836
Shader #465 - gem aura	836
Shader #475 - gem spiral frac	838
Shader #481 - gem kale large	839
Shader #503 - glass mouse	841
Shader #504 - glass mouse rad	842
Shader #505 - glass PI	842
Shader #506 - glitch no noise mouse	843
Shader #529 - gpt halluc	844
Shader #554 - grid by mouse	847
Shader #567 - halluc gem	848
Shader #568 - halluc gem2	850
Shader #570 - halluc pop	852
Shader #571 - halluc pop2	854
Shader #576 - hue mouse	855

Shader #582 - huri af	857
Shader #583 - huri create mouse	858
Shader #593 - inthesky	859
Shader #599 - kale mouse	861
Shader #601 - kscopic 3d mouse	861
Shader #627 - matrix mouse	862
Shader #646 - mirror bubble zoom mouse	863
Shader #647 - mirror bubble	864
Shader #650 - mirror color mouse	864
Shader #653 - mirror comb mouse	865
Shader #654 - mirror fish mouse	867
Shader #655 - mirror frac mouse z2	867
Shader #722 - neon mouse	869
Shader #757 - PI x1.mp4	870
Shader #827 - rainbow mouse x1	871
Shader #846 - random soul by mouse	872
Shader #900 - ripple mouse2	873
Shader #902 - ripple outMouse	874
Shader #916 - seek	874
Shader #917 - seek2	875
Shader #961 - spiral mouse	876
Shader #993 - swirl by mouse	877
Shader #994 - swirl by mouse2	878
Shader #995 - swirl by mouse full	879
Shader #996 - swirl by mouse pos	880
Shader #998 - swirlMouse	881
Shader #1044 - warp ppong by mouse	882
Shader #1045 - warp ppong mouse	883
Shader #1047 - warpcursor	884
Shader #1048 - water cursor	885
Shader #1077 - xor rgb rainbow mouse	885
Shader #1081 - xor rgb smoother mouse	887
Shader #1085 - xorMouse	888
Shader #1092 - zoom in out mouse	889
Distortion & Warp (113 shaders)	889
Shader #0 - 3high	889
Shader #5 - 8huri	890
Shader #20 - air	891
Shader #21 - air full	891
Shader #34 - atan bowl	891
Shader #54 - bend	892
Shader #55 - bend2	893
Shader #57 - bend dir	893
Shader #58 - bend rev	894
Shader #59 - bend twist	895
Shader #77 - bowl	896
Shader #83 - brokenglass	896
Shader #136 - comb	897
Shader #157 - cosine swirl	898
Shader #158 - cosine swirl two	898
Shader #167 - crystalbend	899
Shader #172 - cwarp	901
Shader #179 - d1	901
Shader #180 - damaged vcr	902
Shader #194 - data	903
Shader #202 - deeps1	904
Shader #203 - deeps3	905

Shader #204 - deepseek3	905
Shader #205 - deepseek4	907
Shader #206 - deform	908
Shader #207 - deform cos	908
Shader #208 - deform tan	908
Shader #219 - Dispersion	909
Shader #220 - DispersionRotate	910
Shader #221 - DispersionX	912
Shader #222 - DispersionX Audio	914
Shader #223 - DispersionY	915
Shader #224 - DispersionZ	917
Shader #225 - distort disp	918
Shader #227 - distortf	919
Shader #228 - distortf2	919
Shader #229 - distortf3	920
Shader #230 - distortf4	920
Shader #234 - drain bend	921
Shader #244 - ds1	922
Shader #316 - effect	922
Shader #317 - elastic	923
Shader #341 - fisheye	923
Shader #343 - fisheye pi.mp4	924
Shader #344 - fisheye warp	925
Shader #352 - frac cache	925
Shader #372 - frac shader02 dmd6i air twist	927
Shader #392 - frac shader02 dmdXi3 warp	932
Shader #454 - g5	934
Shader #457 - g swirl	935
Shader #463 - gboil	936
Shader #464 - gem af	936
Shader #468 - gem deep	937
Shader #476 - gem spiral full	938
Shader #477 - gem spoke	939
Shader #479 - gem frac spir	939
Shader #485 - gem polar	941
Shader #488 - gem txt	941
Shader #524 - gltichtest	942
Shader #525 - gmir	943
Shader #527 - goo	945
Shader #534 - gptswirl	945
Shader #535 - gptswirl2	946
Shader #536 - grad	946
Shader #537 - grad2	947
Shader #538 - grad3	948
Shader #559 - gsupercool	949
Shader #560 - gsupercool2	950
Shader #578 - huri	951
Shader #579 - huri1	951
Shader #580 - huri2	952
Shader #581 - huri3	953
Shader #584 - hurixyz	953
Shader #585 - huriz	954
Shader #589 - ice	955
Shader #600 - kscopic	956
Shader #622 - lucy	956
Shader #731 - o3 01	957
Shader #732 - obj stretch	958

Shader #733 - obj stretch2	958
Shader #736 - ol	959
Shader #779 - prism glass size	959
Shader #799 - putty	960
Shader #801 - radial distortion	961
Shader #802 - radial distortion2	962
Shader #803 - radwarp	962
Shader #850 - react	963
Shader #851 - react10	963
Shader #852 - react11	963
Shader #921 - set swirl pos mosue	964
Shader #941 - sine swirl	965
Shader #949 - smoke	966
Shader #958 - spaz	966
Shader #960 - spiral center	967
Shader #962 - spiral by time	968
Shader #992 - sweb	969
Shader #997 - swirl cache	969
Shader #1011 - thick glass	971
Shader #1021 - tridim	971
Shader #1024 - triwavedistort	972
Shader #1026 - twarp	972
Shader #1027 - twarp2	972
Shader #1029 - twist	973
Shader #1030 - twist directions	973
Shader #1031 - twist full	974
Shader #1032 - twist gpt	974
Shader #1033 - twist gpt full	975
Shader #1034 - twistex	976
Shader #1041 - warp atan	977
Shader #1042 - warp ppong time	977
Shader #1043 - warp ppong	978
Shader #1046 - warp tunnel	979
Shader #1065 - wspirai	979
Mirror & Symmetry (93 shaders)	980
Shader #231 - dmd gpt	980
Shader #236 - drain mirror	981
Shader #237 - drain mirror amp	981
Shader #238 - drain mirror top	982
Shader #279 - echo mirror 1	982
Shader #280 - echo mirror 2	983
Shader #281 - echo mirror 3	984
Shader #320 - Electric Fold	984
Shader #321 - Electric Fold Texture	987
Shader #322 - ElectricMirror	990
Shader #346 - fold fov	993
Shader #347 - fold mirror	994
Shader #348 - fold spin	994
Shader #350 - fold	995
Shader #379 - frac shader02 dmd6iFold	995
Shader #452 - funny mirror	997
Shader #453 - funnymirror2	997
Shader #638 - mirdmdrmatch	998
Shader #639 - mirdmdrspeed	998
Shader #641 - mirror air bowl	999
Shader #642 - mirror airshader1	1001
Shader #643 - mirror atan	1002

Shader #644 - mirror bowl by time	1003
Shader #645 - mirror bowl	1003
Shader #648 - mirror c	1004
Shader #649 - mirror center	1005
Shader #651 - mirror color o1	1005
Shader #652 - mirror color swirl	1006
Shader #656 - mirror geometric	1006
Shader #657 - mirror goofy	1007
Shader #658 - mirror gpt	1008
Shader #659 - mirror grad	1009
Shader #660 - mirror in	1010
Shader #661 - mirror l	1010
Shader #663 - mirror pebble	1011
Shader #664 - mirror pong	1011
Shader #665 - mirror pong2	1012
Shader #666 - mirror prism	1013
Shader #667 - mirror psyce wave all	1014
Shader #668 - mirror putty	1014
Shader #669 - mirror self	1015
Shader #670 - mirror sin osc	1015
Shader #671 - mirror spiral aura	1016
Shader #672 - mirror spiral f	1016
Shader #673 - mirror spiral	1017
Shader #674 - mirror swirly	1017
Shader #675 - mirror twist gpt	1018
Shader #676 - mirror twist sin	1019
Shader #677 - mirror twist	1019
Shader #678 - mirror twisted	1020
Shader #679 - mirror ufo wrap	1021
Shader #680 - mirror wrap dmd6i	1021
Shader #681 - mirror wrap edge	1024
Shader #682 - mirror wrap rotate snap	1025
Shader #683 - mirror wrap rotate	1025
Shader #684 - mirror wrap rotate90	1025
Shader #685 - mirror wrap spin	1026
Shader #686 - mirror wrap	1026
Shader #687 - mirror zoom	1027
Shader #688 - mirror1	1028
Shader #689 - mirror1 op	1028
Shader #690 - mirror1 s	1029
Shader #691 - mirror2	1030
Shader #692 - mirror2 op	1030
Shader #693 - mirror2 s	1031
Shader #694 - mirror3	1032
Shader #695 - mirror3 op	1033
Shader #696 - mirror3 s	1034
Shader #697 - mirror 8	1034
Shader #698 - mirror b	1035
Shader #699 - mirror blur1	1035
Shader #700 - mirror blur2	1036
Shader #701 - mirror blur3	1037
Shader #702 - mirror c	1037
Shader #703 - mirror dir	1038
Shader #704 - mirror half dir	1038
Shader #705 - mirror offset	1039
Shader #706 - mirror offset rev	1040
Shader #707 - mirror p	1040

Shader #708 - mirror r4	1041
Shader #709 - mirror r8	1041
Shader #710 - mirror rad	1042
Shader #711 - mirror rev	1043
Shader #712 - mirror rev2	1043
Shader #713 - mirror rev2y	1044
Shader #714 - mirror revy	1045
Shader #715 - mirrorg	1046
Shader #964 - spiral mirror	1047
Shader #965 - spiral mirror 2	1047
Shader #966 - spiral mirror rev	1048
Shader #967 - spiral mirror top	1048
Shader #968 - spiral mirror wave	1049
Shader #1010 - tex fold	1050
Audio Reactive (61 shaders)	1050
Shader #6 - abc	1050
Shader #7 - abc123	1051
Shader #19 - air bowl	1051
Shader #24 - airshader1	1053
Shader #31 - animation	1054
Shader #36 - audio1	1055
Shader #37 - audio2	1056
Shader #44 - aura5	1057
Shader #45 - aura6	1061
Shader #46 - aura7	1063
Shader #47 - aura8	1066
Shader #48 - aura9	1069
Shader #49 - auraXi1	1072
Shader #50 - auraXi2	1074
Shader #51 - auraXi3	1076
Shader #62 - black rings	1079
Shader #68 - blue 2	1081
Shader #69 - blue 3	1082
Shader #70 - blue 4	1082
Shader #169 - crystalprism	1083
Shader #195 - datam	1084
Shader #196 - datam2	1085
Shader #197 - datam3	1086
Shader #198 - datam4	1087
Shader #199 - datam5	1088
Shader #200 - datam a	1088
Shader #323 - empty	1089
Shader #358 - frac shade02 dmdXi3 rot	1090
Shader #365 - frac shader02 dmd2 amp	1092
Shader #371 - frac shader02 dmd6i air	1095
Shader #373 - frac shader02 dmd6i amp	1100
Shader #374 - frac shader02 dmd6i audio	1103
Shader #375 - frac shader02 dmd6i bowl	1106
Shader #387 - frac shader02 dmdi radial	1110
Shader #388 - frac shader02 dmdXi	1113
Shader #389 - frac shader02 dmdXi2	1115
Shader #390 - frac shader02 dmdXi3	1118
Shader #391 - frac shader02 dmdXi3 drain	1120
Shader #393 - frac shader02 dmdXi4	1122
Shader #394 - frac shader02 dmdXi5	1125
Shader #395 - frac shader02 dmdXi amp	1129
Shader #397 - frac shader02 octo amp	1132

Shader #398 - frac shader02 octo strobe	1135
Shader #399 - frac shader02 octo strobe2	1138
Shader #400 - frac shader02 prisim	1142
Shader #402 - frac shader03 worm4 amp	1145
Shader #407 - frac shader03 wormhole amp	1149
Shader #469 - gem fish	1153
Shader #471 - gem image	1153
Shader #480 - gem glass	1154
Shader #483 - gem p	1155
Shader #586 - HyperFcousAmp	1156
Shader #755 - pencil draw	1157
Shader #756 - pencil draw lines	1159
Shader #922 - shader1	1160
Shader #1000 - tdye	1160
Shader #1001 - tdye2	1161
Shader #1008 - test	1163
Shader #1009 - test cache	1164
Shader #1015 - today	1165
Shader #1063 - wormhole audio	1166
Liquid & Fluid (56 shaders)	1169
Shader #92 - c ripple	1169
Shader #159 - cripple	1170
Shader #160 - cripple2	1170
Shader #175 - cxripple	1171
Shader #330 - eye2	1171
Shader #331 - eyes	1173
Shader #349 - fold water	1174
Shader #455 - g drum	1175
Shader #474 - gem ripple	1175
Shader #482 - gem metal	1176
Shader #484 - gem plas	1177
Shader #569 - halluc liquid	1177
Shader #588 - HyperVortex	1180
Shader #591 - inflate ripple	1181
Shader #610 - Liquid Censorship	1182
Shader #611 - Liquid Crystal	1182
Shader #612 - Liquid Crystal 2	1185
Shader #613 - Liquid Crystal Rainbow1	1187
Shader #615 - Liquid Heat	1190
Shader #616 - Liquid Heat blend	1193
Shader #617 - Liquid Light Rainbow Blend	1195
Shader #618 - Liquid Light Time	1198
Shader #619 - Liquid Rainbow	1201
Shader #620 - Liquid Rainbow Wave	1203
Shader #734 - ocean	1205
Shader #735 - ocean fast	1205
Shader #737 - old film water	1206
Shader #788 - psych ripple	1206
Shader #791 - psyche ripple	1208
Shader #797 - purple haze	1208
Shader #891 - ripple amp	1209
Shader #892 - ripple	1210
Shader #893 - ripple a	1210
Shader #894 - ripple cache	1211
Shader #895 - ripple cycle	1212
Shader #896 - ripple cycle2	1212
Shader #897 - ripple cycle circle	1213

Shader #898 - ripple cycle prism	1214
Shader #899 - ripple left	1214
Shader #901 - ripple out	1214
Shader #903 - ripple ppong	1215
Shader #904 - ripple prisim	1215
Shader #905 - ripple prism	1216
Shader #906 - ripple rainbow	1217
Shader #940 - sine ripple	1218
Shader #1022 - tripple	1219
Shader #1023 - tripple2	1219
Shader #1035 - underwater	1219
Shader #1036 - underwaterenchanced	1220
Shader #1049 - water fold ex.	1221
Shader #1050 - water	1222
Shader #1051 - water full	1222
Shader #1052 - water prism	1223
Shader #1053 - water r	1223
Shader #1054 - water rgb	1224
Shader #1055 - waterbend	1225
XOR & Math (55 shaders)	1226
Shader #28 - alpha xor	1226
Shader #113 - color effect xor	1227
Shader #134 - color xor	1228
Shader #182 - dark addup xor	1228
Shader #213 - diamond	1229
Shader #218 - disk2	1230
Shader #263 - echo alpha two xor	1231
Shader #264 - echo and or xor	1232
Shader #286 - echo or and xor	1232
Shader #308 - echo xor	1233
Shader #309 - echo xor color blend	1234
Shader #310 - echo xor or and	1235
Shader #332 - faster	1236
Shader #339 - fill white	1237
Shader #385 - frac shader02 dmdi6i zoom xor	1238
Shader #386 - frac shader02 dmdi6i zoom xor amp	1242
Shader #418 - frac shader xor	1246
Shader #419 - frac shader xor2	1249
Shader #523 - glow	1252
Shader #565 - gxor	1253
Shader #566 - gxor2	1253
Shader #628 - matrix mul	1254
Shader #629 - matrix mul2	1255
Shader #630 - matrix mul3	1256
Shader #631 - matrix mul4	1257
Shader #726 - new xor scale strobe	1257
Shader #727 - nl	1258
Shader #767 - plasma xor	1259
Shader #787 - psych pattern	1260
Shader #790 - psych xor	1261
Shader #818 - rainbow expand noise xor	1261
Shader #860 - react19	1263
Shader #879 - reverse xor	1264
Shader #884 - rgb time xor	1265
Shader #909 - sac geo xor	1265
Shader #919 - self alphablend scale xor	1266
Shader #1013 - time frac xor	1267

Shader #1014 - time xor	1268
Shader #1067 - xor increase	1269
Shader #1068 - xor positional offset	1270
Shader #1069 - xor rgb	1271
Shader #1070 - xor rgb dark	1271
Shader #1071 - xor rgb dark 2	1272
Shader #1072 - xor rgb dark 3	1273
Shader #1073 - xor rgb dark 4	1274
Shader #1074 - xor rgb dark rainbow	1275
Shader #1075 - xor rgb new	1277
Shader #1076 - xor rgb rainbow 1	1278
Shader #1078 - xor rgb rainbow swirl limit	1279
Shader #1082 - xor sine swirl	1281
Shader #1083 - XorKern	1281
Shader #1084 - XorKernBrighter	1283
Shader #1086 - xorsheet	1284
Shader #1087 - xorsheet 2	1285
Shader #1088 - xorstrobe	1286
Trail & Motion (51 shaders)	1287
Shader #247 - echo pingpong	1287
Shader #248 - echo pingpong c	1287
Shader #249 - echo	1288
Shader #250 - echo1	1289
Shader #251 - echo2	1289
Shader #252 - echo3	1289
Shader #253 - echo 2	1290
Shader #254 - echo 3	1290
Shader #255 - echo 4	1291
Shader #256 - echo 4 mix	1291
Shader #257 - echo 4 rev	1292
Shader #258 - echo 5 mix rev	1293
Shader #259 - echo alpha	1293
Shader #260 - echo alpha three	1293
Shader #262 - echo alpha two	1294
Shader #265 - echo bgr	1295
Shader #266 - echo cf	1295
Shader #273 - echo div2	1296
Shader #274 - echo div4	1297
Shader #276 - echo g	1297
Shader #277 - echo loop	1298
Shader #278 - echo loop2	1299
Shader #282 - echo mix	1299
Shader #285 - echo or	1300
Shader #291 - echo rev x1	1301
Shader #292 - echo rev x2	1302
Shader #296 - echo s	1302
Shader #298 - echo shift	1303
Shader #299 - echo sin	1304
Shader #300 - echo sin particle	1304
Shader #301 - echo strobe x1	1305
Shader #302 - echo two	1306
Shader #303 - echo x2	1307
Shader #304 - echo x3	1307
Shader #305 - echo x4	1308
Shader #306 - echo x5	1309
Shader #307 - echo x6	1309
Shader #408 - frac shader04 echo	1310

Shader #409 - frac shader04 echo2	1312
Shader #410 - frac shader04 echo3 spin	1315
Shader #416 - frac shader echo4 spin frac	1317
Shader #417 - frac shader echo4 spin full	1320
Shader #497 - gghost	1323
Shader #500 - gkale echo	1323
Shader #501 - gkale echo2	1325
Shader #528 - gpt echo	1327
Shader #547 - green echo	1327
Shader #562 - gtrail	1328
Shader #563 - gtrail2	1328
Shader #587 - HyperFocusTrails	1329
Shader #1007 - Temporal	1330
Glitch & Digital (43 shaders)	1331
Shader #3 - 8bit norm	1331
Shader #4 - 8bit	1331
Shader #29 - analog	1332
Shader #35 - atan glitch	1333
Shader #52 - average pixels	1334
Shader #65 - block pixels	1334
Shader #117 - color glitch dark	1335
Shader #142 - comp inc	1336
Shader #146 - compo	1337
Shader #152 - composite vhs	1338
Shader #153 - composite vhs flat	1339
Shader #215 - DigitalLight	1340
Shader #216 - DigitalLightStorm	1340
Shader #507 - glitch no noise	1342
Shader #508 - glitch noise	1343
Shader #509 - glitch react color	1344
Shader #510 - glitch react	1345
Shader #511 - glitch zoom	1345
Shader #512 - glitch1	1347
Shader #513 - glitch boil	1347
Shader #514 - glitch boil2	1348
Shader #515 - glitch effect	1349
Shader #516 - glitch jump	1349
Shader #517 - glitch light	1350
Shader #518 - glitch wave	1350
Shader #519 - glitchf	1351
Shader #520 - glitchy rainbow	1352
Shader #521 - glitchy squish	1352
Shader #522 - glitchy	1353
Shader #725 - new glitch	1353
Shader #761 - pixels	1354
Shader #841 - random pixels static	1354
Shader #857 - react16	1355
Shader #858 - react17	1356
Shader #859 - react18	1356
Shader #864 - react5	1357
Shader #865 - react6	1357
Shader #867 - react8	1358
Shader #951 - smooth pixel	1358
Shader #1037 - vhs palette	1359
Shader #1038 - vhs	1359
Shader #1039 - vhs2	1360
Shader #1040 - vhs damage	1361

Fractal & Mandelbrot (33 shaders)	1362
Shader #235 - drain mandella	1362
Shader #383 - frac shader02 dmd mandella	1363
Shader #412 - frac shader04 julia	1368
Shader #431 - fractal fold neon	1371
Shader #432 - fractal fold sketch	1373
Shader #433 - fractal foldzx	1375
Shader #434 - fractal	1377
Shader #435 - Fractal1	1378
Shader #436 - fractal2	1379
Shader #437 - fractal3	1380
Shader #438 - fractal4	1381
Shader #439 - fractal color wave	1382
Shader #440 - fractal diamond rainbow	1384
Shader #441 - fractal diamond rainbow blend	1387
Shader #442 - fractal diamond rainbow no tex	1389
Shader #443 - fractal diamond rainbow overflow	1392
Shader #444 - fractal diamond rainbow overflow const	1394
Shader #445 - fractal liquid no zero	1397
Shader #446 - fractal no zero	1400
Shader #447 - fractal noise	1403
Shader #448 - fractal texture large	1406
Shader #449 - fractal texture large audio	1408
Shader #450 - fractal texture no wrap	1411
Shader #451 - fractal zoom e	1413
Shader #594 - julia	1415
Shader #614 - Liquid Fractal Tunnel	1417
Shader #625 - mandella1	1418
Shader #662 - mirror mandella	1419
Shader #723 - new fractal	1420
Shader #724 - new fractal2	1421
Shader #820 - rainbow fractal	1423
Shader #821 - rainbow fractal random	1425
Shader #826 - rainbow metal fractal	1426
Scale & Transform (29 shaders)	1427
Shader #14 - af scale	1427
Shader #15 - af scale2	1428
Shader #16 - af scale3	1430
Shader #17 - af scale puple	1431
Shader #18 - af scale spectrum	1432
Shader #95 - cd zoom	1433
Shader #96 - cd zoom out	1433
Shader #145 - comp zoom	1434
Shader #177 - cyclone zoom	1435
Shader #326 - expand contract	1435
Shader #327 - expand mul	1436
Shader #328 - expand mul fast	1436
Shader #329 - expand spot	1437
Shader #384 - frac shader02 dmdi6i zoom	1437
Shader #422 - frac zoom1	1440
Shader #423 - frac zoom2	1444
Shader #424 - frac zoom3	1447
Shader #425 - frac zoom4	1450
Shader #426 - frac zoom5	1454
Shader #427 - frac zoom6	1457
Shader #428 - frac zoom7	1461
Shader #429 - frac zoom8	1464

Shader #561 - gt rotate	1467
Shader #875 - ResizeVertical cache	1467
Shader #907 - rotate xyz	1468
Shader #908 - rotate xyz zoom	1469
Shader #912 - scale	1470
Shader #1091 - zoom fish	1471
Shader #1093 - zoom pong	1472
Blur & Smooth (19 shaders)	1472
Shader #30 - and smooth	1472
Shader #66 - bloom	1473
Shader #73 - blur offset	1475
Shader #74 - blur offset color fade	1475
Shader #75 - blurry	1476
Shader #104 - color bloom	1477
Shader #246 - dys	1478
Shader #361 - frac shader01 smooth	1478
Shader #462 - gblur	1482
Shader #532 - gptsmooth	1483
Shader #533 - gptsmooth2	1484
Shader #806 - rainbow blur	1485
Shader #881 - rgb blur	1486
Shader #882 - rgb blur time	1487
Shader #950 - smooth gl	1488
Shader #952 - smooth var	1489
Shader #990 - sum	1490
Shader #1079 - xor rgb smooth	1491
Shader #1080 - xor rgb smoother	1492
Light & Electric (18 shaders)	1493
Shader #226 - distort light	1493
Shader #362 - frac shader01 smooth neon	1494
Shader #377 - frac shader02 dmd6i neon	1498
Shader #472 - gem light frac	1501
Shader #602 - light rings	1502
Shader #603 - light pulse	1503
Shader #604 - Light Rainbow Swirl	1503
Shader #605 - light spiral	1505
Shader #606 - lightfade	1506
Shader #607 - lightfade xor	1507
Shader #608 - lightfadedark	1508
Shader #721 - neon	1509
Shader #775 - pos light	1509
Shader #819 - rainbow expand noise xor light	1510
Shader #825 - rainbow light swirl	1511
Shader #988 - strobe light	1512
Shader #1061 - whitelight	1513
Shader #1062 - wlight	1513
Wave & Oscillation (17 shaders)	1514
Shader #72 - blue wave	1514
Shader #101 - code wave	1515
Shader #102 - code wave rev	1515
Shader #103 - code wave side	1516
Shader #173 - cwave	1516
Shader #174 - cwave2	1516
Shader #184 - dark psych wave	1517
Shader #574 - heat wave	1518
Shader #795 - pulse1	1518
Shader #970 - spiral wave	1519

Shader #971 - spiral wave 1	1520
Shader #1056 - wave 1	1520
Shader #1057 - wave diag	1521
Shader #1058 - wave spiral	1521
Shader #1094 - zoom pulse	1522
Shader #1095 - zoom pulse2	1522
Shader #1096 - zoom pulse3	1523
Noise & Random (15 shaders)	1524
Shader #1 - 4ac rand	1524
Shader #2 - 4rand	1524
Shader #147 - composite static	1525
Shader #289 - echo rand	1527
Shader #290 - echo rand 2	1528
Shader #817 - rainbow expand noise	1529
Shader #840 - random colors	1529
Shader #842 - random pos fish	1530
Shader #843 - random resize	1531
Shader #844 - random rgb	1532
Shader #845 - random soul	1532
Shader #847 - random square	1533
Shader #848 - random strobe	1534
Shader #849 - random uneven square	1535
Shader #945 - singleton noise	1536
Tile & Mosaic (11 shaders)	1536
Shader #411 - frac shader04 grid	1536
Shader #553 - grid spiral	1539
Shader #555 - grid pattern	1539
Shader #556 - grid rand strobe	1540
Shader #557 - grid strobe	1541
Shader #558 - gridcolor gradient	1542
Shader #759 - pigrid	1543
Shader #839 - rand blocks	1543
Shader #885 - rgbchecker	1544
Shader #975 - square grid	1545
Shader #976 - square grid2	1546
Acid & Psychedelic (9 shaders)	1546
Shader #8 - acid color2	1546
Shader #9 - acidcam	1547
Shader #10 - acidcolor	1547
Shader #11 - acidcolorrand	1548
Shader #242 - dream tunnel	1549
Shader #243 - dream tunnel2	1550
Shader #530 - gpt trip	1551
Shader #531 - gpt trip2	1553
Shader #792 - psyche wave	1556
Blend & Composite (7 shaders)	1556
Shader #12 - addup blend	1556
Shader #148 - composite	1557
Shader #149 - composite2	1557
Shader #150 - composite3	1558
Shader #151 - composite crt	1561
Shader #168 - crystalblend2	1562
Shader #918 - self alphablend scale	1563
Edge & Outline (7 shaders)	1564
Shader #311 - edge black	1564
Shader #312 - edge face	1564
Shader #313 - edge fill	1565

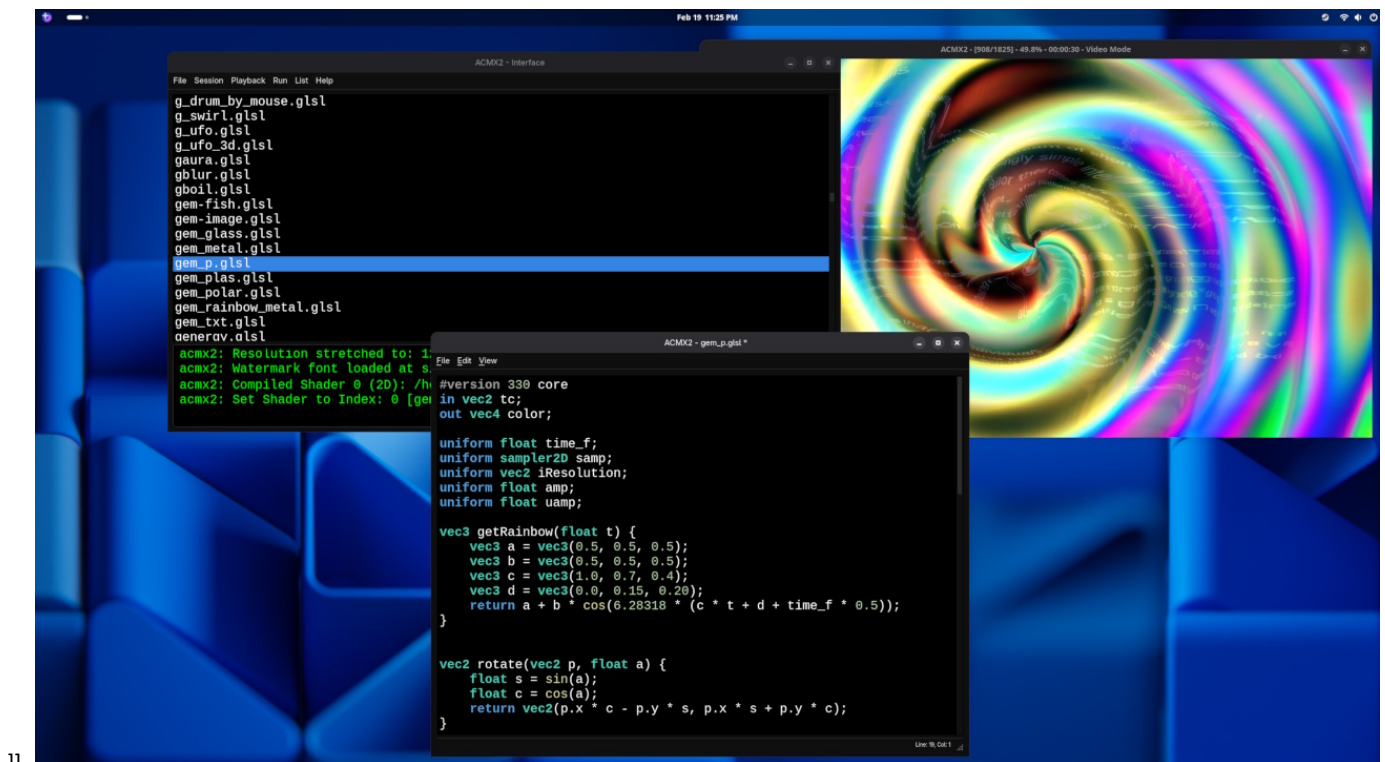
Shader #314 - edge pencil	1566
Shader #315 - edge rainbow	1567
Shader #781 - prism glass wall noedge	1568
Shader #963 - spiral edge	1569

ACMX2 Programming Guide

A complete technical reference and effects catalog for acidcam-gpu and ACMX2.

- Part I: Technical Reference - Architecture, pipeline, API, build system
- Part II: CUDA Filter Reference - All 905 GPU filter kernels with code
- Part III: GLSL Shader Reference - All 1070 fragment shaders with code

LostSideDead - ACMX2 Technical Reference



Source: <https://github.com/lostjared/acidcam-gpu>

acidcam-gpu + ACMX2 - Complete Technical Reference

acidcam-gpu is a CUDA-accelerated real-time video effects engine. It contains a library of 905 GPU filter kernels that run entirely on the NVIDIA GPU, a unified dispatch kernel that chains any ordered subset of those filters per frame, a rotating device-side frame history buffer for temporal effects, and a CLI application that drives the full pipeline from camera or file input through to live display and optional MXWrite output encoding.

ACMX2 is the Qt-based orchestration layer - it wraps the same library and CLI with a visual interface, shader pass ordering, session persistence, and process monitoring. Together they form a layered creative tool: CUDA for pixel-level power, OpenGL for display, Qt for control, and Podman containers for distribution.

The system operates as a two-stage GPU pipeline: first, CUDA kernels process every pixel in parallel on the GPU's streaming multiprocessors (one thread per pixel), applying a user-defined chain of filters entirely in device memory. Then, the CUDA output is transferred to OpenGL via a Pixel Buffer Object (PBO) - a zero-copy GPU-to-GPU transfer - where it becomes a texture. GLSL fragment shaders then apply additional full-frame visual effects in one or more stacked passes using ping-pong framebuffers. Both the CUDA filter chain and the GLSL shader pass stack are independently configurable and orderable, producing a combinatorial explosion of possible visual outcomes.

Stack: C++20 / CUDA 12.x / OpenCV CUDA / OpenGL / Qt6 / MXWrite

Program Screenshot

This is the current desktop ACMX2/acidcam-gpu interface in action.

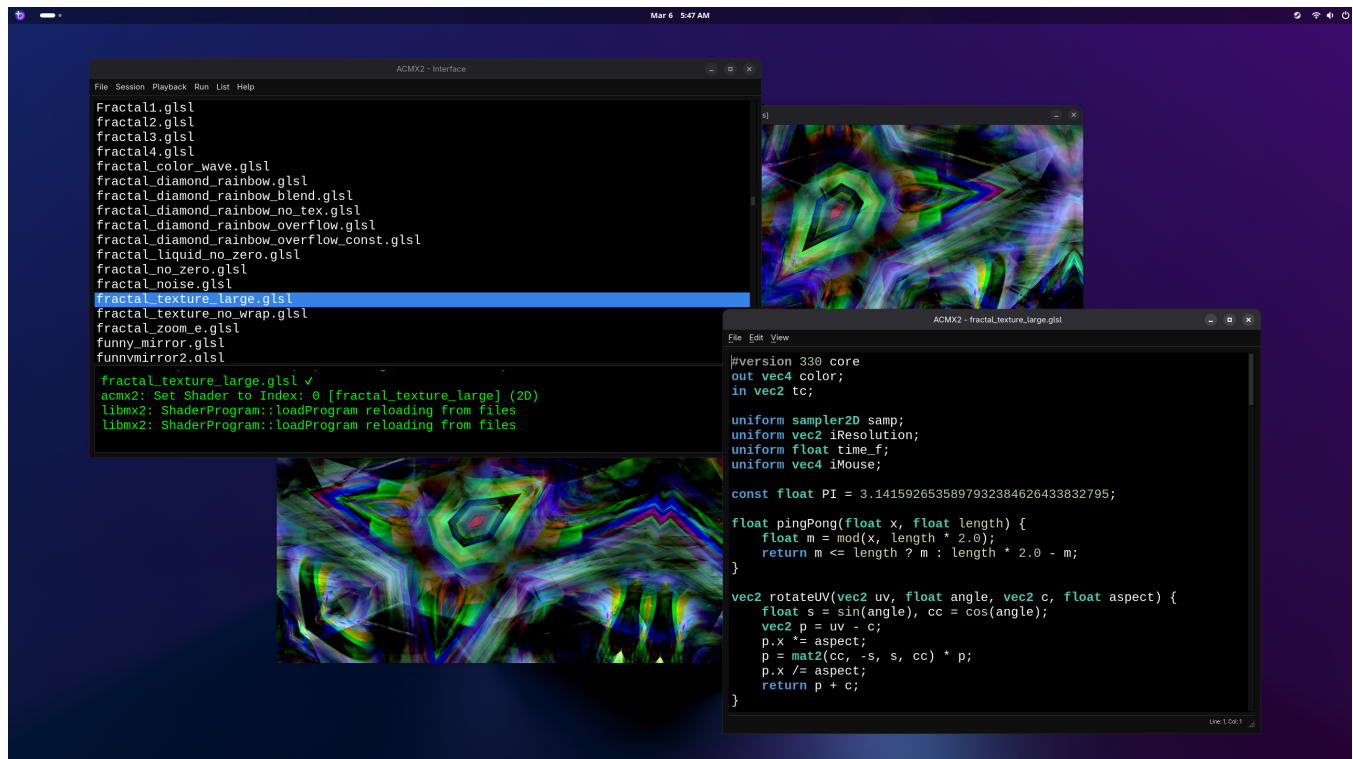


Figure 1: ACMX2 and acidcam-gpu running on desktop

Project Origin and Purpose

The Acid Cam project originated as a CPU-based filter library (libacidcam) that applied artistic glitch effects to camera and video frames. As resolutions grew and filter stacks became more complex, CPU-based pixel processing became a bottleneck - applying dozens of per-pixel transforms on high-resolution video ate into frame budget quickly.

acidcam-gpu was created to solve this. The solution was to port the entire filter library to NVIDIA CUDA, so all pixel work runs massively in parallel on the GPU. A unified dispatch kernel allows any ordered combination of the 905 available filters to run in a single GPU pass per frame, without recompiling. The result is a system capable of applying complex multi-layer visual transforms at full framerate on modern NVIDIA hardware.

Why CUDA Kernels and GLSL Shaders?

Both CUDA kernels and GLSL shaders execute on the same physical hardware - the GPU's streaming multiprocessors (often marketed as "CUDA cores"). The difference is the programming model, not the silicon. CUDA is NVIDIA's general-purpose compute API: kernels can freely read from arbitrary device memory, perform complex conditional logic, access a ring buffer of historical frames, and do per-pixel computation without any graphics pipeline constraints. GLSL runs through the OpenGL graphics pipeline as fragment shader programs, which excel at texture sampling, interpolation, and full-frame image processing with Shadertoy-compatible uniforms like time, resolution, and mouse position. By using both programming models on the same GPU hardware, the system gets the best of each approach.

How Each Technology Is Used

- **CUDA (pixel-level compute):** The 905 filter kernels run as CUDA threads - one thread per pixel - across the GPU's streaming multiprocessors simultaneously. A 1920x1080 frame launches over 2 million threads in parallel. The unified kernel loops through the user's selected filter chain, applying each filter sequentially to the pixel data in-place. Filters can read from a ring buffer of prior frames stored entirely in GPU memory, enabling temporal effects like MedianBlend (averaging across all history frames), AuraTrails (blending against frames at indices 1, 4, and 7), and MatrixOutline (comparing against a frame from 4 steps back). This kind of arbitrary multi-frame memory access is natural in the CUDA compute model but difficult or impossible within the GLSL fragment shader pipeline, which is designed around single-texture-per-pass processing.
- **GLSL (full-frame shader effects):** After CUDA processing, the result is transferred to an OpenGL texture via PBO interop (no CPU round-trip). GLSL fragment shaders - running on the same GPU hardware via the OpenGL graphics pipeline - then process the entire frame as a texture, applying effects like color grading, distortion, glow, CRT simulation, and other post-processing transforms. The shader library supports Shadertoy-compatible uniforms (iTime, iResolution, iMouse, iFrame, etc.) and can render onto 2D quads or 3D model geometry via MXMOD meshes.
- **Composition:** CUDA filters and GLSL shaders compose at the frame level. CUDA runs first on the raw pixel data, then GLSL operates on the CUDA-filtered result. Both stages support stacking - multiple CUDA filters chain inside a single kernel launch, and multiple GLSL shaders chain via ping-pong framebuffer passes. The full pipeline is: Camera -> CUDA filter chain -> PBO transfer -> GLSL shader pass 1 -> GLSL shader pass 2 -> ... -> Screen.

ACMX2 Qt Application - Orchestration Layer

ACMX2 is the Qt6 application that wraps the acidcam-gpu CLI and library into a controllable desktop session. It handles shader pass management, GPU filter ordering, session persistence, and process lifecycle - it does not re-implement any pixel processing itself.

- **Main window:** ACMX2/interface/main_window.cpp - QProcess supervision, log rendering, session menu
- **GPU filter dialog:** ACMX2/interface/gpufilter.cpp - calls --list-filters, parses output, stores ordered selection
- **Shader pass ordering:** ACMX2/interface/shaderpass.cpp - explicit multi-pass chain, output of pass N feeds into pass N+1
- **Audio integration:** ACMX2/audio.cpp/.hpp - RtAudio amplitude extraction, reactive parameter modulation
- **Shader cache:** ACMX2/program.cpp/.hpp - binary cache with source+driver fingerprinting to skip recompile
- **3D geometry:** ACMX2/models/*.mxmod - MXMOD-format geometry used in scene-influenced render stages
- **Build:** ACMX2/CMakeLists.txt and ACMX2/interface/CMakeLists.txt - validates CUDA, OpenCV CUDA headers, FFmpeg, MXWrite, Qt6, RtAudio at configure time

How the Qt Interface Works

1. **Filter discovery:** `gpufilter.cpp` spawns the `acmx2` binary with `--list-filters`, reads `index:name` lines from `stdout`, sorts the list alphabetically, and populates a selection dialog. The user picks from the full 905-filter catalog and reorders them via drag or list controls.
 2. **Chain persistence:** selected filter order is stored as an ordered index list and passed back to `acmx2` as arguments at run time. Changing the order produces a completely different visual output without touching any code.
 3. **Process supervision:** `main_window.cpp` binds a `QProcess` to the `acmx2` executable (built from `ACMX2/acmx.cpp`). `stdout/stderr` streams are captured and shown in the UI log panel so runtime errors (bad camera index, missing CUDA device, encoder failure) are immediately visible.
 4. **Session settings:** `QSettings` persists executable path, shader directory, preferred styles, and last-used filter chain so sessions can be resumed exactly.
 5. **Shader pass layer:** `shaderpass.cpp` manages an ordered list of GLSL pass configs. Each pass uses the previous pass output as input, compositing GLSL effects on top of the CUDA-processed frame.
 6. **Audio-reactive path:** when `RtAudio` is enabled at compile time, an audio callback computes per-buffer amplitude average. This value can modulate alpha or other per-frame parameters, making output visually reactive to microphone or line input.
-

Standalone CLI - `acidcam-gpu/app/main_cv.cu`

The standalone CLI app (`app/main_cv.cu`) is an independent tool that drives the CUDA filter pipeline directly from the command line without `ACMX2`. It handles input device management, argument parsing, frame loop control, animation state evolution, and output encoding. It is written in CUDA C++ (compiled by `nvcc`) and targets C++20. Note: the `ACMX2` Qt interface does not supervise this file - instead, `QProcess` launches and monitors the `acmx2` executable (built from `ACMX2/acmx.cpp` ↪), which contains its own runtime loop and GL/CUDA pipeline.

Startup: Device and Camera Discovery

- `checkDevices()`: calls `cudaGetDeviceCount()`. If the count is zero or the call errors, it prints a detailed message (driver not installed, GPU not seated) and exits immediately. On success, prints the CUDA device short info via OpenCV's `cv::cuda::printShortCudaDeviceInfo()`.
- `listGraphicsCards()`: enumerates all CUDA devices using `cv::cuda::DeviceInfo`, prints each device's name and total VRAM in MB.
- `listCameras()`: probes `/dev/video0` through `/dev/video9` with `cv::VideoCapture`. During probing, `stderr` is redirected to `/dev/null` to suppress OpenCV's verbose device error messages. For each working device, reads its human-readable name from `/sys/class/video4linux/videoN/name`.
- **Signal handling:** a custom Interrupt exception class and a `signalHandler` that throws it allows `SIGINT/SIGTERM` to cleanly unwind the frame loop without leaving the CUDA device in a dirty state.

AnimationState The Per-Frame Parameter Evolution Engine

The `AnimationState` struct (global `gState`) drives the continuous evolution of visual parameters each frame. This is what makes the visuals non-static the same filter chain produces different pixel values every frame because the parameters feeding it change continuously:

- **alpha oscillation:** alpha starts at 1.0 and increments by 0.01f per frame until it reaches 3.0f, then decrements back to 1.0f. This creates a smooth breathing rhythm in blend-intensity across all alpha-parameterized filters. The full cycle takes 400 frames at 60fps 6.7 seconds.
- **frame index oscillation:** `current_frame_index` bounces between 0 and `arraySize - 1`, incrementing or decrementing by 1 each frame based on `index_dir`. This continuously sweeps which historical frame is selected by the `start_index` parameter effects that read from frame history change which past frame they reference every tick.

- **square_size oscillation:** `square_size` oscillates between 2 and 64 pixels, changing by 2 per frame. This controls block and tile dimensions for square-based effects. The oscillation causes visible growth and shrinkage in any block-decomposition filter.

`updateAndDraw()` The Frame Processing Function

Called every frame of the main loop, this function performs the full GPU pipeline:

1. Updates all three `AnimationState` values (`alpha`, `frame_index`, `square_size`) based on their current direction and bounds.
2. Copies device frame pointers from the `DynamicFrameBuffer`'s `rawPointers` vector into the device-side pointer array (`d_ptrList`) via `cudaMemcpy(HostToDevice)`.
3. Copies the most recently uploaded frame from the buffer into the working GPU buffer using `cudaMemcpy2D(DeviceToDevice)` this is a pure device-to-device copy, no round-trip through CPU RAM.
4. Calls `launch_filter()` with the current animation state, the working buffer, and the active filter list. The CUDA kernel runs and transforms the working buffer in-place.
5. The processed frame in the GPU working buffer is then downloaded or blitted for display/encoding.

Output Encoding via `MXWrite`

When recording is enabled, processed frames are passed to the `MXWrite` library which wraps FFmpeg encoding. `MXWrite` accepts raw frame data and handles container muxing, codec selection, and file writing. This keeps encoding off the critical path the CUDA filter loop is not blocked by disk I/O.

The Public API `ac-gpu.hpp`

The header `acidcam-gpu/include/ac-gpu/ac-gpu.hpp` defines the complete ABI contract between the host application and the CUDA engine. Everything the caller needs to know lives here.

Key Types and Definitions

- `AC_FILTER_MAX = 905` the total number of available filters. Filter indices run from 0 to 735 inclusive. This constant lets callers validate index bounds before dispatch.
- `struct GPUFilter { int index; }` the lightweight device-side representation of a filter. Only the integer index crosses to the GPU; names and metadata stay on the host.
- `struct Filter { int index; std::string name; GPUFilter toGPU() const; }` the host-side filter descriptor. `toGPU()` produces the compact `GPUFilter` for device transfer.
- `class ACException` thin exception type for runtime errors (bad resolution strings, missing devices). Carries a message string via `why()`.
- `CHECK_CUDA(call)` macro that wraps any CUDA runtime call, checks the error code, prints file/line/message, and exits on failure. Used throughout both the library and the CLI app for consistent error handling.
- `extern Filter filters[]` the master filter table defined in `filters.cu`, exposed for host-side iteration. Callers can walk this array to build filter lists, populate UI controls, or serialize chain configs.

The `DynamicFrameBuffer` Class

`DynamicFrameBuffer` manages a ring of historical frames entirely in device memory (`cv::cuda::GpuMat` $\hookrightarrow$). This is the mechanism that makes temporal effects possible filters can read from any prior frame in the ring without downloading anything to host memory.

- **Construction:** takes an `arraySize` parameter (e.g., 8) that sets ring depth. Internally creates a `std::vector<cv::cuda::GpuMat>` and a parallel `std::vector<unsigned char*>` of raw device pointers.
- `update(const cv::Mat& inputFrame):` 1. Uploads the CPU frame into `d_uploadBuffer` (a staging `GpuMat`) via `d_uploadBuffer.upload(inputFrame)`. 2. On resolution change, reallocates all

ring slots as CV_8UC4 (8-bit per channel, 4 channels RGBA) at the new dimensions and resets `completedFrames = 0`. 3. Calls `std::rotate(deviceFrames.begin(), deviceFrames.begin()+1, deviceFrames.end())` this shifts the oldest frame to position `back()` without copying pixel data, just rotating smart handles. 4. Converts and copies the new frame into the back slot: if the upload buffer is 3-channel BGR (standard OpenCV camera output), `cv::cuda::cvtColor(d_uploadBuffer, deviceFrames.back(), cv::COLOR_BGR2RGBA)` converts to 4-channel RGBA on the GPU. If already 4-channel, a direct `copyTo` is used. 5. Syncs `rawPointers[i] = deviceFrames[i].data` for all slots, giving the kernel a plain raw pointer into each frame's device memory. 6. Increments `completedFrames` up to `arraySize`, tracking how many valid (non-zero) history entries exist before the ring fills.

- `getDeviceFramePointers()`: returns the `rawPointers.data()` a host-side array of device pointers. This is copied to a device-side pointer array before kernel launch so the kernel can dereference historical frame data.
- **Memory layout:** each `cv::cuda::GpuMat` is a pitched allocation. The `framePitch` value (row stride in bytes) is stored on first allocation and passed to kernel as `step`. Pitched memory improves coalescing for row-based access patterns.

The `launch_filter()` Function

This is the C-linkage function that the host calls every frame to execute the filter chain. It handles lazy filter list management and kernel dispatch:

1. **Guard:** returns immediately if `c == 0` (no filters), dimensions are zero, or `numFrames` is zero.
2. **Lazy filter list rebuild:** if `changed == true` or the device filter list pointer is null, it synchronizes the device (`cudaDeviceSynchronize`), frees any existing device list, converts host `Filter[] GPUFilter[]` using `toGPU()`, allocates a new device array with `cudaMalloc`, and copies with `cudaMemcpy(HostToDevice)`. Sets `changed = false` after update. This means re-ordering the filter chain is essentially free at runtime the rebuild only happens when the chain actually changes.
3. **Grid/block computation:** `dim3 blockSize(16, 16)` 256 threads per block arranged in a 2D tile. `dim3 gridSize((width+15)/16, (height+15)/16)` enough tiles to cover the full frame, rounded up so edge pixels are handled correctly.
4. **Per-frame parameter assembly:** a fresh `FilterParams` struct is built each frame using the current animation state and fresh random values from `rand()`. This per-frame randomness is what makes the visuals continuously evolve without any explicit animation scripting.
5. **Kernel dispatch:** `unifiedFilterKernel<<<gridSize, blockSize>>>(d_list, c, data, allFrames, w, h, step, params)`.
6. **Synchronization:** `cudaDeviceSynchronize()` is called after the kernel launch to ensure the frame is fully processed before the result is read or displayed.

FilterParams The Per-Frame Parameter Bundle

The `FilterParams` struct is defined inside the `ac_gpu` namespace in `filters.cu` and is passed by value into the kernel. Every filter case inside the kernel reads from this shared param bundle rather than maintaining its own state, which is what enables stateless per-pixel execution. The parameters are populated freshly each frame by `launch_filter`:

- **float alpha** blend intensity multiplier. In the CLI, this oscillates between 1.0 and 3.0 (0.01/frame). Used by blending and alpha-scaling filters to modulate mix ratios. At 1.0, blends are subtle; at 3.0, blends are oversaturated and aggressive.
- **bool isNegative** when true, the `setAlpha` device function inverts all three color channels (255 - value) after each pixel's filter chain completes. This globally toggles a photographic negative effect that stacks on top of any filter output.
- **int numFrames** number of valid entries in the history ring (up to `arraySize`). Filters that index into `allFrames[]` use this to avoid reading uninitialized slots before the buffer is full.

- `int square_size` block tile dimension in pixels, oscillates between 2 and 64. Controls the tile/block granularity for effects like `SquareBlockResize`, `DiagPixelated`, `pixelScale`, and others that work in rectangular pixel regions.
- `int start_index` the current bouncing frame selection index (0 to `arraySize-1`). Passed directly to effects that use it as a frame history lookup index, creating smooth temporal sweeping.
- `int start_dir` the direction of the frame index bounce (1 = forward, 0 = reverse). Some effects use this to modulate the direction of their temporal lookup.
- `int int_param1` `rand() % height`. Used as a random vertical offset for pixel shift effects (`PictureShiftDown`, `DistortedLinesY`, etc.). Changes every frame so shifts appear random and non-repeating.
- `int int_param2` `rand() % width`. Same concept as `int_param1` but horizontal. Used for horizontal shift and distortion effects.
- `float float_param1` `(float)(rand() % 255)`. A random brightness or gradient value. Used in effects like `GradientFlashColor` where a random scalar modulates the gradient output per frame.
- `int seed` `rand()`. Per-frame random seed passed to `gpu_rand(x, y, seed)` inside device functions. Since `gpu_rand` is deterministic given the same (x, y, seed), all threads with the same seed get the same noise map but the map changes every frame because the seed is regenerated.
- `int frame_count` monotonically increasing counter (`frame_counter++`). Used for time-based oscillations inside device functions (e.g., `processTripHSV` uses it to advance HSV hue over time, `get_osc_offset` uses it for sinusoidal offsets). This is the primary time variable for all internally animated kernels.
- `int sumR, sumG, sumB` each set to `rand() % 255` per frame. Used in bitwise XOR effects like `XorSumStrobe` where the pixel's channels are XOR'd against a random color triplet, creating a color-shifting strobe.
- `int threshold` fixed at 15. Used by `colorBounds()` in effects that do color comparison (e.g., `StrangeGlitch` switches to a historical frame pixel when the current and historical color differ by more than the threshold). Tuned to ignore slight color noise while catching meaningful motion.
- `int sw, sh` dynamic block dimensions: `16 + (frame_counter % 48)`. Creates a slowly growing/shrinking block size for diagonal pixelated effects like `DiagPixelatedResize`. The range is 16 to 63 pixels wide/tall, cycling over 48 frames.

The Unified Kernel `unifiedFilterKernel`

The `__global__ void unifiedFilterKernel` is the single CUDA kernel that executes the entire filter chain per frame. It is the pixel-level engine of the whole project. Understanding how it works explains why the system can run any combination of 905 effects in real-time.

Thread Layout and Pixel Assignment

Each CUDA thread is assigned one pixel:

- `int x = blockIdx.x * blockDim.x + threadIdx.x`
- `int y = blockIdx.y * blockDim.y + threadIdx.y`

With `blockDim = (16, 16)`, each block contains 256 threads arranged as a 16x16 tile of pixels. A 1920x1080 frame requires `ceil(1920/16) * ceil(1080/16) = 120 * 68 = 8160` blocks. At 256 threads per block, that is 2,088,960 threads one per pixel all executing simultaneously on the GPU.

A bounds check (`if (x >= width || y >= height) return`) handles edge tiles where the grid doesn't align perfectly with frame dimensions.

The Filter Chain Loop

Inside each thread's pixel scope, the kernel loops over the filter list:

```
for (int i = 0; i < count; ++i) {
    switch (filters[i].index) {
```

```

        case 0: processSelfAlphaBlend(x, y, data, step, params); break;
        case 1: processMedianBlend(x, y, data, allFrames, step, params); break;
        // ... 902 more cases ...
        case 904:
            acgl_glitch_NewVarBlendLines(x, y, data, allFrames, width, height, step, params);
            break;
    }
}

```

Each `process*()` call is a `__device__` function that reads and writes the pixel at `data[y * step + x ↵ * 4]` in RGBA byte order. Because all threads within a warp execute the same filter index at the same time, the switch dispatch is warp-coherent when all threads in the warp are processing the same filter which they always are, since the filter list is shared params, not per-pixel data.

After the loop completes, `setAlpha(data, y * step + x * 4, params.isNegative)` is called once to finalize the alpha channel and optionally negate the color.

Device Helper Functions

- `__device__ float gpu_rand(int x, int y, int seed)` a deterministic per-pixel hash function. Uses three multiply-and-XOR operations seeded by x, y, and the per-frame seed, followed by XOR-shift scrambling, returning a float in [0, 1). Used by noise effects (VisualSnow, StaticGlitch, RandomPixels, DistortedLines) to generate pixel-level noise without any global shared state, which is important since CUDA threads cannot safely share mutable state.
- `__device__ void setAlpha(unsigned char* data, int idx, bool isNegative)` if `isNegative`, inverts R, G, B (three bytes at `idx`, `idx+1`, `idx+2`). Always writes 255 to the alpha byte (`idx+3`), ensuring fully opaque output regardless of what filters wrote to that byte.
- `__device__ bool colorBounds(r1,g1,b1, r2,g2,b2, ir,ig,ib)` returns true if the absolute difference of each channel pair is within the given threshold. Used by motion-detection effects: StrangeGlitch uses it to detect when a pixel has changed significantly between frames and swap in the historical pixel value; MatrixOutline uses it to zero out pixels that match a reference frame (creating a motion-outline effect).

The Filter Catalog 905 Effects Across All Categories

The filter table is defined at the top of `filters.cu` as a static array of `Filter` structs. Index 0 through 735 are defined. Each filter has its own `__device__` implementation function but all are dispatched through the single unified kernel switch. Here is a breakdown of the major effect families that make up the library:

Temporal / Trail Effects

Read from historical frames in the `DynamicFrameBuffer` ring to create motion persistence, ghosting, and smear effects.

- MedianBlend averages current pixel against all history frames + XOR + contrast boost
- AuraTrails blends against frames at history indices 1, 4, and 7
- MotionGhostTrails linear blend between current and oldest frame via alpha
- WaveTrails wave-modulated temporal blend
- RGBLineTrails / RGBWideTrails / RGBLongTrails per-channel trail variations with different decay widths
- ProperTrails / ShortTrail clean blend-from-history patterns
- AcidTrailsBlend / GhostTrailsBlend higher-level composites of trail motion
- EchoBlend / EchoShift / TrailEcho echo-delay style history reads
- FrameBlendMulti / FrameBlendMultiX blending across multiple history slots simultaneously

Geometric Distortion

Remap pixel coordinates shift, stretch, warp, flip, tear, and distort spatial layout.

- PictureShiftDown / PictureShiftRight / PictureShiftVariable whole-frame translate by random pixel offset
- StretchR/G/B_Right / StretchR/G/B_Down per-channel horizontal or vertical stretch
- TearRight / TearDown VHS-style tape tear distortions
- DistortionByRow / DistortionByCol row/column-level random displacement
- RippleEffect / ShockWave / TwistEffect radial math-based spatial warps
- FishEye / TunnelEffect / VortexEffect / Kaleidoscope polar coordinate remap effects
- SpiralWave / SpiralTrail spiral coordinate mapping
- ZoomBlur / RadialBlur / RotateBlend zoom and rotational blur patterns
- ExpandContract / ExpandLeftRight / DiagInward pixel expansion/contraction patterns
- MirrorWave / MirrorWaveX / MirrorWaveY wave-modulated mirror remap

Color Manipulation

Alter pixel color values mathematically without spatial remapping.

- SelfAlphaBlend multiplies each channel by (1 + alpha)
- SelfScaleRefined / SelfScaleByFrame channel scaling with clamping
- TripHSV converts to HSV, cycles hue by frame_count, converts back
- GradientRainbow / GradientSelf / GradientDown / GradientHorizontal positional gradient overlays
- FadeRtoGtoB / FadeRGB_Speed / FadeRandomChannel sequential channel fade patterns
- ColorAccumulate1/2/3 / colorAccumulate accumulation blend series
- HueRotate / ChromaticAberration / RGBShift color space shift effects
- Posterize / Solarize / GammaBright / GammaDark / ContrastBoost / ContrastReduce tone-mapping operations
- TruncateColor / TruncateVariable / TruncateVariableScale color depth reduction patterns
- ColorDrift / ColorPulse / ColorPulseRGB / ColorFadeFilter time-animated color drift series

Bitwise / XOR Operations

Apply bitwise logic (XOR, AND, OR) between pixel channels, historical frames, or random values.

- Bitwise_XOR / Bitwise_AND / Bitwise_OR current frame XOR/AND/OR with history frame
- XorSumStrobe XORs each channel against a random sumR/G/B value
- XorAlpha / XorFade / XorSine / XorLag / XorScale XOR variant series
- SelfXorBlend / SelfXorDoubleFlash / SelfOrDoubleFlash self-XOR patterns
- BitwiseXorStrobe / BitwiseRotateBlend / BitwiseXorScaleBlend scaled and rotated XOR blends
- AndStrobe / AndStrobeScale / AndOrXorStrobe / AndOrXorStrobeScale AND-based strobe family
- MedianBlendXor / CollectionAlphaXor XOR composited with median blend

Glitch and Noise

Simulate digital corruption, static, tracking errors, and signal noise.

- StrangeGlitch color-bounds detection, swaps pixel with history when change detected
- HorizontalGlitch / VerticalGlitch glitch lines by row or column history sampling
- PixelGlitch / StaticGlitch / LineGlitch / BoxGlitch progressive glitch types
- GlitchBlock / GlitchBlockXor / GlitchLine / GlitchLineX block and line glitch family
- VisualSnow / VisualSnowX2 gpu_rand-based noise overlay
- NoiseBlend / NoiseBlendX2 / NoiseXor / NoiseBlendX noise blend family
- VHSTracking emulates VHS head-tracking horizontal tear artifact
- DataCorrupt / DigitalArtifact / TapeGlitch / ColorGlitch digital corruption effects
- SliceGlitch / GlitchSort / GlitchMosaic sorted and mosaic-style glitch

Scan / Line Effects

Work on scanline-level patterns, interlacing, and whole-row/column operations.

- InvertedScanlines / ScanSwitch / ScanAlphaSwitch scanline inversion and toggle
- HorizontalLines / DiagonalLines / BlackLines / LongLines line overlay patterns
- LineInLineOut series (3947, 262264) complex oscillating line-in/line-out passes
- YLineDown / YLineDownBlend vertical line drift effects
- LineGlitch / LineGlitchX / LineAcrossX / LinesAcrossX line-based glitch patterns
- BlendedScanLines / InterlaceBlend interlace-style line blending
- ShiftLinesDown whole-scanline vertical displacement

Block / Square Effects

Operate on rectangular pixel regions pixelation, block swapping, tile-based transforms.

- SquareBlockResize divides frame into horizontal bands, each blended against a different history frame
- SquareShrink clamps inner region against history frame based on oscillating offset
- SquareByRow / SquareByRowRev / SquareByRow2 / SquareByRow2Plus row-ordered block history sampling
- DiagPixelated / DiagPixelatedResize diagonal block averaging with fixed or variable tile size
- PixelateBlend / PixelateRect / PixelateWave / MosaicBlend pixelation and mosaic blends
- BlockPixels / BlockScale / BlockXor / BlockStrobe block-level transform series
- BlockyTrails16 / BlockyTrails32 block-sized history trails
- BlockSwap swaps pixel blocks between current and history

Wave / Oscillation Effects

Use sine, cosine, square, triangle, sawtooth, and pulse waveforms to drive pixel displacement or color modulation.

- SineWaveDistort / CosineWaveDistort horizontal/vertical sinusoidal pixel shifting
- SineWaveBlend / CosineWaveBlend / SinCosBlend wave-modulated blends
- SquareWave / SquareWaveX / SquareWaveBlend square wave color/blend modulation
- TriangleWave / TriangleWaveBlend triangle wave modulation
- SawtoothWave / SawtoothWaveBlend sawtooth wave modulation
- PulseWave / PulseWaveBlend / PulseRadial / Pulse pulse-shape modulation
- StepWave / StepWaveBlend step-function wave modulation
- WaveBlend / WaveBlendX2 / WavePattern / WavePatternXor generic wave blend series
- MirrorWave / SpiralWave / VortexEffect / TwistEffect complex wave-geometry hybrids

Mirror / Reflection

Symmetric transforms that fold, flip, or reflect pixel coordinates.

- MirrorReverseColor four-point mirror average (top-left, bottom-left, bottom-right, current) with channel reversal
- AlphaBlendMirror / MirrorXorAlpha mirror with alpha and XOR compositing
- IntertwinedMirror interleaved mirror blend
- MirrorReverseColorBlend blended version of MirrorReverseColor
- FlipAlphaBlend / RandomFlipFilter / FlipPictureShift / FlipMirror flip transform series
- DiagMirror / ShadowMirror / GhostMirror / FacetMirror diagonal and ghost mirror types
- Kaleidoscope / KaleidoscopeBlend / KaleidoBlend / KaleidoScope4 / MirrorKaleid kaleidoscope family
- SplitMirror / TripleSplit / PrismSplit split-panel mirror effects

Pixel Read / Strobe / Blend Collections

Effects that do frame-history collection reads, random collection sampling, or strobe-pattern switching.

- StretchColMatrix8/16/32 samples history at (x / sw) % numFrames column stride
- ColorCollectionSubtleStrobe / CollectionRandom / CollectionAlphaXor collection random/strobe series
- ColorCollection64X / ColorCollectionSwitch / ColorCollectionRGB_Index indexed collection blends

- ColorCollectionGhostTrails / ColorCollectionScale / ColorCollectionReverseStrobe ghost and scale variants
- BlendWithSource25/50/75/100 fixed-ratio blends with history source
- BlendFor360 / BlendForward16 / BlendForward32 / BlendFromXtoY direction-controlled blend series
- FadeInAndOut / FadeBlendXor / FadeBars fade pattern series
- MildStrobe / BrightStrobe / DarkStrobe / StrobeEffect / StrobeXor strobe type family

Advanced / Cinematic Effects

Higher-complexity effects that combine multiple techniques or simulate specific visual phenomena.

- CRTCurvature simulates CRT screen barrel distortion
- FilmGrain per-pixel noise scaled by luma to simulate analog grain
- ChromaticAberration / ChromaticAberrationX RGB channel lateral displacement simulating lens chromatic error
- LensFlare / LightLeak / GlowTrails / NeonGlow / GlowEdge / GlowPulse optical glow/flare family
- NightVision / InfraredView / ThermalBlend synthetic imaging modalities
- MatrixCode / DigitalRain column-drip pattern simulating cascading character streams
- WaterColor / OilSlick / LiquidMetal / LavaLamp fluid/material simulation-inspired effects
- HeatDistort / HeatRipple / HeatWave heat shimmer displacement series
- SobelNorm / DetectEdges / SketchOutline / SobelGlow / ElectricEdge edge detection and outline family
- GalaxySpiral / CosmicDust / StarBurst / Fireworks space/particle aesthetic effects

The Two-Stage GPU Pipeline CUDA Compute + GLSL Shaders

The systems visual power comes from combining two distinct GPU programming models into a single per-frame pipeline. Understanding how each stage works and how they connect explains why the project can produce such a large array of effects.

Stage 1: CUDA Compute Kernels Massively Parallel Pixel Processing

NVIDIA GPUs contain thousands of streaming multiprocessors (SMs), each capable of running many threads in parallel. Both CUDA kernels and GLSL shaders execute on this same hardware the difference is the programming model. acidcam-gpu uses the CUDA compute model to assign one thread to each pixel of the video frame. For a 1920x1080 frame, the system launches 2,088,960 threads simultaneously, organized into 1616 blocks of 256 threads each (8,160 blocks total). Every pixel in the frame is processed in parallel there is no sequential per-pixel loop on the CPU.

Each thread runs the unified dispatch kernel (unifiedFilterKernel), which loops through the users selected filter list and applies each filter to that threads pixel in order. The kernel contains a 905-case switch statement each case calls a __device__ function that reads and writes the pixels RGBA values at data[y * step + x * 4]. Because every thread in a warp processes the same filter index at the same time (the filter list is shared, not per-pixel), the dispatch is warp-coherent and runs at full GPU throughput.

Crucially, the CUDA filters have access to a ring buffer of historical frames stored entirely in GPU device memory (DynamicFrameBuffer). This means filters like MedianBlend can average across all prior frames, AuraTrails can blend against frames at specific history indices, and MatrixOutline can compare against a frame from several steps back all without any data leaving the GPU. This kind of arbitrary random-access memory read across multiple frame buffers is what makes the CUDA compute model essential GLSL fragment shaders run on the same GPU cores but operate within the graphics pipeline, which doesnt support this kind of free-form device memory access.

Stage 2: GLSL Shaders Full-Frame Post-Processing

After CUDA finishes processing every pixel, the result must be displayed. Rather than downloading the frame to the CPU and re-uploading it, the system uses CUDA/OpenGL interop via a Pixel Buffer Object (PBO):

1. The PBO is registered with CUDA using `cudaGraphicsGLRegisterBuffer`
2. CUDA maps the PBO into its address space and performs a `cudaMemcpy2D` (device-to-device) from the working buffer into the PBO **the frame never touches CPU RAM**
3. OpenGL binds the PBO as a pixel unpack buffer and calls `glTexSubImage2D` to populate a texture another zero-copy GPU operation

The CUDA-filtered frame is now an OpenGL texture. At this point, GLSL fragment shaders take over. These are standard OpenGL shader programs (GLSL 330 core) that process the entire frame as a texture. The shader library provides Shadertoy-compatible uniforms `iTime`, `iResolution`, `iMouse`, `iFrame`, `iTimeDelta`, `iDate`, `iFrameRate`, `iSampleRate`, plus audio-reactive uniforms `amp/uamp` enabling a wide range of full-frame effects like color grading, distortion warps, glow effects, CRT simulation, and procedural pattern overlays.

Shaders can render the texture onto a fullscreen 2D quad (standard image processing) or onto 3D model geometry loaded from MXMOD files, projecting the filtered video onto a rotating or animated mesh.

The Transfer: Zero-Copy GPU Interop

The bridge between CUDA and OpenGL is critical to performance. The `TextureUploader` class handles this via PBO interop the CUDA working buffer is copied directly into an OpenGL PBO in device memory, then uploaded to a GL texture, all without a CPU round-trip. This means the entire pipeline from raw camera frame to final displayed output can stay on the GPU throughout both the CUDA and GLSL stages.

Filter and Shader Stacking Two Independent Composable Chains

The system provides two independent stacking mechanisms that compose together: a CUDA filter chain and a GLSL shader pass stack. Users can configure both, and every combination produces a different visual result.

CUDA Filter Chain (Stacking Inside the Kernel)

Users select any subset of the 905 available CUDA filters and arrange them in a specific order. This ordered list is uploaded to GPU memory as an array of `GPUFilter` structs. Inside the kernel, every pixel thread loops through this array:

```
for (int i = 0; i < count; ++i) {
    switch (filters[i].index) {
        case 0: processSelfAlphaBlend(x, y, data, step, params); break;
        case 1: processMedianBlend(x, y, data, allFrames, step, params); break;
        // ... 902 more cases ...
    }
}
```

Each filter modifies the pixel buffer in-place. Filter N reads the output that filter N-1 wrote. This means the filters chain naturally the output of one becomes the input of the next, all within a single kernel launch. There are no intermediate buffer copies between filters; the stacking is purely sequential modification of the same pixel data.

Rebuilding the filter list is lazy and zero-cost at steady state: the device-side filter array is only reallocated when the chain actually changes (tracked by a changed flag). Re-ordering filters at runtime is essentially free.

GLSL Shader Pass Stack (Ping-Pong Framebuffer Passes)

After CUDA processing, GLSL shaders can be stacked using a multi-pass ping-pong framebuffer technique. The user selects and orders GLSL shader passes through the `ShaderPassDialog` interface. Each pass works as follows:

1. The system maintains two offscreen framebuffers (passFBO[0] and passFBO[1]) with corresponding textures (passTexture[0] and passTexture[1])
2. The input starts as the camera texture (which already contains the CUDA-filtered output)
3. For each shader in the pass list: - Bind passFBO[pingpong] as the render target - Activate the pass shader program and set its uniforms (time, resolution, etc.) - Draw a fullscreen quad sampling from inputTex - Set inputTex = passTexture[pingpong] (the output becomes the next pass's input) - Flip pingpong = 1 - pingpong
4. The final texture becomes the input for the main render (2D sprite or 3D mesh)

This allows stacking any number of GLSL shader effects each one processes the output of the previous, building up complexity layer by layer.

Combined Stacking: The Full Pipeline

Both chains compose into a single per-frame pipeline:

```

Camera Frame
  [CUDA Filter 1] [CUDA Filter 2] ... [CUDA Filter N]   (in-kernel chain)
  PBO interop (zero-copy GPU transfer)
  [GLSL Shader Pass 1] [GLSL Shader Pass 2] ... [GLSL Pass M]   (ping-pong FBOs)
  Final composite (2D quad or 3D mesh)
  Screen / Recording

```

The user independently controls both the CUDA filter list and the GLSL shader pass list. Changing either one or just reordering elements produces a completely different visual result.

Why Stacking Produces Infinite Visuals

Each filter and shader is a transformation function. Ordering is not commutative in general, so:

```
F3(F2(F1(frame))) != F1(F2(F3(frame)))
```

If you choose n distinct filters and order them, permutations are $n!$. Even before parameter changes, feedback buffers, and time-varying uniforms, this grows explosively.

- 5 filters 120 orderings
- 8 filters 40,320 orderings
- 10 filters 3,628,800 orderings

With 905 CUDA filters to choose from, plus an independent library of GLSL shaders that also stack and reorder, the combinatorial space multiplies further. Add in the continuously evolving per-frame parameters (alpha oscillation, frame history index, square size, random seeds, frame count) and the visual output becomes effectively unbounded the same filter chain produces different results every frame because the parameters driving it are always changing.

In-Depth: Why Order Dominates Output

In `unifiedFilterKernel`, each thread applies filter case statements in strict list order. Many cases are stateful across time because they read `allFrames` (historical frame pointers) and per-frame randomized params generated in `launch_filter`.

- **Non-commutative transforms:** geometric shift then color XOR yields different values than color XOR then geometric shift, because pixel neighborhoods sampled differ.
- **Temporal dependency:** trail/motion filters sample prior frames; changing order changes which transformed history is fed forward.
- **Parameter evolution:** alpha, square size, random seed, and thresholds evolve frame-to-frame, so visual output is a time series, not a static map.

- **Resolution coupling:** many filters compute indices with width/height and step; same chain on different resolutions creates different artifact structure.
- **Cross-domain stacking:** CUDA filter chains and GLSL shader pass stacks compose independently, so the total combinatorial growth is the product of both far larger than either alone. A 5-filter CUDA chain a 3-shader GLSL stack already produces $120 \cdot 6 = 720$ distinct orderings before any parameter variation.

Build System, Dependencies, and Distribution

The project requires a very specific dependency stack not all of it is available pre-built from most package managers. OpenCV must be compiled from source with CUDA support enabled, which is the most significant build-time requirement.

Required Dependencies

- **NVIDIA GPU:** RTX 20-series or newer. The project is developed and optimized on an RTX 2070. Older hardware may work but is not guaranteed.
- **NVIDIA Proprietary Drivers:** v535 or newer. The CUDA runtime links against driver libraries at version-specific interfaces.
- **CUDA Toolkit 12.x:** provides `nvcc`, `cuda_runtime.h`, `cudaMalloc/cudaMemcpy/cudaDeviceSynchronize`, and device code compilation.
- **OpenCV with CUDA support:** must be compiled from source with `WITH_CUDA=ON` and matching CUDA architecture flags. Provides `cv::cuda::GpuMat`, `cv::cuda::cvtColor`, `cv::cuda::printShortCudaDeviceInfo` $\hookrightarrow$, and the `cv::VideoCapture` backend used in the CLI.
- **libmx2 / MXWrite:** the MX2 ecosystem library provides MXMOD 3D model parsing and the MXWrite encoder wrapper around FFmpeg. Must be built and installed before building `acidcam-gpu`.
- **Qt6:** required only for the ACMX2 Qt interface, not for the library or CLI.
- **C++20 compiler:** the project uses `std::format`, `std::filesystem`, and other C++20 features throughout `main_cv.cu`.

Build Order

1. Build and install `libmx2/libmx` (with OpenGL support): `cmake .. -DEXAMPLES=OFF -DOPENGL=ON && \hookrightarrow make -j$(nproc) && sudo make install`
2. Build and install `acidcam-gpu/MXWrite`: `cmake .. && make -j$(nproc) && sudo make install`
3. Build and install the `acidcam-gpu` library and CLI: `cmake .. && make -j$(nproc) && sudo make \hookrightarrow install`. This installs the shared library, headers, and CMake package config so downstream projects can use `find_package(acidcam-gpu)`.
4. Optionally build and install the Qt ACMX2 interface.

Container Distribution (Podman)

The primary distribution mechanism for end users is a Podman container image: Podman Container file. This automates the build process. To run it:

- Host must have NVIDIA drivers and NVIDIA Container Toolkit for Podman installed.
- `podman build -t acmx2-arch:latest -f Containerfile.arch`
- The run script (`podman/run-acmx2-arch.sh`) passes GPU access, camera device (`/dev/video0`), audio device, and X11 display socket into the container so the app appears on the host desktop with full hardware access.
- This approach means users on Bazzite, Arch, or other NVIDIA-equipped Linux systems can run a fully GPU-accelerated visual effects tool without any build steps.

Development Environment

The project is developed on Bazzite Linux using Arch Linux containers via Distrobox. This means the host OS is an immutable Fedora-based image, and all development tooling (CUDA, OpenCV from AUR, Qt6, gcc) is installed inside an Arch container mounted into the same home directory. This

pattern keeps the base OS clean while maintaining full package manager access for development dependencies.

Source Code Map by Project Part

ACMX2 Core

- ACMX2/CMakeLists.txt dependency checks + executable wiring
- ACMX2/acmx.cpp capture, render, controls, runtime loop
- ACMX2/program.cpp/.hpp shader program loading + binary cache
- ACMX2/data/*.glsl base passthrough/framebuffer shaders

ACMX2 Media + Audio + 3D

- ACMX2/audio.cpp/.hpp RtAudio amplitude/reactive hooks
- ACMX2/audio_transfer.cpp audio transfer helper
- ACMX2/models/*.mxmod MXMOD geometry library
- ACMX2/examples/*.glsl sample shader library/index

ACMX2 Interface + Tools

- ACMX2/interface/* Qt launcher/editor/dialog workflow
- ACMX2/shader_generator/* AI-assisted shader generation utility
- ACMX2/shader.packs/ shader pack metadata
- ACMX2/macros/ platform-specific notes

acidcam-gpu Library

- acidcam-gpu/CMakeLists.txt CUDA/OpenCV/MXWrite package build
- acidcam-gpu/include/ac-gpu/ac-gpu.hpp API structs + launch contract
- acidcam-gpu/src/filters.cu filter table + unifiedFilterKernel dispatch
- acidcam-gpu/app/main_cv.cu CLI runtime/argument pipeline

Ops + Deployment

- podman/Containerfile.arch Build containerized runtime image
- podman/run-acmx2-arch.sh GPU/camera/audio passthrough run script
- acidcam-gpu/scripts/* OpenCV CUDA and environment helper scripts

Expanded Meaning of Each Code Map Item

- ACMX2/CMakeLists.txt: hard-gates feature availability at configure time; this file decides whether your build can legally run all advertised runtime paths.
- ACMX2/acmx.cpp: the largest operational core; input parsing, frame lifecycle, cache management, GL/CUDA interop, and output/control behavior converge here.
- ACMX2/program.cpp/.hpp: shader binary cache layer uses source+driver fingerprinting to skip repeated compile/link cycles and reduce startup latency.
- ACMX2/data/*.glsl: baseline vertex/fragment units that form base render stages and buffer transfer steps.
- ACMX2/audio.cpp/.hpp: real-time amplitude extraction from audio input; callback computes average magnitude used for reactive controls.
- ACMX2/audio_transfer.cpp: utility path for audio transfer/record style workflows and supporting media synchronization tasks.
- ACMX2/models/*.mxmod: model assets for 3D or scene-influenced visuals integrated with runtime rendering.
- ACMX2/examples/*.glsl: curated shader samples used as practical templates and quick-start visual blocks.
- ACMX2/interface/*: user-facing orchestration UI for session setup, process launch, list reordering, and settings persistence.

- `ACMX2/shader_generator/*`: assistant tooling that helps generate shader ideas/workflows while maintaining runtime-compatible output.
- `ACMX2/shader.packs/`: pack metadata and organization layer for large shader collections.
- `acidcam-gpu/include/ac-gpu/ac-gpu.hpp`: ABI/API boundary between host app and CUDA engine.
- `acidcam-gpu/src/filters.cu`: massive effect implementation + dispatch switch; this file is the core visual transformation engine.
- `acidcam-gpu/app/main_cv.cu`: standalone CLI proving the CUDA pipeline without the full ACMX2 UI stack.
- `podman/Containerfile` and `scripts`: reproducible deployment path for camera/GPU-enabled container runs.

End-to-End Runtime Flow

1. Launch ACMX2 or the Qt interface.
2. Load camera/video + shader set + optional GPU filter list.
3. Frames enter OpenCV/CUDA buffers.
4. Ordered filter list is sent into CUDA dispatch.
5. GPU writes transformed pixels back for GL presentation.
6. Optional recording/snapshot/audio-reactive controls are applied.

The key design principle is: **order is the language**. Re-ordering the same operators creates a different visual grammar every time.

Detailed Runtime Inner Workings

1. **Configuration phase**: CLI args or Qt controls define input source, CUDA filters, shader pass order, buffer depth, fps/recording, and optional audio mode.
2. **Device/stream validation**: runtime checks CUDA device count, camera/video availability, and opens required streams; invalid resources fail early with explicit messages.
3. **Frame ingress**: each frame enters as OpenCV matrix data; format conversions normalize channel layout for downstream CUDA/OpenGL pipelines.
4. **History update**: dynamic or fixed frame buffers rotate so current + historical frames are available in device memory simultaneously.
5. **Parameter evolution**: animated values (alpha ramps, square size oscillation, frame index direction) are updated once per frame tick.
6. **Filter dispatch packaging**: selected filter IDs are transformed into GPU list structs; if chain changed, device filter list is rebuilt.
7. **CUDA execution**: unified kernel runs one thread per pixel, then loops each selected filter in order, modifying pixel values cumulatively.
8. **Interop transfer**: resulting CUDA buffer is copied into GL-bound PBO memory and uploaded to texture for immediate rendering without host bounce.
9. **Output fan-out**: displayed frame may also be written to file and/or used by snapshot threads; optional audio state can modulate rendering behavior.
10. **Loop continuation**: next frame repeats with updated temporal state, producing evolving visuals that are path-dependent over time.

Project Architecture Goals and Design Tradeoffs

The project balances three competing goals: maximal visual complexity, real-time responsiveness, and a workflow that remains controllable by artists/operators. The architecture is intentionally modular so each subsystem can evolve without collapsing the full stack.

1) Composability First

Shaders, CUDA filters, and temporal buffers are treated as composable operators. This favors creative exploration and makes the system useful for live experimentation, not just fixed presets.

2) Deterministic Runtime Paths

The same ordered chain and parameter set produce reproducible behavior at a given resolution/device configuration, which matters for iterative artistic workflows and debugging.

3) Throughput-Oriented Execution

CUDA/OpenGL interop and device-side frame history reduce host-device copies. The core design prefers sustained frame throughput over heavyweight per-frame orchestration.

4) Operator Visibility

The Qt layer exposes logs, ordered lists, and persistent settings. This turns complex GPU behavior into a controlled session system that can be repeated and tuned.

Engineering Implications

- **Memory vs flexibility:** temporal filters require multiple historical frames in VRAM; this increases memory usage but enables richer motion feedback effects.
- **Unified kernel benefits:** one dispatch model simplifies orchestration and lets filter chains be data-driven rather than compile-time fixed.
- **Cross-platform pressure:** CMake + container scripts improve portability, but GPU/audio/camera stacks still vary by OS and driver quality.
- **UI/runtime separation:** Qt remains a control plane while CLI/CUDA remains an execution plane; this separation keeps debugging cleaner.
- **Creative-state persistence:** cached settings and ordered chains make sessions restorable, which is important for long-form visual composition.

Build, Deployment, and Operational Model

Beyond visuals, the project includes practical operational pathways for repeatable builds and runtime deployment across development workstations and containerized environments.

Build Surface

- **Dependency validation:** CMake scripts act as a gatekeeper so unsupported environments fail during configure/build, not deep in runtime.
- **Library/executable layering:** the CUDA library can be tested through its own CLI path, while ACMX2 and Qt consume it as a higher-level orchestration stack.
- **Shader and model assets:** runtime behavior depends not only on binaries but also on curated GLSL and MXMOD assets shipped with the project.

Operational Surface

- **Container workflows:** Podman files and scripts define a reproducible execution recipe including GPU/camera/audio passthrough where supported.
- **Session-driven usage:** operators can treat configurations as sessions, allowing repeatable live setups for streaming, VJ work, or iterative rendering passes.
- **Failure visibility:** logs from runtime and UI layers help pinpoint whether issues are input-device, shader compile, CUDA dispatch, or encoder related.

Project Summary

acidcam-gpu + ACMX2 is a layered real-time visual computing system built entirely around NVIDIA CUDA. At the bottom: a 13,891-line CUDA file (filters.cu) implementing 905 per-pixel GPU effects dispatched through a single unified kernel. In the middle: a rotating device-side frame history buffer (DynamicFrameBuffer), a per-frame evolving parameter bundle (FilterParams), and a lazy-rebuild filter list mechanism that makes chain reconfiguration zero-cost. At the top: a CLI application with an oscillating AnimationState that continuously evolves alpha, block size, and temporal

frame selection, plus a Qt6 orchestration interface with session persistence, process supervision, and multi-pass shader support. Distribution via Podman containers removes the painful dependency build requirement for end users.

The core creative insight is that filter order is the language. The same 905 effects, ordered differently, with different animated parameters, on different input material, produce an effectively unbounded space of visual outcomes - all running at real-time framerates on a single consumer NVIDIA GPU.

AcidCam GPU - Effects Reference

Complete reference for all CUDA filter kernels and GLSL fragment shaders in ACMX2.

- CUDA Filters: 905
 - GLSL Shaders: 1097
-

Part I - CUDA Filters

This section documents all 905 CUDA filter kernels in filters.cu. Each filter runs as a `__device__` ↪ function called per-pixel from the unified GPU kernel.

Unified Filter Kernel

The unified GPU filter kernel dispatches all filter effects from a single CUDA `__global__` function. Each thread processes one pixel, iterating over the active filter chain and dispatching via a switch on the filter index.

```
__global__ void unifiedFilterKernel(GPUFilter *filters, size_t count, unsigned char *data, unsigned char
    ↪ **allFrames, int width, int height, size_t step, FilterParams params) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= width || y >= height) {
        return;
    }
    for (int i = 0; i < count; ++i) {
        int idx = y * step + x * 4;
        switch (filters[i].index) {
            case 0:
                processSelfAlphaBlend(x, y, data, step, params);
                break;
            case 1:
                processMedianBlend(x, y, data, allFrames, step, params);
                break;
            case 2:
                processMedianBlend(x, y, data, allFrames, step, params);
                break;
            case 3:
                processSquareBlockResize(x, y, data, allFrames, step, params);
                break;
            case 4:
                processSelfScaleRefined(x, y, data, step, params);
                break;
            case 5:
                processStrangeGlitch(x, y, data, allFrames, step, params);
                break;
            case 6:
                processMatrixOutline(x, y, data, allFrames, step, params);
```

```

        break;
    case 7:
        processAuraTrails(x, y, data, allFrames, step, params);
        break;
    case 8: {
        int px = (width - x - 1);
        int py = (height - y - 1);
        int i0 = py * step + x * 4;
        int i1 = py * step + px * 4;
        int i2 = y * step + px * 4;
        for (int j = 0; j < 3; j++) {
            data[idx + (2 - j)] = (unsigned char)fminf(255.0f, (data[idx + j] * 0.25f + data[i0 + j]
↪ * 0.25f + data[i1 + j] * 0.25f + data[i2 + j] * 0.25f));
        }
    } break;
    case 9:
        processSquareShrink(x, y, data, allFrames, width, height, step, params);
        break;
    case 10: {
        unsigned char *ghost = allFrames[0];
        if (ghost) {
            for (int j = 0; j < 3; j++) {
                data[idx + j] = (unsigned char)fminf(255.0f, (data[idx + j] * (1 - params.alpha) +
↪ ghost[idx + j] * params.alpha));
            }
        }
    } break;
    case 11:
    case 12:
    case 13: {
        int sw = (filters[i].index == 11) ? 8 : (filters[i].index == 12 ? 16 : 32);
        unsigned char *ref = allFrames[(x / sw) % params.numFrames];
        if (ref) {
            for (int j = 0; j < 3; j++) {
                data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
            }
        }
    } break;
    case 14: {
        float a = (0.5f / height) * y;
        for (int j = 0; j < 3; j++) {
            data[idx + j] = (unsigned char)fminf(255.0f, (0.5f * data[idx + j]) + (a * params.
↪ float_param1));
        }
    } break;
    case 15:
    case 16: {
        bool hor = (filters[i].index == 15);
        unsigned char *ref = allFrames[(hor ? y : x) % params.numFrames];
        if (ref) {
            for (int j = 0; j < 3; j++) {
                data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
            }
        }
    } break;
    case 17:
        processWaveTrails(x, y, data, allFrames, step, params);
        break;
    case 18: {
        unsigned char *ref = allFrames[(y * width + x + params.seed) % params.numFrames];
        if (ref) {
            for (int j = 0; j < 3; j++) {

```



```

        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
    }
}
} break;
case 19: {
    processAuraTrails(x, y, data, allFrames, step, params);
    processWaveTrails(x, y, data, allFrames, step, params);
} break;
case 20:
    processParticleSlide(x, y, data, allFrames, width, height, step, params);
    break;
case 21: {
    int sw = 16;
    int sh = 16;
    int sx = (x / sw) * sw;
    int sy = (y / sh) * sh;
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[sy * step + sx * 4 +
↪ j]);
    }
} break;
case 22: {
    int sw = params.sw;
    int sh = params.sh;
    int sx = (x / sw) * sw;
    int sy = (y / sh) * sh;
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[sy * step + sx * 4 +
↪ j]);
    }
} break;
case 23:
    processRGBShiftTrails(x, y, data, allFrames, step, params);
    break;
case 24: {
    int ny = (y + params.int_param1) % height;
    if (ny < 0) {
        ny += height;
    }
    for (int j = 0; j < 3; j++) {
        data[idx + j] = data[ny * step + x * 4 + j];
    }
} break;
case 25: {
    int nx = (x + params.int_param1) % width;
    if (nx < 0) {
        nx += width;
    }
    for (int j = 0; j < 3; j++) {
        data[idx + j] = data[y * step + nx * 4 + j];
    }
} break;
case 26: {
    int nx = (x + params.int_param2) % width;
    int ny = (y + params.int_param1) % height;
    if (nx < 0) {
        nx += width;
    }
    if (ny < 0) {
        ny += height;
    }
    for (int j = 0; j < 3; j++) {

```

```

        data[idx + j] = data[ny * step + nx * 4 + j];
    }
} break;
case 27: {
    int nx = (x + params.int_param1) % width;
    if (nx < 0) {
        nx += width;
    }
    data[idx + 2] = data[y * step + nx * 4 + 2];
} break;
case 28: {
    int nx = (x + params.int_param1) % width;
    if (nx < 0) {
        nx += width;
    }
    data[idx + 1] = data[y * step + nx * 4 + 1];
} break;
case 29: {
    int nx = (x + params.int_param1) % width;
    if (nx < 0) {
        nx += width;
    }
    data[idx] = data[y * step + nx * 4];
} break;
case 30: {
    int ny = (y + params.int_param1) % height;
    if (ny < 0) {
        ny += height;
    }
    data[idx + 2] = data[ny * step + x * 4 + 2];
} break;
case 31: {
    int ny = (y + params.int_param1) % height;
    if (ny < 0) {
        ny += height;
    }
    data[idx + 1] = data[ny * step + x * 4 + 1];
} break;
case 32: {
    int ny = (y + params.int_param1) % height;
    if (ny < 0) {
        ny += height;
    }
    data[idx] = data[ny * step + x * 4];
} break;
case 33: {
    int nx = (x + ((params.int_param1 % 2 == 1) ? 0 : -16) + (int)(gpu_rand(x, y, params.seed) *
↪ 16.0f)) % width;
    if (nx < 0) {
        nx += width;
    }
    for (int j = 0; j < 3; j++) {
        data[idx + j] = data[y * step + nx * 4 + j];
    }
} break;
case 34: {
    int ny = (y + ((params.int_param1 % 2 == 1) ? 0 : -16) + (int)(gpu_rand(x, y, params.seed) *
↪ 16.0f)) % height;
    if (ny < 0) {
        ny += height;
    }
    for (int j = 0; j < 3; j++) {

```

```

        data[idx + j] = data[ny * step + x * 4 + j];
    }
} break;
case 35:
    processTripHSV(x, y, data, step, params.frame_count);
    break;
case 36: {
    data[idx] ^= (unsigned char)params.sumB;
    data[idx + 1] ^= (unsigned char)params.sumG;
    data[idx + 2] ^= (unsigned char)params.sumR;
} break;
case 37:
    processSobel(x, y, data, width, height, step, false, params.threshold);
    break;
case 38:
    processSobel(x, y, data, width, height, step, true, params.threshold);
    break;
case 39:
    processSobel(x, y, data, width, height, step, false, params.threshold);
    break;
case 40: {
    int off = params.int_param1 % 100;
    int sx = (x / (off + 1)) * (off + 1);
    unsigned char *r = allFrames[params.frame_count % params.numFrames];
    if (r) {
        for (int j = 0; j < 3; j++) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + r[y * step + sx * 4 + j] *
↪ 0.5f);
        }
    }
} break;
case 41:
    processLineInLineOut4_Increase(x, y, data, allFrames, width, height, step, params);
    break;
case 42:
    processLineInLineOut_ReverseIncrease(x, y, data, allFrames, width, height, step, params);
    break;
case 43:
    processLineInLineOut_InvertedY(x, y, data, allFrames, width, height, step, params);
    break;
case 44: {
    int idx = y * step + x * 4;
    int offset = get_osc_offset(params.frame_count, 400);
    int zpos = height - y - 1;
    int rpos = width - x - 1;
    unsigned char *hist = allFrames[params.frame_count % params.numFrames];
    if (hist && zpos >= 0 && rpos >= 0) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[zpos * step +
↪ rpos * 4 + j]));
    }
} break;
case 45:
    processLineInLineOut_Vertical(x, y, data, allFrames, width, height, step, params);
    break;
case 46: {
    int idx = y * step + x * 4;
    int offset = get_osc_offset(params.frame_count, 400);
    int src_y = (y + offset) % height;
    unsigned char *hist = allFrames[params.frame_count % params.numFrames];
    if (hist) {
        int src_idx = src_y * step + x * 4;

```

```

        for (int j = 0; j < 3; ++j)
            data[src_idx + j] = (unsigned char)((0.5f * data[src_idx + j]) + (0.5f * hist[idx +
↪ j]));
    }
} break;
case 47:
    processLineInLineOut_IncreaseImage(x, y, data, allFrames, width, height, step, params);
    break;
case 48:
    processSquareByRow(x, y, data, allFrames, width, height, step, params, false, false);
    break;
case 49:
    processSquareByRow(x, y, data, allFrames, width, height, step, params, true, false);
    break;
case 50:
    processSquareByRow(x, y, data, allFrames, width, height, step, params, false, true);
    break;
case 51:
    processDivideByValue(x, y, data, allFrames, step, params);
    break;
case 52:
    processColorCollectionSubtleStrobe(x, y, data, allFrames, step, params);
    break;
case 53:
    processCollectionRandom(x, y, data, allFrames, step, params);
    break;
case 54:
    processCollectionAlphaXor(x, y, data, allFrames, step, params);
    break;
case 55:
    processColorCollection64X(x, y, data, allFrames, step, params);
    break;
case 56:
    processColorCollectionSwitch(x, y, data, allFrames, step, params);
    break;
case 57:
    processColorCollectionRGB_Index(x, y, data, allFrames, step, params);
    break;
case 58:
    processColorCollectionGhostTrails(x, y, data, allFrames, step, params);
    break;
case 59:
    processColorCollectionScale(x, y, data, allFrames, step, params);
    break;
case 60:
    processColorCollectionReverseStrobe(x, y, data, allFrames, step, params);
    break;
case 61:
    processColorCollectionXorPixel(x, y, data, allFrames, step, params);
    break;
case 62:
    processBlendWithSource25(x, y, data, allFrames, step, params);
    break;
case 63:
    processBlendWithSource50(x, y, data, allFrames, step, params);
    break;
case 64:
    processBlendWithSource75(x, y, data, allFrames, step, params);
    break;
case 65:
    processBlendWithSource100(x, y, data, allFrames, step, params);
    break;

```

```

case 66:
    processColorCollectionXorOffsetFlash(x, y, data, allFrames, step, params);
    break;
case 67:
    processColorCollectionMatrixGhost(x, y, data, allFrames, step, params);
    break;
case 68:
    processMildStrobe(x, y, data, allFrames, step, params);
    break;
case 69:
    processReduceBy50(x, y, data, step);
    break;
case 70:
    processColorPositionAverageXor(x, y, data, step, params);
    break;
case 71:
    processColorPositionXor(x, y, data, allFrames, step, params);
    break;
case 72:
    processGrayStrobe(x, y, data, step, params);
    break;
case 73:
    processColorStrobeXor(x, y, data, allFrames, step, params);
    break;
case 74:
    processColorGhost(x, y, data, allFrames, step, params);
    break;
case 75:
    processBlurredOutXor(x, y, data, allFrames, step, params);
    break;
case 76:
    processDizzyFilter(x, y, data, allFrames, step, params);
    break;
case 77:
    processBuzzed(x, y, data, allFrames, step, params);
    break;
case 78:
    processBuzzedDark(x, y, data, allFrames, step, params);
    break;
case 79:
    processAllRed(x, y, data, step);
    break;
case 80:
    processAllGreen(x, y, data, step);
    break;
case 81:
    processAllBlue(x, y, data, step);
    break;
case 82:
    processNegativeStrobe(x, y, data, step, params);
    break;
case 83:
    processXorAddMul(x, y, data, step, params);
    break;
case 84:
    processHorizontalLines(x, y, data, step, params);
    break;
case 85:
    processStrobeRedGreenBlue(x, y, data, step, params);
    break;
case 86:
    processPulse(x, y, data, step, params);

```

```

        break;
case 87:
    processDiamondPattern(x, y, data, step, params);
    break;
case 88:
    processBitwiseXOR(x, y, data, allFrames, step, params);
    break;
case 89:
    processBitwiseAND(x, y, data, allFrames, step, params);
    break;
case 90:
    processBitwiseOR(x, y, data, allFrames, step, params);
    break;
case 91:
    processBlendSwitch(x, y, data, step, params);
    break;
case 92:
    processLineRGB(x, y, data, step);
    break;
case 93:
    processPixelRGB(x, y, data, width, step);
    break;
case 94:
    processInvertedScanlines(x, y, data, step, params);
    break;
case 95:
    processScanSwitch(x, y, data, step, params);
    break;
case 96:
    processScanAlphaSwitch(x, y, data, step, params);
    break;
case 97:
    processRGBFlash(x, y, data, step, params);
    break;
case 98:
    processDiagonalLines(x, y, data, width, height, step, params);
    break;
case 99:
    processDarken(x, y, data, step);
    break;
case 100:
    processSelfXorBlend(x, y, data, step, params);
    break;
case 101:
    processSelfXorDoubleFlash(x, y, data, step, params);
    break;
case 102:
    processSelfOrDoubleFlash(x, y, data, step, params);
    break;
case 103:
    processBlendRowCurvedSqrt(x, y, data, step, params);
    break;
case 104:
    processXorAlpha(x, y, data, step, params);
    break;
case 105:
    processRandomXorBlend(x, y, data, step, params);
    break;
case 106:
    processAndStrobe(x, y, data, step, params);
    break;
case 107:

```

```

        processAndStrobeScale(x, y, data, step, params);
        break;
case 108:
    processAndPixelStrobe(x, y, data, step, params);
    break;
case 109:
    processAndOrXorStrobe(x, y, data, step, params);
    break;
case 110:
    processFadeInAndOut(x, y, data, step, params);
    break;
case 111:
    processBrightStrobe(x, y, data, step, params);
    break;
case 112:
    processDarkStrobe(x, y, data, step, params);
    break;
case 113:
    processRandomXorOpposite(x, y, data, step, params);
    break;
case 114:
    processGradientRainbow(x, y, data, step, params);
    break;
case 115:
    processCossinMultiply(x, y, data, step, params);
    break;
case 116:
    processColorAccumulate1(x, y, data, step, params);
    break;
case 117:
    processColorAccumulate2(x, y, data, step, params);
    break;
case 118:
    processWeakBlend(x, y, data, step, params);
    break;
case 119:
    processStrobeEffect(x, y, data, step, params);
    break;
case 120:
    processBlend3(x, y, data, step, params);
    break;
case 121:
    processNegParadox(x, y, data, step, params);
    break;
case 122:
    processThoughtMode(x, y, data, step, params);
    break;
case 123:
    processTri(x, y, data, step, params);
    break;
case 124:
    processDistort(x, y, data, step, params);
    break;
case 125:
    processColorAccumulate3(x, y, data, step, params);
    break;
case 126:
    processFilter8(x, y, data, step, params);
    break;
case 127:
    processFilter3(x, y, data, step, params);
    break;

```

```

case 128:
    processRainbowBlend(x, y, data, step, params);
    break;
case 129:
    processPixelScale(x, y, data, step, params);
    break;
case 130:
    processGradientSelf(x, y, data, step, params);
    break;
case 131:
    processGradientSelfVertical(x, y, data, step, params);
    break;
case 132:
    processGradientDown(x, y, data, step, params);
    break;
case 133:
    processGraidentHorizontal(x, y, data, step, params);
    break;
case 134:
    processInter(x, y, data, step, params);
    break;
case 135:
    processBlendedScanLines(x, y, data, step, params);
    break;
case 136:
    processGradientStripes(x, y, data, step, params);
    break;
case 137:
    processXorSine(x, y, data, step, params);
    break;
case 138:
    processCircular(x, y, data, step, params);
    break;
case 139:
    processRandomPixels(x, y, data, step, params);
    break;
case 140:
    processDarkRandomPixels(x, y, data, step, params);
    break;
case 141:
    processBars(x, y, data, step, params);
    break;
case 142:
    processNegativeByRow(x, y, data, step);
    break;
case 143:
    processXorScale(x, y, data, step, params);
    break;
case 144:
    processSelfAlphaRGB(x, y, data, step, params);
    break;
case 145:
    processBitwiseXorStrobe(x, y, data, step, params);
    break;
case 146:
    processOrStrobe(x, y, data, allFrames, step, params);
    break;
case 147:
    processDivideAndIncH(x, y, data, width, height, step, params);
    break;
case 148:
    processDivideAndIncW(x, y, data, width, height, step, params);

```



```

        break;
case 149:
    processRandomIncrease(x, y, data, step, params);
    break;
case 150:
    processSelfAlphaScaleBlend(x, y, data, step, params);
    break;
case 151:
    processFadeBars(x, y, data, width, step, params);
    break;
case 152:
    processStrobeXor(x, y, data, step, params);
    break;
case 153:
    processBlank(x, y, data, step, params);
    break;
case 154:
    processColorVariableBlend(x, y, data, step, params);
    break;
case 155:
    processColorXorBlend(x, y, data, step, params);
    break;
case 156:
    processColorAddBlend(x, y, data, step, params);
    break;
case 157:
    processSurroundingPixels(x, y, data, width, height, step, params);
    break;
case 158:
    processSurroundingPixelsAlpha(x, y, data, width, height, step, params);
    break;
case 159:
    processDarkModBlend(x, y, data, step, params);
    break;
case 160:
    processIncreaseDecreaseGamma(x, y, data, step, params);
    break;
case 161:
    processBlendChannelXor(x, y, data, step, params);
    break;
case 162:
    processIncDifference(x, y, data, allFrames, step, params);
    break;
case 163:
    processIncDifferenceAlpha(x, y, data, allFrames, step, params);
    break;
case 164:
    processMirrorXorAlpha(x, y, data, width, height, step, params);
    break;
case 165:
    processIntertwinedMirror(x, y, data, width, height, step, params);
    break;
case 166:
    processColorFadeFilter(x, y, data, step, params);
    break;
case 167:
    processColorChannelMoveUpAndDown(x, y, data, step, params);
    break;
case 168:
    processMedianStrobe(x, y, data, allFrames, step, params);
    break;
case 169:

```

```

        processRGBBlend(x, y, data, step, params);
        break;
    case 170:
        processBGRBlend(x, y, data, step, params);
        break;
    case 171:
        processFlipAlphaBlend(x, y, data, width, height, step, params);
        break;
    case 172:
        processRandomFlipFilter(x, y, data, width, height, step, params);
        break;
    case 173:
        processSelfScaleByFrame(x, y, data, step, params);
        break;
    case 174:
        processAlphaBlendMirror(x, y, data, width, height, step, params);
        break;
    case 175:
        processTwistedVision(x, y, data, width, step, params);
        break;
    case 176:
        processTruncateColor(x, y, data, step, params);
        break;
    case 177:
        processTruncateVariable(x, y, data, step, params);
        break;
    case 178:
        processTruncateVariableScale(x, y, data, step, params);
        break;
    case 179:
        processXorFade(x, y, data, step, params);
        break;
    case 180:
        processSineValue(x, y, data, step, params);
        break;
    case 181:
        processFadeRtoGtoB(x, y, data, step, params);
        break;
    case 182:
        processFadeRandomChannel(x, y, data, step, params);
        break;
    case 183:
        processVariableLines(x, y, data, width, step, params);
        break;
    case 184:
        processVariableLinesVertical(x, y, data, height, step, params);
        break;
    case 185:
        processRowMedianBlend(x, y, data, allFrames, step, params);
        break;
    case 186:
        processMirrorReverseColor(x, y, data, width, height, step, params);
        break;
    case 187:
        processPsychoticVision(x, y, data, step, params);
        break;
    case 188:
        processPixelGlitch(x, y, data, step, params);
        break;
    case 189:
        processStaticGlitch(x, y, data, step, params);
        break;

```

```

case 190:
    processWavePattern(x, y, data, width, step, params);
    break;
case 191:
    processWavePatternXor(x, y, data, step, params);
    break;
case 192:
    processDiagonalXor(x, y, data, step, params);
    break;
case 193:
    processRGBShiftBlend(x, y, data, step, params);
    break;
case 194:
    processChannelShuffle(x, y, data, step, params);
    break;
case 195:
    processChannelShuffleRand(x, y, data, step, params);
    break;
case 196:
    processPixelCounter(x, y, data, step, params);
    break;
case 197:
    processPixelCounterXor(x, y, data, step, params);
    break;
case 198:
    processRowColorBlend(x, y, data, step, params);
    break;
case 199:
    processColumnColorBlend(x, y, data, step, params);
    break;
case 200:
    processCheckerboardXor(x, y, data, step, params);
    break;
case 201:
    processCheckerboardBlend(x, y, data, allFrames, step, params);
    break;
case 202:
    processSineWaveDistort(x, y, data, width, step, params);
    break;
case 203:
    processCosineWaveDistort(x, y, data, height, step, params);
    break;
case 204:
    processSinCosBlend(x, y, data, step, params);
    break;
case 205:
    processPixelReverseXor(x, y, data, width, height, step, params);
    break;
case 206:
    processLinesAcrossX(x, y, data, width, step, params);
    break;
case 207:
    processXorLineX(x, y, data, step, params);
    break;
case 208:
    processAlphaComponentIncrease(x, y, data, step, params);
    break;
case 209:
    processExpandContract(x, y, data, width, step, params);
    break;
case 210:
    processLongLines(x, y, data, step, params);

```

```

        break;
case 211:
    processTearRight(x, y, data, width, step, params);
    break;
case 212:
    processTearDown(x, y, data, height, step, params);
    break;
case 213:
    processDistortionByRow(x, y, data, height, step, params);
    break;
case 214:
    processDistortionByCol(x, y, data, width, step, params);
    break;
case 215:
    processAlternateAlpha(x, y, data, step, params);
    break;
case 216:
    processDiagSquareRGB(x, y, data, allFrames, step, params);
    break;
case 217:
    processShiftPixelsRGB(x, y, data, width, step, params);
    break;
case 218:
    processColorWaveTrailsRGB(x, y, data, allFrames, step, params);
    break;
case 219:
    processProperTrails(x, y, data, allFrames, step, params);
    break;
case 220:
    processXorLag(x, y, data, allFrames, step, params);
    break;
case 221:
    processPixelateBlend(x, y, data, allFrames, step, params);
    break;
case 222:
    processDiagPixel(x, y, data, allFrames, step, params);
    break;
case 223:
    processDiagPixelY(x, y, data, allFrames, step, params);
    break;
case 224:
    processExpandLeftRight(x, y, data, allFrames, width, step, params);
    break;
case 225:
    processDiagSquare(x, y, data, allFrames, step, params);
    break;
case 226:
    processHorizontalColorOffset(x, y, data, width, step, params);
    break;
case 227:
    processPrevFrameNotEqual(x, y, data, allFrames, step, params);
    break;
case 228:
    processBlackLines(x, y, data, width, step, params);
    break;
case 229:
    processDizzyMode(x, y, data, allFrames, step, params);
    break;
case 230:
    processGhostShift(x, y, data, allFrames, step, params);
    break;
case 231:

```

```

        processRGBSplitFilter(x, y, data, allFrames, step, params);
        break;
case 232:
    processPixelateRect(x, y, data, allFrames, step, params);
    break;
case 233:
    processCollectionXor4(x, y, data, allFrames, step, params);
    break;
case 234:
    processRectangleSpin(x, y, data, width, height, step, params);
    break;
case 235:
    processRectanglePlotXY(x, y, data, width, height, step, params);
    break;
case 236:
    processShiftLinesDown(x, y, data, width, height, step, params);
    break;
case 237:
    processPictureStretch(x, y, data, width, height, step, params);
    break;
case 238:
    processPictureStretchPieces(x, y, data, width, height, step, params);
    break;
case 239:
    processVisualSnow(x, y, data, step, params);
    break;
case 240:
    processVisualSnowX2(x, y, data, step, params);
    break;
case 241:
    processLineGlitch(x, y, data, width, step, params);
    break;
case 242:
    processSlitReverse64(x, y, data, width, step, params);
    break;
case 243:
    processSlitReverse64_Increase(x, y, data, width, step, params);
    break;
case 244:
    processSlitStretch(x, y, data, width, step, params);
    break;
case 245:
    processLineLeftRight(x, y, data, width, step, params);
    break;
case 246:
    processLineLeftRightResize(x, y, data, width, step, params);
    break;
case 247:
    processRGBLineTrails(x, y, data, allFrames, step, params);
    break;
case 248:
    processRGBCollectionBlend(x, y, data, allFrames, step, params);
    break;
case 249:
    processRGBCollectionIncrease(x, y, data, allFrames, step, params);
    break;
case 250:
    processRGBLongTrails(x, y, data, allFrames, step, params);
    break;
case 251:
    processFadeRGB_Speed(x, y, data, step, params);
    break;

```

```

case 252:
    processRGBStrobeTrails(x, y, data, allFrames, step, params);
    break;
case 253:
    processBoxGlitch(x, y, data, width, height, step, params);
    break;
case 254:
    processVerticalPictureDistort(x, y, data, height, step, params);
    break;
case 255:
    processShortTrail(x, y, data, allFrames, step, params);
    break;
case 256:
    processDiagInward(x, y, data, width, height, step, params);
    break;
case 257:
    processDiagSquareInward(x, y, data, width, height, step, params);
    break;
case 258:
    processDiagSquareInwardResize(x, y, data, width, height, step, params);
    break;
case 259:
    processPictureShiftDownRight(x, y, data, width, height, step, params);
    break;
case 260:
    processFlipPictureShift(x, y, data, width, height, step, params);
    break;
case 261:
    processRGBWideTrails(x, y, data, allFrames, step, params);
    break;
case 262:
    processLineInLineOut_Increase(x, y, data, width, step, params);
    break;
case 263:
    processLineInLineOut2_Increase(x, y, data, height, step, params);
    break;
case 264:
    processLineInLineOut3_Increase(x, y, data, width, height, step, params);
    break;
case 265:
    processSquareByRow2Plus(x, y, data, width, step, params);
    break;
case 266:
    processFrameSep(x, y, data, allFrames, width, step, params);
    break;
case 267:
    processFrameSep2(x, y, data, allFrames, height, step, params);
    break;
case 268:
    processFrameStopStart(x, y, data, allFrames, step, params);
    break;
case 269:
    processOutOfOrder(x, y, data, allFrames, step, params);
    break;
case 270:
    processTrackingDown(x, y, data, height, step, params);
    break;
case 271:
    processTrackingDownBlend(x, y, data, allFrames, height, step, params);
    break;
case 272:
    processTrackingRev(x, y, data, height, step, params);

```

```

        break;
case 273:
    processTrackingMirror(x, y, data, width, height, step, params);
    break;
case 274:
    processBlockPixels(x, y, data, step, params);
    break;
case 275:
    processFrameChop(x, y, data, allFrames, width, step, params);
    break;
case 276:
    processYLineDown(x, y, data, height, step, params);
    break;
case 277:
    processYLineDownBlend(x, y, data, allFrames, height, step, params);
    break;
case 278:
    processSquareDiff1(x, y, data, allFrames, step, params);
    break;
case 279:
    processLineAcrossXNew(x, y, data, width, step, params);
    break;
case 280:
    processColorGlitch(x, y, data, step, params);
    break;
case 281:
    processPixelShiftUp(x, y, data, height, step, params);
    break;
case 282:
    processPixelShiftDown(x, y, data, height, step, params);
    break;
case 283:
    processPixelShiftLeft(x, y, data, width, step, params);
    break;
case 284:
    processPixelShiftRight(x, y, data, width, step, params);
    break;
case 285:
    processPixelShiftDiagonal(x, y, data, width, height, step, params);
    break;
case 286:
    processWaveBlend(x, y, data, width, step, params);
    break;
case 287:
    processWaveBlendX2(x, y, data, width, height, step, params);
    break;
case 288:
    processSineWaveBlend(x, y, data, step, params);
    break;
case 289:
    processCosineWaveBlend(x, y, data, step, params);
    break;
case 290:
    processSpiralWave(x, y, data, width, height, step, params);
    break;
case 291:
    processRadialBlur(x, y, data, width, height, step, params);
    break;
case 292:
    processZoomBlur(x, y, data, width, height, step, params);
    break;
case 293:

```

```

        processRotateBlend(x, y, data, width, height, step, params);
        break;
case 294:
    processMirrorWave(x, y, data, width, height, step, params);
    break;
case 295:
    processMirrorWaveX(x, y, data, width, step, params);
    break;
case 296:
    processMirrorWaveY(x, y, data, height, step, params);
    break;
case 297:
    processPixelDrift(x, y, data, allFrames, width, height, step, params);
    break;
case 298:
    processPixelDriftX(x, y, data, width, step, params);
    break;
case 299:
    processPixelDriftY(x, y, data, height, step, params);
    break;
case 300:
    processColorPulse(x, y, data, step, params);
    break;
case 301:
    processColorPulseRGB(x, y, data, step, params);
    break;
case 302:
    processColorPulseXor(x, y, data, step, params);
    break;
case 303:
    processGlitchBlock(x, y, data, width, height, step, params);
    break;
case 304:
    processGlitchBlockXor(x, y, data, step, params);
    break;
case 305:
    processGlitchLine(x, y, data, width, step, params);
    break;
case 306:
    processGlitchLineX(x, y, data, height, step, params);
    break;
case 307:
    processNoiseBlend(x, y, data, step, params);
    break;
case 308:
    processNoiseBlendX2(x, y, data, step, params);
    break;
case 309:
    processNoiseXor(x, y, data, step, params);
    break;
case 310:
    processChannelShift(x, y, data, width, step, params);
    break;
case 311:
    processChannelShiftX(x, y, data, height, step, params);
    break;
case 312:
    processChannelRotate(x, y, data, step, params);
    break;
case 313:
    processDiagonalStretch(x, y, data, width, height, step, params);
    break;

```



```

case 314:
    processDiagonalStretchX(x, y, data, width, height, step, params);
    break;
case 315:
    processDiagonalMirror(x, y, data, width, height, step, params);
    break;
case 316:
    processSquareWave(x, y, data, step, params);
    break;
case 317:
    processSquareWaveX(x, y, data, step, params);
    break;
case 318:
    processSquareWaveBlend(x, y, data, allFrames, step, params);
    break;
case 319:
    processTriangleWave(x, y, data, step, params);
    break;
case 320:
    processTriangleWaveBlend(x, y, data, allFrames, step, params);
    break;
case 321:
    processSawtoothWave(x, y, data, step, params);
    break;
case 322:
    processSawtoothWaveBlend(x, y, data, allFrames, step, params);
    break;
case 323:
    processPulseWave(x, y, data, step, params);
    break;
case 324:
    processPulseWaveBlend(x, y, data, allFrames, step, params);
    break;
case 325:
    processStepWave(x, y, data, step, params);
    break;
case 326:
    processStepWaveBlend(x, y, data, allFrames, step, params);
    break;
case 327:
    processRippleEffect(x, y, data, width, height, step, params);
    break;
case 328:
    processRippleEffectX2(x, y, data, width, height, step, params);
    break;
case 329:
    processShockWave(x, y, data, width, height, step, params);
    break;
case 330:
    processShockWaveBlend(x, y, data, allFrames, width, height, step, params);
    break;
case 331:
    processTwistEffect(x, y, data, width, height, step, params);
    break;
case 332:
    processTwistEffectBlend(x, y, data, width, height, step, params);
    break;
case 333:
    processFishEye(x, y, data, width, height, step, params);
    break;
case 334:
    processFishEyeBlend(x, y, data, width, height, step, params);

```

```

        break;
case 335:
    processKaleidoscope(x, y, data, width, height, step, params);
    break;
case 336:
    processKaleidoscopeBlend(x, y, data, width, height, step, params);
    break;
case 337:
    processTunnelEffect(x, y, data, width, height, step, params);
    break;
case 338:
    processTunnelEffectBlend(x, y, data, width, height, step, params);
    break;
case 339:
    processVortexEffect(x, y, data, width, height, step, params);
    break;
case 340:
    processVortexEffectBlend(x, y, data, width, height, step, params);
    break;
case 341:
    processColorDrift(x, y, data, step, params);
    break;
case 342:
    processColorDriftX(x, y, data, step, params);
    break;
case 343:
    processRGBShift(x, y, data, width, step, params);
    break;
case 344:
    processRGBShiftX(x, y, data, height, step, params);
    break;
case 345:
    processChromaticAberration(x, y, data, width, step, params);
    break;
case 346:
    processChromaticAberrationX(x, y, data, width, height, step, params);
    break;
case 347:
    processPosterize(x, y, data, step, params);
    break;
case 348:
    processPosterizeBlend(x, y, data, step, params);
    break;
case 349:
    processSolarize(x, y, data, step, params);
    break;
case 350:
    processSolarizeBlend(x, y, data, step, params);
    break;
case 351:
    processGammaBright(x, y, data, step, params);
    break;
case 352:
    processGammaDark(x, y, data, step, params);
    break;
case 353:
    processContrastBoost(x, y, data, step, params);
    break;
case 354:
    processContrastReduce(x, y, data, step, params);
    break;
case 355:

```

```

        processEdgeGlowBlend(x, y, data, allFrames, width, height, step, params);
        break;
case 356:
    processFrameBlendMulti(x, y, data, allFrames, step, params);
    break;
case 357:
    processFrameBlendMultiX(x, y, data, allFrames, step, params);
    break;
case 358:
    processAcidTrailsBlend(x, y, data, allFrames, step, params);
    break;
case 359:
    processAcidGlitchX(x, y, data, width, height, step, params);
    break;
case 360:
    processAlphaXorBlend(x, y, data, allFrames, step, params);
    break;
case 361:
    processAlphaXorBlendDouble(x, y, data, allFrames, step, params);
    break;
case 362:
    processAndOrXorStrobeScale(x, y, data, step, params);
    break;
case 363:
    processAveragePixelsXorBlend(x, y, data, step, params);
    break;
case 364:
    processBitwiseRotateBlend(x, y, data, step, params);
    break;
case 365:
    processBitwiseRotateDiffBlend(x, y, data, allFrames, step, params);
    break;
case 366:
    processBitwiseXorScaleBlend(x, y, data, step, params);
    break;
case 367:
    processBlackAndWhiteStrobe(x, y, data, step, params);
    break;
case 368:
    processBlendAlphaXorScale(x, y, data, step, params);
    break;
case 369:
    processBlendBuriedXor(x, y, data, allFrames, step, params);
    break;
case 370:
    processBlendCombinedXor(x, y, data, step, params);
    break;
case 371:
    processBlendIncreaseRGB(x, y, data, step, params);
    break;
case 372:
    processBlendThreeXor(x, y, data, allFrames, step, params);
    break;
case 373:
    processBlurDistortionBlend(x, y, data, width, height, step, params);
    break;
case 374:
    processColorAccumulate(x, y, data, step, params);
    break;
case 375:
    processColorAccumulateBlend(x, y, data, allFrames, step, params);
    break;

```

```

case 376:
    processColorAccumulateXor(x, y, data, step, params);
    break;
case 377:
    processColorChannelBlend(x, y, data, step, params);
    break;
case 378:
    processColorChannelXor(x, y, data, step, params);
    break;
case 379:
    processColorCollectionEnergy(x, y, data, allFrames, step, params);
    break;
case 380:
    processColorCollectionWave(x, y, data, step, params);
    break;
case 381:
    processColorFadeXor(x, y, data, step, params);
    break;
case 382:
    processColorIntensityBlend(x, y, data, step, params);
    break;
case 383:
    processColorIntensityXor(x, y, data, step, params);
    break;
case 384:
    processColorMoveBlend(x, y, data, width, height, step, params);
    break;
case 385:
    processColorPixelBlend(x, y, data, step, params);
    break;
case 386:
    processColorPixelXor(x, y, data, step, params);
    break;
case 387:
    processColorScaleBlend(x, y, data, step, params);
    break;
case 388:
    processColorWaveXor(x, y, data, step, params);
    break;
case 389:
    processCosineMultiplyBlend(x, y, data, step, params);
    break;
case 390:
    processDarkModBlendXor(x, y, data, step, params);
    break;
case 391:
    processDifferenceBlend(x, y, data, allFrames, step, params);
    break;
case 392:
    processDifferenceXorBlend(x, y, data, allFrames, step, params);
    break;
case 393:
    processDistortBlend(x, y, data, width, height, step, params);
    break;
case 394:
    processDiamondPatternBlend(x, y, data, width, height, step, params);
    break;
case 395:
    processFadeBlendXor(x, y, data, step, params);
    break;
case 396:
    processFlashBlendXor(x, y, data, step, params);

```

```

        break;
case 397:
    processGhostTrailsBlend(x, y, data, allFrames, step, params);
    break;
case 398:
    processAddInvert(x, y, data, step, params);
    break;
case 399:
    processAlphaBlendSimple(x, y, data, allFrames, step, params);
    break;
case 400:
    processAlphaBlendDoubleX(x, y, data, allFrames, step, params);
    break;
case 401:
    processAlphaStrobeBlendX(x, y, data, allFrames, step, params);
    break;
case 402:
    processBitwiseAndBlend(x, y, data, step, params);
    break;
case 403:
    processBitwiseXorAverage(x, y, data, step, params);
    break;
case 404:
    processBitwiseXorBlendX(x, y, data, allFrames, step, params);
    break;
case 405:
    processBlackStrobe(x, y, data, step, params);
    break;
case 406:
    processBlendAlphaXorX(x, y, data, step, params);
    break;
case 407:
    processBlendCombinedValuesX(x, y, data, step, params);
    break;
case 408:
    processBlendFor360(x, y, data, width, height, step, params);
    break;
case 409:
    processBlendForward16(x, y, data, allFrames, step, params);
    break;
case 410:
    processBlendForward32(x, y, data, allFrames, step, params);
    break;
case 411:
    processBlendFromXtoY(x, y, data, step, params);
    break;
case 412:
    processBlendIncreaseX(x, y, data, step, params);
    break;
case 413:
    processBlendRedGreenBlue(x, y, data, step, params);
    break;
case 414:
    processBlendWithColorX(x, y, data, step, params);
    break;
case 415:
    processBlendAngle(x, y, data, width, height, step, params);
    break;
case 416:
    processBlockScale(x, y, data, width, height, step, params);
    break;
case 417:

```

```

        processBlockStrobe(x, y, data, width, height, step, params);
        break;
case 418:
    processBlockXor(x, y, data, width, height, step, params);
    break;
case 419:
    processBlockyTrails16(x, y, data, allFrames, width, height, step, params);
    break;
case 420:
    processBlockyTrails32(x, y, data, allFrames, width, height, step, params);
    break;
case 421:
    processBlurDistortionX(x, y, data, width, height, step, params);
    break;
case 422:
    processCannyStrobe(x, y, data, width, height, step, params);
    break;
case 423:
    processColorFadeSlow(x, y, data, step, params);
    break;
case 424:
    processColorFibonacci(x, y, data, step, params);
    break;
case 425:
    processCurtainEffect(x, y, data, width, height, step, params);
    break;
case 426:
    processDarkColorFibonacci(x, y, data, step, params);
    break;
case 427:
    processDarkColorsBlend(x, y, data, step, params);
    break;
case 428:
    processEnergizeBlend(x, y, data, allFrames, step, params);
    break;
case 429:
    processAverageLines(x, y, data, width, step, params);
    break;
case 430:
    processAverageLinesBlendX(x, y, data, allFrames, width, step, params);
    break;
case 431:
    processBlendRowAlpha(x, y, data, height, step, params);
    break;
case 432:
    processBlendInOut(x, y, data, allFrames, step, params);
    break;
case 433:
    processColorFlashIncreaseX(x, y, data, step, params);
    break;
case 434:
    processColorIncreaseInOut(x, y, data, step, params);
    break;
case 435:
    processColorLinesX(x, y, data, step, params);
    break;
case 436:
    processColorMoveDownX(x, y, data, height, step, params);
    break;
case 437:
    processColorOrderSwapX(x, y, data, step, params);
    break;

```

```

case 438:
    processColorPulseAlphaX(x, y, data, step, params);
    break;
case 439:
    processColorRowShiftX(x, y, data, width, step, params);
    break;
case 440:
    processColorShiftXorX(x, y, data, step, params);
    break;
case 441:
    processCopyXorAlphaX(x, y, data, allFrames, step, params);
    break;
case 442:
    processCycleShiftRGBX(x, y, data, step, params);
    break;
case 443:
    processDarkNegateX(x, y, data, step, params);
    break;
case 444:
    processDarkSelfAlphaX(x, y, data, step, params);
    break;
case 445:
    processDiagonalGlitch(x, y, data, width, height, step, params);
    break;
case 446:
    processDigitalHaze(x, y, data, step, params);
    break;
case 447:
    processDoubleXorBlend(x, y, data, allFrames, step, params);
    break;
case 448:
    processEchoBlend(x, y, data, allFrames, step, params);
    break;
case 449:
    processElectricEdge(x, y, data, width, height, step, params);
    break;
case 450:
    processFlashColorStrobe(x, y, data, step, params);
    break;
case 451:
    processFrameDiffXor(x, y, data, allFrames, step, params);
    break;
case 452:
    processGhostMirror(x, y, data, allFrames, width, step, params);
    break;
case 453:
    processGlitchSort(x, y, data, width, step, params);
    break;
case 454:
    processHeatWave(x, y, data, width, height, step, params);
    break;
case 455:
    processInterlaceBlend(x, y, data, allFrames, step, params);
    break;
case 456:
    processInvertStrobe(x, y, data, step, params);
    break;
case 457:
    processKaleidoBlend(x, y, data, width, height, step, params);
    break;
case 458:
    processLightStrobe(x, y, data, step, params);

```

```

        break;
case 459:
    processLineGlitchX(x, y, data, width, step, params);
    break;
case 460:
    processMosaicBlend(x, y, data, width, height, step, params);
    break;
case 461:
    processNegatePulse(x, y, data, step, params);
    break;
case 462:
    processOffsetGhost(x, y, data, allFrames, width, height, step, params);
    break;
case 463:
    processPixelateWave(x, y, data, width, height, step, params);
    break;
case 464:
    processQuantizeBlend(x, y, data, step, params);
    break;
case 465:
    processRandomLines(x, y, data, height, step, params);
    break;
case 466:
    processRippleDisplace(x, y, data, width, height, step, params);
    break;
case 467:
    processRotateShift(x, y, data, width, height, step, params);
    break;
case 468:
    processSaturationGlow(x, y, data, step, params);
    break;
case 469:
    processScaleToCenter(x, y, data, width, height, step, params);
    break;
case 470:
    processShadowMirror(x, y, data, width, height, step, params);
    break;
case 471:
    processShiftChannels(x, y, data, step, params);
    break;
case 472:
    processSliceGlitch(x, y, data, width, height, step, params);
    break;
case 473:
    processSobelGlow(x, y, data, width, height, step, params);
    break;
case 474:
    processSpectralShift(x, y, data, step, params);
    break;
case 475:
    processSpiralTrail(x, y, data, allFrames, width, height, step, params);
    break;
case 476:
    processSquareTrails(x, y, data, allFrames, step, params);
    break;
case 477:
    processStrobeNegate(x, y, data, step, params);
    break;
case 478:
    processThermalBlend(x, y, data, step, params);
    break;
case 479:

```



```

        processTintShift(x, y, data, step, params);
        break;
    case 480:
        processTrailEcho(x, y, data, allFrames, step, params);
        break;
    case 481:
        processTransitionBlend(x, y, data, allFrames, step, params);
        break;
    case 482:
        processTwistWarp(x, y, data, width, height, step, params);
        break;
    case 483:
        processVerticalShift(x, y, data, height, step, params);
        break;
    case 484:
        processVortexBlend(x, y, data, width, height, step, params);
        break;
    case 485:
        processWeavePattern(x, y, data, width, height, step, params);
        break;
    case 486:
        processWhiteBurst(x, y, data, width, height, step, params);
        break;
    case 487:
        processWiggleDisplace(x, y, data, width, height, step, params);
        break;
    case 488:
        processXorPulseX(x, y, data, step, params);
        break;
    case 489:
        processYellowShift(x, y, data, step, params);
        break;
    case 490:
        processZigzagGlitch(x, y, data, width, step, params);
        break;
    case 491:
        processAlphaModulate(x, y, data, step, params);
        break;
    case 492:
        processBlockSwap(x, y, data, width, height, step, params);
        break;
    case 493:
        processColorResonance(x, y, data, step, params);
        break;
    case 494:
        processDepthGlitch(x, y, data, width, height, step, params);
        break;
    case 495:
        processEchoShift(x, y, data, allFrames, step, params);
        break;
    case 496:
        processFractalNoise(x, y, data, step, params);
        break;
    case 497:
        processGradientRotate(x, y, data, width, height, step, params);
        break;
    case 498:
        processHarmonicShift(x, y, data, step, params);
        break;
    case 499:
        processAcidWarp(x, y, data, width, height, step, params);
        break;

```

```

case 500:
    processBlendDiagonal(x, y, data, width, height, step, params);
    break;
case 501:
    processChromaFlash(x, y, data, step, params);
    break;
case 502:
    processCircleWave(x, y, data, width, height, step, params);
    break;
case 503:
    processColorCrush(x, y, data, step, params);
    break;
case 504:
    processCrosshatchBlend(x, y, data, step, params);
    break;
case 505:
    processCyberGlitch(x, y, data, width, step, params);
    break;
case 506:
    processDarkPulse(x, y, data, step, params);
    break;
case 507:
    processDiamondPatternX(x, y, data, step, params);
    break;
case 508:
    processDigitalRain(x, y, data, height, step, params);
    break;
case 509:
    processDisplaceX(x, y, data, width, step, params);
    break;
case 510:
    processDriftBlend(x, y, data, allFrames, width, step, params);
    break;
case 511:
    processEdgePulse(x, y, data, width, height, step, params);
    break;
case 512:
    processFlameEffect(x, y, data, height, step, params);
    break;
case 513:
    processFlickerShift(x, y, data, step, params);
    break;
case 514:
    processGhostLayer(x, y, data, allFrames, step, params);
    break;
case 515:
    processGlitchBlockX(x, y, data, width, height, step, params);
    break;
case 516:
    processGlowPulse(x, y, data, step, params);
    break;
case 517:
    processGridDistort(x, y, data, width, height, step, params);
    break;
case 518:
    processHexPattern(x, y, data, step, params);
    break;
case 519:
    processHueRotate(x, y, data, step, params);
    break;
case 520:
    processInterweaveX(x, y, data, allFrames, step, params);

```

```

        break;
case 521:
    processJitterBlend(x, y, data, width, step, params);
    break;
case 522:
    processKaleidoScope4(x, y, data, width, height, step, params);
    break;
case 523:
    processLaserScan(x, y, data, width, height, step, params);
    break;
case 524:
    processLightLeak(x, y, data, width, height, step, params);
    break;
case 525:
    processMeltDown(x, y, data, height, step, params);
    break;
case 526:
    processMirrorDiag(x, y, data, width, height, step, params);
    break;
case 527:
    processNeonGlow(x, y, data, width, height, step, params);
    break;
case 528:
    processNoiseBlendX(x, y, data, step, params);
    break;
case 529:
    processPixelDrift(x, y, data, width, step, params);
    break;
case 530:
    processPlasmaWave(x, y, data, step, params);
    break;
case 531:
    processPrismSplit(x, y, data, width, step, params);
    break;
case 532:
    processPulseRadial(x, y, data, width, height, step, params);
    break;
case 533:
    processRainbowStrobe(x, y, data, step, params);
    break;
case 534:
    processRefractionX(x, y, data, width, step, params);
    break;
case 535:
    processScanlineX(x, y, data, step, params);
    break;
case 536:
    processShatterEffect(x, y, data, width, height, step, params);
    break;
case 537:
    processStaticNoise(x, y, data, step, params);
    break;
case 538:
    processTunnelVision(x, y, data, width, height, step, params);
    break;
case 539:
    processAberrationPulse(x, y, data, width, step, params);
    break;
case 540:
    processAquaWave(x, y, data, width, height, step, params);
    break;
case 541:

```

```

        processBinaryFlash(x, y, data, step, params);
        break;
case 542:
    processBloomGlow(x, y, data, step, params);
    break;
case 543:
    processCellularNoise(x, y, data, step, params);
    break;
case 544:
    processChromaShift2(x, y, data, width, step, params);
    break;
case 545:
    processColorBands(x, y, data, step, params);
    break;
case 546:
    processColorVortex(x, y, data, width, height, step, params);
    break;
case 547:
    processCrystalMosaic(x, y, data, width, height, step, params);
    break;
case 548:
    processCubicDistort(x, y, data, width, height, step, params);
    break;
case 549:
    processDepthFade(x, y, data, height, step, params);
    break;
case 550:
    processDiscoFlash(x, y, data, step, params);
    break;
case 551:
    processDitherBlend(x, y, data, step, params);
    break;
case 552:
    processDoubleVision(x, y, data, width, step, params);
    break;
case 553:
    processDreamHaze(x, y, data, step, params);
    break;
case 554:
    processElectricStorm(x, y, data, step, params);
    break;
case 555:
    processEmbossShift(x, y, data, width, height, step, params);
    break;
case 556:
    processFiberOptic(x, y, data, width, height, step, params);
    break;
case 557:
    processFilmGrain(x, y, data, step, params);
    break;
case 558:
    processFireWorks(x, y, data, width, height, step, params);
    break;
case 559:
    processFluidMotion(x, y, data, width, height, step, params);
    break;
case 560:
    processFogRoll(x, y, data, height, step, params);
    break;
case 561:
    processGlassRefract(x, y, data, width, height, step, params);
    break;

```

```

case 562:
    processGlowTrails(x, y, data, step, params);
    break;
case 563:
    processGridPulse(x, y, data, step, params);
    break;
case 564:
    processHalftoneBlend(x, y, data, step, params);
    break;
case 565:
    processHeatDistort(x, y, data, width, height, step, params);
    break;
case 566:
    processHoloGlitch(x, y, data, step, params);
    break;
case 567:
    processInfraredView(x, y, data, step, params);
    break;
case 568:
    processLavaLamp(x, y, data, width, height, step, params);
    break;
case 569:
    processLensFlare(x, y, data, width, height, step, params);
    break;
case 570:
    processLightningBolt(x, y, data, width, height, step, params);
    break;
case 571:
    processLiquidMetal(x, y, data, step, params);
    break;
case 572:
    processMatrixCode(x, y, data, height, step, params);
    break;
case 573:
    processMirrorKaleid(x, y, data, width, height, step, params);
    break;
case 574:
    processNightVision(x, y, data, step, params);
    break;
case 575:
    processOilSlick(x, y, data, step, params);
    break;
case 576:
    processParticleField(x, y, data, step, params);
    break;
case 577:
    processPinwheelSpin(x, y, data, width, height, step, params);
    break;
case 578:
    processPixelStorm(x, y, data, width, height, step, params);
    break;
case 579:
    processPlaidPattern(x, y, data, step, params);
    break;
case 580:
    processPolarInvert(x, y, data, width, height, step, params);
    break;
case 581:
    processPolychromeTint(x, y, data, step, params);
    break;
case 582:
    processPopArtDots(x, y, data, step, params);

```

```

        break;
case 583:
    processPrismaticEdge(x, y, data, width, height, step, params);
    break;
case 584:
    processPulseWarp(x, y, data, width, height, step, params);
    break;
case 585:
    processQuantumNoise(x, y, data, step, params);
    break;
case 586:
    processQuiltBlend(x, y, data, step, params);
    break;
case 587:
    processRadarSweep(x, y, data, width, height, step, params);
    break;
case 588:
    processRaindropRipple(x, y, data, width, height, step, params);
    break;
case 589:
    processRasterBars(x, y, data, step, params);
    break;
case 590:
    processRetroTube(x, y, data, width, height, step, params);
    break;
case 591:
    processRingWave(x, y, data, width, height, step, params);
    break;
case 592:
    processRippleTank(x, y, data, width, height, step, params);
    break;
case 593:
    processRotatingPrism(x, y, data, width, height, step, params);
    break;
case 594:
    processSandStorm(x, y, data, width, height, step, params);
    break;
case 595:
    processSaturationPulse(x, y, data, step, params);
    break;
case 596:
    processScatterPixel(x, y, data, width, height, step, params);
    break;
case 597:
    processShadowPlay(x, y, data, width, height, step, params);
    break;
case 598:
    processShimmerGlass(x, y, data, width, height, step, params);
    break;
case 599:
    processSilhouetteBlend(x, y, data, step, params);
    break;
case 600:
    processSketchOutline(x, y, data, width, height, step, params);
    break;
case 601:
    processSliceShift(x, y, data, width, step, params);
    break;
case 602:
    processSmearMotion(x, y, data, width, step, params);
    break;
case 603:

```

```

        processSmokeWisp(x, y, data, width, height, step, params);
        break;
case 604:
    processSnowDrift(x, y, data, step, params);
    break;
case 605:
    processSolarFlare(x, y, data, width, height, step, params);
    break;
case 606:
    processSparkShower(x, y, data, height, step, params);
    break;
case 607:
    processSpectrumWave(x, y, data, step, params);
    break;
case 608:
    processSpiralZoom(x, y, data, width, height, step, params);
    break;
case 609:
    processSplitMirror(x, y, data, width, height, step, params);
    break;
case 610:
    processStarBurst(x, y, data, width, height, step, params);
    break;
case 611:
    processStaticPulse(x, y, data, step, params);
    break;
case 612:
    processStencilCut(x, y, data, step, params);
    break;
case 613:
    processStippleShade(x, y, data, step, params);
    break;
case 614:
    processStormCloud(x, y, data, height, step, params);
    break;
case 615:
    processStreakBlur(x, y, data, width, step, params);
    break;
case 616:
    processStrobeEdge(x, y, data, width, height, step, params);
    break;
case 617:
    processSubpixelShift(x, y, data, width, step, params);
    break;
case 618:
    processSwimDistort(x, y, data, width, height, step, params);
    break;
case 619:
    processTangentWarp(x, y, data, width, height, step, params);
    break;
case 620:
    processTapeGlitch(x, y, data, width, step, params);
    break;
case 621:
    processTechnoGrid(x, y, data, step, params);
    break;
case 622:
    processTeleportPixel(x, y, data, width, height, step, params);
    break;
case 623:
    processTemporalBlur(x, y, data, step, params);
    break;

```

```

case 624:
    processTerraFracture(x, y, data, width, height, step, params);
    break;
case 625:
    processTextureWave(x, y, data, step, params);
    break;
case 626:
    processThresholdPulse(x, y, data, step, params);
    break;
case 627:
    processTidalWave(x, y, data, width, height, step, params);
    break;
case 628:
    processTintCycle(x, y, data, step, params);
    break;
case 629:
    processTraceEdge(x, y, data, width, height, step, params);
    break;
case 630:
    processTriangleMosaic(x, y, data, width, height, step, params);
    break;
case 631:
    processTripleSplit(x, y, data, width, step, params);
    break;
case 632:
    processTurbulentFlow(x, y, data, width, height, step, params);
    break;
case 633:
    processUnderwaterCaustic(x, y, data, step, params);
    break;
case 634:
    processUnsharpPulse(x, y, data, width, height, step, params);
    break;
case 635:
    processVaporTrail(x, y, data, width, step, params);
    break;
case 636:
    processVectorField(x, y, data, width, height, step, params);
    break;
case 637:
    processVelocityBlur(x, y, data, width, height, step, params);
    break;
case 638:
    processVerticalMelt(x, y, data, height, step, params);
    break;
case 639:
    processVHSTracking(x, y, data, width, step, params);
    break;
case 640:
    processVibrantPop(x, y, data, step, params);
    break;
case 641:
    processVignetteFlash(x, y, data, width, height, step, params);
    break;
case 642:
    processVoronoiShatter(x, y, data, width, height, step, params);
    break;
case 643:
    processWarpSpeed(x, y, data, width, height, step, params);
    break;
case 644:
    processWaterColor(x, y, data, width, height, step, params);

```



```

        break;
case 645:
    processWaveCollapse(x, y, data, width, height, step, params);
    break;
case 646:
    processWebPattern(x, y, data, width, height, step, params);
    break;
case 647:
    processWhirlpoolSpin(x, y, data, width, height, step, params);
    break;
case 648:
    processWindBlast(x, y, data, width, step, params);
    break;
case 649:
    processWireframePulse(x, y, data, step, params);
    break;
case 650:
    processXRayFlash(x, y, data, step, params);
    break;
case 651:
    processZebraStripe(x, y, data, step, params);
    break;
case 652:
    processZenRipple(x, y, data, width, height, step, params);
    break;
case 653:
    processZigzagWave(x, y, data, width, height, step, params);
    break;
case 654:
    processZoomPulse(x, y, data, width, height, step, params);
    break;
case 655:
    processZoneTint(x, y, data, width, height, step, params);
    break;
case 656:
    processAcidDrip(x, y, data, height, step, params);
    break;
case 657:
    processAuroraWave(x, y, data, height, step, params);
    break;
case 658:
    processBandPass(x, y, data, step, params);
    break;
case 659:
    processBilinearStretch(x, y, data, width, height, step, params);
    break;
case 660:
    processBleedThrough(x, y, data, width, step, params);
    break;
case 661:
    processBlockShatter(x, y, data, width, height, step, params);
    break;
case 662:
    processBlurMask(x, y, data, width, height, step, params);
    break;
case 663:
    processBokehBlur(x, y, data, width, height, step, params);
    break;
case 664:
    processBounceWave(x, y, data, width, height, step, params);
    break;
case 665:

```

```

        processBrokenGlass(x, y, data, width, height, step, params);
        break;
case 666:
    processBubbleWarp(x, y, data, width, height, step, params);
    break;
case 667:
    processCRTCurvature(x, y, data, width, height, step, params);
    break;
case 668:
    processCascadeBlend(x, y, data, step, params);
    break;
case 669:
    processCelShade(x, y, data, step, params);
    break;
case 670:
    processChainReaction(x, y, data, width, height, step, params);
    break;
case 671:
    processChannelDelay(x, y, data, width, step, params);
    break;
case 672:
    processChromaBleed(x, y, data, width, step, params);
    break;
case 673:
    processCircuitTrace(x, y, data, step, params);
    break;
case 674:
    processClockWipe(x, y, data, width, height, step, params);
    break;
case 675:
    processCloudShadow(x, y, data, step, params);
    break;
case 676:
    processColorBurn(x, y, data, step, params);
    break;
case 677:
    processColorHalves(x, y, data, width, step, params);
    break;
case 678:
    processComicDots(x, y, data, step, params);
    break;
case 679:
    processConcentricPulse(x, y, data, width, height, step, params);
    break;
case 680:
    processCopperTone(x, y, data, step, params);
    break;
case 681:
    processCornerStretch(x, y, data, width, height, step, params);
    break;
case 682:
    processCosmicDust(x, y, data, step, params);
    break;
case 683:
    processCrossBlur(x, y, data, width, height, step, params);
    break;
case 684:
    processCrossProcess(x, y, data, step, params);
    break;
case 685:
    processCrystalEdge(x, y, data, width, height, step, params);
    break;

```

```

case 686:
    processCubeRotate(x, y, data, width, height, step, params);
    break;
case 687:
    processCurtainReveal(x, y, data, width, step, params);
    break;
case 688:
    processCyberPunk(x, y, data, step, params);
    break;
case 689:
    processDataCorrupt(x, y, data, width, step, params);
    break;
case 690:
    processDebrisField(x, y, data, width, height, step, params);
    break;
case 691:
    processDeepFry(x, y, data, step, params);
    break;
case 692:
    processDesyncRGB(x, y, data, width, height, step, params);
    break;
case 693:
    processDiagonalWipe(x, y, data, width, height, step, params);
    break;
case 694:
    processDigitalArtifact(x, y, data, step, params);
    break;
case 695:
    processDimensionRift(x, y, data, width, height, step, params);
    break;
case 696:
    processDotCrawl(x, y, data, step, params);
    break;
case 697:
    processDualTone(x, y, data, step, params);
    break;
case 698:
    processEchoFade(x, y, data, allFrames, step, params);
    break;
case 699:
    processEdgeMelt(x, y, data, width, height, step, params);
    break;
case 700:
    processElasticWarp(x, y, data, width, height, step, params);
    break;
case 701:
    processEmberGlow(x, y, data, step, params);
    break;
case 702:
    processEntropyShift(x, y, data, step, params);
    break;
case 703:
    processErosionBlend(x, y, data, width, height, step, params);
    break;
case 704:
    processExplosionBurst(x, y, data, width, height, step, params);
    break;
case 705:
    processFacetMirror(x, y, data, width, height, step, params);
    break;
case 706:
    processFadeStreak(x, y, data, width, step, params);

```

```

        break;
case 707:
    processFeatherEdge(x, y, data, width, height, step, params);
    break;
case 708:
    processFlashFreeze(x, y, data, step, params);
    break;
case 709:
    processFlipMirror(x, y, data, width, height, step, params);
    break;
case 710:
    processFloatDrift(x, y, data, width, height, step, params);
    break;
case 711:
    processFlowField(x, y, data, width, height, step, params);
    break;
case 712:
    processFoldWarp(x, y, data, width, height, step, params);
    break;
case 713:
    processFragmentScatter(x, y, data, width, height, step, params);
    break;
case 714:
    processFrequencyPulse(x, y, data, step, params);
    break;
case 715:
    processFrostBite(x, y, data, step, params);
    break;
case 716:
    processFuseBlend(x, y, data, allFrames, step, params);
    break;
case 717:
    processGalaxySpiral(x, y, data, width, height, step, params);
    break;
case 718:
    processGelWobble(x, y, data, width, height, step, params);
    break;
case 719:
    processGhostEcho(x, y, data, allFrames, width, step, params);
    break;
case 720:
    processGlassShatter(x, y, data, width, height, step, params);
    break;
case 721:
    processGlimmerPulse(x, y, data, step, params);
    break;
case 722:
    processGlitchMosaic(x, y, data, width, height, step, params);
    break;
case 723:
    processGlowEdge(x, y, data, width, height, step, params);
    break;
case 724:
    processGradientMelt(x, y, data, height, step, params);
    break;
case 725:
    processGrainStorm(x, y, data, step, params);
    break;
case 726:
    processGravityPull(x, y, data, width, height, step, params);
    break;
case 727:

```

```

        processGridWarp(x, y, data, width, height, step, params);
        break;
case 728:
    processHazeLayer(x, y, data, height, step, params);
    break;
case 729:
    processHeatRipple(x, y, data, width, height, step, params);
    break;
case 730:
    processHexagonBlur(x, y, data, width, height, step, params);
    break;
case 731:
    processHighContrast(x, y, data, step, params);
    break;
case 732:
    processHologramScan(x, y, data, height, step, params);
    break;
case 733:
    processHorizonBend(x, y, data, width, height, step, params);
    break;
case 734:
    processHotSpot(x, y, data, width, height, step, params);
    break;
case 735:
    processHueWobble(x, y, data, step, params);
    break;
case 736:
    process_glitch_AlphaDiamond1(x, y, data, allFrames, width, height, step, params);
    break;
case 737:
    process_glitch_AlphaDiamond2(x, y, data, allFrames, width, height, step, params);
    break;
case 738:
    process_glitch_AlphaDiamond3(x, y, data, allFrames, width, height, step, params);
    break;
case 739:
    process_glitch_AlphaDiamond4(x, y, data, allFrames, width, height, step, params);
    break;
case 740:
    process_glitch_AlphaDiamond5(x, y, data, allFrames, width, height, step, params);
    break;
case 741:
    process_glitch_Alphatrails(x, y, data, allFrames, width, height, step, params);
    break;
case 742:
    process_glitch_ColorFadeEx1(x, y, data, allFrames, width, height, step, params);
    break;
case 743:
    process_glitch_ColorFadeEx2(x, y, data, allFrames, width, height, step, params);
    break;
case 744:
    process_glitch_ColorShift(x, y, data, allFrames, width, height, step, params);
    break;
case 745:
    process_glitch_Colorxor01(x, y, data, allFrames, width, height, step, params);
    break;
case 746:
    process_glitch_Colorxor02(x, y, data, allFrames, width, height, step, params);
    break;
case 747:
    process_glitch_Colorxor03(x, y, data, allFrames, width, height, step, params);
    break;

```

```

case 748:
    process_glitch_Colorxor04(x, y, data, allFrames, width, height, step, params);
    break;
case 749:
    process_glitch_Colorxor05(x, y, data, allFrames, width, height, step, params);
    break;
case 750:
    process_glitch_Colorxor06(x, y, data, allFrames, width, height, step, params);
    break;
case 751:
    process_glitch_Colorxor07(x, y, data, allFrames, width, height, step, params);
    break;
case 752:
    process_glitch_Colorxor08(x, y, data, allFrames, width, height, step, params);
    break;
case 753:
    process_glitch_Colorxor09(x, y, data, allFrames, width, height, step, params);
    break;
case 754:
    process_glitch_Colorxor10(x, y, data, allFrames, width, height, step, params);
    break;
case 755:
    process_glitch_Colorxor11(x, y, data, allFrames, width, height, step, params);
    break;
case 756:
    process_glitch_Colorxor12(x, y, data, allFrames, width, height, step, params);
    break;
case 757:
    process_glitch_Colorxor13(x, y, data, allFrames, width, height, step, params);
    break;
case 758:
    process_glitch_Colorxor14(x, y, data, allFrames, width, height, step, params);
    break;
case 759:
    process_glitch_Colorxor15(x, y, data, allFrames, width, height, step, params);
    break;
case 760:
    process_glitch_Colorxor16(x, y, data, allFrames, width, height, step, params);
    break;
case 761:
    process_glitch_Colorxor17(x, y, data, allFrames, width, height, step, params);
    break;
case 762:
    process_glitch_Colorxor18(x, y, data, allFrames, width, height, step, params);
    break;
case 763:
    process_glitch_Colorxor19(x, y, data, allFrames, width, height, step, params);
    break;
case 764:
    process_glitch_Colorxor20(x, y, data, allFrames, width, height, step, params);
    break;
case 765:
    process_glitch_DistortPicture(x, y, data, allFrames, width, height, step, params);
    break;
case 766:
    process_glitch_Echo(x, y, data, allFrames, width, height, step, params);
    break;
case 767:
    process_glitch_FastMonocolortrails(x, y, data, allFrames, width, height, step, params);
    break;
case 768:
    process_glitch_FramePixels(x, y, data, allFrames, width, height, step, params);

```

```

        break;
case 769:
    process_glitch_FrameResizeStretch(x, y, data, allFrames, width, height, step, params);
    break;
case 770:
    process_glitch_FrameSkip(x, y, data, allFrames, width, height, step, params);
    break;
case 771:
    process_glitch_FrameSkipResize(x, y, data, allFrames, width, height, step, params);
    break;
case 772:
    process_glitch_FrameSkipShadow(x, y, data, allFrames, width, height, step, params);
    break;
case 773:
    process_glitch_FrameSkipShadowX2(x, y, data, allFrames, width, height, step, params);
    break;
case 774:
    process_glitch_FrameSlide01(x, y, data, allFrames, width, height, step, params);
    break;
case 775:
    process_glitch_FrameSlide02(x, y, data, allFrames, width, height, step, params);
    break;
case 776:
    process_glitch_FrameSlide03(x, y, data, allFrames, width, height, step, params);
    break;
case 777:
    process_glitch_FrameSlide04(x, y, data, allFrames, width, height, step, params);
    break;
case 778:
    process_glitch_FrameSquareCol(x, y, data, allFrames, width, height, step, params);
    break;
case 779:
    process_glitch_FrameStutterFilter(x, y, data, allFrames, width, height, step, params);
    break;
case 780:
    process_glitch_GlitchX1(x, y, data, allFrames, width, height, step, params);
    break;
case 781:
    process_glitch_Gpt1(x, y, data, allFrames, width, height, step, params);
    break;
case 782:
    process_glitch_Gpt2(x, y, data, allFrames, width, height, step, params);
    break;
case 783:
    process_glitch_Gpt3(x, y, data, allFrames, width, height, step, params);
    break;
case 784:
    process_glitch_Invertflash(x, y, data, allFrames, width, height, step, params);
    break;
case 785:
    process_glitch_LineAcrossSide01(x, y, data, allFrames, width, height, step, params);
    break;
case 786:
    process_glitch_LineAcrossSide02(x, y, data, allFrames, width, height, step, params);
    break;
case 787:
    process_glitch_LineAcrossSide03(x, y, data, allFrames, width, height, step, params);
    break;
case 788:
    process_glitch_LineAcrossSide04(x, y, data, allFrames, width, height, step, params);
    break;
case 789:

```

```

        process_glitch_LineAcrossSide05(x, y, data, allFrames, width, height, step, params);
        break;
case 790:
    process_glitch_LineAcrossSide06(x, y, data, allFrames, width, height, step, params);
    break;
case 791:
    process_glitch_LineCollection01(x, y, data, allFrames, width, height, step, params);
    break;
case 792:
    process_glitch_LineCollection02(x, y, data, allFrames, width, height, step, params);
    break;
case 793:
    process_glitch_LineCollection03(x, y, data, allFrames, width, height, step, params);
    break;
case 794:
    process_glitch_LineCollection04(x, y, data, allFrames, width, height, step, params);
    break;
case 795:
    process_glitch_LineCollection05(x, y, data, allFrames, width, height, step, params);
    break;
case 796:
    process_glitch_LineCollection06(x, y, data, allFrames, width, height, step, params);
    break;
case 797:
    process_glitch_LineCollection07(x, y, data, allFrames, width, height, step, params);
    break;
case 798:
    process_glitch_LineOffsetInout(x, y, data, allFrames, width, height, step, params);
    break;
case 799:
    process_glitch_LineOffsetInout2(x, y, data, allFrames, width, height, step, params);
    break;
case 800:
    process_glitch_LineOffsetInout3(x, y, data, allFrames, width, height, step, params);
    break;
case 801:
    process_glitch_Newrandblend(x, y, data, allFrames, width, height, step, params);
    break;
case 802:
    process_glitch_Newstretchlines(x, y, data, allFrames, width, height, step, params);
    break;
case 803:
    process_glitch_Newvarlines(x, y, data, allFrames, width, height, step, params);
    break;
case 804:
    process_glitch_Outoforder(x, y, data, allFrames, width, height, step, params);
    break;
case 805:
    process_glitch_PicAdjust1(x, y, data, allFrames, width, height, step, params);
    break;
case 806:
    process_glitch_PicAdjust2(x, y, data, allFrames, width, height, step, params);
    break;
case 807:
    process_glitch_PictureJump01(x, y, data, allFrames, width, height, step, params);
    break;
case 808:
    process_glitch_PictureJump02(x, y, data, allFrames, width, height, step, params);
    break;
case 809:
    process_glitch_PictureJump03(x, y, data, allFrames, width, height, step, params);
    break;

```



```

case 810:
    process_glitch_PictureJump04(x, y, data, allFrames, width, height, step, params);
    break;
case 811:
    process_glitch_PictureJump05(x, y, data, allFrames, width, height, step, params);
    break;
case 812:
    process_glitch_PictureJump06(x, y, data, allFrames, width, height, step, params);
    break;
case 813:
    process_glitch_PictureJump07(x, y, data, allFrames, width, height, step, params);
    break;
case 814:
    process_glitch_PictureJump08(x, y, data, allFrames, width, height, step, params);
    break;
case 815:
    process_glitch_PictureJump09(x, y, data, allFrames, width, height, step, params);
    break;
case 816:
    process_glitch_PictureJump10(x, y, data, allFrames, width, height, step, params);
    break;
case 817:
    process_glitch_PictureJump11(x, y, data, allFrames, width, height, step, params);
    break;
case 818:
    process_glitch_PictureJump12(x, y, data, allFrames, width, height, step, params);
    break;
case 819:
    process_glitch_PictureJump13(x, y, data, allFrames, width, height, step, params);
    break;
case 820:
    process_glitch_PictureJump14(x, y, data, allFrames, width, height, step, params);
    break;
case 821:
    process_glitch_PictureJump15(x, y, data, allFrames, width, height, step, params);
    break;
case 822:
    process_glitch_PictureJump16(x, y, data, allFrames, width, height, step, params);
    break;
case 823:
    process_glitch_PictureJump17(x, y, data, allFrames, width, height, step, params);
    break;
case 824:
    process_glitch_PictureJump18(x, y, data, allFrames, width, height, step, params);
    break;
case 825:
    process_glitch_PictureJump19(x, y, data, allFrames, width, height, step, params);
    break;
case 826:
    process_glitch_PictureJump20(x, y, data, allFrames, width, height, step, params);
    break;
case 827:
    process_glitch_PictureJump21(x, y, data, allFrames, width, height, step, params);
    break;
case 828:
    process_glitch_PictureJump22(x, y, data, allFrames, width, height, step, params);
    break;
case 829:
    process_glitch_PictureJump23(x, y, data, allFrames, width, height, step, params);
    break;
case 830:
    process_glitch_PictureJump24(x, y, data, allFrames, width, height, step, params);

```

```

        break;
case 831:
    process_glitch_PictureJump25(x, y, data, allFrames, width, height, step, params);
    break;
case 832:
    process_glitch_PictureJump26(x, y, data, allFrames, width, height, step, params);
    break;
case 833:
    process_glitch_PictureJump27(x, y, data, allFrames, width, height, step, params);
    break;
case 834:
    process_glitch_PictureJump28(x, y, data, allFrames, width, height, step, params);
    break;
case 835:
    process_glitch_PictureJump29(x, y, data, allFrames, width, height, step, params);
    break;
case 836:
    process_glitch_PictureJump30(x, y, data, allFrames, width, height, step, params);
    break;
case 837:
    process_glitch_PictureJump31(x, y, data, allFrames, width, height, step, params);
    break;
case 838:
    process_glitch_RectSize(x, y, data, allFrames, width, height, step, params);
    break;
case 839:
    process_glitch_Rsquare2(x, y, data, allFrames, width, height, step, params);
    break;
case 840:
    process_glitch_Rsquare3(x, y, data, allFrames, width, height, step, params);
    break;
case 841:
    process_glitch_Rsquare4(x, y, data, allFrames, width, height, step, params);
    break;
case 842:
    process_glitch_Rsquare5(x, y, data, allFrames, width, height, step, params);
    break;
case 843:
    process_glitch_SliceFrame(x, y, data, allFrames, width, height, step, params);
    break;
case 844:
    process_glitch_SliceFrameW(x, y, data, allFrames, width, height, step, params);
    break;
case 845:
    process_glitch_SquareBlockV2A(x, y, data, allFrames, width, height, step, params);
    break;
case 846:
    process_glitch_SquareBlockV2B(x, y, data, allFrames, width, height, step, params);
    break;
case 847:
    process_glitch_SquareBlockV2C(x, y, data, allFrames, width, height, step, params);
    break;
case 848:
    process_glitch_SquareBlockV2D(x, y, data, allFrames, width, height, step, params);
    break;
case 849:
    process_glitch_SquareBlockV2E(x, y, data, allFrames, width, height, step, params);
    break;
case 850:
    process_glitch_SquareBlockV2F(x, y, data, allFrames, width, height, step, params);
    break;
case 851:

```

```

        process_glitch_SquareBlockV2G(x, y, data, allFrames, width, height, step, params);
        break;
case 852:
    process_glitch_SquareBlockV2H(x, y, data, allFrames, width, height, step, params);
    break;
case 853:
    process_glitch_SquareXor(x, y, data, allFrames, width, height, step, params);
    break;
case 854:
    process_glitch_StoreFrame10(x, y, data, allFrames, width, height, step, params);
    break;
case 855:
    process_glitch_Stuckframe(x, y, data, allFrames, width, height, step, params);
    break;
case 856:
    process_glitch_StutterLong(x, y, data, allFrames, width, height, step, params);
    break;
case 857:
    process_glitch_StutterSbrv(x, y, data, allFrames, width, height, step, params);
    break;
case 858:
    acgl_glitch_AlphaBlendFive(x, y, data, allFrames, width, height, step, params);
    break;
case 859:
    acgl_glitch_AlphaBlendTri(x, y, data, allFrames, width, height, step, params);
    break;
case 860:
    acgl_glitch_AlphaBlendExpand(x, y, data, allFrames, width, height, step, params);
    break;
case 861:
    acgl_glitch_BarsCol(x, y, data, allFrames, width, height, step, params);
    break;
case 862:
    acgl_glitch_BarsColAlpha(x, y, data, allFrames, width, height, step, params);
    break;
case 863:
    acgl_glitch_BarsHoriz(x, y, data, allFrames, width, height, step, params);
    break;
case 864:
    acgl_glitch_BarsHorizAlpha(x, y, data, allFrames, width, height, step, params);
    break;
case 865:
    acgl_glitch_BlackSquare(x, y, data, allFrames, width, height, step, params);
    break;
case 866:
    acgl_glitch_BlendStuck(x, y, data, allFrames, width, height, step, params);
    break;
case 867:
    acgl_glitch_ColorDistort(x, y, data, allFrames, width, height, step, params);
    break;
case 868:
    acgl_glitch_ColorRect(x, y, data, allFrames, width, height, step, params);
    break;
case 869:
    acgl_glitch_ColorOnOff(x, y, data, allFrames, width, height, step, params);
    break;
case 870:
    acgl_glitch_DEM(x, y, data, allFrames, width, height, step, params);
    break;
case 871:
    acgl_glitch_FrameMirror(x, y, data, allFrames, width, height, step, params);
    break;

```

```

case 872:
    acgl_glitch_FramePix(x, y, data, allFrames, width, height, step, params);
    break;
case 873:
    acgl_glitch_FrameReverse(x, y, data, allFrames, width, height, step, params);
    break;
case 874:
    acgl_glitch_FrameReverse2(x, y, data, allFrames, width, height, step, params);
    break;
case 875:
    acgl_glitch_FrameReverseNoBlend(x, y, data, allFrames, width, height, step, params);
    break;
case 876:
    acgl_glitch_FrameSepBand(x, y, data, allFrames, width, height, step, params);
    break;
case 877:
    acgl_glitch_FrameSwap(x, y, data, allFrames, width, height, step, params);
    break;
case 878:
    acgl_glitch_FrameXBlend(x, y, data, allFrames, width, height, step, params);
    break;
case 879:
    acgl_glitch_FrameXBlendXor(x, y, data, allFrames, width, height, step, params);
    break;
case 880:
    acgl_glitch_FrameYBlend(x, y, data, allFrames, width, height, step, params);
    break;
case 881:
    acgl_glitch_FrameYBlendXor(x, y, data, allFrames, width, height, step, params);
    break;
case 882:
    acgl_glitch_AddMulXor(x, y, data, allFrames, width, height, step, params);
    break;
case 883:
    acgl_glitch_AddXor(x, y, data, allFrames, width, height, step, params);
    break;
case 884:
    acgl_glitch_ColorBarsRand(x, y, data, allFrames, width, height, step, params);
    break;
case 885:
    acgl_glitch_ColorBarsX(x, y, data, allFrames, width, height, step, params);
    break;
case 886:
    acgl_glitch_ColorBarsY(x, y, data, allFrames, width, height, step, params);
    break;
case 887:
    acgl_glitch_ColorShiftY(x, y, data, allFrames, width, height, step, params);
    break;
case 888:
    acgl_glitch_LineCollectionRGB(x, y, data, allFrames, width, height, step, params);
    break;
case 889:
    acgl_glitch_FrameX2(x, y, data, allFrames, width, height, step, params);
    break;
case 890:
    acgl_glitch_NewBars(x, y, data, allFrames, width, height, step, params);
    break;
case 891:
    acgl_glitch_NewBars2(x, y, data, allFrames, width, height, step, params);
    break;
case 892:
    acgl_glitch_NewBlendLines(x, y, data, allFrames, width, height, step, params);

```

```

        break;
    case 893:
        acgl_glitch_NewLines(x, y, data, allFrames, width, height, step, params);
        break;
    case 894:
        acgl_glitch_NewOne(x, y, data, allFrames, width, height, step, params);
        break;
    case 895:
        acgl_glitch_StrobeCycle(x, y, data, allFrames, width, height, step, params);
        break;
    case 896:
        acgl_glitch_StuckFrame2(x, y, data, allFrames, width, height, step, params);
        break;
    case 897:
        acgl_glitch_StuckLine(x, y, data, allFrames, width, height, step, params);
        break;
    case 898:
        acgl_glitch_StuckRow(x, y, data, allFrames, width, height, step, params);
        break;
    case 899:
        acgl_glitch_StuckRowLine(x, y, data, allFrames, width, height, step, params);
        break;
    case 900:
        acgl_glitch_SepBlocks(x, y, data, allFrames, width, height, step, params);
        break;
    case 901:
        acgl_glitch_Plug1(x, y, data, allFrames, width, height, step, params);
        break;
    case 902:
        acgl_glitch_OppositeDir(x, y, data, allFrames, width, height, step, params);
        break;
    case 903:
        acgl_glitch_OffStuck(x, y, data, allFrames, width, height, step, params);
        break;
    case 904:
        acgl_glitch_NewVarBlendLines(x, y, data, allFrames, width, height, step, params);
        break;
    }
}
setAlpha(data, y * step + x * 4, params.isNegative);
}

```

Core Processing (0-99)

Filter #0 - SelfAlphaBlend

Multiplies each pixel's color channels by the current alpha parameter, brightening or dimming the image. When negative mode is on, the colors are inverted first. Creates a pulsing glow or fade effect.

Technique: Per-channel alpha self-multiplication with optional color inversion.

```

// Case 0 in unifiedFilterKernel:
processSelfAlphaBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSelfAlphaBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    unsigned char b = data[idx];

```

```

    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    data[idx] = (unsigned char)(b + (b * params.alpha));
    data[idx + 1] = (unsigned char)(g + (g * params.alpha));
    data[idx + 2] = (unsigned char)(r + (r * params.alpha));
    setAlpha(data, idx, params.isNegative);
}

```

Filter #1 - MedianBlend

Averages the pixel across all stored history frames, then XORs the result with the current pixel and blends the two together with a contrast boost. Produces a shimmering, ghostly look with enhanced contrast and subtle digital artifacts.

Technique: Multi-frame temporal averaging + XOR blend + contrast boost.

```

// Case 1 in unifiedFilterKernel:
processMedianBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processMedianBlend(int x, int y, unsigned char *currentFrame, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sumB = 0;
    int sumG = 0;
    int sumR = 0;
    int validFrames = 0;
    for (int j = 0; j < params.numFrames; ++j) {
        unsigned char *framePtr = allFrames[j];
        if (framePtr != nullptr) {
            sumB += framePtr[idx];
            sumG += framePtr[idx + 1];
            sumR += framePtr[idx + 2];
            validFrames++;
        }
    }
    if (validFrames > 0) {
        int avgB = sumB / validFrames;
        int avgG = sumG / validFrames;
        int avgR = sumR / validFrames;
        unsigned char curB = currentFrame[idx];
        unsigned char curG = currentFrame[idx + 1];
        unsigned char curR = currentFrame[idx + 2];
        unsigned char xorB = curB ^ (unsigned char)(1 + sumB);
        unsigned char xorG = curG ^ (unsigned char)(1 + sumG);
        unsigned char xorR = curR ^ (unsigned char)(1 + sumR);
        unsigned char newB = (unsigned char)((xorB * 0.5f) + (avgB * 0.5f));
        unsigned char newG = (unsigned char)((xorG * 0.5f) + (avgG * 0.5f));
        unsigned char newR = (unsigned char)((xorR * 0.5f) + (avgR * 0.5f));
        float boost = 1.8f;
        int midpoint = 128;
        newB = (unsigned char)fminf(255.0f, fmaxf(0.0f, midpoint + (newB - midpoint) * boost));
        newG = (unsigned char)fminf(255.0f, fmaxf(0.0f, midpoint + (newG - midpoint) * boost));
        newR = (unsigned char)fminf(255.0f, fmaxf(0.0f, midpoint + (newR - midpoint) * boost));
        currentFrame[idx] = newB;
        currentFrame[idx + 1] = newG;
        currentFrame[idx + 2] = newR;
    }
    setAlpha(currentFrame, idx, params.isNegative);
}

```

Filter #2 - MedianBlendXor

Variant of MedianBlend with additional XOR strobing layered on top, creating a more aggressively glitchy and pulsating version of the temporal average effect.

Technique: Multi-frame averaging with XOR strobe overlay.

```
// Case 2 in unifiedFilterKernel:
processMedianBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processMedianBlend(int x, int y, unsigned char *currentFrame, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sumB = 0;
    int sumG = 0;
    int sumR = 0;
    int validFrames = 0;
    for (int j = 0; j < params.numFrames; ++j) {
        unsigned char *framePtr = allFrames[j];
        if (framePtr != nullptr) {
            sumB += framePtr[idx];
            sumG += framePtr[idx + 1];
            sumR += framePtr[idx + 2];
            validFrames++;
        }
    }
    if (validFrames > 0) {
        int avgB = sumB / validFrames;
        int avgG = sumG / validFrames;
        int avgR = sumR / validFrames;
        unsigned char curB = currentFrame[idx];
        unsigned char curG = currentFrame[idx + 1];
        unsigned char curR = currentFrame[idx + 2];
        unsigned char xorB = curB ^ (unsigned char)(1 + sumB);
        unsigned char xorG = curG ^ (unsigned char)(1 + sumG);
        unsigned char xorR = curR ^ (unsigned char)(1 + sumR);
        unsigned char newB = (unsigned char)((xorB * 0.5f) + (avgB * 0.5f));
        unsigned char newG = (unsigned char)((xorG * 0.5f) + (avgG * 0.5f));
        unsigned char newR = (unsigned char)((xorR * 0.5f) + (avgR * 0.5f));
        float boost = 1.8f;
        int midpoint = 128;
        newB = (unsigned char)fminf(255.0f, fmaxf(0.0f, midpoint + (newB - midpoint) * boost));
        newG = (unsigned char)fminf(255.0f, fmaxf(0.0f, midpoint + (newG - midpoint) * boost));
        newR = (unsigned char)fminf(255.0f, fmaxf(0.0f, midpoint + (newR - midpoint) * boost));
        currentFrame[idx] = newB;
        currentFrame[idx + 1] = newG;
        currentFrame[idx + 2] = newR;
    }
    setAlpha(currentFrame, idx, params.isNegative);
}
```

Filter #3 - SquareBlockResize

Divides the frame into horizontal block rows. Each row pulls from a different history frame based on a ping-pong pattern, then blends 50/50 with the current frame. Creates a layered time-slicing effect where different horizontal bands show different moments in time.

Technique: Row-based frame history selection with ping-pong indexing.

```
// Case 3 in unifiedFilterKernel:
processSquareBlockResize(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processSquareBlockResize(int x, int y, unsigned char *currentFrame, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int block_row = y / params.square_size;
    int period = 2 * (params.numFrames - 1);
    if (period <= 0) {
        period = 1;
    }
    int start_pos = (params.start_dir == 1) ? params.start_index : (2 * (params.numFrames - 1)) - params
    ↪ .start_index;
    int pos = (start_pos + block_row) % period;
    int frame_index = (pos < params.numFrames) ? pos : period - pos;
    if (frame_index < 0) {
        frame_index = 0;
    }
    if (frame_index >= params.numFrames) {
        frame_index = params.numFrames - 1;
    }
    int idx = y * step + x * 4;
    unsigned char *historyFrame = allFrames[frame_index];
    if (historyFrame != nullptr) {
        currentFrame[idx] = (unsigned char)(0.5f * currentFrame[idx] + 0.5f * historyFrame[idx]);
        currentFrame[idx + 1] = (unsigned char)(0.5f * currentFrame[idx + 1] + 0.5f * historyFrame[idx +
    ↪ 1]);
        currentFrame[idx + 2] = (unsigned char)(0.5f * currentFrame[idx + 2] + 0.5f * historyFrame[idx +
    ↪ 2]);
    }
}
```

Filter #4 - SelfScaleRefined

Scales each pixel's color channels by the alpha parameter, clamping to 255. Produces a clean brightness/exposure adjustment that can push colors toward white at higher alpha values.

Technique: Direct per-channel scaling with clamping.

```
// Case 4 in unifiedFilterKernel:
processSelfScaleRefined(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSelfScaleRefined(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    float alpha = params.alpha;
    data[idx] = (unsigned char)fminf(255.0f, b * alpha);
    data[idx + 1] = (unsigned char)fminf(255.0f, g * alpha);
}
```



```

    data[idx + 2] = (unsigned char)fminf(255.0f, r * alpha);
    setAlpha(data, idx, params.isNegative);
}

```

Filter #5 - StrangeGlitch

Compares the current pixel against history frames. If the colors differ by more than a threshold (30 per channel), it replaces the pixel with the first significantly different history frame's pixel. Creates a glitchy freeze effect where only moving areas update.

Technique: Color-distance threshold replacement from frame history.

```

// Case 5 in unifiedFilterKernel:
processStrangeGlitch(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processStrangeGlitch(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    for (int q = 0; q < params.numFrames; ++q) {
        unsigned char *otherFrame = allFrames[q];
        if (otherFrame != nullptr) {
            unsigned char ob = otherFrame[idx];
            unsigned char og = otherFrame[idx + 1];
            unsigned char or_ = otherFrame[idx + 2];
            if (!colorBounds(r, g, b, or_, og, ob, 30, 30, 30)) {
                data[idx] = ob;
                data[idx + 1] = og;
                data[idx + 2] = or_;
                break;
            }
        }
    }
}

```

Filter #6 - MatrixOutline

Compares the current pixel against a specific history frame. If the colors are similar (within threshold), the pixel is set to black. Only pixels that have changed significantly remain visible, creating a motion-outline "Matrix"-style wireframe look.

Technique: Color-similarity detection rendering similar pixels black.

```

// Case 6 in unifiedFilterKernel:
processMatrixOutline(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processMatrixOutline(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *compareFrame = allFrames[4 % params.numFrames];
    if (compareFrame != nullptr) {
        if (colorBounds(data[idx + 2], data[idx + 1], data[idx], compareFrame[idx + 2], compareFrame[idx]
            ↪ + 1, compareFrame[idx], 30, 30, 30)) {

```

```

        data[idx] = 0;
        data[idx + 1] = 0;
        data[idx + 2] = 0;
    }
}

```

Filter #7 - AuraTrails

Blends the current pixel with three specific history frames (indices 1, 4, 7) using cascaded 50/50 averaging. Creates smooth, glowing trails behind moving objects with an ethereal aura.

Technique: Cascaded multi-frame 50/50 blending from selected history frames.

```

// Case 7 in unifiedFilterKernel:
processAuraTrails(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processAuraTrails(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int indices[] = {1, 4, 7};
    float sumB = (float)data[idx];
    float sumG = (float)data[idx + 1];
    float sumR = (float)data[idx + 2];
    for (int i = 0; i < 3; ++i) {
        unsigned char *f = allFrames[indices[i] % params.numFrames];
        if (f != nullptr) {
            sumB = (sumB * 0.5f) + (f[idx] * 0.5f);
            sumG = (sumG * 0.5f) + (f[idx + 1] * 0.5f);
            sumR = (sumR * 0.5f) + (f[idx + 2] * 0.5f);
        }
    }
    data[idx] = (unsigned char)sumB;
    data[idx + 1] = (unsigned char)sumG;
    data[idx + 2] = (unsigned char)sumR;
}

```

Filter #8 - MirrorReverseColor

Mirrors the frame while reversing color channels, creating a symmetrical image with swapped color tones.

Technique: Coordinate mirroring with channel reordering.

```

// Case 8 in unifiedFilterKernel:
int px = (width - x - 1);
int py = (height - y - 1);
int i0 = py * step + x * 4;
int i1 = py * step + px * 4;
int i2 = y * step + px * 4;
for (int j = 0; j < 3; j++) {
    data[idx + (2 - j)] = (unsigned char)fminf(255.0f, (data[idx + j] * 0.25f + data[i0 + j] * 0.25f +
    ↪ data[i1 + j] * 0.25f + data[i2 + j] * 0.25f));
}

```

Filter #9 - SquareShrink

Blends the current pixel with frame history only within a shrinking rectangular border. The border offset animates over time, creating a picture-within-a-picture that shrinks inward.

Technique: Border-masked 50/50 frame blending with animated inset.

```
// Case 9 in unifiedFilterKernel:
processSquareShrink(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSquareShrink(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int off = params.start_index;
    if (y >= off && y < (height - off) && x >= off && x < (width - off)) {
        int idx = y * step + x * 4;
        unsigned char *reimage = allFrames[0];
        if (reimage != nullptr) {
            data[idx] = (unsigned char)((data[idx] * 0.5f) + (reimage[idx] * 0.5f));
            data[idx + 1] = (unsigned char)((data[idx + 1] * 0.5f) + (reimage[idx + 1] * 0.5f));
            data[idx + 2] = (unsigned char)((data[idx + 2] * 0.5f) + (reimage[idx + 2] * 0.5f));
        }
    }
}
```

Filter #10 - MotionGhostTrails

Layered ghost trails from multiple history frames create persistent semi-transparent echoes of past motion.

Technique: Multi-frame weighted overlay for motion trailing.

```
// Case 10 in unifiedFilterKernel:
unsigned char *ghost = allFrames[0];
if (ghost) {
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)fminf(255.0f, (data[idx + j] * (1 - params.alpha) + ghost[idx + j]
    ↪ * params.alpha));
    }
}
```

Filter #11 - StretchColMatrix8

Stretches columns of pixels at an 8-pixel interval, pulling color data from frame history to create a horizontal stretch-line matrix pattern.

Technique: Column-interval frame history sampling at 8px blocks.

```
// Case 11 in unifiedFilterKernel:
int sw = (filters[i].index == 11) ? 8 : (filters[i].index == 12 ? 16 : 32);
unsigned char *ref = allFrames[(x / sw) % params.numFrames];
if (ref) {
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
    }
}
```

Filter #12 - StretchColMatrix16

Same as StretchColMatrix8 but with 16-pixel column intervals, creating wider horizontal stretch bands.

Technique: Column-interval frame history sampling at 16px blocks.

```
// Case 12 in unifiedFilterKernel:
int sw = (filters[i].index == 11) ? 8 : (filters[i].index == 12 ? 16 : 32);
unsigned char *ref = allFrames[(x / sw) % params.numFrames];
if (ref) {
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
    }
}
```

Filter #13 - StretchColMatrix32

Same concept at 32-pixel column intervals, producing very wide horizontal stretch bands with a blocky, datamosh-like appearance.

Technique: Column-interval frame history sampling at 32px blocks.

```
// Case 13 in unifiedFilterKernel:
int sw = (filters[i].index == 11) ? 8 : (filters[i].index == 12 ? 16 : 32);
unsigned char *ref = allFrames[(x / sw) % params.numFrames];
if (ref) {
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
    }
}
```

Filter #14 - GradientFlashColor

Adds animated gradient colors that flash and shift over time, mixing with the existing frame to create a colorful pulsing gradient overlay.

Technique: Animated color gradient addition.

```
// Case 14 in unifiedFilterKernel:
float a = (0.5f / height) * y;
for (int j = 0; j < 3; j++) {
    data[idx + j] = (unsigned char)fminf(255.0f, (0.5f * data[idx + j]) + (a * params.float_param1));
}
```

Filter #15 - HorizontalGlitch

Shifts horizontal rows of pixels by variable amounts, creating a broken/torn horizontal glitch distortion.

Technique: Per-row horizontal pixel displacement.

```
// Case 15 in unifiedFilterKernel:
```

```

bool hor = (filters[i].index == 15);
unsigned char *ref = allFrames[(hor ? y : x) % params.numFrames];
if (ref) {
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
    }
}

```

Filter #16 - VerticalGlitch

Shifts vertical columns of pixels by variable amounts, creating a vertical tearing/glitch distortion.

Technique: Per-column vertical pixel displacement.

```

// Case 16 in unifiedFilterKernel:
bool hor = (filters[i].index == 15);
unsigned char *ref = allFrames[(hor ? y : x) % params.numFrames];
if (ref) {
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
    }
}

```

Filter #17 - WaveTrails

Blends the current pixel with two history frames at equal weight (33% each). The history frame indices animate over time, producing smooth trails with a wave-like temporal selection pattern.

Technique: Three-way equal blend with animated frame selection.

```

// Case 17 in unifiedFilterKernel:
processWaveTrails(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processWaveTrails(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
↳ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *frame1 = allFrames[params.int_param1 % params.numFrames];
    unsigned char *frame2 = allFrames[params.int_param2 % params.numFrames];
    if (frame1 && frame2) {
        for (int j = 0; j < 3; ++j) {
            float val = (0.33f * data[idx + j]) + (0.33f * frame1[idx + j]) + (0.33f * frame2[idx + j]);
            data[idx + j] = (unsigned char)fminf(255.0f, val);
        }
    }
}

```

Filter #18 - PixelInterlace

Interlaces pixels from even/odd rows or columns with frame history, creating a woven/interleaved temporal pattern.

Technique: Even/odd row interlacing with frame history.

```

// Case 18 in unifiedFilterKernel:

```

```

unsigned char *ref = allFrames[(y * width + x + params.seed) % params.numFrames];
if (ref) {
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
    }
}
}

```

Filter #19 - ColorWaveTrails

Motion trails with per-channel wave-modulated color, producing trails that shimmer with different hues as they fade.

Technique: Wave-modulated per-channel frame trailing.

```

// Case 19 in unifiedFilterKernel:
processAuraTrails(x, y, data, allFrames, step, params);
processWaveTrails(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processAuraTrails(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int indices[] = {1, 4, 7};
    float sumB = (float)data[idx];
    float sumG = (float)data[idx + 1];
    float sumR = (float)data[idx + 2];
    for (int i = 0; i < 3; ++i) {
        unsigned char *f = allFrames[indices[i] % params.numFrames];
        if (f != nullptr) {
            sumB = (sumB * 0.5f) + (f[idx] * 0.5f);
            sumG = (sumG * 0.5f) + (f[idx + 1] * 0.5f);
            sumR = (sumR * 0.5f) + (f[idx + 2] * 0.5f);
        }
    }
    data[idx] = (unsigned char)sumB;
    data[idx + 1] = (unsigned char)sumG;
    data[idx + 2] = (unsigned char)sumR;
}

__device__ void processWaveTrails(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *frame1 = allFrames[params.int_param1 % params.numFrames];
    unsigned char *frame2 = allFrames[params.int_param2 % params.numFrames];
    if (frame1 && frame2) {
        for (int j = 0; j < 3; ++j) {
            float val = (0.33f * data[idx + j]) + (0.33f * frame1[idx + j]) + (0.33f * frame2[idx + j]);
            data[idx + j] = (unsigned char)fminf(255.0f, val);
        }
    }
}
}

```

Filter #20 - ParticleSlide

Randomly offsets each pixel's source position using GPU-generated pseudo-random values, then blends with a history frame. Makes the image look like it's dissolving into scattered particles sliding around.

Technique: Random coordinate jitter + 50/50 frame blend.

```
// Case 20 in unifiedFilterKernel:
processParticleSlide(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processParticleSlide(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float rx = gpu_rand(x, y, params.seed) * 200.0f - 100.0f;
    float ry = gpu_rand(y, x, params.seed) * 200.0f - 100.0f;
    int px = (x + (int)rx) % width;
    int py = (y + (int)ry) % height;
    if (px < 0) {
        px += width;
    }
    if (py < 0) {
        py += height;
    }
    unsigned char *ref = allFrames[params.frame_count % params.numFrames];
    if (ref) {
        int p_idx = py * step + px * 4;
        data[idx] = (unsigned char)(0.5f * data[idx] + 0.5f * ref[p_idx]);
        data[idx + 1] = (unsigned char)(0.5f * data[idx + 1] + 0.5f * ref[p_idx + 1]);
        data[idx + 2] = (unsigned char)(0.5f * data[idx + 2] + 0.5f * ref[p_idx + 2]);
    }
}
```

Filter #21 - DiagPixelated

Applies diagonal pixelation, grouping pixels along diagonal lines into blocks that share the same color.

Technique: Diagonal block averaging/sampling.

```
// Case 21 in unifiedFilterKernel:
int sw = 16;
int sh = 16;
int sx = (x / sw) * sw;
int sy = (y / sh) * sh;
for (int j = 0; j < 3; j++) {
    data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[sy * step + sx * 4 + j]);
}
```

Filter #22 - DiagPixelatedResize

Diagonal pixelation with an animated block size that grows and shrinks over time.

Technique: Animated-size diagonal block sampling.

```
// Case 22 in unifiedFilterKernel:
int sw = params.sw;
int sh = params.sh;
int sx = (x / sw) * sw;
int sy = (y / sh) * sh;
for (int j = 0; j < 3; j++) {
    data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[sy * step + sx * 4 + j]);
}
```

Filter #23 - RGBShiftTrails

Pulls each R, G, B channel from a different history frame (indices 0, 7, 15), cycling the channel assignment every frame. Creates vivid chromatic separation trails where red, green, and blue ghost independently.

Technique: Per-channel frame history separation with rotating assignment.

```
// Case 23 in unifiedFilterKernel:
processRGBShiftTrails(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processRGBShiftTrails(int x, int y, unsigned char *data, unsigned char **allFrames,
    ⇨ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *f0 = allFrames[0];
    unsigned char *f7 = allFrames[7 % params.numFrames];
    unsigned char *f15 = allFrames[15 % params.numFrames];
    if (f0 && f7 && f15) {
        int off = params.frame_count % 3;
        if (off == 0) {
            data[idx] = f0[idx];
            data[idx + 1] = f7[idx + 1];
            data[idx + 2] = f15[idx + 2];
        } else if (off == 1) {
            data[idx] = f7[idx];
            data[idx + 1] = f15[idx + 1];
            data[idx + 2] = f0[idx + 2];
        } else {
            data[idx] = f15[idx];
            data[idx + 1] = f0[idx + 1];
            data[idx + 2] = f7[idx + 2];
        }
    }
}
```

Filter #24 - PictureShiftDown

Shifts the entire image downward over time, with the top filling from frame history. Creates a scrolling/falling picture effect.

Technique: Vertical coordinate offset with frame history fill.

```
// Case 24 in unifiedFilterKernel:
int ny = (y + params.int_param1) % height;
if (ny < 0) {
    ny += height;
}
for (int j = 0; j < 3; j++) {
    data[idx + j] = data[ny * step + x * 4 + j];
}
```


Filter #25 - PictureShiftRight

Shifts the entire image rightward over time, with the left edge filling from frame history. Creates a horizontal scrolling effect.

Technique: Horizontal coordinate offset with frame history fill.

```
// Case 25 in unifiedFilterKernel:
int nx = (x + params.int_param1) % width;
if (nx < 0) {
    nx += width;
}
for (int j = 0; j < 3; j++) {
    data[idx + j] = data[y * step + nx * 4 + j];
}
```

Filter #26 - PictureShiftVariable

Shifts the image by a variable amount in both directions, creating an unpredictable sliding picture effect.

Technique: Variable-offset coordinate displacement.

```
// Case 26 in unifiedFilterKernel:
int nx = (x + params.int_param2) % width;
int ny = (y + params.int_param1) % height;
if (nx < 0) {
    nx += width;
}
if (ny < 0) {
    ny += height;
}
for (int j = 0; j < 3; j++) {
    data[idx + j] = data[ny * step + nx * 4 + j];
}
```

Filter #27 - StretchR_Right

Stretches only the red channel rightward while leaving green and blue in place, creating a directional red smear.

Technique: Single-channel horizontal coordinate offset (red).

```
// Case 27 in unifiedFilterKernel:
int nx = (x + params.int_param1) % width;
if (nx < 0) {
    nx += width;
}
data[idx + 2] = data[y * step + nx * 4 + 2];
```

Filter #28 - StretchG_Right

Stretches only the green channel rightward, creating a directional green smear trail.

Technique: Single-channel horizontal coordinate offset (green).

```
// Case 28 in unifiedFilterKernel:
int nx = (x + params.int_param1) % width;
if (nx < 0) {
    nx += width;
}
data[idx + 1] = data[y * step + nx * 4 + 1];
```

Filter #29 - StretchB_Right

Stretches only the blue channel rightward, creating a directional blue smear trail.

Technique: Single-channel horizontal coordinate offset (blue).

```
// Case 29 in unifiedFilterKernel:
int nx = (x + params.int_param1) % width;
if (nx < 0) {
    nx += width;
}
data[idx] = data[y * step + nx * 4];
```

Filter #30 - StretchR_Down

Stretches the red channel downward, producing vertical red trailing/smearing.

Technique: Single-channel vertical coordinate offset (red).

```
// Case 30 in unifiedFilterKernel:
int ny = (y + params.int_param1) % height;
if (ny < 0) {
    ny += height;
}
data[idx + 2] = data[ny * step + x * 4 + 2];
```

Filter #31 - StretchG_Down

Stretches the green channel downward, producing vertical green trailing/smearing.

Technique: Single-channel vertical coordinate offset (green).

```
// Case 31 in unifiedFilterKernel:
int ny = (y + params.int_param1) % height;
if (ny < 0) {
    ny += height;
}
data[idx + 1] = data[ny * step + x * 4 + 1];
```

Filter #32 - StretchB_Down

Stretches the blue channel downward, producing vertical blue trailing/smearing.

Technique: Single-channel vertical coordinate offset (blue).

```
// Case 32 in unifiedFilterKernel:
int ny = (y + params.int_param1) % height;
if (ny < 0) {
    ny += height;
}
data[idx] = data[ny * step + x * 4];
```

Filter #33 - Distorted_LinesY

Distorts the image along horizontal lines using variable vertical offsets, creating wavy horizontal bands of distortion.

Technique: Per-row vertical pixel displacement.

```
// Case 33 in unifiedFilterKernel:
int nx = (x + ((params.int_param1 % 2 == 1) ? 0 : -16) + (int)(gpu_rand(x, y, params.seed) * 16.0f)) %
    ↪ width;
if (nx < 0) {
    nx += width;
}
for (int j = 0; j < 3; j++) {
    data[idx + j] = data[y * step + nx * 4 + j];
}
```

Filter #34 - Distorted_LinesX

Distorts the image along vertical lines using variable horizontal offsets, creating wavy vertical bands of distortion.

Technique: Per-column horizontal pixel displacement.

```
// Case 34 in unifiedFilterKernel:
int ny = (y + ((params.int_param1 % 2 == 1) ? 0 : -16) + (int)(gpu_rand(x, y, params.seed) * 16.0f)) %
    ↪ height;
if (ny < 0) {
    ny += height;
}
for (int j = 0; j < 3; j++) {
    data[idx + j] = data[ny * step + x * 4 + j];
}
```

Filter #35 - TripHSV

Converts each pixel to HSV color space, rotates the hue by an animated shift amount, then converts back to RGB. Creates a psychedelic color-cycling effect where all colors rotate around the color wheel.

Technique: HSV hue rotation with animated shift.

```
// Case 35 in unifiedFilterKernel:
processTripHSV(x, y, data, step, params.frame_count);

// Referenced __device__ function(s):
__device__ void processTripHSV(int x, int y, unsigned char *data, size_t step, int shift) {
```

```

int idx = y * step + x * 4;
float b = data[idx] / 255.0f;
float g = data[idx + 1] / 255.0f;
float r = data[idx + 2] / 255.0f;
float max_v = fmaxf(r, fmaxf(g, b));
float min_v = fminf(r, fminf(g, b));
float delta = max_v - min_v;
float h = 0.0f;
if (delta > 0) {
    if (max_v == r) {
        h = 60.0f * fmodf(((g - b) / delta), 6.0f);
    } else if (max_v == g) {
        h = 60.0f * (((b - r) / delta) + 2.0f);
    } else {
        h = 60.0f * (((r - g) / delta) + 4.0f);
    }
}
h = fmodf(h + shift, 360.0f);
if (h < 0) {
    h += 360.0f;
}
float s = (max_v == 0) ? 0 : delta / max_v;
float c = max_v * s;
float x_h = c * (1.0f - fabsf(fmodf(h / 60.0f, 2.0f) - 1.0f));
float m = max_v - c;
float r1, g1, b1;
if (h < 60) {
    r1 = c;
    g1 = x_h;
    b1 = 0;
} else if (h < 120) {
    r1 = x_h;
    g1 = c;
    b1 = 0;
} else if (h < 180) {
    r1 = 0;
    g1 = c;
    b1 = x_h;
} else if (h < 240) {
    r1 = 0;
    g1 = x_h;
    b1 = c;
} else if (h < 300) {
    r1 = x_h;
    g1 = 0;
    b1 = c;
} else {
    r1 = c;
    g1 = 0;
    b1 = x_h;
}
data[idx] = (unsigned char)((b1 + m) * 255.0f);
data[idx + 1] = (unsigned char)((g1 + m) * 255.0f);
data[idx + 2] = (unsigned char)((r1 + m) * 255.0f);
}

```

Filter #36 - XorSumStrobe

XORs pixel values with animated sums of color components, creating a pulsating digital distortion with rapidly shifting false colors.

Technique: Animated XOR with accumulated color sums.

```
// Case 36 in unifiedFilterKernel:
data[idx] ^= (unsigned char)params.sumB;
data[idx + 1] ^= (unsigned char)params.sumG;
data[idx + 2] ^= (unsigned char)params.sumR;
```

Filter #37 - DetectEdges

Applies edge detection (Sobel operator) to highlight boundaries and outlines in the image while suppressing flat areas.

Technique: Sobel gradient edge detection.

```
// Case 37 in unifiedFilterKernel:
processSobel(x, y, data, width, height, step, false, params.threshold);

// Referenced __device__ function(s):
__device__ void processSobel(int x, int y, unsigned char *data, int width, int height, size_t step, bool
    ↪ norm, int threshold) {
    if (x <= 0 || x >= width - 1 || y <= 0 || y >= height - 1) {
        return;
    }
    int idx = y * step + x * 4;
    float gx = 0;
    float gy = 0;
    int weights[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    for (int i = -1; i <= 1; ++i) {
        for (int j = -1; j <= 1; ++j) {
            int c_idx = (y + i) * step + (x + j) * 4;
            float val = (data[c_idx] + data[c_idx + 1] + data[c_idx + 2]) / 3.0f;
            gx += val * weights[i + 1][j + 1];
            gy += val * weights[j + 1][i + 1];
        }
    }
    float mag = fabsf(gx) + fabsf(gy);
    unsigned char v = norm ? (unsigned char)fminf(255.0f, mag) : ((mag > threshold) ? 255 : 0);
    data[idx] = v;
    data[idx + 1] = v;
    data[idx + 2] = v;
}
```

Filter #38 - SobelNorm

Normalized Sobel edge detection with thresholding, producing clean black-and-white line art from the video.

Technique: Sobel edge detection with normalization and threshold.

```
// Case 38 in unifiedFilterKernel:
processSobel(x, y, data, width, height, step, true, params.threshold);

// Referenced __device__ function(s):
```

```

__device__ void processSobel(int x, int y, unsigned char *data, int width, int height, size_t step, bool
↳ norm, int threshold) {
    if (x <= 0 || x >= width - 1 || y <= 0 || y >= height - 1) {
        return;
    }
    int idx = y * step + x * 4;
    float gx = 0;
    float gy = 0;
    int weights[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    for (int i = -1; i <= 1; ++i) {
        for (int j = -1; j <= 1; ++j) {
            int c_idx = (y + i) * step + (x + j) * 4;
            float val = (data[c_idx] + data[c_idx + 1] + data[c_idx + 2]) / 3.0f;
            gx += val * weights[i + 1][j + 1];
            gy += val * weights[j + 1][i + 1];
        }
    }
    float mag = fabsf(gx) + fabsf(gy);
    unsigned char v = norm ? (unsigned char)fminf(255.0f, mag) : ((mag > threshold) ? 255 : 0);
    data[idx] = v;
    data[idx + 1] = v;
    data[idx + 2] = v;
}

```

Filter #39 - LineInLineOut

Alternating horizontal lines pull from different frame history indices, creating a scan-line interlace effect where even and odd rows show different moments in time.

Technique: Row-alternating frame history selection.

```

// Case 39 in unifiedFilterKernel:
processSobel(x, y, data, width, height, step, false, params.threshold);

// Referenced __device__ function(s):
__device__ void processSobel(int x, int y, unsigned char *data, int width, int height, size_t step, bool
↳ norm, int threshold) {
    if (x <= 0 || x >= width - 1 || y <= 0 || y >= height - 1) {
        return;
    }
    int idx = y * step + x * 4;
    float gx = 0;
    float gy = 0;
    int weights[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    for (int i = -1; i <= 1; ++i) {
        for (int j = -1; j <= 1; ++j) {
            int c_idx = (y + i) * step + (x + j) * 4;
            float val = (data[c_idx] + data[c_idx + 1] + data[c_idx + 2]) / 3.0f;
            gx += val * weights[i + 1][j + 1];
            gy += val * weights[j + 1][i + 1];
        }
    }
    float mag = fabsf(gx) + fabsf(gy);
    unsigned char v = norm ? (unsigned char)fminf(255.0f, mag) : ((mag > threshold) ? 255 : 0);
    data[idx] = v;
    data[idx + 1] = v;
    data[idx + 2] = v;
}

```

Filter #40 - LineInLineOut4_Increase

Four-way line interlace with increasing frame offsets. Different rows pull from increasingly distant history frames, creating a fanning time-spread across horizontal bands.

Technique: Multi-level row-based frame history with increasing offset.

```
// Case 40 in unifiedFilterKernel:
int off = params.int_param1 % 100;
int sx = (x / (off + 1)) * (off + 1);
unsigned char *r = allFrames[params.frame_count % params.numFrames];
if (r) {
    for (int j = 0; j < 3; j++) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + r[y * step + sx * 4 + j] * 0.5f);
    }
}
```

Filter #41 - LineInLineOut_ReverseIncrease

Reverse direction variant of the line-interlace increase-rows count backward through frame history, producing a reverse temporal cascade.

Technique: Reverse-indexed row-based frame history interlace.

```
// Case 41 in unifiedFilterKernel:
processLineInLineOut4_Increase(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processLineInLineOut4_Increase(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = get_osc_offset(params.frame_count, 250);
    unsigned char *hist = allFrames[params.frame_count % params.numFrames];
    if (hist) {
        int h_idx = y * step + x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[h_idx + j]));
        }
    }
}
```

Filter #42 - LineInLineOut_ReverseIncrease2

Second variant of the reverse-increase line interlace with different offset calculations for a subtly different temporal pattern.

Technique: Alternate reverse-indexed row interlace.

```
// Case 42 in unifiedFilterKernel:
processLineInLineOut_ReverseIncrease(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processLineInLineOut_ReverseIncrease(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = get_osc_offset(params.frame_count, 250);
```

```

    int src_x = (x + offset) % width;
    int tpos = width - src_x - 1;
    if (tpos >= 0 && tpos < width) {
        unsigned char *hist = allFrames[params.frame_count % params.numFrames];
        if (hist) {
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[idx + j]));
            }
        }
    }
}

```

Filter #43 - LineInLineOut_InvertedY

Line interlace where the Y-axis mapping is inverted, so the bottom of the frame shows the most recent history and the top shows the oldest, flipping the temporal spread upside-down.

Technique: Y-inverted row-based frame history interlace.

```

// Case 43 in unifiedFilterKernel:
processLineInLineOut_InvertedY(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processLineInLineOut_InvertedY(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = get_osc_offset(params.frame_count, 400);
    int zpos = height - y - 1;
    if (zpos >= 0 && zpos < height) {
        unsigned char *hist = allFrames[params.frame_count % params.numFrames];
        if (hist) {
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[idx + j]));
            }
        }
    }
}

```

Filter #44 - LineInLineOut_ReverseInvertedY

Combines reverse indexing with Y-inversion for a doubly-flipped temporal interlace pattern.

Technique: Reverse + Y-inverted row interlace.

```

// Case 44 in unifiedFilterKernel:
int idx = y * step + x * 4;
int offset = get_osc_offset(params.frame_count, 400);
int zpos = height - y - 1;
int rpos = width - x - 1;
unsigned char *hist = allFrames[params.frame_count % params.numFrames];
if (hist && zpos >= 0 && rpos >= 0) {
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[zpos * step + rpos * 4 + j
    ↪ ]));
}

```


Filter #45 - LineInLineOut_Vertical

Vertical variant: columns (instead of rows) pull from different history frames, creating vertical band interlacing.

Technique: Column-based frame history interlace.

```
// Case 45 in unifiedFilterKernel:
processLineInLineOut_Vertical(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processLineInLineOut_Vertical(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = get_osc_offset(params.frame_count, 100);
    int src_y = (y + offset) % height;
    unsigned char *hist = allFrames[params.frame_count % params.numFrames];
    if (hist) {
        int src_idx = src_y * step + x * 4;
        for (int j = 0; j < 3; ++j) {
            data[src_idx + j] = (unsigned char)((0.5f * data[src_idx + j]) + (0.5f * hist[idx + j]));
        }
    }
}
```

Filter #46 - LineInLineOut_VerticalIncrease

Vertical column interlace with increasing frame offsets across columns.

Technique: Column-based frame history with increasing offset.

```
// Case 46 in unifiedFilterKernel:
int idx = y * step + x * 4;
int offset = get_osc_offset(params.frame_count, 400);
int src_y = (y + offset) % height;
unsigned char *hist = allFrames[params.frame_count % params.numFrames];
if (hist) {
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j)
        data[src_idx + j] = (unsigned char)((0.5f * data[src_idx + j]) + (0.5f * hist[idx + j]));
}
```

Filter #47 - LineInLineOut_IncreaseImage

Line interlace with increasing offsets applied to the full image blend, producing a graduated time-spread across the frame.

Technique: Full-image graduated frame history interlace.

```
// Case 47 in unifiedFilterKernel:
processLineInLineOut_IncreaseImage(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processLineInLineOut_IncreaseImage(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *reimage = allFrames[0];
    unsigned char *hist = allFrames[params.frame_count % params.numFrames];
```

```

    if (reimage && hist) {
        for (int j = 0; j < 3; ++j) {
            float val = (0.3f * data[idx + j]) + (0.3f * hist[idx + j]) + (0.3f * reimage[idx + j]);
            data[idx + j] = (unsigned char)fminf(255.0f, val);
        }
    }
}

```

Filter #48 - SquareByRow

Divides the image into rows of square blocks. Each row selects a different history frame using a ping-pong offset, and blends with XOR. Creates blocky temporal bands with digital artifacts.

Technique: Row-block frame selection with XOR blending.

```

// Case 48 in unifiedFilterKernel:
processSquareByRow(x, y, data, allFrames, width, height, step, params, false, false);

// Referenced __device__ function(s):
__device__ void processSquareByRow(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params, bool reverse, bool strobe) {
    int SIZE_VALUE = 96;
    int idx = y * step + x * 4;
    int block_x = x / SIZE_VALUE;
    int block_y = y / SIZE_VALUE;
    float r_val = gpu_rand(block_x, block_y + params.frame_count, params.seed);
    int index = (int)(r_val * 64.0f) % params.numFrames;
    unsigned char *hist = allFrames[index];
    if (hist) {
        int target_x = x;
        if (reverse) {
            if ((block_y % 2) != 0) {
                target_x = width - x - 1;
            }
        }
        if (target_x >= 0 && target_x < width) {
            int h_idx = y * step + target_x * 4;
            if (strobe) {
                for (int j = 0; j < 3; ++j) {
                    data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[h_idx + j]));
                }
            } else {
                data[idx] = hist[h_idx];
                data[idx + 1] = hist[h_idx + 1];
                data[idx + 2] = hist[h_idx + 2];
            }
        }
    }
}
}

```

Filter #49 - SquareByRowRev

Reverse variant of SquareByRow-the frame selection order is reversed, producing a mirrored temporal band pattern.

Technique: Reverse row-block frame selection with XOR blending.

```

// Case 49 in unifiedFilterKernel:
processSquareByRow(x, y, data, allFrames, width, height, step, params, true, false);

// Referenced __device__ function(s):
__device__ void processSquareByRow(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params, bool reverse, bool strobe) {
    int SIZE_VALUE = 96;
    int idx = y * step + x * 4;
    int block_x = x / SIZE_VALUE;
    int block_y = y / SIZE_VALUE;
    float r_val = gpu_rand(block_x, block_y + params.frame_count, params.seed);
    int index = (int)(r_val * 64.0f) % params.numFrames;
    unsigned char *hist = allFrames[index];
    if (hist) {
        int target_x = x;
        if (reverse) {
            if ((block_y % 2) != 0) {
                target_x = width - x - 1;
            }
        }
        if (target_x >= 0 && target_x < width) {
            int h_idx = y * step + target_x * 4;
            if (strobe) {
                for (int j = 0; j < 3; ++j) {
                    data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[h_idx + j]));
                }
            } else {
                data[idx] = hist[h_idx];
                data[idx + 1] = hist[h_idx + 1];
                data[idx + 2] = hist[h_idx + 2];
            }
        }
    }
}

```

Filter #50 - SquareByRow2

Second variant of SquareByRow with a different block-size and frame offset calculation, producing a denser temporal block pattern.

Technique: Alternate row-block frame selection.

```

// Case 50 in unifiedFilterKernel:
processSquareByRow(x, y, data, allFrames, width, height, step, params, false, true);

// Referenced __device__ function(s):
__device__ void processSquareByRow(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params, bool reverse, bool strobe) {
    int SIZE_VALUE = 96;
    int idx = y * step + x * 4;
    int block_x = x / SIZE_VALUE;
    int block_y = y / SIZE_VALUE;
    float r_val = gpu_rand(block_x, block_y + params.frame_count, params.seed);
    int index = (int)(r_val * 64.0f) % params.numFrames;
    unsigned char *hist = allFrames[index];
    if (hist) {
        int target_x = x;
        if (reverse) {
            if ((block_y % 2) != 0) {
                target_x = width - x - 1;
            }
        }
    }
}

```

```

    }
}
if (target_x >= 0 && target_x < width) {
    int h_idx = y * step + target_x * 4;
    if (strobe) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[h_idx + j]));
        }
    } else {
        data[idx] = hist[h_idx];
        data[idx + 1] = hist[h_idx + 1];
        data[idx + 2] = hist[h_idx + 2];
    }
}
}
}
}

```

Filter #51 - DivideByValue

Averages pixel values across up to 8 history frames, then XORs that average with the current pixel. Creates a shimmering digital interference pattern that reacts to motion.

Technique: Multi-frame average XOR blend.

```

// Case 51 in unifiedFilterKernel:
processDivideByValue(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDivideByValue(int x, int y, unsigned char *data, unsigned char **allFrames,
    ⇨ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned int total[3] = {0, 0, 0};
    int count = (params.numFrames > 8) ? 8 : params.numFrames;
    if (count == 0)
        return;
    for (int j = 0; j < count; ++j) {
        unsigned char *f = allFrames[j];
        if (f) {
            total[0] += f[idx];
            total[1] += f[idx + 1];
            total[2] += f[idx + 2];
        }
    }
    for (int j = 0; j < 3; ++j) {
        unsigned char avg = (unsigned char)(total[j] / 5);
        data[idx + j] = data[idx + j] ^ avg;
    }
}

```

Filter #52 - ColorCollectionSubtleStrobe

Pulls each R, G, B channel from three different history frames (indices 1, 3, 6). Creates a subtle color-separated temporal effect where colors come from different moments in time.

Technique: Per-channel selection from fixed history frame indices.

```

// Case 52 in unifiedFilterKernel:

```

```

processColorCollectionSubtleStrobe(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollectionSubtleStrobe(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int idx1 = 1 % params.numFrames;
    int idx2 = 3 % params.numFrames;
    int idx3 = 6 % params.numFrames;
    unsigned char *f1 = allFrames[idx1];
    unsigned char *f2 = allFrames[idx2];
    unsigned char *f3 = allFrames[idx3];
    if (f1 && f2 && f3) {
        data[idx] = f1[idx];
        data[idx + 1] = f2[idx + 1];
        data[idx + 2] = f3[idx + 2];
    }
}

```

Filter #53 - CollectionRandom

Replaces the current pixel with one from a pseudo-randomly selected history frame (based on seed, x, y). Each pixel may come from a different point in time, creating a shattered temporal mosaic.

Technique: Random frame selection per pixel.

```

// Case 53 in unifiedFilterKernel:
processCollectionRandom(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processCollectionRandom(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int frameIdx = (params.seed + x + y) % (params.numFrames > 1 ? params.numFrames - 1 : 1);
    unsigned char *f = allFrames[frameIdx];
    if (f) {
        data[idx] = f[idx];
        data[idx + 1] = f[idx + 1];
        data[idx + 2] = f[idx + 2];
    }
}

```

Filter #54 - CollectionAlphaXor

XORs each color channel of the current pixel with a different history frame (indices 1, 3, and the newest). Produces colorful digital distortion that shifts based on motion between frames.

Technique: Per-channel XOR with different history frames.

```

// Case 54 in unifiedFilterKernel:
processCollectionAlphaXor(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processCollectionAlphaXor(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;

```

```

    int idx1 = 1 % params.numFrames;
    int idx2 = 3 % params.numFrames;
    int idx3 = (params.numFrames - 1) % params.numFrames;
    unsigned char *f1 = allFrames[idx1];
    unsigned char *f2 = allFrames[idx2];
    unsigned char *f3 = allFrames[idx3];
    if (f1 && f2 && f3) {
        data[idx] = data[idx] ^ f1[idx];
        data[idx + 1] = data[idx + 1] ^ f2[idx + 1];
        data[idx + 2] = data[idx + 2] ^ f3[idx + 2];
    }
}

```

Filter #55 - ColorCollection64X

Assembles each color channel from three history frames: frame 1, the middle frame, and an animated frame index. Creates a smooth color-shifting temporal blend.

Technique: Per-channel history frame assembly with animated index.

```

// Case 55 in unifiedFilterKernel:
processColorCollection64X(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollection64X(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int midIdx = params.numFrames / 2;
    int animIdx = params.start_index % params.numFrames;
    unsigned char *f1 = allFrames[1 % params.numFrames];
    unsigned char *f2 = allFrames[midIdx];
    unsigned char *f3 = allFrames[animIdx];
    if (f1 && f2 && f3) {
        data[idx] = f1[idx];
        data[idx + 1] = f2[idx + 1];
        data[idx + 2] = f3[idx + 2];
    }
}

```

Filter #56 - ColorCollectionSwitch

Rotates which history frame provides which color channel on a 3-frame cycle. Every frame, the R/G/B channel source swaps, creating a rotating color-temporal strobe.

Technique: Cycling per-channel frame assignment.

```

// Case 56 in unifiedFilterKernel:
processColorCollectionSwitch(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollectionSwitch(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int switchIdx = params.frame_count % 3;
    int fidx1 = 1 % params.numFrames;
    int fidx2 = params.numFrames / 2;
    int fidx3 = (params.numFrames - 1);

```

```

    unsigned char *frames[3];
    frames[0] = allFrames[fidx1];
    frames[1] = allFrames[fidx2];
    frames[2] = allFrames[fidx3];
    if (frames[0] && frames[1] && frames[2]) {
        switch (switchIdx) {
            case 0:
                data[idx] = frames[0][idx];
                data[idx + 1] = frames[1][idx + 1];
                data[idx + 2] = frames[2][idx + 2];
                break;
            case 1:
                data[idx] = frames[2][idx];
                data[idx + 1] = frames[0][idx + 1];
                data[idx + 2] = frames[1][idx + 2];
                break;
            case 2:
                data[idx] = frames[1][idx];
                data[idx + 1] = frames[2][idx + 1];
                data[idx + 2] = frames[0][idx + 2];
                break;
        }
    }
}

```

Filter #57 - ColorCollectionRGB_Index

Replaces only one color channel per frame (cycling R->G->B) with data from a corresponding history frame. Creates a subtle per-channel flickering update.

Technique: Single-channel-per-frame history replacement.

```

// Case 57 in unifiedFilterKernel:
processColorCollectionRGB_Index(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollectionRGB_Index(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int channelId = params.frame_count % 3;
    int frameIdx = (channelId + 1) % params.numFrames;
    unsigned char *f = allFrames[frameIdx];
    if (f) {
        data[idx + channelId] = f[idx + channelId];
    }
}

```

Filter #58 - ColorCollectionGhostTrails

Blends three history frames (near, middle, far) at equal 33% weight, creating smooth ghost trails that average the recent past into a single ethereal image.

Technique: Three-frame equal-weight temporal averaging.

```

// Case 58 in unifiedFilterKernel:
processColorCollectionGhostTrails(x, y, data, allFrames, step, params);

```

```
// Referenced __device__ function(s):
__device__ void processColorCollectionGhostTrails(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int idx1 = 1 % params.numFrames;
    int idx2 = (params.numFrames / 2) % params.numFrames;
    int idx3 = (params.numFrames - 1) % params.numFrames;
    unsigned char *f1 = allFrames[idx1];
    unsigned char *f2 = allFrames[idx2];
    unsigned char *f3 = allFrames[idx3];
    if (f1 && f2 && f3) {
        for (int j = 0; j < 3; ++j) {
            float val = (f1[idx + j] * 0.33f) + (f2[idx + j] * 0.33f) + (f3[idx + j] * 0.33f);
            data[idx + j] = (unsigned char)fminf(255.0f, val);
        }
    }
}
```

Filter #59 - ColorCollectionScale

Assembles channels from three history frames like ColorCollectionSubtleStrobe, then scales one channel by the alpha parameter. Creates a color-biased temporal composite.

Technique: Per-channel history assembly with alpha scaling.

```
// Case 59 in unifiedFilterKernel:
processColorCollectionScale(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollectionScale(int x, int y, unsigned char *data, unsigned char **allFrames
    ↪ , size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int idx1 = 1 % params.numFrames;
    int idx2 = (params.numFrames / 2) % params.numFrames;
    int idx3 = (params.numFrames - 1) % params.numFrames;
    int channelIdx = params.frame_count % 3;
    unsigned char *f1 = allFrames[idx1];
    unsigned char *f2 = allFrames[idx2];
    unsigned char *f3 = allFrames[idx3];
    if (f1 && f2 && f3) {
        data[idx] = f1[idx];
        data[idx + 1] = f2[idx + 1];
        data[idx + 2] = f3[idx + 2];
        data[idx + channelIdx] = (unsigned char)fminf(255.0f, data[idx + channelIdx] * params.alpha);
    }
}
```

Filter #60 - ColorCollectionReverseStrobe

Alternates the order of history frame channel assignment every other frame-forward order on even frames, reversed on odd frames. Creates a strobing color-swap effect.

Technique: Alternating forward/reverse per-channel frame assignment.

```
// Case 60 in unifiedFilterKernel:
processColorCollectionReverseStrobe(x, y, data, allFrames, step, params);
```



```

// Referenced __device__ function(s):
__device__ void processColorCollectionReverseStrobe(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int idx1 = 1 % params.numFrames;
    int idx2 = 4 % params.numFrames;
    int idx3 = (params.numFrames - 1) % params.numFrames;
    unsigned char *f1 = allFrames[idx1];
    unsigned char *f2 = allFrames[idx2];
    unsigned char *f3 = allFrames[idx3];
    bool reversed = (params.frame_count % 2) == 0;
    if (f1 && f2 && f3) {
        if (reversed) {
            data[idx] = f3[idx];
            data[idx + 1] = f2[idx + 1];
            data[idx + 2] = f1[idx + 2];
        } else {
            data[idx] = f1[idx];
            data[idx + 1] = f2[idx + 1];
            data[idx + 2] = f3[idx + 2];
        }
    }
}

```

Filter #61 - ColorCollectionXorPixel

XORs each pixel channel with a corresponding history frame's channel, with alpha scaling applied to both sides. Alternates between forward and reverse frame ordering every frame.

Technique: Alpha-scaled per-channel XOR with alternating frame order.

```

// Case 61 in unifiedFilterKernel:
processColorCollectionXorPixel(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollectionXorPixel(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int idx1 = 1 % params.numFrames;
    int idx2 = 4 % params.numFrames;
    int idx3 = (params.numFrames - 1) % params.numFrames;
    unsigned char *f1 = allFrames[idx1];
    unsigned char *f2 = allFrames[idx2];
    unsigned char *f3 = allFrames[idx3];
    unsigned char *frames[3] = {f1, f2, f3};
    bool reversed = (params.frame_count % 2) == 0;
    if (f1 && f2 && f3) {
        for (int j = 0; j < 3; ++j) {
            unsigned char *f = reversed ? frames[2 - j] : frames[j];
            unsigned char scaled_pix = (unsigned char)fminf(255.0f, data[idx + j] * params.alpha);
            unsigned char scaled_val = (unsigned char)fminf(255.0f, f[idx + j] * params.alpha);
            data[idx + j] = scaled_pix ^ scaled_val;
        }
    }
}

```

Filter #62 - BlendWithSource25

Blends the current (filtered) frame with the original unfiltered source frame at 25% source / 75% current. Gently grounds the image back toward reality.

Technique: 25/75 source frame blend-back.

```
// Case 62 in unifiedFilterKernel:
processBlendWithSource25(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlendWithSource25(int x, int y, unsigned char *data, unsigned char **allFrames,
    ⇨ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *orig = allFrames[0];
    if (orig) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.75f + orig[idx + j] * 0.25f);
        }
    }
}
```

Filter #63 - BlendWithSource50

50/50 blend between the current filtered frame and the original source. Cuts filter intensity in half, keeping one foot in reality.

Technique: 50/50 source frame blend-back.

```
// Case 63 in unifiedFilterKernel:
processBlendWithSource50(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlendWithSource50(int x, int y, unsigned char *data, unsigned char **allFrames,
    ⇨ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *orig = allFrames[0];
    if (orig) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + orig[idx + j] * 0.5f);
        }
    }
}
```

Filter #64 - BlendWithSource75

75% original source / 25% current filtered frame. Only a hint of the filter effect shows through.

Technique: 75/25 source frame blend-back.

```
// Case 64 in unifiedFilterKernel:
processBlendWithSource75(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlendWithSource75(int x, int y, unsigned char *data, unsigned char **allFrames,
    ⇨ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *orig = allFrames[0];
```

```

    if (orig) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.25f + orig[idx + j] * 0.75f);
        }
    }
}

```

Filter #65 - BlendWithSource100

Full alpha-controlled blend back to the original source. At alpha=1, the image is entirely the original source; at lower alpha, the filter shows through.

Technique: Alpha-parameterized source frame blend-back.

```

// Case 65 in unifiedFilterKernel:
processBlendWithSource100(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlendWithSource100(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *orig = allFrames[0];
    if (orig) {
        float a = fminf(1.0f, params.alpha);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - a) + orig[idx + j] * a);
        }
    }
}

```

Filter #66 - ColorCollectionXorOffsetFlash

Assembles pixels from three history frames, but on a cycling 30-frame interval one channel gets XORed with an offset channel from its source frame instead of directly copied. Creates a periodic flash of XOR distortion.

Technique: Periodic single-channel XOR flash from history frames.

```

// Case 66 in unifiedFilterKernel:
processColorCollectionXorOffsetFlash(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollectionXorOffsetFlash(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offsetVal = (params.frame_count / 30) % 3;
    int idx1 = 1 % params.numFrames;
    int idx2 = (params.numFrames / 2) % params.numFrames;
    int idx3 = (params.numFrames - 1) % params.numFrames;
    unsigned char *f1 = allFrames[idx1];
    unsigned char *f2 = allFrames[idx2];
    unsigned char *f3 = allFrames[idx3];
    unsigned char *frames[3] = {f1, f2, f3};
    if (f1 && f2 && f3) {
        for (int j = 0; j < 3; ++j) {
            if (offsetVal == j) {
                data[idx + j] = data[idx + j] ^ frames[j][idx + offsetVal];
            }
        }
    }
}

```

```

        } else {
            data[idx + j] = frames[j][idx + j];
        }
    }
}
}

```

Filter #67 - ColorCollectionMatrixGhost

Averages three history frames (first, middle, last) at equal weight. Similar to GhostTrails but using the full range of the frame buffer. Creates a smooth, ghostly temporal composite.

Technique: Full-range three-frame temporal averaging.

```

// Case 67 in unifiedFilterKernel:
processColorCollectionMatrixGhost(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollectionMatrixGhost(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int idx1 = 0;
    int idx2 = params.numFrames / 2;
    int idx3 = params.numFrames - 1;
    unsigned char *f1 = allFrames[idx1];
    unsigned char *f2 = allFrames[idx2];
    unsigned char *f3 = allFrames[idx3];
    if (f1 && f2 && f3) {
        for (int j = 0; j < 3; ++j) {
            float val = (f1[idx + j] * 0.33f) + (f2[idx + j] * 0.33f) + (f3[idx + j] * 0.33f);
            data[idx + j] = (unsigned char)fminf(255.0f, val);
        }
    }
}

```

Filter #68 - MildStrobe

Rotates which color channel each of three history frames provides on a 3-frame cycle. The channel *index* also rotates, creating a gentle strobing color-swap between time-separated frames.

Technique: Cycling channel-index rotation across history frames.

```

// Case 68 in unifiedFilterKernel:
processMildStrobe(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processMildStrobe(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int modeIdx = params.frame_count % 3;
    int fidx0 = 0;
    int fidx1 = 3 % params.numFrames;
    int fidx2 = 6 % params.numFrames;
    unsigned char *f0 = allFrames[fidx0];
    unsigned char *f1 = allFrames[fidx1];
    unsigned char *f2 = allFrames[fidx2];
    if (f0 && f1 && f2) {

```

```

    int values[3];
    switch (modeIdx) {
    case 0:
        values[0] = 0;
        values[1] = 1;
        values[2] = 2;
        break;
    case 1:
        values[0] = 1;
        values[1] = 2;
        values[2] = 0;
        break;
    case 2:
        values[0] = 0;
        values[1] = 2;
        values[2] = 1;
        break;
    }
    unsigned char *frames[3] = {f0, f1, f2};
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = frames[j][idx + values[j]];
    }
}
}

```

Filter #69 - ReduceBy50

Halves every pixel's brightness (multiplies by 0.5). Darkens the entire image uniformly, useful as a utility filter combined with others.

Technique: Uniform 50% brightness reduction.

```

// Case 69 in unifiedFilterKernel:
processReduceBy50(x, y, data, step);

// Referenced __device__ function(s):
__device__ void processReduceBy50(int x, int y, unsigned char *data, size_t step) {
    int idx = y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f);
    }
}

```

Filter #70 - ColorPositionAverageXor

XORs each pixel with a global color value derived from the sum of all pixels in the frame (modulo 256). The XOR value is the same for all pixels in a frame but changes every frame, creating a uniform color-shifting distortion.

Technique: Global average color XOR.

```

// Case 70 in unifiedFilterKernel:
processColorPositionAverageXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorPositionAverageXor(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {

```

```

    int idx = y * step + x * 4;
    int values[3];
    values[0] = (params.sumB) % 256;
    values[1] = (params.sumG) % 256;
    values[2] = (params.sumR) % 256;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[idx + j] ^ values[j];
    }
}

```

Filter #71 - ColorPositionXor

XORs the current pixel with the corresponding pixel from the newest history frame, then XORs again with the global average color. Combines temporal and global XOR distortion.

Technique: History frame XOR + global average color XOR.

```

// Case 71 in unifiedFilterKernel:
processColorPositionXor(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorPositionXor(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *hist = allFrames[(params.numFrames - 1) % params.numFrames];
    int values[3];
    values[0] = (params.sumB) % 256;
    values[1] = (params.sumG) % 256;
    values[2] = (params.sumR) % 256;
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (data[idx + j] ^ hist[idx + j]) ^ values[j];
        }
    }
}

```

Filter #72 - GrayStrobe

Converts the image to grayscale on every other frame using standard luminance weights (0.299R + 0.587G + 0.114B). Alternates between full color and black-and-white.

Technique: Alternating-frame grayscale conversion.

```

// Case 72 in unifiedFilterKernel:
processGrayStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGrayStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    if ((params.frame_count % 2) == 0) {
        float gray = 0.299f * data[idx + 2] + 0.587f * data[idx + 1] + 0.114f * data[idx];
        unsigned char g = (unsigned char)fminf(255.0f, gray);
        data[idx] = g;
        data[idx + 1] = g;
        data[idx + 2] = g;
    }
}

```

```
}
```

Filter #73 - ColorStrobeXor

Averages up to 8 history frames per pixel, then XORs with both the current pixel and a global random color derived from frame sums. Creates a busy digital strobe with color artifacts.

Technique: Multi-frame average XOR with global color XOR.

```
// Case 73 in unifiedFilterKernel:
processColorStrobeXor(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorStrobeXor(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char randVal[3] = {
        (unsigned char)(params.sumB % 256),
        (unsigned char)(params.sumG % 256),
        (unsigned char)(params.sumR % 256)};
    float sval[3] = {0, 0, 0};
    int count = (params.numFrames > 8) ? 8 : params.numFrames;
    for (int q = 0; q < count; ++q) {
        unsigned char *f = allFrames[q];
        if (f) {
            for (int j = 0; j < 3; ++j) {
                sval[j] += f[idx + j];
            }
        }
    }
    for (int j = 0; j < 3; ++j) {
        int avg = (int)(sval[j] / count);
        data[idx + j] = data[idx + j] ^ avg ^ randVal[j];
    }
}
```

Filter #74 - ColorGhost

Assembles R, G, B channels from three different history frames—one animated, one from the middle, one from near the end. Creates a smooth color-ghost composite with animated channel drift.

Technique: Animated per-channel history frame assembly.

```
// Case 74 in unifiedFilterKernel:
processColorGhost(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorGhost(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int animIdx = params.start_index % params.numFrames;
    int midIdx = params.numFrames / 2;
    int endIdx = params.numFrames - 2;
    if (endIdx < 0)
        endIdx = 0;
    unsigned char *f1 = allFrames[animIdx];
    unsigned char *f2 = allFrames[midIdx];
```

```

    unsigned char *f3 = allFrames[endIdx];
    if (f1 && f2 && f3) {
        data[idx] = f1[idx];
        data[idx + 1] = f2[idx + 1];
        data[idx + 2] = f3[idx + 2];
    }
}

```

Filter #75 - BlurredOutXor

Averages three history frames (near, middle, far), then XORs the averaged pixel with the current pixel. Produces a soft, blurred XOR distortion.

Technique: Three-frame temporal average then XOR blend.

```

// Case 75 in unifiedFilterKernel:
processBlurredOutXor(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlurredOutXor(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int idx1 = 1 % params.numFrames;
    int idx2 = params.numFrames / 2;
    int idx3 = params.numFrames - 1;
    unsigned char *f1 = allFrames[idx1];
    unsigned char *f2 = allFrames[idx2];
    unsigned char *f3 = allFrames[idx3];
    if (f1 && f2 && f3) {
        for (int j = 0; j < 3; ++j) {
            unsigned char avg = (unsigned char)((f1[idx + j] + f2[idx + j] + f3[idx + j]) / 3);
            data[idx + j] = data[idx + j] ^ avg;
        }
    }
}

```

Filter #76 - DizzyFilter

Averages three history frames (near, middle, far) at 33% each, replacing the current pixel entirely. Creates a dizzy, blurred composite of the recent past.

Technique: Three-frame equal-weight temporal replacement.

```

// Case 76 in unifiedFilterKernel:
processDizzyFilter(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDizzyFilter(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *f1 = allFrames[1 % params.numFrames];
    unsigned char *f2 = allFrames[params.numFrames / 2];
    unsigned char *f3 = allFrames[params.numFrames - 1];
    if (f1 && f2 && f3) {
        for (int j = 0; j < 3; ++j) {
            float val = (f1[idx + j] * 0.33f) + (f2[idx + j] * 0.33f) + (f3[idx + j] * 0.33f);
            data[idx + j] = (unsigned char)val;
        }
    }
}

```



```

    }
}
}

```

Filter #77 - Buzzed

Averages up to 8 history frames with equal weighting, replacing the current pixel. Creates a buzzy, smoothed temporal blur like seeing through vibrating glass.

Technique: Multi-frame (up to 8) equal-weight averaging.

```

// Case 77 in unifiedFilterKernel:
processBuzzed(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBuzzed(int x, int y, unsigned char *data, unsigned char **allFrames, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int count = (params.numFrames > 8) ? 8 : params.numFrames;
    float sep_value = 1.0f / count;
    float value[3] = {0, 0, 0};
    for (int q = 0; q < count; ++q) {
        unsigned char *f = allFrames[q];
        if (f) {
            for (int j = 0; j < 3; ++j) {
                value[j] += f[idx + j] * sep_value;
            }
        }
    }
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)value[j];
    }
}

```

Filter #78 - BuzzedDark

Same as Buzzed but using up to 16 frames and then darkening the result to 70% brightness. Creates a darker, more smeared temporal blur.

Technique: Multi-frame averaging with 70% brightness reduction.

```

// Case 78 in unifiedFilterKernel:
processBuzzedDark(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBuzzedDark(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int count = (params.numFrames > 16) ? 16 : params.numFrames;
    float sep_value = 1.0f / (count > 1 ? count - 1 : 1);
    float value[3] = {0, 0, 0};
    for (int q = 0; q < count; ++q) {
        unsigned char *f = allFrames[q];
        if (f) {
            for (int j = 0; j < 3; ++j) {
                value[j] += f[idx + j] * sep_value;
            }
        }
    }
}

```

```

    }
}
for (int j = 0; j < 3; ++j) {
    float v = value[j] * 0.7f;
    data[idx + j] = (unsigned char)fminf(255.0f, v);
}
}

```

Filter #79 - AllRed

Zeros out the blue and green channels, leaving only the red channel. Turns the entire image into a red monochrome view.

Technique: Blue and green channel zeroing.

```

// Case 79 in unifiedFilterKernel:
processAllRed(x, y, data, step);

// Referenced __device__ function(s):
__device__ void processAllRed(int x, int y, unsigned char *data, size_t step) {
    int idx = y * step + x * 4;
    data[idx] = 0;
    data[idx + 1] = 0;
}

```

Filter #80 - AllGreen

Zeros out the blue and red channels, leaving only green. Turns the image into a green monochrome view.

Technique: Blue and red channel zeroing.

```

// Case 80 in unifiedFilterKernel:
processAllGreen(x, y, data, step);

// Referenced __device__ function(s):
__device__ void processAllGreen(int x, int y, unsigned char *data, size_t step) {
    int idx = y * step + x * 4;
    data[idx] = 0;
    data[idx + 2] = 0;
}

```

Filter #81 - AllBlue

Zeros out the green and red channels, leaving only blue. Turns the image into a blue monochrome view.

Technique: Green and red channel zeroing.

```

// Case 81 in unifiedFilterKernel:
processAllBlue(x, y, data, step);

// Referenced __device__ function(s):
__device__ void processAllBlue(int x, int y, unsigned char *data, size_t step) {
    int idx = y * step + x * 4;

```

```

    data[idx + 1] = 0;
    data[idx + 2] = 0;
}

```

Filter #82 - NegativeStrobe

Inverts all colors (bitwise NOT) on every other frame, alternating between the normal image and its photographic negative.

Technique: Alternating-frame bitwise color inversion.

```

// Case 82 in unifiedFilterKernel:
processNegativeStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processNegativeStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    if ((params.frame_count % 2) == 0) {
        data[idx] = ~data[idx];
        data[idx + 1] = ~data[idx + 1];
        data[idx + 2] = ~data[idx + 2];
    }
}

```

Filter #83 - XorAddMul

Applies three different operations to the three channels: XOR+add on blue, plain add on green, multiply on red, all with an animated integer value. Creates chaotic per-channel math distortion.

Technique: Mixed XOR/add/multiply per-channel with animated operand.

```

// Case 83 in unifiedFilterKernel:
processXorAddMul(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processXorAddMul(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float blend = 1.0f + (params.frame_count % 13);
    int b = (int)blend;
    data[idx] = (unsigned char)(data[idx] + (data[idx] ^ b));
    data[idx + 1] = (unsigned char)(data[idx + 1] + b);
    data[idx + 2] = (unsigned char)(data[idx + 2] * b);
}

```

Filter #84 - HorizontalLines

Adds position-and-time-based values to each color channel with offset cycling, creating scrolling diagonal color line patterns across the image.

Technique: Position-modulated color addition with time offset.

```

// Case 84 in unifiedFilterKernel:
processHorizontalLines(x, y, data, step, params);

```

```
// Referenced __device__ function(s):
__device__ void processHorizontalLines(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float pos0 = (float)((x + y + params.frame_count) % 100);
    float pos1 = (float)((x + y + params.frame_count + 16) % 100);
    float pos2 = (float)((x + y + params.frame_count + 32) % 100);
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + pos0);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + pos1);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + pos2);
}
```

Filter #85 - StrobeRedGreenBlue

Cycles through showing only red, only green, or only blue on each frame (zeroing the other two channels). Produces a rapid R->G->B color strobe.

Technique: Single-channel isolation cycling per frame.

```
// Case 85 in unifiedFilterKernel:
processStrobeRedGreenBlue(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processStrobeRedGreenBlue(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int color = params.frame_count % 3;
    switch (color) {
    case 0:
        data[idx + 1] = 0;
        data[idx + 2] = 0;
        break;
    case 1:
        data[idx] = 0;
        data[idx + 2] = 0;
        break;
    case 2:
        data[idx] = 0;
        data[idx + 1] = 0;
        break;
    }
}
```

Filter #86 - Pulse

Multiplies all channels by a sine-wave-modulated brightness factor, causing the entire image to rhythmically pulse brighter and darker.

Technique: Sine-wave global brightness modulation.

```
// Case 86 in unifiedFilterKernel:
processPulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPulse(int x, int y, unsigned char *data, size_t step, const FilterParams &params)
    ↪ {
```

```

int idx = y * step + x * 4;
float pos = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
data[idx] = (unsigned char)fminf(255.0f, data[idx] + data[idx] * pos);
data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + data[idx + 1] * pos);
data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + data[idx + 2] * pos);
}

```

Filter #87 - DiamondPattern

Creates a diamond/checkerboard pattern by applying different math operations to pixels at even/odd x,y coordinates. Produces a geometric diamond lattice overlay with position-derived colors.

Technique: Even/odd coordinate conditional per-channel math.

```

// Case 87 in unifiedFilterKernel:
processDiamondPattern(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDiamondPattern(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float pos = 1.0f + 0.3f * sinf(params.frame_count * 0.05f);
    if ((x % 2) == 0) {
        if ((y % 2) == 0) {
            data[idx] = (unsigned char)fminf(255.0f, fmaxf(0.0f, (1.0f - pos * data[idx])));
            data[idx + 2] = (unsigned char)((x + y) * pos);
        } else {
            data[idx] = (unsigned char)fminf(255.0f, fmaxf(0.0f, pos * data[idx] - y));
            data[idx + 2] = (unsigned char)fabs((x - y) * pos);
        }
    } else {
        if ((y % 2) == 0) {
            data[idx + 1] = (unsigned char)fminf(255.0f, fmaxf(0.0f, (1.0f - pos * data[idx + 1])));
            data[idx + 2] = (unsigned char)((x + y) * pos);
        } else {
            data[idx + 1] = (unsigned char)fminf(255.0f, fmaxf(0.0f, pos * data[idx + 1] - y));
        }
    }
}
}

```

Filter #88 - Bitwise_XOR

XORs each pixel with the corresponding pixel from a recent history frame. Creates ghostly digital artifacts where the image has changed between frames.

Technique: Frame-to-frame bitwise XOR.

```

// Case 88 in unifiedFilterKernel:
processBitwiseXOR(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBitwiseXOR(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[1 % params.numFrames];
    if (prev) {
        data[idx] ^= prev[idx];
    }
}

```

```

        data[idx + 1] ^= prev[idx + 1];
        data[idx + 2] ^= prev[idx + 2];
    }
}

```

Filter #89 - Bitwise_AND

ANDs each pixel with a recent history frame pixel. Darkens the image, keeping only the bits that are set in both frames. Moving areas get partially blacked out.

Technique: Frame-to-frame bitwise AND.

```

// Case 89 in unifiedFilterKernel:
processBitwiseAND(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBitwiseAND(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[1 % params.numFrames];
    if (prev) {
        data[idx] &= prev[idx];
        data[idx + 1] &= prev[idx + 1];
        data[idx + 2] &= prev[idx + 2];
    }
}

```

Filter #90 - Bitwise_OR

ORs each pixel with a recent history frame pixel. Brightens the image, setting any bit that was set in either frame. Creates bright ghost overlays.

Technique: Frame-to-frame bitwise OR.

```

// Case 90 in unifiedFilterKernel:
processBitwiseOR(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBitwiseOR(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[1 % params.numFrames];
    if (prev) {
        data[idx] |= prev[idx];
        data[idx + 1] |= prev[idx + 1];
        data[idx + 2] |= prev[idx + 2];
    }
}

```

Filter #91 - BlendSwitch

Modulates one color channel (cycling which one based on x position) by a pixel-position-animated blend value. Creates subtle shifting color tones across the frame.

Technique: Position-modulated single-channel blend.

```
// Case 91 in unifiedFilterKernel:
processBlendSwitch(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendSwitch(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int pos = x % 3;
    unsigned char blend_pixel = (unsigned char)((x + y + params.frame_count) % 256);
    data[idx + pos] = (unsigned char)(data[idx + pos] * blend_pixel / 255);
}
```

Filter #92 - LineRGB

Assigns each row to show only one color channel in a repeating R/G/B pattern (row 0 = red only, row 1 = green only, row 2 = blue only). Creates horizontal RGB-striped scan lines.

Technique: Row-based single-channel isolation.

```
// Case 92 in unifiedFilterKernel:
processLineRGB(x, y, data, step);

// Referenced __device__ function(s):
__device__ void processLineRGB(int x, int y, unsigned char *data, size_t step) {
    int idx = y * step + x * 4;
    int counter = y % 3;
    switch (counter) {
    case 0:
        data[idx] = 0;
        data[idx + 1] = 0;
        break;
    case 1:
        data[idx] = 0;
        data[idx + 2] = 0;
        break;
    case 2:
        data[idx + 1] = 0;
        data[idx + 2] = 0;
        break;
    }
}
```

Filter #93 - PixelRGB

Same as LineRGB but per-pixel instead of per-row: each individual pixel shows only one channel in a cycling R/G/B pattern. Creates a fine-grained RGB mosaic like a Bayer filter.

Technique: Per-pixel single-channel isolation.

```
// Case 93 in unifiedFilterKernel:
processPixelRGB(x, y, data, width, step);

// Referenced __device__ function(s):
__device__ void processPixelRGB(int x, int y, unsigned char *data, int width, size_t step) {
    int idx = y * step + x * 4;
    int counter = (y * width + x) % 3;
    switch (counter) {
```

```

    case 0:
        data[idx] = 0;
        data[idx + 1] = 0;
        break;
    case 1:
        data[idx] = 0;
        data[idx + 2] = 0;
        break;
    case 2:
        data[idx + 1] = 0;
        data[idx + 2] = 0;
        break;
}
}

```

Filter #94 - InvertedScanlines

Inverts alternate scan lines (rows), creating horizontal stripes of normal and inverted color.

Technique: Row-alternating color inversion.

```

// Case 94 in unifiedFilterKernel:
processInvertedScanlines(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processInvertedScanlines(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.3f * sinf(params.frame_count * 0.1f);
    int index = (x + y) % 3;
    switch (index) {
    case 0:
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)fminf(255.0f, (~data[idx + j]) * alpha);
        break;
    case 1: {
        unsigned char temp = data[idx];
        data[idx] = (unsigned char)fminf(255.0f, data[idx + 2] * alpha);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * alpha);
        data[idx + 2] = (unsigned char)fminf(255.0f, temp * alpha);
        break;
    }
    case 2:
        break;
    }
}
}

```

Filter #95 - ScanSwitch

Switches between normal and history-frame pixels on alternating scan lines, creating an interlaced blend of present and past.

Technique: Row-alternating current/history frame switching.

```

// Case 95 in unifiedFilterKernel:
processScanSwitch(x, y, data, step, params);

```



```
// Referenced __device__ function(s):
__device__ void processScanSwitch(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int start_index = params.frame_count % 2;
    int index = (start_index + x + y) % 2;
    if (index == 1) {
        data[idx] = ~data[idx];
        data[idx + 1] = ~data[idx + 1];
        data[idx + 2] = ~data[idx + 2];
    }
}
```

Filter #96 - ScanAlphaSwitch

Scan-line switching with alpha blending-alternating rows alpha-blend with a history frame rather than hard-switching. Softer interlace effect.

Technique: Row-alternating alpha-blended frame switching.

```
// Case 96 in unifiedFilterKernel:
processScanAlphaSwitch(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processScanAlphaSwitch(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    int start_index = params.frame_count % 2;
    int index = (start_index + x + y) % 2;
    if (index == 0) {
        data[idx] = ~data[idx];
        data[idx + 1] = ~data[idx + 1];
        data[idx + 2] = ~data[idx + 2];
    } else {
        data[idx] = (unsigned char)fminf(255.0f, data[idx] + data[idx] * alpha);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + data[idx + 1] * alpha);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + data[idx + 2] * alpha);
    }
}
```

Filter #97 - RGBFlash

Rapidly flashes the image with different single-channel emphasis each frame, creating a fast R/G/B flash strobe similar to StrobeRedGreenBlue but with blending.

Technique: Per-frame channel emphasis with flash.

```
// Case 97 in unifiedFilterKernel:
processRGBFlash(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processRGBFlash(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pos = 1.0f + 0.3f * sinf(params.frame_count * 0.2f);
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + pos * params.sumB);
}
```

```

    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + pos * params.sumG);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + pos * params.sumR);
}

```

Filter #98 - DiagonalLines

Adds diagonal-stripe color patterns to the image, with the stripe offset animating over time.

Technique: Diagonal coordinate-based color addition.

```

// Case 98 in unifiedFilterKernel:
processDiagonalLines(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDiagonalLines(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int tz = height - y - 1;
    if (tz >= 0 && tz < height) {
        int tidx = tz * step + x * 4;
        float pos = (float)((x + y + params.frame_count) % 100);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)fminf(255.0f, (data[idx + j] + data[tidx + j]) + pos);
        }
    }
}

```

Filter #99 - Darken

Reduces pixel brightness by subtracting or scaling down all channels, creating a uniform darkening effect.

Technique: Per-channel brightness reduction.

```

// Case 99 in unifiedFilterKernel:
processDarken(x, y, data, step);

// Referenced __device__ function(s):
__device__ void processDarken(int x, int y, unsigned char *data, size_t step) {
    int idx = y * step + x * 4;
    data[idx] /= 6;
    data[idx + 1] /= 6;
    data[idx + 2] /= 6;
}

```

Color & Temporal (100-199)

Filter #100 - SelfXorBlend

XORs each pixel with a scaled version of itself, creating a self-referencing digital distortion that varies with alpha intensity.

Technique: Self-pixel XOR with alpha scaling.

```

// Case 100 in unifiedFilterKernel:

```

```

processSelfXorBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSelfXorBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    unsigned char index0 = (unsigned char)((params.frame_count) % 256);
    unsigned char index1 = (unsigned char)((params.frame_count + 85) % 256);
    unsigned char index2 = (unsigned char)((params.frame_count + 170) % 256);
    data[idx] ^= index0;
    data[idx + 1] ^= index1;
    data[idx + 2] ^= index2;
}

```

Filter #101 - SelfXorDoubleFlash

Double-intensity self-XOR that alternates with a bright flash, producing an aggressive strobe-distortion combo.

Technique: Intensity-doubled self-XOR with alternating flash.

```

// Case 101 in unifiedFilterKernel:
processSelfXorDoubleFlash(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSelfXorDoubleFlash(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 128.0f + 127.0f * sinf(params.frame_count * 0.1f);
    unsigned char a = (unsigned char)alpha;
    data[idx] ^= (a + params.sumB);
    data[idx + 1] ^= (a + params.sumG);
    data[idx + 2] ^= (a + params.sumR);
}

```

Filter #102 - SelfOrDoubleFlash

Bitwise OR of the pixel with itself scaled, plus alternating flash. Brightens and washes out the image with periodic bright bursts.

Technique: Self-pixel OR with alternating flash.

```

// Case 102 in unifiedFilterKernel:
processSelfOrDoubleFlash(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSelfOrDoubleFlash(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 128.0f + 127.0f * sinf(params.frame_count * 0.1f);
    unsigned char a = (unsigned char)alpha;
    data[idx] |= (a ^ params.sumB);
    data[idx + 1] |= (a ^ params.sumG);
    data[idx + 2] |= (a ^ params.sumR);
}

```

Filter #103 - BlendRowCurvedSqrt

Blends rows using a square-root curve for the alpha interpolation, creating a non-linear gradient blend that emphasizes midtones.

Technique: Square-root curved row-based alpha blending.

```
// Case 103 in unifiedFilterKernel:
processBlendRowCurvedSqrt(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendRowCurvedSqrt(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    float amount = sqrtf((float)(x * y));
    unsigned char amt = (unsigned char)fmodf(amount, 256.0f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, (data[idx + j] ^ amt) * alpha);
    }
}
```

Filter #104 - XorAlpha

XORs pixel channels with alpha-modulated values, creating a shifting digital mask that responds to the alpha parameter.

Technique: Alpha-parameterized XOR.

```
// Case 104 in unifiedFilterKernel:
processXorAlpha(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processXorAlpha(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 2.0f * sinf(params.frame_count * 0.05f);
    data[idx] ^= (unsigned char)(params.sumB * alpha);
    data[idx + 1] ^= (unsigned char)(params.sumG * alpha);
    data[idx + 2] ^= (unsigned char)(params.sumR * alpha);
}
```

Filter #105 - RandomXorBlend

XORs with random-offset values and blends with frame history, creating unpredictable glitch-like color artifacts.

Technique: Random-value XOR combined with frame blending.

```
// Case 105 in unifiedFilterKernel:
processRandomXorBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processRandomXorBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    data[idx] ^= params.sumB;
    data[idx + 1] ^= params.sumG;
```

```

    data[idx + 2] ^= params.sumR;
}

```

Filter #106 - AndStrobe

Bitwise AND with animated strobe values, periodically darkening the image in a rhythmic pattern.

Technique: Animated AND strobe.

```

// Case 106 in unifiedFilterKernel:
processAndStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAndStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    data[idx] ^= params.sumB;
    data[idx + 1] ^= params.sumG;
    data[idx + 2] ^= params.sumR;
}

```

Filter #107 - AndStrobeScale

AND strobe with an additional scaling factor, creating a more intense darkening strobe with brightness variation.

Technique: Scaled AND strobe.

```

// Case 107 in unifiedFilterKernel:
processAndStrobeScale(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAndStrobeScale(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 2.0f * sinf(params.frame_count * 0.1f);
    data[idx] = (unsigned char)fminf(255.0f, (data[idx] & params.sumB) * alpha);
    data[idx + 1] = (unsigned char)fminf(255.0f, (data[idx + 1] & params.sumG) * alpha);
    data[idx + 2] = (unsigned char)fminf(255.0f, (data[idx + 2] & params.sumR) * alpha);
}

```

Filter #108 - AndPixelStrobe

AND operation applied per-pixel with a strobing mask, selectively darkening individual pixels in a rhythmic pattern.

Technique: Per-pixel AND with strobing mask.

```

// Case 108 in unifiedFilterKernel:
processAndPixelStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAndPixelStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;

```

```

    unsigned char col = (unsigned char)((params.frame_count * 50) % 256);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[idx + j] & col;
        data[idx + j] = data[idx + j] ^ col;
    }
}

```

Filter #109 - AndOrXorStrobe

Cycles through AND, OR, and XOR operations on alternating frames, creating a complex tri-phase bitwise strobe.

Technique: Cycling AND/OR/XOR bitwise strobe.

```

// Case 109 in unifiedFilterKernel:
processAndOrXorStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAndOrXorStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams
    &params) {
    int idx = y * step + x * 4;
    int index = params.frame_count % 3;
    for (int j = 0; j < 3; ++j) {
        unsigned char col = (j == 0) ? params.sumB : (j == 1 ? params.sumG : params.sumR);
        switch (index) {
            case 0:
                data[idx + j] &= col;
                break;
            case 1:
                data[idx + j] |= col;
                break;
            case 2:
                data[idx + j] ^= col;
                break;
        }
    }
}

```

Filter #110 - FadeInAndOut

Smoothly fades the image brightness up and down using a sine wave, creating a breathing/pulsing fade effect.

Technique: Sine-wave brightness fade modulation.

```

// Case 110 in unifiedFilterKernel:
processFadeInAndOut(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFadeInAndOut(int x, int y, unsigned char *data, size_t step, const FilterParams &
    params) {
    int idx = y * step + x * 4;
    float s0 = 128.0f + 127.0f * sinf(params.frame_count * 0.05f);
    float s1 = 128.0f + 127.0f * sinf(params.frame_count * 0.07f);
    float s2 = 128.0f + 127.0f * sinf(params.frame_count * 0.09f);
    data[idx] ^= (unsigned char)s0;
    data[idx + 1] ^= (unsigned char)s1;
}

```

```

    data[idx + 2] ^= (unsigned char)s2;
}

```

Filter #111 - BrightStrobe

Strobes between normal and boosted brightness, creating a pulsing flash-bright effect.

Technique: Alternating brightness boost strobe.

```

// Case 111 in unifiedFilterKernel:
processBrightStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBrightStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int speed = 1 + (params.frame_count % 25);
    data[idx] ^= (unsigned char)((params.sumB * speed) % 256);
    data[idx + 1] ^= (unsigned char)((params.sumG * speed) % 256);
    data[idx + 2] ^= (unsigned char)((params.sumR * speed) % 256);
}

```

Filter #112 - DarkStrobe

Strobes between normal and darkened image, creating a pulsing dim/shadow effect.

Technique: Alternating brightness reduction strobe.

```

// Case 112 in unifiedFilterKernel:
processDarkStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDarkStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 2.0f * sinf(params.frame_count * 0.1f);
    unsigned char c0 = (params.sumB > 0) ? params.sumB : 1;
    unsigned char c1 = (params.sumG > 0) ? params.sumG : 1;
    unsigned char c2 = (params.sumR > 0) ? params.sumR : 1;
    data[idx] = (unsigned char)fminf(255.0f, (data[idx] % c0) * alpha);
    data[idx + 1] = (unsigned char)fminf(255.0f, (data[idx + 1] % c1) * alpha);
    data[idx + 2] = (unsigned char)fminf(255.0f, (data[idx + 2] % c2) * alpha);
}

```

Filter #113 - RandomXorOpposite

XORs each pixel with a random-offset value and the inverted version of the pixel, creating chaotic complementary-color glitch artifacts.

Technique: Random XOR with inverted self-pixel.

```

// Case 113 in unifiedFilterKernel:
processRandomXorOpposite(x, y, data, step, params);

// Referenced __device__ function(s):

```

```

__device__ void processRandomXorOpposite(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        unsigned char col1 = (j == 0) ? params.sumB : (j == 1 ? params.sumG : params.sumR);
        unsigned char col2 = (unsigned char)((col1 + 128) % 256);
        data[idx + j] = ~data[idx + j] ^ col1 ^ ~col2;
    }
}

```

Filter #114 - GradientRainbow

Adds a rainbow gradient overlay that shifts vertically across the frame, with the colors animated over time. The gradient uses different frequencies for R, G, B creating an animated spectral sweep.

Technique: Position-based multi-frequency color gradient addition.

```

// Case 114 in unifiedFilterKernel:
processGradientRainbow(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGradientRainbow(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float start_color = ((params.sumR % 255) + 1) * 0.5f + y * 0.1f;
    int color_R = (int)(start_color * 4) % 256;
    int color_G = (int)(start_color * 6) % 256;
    int color_B = (int)(start_color * 8) % 256;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + color_B);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + color_G);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + color_R);
}

```

Filter #115 - cossinMultiply

Multiplies the pixel's blue channel additive blend with $\sin(\alpha * x)$, green with $\cos(\alpha * y)$, and red with the average. Creates swirling trigonometric color bands that evolve over time.

Technique: Sine/cosine position-modulated per-channel addition.

```

// Case 115 in unifiedFilterKernel:
processCossinMultiply(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCossinMultiply(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + (params.frame_count % 24) * 0.05f;
    float sinVal = sinf(alpha) * x;
    float cosVal = cosf(alpha) * y;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + fabs(sinVal));
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + fabs(cosVal));
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + (data[idx] + data[idx + 1] + data[idx +
    ↪ 2]) / 3.0f);
}

```

Filter #116 - colorAccumulate1

Cross-accumulates color channels: blue gets red x alpha, green gets blue x alpha, red gets green x alpha. Creates a psychedelic color-rotation feedback effect.

Technique: Cross-channel accumulation with alpha scaling.

```
// Case 116 in unifiedFilterKernel:
processColorAccumulate1(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorAccumulate1(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + (params.frame_count % 24) * 0.05f;
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    data[idx] = (unsigned char)fminf(255.0f, b + r * alpha);
    data[idx + 1] = (unsigned char)fminf(255.0f, g + b * alpha);
    data[idx + 2] = (unsigned char)fminf(255.0f, r + g * alpha);
}
```

Filter #117 - colorAccumulate2

Cross-channel accumulation like colorAccumulate1, but with x and y position offsets added to the channels. Creates position-dependent color-rotation gradients.

Technique: Cross-channel accumulation with position offsets.

```
// Case 117 in unifiedFilterKernel:
processColorAccumulate2(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorAccumulate2(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + (params.frame_count % 24) * 0.05f;
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    data[idx] = (unsigned char)fminf(255.0f, b + r * alpha + y);
    data[idx + 1] = (unsigned char)fminf(255.0f, g + b * alpha + x);
    data[idx + 2] = (unsigned char)fminf(255.0f, r + g * alpha + y - x);
}
```

Filter #118 - WeakBlend

Mildly scales one channel (selected by pixel position) with a small animated value. Creates a subtle, gentle color shimmer.

Technique: Single-channel mild animated scaling.

```
// Case 118 in unifiedFilterKernel:
processWeakBlend(x, y, data, step, params);
```

```
// Referenced __device__ function(s):
__device__ void processWeakBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int index = (x + y) % 3;
    float value = 1.0f + (params.sumR % 10) * 0.5f;
    int val = (int)(data[idx + index] + (data[idx + index] * value)) / 2;
    data[idx + index] = (unsigned char)fminf(255.0f, (float)val);
}
```

Filter #119 - StrobeEffect

Cycles through four different per-channel scaling modes on each frame, with sine-wave modulated alpha. Each mode emphasizes different channel combinations, creating a 4-phase color strobe.

Technique: 4-phase per-channel scaling strobe with sine modulation.

```
// Case 119 in unifiedFilterKernel:
processStrobeEffect(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processStrobeEffect(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    int passIndex = params.frame_count % 4;
    switch (passIndex) {
    case 0:
        data[idx] = (unsigned char)fminf(255.0f, data[idx] * (-alpha + 2.0f));
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * alpha);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * alpha);
        break;
    case 1:
        data[idx] = (unsigned char)fminf(255.0f, data[idx] + data[idx] * alpha);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * (-alpha + 2.0f));
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + data[idx + 2] * alpha);
        break;
    case 2:
        data[idx] = (unsigned char)fminf(255.0f, data[idx] * alpha);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * alpha);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * (-alpha + 2.0f));
        break;
    case 3:
        data[idx] = (unsigned char)fminf(255.0f, data[idx] + data[idx] * alpha);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + params.sumG * alpha);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + params.sumR * alpha);
        break;
    }
}
```

Filter #120 - Blend3

Modulates each color channel with a different-frequency sine wave (0.1, 0.13, 0.17), creating subtle out-of-phase color pulsing.

Technique: Multi-frequency sine-wave per-channel modulation.

```
// Case 120 in unifiedFilterKernel:
processBlend3(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlend3(int x, int y, unsigned char *data, size_t step, const FilterParams &params
    ↪ ) {
    int idx = y * step + x * 4;
    float rValue0 = 0.1f * sinf(params.frame_count * 0.1f);
    float rValue1 = 0.1f * sinf(params.frame_count * 0.13f);
    float rValue2 = 0.1f * sinf(params.frame_count * 0.17f);
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + data[idx] * rValue0);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + data[idx + 1] * rValue1);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + data[idx + 2] * rValue2);
}
```

Filter #121 - NegParadox

Doubles plus alpha-blends each channel, with green mixing in a global sum value. Creates an over-bright paradoxical look with color bleeding.

Technique: Self-doubling with alpha and global color bleed.

```
// Case 121 in unifiedFilterKernel:
processNegParadox(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processNegParadox(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.3f * sinf(params.frame_count * 0.05f);
    data[idx] = (unsigned char)fminf(255.0f, (data[idx] * alpha) + (data[idx] * alpha));
    data[idx + 1] = (unsigned char)fminf(255.0f, (data[idx + 1] * alpha) + (params.sumG * alpha));
    data[idx + 2] = (unsigned char)fminf(255.0f, (data[idx + 1] * alpha) + (data[idx + 2] * alpha));
}
```

Filter #122 - ThoughtMode

Conditionally adds or subtracts channel values based on position, frame count, and global sums. Creates an unpredictable, "thinking" color mutation that varies across the frame.

Technique: Conditional position-dependent channel mutation.

```
// Case 122 in unifiedFilterKernel:
processThoughtMode(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processThoughtMode(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    int mode = (x + y) % 3;
    int sw = params.frame_count % 2;
    int tr = (params.frame_count + 1) % 2;
    if (sw == 1)
        data[idx] = (unsigned char)fminf(255.0f, data[idx] + data[mode] * alpha);
    if (tr == 0)
        data[idx + mode] = (unsigned char)fmaxf(0.0f, data[idx + mode] - data[params.sumR % 3] * alpha);
}
```

```

    data[idx + mode] = (unsigned char)fminf(255.0f, data[idx + mode] + data[idx + mode] * alpha);
}

```

Filter #123 - Tri

Self-adds each channel with alpha modulation, plus adds global sum values to green and red. Creates a bright, washed-out tricolor effect.

Technique: Self-add with global color sum injection.

```

// Case 123 in unifiedFilterKernel:
processTri(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processTri(int x, int y, unsigned char *data, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.3f * sinf(params.frame_count * 0.1f);
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + data[idx] * alpha);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + data[idx + 1] + params.sumG * alpha);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + data[idx + 2] + params.sumR * alpha);
}

```

Filter #124 - Distort

Adds scaled x-position to red, y-position to blue, and alpha-scales green. Creates a diagonal position-based color gradient distortion that shifts over time.

Technique: Position-to-color mapping with alpha animation.

```

// Case 124 in unifiedFilterKernel:
processDistort(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDistort(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.3f * sinf(params.frame_count * 0.1f);
    data[idx] = (unsigned char)fminf(255.0f, (y * alpha) + data[idx]);
    data[idx + 2] = (unsigned char)fminf(255.0f, (x * alpha) + data[idx + 2]);
    data[idx + 1] = (unsigned char)fminf(255.0f, alpha * data[idx + 1]);
}

```

Filter #125 - colorAccumulate3

Cross-channel accumulation with x added to green and y to red, creating position-aware color rotation with a spatial gradient.

Technique: Cross-channel accumulation with position-biased offsets.

```

// Case 125 in unifiedFilterKernel:
processColorAccumulate3(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorAccumulate3(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {

```

```

    int idx = y * step + x * 4;
    float alpha = 1.0f + (params.frame_count % 24) * 0.05f;
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    data[idx] = (unsigned char)fminf(255.0f, b + (r * alpha));
    data[idx + 1] = (unsigned char)fminf(255.0f, g + (b * alpha) + x);
    data[idx + 2] = (unsigned char)fminf(255.0f, r + (g * alpha) + y);
}

```

Filter #126 - filter8

Adds the sum of $(x + y) \times \alpha$ to all three channels equally. Creates a diagonal brightness gradient that animates with sine-wave α .

Technique: Uniform diagonal position-based brightness addition.

```

// Case 126 in unifiedFilterKernel:
processFilter8(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFilter8(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + ((x + y) * alpha));
    }
}

```

Filter #127 - filter3

Adds each channel to the blue channel value scaled by α . Tends to shift the entire image toward the blue channel's brightness pattern.

Technique: Blue-channel-biased brightness redistribution.

```

// Case 127 in unifiedFilterKernel:
processFilter3(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFilter3(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    unsigned char b = data[idx];
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, b + (data[idx + j]) * alpha);
    }
}

```

Filter #128 - rainbowBlend

Adds animated rainbow colors derived from global sums and frame count to each pixel. The R, G, B offsets animate at different speeds, creating an evolving rainbow wash overlay.

Technique: Animated per-channel global-sum color addition.

```
// Case 128 in unifiedFilterKernel:
processRainbowBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processRainbowBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.3f * sinf(params.frame_count * 0.05f);
    int rb = (params.sumR + params.frame_count) % 256;
    int gb = (params.sumG + params.frame_count * 2) % 256;
    int bb = (params.sumB + params.frame_count * 3) % 256;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + alpha * bb);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + alpha * gb);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + alpha * rb);
}
```

Filter #129 - pixelScale

Scales each channel by a sine-animated factor, then adds inter-channel differences/sums. Creates a contrasty, color-shifting effect that oscillates.

Technique: Sine-animated per-channel scaling with cross-channel math.

```
// Case 129 in unifiedFilterKernel:
processPixelScale(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPixelScale(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pos = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    data[idx] = (unsigned char)fminf(255.0f, (b * pos) + (b - r));
    data[idx + 1] = (unsigned char)fminf(255.0f, (g * pos) + (g + g));
    data[idx + 2] = (unsigned char)fminf(255.0f, (r * pos) + (r - b));
}
```

Filter #130 - GradientSelf

Modulates one channel (selected by row position) with a sine-animated scale, then adds a position-derived counter. Creates a row-cycling color gradient.

Technique: Row-selected channel with position gradient.

```
// Case 130 in unifiedFilterKernel:
processGradientSelf(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGradientSelf(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pos = 1.0f + 2.0f * sinf(params.frame_count * 0.1f);
    int index = y % 3;
```

```

    int count = (x + y) % 256;
    data[idx + index] = (unsigned char)fminf(255.0f, (data[idx + index] * pos) + count);
}

```

Filter #131 - GradientSelfVertical

Same as GradientSelf but the channel is selected by column position instead of row, creating vertical color band gradients.

Technique: Column-selected channel with position gradient.

```

// Case 131 in unifiedFilterKernel:
processGradientSelfVertical(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGradientSelfVertical(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float pos = 1.0f + 2.0f * sinf(params.frame_count * 0.1f);
    int index = x % 3;
    int count = (x + y) % 256;
    data[idx + index] = (unsigned char)fminf(255.0f, (data[idx + index] * pos) + count);
}

```

Filter #132 - GradientDown

Modulates a position-selected channel and adds a vertical (y-based) gradient counter. Creates a top-to-bottom color gradient.

Technique: Vertical gradient with position-selected channel.

```

// Case 132 in unifiedFilterKernel:
processGradientDown(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGradientDown(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pos = 1.0f + 2.0f * sinf(params.frame_count * 0.1f);
    int index = (x + y) % 3;
    int count = y % 256;
    data[idx + index] = (unsigned char)fminf(255.0f, (data[idx + index] * pos) + count);
}

```

Filter #133 - GradientHorizontal

Modulates a position-selected channel and adds a horizontal (x-based) gradient counter. Creates a left-to-right color gradient.

Technique: Horizontal gradient with position-selected channel.

```

// Case 133 in unifiedFilterKernel:
processGradientHorizontal(x, y, data, step, params);

// Referenced __device__ function(s):

```

```

__device__ void processGraidentHorizontal(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float pos = 1.0f + 2.0f * sinf(params.frame_count * 0.1f);
    int index = (x + y) % 3;
    int count = x % 256;
    data[idx + index] = (unsigned char)fminf(255.0f, (data[idx + index] * pos) + count);
}

```

Filter #134 - Inter

Blacks out every other row, with the visible/blacked rows alternating each frame. Creates a flickering interlace/scan-line effect.

Technique: Alternating row blanking with frame toggle.

```

// Case 134 in unifiedFilterKernel:
processInter(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processInter(int x, int y, unsigned char *data, size_t step, const FilterParams &params)
    ↪ {
    int idx = y * step + x * 4;
    int start_x = params.frame_count % 2;
    if ((y + start_x) % 2 == 0) {
        data[idx] = 0;
        data[idx + 1] = 0;
        data[idx + 2] = 0;
    }
}

```

Filter #135 - BlendedScanLines

Adds a row-and-position-based color value to one channel (cycling which), creating colored scan-line bands that scroll diagonally.

Technique: Position-modulated single-channel scan-line addition.

```

// Case 135 in unifiedFilterKernel:
processBlendedScanLines(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendedScanLines(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int cnt = (x + y) % 3;
    int r = (params.sumR + y) % 256;
    data[idx + cnt] = (unsigned char)fminf(255.0f, data[idx + cnt] + r);
}

```

Filter #136 - GradientStripes

Adds a vertical gradient to one channel while subtracting from the opposite channel, cycling which channels are affected. Creates animated moving color stripes.

Technique: Channel-opposed vertical gradient with cycling offset.

```
// Case 136 in unifiedFilterKernel:
processGradientStripes(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGradientStripes(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = params.frame_count % 3;
    int count = y % 256;
    int count_i = (params.sumR + params.sumG) % 256;
    data[idx + offset] = (unsigned char)fminf(255.0f, data[idx + offset] + count);
    data[idx + (2 - offset)] = (unsigned char)fmaxf(0.0f, data[idx + (2 - offset)] - count_i);
}
```

Filter #137 - XorSine

XORs each channel with its own sine value scaled by an animated factor and global color sums. Creates a wavy digital distortion with sine-modulated XOR patterns.

Technique: Sine-of-value XOR with animated scaling.

```
// Case 137 in unifiedFilterKernel:
processXorSine(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processXorSine(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pos = 1.0f + 2.0f * sinf(params.frame_count * 0.1f);
    int val0 = (params.sumB % 10) + 1;
    int val1 = (params.sumG % 10) + 1;
    int val2 = (params.sumR % 10) + 1;
    data[idx] ^= (unsigned char)(sinf(data[idx]) * val0 * pos);
    data[idx + 1] ^= (unsigned char)(sinf(data[idx + 1]) * val1 * pos);
    data[idx + 2] ^= (unsigned char)(sinf(data[idx + 2]) * val2 * pos);
}
```

Filter #138 - Circular

Adds cosine- and sine-derived color values based on pixel position with an animated radius. Creates circular/radial color patterns that evolve over time.

Technique: Polar-coordinate-derived color pattern addition.

```
// Case 138 in unifiedFilterKernel:
processCircular(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCircular(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pos = 1.0f + 2.0f * sinf(params.frame_count * 0.1f);
    float rad = 50.0f + (params.frame_count % 50);
    float deg = (x + y) * 0.1f;
    int X_color = (int)(rad * cosf(deg));
```

```

    int Y_color = (int)(rad * sinf(deg));
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + pos * X_color);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * pos);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + pos * Y_color);
}

```

Filter #139 - RandomPixels

Adds random brightness values (0-255) to each channel using GPU pseudo-random function. Creates a heavy static/noise overlay.

Technique: Per-pixel random noise addition.

```

// Case 139 in unifiedFilterKernel:
processRandomPixels(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processRandomPixels(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + r * 255);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + gpu_rand(x + 1, y, params.seed) * 255);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + gpu_rand(x, y + 1, params.seed) * 255);
}

```

Filter #140 - DarkRandomPixels

Adds random brightness up to a variable max, then divides by 4 to darken the result. Creates a darker, grittier static noise.

Technique: Scaled random noise with brightness reduction.

```

// Case 140 in unifiedFilterKernel:
processDarkRandomPixels(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDarkRandomPixels(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int max_v = 1 + (params.sumR % 255);
    float r0 = gpu_rand(x, y, params.seed) * max_v;
    float r1 = gpu_rand(x + 1, y, params.seed) * max_v;
    float r2 = gpu_rand(x, y + 1, params.seed) * max_v;
    data[idx] = (unsigned char)fminf(255.0f, (data[idx] + r0) / 4);
    data[idx + 1] = (unsigned char)fminf(255.0f, (data[idx + 1] + r1) / 4);
    data[idx + 2] = (unsigned char)fminf(255.0f, (data[idx + 2] + r2) / 4);
}

```

Filter #141 - Bars

Sets one channel to full 255 based on cycling x-position bands, creating vertical color bars that shift through R, G, B.

Technique: Position-based single-channel maxing for color bars.

```
// Case 141 in unifiedFilterKernel:
processBars(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBars(int x, int y, unsigned char *data, size_t step, const FilterParams &params)
    ↪ {
    int idx = y * step + x * 4;
    int start = params.frame_count % 3;
    int index = (start + x / 3) % 3;
    data[idx + index] = 255;
}
```

Filter #142 - NegativeByRow

Inverts colors on every other row, creating horizontal alternating-stripe negative bands.

Technique: Row-alternating bitwise NOT.

```
// Case 142 in unifiedFilterKernel:
processNegativeByRow(x, y, data, step);

// Referenced __device__ function(s):
__device__ void processNegativeByRow(int x, int y, unsigned char *data, size_t step) {
    int idx = y * step + x * 4;
    if (y % 2 == 0) {
        data[idx] = ~data[idx];
        data[idx + 1] = ~data[idx + 1];
        data[idx + 2] = ~data[idx + 2];
    }
}
```

Filter #143 - XorScale

XORs each channel with a different animated scale value derived from global sums + frame count. Creates a globally-shifting color distortion.

Technique: Per-channel XOR with animated global scale values.

```
// Case 143 in unifiedFilterKernel:
processXorScale(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processXorScale(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int scale0 = (params.sumB + params.frame_count) % 256;
    int scale1 = (params.sumG + params.frame_count) % 256;
    int scale2 = (params.sumR + params.frame_count) % 256;
    data[idx] ^= scale0;
    data[idx + 1] ^= scale1;
    data[idx + 2] ^= scale2;
}
```

Filter #144 - SelfAlphaRGB

Cycles through 4 modes that variously scale and cross-combine channels with alpha: additive cascading, reversed cascading, shuffled cascading, or XOR cascading. Creates an animated multi-mode color transformation.

Technique: 4-phase cross-channel alpha scaling with mode rotation.

```
// Case 144 in unifiedFilterKernel:
processSelfAlphaRGB(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSelfAlphaRGB(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    int index = params.frame_count % 4;
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    switch (index) {
    case 0:
        data[idx] = (unsigned char)fminf(255.0f, b * alpha);
        data[idx + 1] = (unsigned char)fminf(255.0f, (b + g) * alpha);
        data[idx + 2] = (unsigned char)fminf(255.0f, (b + g + r) * alpha);
        break;
    case 1:
        data[idx + 2] = (unsigned char)fminf(255.0f, b * alpha);
        data[idx + 1] = (unsigned char)fminf(255.0f, (b + g) * alpha);
        data[idx] = (unsigned char)fminf(255.0f, (b + g + r) * alpha);
        break;
    case 2:
        data[idx + 1] = (unsigned char)fminf(255.0f, b * alpha);
        data[idx] = (unsigned char)fminf(255.0f, (b + g) * alpha);
        data[idx + 2] = (unsigned char)fminf(255.0f, (b + g + r) * alpha);
        break;
    case 3:
        data[idx] ^= (unsigned char)(b * alpha);
        data[idx + 1] ^= (unsigned char)((b + g) * alpha);
        data[idx + 2] ^= (unsigned char)((b + g + r) * alpha);
        break;
    }
}
```

Filter #145 - BitwiseXorStrobe

Applies different sine-frequency alpha scaling to each channel, rotating which channel gets which frequency every frame. Creates a smooth multi-tempo color strobe.

Technique: Multi-frequency sine-scaled channel rotation.

```
// Case 145 in unifiedFilterKernel:
processBitwiseXorStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBitwiseXorStrobe(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha1 = 1.0f + 0.3f * sinf(params.frame_count * 0.1f);
    float alpha2 = 1.0f + 0.5f * sinf(params.frame_count * 0.15f);
```

```

float alpha3 = 1.0f + 0.7f * sinf(params.frame_count * 0.2f);
int index = params.frame_count % 3;
switch (index) {
case 0:
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + data[idx] * alpha1);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + data[idx + 1] * alpha2);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + data[idx + 2] * alpha3);
    break;
case 1:
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx] + data[idx] * alpha1);
    data[idx] = (unsigned char)fminf(255.0f, data[idx + 1] + data[idx + 1] * alpha2);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 2] + data[idx + 2] * alpha3);
    break;
case 2:
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx] + data[idx] * alpha1);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 1] + data[idx + 1] * alpha2);
    data[idx] = (unsigned char)fminf(255.0f, data[idx + 2] + data[idx + 2] * alpha3);
    break;
}
}

```

Filter #146 - OrStrobe

ORs the current pixel with a random global color and the previous frame pixel, all scaled by sine-animated alpha. Creates a bright, OR-heavy strobe with color drift.

Technique: Animated OR with random color and history frame.

```

// Case 146 in unifiedFilterKernel:
processOrStrobe(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processOrStrobe(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.05f);
    unsigned char *prev = allFrames[0];
    unsigned char randB = (params.sumB + params.frame_count) % 256;
    unsigned char randG = (params.sumG + params.frame_count * 2) % 256;
    unsigned char randR = (params.sumR + params.frame_count * 3) % 256;
    if (prev) {
        data[idx] = (unsigned char)((data[idx] | randB | prev[idx]) * alpha);
        data[idx + 1] = (unsigned char)((data[idx + 1] | randG | prev[idx + 1]) * alpha);
        data[idx + 2] = (unsigned char)((data[idx + 2] | randR | prev[idx + 2]) * alpha);
    } else {
        data[idx] = (unsigned char)(data[idx] | randB);
        data[idx + 1] = (unsigned char)(data[idx + 1] | randG);
        data[idx + 2] = (unsigned char)(data[idx + 2] | randR);
    }
}

```

Filter #147 - DivideAndIncH

Adds horizontal and vertical position-mapped gradient increments to the blue and green channels, creating a grid-aligned color gradient based on frame dimensions.

Technique: Dimension-normalized position-to-color gradient (horizontal).

```
// Case 147 in unifiedFilterKernel:
processDivideAndIncH(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDivideAndIncH(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int row_x = width / 255;
    int row_y = height / 255;
    if (row_x == 0)
        row_x = 1;
    if (row_y == 0)
        row_y = 1;
    int inc_x = (x / row_x) % 256;
    int inc_y = (y / row_y) % 256;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + inc_x);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + inc_y);
}
```

Filter #148 - DivideAndIncW

Same as DivideAndIncH but with x and y mapping swapped, creating the orthogonal gradient direction.

Technique: Dimension-normalized position-to-color gradient (vertical).

```
// Case 148 in unifiedFilterKernel:
processDivideAndIncW(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDivideAndIncW(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int row_x = width / 255;
    int row_y = height / 255;
    if (row_x == 0)
        row_x = 1;
    if (row_y == 0)
        row_y = 1;
    int inc_x = (y / row_x) % 256;
    int inc_y = (x / row_y) % 256;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + inc_x);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + inc_y);
}
```

Filter #149 - RandomIncrease

Adds pseudo-random color offsets derived from global sums and frame count (different multipliers per channel). Creates animated, frame-wide random color shifting.

Technique: Global random per-channel color addition.

```
// Case 149 in unifiedFilterKernel:
processRandomIncrease(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processRandomIncrease(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
```

```

    int idx = y * step + x * 4;
    unsigned char r0 = (params.sumB + params.frame_count * 17) % 256;
    unsigned char r1 = (params.sumG + params.frame_count * 23) % 256;
    unsigned char r2 = (params.sumR + params.frame_count * 31) % 256;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + r0);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + r1);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + r2);
}

```

Filter #150 - SelfAlphaScaleBlend

Scales each channel by (alpha + 1), accumulates a running sum across channels, then XORs each channel with that sum. Creates a cascading self-referencing digital distortion.

Technique: Cumulative cross-channel alpha scaling with XOR feedback.

```

// Case 150 in unifiedFilterKernel:
processSelfAlphaScaleBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSelfAlphaScaleBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + (params.frame_count % 24) * 0.05f;
    unsigned int val = 0;
    for (int j = 0; j < 3; ++j) {
        val += (unsigned char)(data[idx + j] * (alpha + 1));
        data[idx + j] = data[idx + j] ^ (unsigned char)val;
    }
}

```

Filter #151 - FadeBars

XORs each channel with a position-derived color, then scales by animated alpha. Creates fading vertical color bars with XOR artifacts.

Technique: Position-color XOR with alpha fade.

```

// Case 151 in unifiedFilterKernel:
processFadeBars(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processFadeBars(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + (params.frame_count % 24) * 0.1f;
    unsigned char ch0 = (params.sumB + x) % 256;
    unsigned char ch1 = (params.sumG + x * 2) % 256;
    unsigned char ch2 = (params.sumR + x * 3) % 256;
    data[idx] = (unsigned char)((data[idx] ^ ch0) * alpha);
    data[idx + 1] = (unsigned char)((data[idx + 1] ^ ch1) * alpha);
    data[idx + 2] = (unsigned char)((data[idx + 2] ^ ch2) * alpha);
}

```

Filter #152 - StrobeXor

XORs with an animated strobe value that changes per frame, creating a rapid digital strobe effect.

Technique: Frame-animated XOR strobe.

```
// Case 152 in unifiedFilterKernel:
processStrobeXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processStrobeXor(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    data[idx] ^= (unsigned char)params.sumB;
    data[idx + 1] ^= (unsigned char)params.sumG;
    data[idx + 2] ^= (unsigned char)params.sumR;
}
```

Filter #153 - Blank

Sets all pixels to black (or a uniform color), effectively clearing the frame. Useful as a reset or transition filter.

Technique: Full-frame pixel zeroing.

```
// Case 153 in unifiedFilterKernel:
processBlank(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlank(int x, int y, unsigned char *data, size_t step, const FilterParams &params)
    ↪ {
    int idx = y * step + x * 4;
    float alpha = 1.0f + (params.frame_count % 24) * 0.1f;
    bool color_switch = (params.frame_count % 2) == 0;
    unsigned char val[3];
    for (int j = 0; j < 3; ++j) {
        val[j] = (unsigned char)((alpha * data[idx + j]) / (2 - j + 1));
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + (val[2 - j] / (j + 1)));
        if (color_switch)
            data[idx + j] = ~data[idx + j];
    }
}
```

Filter #154 - ColorVariableBlend

Blends channels with animated variable amounts, creating color-shifting effects that smoothly vary over time.

Technique: Per-channel variable-rate blending.

```
// Case 154 in unifiedFilterKernel:
processColorVariableBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorVariableBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
```



```

    int rgb0 = (params.sumB + params.frame_count * 10) % 256;
    int rgb1 = (params.sumG + params.frame_count * 10) % 256;
    int rgb2 = (params.sumR + params.frame_count * 10) % 256;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + rgb0);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + rgb1);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + rgb2);
}

```

Filter #155 - ColorXorBlend

XOR blending with animated color channel mixing, creating shifting digital color distortion.

Technique: Color-animated XOR blend.

```

// Case 155 in unifiedFilterKernel:
processColorXorBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorXorBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int rgb0 = (params.sumB + params.frame_count * 5) % 256;
    int rgb1 = (params.sumG + params.frame_count * 7) % 256;
    int rgb2 = (params.sumR + params.frame_count * 11) % 256;
    data[idx] ^= (unsigned char)rgb0;
    data[idx + 1] ^= (unsigned char)rgb1;
    data[idx + 2] ^= (unsigned char)rgb2;
}

```

Filter #156 - ColorAddBlend

Additive blending of color channels with animated mixing, creating a bright, washed-out color-shifting overlay.

Technique: Animated per-channel additive blend.

```

// Case 156 in unifiedFilterKernel:
processColorAddBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorAddBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    int rgb0 = (int)(params.sumB + r * 50) % 256;
    int rgb1 = (int)(params.sumG + r * 50) % 256;
    int rgb2 = (int)(params.sumR + r * 50) % 256;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] + rgb0);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + rgb1);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + rgb2);
}

```

Filter #157 - SurroundingPixels

Averages each pixel with its immediate neighbors (3x3 box filter kernel), creating a simple spatial blur effect.

Technique: 3x3 box filter spatial averaging.

```
// Case 157 in unifiedFilterKernel:
processSurroundingPixels(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSurroundingPixels(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    if (x >= width - 1 || y >= height - 1)
        return;
    int idx = y * step + x * 4;
    int idx1 = y * step + (x + 1) * 4;
    int idx2 = (y + 1) * step + x * 4;
    int idx3 = (y + 1) * step + (x + 1) * 4;
    for (int j = 0; j < 3; ++j) {
        int sum = data[idx + j] + data[idx1 + j] + data[idx2 + j] + data[idx3 + j];
        int avg = sum / 4;
        data[idx + j] ^= (unsigned char)avg;
    }
}
```

Filter #158 - SurroundingPixelsAlpha

3x3 box blur with alpha-modulated intensity, controlling how much the blur mixes with the original.

Technique: Alpha-weighted 3x3 spatial blur.

```
// Case 158 in unifiedFilterKernel:
processSurroundingPixelsAlpha(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSurroundingPixelsAlpha(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    if (x >= width - 1 || y >= height - 1)
        return;
    int idx = y * step + x * 4;
    int idx1 = y * step + (x + 1) * 4;
    int idx2 = (y + 1) * step + x * 4;
    int idx3 = (y + 1) * step + (x + 1) * 4;
    float alpha = 1.0f + (params.frame_count % 24) * 0.05f;
    for (int j = 0; j < 3; ++j) {
        int sum = data[idx + j] + data[idx1 + j] + data[idx2 + j] + data[idx3 + j];
        int avg = sum / 3;
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * (alpha + 1) + avg * alpha);
    }
}
```

Filter #159 - DarkModBlend

Blends with frame history using modular arithmetic that tends to darken the result. Creates dark, moody motion trails.

Technique: Modular dark-biased frame blending.

```
// Case 159 in unifiedFilterKernel:
processDarkModBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDarkModBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int rgb0 = (params.sumB + params.frame_count * 5) % 256;
    int rgb1 = (params.sumG + params.frame_count * 5) % 256;
    int rgb2 = (params.sumR + params.frame_count * 5) % 256;
    for (int j = 0; j < 3; ++j) {
        unsigned char val = (j == 0) ? rgb0 : (j == 1 ? rgb1 : rgb2);
        if (data[idx + j] == 0)
            data[idx + j] = 1;
        data[idx + j] = (val ^ data[idx + j]);
        if (data[idx + j] == 0)
            data[idx + j] = 1;
        data[idx + j] = val % data[idx + j];
    }
}
```

Filter #160 - IncreaseDecreaseGamma

Cycles gamma brightness up and then back down over time, creating an animated exposure pulsation.

Technique: Cycling gamma adjustment.

```
// Case 160 in unifiedFilterKernel:
processIncreaseDecreaseGamma(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processIncreaseDecreaseGamma(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int light = 1 + (params.frame_count % 10);
    float gamma = 1.0f + light * 0.1f;
    for (int j = 0; j < 3; ++j) {
        float normalized = data[idx + j] / 255.0f;
        data[idx + j] = (unsigned char)(powf(normalized, gamma) * 255.0f);
    }
}
```

Filter #161 - BlendChannelXor

XORs individual channels with values from frame history and blends, creating channel-specific digital artifacts.

Technique: Per-channel history XOR blend.

```
// Case 161 in unifiedFilterKernel:
processBlendChannelXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendChannelXor(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
```

```

    int cur_color = (params.sumR + params.frame_count) % 100;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * alpha) ^ cur_color;
    }
}

```

Filter #162 - IncDifference

Computes the difference between the current frame and a history frame, with an increasing amplification factor. Highlights motion with growing intensity.

Technique: Amplified frame differencing.

```

// Case 162 in unifiedFilterKernel:
processIncDifference(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processIncDifference(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + prev[idx + j]);
        }
    }
}

```

Filter #163 - IncDifferenceAlpha

Frame difference with alpha-modulated amplification, controlling how strongly motion is highlighted.

Technique: Alpha-controlled amplified frame differencing.

```

// Case 163 in unifiedFilterKernel:
processIncDifferenceAlpha(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processIncDifferenceAlpha(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)fminf(255.0f, (data[idx + j] + prev[idx + j]) * alpha);
        }
    }
}

```

Filter #164 - MirrorXorAlpha

XORs the pixel with its mirrored counterpart from the opposite side of the frame, with alpha blending. Creates symmetric digital distortion.

Technique: Mirror-coordinate XOR with alpha blend.

```
// Case 164 in unifiedFilterKernel:
processMirrorXorAlpha(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processMirrorXorAlpha(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha0 = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    float alpha1 = 1.0f + 0.5f * sinf(params.frame_count * 0.15f + 1.0f);
    float alpha2 = 1.0f + 0.5f * sinf(params.frame_count * 0.2f + 2.0f);
    int mx = width - x - 1;
    int my = height - y - 1;
    if (mx < 0)
        mx = 0;
    if (my < 0)
        my = 0;
    int idx0 = my * step + mx * 4;
    int idx1 = my * step + x * 4;
    int idx2 = y * step + mx * 4;
    unsigned char colorB = (params.sumB + params.frame_count) % 256;
    unsigned char colorG = (params.sumG + params.frame_count) % 256;
    unsigned char colorR = (params.sumR + params.frame_count) % 256;
    for (int j = 0; j < 3; ++j) {
        unsigned char p = (unsigned char)(data[idx + j] * alpha0);
        unsigned char v0 = (unsigned char)(data[idx0 + j] * alpha0);
        unsigned char v1 = (unsigned char)(data[idx1 + j] * alpha1);
        unsigned char v2 = (unsigned char)(data[idx2 + j] * alpha2);
        unsigned char col = (j == 0) ? colorB : (j == 1 ? colorG : colorR);
        data[idx + j] = (p ^ v0 ^ v1 ^ v2) ^ col;
    }
}
```

Filter #165 - IntertwinedMirror

Intertwines normal pixels with horizontally-mirrored pixels in alternating bands. Creates a woven symmetrical pattern.

Technique: Alternating-band mirror interleaving.

```
// Case 165 in unifiedFilterKernel:
processIntertwinedMirror(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processIntertwinedMirror(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int mx = width - x - 1;
    int my = height - y - 1;
    if (mx < 0)
        mx = 0;
    if (my < 0)
        my = 0;
    int idx0 = my * step + mx * 4;
    int idx1 = my * step + x * 4;
    int idx2 = y * step + mx * 4;
    int lines = y % 4;
    for (int j = 0; j < 3; ++j) {
        unsigned char val;
```

```

        switch (lines) {
        case 0:
            val = data[idx0 + j];
            break;
        case 1:
            val = data[idx1 + j];
            break;
        case 2:
            val = data[idx2 + j];
            break;
        default:
            val = data[idx + j];
            break;
        }
        data[idx + j] = val;
    }
}

```

Filter #166 - ColorFadeFilter

Smoothly fades colors toward a target hue over time, creating a gradual color wash transition.

Technique: Time-interpolated color fade.

```

// Case 166 in unifiedFilterKernel:
processColorFadeFilter(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorFadeFilter(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float fade = 0.5f + 0.5f * sinf(params.frame_count * 0.05f);
    int rgb0 = (params.sumB + params.frame_count * 3) % 256;
    int rgb1 = (params.sumG + params.frame_count * 5) % 256;
    int rgb2 = (params.sumR + params.frame_count * 7) % 256;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] * fade + rgb0 * (1 - fade));
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * fade + rgb1 * (1 - fade));
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * fade + rgb2 * (1 - fade));
}

```

Filter #167 - ColorChannelMoveUpAndDown

Shifts color channels vertically in opposite directions (one up, one down), creating vertical chromatic separation.

Technique: Per-channel vertical displacement.

```

// Case 167 in unifiedFilterKernel:
processColorChannelMoveUpAndDown(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorChannelMoveUpAndDown(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int channel = params.frame_count % 3;
    int dir = (params.frame_count / 3) % 2;
    int shift = 10 + (params.frame_count % 20);
}

```

```

    if (dir == 0) {
        data[idx + channel] = (unsigned char)fminf(255.0f, data[idx + channel] + shift);
    } else {
        data[idx + channel] = (unsigned char)fmaxf(0.0f, data[idx + channel] - shift);
    }
}

```

Filter #168 - MedianStrobe

Alternates between median-blended and normal frames. On median frames, pixels are averaged from history; on normal frames, they pass through. Creates a strobed smoothing effect.

Technique: Alternating median temporal blur.

```

// Case 168 in unifiedFilterKernel:
processMedianStrobe(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processMedianStrobe(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sw = params.frame_count % 2;
    unsigned char *hist = allFrames[params.frame_count % params.numFrames];
    if (sw == 0 && hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = hist[idx + j];
        }
    }
}

```

Filter #169 - RGBBlend

Blends R, G, B channels from different history frames with the current frame, creating a per-channel temporal blend effect.

Technique: Per-channel history blending (RGB order).

```

// Case 169 in unifiedFilterKernel:
processRGBBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processRGBBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float scale = 1.5f;
    int sum0 = data[idx];
    int sum1 = data[idx + 1];
    int sum2 = data[idx + 2];
    for (int q = 0; q < 3; ++q) {
        sum0 = (int)(sum0 / scale);
        sum1 = (int)(sum1 / scale);
        sum2 = (int)(sum2 / scale);
    }
    data[idx] ^= (unsigned char)sum0;
    data[idx + 1] ^= (unsigned char)sum1;
    data[idx + 2] ^= (unsigned char)sum2;
}

```

Filter #170 - BGRBlend

Same as RGBBlend but with reversed channel order (B, G, R), creating a different color-temporal combination.

Technique: Per-channel history blending (BGR order).

```
// Case 170 in unifiedFilterKernel:
processBGRBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBGRBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    int sum0 = b + b + b;
    int sum1 = g + g + g;
    int sum2 = r + r + r;
    data[idx] = r ^ (unsigned char)sum0;
    data[idx + 1] = g ^ (unsigned char)sum1;
    data[idx + 2] = b ^ (unsigned char)sum2;
}
```

Filter #171 - FlipAlphaBlend

Blends the current frame with a vertically-flipped version using alpha, creating a semi-transparent mirror reflection effect.

Technique: Vertical flip with alpha blend.

```
// Case 171 in unifiedFilterKernel:
processFlipAlphaBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFlipAlphaBlend(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int fx = width - x - 1;
    int fy = height - y - 1;
    if (fx < 0)
        fx = 0;
    if (fy < 0)
        fy = 0;
    int fidx = fy * step + fx * 4;
    float alpha = 0.5f + 0.3f * sinf(params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (1 - alpha) + data[fidx + j] * alpha);
    }
}
```


Filter #172 - RandomFlipFilter

Randomly decides per-pixel whether to show the normal or flipped pixel, creating a chaotic partial-mirror effect.

Technique: Random per-pixel flip selection.

```
// Case 172 in unifiedFilterKernel:
processRandomFlipFilter(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRandomFlipFilter(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int mode = (params.seed + x + y) % 3;
    int fx = (mode == 0 || mode == 2) ? (width - x - 1) : x;
    int fy = (mode == 1 || mode == 2) ? (height - y - 1) : y;
    if (fx < 0)
        fx = 0;
    if (fy < 0)
        fy = 0;
    int fidx = fy * step + fx * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)((data[idx + j] + data[fidx + j]) / 2);
    }
}
```

Filter #173 - SelfScaleByFrame

Scales pixel brightness by a factor derived from the frame counter, creating brightness that grows or shifts over time.

Technique: Frame-count-derived brightness scaling.

```
// Case 173 in unifiedFilterKernel:
processSelfScaleByFrame(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSelfScaleByFrame(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float scale = 1.0f + (params.frame_count % 50) * 0.02f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * scale);
    }
}
```

Filter #174 - AlphaBlendMirror

Alpha-blends the current pixel with its horizontally-mirrored counterpart, creating a translucent symmetrical overlay.

Technique: Horizontal mirror alpha blend.

```
// Case 174 in unifiedFilterKernel:
processAlphaBlendMirror(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processAlphaBlendMirror(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int mx = width - x - 1;
    int my = height - y - 1;
    if (mx < 0)
        mx = 0;
    if (my < 0)
        my = 0;
    int midx = my * step + mx * 4;
    float alpha = 0.5f + 0.3f * sinf(params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (1 - alpha) + data[midx + j] * alpha);
    }
}

```

Filter #175 - TwistedVision

Applies a sine/cosine-based coordinate twist to pixel sampling, creating a swirling, twisted distortion of the image.

Technique: Trigonometric coordinate twist distortion.

```

// Case 175 in unifiedFilterKernel:
processTwistedVision(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processTwistedVision(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = 2 + (params.frame_count % 20);
    int cx = x + offset;
    if (cx >= 0 && cx < width) {
        int cidx = y * step + cx * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= data[cidx + j];
        }
    }
}

```

Filter #176 - TruncateColor

Truncates color values to lower bit depths (e.g., multiples of 32 or 64), creating a posterized/quantized color look with fewer distinct colors.

Technique: Bit-depth color truncation/quantization.

```

// Case 176 in unifiedFilterKernel:
processTruncateColor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processTruncateColor(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int max_val = 128;
    for (int j = 0; j < 3; ++j) {
        if (data[idx + j] > max_val) {

```

```

        data[idx + j] = max_val;
    }
}

```

Filter #177 - TruncateVariable

Variable-strength color truncation that changes over time, creating an animated posterization that cycles between smooth and blocky color.

Technique: Animated variable-rate color quantization.

```

// Case 177 in unifiedFilterKernel:
processTruncateVariable(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processTruncateVariable(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int max_val = 150 + (int)(50.0f * sinf(params.frame_count * 0.05f));
    for (int j = 0; j < 3; ++j) {
        if (data[idx + j] > max_val) {
            data[idx + j] = max_val;
        }
    }
}

```

Filter #178 - TruncateVariableScale

Variable truncation with additional scaling, creating animated posterization with brightness modulation.

Technique: Scaled animated color quantization.

```

// Case 178 in unifiedFilterKernel:
processTruncateVariableScale(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processTruncateVariableScale(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int max_val = 75 + (int)(100.0f * sinf(params.frame_count * 0.05f));
    for (int j = 0; j < 3; ++j) {
        if (data[idx + j] > max_val) {
            data[idx + j] = data[idx + j] ^ max_val;
        }
    }
}

```

Filter #179 - XorFade

XOR operation that fades in and out over time-the XOR operand animates from 0 to max and back, creating a smooth digital distortion fade.

Technique: Sine-animated XOR fade.

```
// Case 179 in unifiedFilterKernel:
processXorFade(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processXorFade(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 0.1f + 0.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.01f));
    for (int j = 0; j < 3; ++j) {
        data[idx + j] ^= (unsigned char)(alpha * data[idx + j]);
    }
}
```

Filter #180 - SineValue

Applies the sine function to each pixel's color values, remapping brightness through a sinusoidal curve. Creates a wavy tone-mapping effect.

Technique: Sine-of-value tone remapping.

```
// Case 180 in unifiedFilterKernel:
processSineValue(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSineValue(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 0.5f * sinf(params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j) {
        int val = 1 + (int)(sinf(alpha) * (x + y));
        data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * (val % 256)));
    }
}
```

Filter #181 - FadeRtoGtoB

Gradually transitions emphasis from red to green to blue channels over time, creating a slow color wash that cycles through the primary colors.

Technique: Cycling primary channel emphasis.

```
// Case 181 in unifiedFilterKernel:
processFadeRtoGtoB(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFadeRtoGtoB(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int colors[3];
    colors[0] = (params.sumB + params.frame_count) % 256;
    colors[1] = (params.sumG + params.frame_count * 2) % 256;
    colors[2] = (params.sumR + params.frame_count * 3) % 256;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)((data[idx + j] * 0.5f) + (0.5f * colors[j]));
    }
}
```

Filter #182 - FadeRandomChannel

Fades a randomly selected channel in and out each cycle, creating unpredictable per-channel brightness pulsing.

Technique: Random channel selection with fade.

```
// Case 182 in unifiedFilterKernel:
processFadeRandomChannel(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFadeRandomChannel(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int rgb = (params.seed + params.frame_count) % 3;
    int color = (params.sumB + params.sumG + params.sumR + params.frame_count) % 256;
    data[idx + rgb] = (unsigned char)((data[idx + rgb] * 0.5f) + (0.5f * color));
}
```

Filter #183 - VariableLines

Draws horizontal lines with variable spacing and color, creating an animated line-pattern overlay.

Technique: Variable-pitch horizontal line rendering.

```
// Case 183 in unifiedFilterKernel:
processVariableLines(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processVariableLines(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = (params.seed + y * 17) % width;
    int src_x = (x + offset) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)((0.7f * data[idx + j]) + (0.3f * data[src_idx + j]));
    }
}
```

Filter #184 - VariableLinesVertical

Same as VariableLines but vertical, creating animated vertical line-pattern overlay.

Technique: Variable-pitch vertical line rendering.

```
// Case 184 in unifiedFilterKernel:
processVariableLinesVertical(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processVariableLinesVertical(int x, int y, unsigned char *data, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = (params.seed + x * 13) % height;
    int src_y = (y + offset) % height;
    int src_idx = src_y * step + x * 4;
}
```

```

    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)((0.7f * data[idx + j]) + (0.3f * data[src_idx + j]));
    }
}

```

Filter #185 - RowMedianBlend

Computes median/average values along each row from frame history, then blends. Creates horizontally-coherent temporal smoothing.

Technique: Row-wise temporal median blending.

```

// Case 185 in unifiedFilterKernel:
processRowMedianBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processRowMedianBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int fidx = (y + params.frame_count) % params.numFrames;
    unsigned char *hist = allFrames[fidx];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[idx + j]));
        }
    }
}

```

Filter #186 - MirrorReverseColorBlend

Mirrors the image and reverses color channels while blending with the original, creating a blended symmetric false-color effect.

Technique: Mirror + channel reversal + alpha blend.

```

// Case 186 in unifiedFilterKernel:
processMirrorReverseColor(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processMirrorReverseColor(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int px = width - x - 1;
    int py = height - y - 1;
    if (px < 0)
        px = 0;
    if (py < 0)
        py = 0;
    int idx0 = py * step + x * 4;
    int idx1 = py * step + px * 4;
    int idx2 = y * step + px * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + (2 - j)] = (unsigned char)((data[idx + j] * 0.25f) + (data[idx0 + j] * 0.25f) + (data
            ↪ [idx1 + j] * 0.25f) + (data[idx2 + j] * 0.25f));
    }
}

```

Filter #187 - PsychoticVision

Extreme animated color distortion combining multiple techniques-channel swaps, XOR, and sine modulation-for a chaotic psychedelic look.

Technique: Multi-technique combined per-channel distortion.

```
// Case 187 in unifiedFilterKernel:
processPsychoticVision(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPsychoticVision(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 0.1f + 0.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.1f));
    int speed = 1 + (params.frame_count % 5);
    for (int j = 0; j < 3; ++j) {
        int col = (params.sumB + j * 50 + params.frame_count * speed) % 200;
        data[idx + j] = (unsigned char)((0.7f * data[idx + j]) - (0.3f * alpha * col));
    }
}
```

Filter #188 - PixelGlitch

Randomly replaces some pixels with offset-sourced or corrupted values, simulating digital glitch artifacts that flicker and scatter.

Technique: Random pixel displacement/corruption.

```
// Case 188 in unifiedFilterKernel:
processPixelGlitch(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPixelGlitch(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.1f) {
        int glitch = (int)(gpu_rand(x + 1, y + 1, params.seed) * 255);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= glitch;
        }
    }
}
```

Filter #189 - StaticGlitch

Adds random static noise blocks that move around the frame, simulating television static interference.

Technique: Random-block noise overlay.

```
// Case 189 in unifiedFilterKernel:
processStaticGlitch(x, y, data, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processStaticGlitch(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    if (r < 0.05f) {
        int noise = (int)(r * 255 * 10) % 256;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = noise;
        }
    }
}

```

Filter #190 - WavePattern

Applies a sine-wave pattern to pixel brightness, creating undulating wave-shaped brightness bands across the image.

Technique: Sine-wave brightness modulation pattern.

```

// Case 190 in unifiedFilterKernel:
processWavePattern(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processWavePattern(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf((x + params.frame_count) * 0.05f) * 30.0f;
    int src_y = (int)(y + wave) % width;
    if (src_y < 0)
        src_y = 0;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + fabs(wave));
    }
}

```

Filter #191 - WavePatternXor

Wave brightness pattern combined with XOR operations, creating undulating bands of digital distortion.

Technique: Sine-wave XOR pattern.

```

// Case 191 in unifiedFilterKernel:
processWavePatternXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processWavePatternXor(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float wave = sinf((x + y + params.frame_count) * 0.05f) * 50.0f;
    int wave_int = (int)fabs(wave) % 256;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] ^= wave_int;
    }
}

```


Filter #192 - DiagonalXor

XOR applied along diagonal axes, creating diagonal stripe-shaped digital artifacts.

Technique: Diagonal position-based XOR.

```
// Case 192 in unifiedFilterKernel:
processDiagonalXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDiagonalXor(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int diag = (x + y + params.frame_count) % 256;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] ^= diag;
    }
}
```

Filter #193 - RGBShiftBlend

Shifts each R, G, B channel to different spatial positions and blends, creating a chromatic aberration/prism-split effect with blending.

Technique: Per-channel spatial offset with blend.

```
// Case 193 in unifiedFilterKernel:
processRGBShiftBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processRGBShiftBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int shift = params.frame_count % 3;
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    switch (shift) {
    case 0:
        data[idx] = g;
        data[idx + 1] = r;
        data[idx + 2] = b;
        break;
    case 1:
        data[idx] = r;
        data[idx + 1] = b;
        data[idx + 2] = g;
        break;
    case 2:
        data[idx] = (b + g) / 2;
        data[idx + 1] = (g + r) / 2;
        data[idx + 2] = (r + b) / 2;
        break;
    }
}
```

Filter #194 - ChannelShuffle

Randomly permutes which color channel is assigned to which output channel, creating false-color images.

Technique: Random channel permutation.

```
// Case 194 in unifiedFilterKernel:
processChannelShuffle(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processChannelShuffle(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int pos = (x + y) % 3;
    unsigned char temp = data[idx + pos];
    data[idx + pos] = data[idx + (pos + 1) % 3];
    data[idx + (pos + 1) % 3] = temp;
}
```

Filter #195 - ChannelShuffleRand

Randomized channel shuffle that changes per pixel/frame, creating a more chaotic false-color mosaic.

Technique: Per-frame random channel reassignment.

```
// Case 195 in unifiedFilterKernel:
processChannelShuffleRand(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processChannelShuffleRand(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    int pos = (int)(r * 3) % 3;
    unsigned char temp = data[idx + pos];
    data[idx + pos] = data[idx + (pos + 1) % 3];
    data[idx + (pos + 1) % 3] = temp;
}
```

Filter #196 - PixelCounter

Uses a running pixel counter to modify color values, creating position-dependent progressive color shifts.

Technique: Sequential pixel counter color modification.

```
// Case 196 in unifiedFilterKernel:
processPixelCounter(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPixelCounter(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int count = (x + y + params.frame_count) % 256;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + count * 0.3f);
    }
}
```

```
}  
}
```

Filter #197 - PixelCounterXor

Pixel counter combined with XOR, creating position-dependent digital distortion patterns.

Technique: Sequential pixel counter XOR.

```
// Case 197 in unifiedFilterKernel:  
processPixelCounterXor(x, y, data, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processPixelCounterXor(int x, int y, unsigned char *data, size_t step, const  
    ↪ FilterParams &params) {  
    int idx = y * step + x * 4;  
    int count = (x + y + params.frame_count) % 256;  
    for (int j = 0; j < 3; ++j) {  
        data[idx + j] ^= count;  
    }  
}
```

Filter #198 - RowColorBlend

Blends each row with a different animated color, creating horizontal color bands.

Technique: Per-row animated color blending.

```
// Case 198 in unifiedFilterKernel:  
processRowColorBlend(x, y, data, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processRowColorBlend(int x, int y, unsigned char *data, size_t step, const FilterParams  
    ↪ &params) {  
    int idx = y * step + x * 4;  
    int color = (y + params.frame_count) % 256;  
    int channel = y % 3;  
    data[idx + channel] = (unsigned char)((data[idx + channel] * 0.5f) + (color * 0.5f));  
}
```

Filter #199 - ColumnColorBlend

Blends each column with a different animated color, creating vertical color bands.

Technique: Per-column animated color blending.

```
// Case 199 in unifiedFilterKernel:  
processColumnColorBlend(x, y, data, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processColumnColorBlend(int x, int y, unsigned char *data, size_t step, const  
    ↪ FilterParams &params) {  
    int idx = y * step + x * 4;  
    int color = (x + params.frame_count) % 256;  
    int channel = x % 3;
```

```

    data[idx + channel] = (unsigned char)((data[idx + channel] * 0.5f) + (color * 0.5f));
}

```

Spatial & Pattern (200-299)

Filter #200 - CheckerboardXor

XOR applied in a checkerboard pattern-even squares XOR with one value, odd squares with another. Creates a tiled digital distortion grid.

Technique: Checkerboard-masked XOR.

```

// Case 200 in unifiedFilterKernel:
processCheckerboardXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCheckerboardXor(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int size = 8 + (params.frame_count % 24);
    int check = ((x / size) + (y / size)) % 2;
    if (check) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= 128;
        }
    }
}

```

Filter #201 - CheckerboardBlend

Blends current and history frames in a checkerboard pattern-alternating tiles show different time points.

Technique: Checkerboard-masked temporal blending.

```

// Case 201 in unifiedFilterKernel:
processCheckerboardBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processCheckerboardBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size = 16 + (params.frame_count % 32);
    int check = ((x / size) + (y / size)) % 2;
    unsigned char *hist = allFrames[params.frame_count % params.numFrames];
    if (check && hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[idx + j]));
        }
    }
}

```

Filter #202 - SineWaveDistort

Displaces pixel positions using a sine wave, creating horizontal wavy distortion like looking through rippled glass.

Technique: Sine-modulated horizontal coordinate displacement.

```
// Case 202 in unifiedFilterKernel:
processSineWaveDistort(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processSineWaveDistort(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf(y * 0.1f + params.frame_count * 0.1f) * 10.0f;
    int src_x = ((int)(x + wave) % width + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)((0.6f * data[idx + j]) + (0.4f * data[src_idx + j]));
    }
}
```

Filter #203 - CosineWaveDistort

Displaces pixels using a cosine wave, creating vertical wavy distortion complementary to SineWaveDistort.

Technique: Cosine-modulated vertical coordinate displacement.

```
// Case 203 in unifiedFilterKernel:
processCosineWaveDistort(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processCosineWaveDistort(int x, int y, unsigned char *data, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = cosf(x * 0.1f + params.frame_count * 0.1f) * 10.0f;
    int src_y = ((int)(y + wave) % height + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)((0.6f * data[idx + j]) + (0.4f * data[src_idx + j]));
    }
}
```

Filter #204 - SinCosBlend

Combines sine and cosine wave blending of color channels, creating smooth oscillating color transitions.

Technique: Combined sine/cosine per-channel blending.

```
// Case 204 in unifiedFilterKernel:
processSinCosBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSinCosBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float sinVal = sinf((x + params.frame_count) * 0.05f) * 50.0f;
```

```

float cosVal = cosf((y + params.frame_count) * 0.05f) * 50.0f;
data[idx] = (unsigned char)fminf(255.0f, fmaxf(0.0f, data[idx] + sinVal));
data[idx + 1] = (unsigned char)fminf(255.0f, fmaxf(0.0f, data[idx + 1] + cosVal));
data[idx + 2] = (unsigned char)fminf(255.0f, fmaxf(0.0f, data[idx + 2] + (sinVal + cosVal) / 2));
}

```

Filter #205 - PixelReverseXor

XORs pixels with their reverse-ordered counterpart (reading the frame backward), creating mirrored digital artifacts.

Technique: Reverse-position pixel XOR.

```

// Case 205 in unifiedFilterKernel:
processPixelReverseXor(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPixelReverseXor(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int rx = width - x - 1;
    int ry = height - y - 1;
    if (rx < 0)
        rx = 0;
    if (ry < 0)
        ry = 0;
    int ridx = ry * step + rx * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] ^= data[ridx + j];
    }
}

```

Filter #206 - LinesAcrossX

Draws horizontal lines by replacing certain rows with data from frame history across the x-axis.

Technique: Row-based history frame line replacement.

```

// Case 206 in unifiedFilterKernel:
processLinesAcrossX(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processLinesAcrossX(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0) {
        int prev_idx = y * step + (x - 1) * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= data[prev_idx + j];
        }
    }
}

```

Filter #207 - XorLineX

XOR applied on specific horizontal lines/rows, creating periodic horizontal digital distortion bands.

Technique: Row-selective XOR.

```
// Case 207 in unifiedFilterKernel:
processXorLineX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processXorLineX(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int avg = (params.sumR + params.sumG + params.sumB) / 3;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)((0.5f * (data[idx + j] ^ avg)) + (0.5f * data[idx + j]));
    }
}
```

Filter #208 - AlphaComponentIncrease

Progressively increases alpha-blending intensity over time, creating a gradually intensifying blend effect.

Technique: Time-increasing alpha blend.

```
// Case 208 in unifiedFilterKernel:
processAlphaComponentIncrease(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAlphaComponentIncrease(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int channel = params.frame_count % 3;
    float alpha = 0.1f + 0.9f * fabsf(sinf(params.frame_count * 0.01f));
    data[idx + channel] ^= (unsigned char)(alpha * data[idx + channel]);
}
```

Filter #209 - ExpandContract

Expands and contracts the image from the center using animated scaling, like a breathing zoom effect.

Technique: Animated center-point scaling.

```
// Case 209 in unifiedFilterKernel:
processExpandContract(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processExpandContract(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int nw = width + (int)(sinf(params.frame_count * 0.02f) * width);
    if (nw < width)
        nw = width;
    int src_x = (x * nw) / width;
    if (src_x >= 0 && src_x < width) {
```

```

        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}

```

Filter #210 - LongLines

Creates long continuous horizontal lines using frame history, with lines extending across the full frame width.

Technique: Full-width horizontal history line replacement.

```

// Case 210 in unifiedFilterKernel:
processLongLines(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processLongLines(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, 0, params.seed);
    if (r < 0.007f) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = 0;
        }
    }
}

```

Filter #211 - TearRight

Tears the image rightward-horizontal slices shift right by increasing amounts, creating a rightward shearing/tearing distortion.

Technique: Row-based rightward pixel displacement.

```

// Case 211 in unifiedFilterKernel:
processTearRight(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processTearRight(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int max_off = 50 + (params.frame_count % 350);
    float r = gpu_rand(y, params.seed, params.frame_count);
    int offset = 25 + (int)(r * max_off);
    int total_size = width + offset;
    int src_x = (x * total_size) / width;
    if (src_x >= width)
        src_x = width - 1;
    if (src_x < 0)
        src_x = 0;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}

```

Filter #212 - TearDown

Tears the image downward-vertical columns shift down by increasing amounts, creating a downward shearing distortion.

Technique: Column-based downward pixel displacement.

```
// Case 212 in unifiedFilterKernel:
processTearDown(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processTearDown(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int max_off = 50 + (params.frame_count % 350);
    float r = gpu_rand(x, params.seed, params.frame_count);
    int offset = 25 + (int)(r * max_off);
    int total_size = height + offset;
    int src_y = (y * total_size) / height;
    if (src_y >= height)
        src_y = height - 1;
    if (src_y < 0)
        src_y = 0;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #213 - DistortionByRow

Each row shifts horizontally by a different animated amount, creating wavy row-by-row horizontal distortion.

Technique: Per-row variable horizontal displacement.

```
// Case 213 in unifiedFilterKernel:
processDistortionByRow(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processDistortionByRow(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int nw = height / 16 + (y % (height - height / 16));
    int src_y = (y * nw) / height;
    if (src_y >= height)
        src_y = height - 1;
    if (src_y < 0)
        src_y = 0;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #214 - DistortionByCol

Each column shifts vertically by a different animated amount, creating wavy column-by-column vertical distortion.

Technique: Per-column variable vertical displacement.

```
// Case 214 in unifiedFilterKernel:
processDistortionByCol(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processDistortionByCol(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int nh = width / 16 + (x % (width - width / 16));
    int src_x = (x * nh) / width;
    if (src_x >= width)
        src_x = width - 1;
    if (src_x < 0)
        src_x = 0;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #215 - AlternateAlpha

Alternates between two different alpha-blend settings on different frames or pixel groups, creating a dual-mode blending strobe.

Technique: Alternating alpha blend modes.

```
// Case 215 in unifiedFilterKernel:
processAlternateAlpha(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAlternateAlpha(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int rgb = params.frame_count % 3;
    float alpha = 0.1f + 0.4f * fabsf(sinf(params.frame_count * 0.01f));
    data[idx + rgb] = (unsigned char)((data[idx + rgb] * 0.5f) * alpha);
}
```

Filter #216 - DiagSquareRGB

Diagonal square blocks with per-channel color selection from frame history. Creates a diagonal tiled mosaic with RGB separation.

Technique: Diagonal block layout with per-channel history sampling.

```
// Case 216 in unifiedFilterKernel:
processDiagSquareRGB(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDiagSquareRGB(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
```

```

    int idx = y * step + x * 4;
    int rgb = params.frame_count % 3;
    int sq_x = x / 32;
    int sq_y = y / 32;
    int offset = (sq_x + sq_y) % params.numFrames;
    unsigned char *hist = allFrames[offset];
    if (hist) {
        data[idx + rgb] = hist[idx + rgb];
    }
}

```

Filter #217 - ShiftPixelsRGB

Shifts each R, G, B channel by different pixel offsets, creating a per-channel spatial displacement/chromatic aberration.

Technique: Per-channel pixel position shifting.

```

// Case 217 in unifiedFilterKernel:
processShiftPixelsRGB(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processShiftPixelsRGB(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int rgb = params.frame_count % 3;
    int offset = 1 + (params.frame_count % (width / 16));
    int src_x = (x + offset) % width;
    int src_idx = y * step + src_x * 4;
    data[idx + rgb] += data[src_idx + rgb];
}

```

Filter #218 - ColorWaveTrailsRGB

Trails with per-channel wave-modulated offsets-each R, G, B channel follows a different wave pattern through frame history. Creates vivid chromatic trail separations.

Technique: Per-channel sine-wave frame history offset.

```

// Case 218 in unifiedFilterKernel:
processColorWaveTrailsRGB(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorWaveTrailsRGB(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int rgb = params.frame_count % 3;
    int fi = y % params.numFrames;
    unsigned char *hist = allFrames[fi];
    if (hist) {
        data[idx + rgb] = (unsigned char)((0.3f * data[idx + rgb]) + (0.7f * hist[idx + rgb]));
    }
}

```

Filter #219 - ProperTrails

Clean motion trails built by averaging the current frame with multiple history frames at graduated weights. Creates smooth, natural-looking motion blur trails.

Technique: Weighted multi-frame temporal trailing.

```
// Case 219 in unifiedFilterKernel:
processProperTrails(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processProperTrails(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *hist = allFrames[params.numFrames > 7 ? 7 : params.numFrames - 1];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[idx + j]));
        }
    }
}
```

Filter #220 - XorLag

XORs the current pixel with a frame from a few steps back in history, creating a lagged digital distortion that highlights what changed recently.

Technique: Time-lagged frame XOR.

```
// Case 220 in unifiedFilterKernel:
processXorLag(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processXorLag(int x, int y, unsigned char *data, unsigned char **allFrames, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int fi = params.numFrames > 31 ? 31 : params.numFrames - 1;
    unsigned char *hist = allFrames[fi];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= hist[idx + j];
        }
    }
}
```

Filter #221 - PixelateBlend

Pixelates the image into blocks and blends with frame history. Combines a mosaic/pixelation look with temporal ghosting.

Technique: Block pixelation + history frame blending.

```
// Case 221 in unifiedFilterKernel:
processPixelateBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processPixelateBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
```

```

    int idx = y * step + x * 4;
    int wait = 1 + ((x + y + params.seed) % 50);
    int fi = ((x * params.sh + y) / wait) % params.numFrames;
    unsigned char *hist = allFrames[fi];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[idx + j]));
        }
    }
}

```

Filter #222 - DiagPixel

Samples pixels along diagonal lines, replacing each pixel with one found along its diagonal. Creates diagonal smear/streak artifacts.

Technique: Diagonal pixel position remapping.

```

// Case 222 in unifiedFilterKernel:
processDiagPixel(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDiagPixel(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = y % params.numFrames;
    unsigned char *hist = allFrames[off];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[idx + j]));
        }
    }
}

```

Filter #223 - DiagPixelY

Diagonal pixel remapping along the Y axis variant, creating differently-angled diagonal streaks.

Technique: Y-biased diagonal position remapping.

```

// Case 223 in unifiedFilterKernel:
processDiagPixelY(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDiagPixelY(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = (x + y) % params.numFrames;
    unsigned char *hist = allFrames[off];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = hist[idx + j];
        }
    }
}

```

Filter #224 - ExpandLeftRight

Stretches the image outward from the center, expanding left and right halves away from the midline. Creates a mirror-expand spread effect.

Technique: Center-outward horizontal expansion.

```
// Case 224 in unifiedFilterKernel:
processExpandLeftRight(x, y, data, allFrames, width, step, params);

// Referenced __device__ function(s):
__device__ void processExpandLeftRight(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = (params.frame_count + y) % width;
    int fi = y % params.numFrames;
    unsigned char *hist = allFrames[fi];
    if (hist) {
        int src_x = (x + off) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * hist[src_idx + j]));
        }
    }
}
```

Filter #225 - DiagSquare

Diagonal square block pattern applied to the image, grouping pixels into diagonal tiles.

Technique: Diagonal square block grouping.

```
// Case 225 in unifiedFilterKernel:
processDiagSquare(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDiagSquare(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq_x = x / 32;
    int sq_y = y / 32;
    int offset = (sq_x + sq_y) % params.numFrames;
    unsigned char *hist = allFrames[offset];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = hist[idx + j];
        }
    }
}
```

Filter #226 - HorizontalColorOffset

Offsets each color channel horizontally by different amounts, creating a horizontal chromatic spread/aberration.

Technique: Per-channel horizontal offset.

```
// Case 226 in unifiedFilterKernel:
```

```

processHorizontalColorOffset(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processHorizontalColorOffset(int x, int y, unsigned char *data, int width, size_t step,
    ⇨ const FilterParams &params) {
    int idx = y * step + x * 4;
    int coffset = (y / 50) % 3;
    float color = 0.3f + 0.7f * fabsf(sinf((x + params.frame_count) * 0.01f));
    data[idx + coffset] = (unsigned char)(data[idx + coffset] * color);
}

```

Filter #227 - PrevFrameNotEqual

Replaces pixels only where the current frame differs from the previous frame, creating a motion-only display where static areas freeze.

Technique: Frame-difference conditional replacement.

```

// Case 227 in unifiedFilterKernel:
processPrevFrameNotEqual(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processPrevFrameNotEqual(int x, int y, unsigned char *data, unsigned char **allFrames,
    ⇨ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        int threshold = 25;
        for (int j = 0; j < 3; ++j) {
            if (data[idx + j] >= prev[idx + j] && data[idx + j] <= prev[idx + j] + threshold) {
                data[idx + j] = prev[idx + j];
            }
        }
    }
}

```

Filter #228 - BlackLines

Draws black horizontal lines at regular intervals, creating a scan-line/venetian-blind overlay.

Technique: Row-interval black line rendering.

```

// Case 228 in unifiedFilterKernel:
processBlackLines(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processBlackLines(int x, int y, unsigned char *data, int width, size_t step, const
    ⇨ FilterParams &params) {
    int idx = y * step + x * 4;
    int depth = params.seed % 50;
    bool on = ((y / 50) % 2) == 0;
    if (on && x < depth) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = 0;
        }
    } else if (on && x + depth < width) {
        int src_idx = y * step + (x + depth) * 4;

```

```

        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * data[src_idx + j]));
        }
    }
}

```

Filter #229 - DizzyMode

Averages multiple history frames together, creating a heavily blurred temporal composite that looks dizzy and smeared.

Technique: Multi-frame temporal averaging.

```

// Case 229 in unifiedFilterKernel:
processDizzyMode(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDizzyMode(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((0.5f * data[idx + j]) + (0.5f * prev[idx + j]));
        }
    }
}

```

Filter #230 - GhostShift

Shifts ghost images from frame history horizontally or vertically over time, creating moving transparent afterimages.

Technique: Animated-offset history frame ghosting.

```

// Case 230 in unifiedFilterKernel:
processGhostShift(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processGhostShift(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for (int i = 0; i < params.numFrames && i < 8; ++i) {
        unsigned char *hist = allFrames[i];
        if (hist) {
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = (unsigned char)((0.6f * data[idx + j]) + (0.4f * hist[idx + j]));
            }
        }
    }
}

```


Filter #231 - RGBSplitFilter

Splits the image into separate R, G, B components displayed in different spatial positions, creating a prism-like color separation.

Technique: Spatially-separated RGB channel display.

```
// Case 231 in unifiedFilterKernel:
processRGBSplitFilter(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processRGBSplitFilter(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int f0 = params.numFrames > 15 ? 15 : params.numFrames - 1;
    int f1 = params.numFrames > 7 ? 7 : params.numFrames / 2;
    int f2 = 0;
    unsigned char *rgb0 = allFrames[f0];
    unsigned char *rgb1 = allFrames[f1];
    unsigned char *rgb2 = allFrames[f2];
    if (rgb0)
        data[idx] = rgb0[idx];
    if (rgb1)
        data[idx + 1] = rgb1[idx + 1];
    if (rgb2)
        data[idx + 2] = rgb2[idx + 2];
}
```

Filter #232 - PixelateRect

Combines rectangular blocks of pixels into single colors (block averaging), creating a classic mosaic/pixelation effect.

Technique: Rectangular block color averaging.

```
// Case 232 in unifiedFilterKernel:
processPixelateRect(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processPixelateRect(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.1f) {
        int fi = (int)(r * params.numFrames * 10) % params.numFrames;
        unsigned char *hist = allFrames[fi];
        if (hist) {
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = hist[idx + j];
            }
        }
    }
}
```

Filter #233 - CollectionXor4

XORs the current pixel with four different history frames in sequence, layering multiple XOR operations for complex digital artifacts.

Technique: Quad-frame sequential XOR.

```
// Case 233 in unifiedFilterKernel:
processCollectionXor4(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processCollectionXor4(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 0.1f + 0.9f * fabsf(sinf(params.frame_count * 0.01f));
    int colors[3] = {0, 0, 0};
    int count = params.numFrames < 4 ? params.numFrames : 4;
    for (int q = 0; q < count; ++q) {
        unsigned char *hist = allFrames[q];
        if (hist) {
            for (int j = 0; j < 3; ++j) {
                colors[j] += hist[idx + j];
            }
        }
    }
    for (int j = 0; j < 3; ++j) {
        int avg_color = colors[j] / count;
        int pix_val = data[idx + j] ^ avg_color;
        data[idx + j] = (unsigned char)((1.0f - alpha) * data[idx + j] + alpha * pix_val);
    }
}
```

Filter #234 - RectangleSpin

Rotates/spins rectangular regions of the image at different rates, creating a spinning tile mosaic.

Technique: Per-rectangle rotation animation.

```
// Case 234 in unifiedFilterKernel:
processRectangleSpin(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRectangleSpin(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.05f) {
        int src_y = (int)(r * height * 20) % height;
        int src_idx = src_y * step + x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}
```

Filter #235 - RectanglePlotXY

Renders rectangles at positions derived from x/y coordinates, creating a geometric grid plot overlay.

Technique: Coordinate-derived rectangle rendering.

```
// Case 235 in unifiedFilterKernel:
processRectanglePlotXY(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRectanglePlotXY(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.1f) {
        int src_x = (int)(r * width * 10) % width;
        int src_y = (int)(r * height * 10) % height;
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= data[src_idx + j];
        }
    }
}
```

Filter #236 - ShiftLinesDown

Shifts alternating horizontal lines downward by different amounts, creating a cascading downward slip distortion.

Technique: Alternating row downward shift.

```
// Case 236 in unifiedFilterKernel:
processShiftLinesDown(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processShiftLinesDown(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int shiftAmount = (params.frame_count / 4) % height;
    int src_y = (y + shiftAmount) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
    }
}
```

Filter #237 - PictureStretch

Stretches the image from the center or edges, creating a rubber-like distortion effect.

Technique: Position-scaled coordinate stretching.

```
// Case 237 in unifiedFilterKernel:
processPictureStretch(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPictureStretch(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float stretch = 1.0f + 0.5f * sinf(params.frame_count * 0.02f);
    int src_x = (int)(x / stretch) % width;
    if (src_x < 0)
        src_x += width;
}
```

```

    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}

```

Filter #238 - PictureStretchPieces

Stretches the image in separate pieces/segments, each with different stretch amounts, creating a fragmented stretch distortion.

Technique: Segmented variable stretching.

```

// Case 238 in unifiedFilterKernel:
processPictureStretchPieces(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPictureStretchPieces(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int piece = y / 32;
    float stretch = 1.0f + 0.3f * sinf((params.frame_count + piece * 10) * 0.03f);
    int src_x = (int)(x / stretch) % width;
    if (src_x < 0)
        src_x += width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}

```

Filter #239 - VisualSnow

Adds fine-grained random noise (visual snow) to the image, simulating the "visual snow" neurological phenomenon.

Technique: Fine per-pixel random noise addition.

```

// Case 239 in unifiedFilterKernel:
processVisualSnow(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processVisualSnow(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.1f) {
        unsigned char snow = (unsigned char)(r * 255 * 10);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(0.7f * data[idx + j] + 0.3f * snow);
        }
    }
}

```

Filter #240 - VisualSnowX2

Double-intensity visual snow with more aggressive noise, creating heavier static interference.

Technique: Doubled per-pixel random noise.

```
// Case 240 in unifiedFilterKernel:
processVisualSnowX2(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processVisualSnowX2(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.2f) {
        unsigned char snow = (unsigned char)(r * 255 * 5);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * snow);
        }
    }
}
```

Filter #241 - LineGlitch

Randomly glitches entire horizontal lines, replacing them with displaced or corrupted data.

Technique: Random row glitch/displacement.

```
// Case 241 in unifiedFilterKernel:
processLineGlitch(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processLineGlitch(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 32, y, params.seed);
    if (r < 0.15f) {
        int offset = (int)(r * 100) % 50;
        int src_x = (x + offset) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}
```

Filter #242 - SlitReverse64

Reverses pixel order within 64-pixel slits/segments, creating a segmented mirror effect.

Technique: 64-pixel segment order reversal.

```
// Case 242 in unifiedFilterKernel:
processSlitReverse64(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processSlitReverse64(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
```

```

    int idx = y * step + x * 4;
    int slit = y / 64;
    if (slit % 2 == 0) {
        int src_x = width - 1 - x;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
        }
    }
}

```

Filter #243 - SlitReverse64_Increase

Slit reversal with increasing slit sizes over time, creating an evolving segmented mirror pattern.

Technique: Animated-size segment reversal.

```

// Case 243 in unifiedFilterKernel:
processSlitReverse64_Increase(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processSlitReverse64_Increase(int x, int y, unsigned char *data, int width, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int slitSize = 32 + (params.frame_count % 64);
    int slit = y / slitSize;
    if (slit % 2 == 0) {
        int src_x = width - 1 - x;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
        }
    }
}

```

Filter #244 - SlitStretch

Stretches pixels within slit segments, creating elongated repeated pixel bands.

Technique: Segment-internal pixel stretching.

```

// Case 244 in unifiedFilterKernel:
processSlitStretch(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processSlitStretch(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int slit = y / 32;
    float stretch = 1.0f + 0.5f * (slit % 4) * 0.25f;
    int src_x = (int)(x * stretch) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}

```

Filter #245 - LineLeftRight

Shifts alternating rows left and right by animated amounts, creating a horizontal shaking/jitter distortion.

Technique: Alternating-row horizontal oscillation.

```
// Case 245 in unifiedFilterKernel:
processLineLeftRight(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processLineLeftRight(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int dir = (y / 16) % 2;
    int offset = params.frame_count % width;
    int src_x = dir == 0 ? (x + offset) % width : (x - offset + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #246 - LineLeftRightResize

LineLeftRight with animated line height changes, creating a shaking distortion with growing/shrinking row bands.

Technique: Animated-height alternating row oscillation.

```
// Case 246 in unifiedFilterKernel:
processLineLeftRightResize(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processLineLeftRightResize(int x, int y, unsigned char *data, int width, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int lineSize = 8 + (params.frame_count / 8) % 32;
    int dir = (y / lineSize) % 2;
    int offset = params.frame_count % width;
    int src_x = dir == 0 ? (x + offset) % width : (x - offset + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #247 - RGBLineTrails

Horizontal line trails where each R, G, B channel trails independently on different rows, creating a chromatic line trail pattern.

Technique: Per-channel row-based temporal trailing.

```
// Case 247 in unifiedFilterKernel:
processRGBLineTrails(x, y, data, allFrames, step, params);
```

```

// Referenced __device__ function(s):
__device__ void processRGBLineTrails(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int channel = y % 3;
    unsigned char *prev = allFrames[0];
    if (prev) {
        data[idx + channel] = (unsigned char)(0.6f * data[idx + channel] + 0.4f * prev[idx + channel]);
    }
}

```

Filter #248 - RGBCollectionBlend

Collects each R, G, B channel from different history frames and blends them together, creating a temporal chromatic composite.

Technique: Per-channel multi-frame collection blend.

```

// Case 248 in unifiedFilterKernel:
processRGBCollectionBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processRGBCollectionBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        int fi = j % params.numFrames;
        unsigned char *hist = allFrames[fi];
        if (hist) {
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * hist[idx + j]);
        }
    }
}

```

Filter #249 - RGBCollectionIncrease

RGB collection with increasing intensity over successive frames, building up channel separation over time.

Technique: Accumulating per-channel history collection.

```

// Case 249 in unifiedFilterKernel:
processRGBCollectionIncrease(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processRGBCollectionIncrease(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 0.1f + 0.9f * fabsf(sinf(params.frame_count * 0.02f));
    for (int j = 0; j < 3; ++j) {
        int fi = (j + params.frame_count) % params.numFrames;
        unsigned char *hist = allFrames[fi];
        if (hist) {
            data[idx + j] = (unsigned char)((1.0f - alpha) * data[idx + j] + alpha * hist[idx + j]);
        }
    }
}

```



```
}  
}
```

Filter #250 - RGBLongTrails

Extended R, G, B channel trails spanning many history frames, creating long vivid chromatic motion trails.

Technique: Long-range per-channel frame trailing.

```
// Case 250 in unifiedFilterKernel:  
processRGBLongTrails(x, y, data, allFrames, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processRGBLongTrails(int x, int y, unsigned char *data, unsigned char **allFrames,  
    ↪ size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    for (int j = 0; j < 3; ++j) {  
        int fi = (j * 5) % params.numFrames;  
        unsigned char *hist = allFrames[fi];  
        if (hist) {  
            data[idx + j] = (unsigned char)(0.3f * data[idx + j] + 0.7f * hist[idx + j]);  
        }  
    }  
}
```

Filter #251 - FadeRGB_Speed

Fades each R, G, B channel at different speeds, creating a desynchronized color fade effect.

Technique: Per-channel variable-rate fade.

```
// Case 251 in unifiedFilterKernel:  
processFadeRGB_Speed(x, y, data, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processFadeRGB_Speed(int x, int y, unsigned char *data, size_t step, const FilterParams  
    ↪ &params) {  
    int idx = y * step + x * 4;  
    float speedR = fabsf(sinf(params.frame_count * 0.05f));  
    float speedG = fabsf(sinf(params.frame_count * 0.07f));  
    float speedB = fabsf(sinf(params.frame_count * 0.09f));  
    data[idx] = (unsigned char)(data[idx] * speedB);  
    data[idx + 1] = (unsigned char)(data[idx + 1] * speedG);  
    data[idx + 2] = (unsigned char)(data[idx + 2] * speedR);  
}
```

Filter #252 - RGBStrobeTrails

Combines RGB-separated trails with strobing visibility, creating pulsing chromatic trails.

Technique: Strobed per-channel trailing.

```
// Case 252 in unifiedFilterKernel:  
processRGBStrobeTrails(x, y, data, allFrames, step, params);
```

```
// Referenced __device__ function(s):
__device__ void processRGBStrobeTrails(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int strobe = params.frame_count % 3;
    unsigned char *prev = allFrames[0];
    if (prev) {
        data[idx + strobe] = (unsigned char)(0.5f * data[idx + strobe] + 0.5f * prev[idx + strobe]);
    }
}
```

Filter #253 - BoxGlitch

Randomly displaces rectangular blocks of pixels, creating a classic digital "box glitch" artifact.

Technique: Random rectangular block displacement.

```
// Case 253 in unifiedFilterKernel:
processBoxGlitch(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBoxGlitch(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 64, y / 64, params.seed);
    if (r < 0.1f) {
        int offset_x = (int)(r * 100) % 32;
        int offset_y = (int)(r * 50) % 16;
        int src_x = (x + offset_x) % width;
        int src_y = (y + offset_y) % height;
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}
```

Filter #254 - VerticalPictureDistort

Distorts the image with vertical displacement that varies across the frame, creating a wavy vertical stretch.

Technique: Variable vertical position distortion.

```
// Case 254 in unifiedFilterKernel:
processVerticalPictureDistort(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processVerticalPictureDistort(int x, int y, unsigned char *data, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float distort = sinf(x * 0.05f + params.frame_count * 0.02f) * 10;
    int src_y = ((int)(y + distort) % height + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

```
}
```

Filter #255 - ShortTrail

Minimal motion trail using just one or two history frames blended lightly, for a subtle ghosting effect.

Technique: Minimal two-frame temporal blend.

```
// Case 255 in unifiedFilterKernel:
processShortTrail(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processShortTrail(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(0.8f * data[idx + j] + 0.2f * prev[idx + j]);
        }
    }
}
```

Filter #256 - DiagInward

Diagonal lines sweep inward from corners toward the center, each line pulling from different history frames.

Technique: Inward-diagonal frame history sweep.

```
// Case 256 in unifiedFilterKernel:
processDiagInward(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDiagInward(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    int dx = x - cx;
    int dy = y - cy;
    float dist = sqrtf((float)(dx * dx + dy * dy));
    float factor = 0.5f + 0.5f * sinf(dist * 0.05f - params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}
```

Filter #257 - DiagSquareInward

Diagonal square blocks sweep inward, with each block layer showing a different temporal frame.

Technique: Inward diagonal square temporal layering.

```
// Case 257 in unifiedFilterKernel:
processDiagSquareInward(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDiagSquareInward(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = params.square_size > 0 ? params.square_size : 32;
    int sx = (x / sq) * sq + sq / 2;
    int sy = (y / sq) * sq + sq / 2;
    int cx = width / 2;
    int cy = height / 2;
    float dist = sqrtf((float)((sx - cx) * (sx - cx) + (sy - cy) * (sy - cy)));
    float factor = 0.5f + 0.5f * sinf(dist * 0.02f - params.frame_count * 0.03f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}
```

Filter #258 - DiagSquareInwardResize

Diagonal square inward sweep with animated block sizes that grow and shrink.

Technique: Animated-size inward diagonal block sweep.

```
// Case 258 in unifiedFilterKernel:
processDiagSquareInwardResize(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDiagSquareInwardResize(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 16 + (params.frame_count / 4) % 48;
    int sx = (x / sq) * sq + sq / 2;
    int sy = (y / sq) * sq + sq / 2;
    int cx = width / 2;
    int cy = height / 2;
    float dist = sqrtf((float)((sx - cx) * (sx - cx) + (sy - cy) * (sy - cy)));
    float factor = 0.5f + 0.5f * sinf(dist * 0.02f - params.frame_count * 0.03f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}
```

Filter #259 - PictureShiftDownRight

Shifts the image simultaneously down and to the right over time.

Technique: Combined downward + rightward coordinate shift.

```
// Case 259 in unifiedFilterKernel:
processPictureShiftDownRight(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPictureShiftDownRight(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
```

```

    int offset = params.int_param1 % 100;
    int src_x = (x - offset + width) % width;
    int src_y = (y - offset + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}

```

Filter #260 - FlipPictureShift

Flips the image (vertically or horizontally) with an animated shift offset, creating a rolling mirror transition.

Technique: Flip + animated coordinate offset.

```

// Case 260 in unifiedFilterKernel:
processFlipPictureShift(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFlipPictureShift(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int flip = (params.frame_count / 30) % 4;
    int src_x = x, src_y = y;
    if (flip == 1) {
        src_x = width - 1 - x;
    } else if (flip == 2) {
        src_y = height - 1 - y;
    } else if (flip == 3) {
        src_x = width - 1 - x;
        src_y = height - 1 - y;
    }
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}

```

Filter #261 - RGBWideTrails

Wide spatial RGB-separated trails where each channel smears over a large area.

Technique: Wide per-channel spatial trailing.

```

// Case 261 in unifiedFilterKernel:
processRGBWideTrails(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processRGBWideTrails(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        int fi = (j * 8) % params.numFrames;
        unsigned char *hist = allFrames[fi];
        if (hist) {
            data[idx + j] = (unsigned char)(0.4f * data[idx + j] + 0.6f * hist[idx + j]);
        }
    }
}

```

```

    }
}

```

Filter #262 - LineInLineOut_Increase

Line interlace effect with increasing frame offsets over time, gradually spreading the temporal range.

Technique: Time-increasing row interlace offset.

```

// Case 262 in unifiedFilterKernel:
processLineInLineOut_Increase(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processLineInLineOut_Increase(int x, int y, unsigned char *data, int width, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int lineSize = 4 + (params.frame_count / 4) % 32;
    int segment = x / lineSize;
    float factor = (segment % 2 == 0) ? 1.0f + params.alpha * 0.5f : 1.0f - params.alpha * 0.3f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
    }
}

```

Filter #263 - LineInLineOut2_Increase

Second variant of increasing line interlace with different offset progression.

Technique: Alternate time-increasing row interlace.

```

// Case 263 in unifiedFilterKernel:
processLineInLineOut2_Increase(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processLineInLineOut2_Increase(int x, int y, unsigned char *data, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int lineSize = 4 + (params.frame_count / 4) % 32;
    int segment = y / lineSize;
    float factor = (segment % 2 == 0) ? 1.0f + params.alpha * 0.5f : 1.0f - params.alpha * 0.3f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
    }
}

```

Filter #264 - LineInLineOut3_Increase

Third variant with yet another offset progression pattern for line interlacing.

Technique: Third-variant time-increasing row interlace.

```

// Case 264 in unifiedFilterKernel:
processLineInLineOut3_Increase(x, y, data, width, height, step, params);

```

```

// Referenced __device__ function(s):
__device__ void processLineInLineOut3_Increase(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int lineSize = 8 + (params.frame_count / 8) % 24;
    int segX = x / lineSize;
    int segY = y / lineSize;
    float factor = ((segX + segY) % 2 == 0) ? 1.2f : 0.8f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
    }
}

```

Filter #265 - SquareByRow2Plus

Enhanced SquareByRow2 with additional blending or offset, creating richer temporal block banding.

Technique: Enhanced row-block temporal blending.

```

// Case 265 in unifiedFilterKernel:
processSquareByRow2Plus(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processSquareByRow2Plus(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = params.square_size > 0 ? params.square_size : 16;
    int row = y / sq;
    int offset = (row * 5) % width;
    int src_x = (x + offset) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
    }
}

```

Filter #266 - FrameSep

Separates the frame into sections that each display a different history frame, creating a split-screen temporal montage.

Technique: Frame-section temporal separation.

```

// Case 266 in unifiedFilterKernel:
processFrameSep(x, y, data, allFrames, width, step, params);

// Referenced __device__ function(s):
__device__ void processFrameSep(int x, int y, unsigned char *data, unsigned char **allFrames, int width,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int half = width / 2;
    int fi = (x < half) ? 0 : (params.numFrames > 1 ? 1 : 0);
    unsigned char *hist = allFrames[fi];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = hist[idx + j];
        }
    }
}

```

```

    }
}

```

Filter #267 - FrameSep2

Second variant of frame separation with different sectioning geometry.

Technique: Alternate frame-section temporal layout.

```

// Case 267 in unifiedFilterKernel:
processFrameSep2(x, y, data, allFrames, height, step, params);

// Referenced __device__ function(s):
__device__ void processFrameSep2(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ⇨ height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int half = height / 2;
    int fi = (y < half) ? 0 : (params.numFrames > 1 ? 1 : 0);
    unsigned char *hist = allFrames[fi];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = hist[idx + j];
        }
    }
}

```

Filter #268 - FrameStopStart

Alternately freezes (stops) and resumes (starts) frame updates in sections, creating a stuttered/freeze-frame mosaic.

Technique: Section-based frame freeze/resume alternation.

```

// Case 268 in unifiedFilterKernel:
processFrameStopStart(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processFrameStopStart(int x, int y, unsigned char *data, unsigned char **allFrames,
    ⇨ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cycle = (params.frame_count / 30) % 2;
    if (cycle == 0) {
        unsigned char *hist = allFrames[0];
        if (hist) {
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = hist[idx + j];
            }
        }
    }
}

```


Filter #269 - OutOfOrder

Displays frame history in a shuffled/out-of-order arrangement across the image, creating a temporal puzzle effect.

Technique: Shuffled frame history spatial mapping.

```
// Case 269 in unifiedFilterKernel:
processOutOfOrder(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processOutOfOrder(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int fi = (x ^ y ^ params.seed) % params.numFrames;
    unsigned char *hist = allFrames[fi];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * hist[idx + j]);
        }
    }
}
```

Filter #270 - TrackingDown

Shifts rows downward progressively like a VHS tracking error, with the displacement increasing from top to bottom.

Technique: Progressive downward row displacement.

```
// Case 270 in unifiedFilterKernel:
processTrackingDown(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processTrackingDown(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int track = (params.frame_count * 2) % height;
    if (y == track) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = 255;
        }
    }
}
```

Filter #271 - TrackingDownBlend

Tracking down effect blended with the original for a softer VHS-tracking look.

Technique: Blended progressive downward row displacement.

```
// Case 271 in unifiedFilterKernel:
processTrackingDownBlend(x, y, data, allFrames, height, step, params);

// Referenced __device__ function(s):
__device__ void processTrackingDownBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
```

```

    int track = (params.frame_count * 2) % height;
    if (abs(y - track) < 5) {
        unsigned char *prev = allFrames[0];
        if (prev) {
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = (unsigned char)(0.3f * data[idx + j] + 0.7f * prev[idx + j]);
            }
        }
    }
}

```

Filter #272 - TrackingRev

Reverse tracking-rows shift upward instead of downward.

Technique: Progressive upward row displacement.

```

// Case 272 in unifiedFilterKernel:
processTrackingRev(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processTrackingRev(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int track = height - 1 - ((params.frame_count * 2) % height);
    if (y == track) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = 255 - data[idx + j];
        }
    }
}

```

Filter #273 - TrackingMirror

Tracking displacement mirrored from center, creating symmetrical tracking errors spreading from the middle.

Technique: Center-mirrored row displacement.

```

// Case 273 in unifiedFilterKernel:
processTrackingMirror(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTrackingMirror(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int track = (params.frame_count * 2) % height;
    if (abs(y - track) < 10) {
        int src_x = width - 1 - x;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}

```

Filter #274 - BlockPixels

Groups pixels into large blocks (larger than PixelateRect), each showing a single averaged color. Creates a heavily pixelated, blocky look.

Technique: Large-block pixel averaging.

```
// Case 274 in unifiedFilterKernel:
processBlockPixels(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlockPixels(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int bx = x / 8;
    int by = y / 8;
    float r = gpu_rand(bx, by, params.seed);
    if (r < 0.3f) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + r));
        }
    }
}
```

Filter #275 - FrameChop

Chops the frame into segments and rearranges them, creating a cut-up/shuffled image.

Technique: Frame segment rearrangement.

```
// Case 275 in unifiedFilterKernel:
processFrameChop(x, y, data, allFrames, width, step, params);

// Referenced __device__ function(s):
__device__ void processFrameChop(int x, int y, unsigned char *data, unsigned char **allFrames, int width
    ↪ , size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int chop = width / 4;
    int section = x / chop;
    int fi = section % params.numFrames;
    unsigned char *hist = allFrames[fi];
    if (hist) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = hist[idx + j];
        }
    }
}
```

Filter #276 - YLineDown

Shifts pixels downward along Y-axis lines with frame history blending.

Technique: Y-line-based downward shift with blending.

```
// Case 276 in unifiedFilterKernel:
processYLineDown(x, y, data, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processYLineDown(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int linePos = (params.frame_count + x) % height;
    if (y == linePos) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = 255;
        }
    }
}

```

Filter #277 - YLineDownBlend

YLineDown with additional alpha blending for smoother transitions.

Technique: Blended Y-line downward shift.

```

// Case 277 in unifiedFilterKernel:
processYLineDownBlend(x, y, data, allFrames, height, step, params);

// Referenced __device__ function(s):
__device__ void processYLineDownBlend(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int linePos = (params.frame_count + x) % height;
    if (abs(y - linePos) < 3) {
        unsigned char *prev = allFrames[0];
        if (prev) {
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * prev[idx + j]);
            }
        }
    }
}

```

Filter #278 - SquareDiff1

Computes difference between square-block-sampled history frames, highlighting changes at block level.

Technique: Block-level temporal difference.

```

// Case 278 in unifiedFilterKernel:
processSquareDiff1(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processSquareDiff1(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 16;
    int sx = (x / sq) * sq;
    int sy = (y / sq) * sq;
    int src_idx = sy * step + sx * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            int diff = abs(data[idx + j] - prev[src_idx + j]);
        }
    }
}

```

```

        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + diff * 0.5f);
    }
}

```

Filter #279 - LineAcrossX

Draws a line across the frame at an animated position, with history frame data along the line.

Technique: Animated horizontal line from history.

```

// Case 279 in unifiedFilterKernel:
processLineAcrossXNew(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processLineAcrossXNew(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int lineX = (params.frame_count * 3) % width;
    if (abs(x - lineX) < 2) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = 255 - data[idx + j];
        }
    }
}

```

Filter #280 - ColorGlitch

Randomly glitches color values-swapping, corrupting, or shifting channels unpredictably. Creates a colorful digital corruption effect.

Technique: Random per-channel color corruption.

```

// Case 280 in unifiedFilterKernel:
processColorGlitch(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorGlitch(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.05f) {
        int channel = (int)(r * 60) % 3;
        data[idx + channel] = 255 - data[idx + channel];
    }
}

```

Filter #281 - PixelShiftUp

Shifts all pixels upward by an animated amount, with the bottom filling from frame history.

Technique: Animated upward pixel shift.

```

// Case 281 in unifiedFilterKernel:
processPixelShiftUp(x, y, data, height, step, params);

```

```
// Referenced __device__ function(s):
__device__ void processPixelShiftUp(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = params.int_param1 % height;
    int src_y = (y + offset) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #282 - PixelShiftDown

Shifts all pixels downward by an animated amount, with the top filling from frame history.

Technique: Animated downward pixel shift.

```
// Case 282 in unifiedFilterKernel:
processPixelShiftDown(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processPixelShiftDown(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = params.int_param1 % height;
    int src_y = (y - offset + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #283 - PixelShiftLeft

Shifts all pixels leftward by an animated amount.

Technique: Animated leftward pixel shift.

```
// Case 283 in unifiedFilterKernel:
processPixelShiftLeft(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processPixelShiftLeft(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = params.int_param2 % width;
    int src_x = (x + offset) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #284 - PixelShiftRight

Shifts all pixels rightward by an animated amount.

Technique: Animated rightward pixel shift.

```
// Case 284 in unifiedFilterKernel:
processPixelShiftRight(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processPixelShiftRight(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = params.int_param2 % width;
    int src_x = (x - offset + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #285 - PixelShiftDiagonal

Shifts pixels diagonally (both X and Y simultaneously), creating a diagonal scrolling effect.

Technique: Animated diagonal pixel shift.

```
// Case 285 in unifiedFilterKernel:
processPixelShiftDiagonal(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPixelShiftDiagonal(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset_x = params.int_param2 % width;
    int offset_y = params.int_param1 % height;
    int src_x = (x + offset_x) % width;
    int src_y = (y + offset_y) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #286 - WaveBlend

Blends the current frame with a wave-distorted version of itself, creating a soft waving blend effect.

Technique: Wave-distorted self-blend.

```
// Case 286 in unifiedFilterKernel:
processWaveBlend(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processWaveBlend(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
```

```

float wave = sinf((x + params.frame_count) * 0.05f) * 20.0f;
int src_x = ((int)(x + wave) % width + width) % width;
int src_idx = y * step + src_x * 4;
for (int j = 0; j < 3; ++j) {
    data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
}
}

```

Filter #287 - WaveBlendX2

Double-intensity wave blending for a more pronounced wavy distortion.

Technique: Intensity-doubled wave self-blend.

```

// Case 287 in unifiedFilterKernel:
processWaveBlendX2(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processWaveBlendX2(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float waveX = sinf((x + params.frame_count) * 0.05f) * 20.0f;
    float waveY = cosf((y + params.frame_count) * 0.05f) * 20.0f;
    int src_x = ((int)(x + waveX) % width + width) % width;
    int src_y = ((int)(y + waveY) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
    }
}

```

Filter #288 - SineWaveBlend

Blends frames using sine-wave modulated alpha, so the blend strength ripples across the image.

Technique: Sine-wave alpha modulation blending.

```

// Case 288 in unifiedFilterKernel:
processSineWaveBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSineWaveBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float factor = 0.5f + 0.5f * sinf((x + y + params.frame_count) * 0.03f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}

```

Filter #289 - CosineWaveBlend

Blends using cosine-wave modulated alpha-complementary wave pattern to SineWaveBlend.

Technique: Cosine-wave alpha modulation blending.


```
// Case 289 in unifiedFilterKernel:
processCosineWaveBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCosineWaveBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float factor = 0.5f + 0.5f * cosf((x - y + params.frame_count) * 0.03f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}
```

Filter #290 - SpiralWave

Applies a spiral-shaped wave distortion from the center, twisting the image outward in a spiral pattern.

Technique: Spiral polar-coordinate wave distortion.

```
// Case 290 in unifiedFilterKernel:
processSpiralWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSpiralWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float angle = atan2f(dy, dx) + params.frame_count * 0.02f;
    float dist = sqrtf(dx * dx + dy * dy);
    float factor = 0.5f + 0.5f * sinf(angle * 3.0f + dist * 0.05f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}
```

Filter #291 - RadialBlur

Motion blur radiating outward from the center point, creating a speed-zoom blur.

Technique: Radial vector motion blur.

```
// Case 291 in unifiedFilterKernel:
processRadialBlur(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRadialBlur(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float dist = sqrtf(dx * dx + dy * dy);
```

```

float blur = fminf(dist * 0.01f, 0.5f);
int src_x = (int)(cx + dx * (1.0f - blur));
int src_y = (int)(cy + dy * (1.0f - blur));
src_x = (src_x % width + width) % width;
src_y = (src_y % height + height) % height;
int src_idx = src_y * step + src_x * 4;
for (int j = 0; j < 3; ++j) {
    data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
}
}

```

Filter #292 - ZoomBlur

Zooming motion blur creating a rush/tunnel-speed effect toward or away from center.

Technique: Zoom-direction motion blur.

```

// Case 292 in unifiedFilterKernel:
processZoomBlur(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processZoomBlur(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float zoom = 1.0f + 0.1f * sinf(params.frame_count * 0.05f);
    int src_x = (int)(cx + (x - cx) / zoom);
    int src_y = (int)(cy + (y - cy) / zoom);
    src_x = (src_x % width + width) % width;
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}

```

Filter #293 - RotateBlend

Blends the current frame with a rotated version, creating a spinning overlay effect.

Technique: Rotation blending.

```

// Case 293 in unifiedFilterKernel:
processRotateBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRotateBlend(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float angle = params.frame_count * 0.01f;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    int src_x = (int)(cx + dx * cosf(angle) - dy * sinf(angle));
    int src_y = (int)(cy + dx * sinf(angle) + dy * cosf(angle));
    src_x = (src_x % width + width) % width;

```

```

    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
    }
}

```

Filter #294 - MirrorWave

Mirrors the image at a wave-shaped boundary that undulates over time.

Technique: Wave-boundary mirror reflection.

```

// Case 294 in unifiedFilterKernel:
processMirrorWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processMirrorWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf((y + params.frame_count) * 0.1f) * 30.0f;
    int src_x = (int)(width - 1 - x + wave);
    src_x = (src_x % width + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
    }
}

```

Filter #295 - MirrorWaveX

Horizontal variant of wave mirror-the mirror boundary undulates horizontally.

Technique: Horizontal wave-boundary mirror.

```

// Case 295 in unifiedFilterKernel:
processMirrorWaveX(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processMirrorWaveX(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf((x + params.frame_count) * 0.08f) * 20.0f;
    int src_x = (int)(width - 1 - x + wave);
    src_x = (src_x % width + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.6f * data[idx + j] + 0.4f * data[src_idx + j]);
    }
}

```

Filter #296 - MirrorWaveY

Vertical variant of wave mirror-the mirror boundary undulates vertically.

Technique: Vertical wave-boundary mirror.

```
// Case 296 in unifiedFilterKernel:
processMirrorWaveY(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processMirrorWaveY(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf((y + params.frame_count) * 0.08f) * 20.0f;
    int src_y = (int)(height - 1 - y + wave);
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.6f * data[idx + j] + 0.4f * data[src_idx + j]);
    }
}
```

Filter #297 - PixelDrift

Gradually shifts pixel positions over time using sine/cosine modulation, creating a floating drift effect.

Technique: Sine/cosine animated position drift.

```
// Case 297 in unifiedFilterKernel:
processPixelDrift(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPixelDrift(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int drift = ((x / 4 + params.frame_count) % 8) - 4;
    int src_x = (x + drift + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #298 - PixelDriftX

Horizontal-only pixel drift variant.

Technique: Horizontal sine-animated position drift.

```
// Case 298 in unifiedFilterKernel:
processPixelDriftX(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processPixelDriftX(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float drift = sinf(y * 0.03f + params.frame_count * 0.05f) * 15.0f;
    int src_x = ((int)(x + drift) % width + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

```
}  
}
```

Filter #299 - PixelDriftY

Vertical-only pixel drift variant.

Technique: Vertical sine-animated position drift.

```
// Case 299 in unifiedFilterKernel:  
processPixelDriftY(x, y, data, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processPixelDriftY(int x, int y, unsigned char *data, int height, size_t step, const  
    ↪ FilterParams &params) {  
    int idx = y * step + x * 4;  
    float drift = cosf(x * 0.03f + params.frame_count * 0.05f) * 15.0f;  
    int src_y = ((int)(y + drift) % height + height) % height;  
    int src_idx = src_y * step + x * 4;  
    for (int j = 0; j < 3; ++j) {  
        data[idx + j] = data[src_idx + j];  
    }  
}
```

Advanced Blend (300-399)

Filter #300 - ColorPulse

Pulses color intensity using sine/cosine modulation per channel, creating a rhythmic color brightness oscillation across the image.

Technique: Per-channel sine/cosine amplitude modulation.

```
// Case 300 in unifiedFilterKernel:  
processColorPulse(x, y, data, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processColorPulse(int x, int y, unsigned char *data, size_t step, const FilterParams &  
    ↪ params) {  
    int idx = y * step + x * 4;  
    float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.1f);  
    for (int j = 0; j < 3; ++j) {  
        data[idx + j] = (unsigned char)(data[idx + j] * pulse);  
    }  
}
```

Filter #301 - ColorPulseRGB

Each R, G, B channel pulses at a different frequency, creating desynchronized color pulsing.

Technique: Multi-frequency per-channel pulse.

```
// Case 301 in unifiedFilterKernel:  
processColorPulseRGB(x, y, data, step, params);  
  
// Referenced __device__ function(s):
```

```

__device__ void processColorPulseRGB(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float pulseR = 0.5f + 0.5f * sinf(params.frame_count * 0.08f);
    float pulseG = 0.5f + 0.5f * sinf(params.frame_count * 0.1f);
    float pulseB = 0.5f + 0.5f * sinf(params.frame_count * 0.12f);
    data[idx] = (unsigned char)(data[idx] * pulseB);
    data[idx + 1] = (unsigned char)(data[idx + 1] * pulseG);
    data[idx + 2] = (unsigned char)(data[idx + 2] * pulseR);
}

```

Filter #302 - ColorPulseXor

Color pulsing combined with XOR operations for a digital pulsating distortion.

Technique: XOR-modulated color pulse.

```

// Case 302 in unifiedFilterKernel:
processColorPulseXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorPulseXor(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int pulse = (int)(128 * sinf(params.frame_count * 0.05f));
    for (int j = 0; j < 3; ++j) {
        data[idx + j] ^= (unsigned char)abs(pulse);
    }
}

```

Filter #303 - GlitchBlock

Randomly displaces rectangular blocks, creating the classic datamosh/macroblock glitch look.

Technique: Random rectangular block displacement.

```

// Case 303 in unifiedFilterKernel:
processGlitchBlock(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGlitchBlock(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int bx = x / 32;
    int by = y / 32;
    float r = gpu_rand(bx, by, params.seed);
    if (r < 0.15f) {
        int offset = (int)(r * 200) - 100;
        int src_x = (x + offset) % width;
        if (src_x < 0)
            src_x += width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}

```

Filter #304 - GlitchBlockXor

Block displacement combined with XOR on the displaced blocks for added digital artifact texture.

Technique: XOR'd random block displacement.

```
// Case 304 in unifiedFilterKernel:
processGlitchBlockXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGlitchBlockXor(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int bx = x / 32;
    int by = y / 32;
    float r = gpu_rand(bx, by, params.seed);
    if (r < 0.2f) {
        unsigned char xv = (unsigned char)(r * 255);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= xv;
        }
    }
}
```

Filter #305 - GlitchLine

Randomly displaces entire horizontal lines, creating a horizontal shatter/glitch.

Technique: Random horizontal line displacement.

```
// Case 305 in unifiedFilterKernel:
processGlitchLine(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processGlitchLine(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(0, y, params.seed);
    if (r < 0.1f) {
        int offset = (int)(r * 100) % 50;
        int src_x = (x + offset) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}
```

Filter #306 - GlitchLineX

Extended variant of line glitch with larger/more frequent displacements.

Technique: Intensified random line displacement.

```
// Case 306 in unifiedFilterKernel:
processGlitchLineX(x, y, data, height, step, params);
```

```

// Referenced __device__ function(s):
__device__ void processGlitchLineX(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, 0, params.seed);
    if (r < 0.1f) {
        int offset = (int)(r * 100) % 50;
        int src_y = (y + offset) % height;
        int src_idx = src_y * step + x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}

```

Filter #307 - NoiseBlend

Blends random noise with the current frame, creating a film-grain-like static overlay.

Technique: Random noise alpha blending.

```

// Case 307 in unifiedFilterKernel:
processNoiseBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processNoiseBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    unsigned char noise = (unsigned char)(r * 255);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.8f * data[idx + j] + 0.2f * noise);
    }
}

```

Filter #308 - NoiseBlendX2

Double-intensity noise blend for heavier static interference.

Technique: Intensified noise blending.

```

// Case 308 in unifiedFilterKernel:
processNoiseBlendX2(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processNoiseBlendX2(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    unsigned char noise = (unsigned char)(r * 255);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.6f * data[idx + j] + 0.4f * noise);
    }
}

```


Filter #309 - NoiseXor

XORs the frame with random noise values, creating digital static distortion.

Technique: Random noise XOR.

```
// Case 309 in unifiedFilterKernel:
processNoiseXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processNoiseXor(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.3f) {
        unsigned char noise = (unsigned char)(r * 255 * 3.33f);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= noise;
        }
    }
}
```

Filter #310 - ChannelShift

Rotates color channels by one position (R->G->B->R) with animated shift amount. Creates a cycling false-color effect.

Technique: Animated circular channel rotation.

```
// Case 310 in unifiedFilterKernel:
processChannelShift(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processChannelShift(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = params.frame_count % 30;
    int src_x_r = (x + shift) % width;
    int src_x_b = (x - shift + width) % width;
    unsigned char r_val = data[y * step + src_x_r * 4 + 2];
    unsigned char b_val = data[y * step + src_x_b * 4];
    data[idx + 2] = r_val;
    data[idx] = b_val;
}
```

Filter #311 - ChannelShiftX

Extended channel shift with larger or variable rotation amounts.

Technique: Variable-rate channel rotation.

```
// Case 311 in unifiedFilterKernel:
processChannelShiftX(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processChannelShiftX(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
```

```

    int shift = params.frame_count % 20;
    int src_y_r = (y + shift) % height;
    int src_y_b = (y - shift + height) % height;
    unsigned char r_val = data[src_y_r * step + x * 4 + 2];
    unsigned char b_val = data[src_y_b * step + x * 4];
    data[idx + 2] = r_val;
    data[idx] = b_val;
}

```

Filter #312 - ChannelRotate

Continuously and smoothly rotates color channels, creating an evolving false-color animation.

Technique: Continuous channel rotation.

```

// Case 312 in unifiedFilterKernel:
processChannelRotate(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processChannelRotate(int x, int y, unsigned char *data, size_t step, const FilterParams
    &params) {
    int idx = y * step + x * 4;
    int rot = params.frame_count % 3;
    unsigned char b = data[idx];
    unsigned char g = data[idx + 1];
    unsigned char r = data[idx + 2];
    if (rot == 0) {
        data[idx] = r;
        data[idx + 1] = b;
        data[idx + 2] = g;
    } else if (rot == 1) {
        data[idx] = g;
        data[idx + 1] = r;
        data[idx + 2] = b;
    }
}

```

Filter #313 - DiagonalStretch

Stretches the image diagonally, creating a sheared distortion along a 45-degree axis.

Technique: Diagonal coordinate scaling.

```

// Case 313 in unifiedFilterKernel:
processDiagonalStretch(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDiagonalStretch(int x, int y, unsigned char *data, int width, int height, size_t
    &step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float stretch = 1.0f + 0.3f * sinf(params.frame_count * 0.03f);
    int diag = x + y;
    int src_x = (int)(x + (diag % 50) * (stretch - 1.0f)) % width;
    if (src_x < 0)
        src_x += width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}

```

```
}  
}
```

Filter #314 - DiagonalStretchX

Extended diagonal stretch with greater intensity.

Technique: Intensified diagonal coordinate scaling.

```
// Case 314 in unifiedFilterKernel:  
processDiagonalStretchX(x, y, data, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processDiagonalStretchX(int x, int y, unsigned char *data, int width, int height, size_t  
    ↪ step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    float stretch = 1.0f + 0.3f * cosf(params.frame_count * 0.03f);  
    int diag = x - y;  
    int src_y = (int)(y + (abs(diag) % 50) * (stretch - 1.0f)) % height;  
    if (src_y < 0)  
        src_y += height;  
    int src_idx = src_y * step + x * 4;  
    for (int j = 0; j < 3; ++j) {  
        data[idx + j] = data[src_idx + j];  
    }  
}
```

Filter #315 - DiagonalMirror

Mirrors the image along a diagonal axis, creating diagonal symmetry.

Technique: Diagonal-axis coordinate reflection.

```
// Case 315 in unifiedFilterKernel:  
processDiagonalMirror(x, y, data, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processDiagonalMirror(int x, int y, unsigned char *data, int width, int height, size_t  
    ↪ step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    if (x + y < (width + height) / 2) {  
        int src_x = y % width;  
        int src_y = x % height;  
        int src_idx = src_y * step + src_x * 4;  
        for (int j = 0; j < 3; ++j) {  
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);  
        }  
    }  
}
```

Filter #316 - SquareWave

Applies a square wave function to pixel brightness, creating hard-edged alternating bright/dark bands.

Technique: Square wave brightness modulation.

```
// Case 316 in unifiedFilterKernel:
processSquareWave(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSquareWave(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int period = 32;
    int phase = (x + params.frame_count) / period;
    float factor = (phase % 2 == 0) ? 1.2f : 0.8f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
    }
}
```

Filter #317 - SquareWaveX

Extended square wave with variable frequency/amplitude.

Technique: Variable square wave modulation.

```
// Case 317 in unifiedFilterKernel:
processSquareWaveX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSquareWaveX(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int period = 32;
    int phase = (y + params.frame_count) / period;
    float factor = (phase % 2 == 0) ? 1.2f : 0.8f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
    }
}
```

Filter #318 - SquareWaveBlend

Square wave brightness banding blended with frame history.

Technique: Square wave with temporal blending.

```
// Case 318 in unifiedFilterKernel:
processSquareWaveBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processSquareWaveBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int period = 48;
    int phase = (x + y + params.frame_count) / period;
    unsigned char *src = (phase % 2 == 0) ? allFrames[0] : NULL;
    if (src) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * src[idx + j]);
        }
    }
}
```

```
}  
}
```

Filter #319 - TriangleWave

Applies a triangle wave function to brightness, creating softer saw-like alternating bright/dark bands.

Technique: Triangle wave brightness modulation.

```
// Case 319 in unifiedFilterKernel:  
processTriangleWave(x, y, data, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processTriangleWave(int x, int y, unsigned char *data, size_t step, const FilterParams &  
    ↪ params) {  
    int idx = y * step + x * 4;  
    int period = 64;  
    int pos = (x + params.frame_count) % period;  
    float factor = (pos < period / 2) ? (float)pos / (period / 2) : 2.0f - (float)pos / (period / 2);  
    factor = 0.5f + 0.5f * factor;  
    for (int j = 0; j < 3; ++j) {  
        data[idx + j] = (unsigned char)(data[idx + j] * factor);  
    }  
}
```

Filter #320 - TriangleWaveBlend

Triangle wave banding blended with frame history.

Technique: Triangle wave with temporal blending.

```
// Case 320 in unifiedFilterKernel:  
processTriangleWaveBlend(x, y, data, allFrames, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processTriangleWaveBlend(int x, int y, unsigned char *data, unsigned char **allFrames,  
    ↪ size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    int period = 64;  
    int pos = (y + params.frame_count) % period;  
    float alpha = (pos < period / 2) ? (float)pos / (period / 2) : 2.0f - (float)pos / (period / 2);  
    unsigned char *prev = allFrames[0];  
    if (prev) {  
        for (int j = 0; j < 3; ++j) {  
            data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - alpha) + prev[idx + j] * alpha);  
        }  
    }  
}
```

Filter #321 - SawtoothWave

Applies a sawtooth wave-brightness ramps up then drops sharply in repeating bands.

Technique: Sawtooth wave brightness modulation.

```
// Case 321 in unifiedFilterKernel:
processSawtoothWave(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSawtoothWave(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int period = 64;
    int pos = (x + params.frame_count) % period;
    float factor = 0.5f + 0.5f * ((float)pos / period);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}
```

Filter #322 - SawtoothWaveBlend

Sawtooth wave with frame history blending.

Technique: Sawtooth wave with temporal blending.

```
// Case 322 in unifiedFilterKernel:
processSawtoothWaveBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processSawtoothWaveBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int period = 64;
    int pos = (y + params.frame_count) % period;
    float alpha = (float)pos / period;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - alpha) + prev[idx + j] * alpha);
        }
    }
}
```

Filter #323 - PulseWave

Short-duty-cycle pulse wave creating narrow bright stripes on a darker background.

Technique: Pulse wave brightness pattern.

```
// Case 323 in unifiedFilterKernel:
processPulseWave(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPulseWave(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int period = 48;
    int pos = (x + y + params.frame_count) % period;
    float factor = (pos < period / 4) ? 1.3f : 0.9f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
    }
}
```

```
}  
}
```

Filter #324 - PulseWaveBlend

Pulse wave with frame blending.

Technique: Pulse wave with temporal blending.

```
// Case 324 in unifiedFilterKernel:  
processPulseWaveBlend(x, y, data, allFrames, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processPulseWaveBlend(int x, int y, unsigned char *data, unsigned char **allFrames,  
    ↪ size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    int period = 48;  
    int pos = (x + y + params.frame_count) % period;  
    float blend = (pos < period / 4) ? 0.8f : 0.3f;  
    unsigned char *prev = allFrames[0];  
    if (prev) {  
        for (int j = 0; j < 3; ++j) {  
            data[idx + j] = (unsigned char)(data[idx + j] * blend + prev[idx + j] * (1.0f - blend));  
        }  
    }  
}
```

Filter #325 - StepWave

Staircase/stepped brightness pattern creating quantized brightness bands.

Technique: Stepped quantized brightness.

```
// Case 325 in unifiedFilterKernel:  
processStepWave(x, y, data, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processStepWave(int x, int y, unsigned char *data, size_t step, const FilterParams &  
    ↪ params) {  
    int idx = y * step + x * 4;  
    int steps = 4;  
    int period = 64;  
    int pos = (x + params.frame_count) % period;  
    int stepVal = (pos * steps) / period;  
    float factor = 0.5f + 0.5f * ((float)stepVal / steps);  
    for (int j = 0; j < 3; ++j) {  
        data[idx + j] = (unsigned char)(data[idx + j] * factor);  
    }  
}
```

Filter #326 - StepWaveBlend

Stepped brightness blended with frame history.

Technique: Step wave with temporal blending.

```
// Case 326 in unifiedFilterKernel:
processStepWaveBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processStepWaveBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int steps = 4;
    int period = 64;
    int pos = (y + params.frame_count) % period;
    int stepVal = (pos * steps) / period;
    float blend = (float)stepVal / steps;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - blend) + prev[idx + j] * blend);
        }
    }
}
```

Filter #327 - RippleEffect

Concentric ripple distortion from center, displacing pixels outward in circular wavefronts like a water drop impact.

Technique: Radial sine-wave coordinate displacement.

```
// Case 327 in unifiedFilterKernel:
processRippleEffect(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRippleEffect(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float dist = sqrtf(dx * dx + dy * dy);
    float ripple = sinf(dist * 0.1f - params.frame_count * 0.1f) * 10.0f;
    int src_x = (int)(x + ripple * dx / (dist + 1.0f));
    int src_y = (int)(y + ripple * dy / (dist + 1.0f));
    src_x = (src_x % width + width) % width;
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #328 - RippleEffectX2

Double-intensity ripple with more wave cycles visible.

Technique: Intensified radial ripple distortion.

```
// Case 328 in unifiedFilterKernel:
```



```

processRippleEffectX2(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRippleEffectX2(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float dist = sqrtf(dx * dx + dy * dy);
    float ripple = sinf(dist * 0.15f - params.frame_count * 0.15f) * 15.0f;
    int src_x = (int)(x + ripple * dx / (dist + 1.0f));
    int src_y = (int)(y + ripple * dy / (dist + 1.0f));
    src_x = (src_x % width + width) % width;
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
    }
}

```

Filter #329 - ShockWave

Expanding ring of distortion from center, like a single explosion shockwave propagating outward.

Technique: Animated expanding-radius ring distortion.

```

// Case 329 in unifiedFilterKernel:
processShockWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processShockWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float dist = sqrtf(dx * dx + dy * dy);
    float wave_pos = (params.frame_count * 3) % (int)(sqrtf((float)(width * width + height * height)));
    float wave_width = 30.0f;
    if (fabsf(dist - wave_pos) < wave_width) {
        float offset = sinf((dist - wave_pos) * 0.2f) * 10.0f;
        int src_x = (int)(x + offset * dx / (dist + 1.0f));
        int src_y = (int)(y + offset * dy / (dist + 1.0f));
        src_x = (src_x % width + width) % width;
        src_y = (src_y % height + height) % height;
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}

```

Filter #330 - ShockWaveBlend

Shockwave distortion blended with frame history.

Technique: Shockwave with temporal blending.

```
// Case 330 in unifiedFilterKernel:
processShockWaveBlend(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processShockWaveBlend(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float dist = sqrtf(dx * dx + dy * dy);
    float wave_pos = (params.frame_count * 3) % (int)(sqrtf((float)(width * width + height * height)));
    float wave_width = 40.0f;
    unsigned char *prev = allFrames[0];
    if (fabsf(dist - wave_pos) < wave_width && prev) {
        float blend = 1.0f - fabsf(dist - wave_pos) / wave_width;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - blend) + prev[idx + j] * blend);
        }
    }
}
```

Filter #331 - TwistEffect

Rotational twisting distortion around the center-pixels near center rotate more than those at edges, creating a spiral twist.

Technique: Radius-dependent rotational coordinate warp.

```
// Case 331 in unifiedFilterKernel:
processTwistEffect(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTwistEffect(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float dist = sqrtf(dx * dx + dy * dy);
    float max_dist = sqrtf((float)(cx * cx + cy * cy));
    float twist = (1.0f - dist / max_dist) * sinf(params.frame_count * 0.02f) * 1.5f;
    float angle = atan2f(dy, dx) + twist;
    int src_x = (int)(cx + dist * cosf(angle));
    int src_y = (int)(cy + dist * sinf(angle));
    src_x = (src_x % width + width) % width;
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #332 - TwistEffectBlend

Twist effect blended with frame history.

Technique: Spiral twist with temporal blending.

```
// Case 332 in unifiedFilterKernel:
processTwistEffectBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTwistEffectBlend(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float dist = sqrtf(dx * dx + dy * dy);
    float max_dist = sqrtf((float)(cx * cx + cy * cy));
    float twist = (1.0f - dist / max_dist) * sinf(params.frame_count * 0.03f) * 2.0f;
    float angle = atan2f(dy, dx) + twist;
    int src_x = (int)(cx + dist * cosf(angle));
    int src_y = (int)(cy + dist * sinf(angle));
    src_x = (src_x % width + width) % width;
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
    }
}
```

Filter #333 - FishEye

Barrel distortion simulating a fish-eye lens-center area magnified, edges compressed.

Technique: Radial barrel coordinate distortion.

```
// Case 333 in unifiedFilterKernel:
processFishEye(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFishEye(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx) / cx;
    float dy = (float)(y - cy) / cy;
    float dist = sqrtf(dx * dx + dy * dy);
    if (dist < 1.0f) {
        float factor = powf(dist, 1.5f) / dist;
        int src_x = (int)(cx + dx * factor * cx);
        int src_y = (int)(cy + dy * factor * cy);
        src_x = (src_x % width + width) % width;
        src_y = (src_y % height + height) % height;
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}
```

```

    }
}
}

```

Filter #334 - FishEyeBlend

Fish-eye distortion blended with frame history.

Technique: Fish-eye with temporal blending.

```

// Case 334 in unifiedFilterKernel:
processFishEyeBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFishEyeBlend(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx) / cx;
    float dy = (float)(y - cy) / cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float strength = 0.5f + 0.5f * sinf(params.frame_count * 0.05f);
    if (dist < 1.0f) {
        float factor = powf(dist, 1.0f + strength) / dist;
        int src_x = (int)(cx + dx * factor * cx);
        int src_y = (int)(cy + dy * factor * cy);
        src_x = (src_x % width + width) % width;
        src_y = (src_y % height + height) % height;
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
        }
    }
}
}

```

Filter #335 - Kaleidoscope

Folds the image into mirrored symmetrical sectors like a kaleidoscope, creating mandala-like symmetry.

Technique: Angular sector folding and mirroring.

```

// Case 335 in unifiedFilterKernel:
processKaleidoscope(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processKaleidoscope(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float angle = atan2f(dy, dx);
    float dist = sqrtf(dx * dx + dy * dy);
    int segments = 6;

```

```

float segment_angle = (2.0f * 3.14159265f) / segments;
float new_angle = fmodf(fabsf(angle), segment_angle);
if ((int)(angle / segment_angle) % 2 == 1)
    new_angle = segment_angle - new_angle;
new_angle += params.frame_count * 0.01f;
int src_x = (int)(cx + dist * cosf(new_angle));
int src_y = (int)(cy + dist * sinf(new_angle));
src_x = (src_x % width + width) % width;
src_y = (src_y % height + height) % height;
int src_idx = src_y * step + src_x * 4;
for (int j = 0; j < 3; ++j) {
    data[idx + j] = data[src_idx + j];
}
}

```

Filter #336 - KaleidoscopeBlend

Kaleidoscope effect with frame history blending for temporal depth.

Technique: Kaleidoscope with temporal blending.

```

// Case 336 in unifiedFilterKernel:
processKaleidoscopeBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processKaleidoscopeBlend(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float angle = atan2f(dy, dx);
    float dist = sqrtf(dx * dx + dy * dy);
    int segments = 8;
    float segment_angle = (2.0f * 3.14159265f) / segments;
    float new_angle = fmodf(fabsf(angle), segment_angle);
    if ((int)(angle / segment_angle) % 2 == 1)
        new_angle = segment_angle - new_angle;
    new_angle += params.frame_count * 0.02f;
    int src_x = (int)(cx + dist * cosf(new_angle));
    int src_y = (int)(cy + dist * sinf(new_angle));
    src_x = (src_x % width + width) % width;
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
    }
}

```

Filter #337 - TunnelEffect

Warps the image into a receding tunnel using logarithmic polar coordinates, creating the illusion of depth.

Technique: Logarithmic polar coordinate transformation.

```
// Case 337 in unifiedFilterKernel:
processTunnelEffect(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTunnelEffect(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float angle = atan2f(dy, dx);
    float dist = sqrtf(dx * dx + dy * dy) + 1.0f;
    float tunnel_x = angle / 3.14159265f * width;
    float tunnel_y = 100.0f / dist * height + params.frame_count * 2.0f;
    int src_x = ((int)tunnel_x % width + width) % width;
    int src_y = ((int)tunnel_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #338 - TunnelEffectBlend

Tunnel effect blended with frame history.

Technique: Tunnel warp with temporal blending.

```
// Case 338 in unifiedFilterKernel:
processTunnelEffectBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTunnelEffectBlend(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float angle = atan2f(dy, dx);
    float dist = sqrtf(dx * dx + dy * dy) + 1.0f;
    float tunnel_x = angle / 3.14159265f * width;
    float tunnel_y = 100.0f / dist * height + params.frame_count * 2.0f;
    int src_x = ((int)tunnel_x % width + width) % width;
    int src_y = ((int)tunnel_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.6f * data[idx + j] + 0.4f * data[src_idx + j]);
    }
}
```

Filter #339 - VortexEffect

Spinning vortex distortion that swirls pixels around the center with intensity increasing toward the core.

Technique: Spiral vortex coordinate transformation.

```
// Case 339 in unifiedFilterKernel:
processVortexEffect(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processVortexEffect(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float dist = sqrtf(dx * dx + dy * dy);
    float angle = atan2f(dy, dx) + dist * 0.01f * sinf(params.frame_count * 0.03f);
    int src_x = (int)(cx + dist * cosf(angle));
    int src_y = (int)(cy + dist * sinf(angle));
    src_x = (src_x % width + width) % width;
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
```

Filter #340 - VortexEffectBlend

Vortex effect with frame blending.

Technique: Vortex with temporal blending.

```
// Case 340 in unifiedFilterKernel:
processVortexEffectBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processVortexEffectBlend(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    float dx = (float)(x - cx);
    float dy = (float)(y - cy);
    float dist = sqrtf(dx * dx + dy * dy);
    float angle = atan2f(dy, dx) + dist * 0.015f * sinf(params.frame_count * 0.04f);
    int src_x = (int)(cx + dist * cosf(angle));
    int src_y = (int)(cy + dist * sinf(angle));
    src_x = (src_x % width + width) % width;
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
    }
}
```

Filter #341 - ColorDrift

Gradually drifts each channel's brightness in a random direction over time, creating slow color wandering.

Technique: Animated per-channel brightness drift.

```
// Case 341 in unifiedFilterKernel:
processColorDrift(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorDrift(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float drift = sinf(params.frame_count * 0.02f);
    int shift = (int)(drift * 50);
    data[idx] = (unsigned char)((data[idx] + shift + 256) % 256);
    data[idx + 1] = (unsigned char)((data[idx + 1] - shift + 256) % 256);
    data[idx + 2] = (unsigned char)((data[idx + 2] + shift / 2 + 256) % 256);
}
```

Filter #342 - ColorDriftX

Extended color drift with larger drift range.

Technique: Intensified per-channel brightness drift.

```
// Case 342 in unifiedFilterKernel:
processColorDriftX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorDriftX(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float drift = cosf(params.frame_count * 0.025f);
    int shift = (int)(drift * 40);
    data[idx] = (unsigned char)((data[idx] - shift + 256) % 256);
    data[idx + 1] = (unsigned char)((data[idx + 1] + shift + 256) % 256);
    data[idx + 2] = (unsigned char)((data[idx + 2] - shift / 2 + 256) % 256);
}
```

Filter #343 - RGBShift

Shifts each R, G, B channel to a slightly different spatial position, creating chromatic aberration/color fringing at edges.

Technique: Per-channel spatial coordinate offset.

```
// Case 343 in unifiedFilterKernel:
processRGBShift(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processRGBShift(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = 3 + (params.frame_count % 10);
    int r_x = (x + shift) % width;
    int b_x = (x - shift + width) % width;
```



```

    unsigned char r = data[y * step + r_x * 4 + 2];
    unsigned char b = data[y * step + b_x * 4];
    data[idx + 2] = r;
    data[idx] = b;
}

```

Filter #344 - RGBShiftX

Extended RGB shift with larger offsets for more pronounced chromatic aberration.

Technique: Intensified per-channel spatial offset.

```

// Case 344 in unifiedFilterKernel:
processRGBShiftX(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processRGBShiftX(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = 3 + (params.frame_count % 10);
    int r_y = (y + shift) % height;
    int b_y = (y - shift + height) % height;
    unsigned char r = data[r_y * step + x * 4 + 2];
    unsigned char b = data[b_y * step + x * 4];
    data[idx + 2] = r;
    data[idx] = b;
}

```

Filter #345 - ChromaticAberration

Simulates lens chromatic aberration-channels spread radially outward from center, with more separation at edges.

Technique: Radial distance-proportional per-channel offset.

```

// Case 345 in unifiedFilterKernel:
processChromaticAberration(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processChromaticAberration(int x, int y, unsigned char *data, int width, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = 5;
    int r_x = (x + shift) % width;
    int g_x = x;
    int b_x = (x - shift + width) % width;
    data[idx + 2] = data[y * step + r_x * 4 + 2];
    data[idx + 1] = data[y * step + g_x * 4 + 1];
    data[idx] = data[y * step + b_x * 4];
}

```

Filter #346 - ChromaticAberrationX

Extended chromatic aberration with greater radial spread.

Technique: Intensified radial per-channel offset.

```
// Case 346 in unifiedFilterKernel:
processChromaticAberrationX(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processChromaticAberrationX(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = 5;
    int r_x = (x + shift) % width;
    int r_y = (y + shift) % height;
    int b_x = (x - shift + width) % width;
    int b_y = (y - shift + height) % height;
    data[idx + 2] = data[r_y * step + r_x * 4 + 2];
    data[idx] = data[b_y * step + b_x * 4];
}
```

Filter #347 - Posterize

Reduces the number of distinct color levels per channel (quantization), creating a poster-art look with flat color regions.

Technique: Color level quantization.

```
// Case 347 in unifiedFilterKernel:
processPosterize(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPosterize(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int levels = 4;
    float factor = 255.0f / levels;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(roundf(data[idx + j] / factor) * factor);
    }
}
```

Filter #348 - PosterizeBlend

Posterization blended with frame history for temporal depth.

Technique: Posterization with temporal blending.

```
// Case 348 in unifiedFilterKernel:
processPosterizeBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPosterizeBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int levels = 4 + (params.frame_count % 4);
    float factor = 255.0f / levels;
    for (int j = 0; j < 3; ++j) {
        unsigned char poster = (unsigned char)(roundf(data[idx + j] / factor) * factor);
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * poster);
    }
}
```

```
}
```

Filter #349 - Solarize

Applies a solarization curve-pixels above a threshold are inverted, creating the classic darkroom solarization effect with partial negative tones.

Technique: Threshold-based partial color inversion.

```
// Case 349 in unifiedFilterKernel:
processSolarize(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSolarize(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int threshold = 128;
    for (int j = 0; j < 3; ++j) {
        if (data[idx + j] > threshold) {
            data[idx + j] = 255 - data[idx + j];
        }
    }
}
```

Filter #350 - SolarizeBlend

Solarization blended with frame history.

Technique: Solarization with temporal blending.

```
// Case 350 in unifiedFilterKernel:
processSolarizeBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSolarizeBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int threshold = 100 + (params.frame_count % 100);
    for (int j = 0; j < 3; ++j) {
        unsigned char solar = (data[idx + j] > threshold) ? (255 - data[idx + j]) : data[idx + j];
        data[idx + j] = (unsigned char)(0.6f * data[idx + j] + 0.4f * solar);
    }
}
```

Filter #351 - GammaBright

Applies a gamma curve greater than 1, brightening midtones while preserving blacks and whites. Creates a lifted, bright look.

Technique: Gamma > 1 brightness adjustment.

```
// Case 351 in unifiedFilterKernel:
processGammaBright(x, y, data, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processGammaBright(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float gamma = 0.7f;
    for (int j = 0; j < 3; ++j) {
        float normalized = data[idx + j] / 255.0f;
        data[idx + j] = (unsigned char)(powf(normalized, gamma) * 255.0f);
    }
}

```

Filter #352 - GammaDark

Applies a gamma curve less than 1, darkening midtones. Creates a moody, shadowed look.

Technique: $\text{Gamma} < 1$ darkness adjustment.

```

// Case 352 in unifiedFilterKernel:
processGammaDark(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGammaDark(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float gamma = 1.5f;
    for (int j = 0; j < 3; ++j) {
        float normalized = data[idx + j] / 255.0f;
        data[idx + j] = (unsigned char)(powf(normalized, gamma) * 255.0f);
    }
}

```

Filter #353 - ContrastBoost

Increases contrast by pushing values away from midpoint (128)-darks get darker, brights get brighter.

Technique: Midpoint-based contrast expansion.

```

// Case 353 in unifiedFilterKernel:
processContrastBoost(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processContrastBoost(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float contrast = 1.5f;
    for (int j = 0; j < 3; ++j) {
        float val = (data[idx + j] - 128) * contrast + 128;
        data[idx + j] = (unsigned char)fminf(255.0f, fmaxf(0.0f, val));
    }
}

```

Filter #354 - ContrastReduce

Decreases contrast by pulling values toward midpoint, creating a flat, washed-out look.

Technique: Midpoint-based contrast compression.

```
// Case 354 in unifiedFilterKernel:
processContrastReduce(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processContrastReduce(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float contrast = 0.6f;
    for (int j = 0; j < 3; ++j) {
        float val = (data[idx + j] - 128) * contrast + 128;
        data[idx + j] = (unsigned char)fminf(255.0f, fmaxf(0.0f, val));
    }
}
```

Filter #355 - EdgeGlowBlend

Applies edge detection then adds a glow around detected edges, blended with frame history.

Technique: Edge detection + glow + temporal blending.

```
// Case 355 in unifiedFilterKernel:
processEdgeGlowBlend(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processEdgeGlowBlend(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (x > 0 && x < width - 1 && y > 0 && y < height - 1 && prev) {
        int sum = 0;
        for (int j = 0; j < 3; ++j) {
            int gx = data[(y - 1) * step + (x + 1) * 4 + j] - data[(y - 1) * step + (x - 1) * 4 + j] + 2
    ↪ * data[y * step + (x + 1) * 4 + j] - 2 * data[y * step + (x - 1) * 4 + j] + data[(y + 1) * step
    ↪ + (x + 1) * 4 + j] - data[(y + 1) * step + (x - 1) * 4 + j];
            sum += abs(gx);
        }
        int edge = sum / 3;
        float blend = fminf(1.0f, edge / 100.0f);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - blend) + prev[idx + j] * blend);
        }
    }
}
```

Filter #356 - FrameBlendMulti

Blends multiple history frames (more than 3) together for heavy temporal smoothing.

Technique: Multi-frame temporal blend averaging.

```
// Case 356 in unifiedFilterKernel:
processFrameBlendMulti(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processFrameBlendMulti(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
```

```

    int idx = y * step + x * 4;
    int count = min(params.numFrames, 4);
    for (int j = 0; j < 3; ++j) {
        int sum = data[idx + j];
        for (int f = 0; f < count; ++f) {
            if (allFrames[f])
                sum += allFrames[f][idx + j];
        }
        data[idx + j] = (unsigned char)(sum / (count + 1));
    }
}

```

Filter #357 - FrameBlendMultiX

Extended multi-frame blend with more history frames included.

Technique: Extended multi-frame temporal averaging.

```

// Case 357 in unifiedFilterKernel:
processFrameBlendMultiX(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processFrameBlendMultiX(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int count = min(params.numFrames, 8);
    float weights[9] = {1.0f, 0.9f, 0.8f, 0.7f, 0.6f, 0.5f, 0.4f, 0.3f, 0.2f};
    float total_weight = weights[0];
    for (int j = 0; j < 3; ++j) {
        float sum = data[idx + j] * weights[0];
        for (int f = 0; f < count; ++f) {
            if (allFrames[f]) {
                sum += allFrames[f][idx + j] * weights[f + 1];
                if (j == 0)
                    total_weight += weights[f + 1];
            }
        }
        data[idx + j] = (unsigned char)(sum / total_weight);
    }
}

```

Filter #358 - AcidTrailsBlend

Acid-style trails with strong color saturation and wave distortion blended together-a signature psychedelic trail effect.

Technique: Saturated wave-distorted temporal trails.

```

// Case 358 in unifiedFilterKernel:
processAcidTrailsBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processAcidTrailsBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {

```

```

        float factor = 0.5f + 0.3f * sinf(params.frame_count * 0.05f);
        for (int j = 0; j < 3; ++j) {
            int val = (int)(data[idx + j] * factor + prev[idx + j] * (1.0f - factor));
            data[idx + j] = (unsigned char)fminf(255.0f, (float)val);
        }
    }
}

```

Filter #359 - AcidGlitchX

Acid-colored glitch artifacts with strong color shifts and random displacement, creating a vivid digital acid trip.

Technique: Color-boosted random glitch displacement.

```

// Case 359 in unifiedFilterKernel:
processAcidGlitchX(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processAcidGlitchX(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 16, y / 16, params.seed);
    if (r < 0.15f) {
        int shift_x = (int)(r * 50) - 25;
        int shift_y = (int)(r * 30) - 15;
        int src_x = (x + shift_x + width) % width;
        int src_y = (y + shift_y + height) % height;
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}

```

Filter #360 - AlphaXorBlend

XOR blending controlled by alpha parameter—at low alpha, subtle XOR artifacts; at high alpha, full digital distortion.

Technique: Alpha-parameterized XOR blending.

```

// Case 360 in unifiedFilterKernel:
processAlphaXorBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processAlphaXorBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            unsigned char xored = data[idx + j] ^ prev[idx + j];
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + xored * 0.5f);
        }
    }
}

```

Filter #361 - AlphaXorBlendDouble

Double-intensity alpha XOR blend for more aggressive distortion.

Technique: Intensity-doubled alpha XOR blending.

```
// Case 361 in unifiedFilterKernel:
processAlphaXorBlendDouble(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processAlphaXorBlendDouble(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev1 = allFrames[0];
    unsigned char *prev2 = (params.numFrames > 1) ? allFrames[1] : NULL;
    if (prev1 && prev2) {
        for (int j = 0; j < 3; ++j) {
            unsigned char xored = data[idx + j] ^ prev1[idx + j] ^ prev2[idx + j];
            data[idx + j] = (unsigned char)(data[idx + j] * 0.3f + xored * 0.7f);
        }
    }
}
```

Filter #362 - AndOrXorStrobeScale

Cycles through AND, OR, XOR with additional per-channel scaling, creating an evolving multi-phase bitwise strobe.

Technique: Cycling AND/OR/XOR with scaling.

```
// Case 362 in unifiedFilterKernel:
processAndOrXorStrobeScale(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAndOrXorStrobeScale(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int mode = params.frame_count % 3;
    float scale = 0.8f + 0.4f * sinf(params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j) {
        unsigned char val = data[idx + j];
        if (mode == 0)
            val = val & (unsigned char)(params.sumR);
        else if (mode == 1)
            val = val | (unsigned char)(params.sumG);
        else
            val = val ^ (unsigned char)(params.sumB);
        data[idx + j] = (unsigned char)fminf(255.0f, val * scale);
    }
}
```

Filter #363 - AveragePixelsXorBlend

Averages neighboring pixels then XORs with frame history, creating a blurred-XOR hybrid.

Technique: Spatial average + history XOR blend.


```
// Case 363 in unifiedFilterKernel:
processAveragePixelsXorBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAveragePixelsXorBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)((data[idx + j] ^ avg) * 0.5f + data[idx + j] * 0.5f);
    }
}
```

Filter #364 - BitwiseRotateBlend

Bit-rotates pixel channel values and blends with history, creating subtle binary-level distortion.

Technique: Bit rotation + temporal blending.

```
// Case 364 in unifiedFilterKernel:
processBitwiseRotateBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBitwiseRotateBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = params.frame_count % 8;
    for (int j = 0; j < 3; ++j) {
        unsigned char val = data[idx + j];
        unsigned char rotated = (val << shift) | (val >> (8 - shift));
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + rotated * 0.5f);
    }
}
```

Filter #365 - BitwiseRotateDiffBlend

Bit rotation of the frame difference between current and history, highlighting motion with binary artifacts.

Technique: Bit-rotated frame difference blend.

```
// Case 365 in unifiedFilterKernel:
processBitwiseRotateDiffBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBitwiseRotateDiffBlend(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    int shift = params.frame_count % 8;
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            unsigned char val = data[idx + j];
            unsigned char rotated = (val << shift) | (val >> (8 - shift));
            int diff = abs(rotated - prev[idx + j]);
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + diff * 0.5f);
        }
    }
}
```

```
}  
}
```

Filter #366 - BitwiseXorScaleBlend

XOR with animated scale factor and frame history blending.

Technique: Scaled XOR + temporal blend.

```
// Case 366 in unifiedFilterKernel:  
processBitwiseXorScaleBlend(x, y, data, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processBitwiseXorScaleBlend(int x, int y, unsigned char *data, size_t step, const  
    ↪ FilterParams &params) {  
    int idx = y * step + x * 4;  
    float scale = 0.5f + 0.5f * sinf(params.frame_count * 0.05f);  
    for (int j = 0; j < 3; ++j) {  
        unsigned char val = data[idx + j];  
        unsigned char xored = val ^ (unsigned char)(params.sumR + j * 30);  
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * (1.0f - scale) + xored * scale);  
    }  
}
```

Filter #367 - BlackAndWhiteStrobe

Alternates rapidly between a grayscale and a full-color version of the image.

Technique: Alternating desaturation strobe.

```
// Case 367 in unifiedFilterKernel:  
processBlackAndWhiteStrobe(x, y, data, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processBlackAndWhiteStrobe(int x, int y, unsigned char *data, size_t step, const  
    ↪ FilterParams &params) {  
    int idx = y * step + x * 4;  
    int strobe = params.frame_count % 10;  
    if (strobe < 3) {  
        int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;  
        for (int j = 0; j < 3; ++j) {  
            data[idx + j] = (unsigned char)avg;  
        }  
    }  
}
```

Filter #368 - BlendAlphaXorScale

Alpha-blended XOR with additional scaling for controlled-intensity digital distortion.

Technique: Scaled alpha XOR blend.

```
// Case 368 in unifiedFilterKernel:  
processBlendAlphaXorScale(x, y, data, step, params);
```

```
// Referenced __device__ function(s):
__device__ void processBlendAlphaXorScale(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 0.5f + 0.5f * cosf(params.frame_count * 0.03f);
    for (int j = 0; j < 3; ++j) {
        unsigned char xored = data[idx + j] ^ (unsigned char)(params.sumB);
        data[idx + j] = (unsigned char)(data[idx + j] * alpha + xored * (1.0f - alpha));
    }
}
```

Filter #369 - BlendBuriedXor

Blurs the frame, then XORs with history, creating a soft-focus digital distortion.

Technique: Blur + XOR blend.

```
// Case 369 in unifiedFilterKernel:
processBlendBuriedXor(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlendBuriedXor(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            unsigned char blurred = (unsigned char)((data[idx + j] + prev[idx + j]) / 2);
            data[idx + j] = blurred ^ data[idx + j];
        }
    }
}
```

Filter #370 - BlendCombinedXor

Multi-layer combined XOR using several history frames, creating dense layered digital artifacts.

Technique: Multi-frame combined XOR.

```
// Case 370 in unifiedFilterKernel:
processBlendCombinedXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendCombinedXor(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char combined = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.6f + (data[idx + j] ^ combined) * 0.4f);
    }
}
```

Filter #371 - BlendIncreaseRGB

Increases each R, G, B channel incrementally while blending with history, creating a color-accumulating glow.

Technique: Incremental per-channel boost + blending.

```
// Case 371 in unifiedFilterKernel:
processBlendIncreaseRGB(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendIncreaseRGB(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float inc = (params.frame_count % 128) / 128.0f;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] * (1.0f + inc * 0.3f));
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * (1.0f + inc * 0.2f));
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * (1.0f + inc * 0.1f));
}
```

Filter #372 - BlendThreeXor

XOR of three different frame history layers with the current frame, creating triple-source digital distortion.

Technique: Three-frame XOR combination.

```
// Case 372 in unifiedFilterKernel:
processBlendThreeXor(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlendThreeXor(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *p1 = allFrames[0];
    unsigned char *p2 = (params.numFrames > 1) ? allFrames[1] : NULL;
    if (p1 && p2) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (data[idx + j] ^ p1[idx + j] ^ p2[idx + j]);
        }
    }
}
```

Filter #373 - BlurDistortionBlend

Combines spatial blur with coordinate distortion and frame blending—a triple-effect combo.

Technique: Blur + distortion + temporal blend.

```
// Case 373 in unifiedFilterKernel:
processBlurDistortionBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlurDistortionBlend(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float distort = sinf((x + params.frame_count) * 0.05f) * 5.0f;
    int src_y = (int)(y + distort);
}
```

```

    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[src_idx + j] * 0.5f);
    }
}

```

Filter #374 - ColorAccumulate

Accumulates color channel values over time, letting brightness build up in a feedback-like way.

Technique: Temporal color accumulation.

```

// Case 374 in unifiedFilterKernel:
processColorAccumulate(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorAccumulate(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float factor = (params.frame_count % 64) / 64.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * (1.0f + factor * 0.5f));
    }
}

```

Filter #375 - ColorAccumulateBlend

Color accumulation with frame history blending for controlled build-up.

Technique: Blended color accumulation.

```

// Case 375 in unifiedFilterKernel:
processColorAccumulateBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorAccumulateBlend(int x, int y, unsigned char *data, unsigned char **allFrames
    ↪ , size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    float factor = (params.frame_count % 64) / 64.0f;
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            int accum = (int)(data[idx + j] * (1.0f + factor * 0.3f));
            data[idx + j] = (unsigned char)fminf(255.0f, accum * 0.5f + prev[idx + j] * 0.5f);
        }
    }
}

```

Filter #376 - ColorAccumulateXor

Color accumulation combined with XOR operations, creating an evolving digital buildup.

Technique: XOR-modulated color accumulation.

```
// Case 376 in unifiedFilterKernel:
processColorAccumulateXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorAccumulateXor(int x, int y, unsigned char *data, size_t step, const
↳ FilterParams &params) {
    int idx = y * step + x * 4;
    float factor = (params.frame_count % 64) / 64.0f;
    for (int j = 0; j < 3; ++j) {
        unsigned char accum = (unsigned char)fminf(255.0f, data[idx + j] * (1.0f + factor * 0.5f));
        data[idx + j] = data[idx + j] ^ accum;
    }
}
```

Filter #377 - ColorChannelBlend

Blends individual color channels with corresponding channels from history frames.

Technique: Per-channel temporal blending.

```
// Case 377 in unifiedFilterKernel:
processColorChannelBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorChannelBlend(int x, int y, unsigned char *data, size_t step, const
↳ FilterParams &params) {
    int idx = y * step + x * 4;
    int channel = params.frame_count % 3;
    float blend = 0.5f + 0.5f * sinf(params.frame_count * 0.08f);
    unsigned char avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    data[idx + channel] = (unsigned char)(data[idx + channel] * blend + avg * (1.0f - blend));
}
```

Filter #378 - ColorChannelXor

XORs individual channels with history frame channels.

Technique: Per-channel temporal XOR.

```
// Case 378 in unifiedFilterKernel:
processColorChannelXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorChannelXor(int x, int y, unsigned char *data, size_t step, const
↳ FilterParams &params) {
    int idx = y * step + x * 4;
    int channel = params.frame_count % 3;
    data[idx + channel] ^= (unsigned char)(params.sumR);
}
```

Filter #379 - ColorCollectionEnergy

Energized/boosted color values collected from multiple history frames, creating a vibrant multi-frame composite.

Technique: Boosted multi-frame color collection.

```
// Case 379 in unifiedFilterKernel:
processColorCollectionEnergy(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollectionEnergy(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            int diff = abs(data[idx + j] - prev[idx + j]);
            int energy = data[idx + j] + diff / 2;
            data[idx + j] = (unsigned char)fminf(255.0f, (float)energy);
        }
    }
}
```

Filter #380 - ColorCollectionWave

Collects colors from history frames using wave-modulated selection indices, creating a pulsating temporal color composite.

Technique: Wave-indexed multi-frame color collection.

```
// Case 380 in unifiedFilterKernel:
processColorCollectionWave(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorCollectionWave(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf((x + y + params.frame_count) * 0.03f);
    for (int j = 0; j < 3; ++j) {
        float factor = 0.8f + 0.2f * wave * (j + 1) / 3.0f;
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
    }
}
```

Filter #381 - ColorFadeXor

Gradual color fade combined with XOR operations for a morphing digital transition.

Technique: Fading XOR blend.

```
// Case 381 in unifiedFilterKernel:
processColorFadeXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorFadeXor(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float fade = (params.frame_count % 128) / 128.0f;
    for (int j = 0; j < 3; ++j) {
        unsigned char faded = (unsigned char)(data[idx + j] * fade);
        data[idx + j] = data[idx + j] ^ faded;
    }
}
```

```
}
```

Filter #382 - ColorIntensityBlend

Blends with frame history weighted by pixel color intensity-brighter pixels blend more.

Technique: Luminance-weighted temporal blending.

```
// Case 382 in unifiedFilterKernel:
processColorIntensityBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorIntensityBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int intensity = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float factor = intensity / 255.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + 0.5f * factor));
    }
}
```

Filter #383 - ColorIntensityXor

XOR operations weighted by luminance for brightness-dependent digital distortion.

Technique: Luminance-weighted XOR.

```
// Case 383 in unifiedFilterKernel:
processColorIntensityXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorIntensityXor(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int intensity = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] ^= (unsigned char)intensity;
    }
}
```

Filter #384 - ColorMoveBlend

Shifts colors in a direction while blending with history, creating moving color trails.

Technique: Directional color shift + blend.

```
// Case 384 in unifiedFilterKernel:
processColorMoveBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processColorMoveBlend(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = params.frame_count % 32;
```



```

    int src_x = (x + offset) % width;
    int src_y = (y + offset) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.6f + data[src_idx + j] * 0.4f);
    }
}

```

Filter #385 - ColorPixelBlend

Per-pixel conditional color blending based on color values.

Technique: Conditional per-pixel color blend.

```

// Case 385 in unifiedFilterKernel:
processColorPixelBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorPixelBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.3f) {
        unsigned char avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.7f + avg * 0.3f);
        }
    }
}

```

Filter #386 - ColorPixelXor

Per-pixel conditional XOR based on color values.

Technique: Conditional per-pixel XOR.

```

// Case 386 in unifiedFilterKernel:
processColorPixelXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorPixelXor(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.25f) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= (unsigned char)(r * 255);
        }
    }
}

```

Filter #387 - ColorScaleBlend

Scales colors dynamically while blending with history.

Technique: Dynamic color scaling + blend.

```
// Case 387 in unifiedFilterKernel:
processColorScaleBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorScaleBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float scale = 0.7f + 0.6f * sinf(params.frame_count * 0.04f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * scale);
    }
}
```

Filter #388 - ColorWaveXor

Wave-modulated XOR per channel, creating sine-pattern digital artifacts.

Technique: Sine-wave XOR per channel.

```
// Case 388 in unifiedFilterKernel:
processColorWaveXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorWaveXor(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float wave = sinf((x + y + params.frame_count) * 0.05f);
    unsigned char xorVal = (unsigned char)(128 + 127 * wave);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] ^= xorVal;
    }
}
```

Filter #389 - CosineMultiplyBlend

Multiplies pixel values by cosine-derived factors and blends with history, creating smooth oscillating brightness variations.

Technique: Cosine-weighted multiply + blend.

```
// Case 389 in unifiedFilterKernel:
processCosineMultiplyBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCosineMultiplyBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float cos_val = cosf((x + y) * 0.02f + params.frame_count * 0.03f);
    float factor = 0.7f + 0.3f * cos_val;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
    }
}
```

Filter #390 - DarkModBlendXor

Dark modular blending with additional XOR, creating dark moody artifacts.

Technique: Dark modular blend + XOR.

```
// Case 390 in unifiedFilterKernel:
processDarkModBlendXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDarkModBlendXor(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        unsigned char dark = (unsigned char)(data[idx + j] * 0.5f);
        data[idx + j] = (data[idx + j] ^ dark);
    }
}
```

Filter #391 - DifferenceBlend

Computes absolute difference between current and history frames and blends, highlighting motion as bright areas.

Technique: Frame difference blend.

```
// Case 391 in unifiedFilterKernel:
processDifferenceBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDifferenceBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            int diff = abs(data[idx + j] - prev[idx + j]);
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + diff * 0.5f);
        }
    }
}
```

Filter #392 - DifferenceXorBlend

Frame difference combined with XOR for dual-layer motion-highlighting digital distortion.

Technique: Frame difference + XOR blend.

```
// Case 392 in unifiedFilterKernel:
processDifferenceXorBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDifferenceXorBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
```

```

        int diff = abs(data[idx + j] - prev[idx + j]);
        data[idx + j] = (unsigned char)((data[idx + j] ^ diff) * 0.7f + data[idx + j] * 0.3f);
    }
}

```

Filter #393 - DistortBlend

Coordinate distortion (sine warp) combined with multi-frame blending.

Technique: Sine-distortion + temporal blend.

```

// Case 393 in unifiedFilterKernel:
processDistortBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDistortBlend(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float dx = sinf(y * 0.03f + params.frame_count * 0.05f) * 10.0f;
    float dy = cosf(x * 0.03f + params.frame_count * 0.05f) * 10.0f;
    int src_x = ((int)(x + dx) % width + width) % width;
    int src_y = ((int)(y + dy) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[src_idx + j] * 0.5f);
    }
}

```

Filter #394 - DiamondPatternBlend

Diamond pattern overlay blended with frame history for a tiled temporal effect.

Technique: Diamond pattern + temporal blend.

```

// Case 394 in unifiedFilterKernel:
processDiamondPatternBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDiamondPatternBlend(int x, int y, unsigned char *data, int width, int height,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    int diamond = abs(x - cx) + abs(y - cy);
    float factor = 0.5f + 0.5f * sinf(diamond * 0.02f + params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}

```

Filter #395 - FadeBlendXor

Gradual fade combined with XOR and blending for a smooth evolving digital distortion.

Technique: Fade + XOR + blend.

```
// Case 395 in unifiedFilterKernel:
processFadeBlendXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFadeBlendXor(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float fade = (params.frame_count % 256) / 256.0f;
    for (int j = 0; j < 3; ++j) {
        unsigned char faded = (unsigned char)(data[idx + j] * fade);
        data[idx + j] = (unsigned char)((data[idx + j] ^ faded) * 0.5f + data[idx + j] * 0.5f);
    }
}
```

Filter #396 - FlashBlendXor

Bright flash with XOR and blending, creating periodic digital flash bursts.

Technique: Flash + XOR + blend.

```
// Case 396 in unifiedFilterKernel:
processFlashBlendXor(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFlashBlendXor(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int flash = params.frame_count % 20;
    if (flash < 5) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] ^= 0xFF;
        }
    } else {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.9f);
        }
    }
}
```

Filter #397 - GhostTrailsBlend

Layered ghost trails from multiple history frames blended together for smooth spectral motion echoes.

Technique: Multi-frame ghost trail blending.

```
// Case 397 in unifiedFilterKernel:
processGhostTrailsBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processGhostTrailsBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int count = min(params.numFrames, 4);
    for (int j = 0; j < 3; ++j) {
        float sum = data[idx + j];
    }
}
```

```

        float weight = 1.0f;
        float total_weight = weight;
        for (int f = 0; f < count; ++f) {
            if (allFrames[f]) {
                weight *= 0.7f;
                sum += allFrames[f][idx + j] * weight;
                total_weight += weight;
            }
        }
        data[idx + j] = (unsigned char)(sum / total_weight);
    }
}

```

Filter #398 - AddInvert

Adds the inverted (negative) version of frame history to the current frame, creating surreal doubled imagery with reversed tones.

Technique: Additive inverted frame blending.

```

// Case 398 in unifiedFilterKernel:
processAddInvert(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAddInvert(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        int val = data[idx + j] + (255 - data[idx + j]) / 2;
        data[idx + j] = (unsigned char)fminf(255.0f, (float)val);
    }
}

```

Filter #399 - AlphaBlendSimple

Simple 50/50 alpha blend between current frame and first history frame.

Technique: Simple half-and-half temporal blend.

```

// Case 399 in unifiedFilterKernel:
processAlphaBlendSimple(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processAlphaBlendSimple(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        float alpha = params.alpha;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * alpha + prev[idx + j] * (1.0f - alpha));
        }
    }
}

```

Extended Effects (400-499)

Filter #400 - AlphaBlendDoubleX

Double-intensity alpha blending with accelerated cross-fade animation between frames.

Technique: Intensity-doubled alpha blending.

```
// Case 400 in unifiedFilterKernel:
processAlphaBlendDoubleX(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processAlphaBlendDoubleX(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *p1 = allFrames[0];
    unsigned char *p2 = (params.numFrames > 1) ? allFrames[1] : NULL;
    if (p1 && p2) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.34f + p1[idx + j] * 0.33f + p2[idx + j] *
    ↪ 0.33f);
        }
    }
}
```

Filter #401 - AlphaStrobeBlendX

Strobing alpha blend that rapidly switches blend intensity, combined with frame history.

Technique: Strobed alpha blending.

```
// Case 401 in unifiedFilterKernel:
processAlphaStrobeBlendX(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processAlphaStrobeBlendX(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    int strobe = params.frame_count % 6;
    if (prev) {
        float blend = (strobe < 3) ? 0.3f : 0.7f;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * blend + prev[idx + j] * (1.0f - blend));
        }
    }
}
```

Filter #402 - BitwiseAndBlend

Bitwise AND between current and history frames with additional blending.

Technique: Bitwise AND + blend.

```
// Case 402 in unifiedFilterKernel:
processBitwiseAndBlend(x, y, data, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processBitwiseAndBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char mask = (unsigned char)(128 + 64 * sinf(params.frame_count * 0.05f));
    for (int j = 0; j < 3; ++j) {
        unsigned char anded = data[idx + j] & mask;
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + anded * 0.5f);
    }
}

```

Filter #403 - BitwiseXorAverage

XORs the pixel with the average of multiple history frames.

Technique: Multi-frame average XOR.

```

// Case 403 in unifiedFilterKernel:
processBitwiseXorAverage(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBitwiseXorAverage(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] ^= avg;
    }
}

```

Filter #404 - BitwiseXorBlendX

Extended XOR blend using more history frames for deeper artifact layering.

Technique: Extended multi-frame XOR blend.

```

// Case 404 in unifiedFilterKernel:
processBitwiseXorBlendX(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBitwiseXorBlendX(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            unsigned char xored = data[idx + j] ^ prev[idx + j];
            data[idx + j] = (unsigned char)(xored * 0.6f + data[idx + j] * 0.4f);
        }
    }
}

```

Filter #405 - BlackStrobe

Alternates between the normal image and a fully black frame, creating a harsh on/off strobe.

Technique: Alternating black frame strobe.

```
// Case 405 in unifiedFilterKernel:
processBlackStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlackStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int strobe = params.frame_count % 8;
    if (strobe < 2) {
        data[idx] = data[idx + 1] = data[idx + 2] = 0;
    }
}
```

Filter #406 - BlendAlphaXorX

Extended alpha XOR blend with additional complexity.

Technique: Extended alpha XOR blend.

```
// Case 406 in unifiedFilterKernel:
processBlendAlphaXorX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendAlphaXorX(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float alpha = 0.5f + 0.5f * sinf(params.frame_count * 0.06f);
    for (int j = 0; j < 3; ++j) {
        unsigned char xored = data[idx + j] ^ (unsigned char)(params.sumB + j * 40);
        data[idx + j] = (unsigned char)(data[idx + j] * alpha + xored * (1.0f - alpha));
    }
}
```

Filter #407 - BlendCombinedValuesX

Combines multiple blending operations-add, XOR, average-into a single compound filter.

Technique: Multi-operation combined blend.

```
// Case 407 in unifiedFilterKernel:
processBlendCombinedValuesX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendCombinedValuesX(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int combined = (data[idx] + data[idx + 1] + data[idx + 2]);
    float factor = (combined % 256) / 255.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.6f + 0.4f * factor));
    }
}
```

Filter #408 - BlendFor360

Blends across a large range of frame history (up to 360-degree coverage), creating a full-cycle temporal composite.

Technique: Wide-range temporal blending.

```
// Case 408 in unifiedFilterKernel:
processBlendFor360(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlendFor360(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float angle = atan2f((float)(y - height / 2), (float)(x - width / 2));
    float factor = 0.5f + 0.5f * sinf(angle * 4.0f + params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}
```

Filter #409 - BlendForward16

Blends 16 consecutive history frames forward, creating a 16-frame-deep motion blur.

Technique: 16-frame forward temporal blur.

```
// Case 409 in unifiedFilterKernel:
processBlendForward16(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlendForward16(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int frame_idx = params.frame_count % min(params.numFrames, 16);
    unsigned char *frame = allFrames[frame_idx];
    if (frame) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + frame[idx + j] * 0.5f);
        }
    }
}
```

Filter #410 - BlendForward32

Blends 32 consecutive history frames for a very deep, heavy motion blur.

Technique: 32-frame forward temporal blur.

```
// Case 410 in unifiedFilterKernel:
processBlendForward32(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlendForward32(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int frame_idx = params.frame_count % min(params.numFrames, 32);
    unsigned char *frame = allFrames[frame_idx];
```

```

    if (frame) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.6f + frame[idx + j] * 0.4f);
        }
    }
}

```

Filter #411 - BlendFromXtoY

Blends between two specific history frame indices, with the range animated over time.

Technique: Animated frame-range blending.

```

// Case 411 in unifiedFilterKernel:
processBlendFromXtoY(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendFromXtoY(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float blend = (params.frame_count % 256) / 255.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - blend) + (255 - data[idx + j]) * blend);
    }
}

```

Filter #412 - BlendIncreaseX

Blending with progressively increasing intensity over time.

Technique: Time-increasing blend intensity.

```

// Case 412 in unifiedFilterKernel:
processBlendIncreaseX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendIncreaseX(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float inc = 1.0f + (params.frame_count % 100) / 200.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * inc);
    }
}

```

Filter #413 - BlendRedGreenBlue

Selectively blends R, G, B channels from different history frames-red from one, green from another, blue from a third.

Technique: Per-channel history frame selection blend.

```

// Case 413 in unifiedFilterKernel:
processBlendRedGreenBlue(x, y, data, step, params);

```

```
// Referenced __device__ function(s):
__device__ void processBlendRedGreenBlue(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int cycle = params.frame_count % 3;
    float factors[3] = {0.7f, 0.8f, 0.9f};
    factors[cycle] = 1.2f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factors[j]);
    }
}
```

Filter #414 - BlendWithColorX

Blends with an animated solid color that cycles through hues over time.

Technique: Animated solid color blend.

```
// Case 414 in unifiedFilterKernel:
processBlendWithColorX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBlendWithColorX(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char colors[3] = {(unsigned char)params.sumR, (unsigned char)params.sumG, (unsigned char)
    ↪ params.sumB};
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.7f + colors[j] * 0.3f);
    }
}
```

Filter #415 - BlendAngle

Blends frames at angle-derived positions, creating a rotational gradient between time points.

Technique: Angle-based frame blending.

```
// Case 415 in unifiedFilterKernel:
processBlendAngle(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlendAngle(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float angle = atan2f((float)(y - height / 2), (float)(x - width / 2));
    float norm = (angle + 3.14159f) / (2.0f * 3.14159f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + 0.5f * norm));
    }
}
```

Filter #416 - BlockScale

Scales pixel values within blocks, with different blocks getting different scale factors.

Technique: Per-block brightness scaling.

```
// Case 416 in unifiedFilterKernel:
processBlockScale(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlockScale(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int block_x = x / 16;
    int block_y = y / 16;
    float scale = 0.5f + 0.5f * sinf((block_x + block_y + params.frame_count) * 0.2f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * scale);
    }
}
```

Filter #417 - BlockStrobe

Alternates block visibility in a tiled pattern, creating a checkerboard strobe.

Technique: Block-tiled strobing.

```
// Case 417 in unifiedFilterKernel:
processBlockStrobe(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlockStrobe(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int block_x = x / 32;
    int block_y = y / 32;
    int strobe = (block_x + block_y + params.frame_count) % 4;
    if (strobe == 0) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(255 - data[idx + j]);
        }
    }
}
```

Filter #418 - BlockXor

XORs pixels within tiled blocks with history frame blocks.

Technique: Block-tiled XOR.

```
// Case 418 in unifiedFilterKernel:
processBlockXor(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlockXor(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int block_x = x / 16;
    int block_y = y / 16;
    unsigned char xor_val = (unsigned char)((block_x ^ block_y ^ params.frame_count) % 256);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] ^= xor_val;
    }
}
```

```
}  
}
```

Filter #419 - BlockyTrails16

Creates blocky 16x16 pixel motion trails from frame history, like trailing pixelated afterimages.

Technique: 16x16 block temporal trailing.

```
// Case 419 in unifiedFilterKernel:  
processBlockyTrails16(x, y, data, allFrames, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processBlockyTrails16(int x, int y, unsigned char *data, unsigned char **allFrames, int  
    ↪ width, int height, size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    int block_y = y / 16;  
    int frame_idx = block_y % min(params.numFrames, 8);  
    unsigned char *frame = allFrames[frame_idx];  
    if (frame) {  
        for (int j = 0; j < 3; ++j) {  
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + frame[idx + j] * 0.5f);  
        }  
    }  
}
```

Filter #420 - BlockyTrails32

32x32 pixel block trails from frame history-larger, more prominent blocky afterimages.

Technique: 32x32 block temporal trailing.

```
// Case 420 in unifiedFilterKernel:  
processBlockyTrails32(x, y, data, allFrames, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processBlockyTrails32(int x, int y, unsigned char *data, unsigned char **allFrames, int  
    ↪ width, int height, size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    int block_y = y / 32;  
    int frame_idx = block_y % min(params.numFrames, 8);  
    unsigned char *frame = allFrames[frame_idx];  
    if (frame) {  
        for (int j = 0; j < 3; ++j) {  
            data[idx + j] = (unsigned char)(data[idx + j] * 0.6f + frame[idx + j] * 0.4f);  
        }  
    }  
}
```

Filter #421 - BlurDistortionX

Blur combined with sine-based positional distortion for a ghostly, warped soft-focus look.

Technique: Spatial blur + coordinate distortion.

```
// Case 421 in unifiedFilterKernel:
processBlurDistortionX(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlurDistortionX(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float dx = sinf((y + params.frame_count) * 0.03f) * 8.0f;
    float dy = cosf((x + params.frame_count) * 0.03f) * 8.0f;
    int src_x = ((int)(x + dx) % width + width) % width;
    int src_y = ((int)(y + dy) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[src_idx + j] * 0.5f);
    }
}
```

Filter #422 - CannyStrobe

Canny-style edge detection that strobes on and off, alternating between the outline and the full image.

Technique: Alternating edge detection display.

```
// Case 422 in unifiedFilterKernel:
processCannyStrobe(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processCannyStrobe(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int strobe = params.frame_count % 4;
    if (strobe < 2 && x > 0 && x < width - 1 && y > 0 && y < height - 1) {
        int gx = 0, gy = 0;
        for (int j = 0; j < 3; ++j) {
            int val_left = data[y * step + (x - 1) * 4 + j];
            int val_right = data[y * step + (x + 1) * 4 + j];
            int val_up = data[(y - 1) * step + x * 4 + j];
            int val_down = data[(y + 1) * step + x * 4 + j];
            gx += abs(val_right - val_left);
            gy += abs(val_down - val_up);
        }
        int edge = min(255, (gx + gy) / 3);
        data[idx] = data[idx + 1] = data[idx + 2] = (unsigned char)edge;
    }
}
```

Filter #423 - ColorFadeSlow

Very slow color fade toward a target hue, creating a gradual tinting effect over many frames.

Technique: Slow time-interpolated color tint.

```
// Case 423 in unifiedFilterKernel:
processColorFadeSlow(x, y, data, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processColorFadeSlow(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float fade = (params.frame_count % 512) / 512.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + 0.5f * sinf(fade * 6.28f + j * 2.0f)));
    }
}

```

Filter #424 - ColorFibonacci

Selects colors from a Fibonacci-indexed pattern, creating mathematically-beautiful color progression.

Technique: Fibonacci-sequence color selection.

```

// Case 424 in unifiedFilterKernel:
processColorFibonacci(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorFibonacci(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int fib[8] = {1, 1, 2, 3, 5, 8, 13, 21};
    int fib_idx = (x + y + params.frame_count) % 8;
    float factor = 0.5f + 0.5f * (fib[fib_idx] / 21.0f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}

```

Filter #425 - CurtainEffect

Opens/closes like a curtain-a vertical split wipe that reveals/hides frame history behind the current frame.

Technique: Animated vertical split transition.

```

// Case 425 in unifiedFilterKernel:
processCurtainEffect(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processCurtainEffect(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int curtain_pos = (params.frame_count * 4) % width;
    if (x < curtain_pos) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.3f);
        }
    }
}

```

Filter #426 - DarkColorFibonacci

Dark-variant Fibonacci color selection for a moody, mathematically-ordered color palette.

Technique: Dark Fibonacci-sequence color selection.

```
// Case 426 in unifiedFilterKernel:
processDarkColorFibonacci(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDarkColorFibonacci(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int fib[8] = {1, 1, 2, 3, 5, 8, 13, 21};
    int fib_idx = (x + y + params.frame_count) % 8;
    float factor = 0.3f + 0.4f * (fib[fib_idx] / 21.0f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}
```

Filter #427 - DarkColorsBlend

Blends selectively with darker color values from history, biasing toward shadows.

Technique: Dark-biased selective blending.

```
// Case 427 in unifiedFilterKernel:
processDarkColorsBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDarkColorsBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        if (data[idx + j] > 128) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.6f);
        } else {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.9f);
        }
    }
}
```

Filter #428 - EnergizeBlend

Amplifies the difference between current and history frames, creating high-energy emphasis on motion.

Technique: Amplified frame difference blending.

```
// Case 428 in unifiedFilterKernel:
processEnergizeBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processEnergizeBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            int diff = abs(data[idx + j] - prev[idx + j]);
            int energized = data[idx + j] + diff;
        }
    }
}
```

```

        data[idx + j] = (unsigned char)fminf(255.0f, (float)energized);
    }
}

```

Filter #429 - AverageLines

Averages each row's pixels for a horizontally-smoothed, banded look.

Technique: Row-wise pixel averaging.

```

// Case 429 in unifiedFilterKernel:
processAverageLines(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processAverageLines(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int line_width = 8 + (params.frame_count % 16);
    int block = x / line_width;
    if (block % 2 == 0) {
        int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + avg * 0.5f);
        }
    }
}

```

Filter #430 - AverageLinesBlendX

Row averaging blended with frame history for temporal depth.

Technique: Row averaging + temporal blend.

```

// Case 430 in unifiedFilterKernel:
processAverageLinesBlendX(x, y, data, allFrames, width, step, params);

// Referenced __device__ function(s):
__device__ void processAverageLinesBlendX(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    int line_width = 16;
    int block = x / line_width;
    if (prev && block % 2 == 0) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)((data[idx + j] + prev[idx + j]) / 2);
        }
    }
}

```

Filter #431 - BlendRowAlpha

Alpha-blends specific rows with history frames, creating selective horizontal band blending.

Technique: Row-selective alpha blending.

```
// Case 431 in unifiedFilterKernel:
processBlendRowAlpha(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlendRowAlpha(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float row_alpha = (float)(y % 64) / 64.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.6f + 0.4f * row_alpha));
    }
}
```

Filter #432 - BlendInOut

Progressive fade in/out effect cycling between full opacity and transparency.

Technique: Cycling opacity modulation.

```
// Case 432 in unifiedFilterKernel:
processBlendInOut(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processBlendInOut(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    float cycle = 0.5f + 0.5f * sinf(params.frame_count * 0.03f);
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * cycle + prev[idx + j] * (1.0f - cycle));
        }
    }
}
```

Filter #433 - ColorFlashIncreaseX

Color flash with increasing intensity over time-flashes get progressively brighter.

Technique: Time-increasing color flash.

```
// Case 433 in unifiedFilterKernel:
processColorFlashIncreaseX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorFlashIncreaseX(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int flash = params.frame_count % 30;
    float inc = 1.0f + (flash < 10 ? flash * 0.05f : 0.0f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * inc);
    }
}
```

Filter #434 - ColorIncreaseInOut

Color intensity increases then decreases in a cycle, creating a breathing color effect.

Technique: Cycling color intensity modulation.

```
// Case 434 in unifiedFilterKernel:
processColorIncreaseInOut(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorIncreaseInOut(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float cycle = 0.5f + 0.5f * sinf(params.frame_count * 0.04f);
    float inc = 0.8f + 0.4f * cycle;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * inc);
    }
}
```

Filter #435 - ColorLinesX

Horizontal color lines with variable color and spacing, animated over time.

Technique: Animated horizontal colored lines.

```
// Case 435 in unifiedFilterKernel:
processColorLinesX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorLinesX(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int line = y % 8;
    if (line < 4) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.7f);
        }
    }
}
```

Filter #436 - ColorMoveDownX

Shifts color channels downward at different rates, creating vertical chromatic dragging.

Technique: Per-channel downward shift.

```
// Case 436 in unifiedFilterKernel:
processColorMoveDownX(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processColorMoveDownX(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = params.frame_count % height;
    int src_y = (y + offset) % height;
```

```

    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[src_idx + j] * 0.5f);
    }
}

```

Filter #437 - ColorOrderSwapX

Swaps the order of color channels with animation-cycling through all permutations.

Technique: Animated channel permutation cycling.

```

// Case 437 in unifiedFilterKernel:
processColorOrderSwapX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorOrderSwapX(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int mode = params.frame_count % 6;
    unsigned char b = data[idx], g = data[idx + 1], r = data[idx + 2];
    switch (mode) {
    case 0:
        data[idx] = r;
        data[idx + 1] = g;
        data[idx + 2] = b;
        break;
    case 1:
        data[idx] = g;
        data[idx + 1] = b;
        data[idx + 2] = r;
        break;
    case 2:
        data[idx] = b;
        data[idx + 1] = r;
        data[idx + 2] = g;
        break;
    case 3:
        data[idx] = r;
        data[idx + 1] = b;
        data[idx + 2] = g;
        break;
    case 4:
        data[idx] = g;
        data[idx + 1] = r;
        data[idx + 2] = b;
        break;
    default:
        break;
    }
}

```

Filter #438 - ColorPulseAlphaX

Color pulse modulated by alpha parameter for controllable pulse intensity.

Technique: Alpha-controlled color pulsing.

```
// Case 438 in unifiedFilterKernel:
processColorPulseAlphaX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorPulseAlphaX(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.08f);
    float alpha = params.alpha * pulse;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * alpha + data[idx + j] * (1.0f - alpha) * 0.5f);
    }
}
```

Filter #439 - ColorRowShiftX

Shifts colors differently on each row, creating horizontal rainbow banding.

Technique: Per-row color channel shifting.

```
// Case 439 in unifiedFilterKernel:
processColorRowShiftX(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processColorRowShiftX(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = (y + params.frame_count) % 32;
    int src_x = (x + shift) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[src_idx + j] * 0.5f);
    }
}
```

Filter #440 - ColorShiftXorX

Color shifting combined with XOR operations for animated digital color distortion.

Technique: Color shift + XOR.

```
// Case 440 in unifiedFilterKernel:
processColorShiftXorX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorShiftXorX(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int shift = params.frame_count % 8;
    for (int j = 0; j < 3; ++j) {
        unsigned char shifted = (data[idx + j] << shift) | (data[idx + j] >> (8 - shift));
        data[idx + j] ^= shifted;
    }
}
```

Filter #441 - CopyXorAlphaX

Copies frame with XOR and alpha blending for a layered digital copy effect.

Technique: Copy + XOR + alpha blend.

```
// Case 441 in unifiedFilterKernel:
processCopyXorAlphaX(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processCopyXorAlphaX(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            unsigned char xored = data[idx + j] ^ prev[idx + j];
            data[idx + j] = (unsigned char)(xored * params.alpha + data[idx + j] * (1.0f - params.alpha)
    ↪ );
        }
    }
}
```

Filter #442 - CycleShiftRGBX

Cycles through all 6 RGB permutations over time, creating a continuously rotating false-color animation.

Technique: Cycling RGB channel permutation.

```
// Case 442 in unifiedFilterKernel:
processCycleShiftRGBX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCycleShiftRGBX(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int cycle = params.frame_count % 3;
    unsigned char temp[3] = {data[idx], data[idx + 1], data[idx + 2]};
    data[idx] = temp[(0 + cycle) % 3];
    data[idx + 1] = temp[(1 + cycle) % 3];
    data[idx + 2] = temp[(2 + cycle) % 3];
}
```

Filter #443 - DarkNegateX

Darkens the image then applies partial negation, creating a dark, inverted-tone look.

Technique: Darken + partial inversion.

```
// Case 443 in unifiedFilterKernel:
processDarkNegateX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDarkNegateX(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
```

```

    if (avg < 100) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(255 - data[idx + j]);
        }
    }
}

```

Filter #444 - DarkSelfAlphaX

Dark self-alpha blend creating a moody, dimmed version of the self-blend effect.

Technique: Dark alpha self-blend.

```

// Case 444 in unifiedFilterKernel:
processDarkSelfAlphaX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDarkSelfAlphaX(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float alpha = (avg < 128) ? 0.5f : 1.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * alpha);
    }
}

```

Filter #445 - DiagonalGlitch

Glitches along diagonal lines-displacing pixel data diagonally for a sheared glitch look.

Technique: Diagonal line displacement.

```

// Case 445 in unifiedFilterKernel:
processDiagonalGlitch(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDiagonalGlitch(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int diag = (x + y + params.frame_count) % 64;
    if (diag < 8) {
        int shift = diag * 2;
        int src_x = (x + shift) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}

```

Filter #446 - DigitalHaze

Soft digital haze/fog effect using blurred history frame overlay.

Technique: Blurred history overlay for haze.

```
// Case 446 in unifiedFilterKernel:
processDigitalHaze(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDigitalHaze(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float haze = 0.1f + 0.1f * sinf((x + y) * 0.02f + params.frame_count * 0.03f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - haze) + 200 * haze);
    }
}
```

Filter #447 - DoubleXorBlend

XOR between current and history frames applied twice for deeper, more complex digital artifacts.

Technique: Double-pass XOR blend.

```
// Case 447 in unifiedFilterKernel:
processDoubleXorBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processDoubleXorBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *p1 = allFrames[0];
    unsigned char *p2 = (params.numFrames > 1) ? allFrames[1] : NULL;
    if (p1 && p2) {
        for (int j = 0; j < 3; ++j) {
            unsigned char xor1 = data[idx + j] ^ p1[idx + j];
            unsigned char xor2 = xor1 ^ p2[idx + j];
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + xor2 * 0.5f);
        }
    }
}
```

Filter #448 - EchoBlend

Creates an echo/delay effect by blending progressively older frames with decreasing opacity-like an audio echo but for video.

Technique: Multi-frame diminishing-weight echo blend.

```
// Case 448 in unifiedFilterKernel:
processEchoBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processEchoBlend(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int numEcho = min(params.numFrames, 4);
    for (int j = 0; j < 3; ++j) {
        float sum = data[idx + j];
        float weight = 1.0f;
        for (int f = 0; f < numEcho; ++f) {
```

```

        if (allFrames[f]) {
            weight *= 0.6f;
            sum += allFrames[f][idx + j] * weight;
        }
    }
    data[idx + j] = (unsigned char)(sum / (1.0f + weight * numEcho * 0.3f));
}
}

```

Filter #449 - ElectricEdge

Applies Sobel edge detection then renders edges as bright, jagged electric-looking lines.

Technique: Sobel edges rendered as electric arcs.

```

// Case 449 in unifiedFilterKernel:
processElectricEdge(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processElectricEdge(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0 && x < width - 1 && y > 0 && y < height - 1) {
        int edge = 0;
        for (int j = 0; j < 3; ++j) {
            int dx = abs(data[y * step + (x + 1) * 4 + j] - data[y * step + (x - 1) * 4 + j]);
            int dy = abs(data[(y + 1) * step + x * 4 + j] - data[(y - 1) * step + x * 4 + j]);
            edge += dx + dy;
        }
        edge /= 6;
        if (edge > 30) {
            int pulse = (params.frame_count % 10 < 5) ? 255 : 180;
            data[idx] = (unsigned char)fminf(255.0f, data[idx] * 0.3f + pulse * 0.7f);
            data[idx + 1] = (unsigned char)(data[idx + 1] * 0.5f);
            data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * 0.3f + pulse * 0.7f);
        }
    }
}

```

Filter #450 - FlashColorStrobe

Bright colored flash strobe that cycles through different colors on each flash.

Technique: Cycling color flash strobe.

```

// Case 450 in unifiedFilterKernel:
processFlashColorStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFlashColorStrobe(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int strobe = params.frame_count % 15;
    if (strobe < 3) {
        int channel = strobe % 3;
        data[idx + channel] = 255;
    }
}

```

Filter #451 - FrameDiffXor

XORs the frame difference (current minus previous) with the current frame, highlighting motion with digital artifacts.

Technique: Frame difference XOR.

```
// Case 451 in unifiedFilterKernel:
processFrameDiffXor(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processFrameDiffXor(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            int diff = abs(data[idx + j] - prev[idx + j]);
            data[idx + j] ^= (unsigned char)diff;
        }
    }
}
```

Filter #452 - GhostMirror

Ghost trails combined with horizontal mirroring-transparent echoes appear mirrored across the center.

Technique: Mirrored ghost trail overlay.

```
// Case 452 in unifiedFilterKernel:
processGhostMirror(x, y, data, allFrames, width, step, params);

// Referenced __device__ function(s):
__device__ void processGhostMirror(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int mirror_x = width - x - 1;
    int mirror_idx = y * step + mirror_x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + prev[mirror_idx + j] * 0.5f);
        }
    }
}
```

Filter #453 - GlitchSort

Sorts pixels/blocks by brightness or color value, creating a pixel-sorting glitch art effect.

Technique: Pixel/block brightness sorting.

```
// Case 453 in unifiedFilterKernel:
processGlitchSort(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processGlitchSort(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 32, y / 32, params.seed);
    if (r < 0.1f) {
        int offset = (int)(r * 20) - 10;
        int src_x = (x + offset + width) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}
```

Filter #454 - HeatWave

Simulates heat-shimmer distortion-vertical wavy displacement that increases from bottom to top, like hot rising air.

Technique: Bottom-up increasing vertical wave distortion.

```
// Case 454 in unifiedFilterKernel:
processHeatWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processHeatWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf((y + params.frame_count * 2) * 0.05f) * 5.0f;
    int src_x = ((int)(x + wave) % width + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.3f + data[src_idx + j] * 0.7f);
    }
}
```

Filter #455 - InterlaceBlend

Interlaces even and odd rows from different time points, creating a temporal interlace with blending.

Technique: Even/odd row temporal interlace + blend.

```
// Case 455 in unifiedFilterKernel:
processInterlaceBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processInterlaceBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev && y % 2 == 0) {
        for (int j = 0; j < 3; ++j) {
```

```

        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + prev[idx + j] * 0.5f);
    }
}

```

Filter #456 - InvertStrobe

Strobes with color inversion-alternates between normal and inverted colors rapidly.

Technique: Alternating color inversion strobe.

```

// Case 456 in unifiedFilterKernel:
processInvertStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processInvertStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int strobe = params.frame_count % 12;
    if (strobe < 4) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(255 - data[idx + j]);
        }
    }
}

```

Filter #457 - KaleidoBlend

Simplified kaleidoscope with frame blending, creating symmetric patterns with temporal depth.

Technique: Simplified kaleidoscope + blend.

```

// Case 457 in unifiedFilterKernel:
processKaleidoBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processKaleidoBlend(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2;
    int cy = height / 2;
    int dx = x - cx;
    int dy = y - cy;
    float angle = atan2f((float)dy, (float)dx);
    float sector = fmodf(angle + 3.14159f, 3.14159f / 3.0f);
    float factor = 0.5f + 0.5f * cosf(sector * 6.0f + params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
    }
}

```

Filter #458 - LightStrobe

Bright white strobe flash effect-periodically flashes the entire frame to white.

Technique: Periodic white flash overlay.

```
// Case 458 in unifiedFilterKernel:
processLightStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processLightStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int strobe = params.frame_count % 8;
    if (strobe < 2) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * 1.5f);
        }
    }
}
```

Filter #459 - LineGlitchX

Extended line glitch with more aggressive and frequent horizontal line displacement.

Technique: Intensified horizontal line glitch.

```
// Case 459 in unifiedFilterKernel:
processLineGlitchX(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processLineGlitchX(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(0, y, params.seed + params.frame_count);
    if (r < 0.05f) {
        int shift = (int)(r * 100) - 50;
        int src_x = (x + shift + width) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}
```

Filter #460 - MosaicBlend

Mosaic tile pattern blended with frame history, creating a tiled temporal composite.

Technique: Block mosaic + temporal blend.

```
// Case 460 in unifiedFilterKernel:
processMosaicBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processMosaicBlend(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int block = 4 + (params.frame_count / 10) % 4;
    int bx = (x / block) * block;
    int by = (y / block) * block;
    int bidx = by * step + bx * 4;
```

```

    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.6f + data[bidx + j] * 0.4f);
    }
}

```

Filter #461 - NegatePulse

Pulsing negation-the image smoothly oscillates between normal and inverted using sine modulation.

Technique: Sine-modulated color inversion.

```

// Case 461 in unifiedFilterKernel:
processNegatePulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processNegatePulse(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int pulse = (params.frame_count / 5) % 2;
    if (pulse == 1) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(255 - data[idx + j]);
        }
    }
}

```

Filter #462 - OffsetGhost

Ghost overlay from history frames at an animated spatial offset, creating moving transparent echoes.

Technique: Spatially-offset ghost overlay.

```

// Case 462 in unifiedFilterKernel:
processOffsetGhost(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processOffsetGhost(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        int offset = params.frame_count % 8;
        int src_x = (x + offset) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.7f + prev[src_idx + j] * 0.3f);
        }
    }
}

```

Filter #463 - PixelateWave

Pixelation with a wave-shaped boundary-the block size undulates across the frame.

Technique: Wave-modulated pixelation.

```
// Case 463 in unifiedFilterKernel:
processPixelateWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPixelateWave(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf((y + params.frame_count * 2) * 0.05f) * 4.0f;
    int px = ((int)(x + wave) % width + width) % width;
    int pidx = y * step + px * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[pidx + j] * 0.5f);
    }
}
```

Filter #464 - QuantizeBlend

Color quantization (reduced palette) blended with frame history.

Technique: Color quantization + temporal blend.

```
// Case 464 in unifiedFilterKernel:
processQuantizeBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processQuantizeBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int levels = 4 + (params.frame_count % 5);
    for (int j = 0; j < 3; ++j) {
        int val = (data[idx + j] * levels) / 256;
        data[idx + j] = (unsigned char)((val * 256) / levels);
    }
}
```

Filter #465 - RandomLines

Draws lines at random positions and orientations across the frame.

Technique: Random line position rendering.

```
// Case 465 in unifiedFilterKernel:
processRandomLines(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processRandomLines(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    if (r < 0.1f) {
        int line = y % (4 + (params.frame_count % 3));
        if (line == 0) {
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = (unsigned char)(data[idx + j] * 0.5f);
            }
        }
    }
}
```



```
}
}
```

Filter #466 - RippleDisplace

Water-ripple displacement-pixels are pushed outward from a center point in concentric rings.

Technique: Radial ripple coordinate displacement.

```
// Case 466 in unifiedFilterKernel:
processRippleDisplace(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRippleDisplace(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float ripple = sinf((y + params.frame_count * 3) * 0.04f) * 6.0f;
    int src_x = ((int)(x + ripple) % width + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.6f + data[src_idx + j] * 0.4f);
    }
}
```

Filter #467 - RotateShift

Rotates the image with an animated shift, creating a spinning/rotating effect.

Technique: Animated rotation + shift.

```
// Case 467 in unifiedFilterKernel:
processRotateShift(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRotateShift(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = (params.frame_count / 3) % width;
    int src_x = (x + shift) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[src_idx + j] * 0.5f);
    }
}
```

Filter #468 - SaturationGlow

Boosts color saturation and adds a glowing bloom around saturated areas.

Technique: Saturation boost + bloom.

```
// Case 468 in unifiedFilterKernel:
processSaturationGlow(x, y, data, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processSaturationGlow(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int max_c = max(data[idx], max(data[idx + 1], data[idx + 2]));
    float glow = (params.frame_count % 100) / 100.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + max_c * 0.3f * glow);
    }
}

```

Filter #469 - ScaleToCenter

Scales pixel positions toward the center, creating a zoom-in/magnification effect.

Technique: Center-point coordinate scaling.

```

// Case 469 in unifiedFilterKernel:
processScaleToCenter(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processScaleToCenter(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f;
    float cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float scale = 0.5f + 0.5f * sinf(dist * 0.02f + params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * scale);
    }
}

```

Filter #470 - ShadowMirror

Creates a shadow/darkened mirrored copy of the image, like a reflection on a dark surface.

Technique: Darkened mirror overlay.

```

// Case 470 in unifiedFilterKernel:
processShadowMirror(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processShadowMirror(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int mirror_x = width - 1 - x;
    int mirror_idx = y * step + mirror_x * 4;
    int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    if (avg < 100) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[mirror_idx + j] * 0.5f);
        }
    }
}

```

Filter #471 - ShiftChannels

Shifts each color channel by a different frame-animated offset, creating desynchronized channel movement.

Technique: Per-channel animated offset shift.

```
// Case 471 in unifiedFilterKernel:
processShiftChannels(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processShiftChannels(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int mode = (params.frame_count / 10) % 3;
    unsigned char b = data[idx], g = data[idx + 1], r = data[idx + 2];
    if (mode == 0) {
        data[idx] = g;
        data[idx + 1] = b;
        data[idx + 2] = r;
    } else if (mode == 1) {
        data[idx] = b;
        data[idx + 1] = r;
        data[idx + 2] = g;
    }
}
```

Filter #472 - SliceGlitch

Slices the image into horizontal strips and randomly offsets some of them, creating a sliced glitch effect.

Technique: Random horizontal slice displacement.

```
// Case 472 in unifiedFilterKernel:
processSliceGlitch(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSliceGlitch(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int slice = (params.frame_count / 2) % 8;
    if (y % 8 == slice) {
        int offset = ((params.frame_count % 30) - 15);
        int src_x = (x + offset + width) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}
```

Filter #473 - SobelGlow

Sobel edge detection with a bright glow added to the detected edges, creating luminous outlines.

Technique: Sobel edges + glow bloom.

```
// Case 473 in unifiedFilterKernel:
processSobelGlow(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSobelGlow(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0 && x < width - 1 && y > 0 && y < height - 1) {
        int gx = 0, gy = 0;
        for (int j = 0; j < 3; ++j) {
            gx += abs(data[y * step + (x + 1) * 4 + j] - data[y * step + (x - 1) * 4 + j]);
            gy += abs(data[(y + 1) * step + x * 4 + j] - data[(y - 1) * step + x * 4 + j]);
        }
        int edge = (gx + gy) / 6;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + edge * 0.4f);
        }
    }
}
```

Filter #474 - SpectralShift

Shifts colors through the visible spectrum over time, creating a continuously cycling rainbow-wash effect.

Technique: Continuous spectral hue cycling.

```
// Case 474 in unifiedFilterKernel:
processSpectralShift(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSpectralShift(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float shift = (params.frame_count % 256) / 256.0f;
    float r_shift = shift;
    float g_shift = fmodf(shift + 0.33f, 1.0f);
    float b_shift = fmodf(shift + 0.66f, 1.0f);
    data[idx] = (unsigned char)(data[idx] * (0.5f + 0.5f * sinf(b_shift * 6.28f)));
    data[idx + 1] = (unsigned char)(data[idx + 1] * (0.5f + 0.5f * sinf(g_shift * 6.28f)));
    data[idx + 2] = (unsigned char)(data[idx + 2] * (0.5f + 0.5f * sinf(r_shift * 6.28f)));
}
```

Filter #475 - SpiralTrail

Motion trails arranged in a spiral pattern from center, creating a swirling trail vortex.

Technique: Spiral-coordinate temporal trailing.

```
// Case 475 in unifiedFilterKernel:
processSpiralTrail(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSpiralTrail(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
```

```

    if (prev) {
        float angle = atan2f(y - height / 2.0f, x - width / 2.0f);
        float dist = sqrtf((x - width / 2.0f) * (x - width / 2.0f) + (y - height / 2.0f) * (y - height /
↪ 2.0f));
        float spiral = sinf(angle * 4.0f + dist * 0.05f + params.frame_count * 0.1f);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + 0.5f * spiral) + prev[idx + j] *
↪ (0.5f - 0.5f * spiral));
        }
    }
}

```

Filter #476 - SquareTrails

Block-shaped motion trails where each block retains prev frame data, creating a tiled trailing effect.

Technique: Block-aligned temporal trailing.

```

// Case 476 in unifiedFilterKernel:
processSquareTrails(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processSquareTrails(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        int block = 8 + (params.frame_count / 20) % 4;
        int bx = (x / block) % 2;
        int by = (y / block) % 2;
        if ((bx + by) % 2 == 0) {
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + prev[idx + j] * 0.5f);
            }
        }
    }
}

```

Filter #477 - StrobeNegate

Strobing with alternating color negation for a harsh flashing negative effect.

Technique: Alternating strobe + negation.

```

// Case 477 in unifiedFilterKernel:
processStrobeNegate(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processStrobeNegate(int x, int y, unsigned char *data, size_t step, const FilterParams &
↪ params) {
    int idx = y * step + x * 4;
    int strobe = (params.frame_count / 8) % 2;
    if (strobe == 1) {
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(255 - data[idx + j]);
        }
    }
}

```

```
}
}
```

Filter #478 - ThermalBlend

Maps pixel brightness to a thermal (heat-map) color palette-blue for cold/dark, red/white for hot/bright-and blends with history.

Technique: False-color thermal mapping + blend.

```
// Case 478 in unifiedFilterKernel:
processThermalBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processThermalBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float heat = (avg / 255.0f);
    data[idx] = (unsigned char)fminf(255.0f, data[idx] * (1.0f + heat * 0.3f));
    data[idx + 1] = (unsigned char)(data[idx + 1] * (1.0f - heat * 0.2f));
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * (1.0f + heat * 0.1f));
}
```

Filter #479 - TintShift

Shifts the overall color tint of the image over time, cycling through different color washes.

Technique: Animated color tint cycling.

```
// Case 479 in unifiedFilterKernel:
processTintShift(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processTintShift(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int tint = (params.frame_count / 15) % 4;
    if (tint == 0)
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * 1.2f);
    else if (tint == 1)
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * 1.2f);
    else if (tint == 2)
        data[idx] = (unsigned char)fminf(255.0f, data[idx] * 1.2f);
}
```

Filter #480 - TrailEcho

Echoing motion trails with each echo layer at decreasing opacity, like multiple afterimages fading away.

Technique: Multi-layer diminishing-opacity trailing.

```
// Case 480 in unifiedFilterKernel:
processTrailEcho(x, y, data, allFrames, step, params);
```

```

// Referenced __device__ function(s):
__device__ void processTrailEcho(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for (int i = 1; i < params.numFrames && i < 4; ++i) {
        unsigned char *frame = allFrames[i];
        if (frame) {
            float blend = 1.0f / (i + 1.0f);
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - blend * 0.5f) + frame[idx + j] *
    ↪ blend * 0.5f);
            }
        }
    }
}

```

Filter #481 - TransitionBlend

Generic smooth blend transition between current and history frames with animated alpha.

Technique: Animated alpha transition blend.

```

// Case 481 in unifiedFilterKernel:
processTransitionBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processTransitionBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *next = allFrames[0];
    if (next) {
        float transition = ((params.frame_count % 100) / 100.0f);
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - transition) + next[idx + j] *
    ↪ transition);
        }
    }
}

```

Filter #482 - TwistWarp

Rotational twist warping around center-combines rotation and radial distortion for a vortex-warp effect.

Technique: Combined rotational twist + radial warp.

```

// Case 482 in unifiedFilterKernel:
processTwistWarp(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTwistWarp(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f;
    float cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
}

```

```

float twist = dist * 0.01f * sinf(params.frame_count * 0.05f);
int src_x = (int)(cx + (x - cx) * cosf(twist) - (y - cy) * sinf(twist));
int src_y = (int)(cy + (x - cx) * sinf(twist) + (y - cy) * cosf(twist));
if (src_x >= 0 && src_x < width && src_y >= 0 && src_y < height) {
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}
}

```

Filter #483 - VerticalShift

Shifts the image vertically with animated offset.

Technique: Animated vertical position shift.

```

// Case 483 in unifiedFilterKernel:
processVerticalShift(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processVerticalShift(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = (params.frame_count / 2) % height;
    int src_y = (y + offset) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.6f + data[src_idx + j] * 0.4f);
    }
}

```

Filter #484 - VortexBlend

Spinning vortex distortion blended with frame history for temporal depth.

Technique: Vortex warp + temporal blend.

```

// Case 484 in unifiedFilterKernel:
processVortexBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processVortexBlend(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f;
    float cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float max_dist = sqrtf(cx * cx + cy * cy);
    float intensity = 1.0f - (dist / max_dist);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + 0.5f * intensity));
    }
}

```


Filter #485 - WeavePattern

Creates an interlocking woven/weave pattern overlay from alternating horizontal and vertical strips.

Technique: Orthogonal strip interleaving pattern.

```
// Case 485 in unifiedFilterKernel:
processWeavePattern(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processWeavePattern(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int weave_x = (x + params.frame_count / 2) % 32;
    int weave_y = (y + params.frame_count / 3) % 32;
    float pattern = sinf(weave_x * 0.2f) * cosf(weave_y * 0.2f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.6f + 0.4f * (pattern + 1.0f) / 2.0f));
    }
}
```

Filter #486 - WhiteBurst

Expanding white burst from center, like a flash-bang transitioning from a central bright point outward.

Technique: Expanding radial white transition.

```
// Case 486 in unifiedFilterKernel:
processWhiteBurst(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processWhiteBurst(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f;
    float cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float max_dist = sqrtf(cx * cx + cy * cy);
    float burst = sinf((dist / max_dist - params.frame_count * 0.05f) * 6.28f);
    float white = fmaxf(0.0f, burst);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + white * 100);
    }
}
```

Filter #487 - WiggleDisplace

Jittery per-pixel displacement creating a nervous, wiggling distortion.

Technique: Random coordinate jitter per pixel.

```
// Case 487 in unifiedFilterKernel:
processWiggleDisplace(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processWiggleDisplace(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float wiggle_x = sinf(y * 0.05f + params.frame_count * 0.03f) * 5.0f;
    float wiggle_y = cosf(x * 0.05f + params.frame_count * 0.02f) * 5.0f;
    int src_x = ((int)(x + wiggle_x) % width + width) % width;
    int src_y = ((int)(y + wiggle_y) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = data[src_idx + j];
    }
}

```

Filter #488 - XorPulseX

Pulsing XOR with animated intensity-the XOR strength oscillates over time.

Technique: Sine-animated XOR intensity.

```

// Case 488 in unifiedFilterKernel:
processXorPulseX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processXorPulseX(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.05f);
    unsigned char xor_val = (unsigned char)(params.frame_count * pulse);
    for (int j = 0; j < 3; ++j) {
        unsigned char xored = data[idx + j] ^ xor_val;
        data[idx + j] = (unsigned char)(xored * 0.7f + data[idx + j] * 0.3f);
    }
}

```

Filter #489 - YellowShift

Shifts the image's color balance toward yellow by boosting red and green while reducing blue.

Technique: Yellow color bias shift.

```

// Case 489 in unifiedFilterKernel:
processYellowShift(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processYellowShift(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int shift = (params.frame_count / 20) % 2;
    if (shift == 0) {
        data[idx] = (unsigned char)(data[idx] * 0.5f);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * 1.2f);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * 1.2f);
    }
}

```

Filter #490 - ZigzagGlitch

Zigzag-shaped row displacement-every other group of rows shifts in alternate directions, creating a zigzag tear pattern.

Technique: Alternating-direction row group displacement.

```
// Case 490 in unifiedFilterKernel:
processZigzagGlitch(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processZigzagGlitch(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int zigzag = (params.frame_count / 3) % 10;
    if (x % 10 == zigzag) {
        int offset = ((params.frame_count % 20) - 10);
        int src_x = (x + offset + width) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j) {
            data[idx + j] = data[src_idx + j];
        }
    }
}
```

Filter #491 - AlphaModulate

Modulates alpha blending intensity per pixel based on position, creating spatially-varying blend strength.

Technique: Position-dependent alpha modulation.

```
// Case 491 in unifiedFilterKernel:
processAlphaModulate(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processAlphaModulate(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float modulation = 0.5f + 0.5f * sinf((x + y) * 0.01f + params.frame_count * 0.04f);
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * modulation);
    }
}
```

Filter #492 - BlockSwap

Swaps rectangular blocks between the current frame and history frames, creating a tiled time-shuffle.

Technique: Block-level frame history swapping.

```
// Case 492 in unifiedFilterKernel:
processBlockSwap(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlockSwap(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
```

```

int block = 16;
int bx = x / block;
int swap_px = ((bx + (params.frame_count / 5)) % (width / block)) * block + (x % block);
int swap_py = y;
if (swap_px >= 0 && swap_px < width) {
    int swap_idx = swap_py * step + swap_px * 4;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[swap_idx + j] * 0.5f);
    }
}
}

```

Filter #493 - ColorResonance

Amplifies specific color frequencies/harmonics, creating ringing/oscillating color emphasis.

Technique: Harmonic color amplification.

```

// Case 493 in unifiedFilterKernel:
processColorResonance(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorResonance(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float freq = 0.03f + (params.frame_count % 100) * 0.001f;
    float res = sinf(x * freq) + cosf(y * freq);
    data[idx] = (unsigned char)fminf(255.0f, fmaxf(0.0f, data[idx] + res * 30));
    data[idx + 1] = (unsigned char)fminf(255.0f, fmaxf(0.0f, data[idx + 1] + res * 20));
    data[idx + 2] = (unsigned char)fminf(255.0f, fmaxf(0.0f, data[idx + 2] + res * 40));
}

```

Filter #494 - DepthGlitch

Glitch artifacts that simulate depth-far areas glitch more than near areas, creating a 3D glitch illusion.

Technique: Brightness-as-depth glitch modulation.

```

// Case 494 in unifiedFilterKernel:
processDepthGlitch(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDepthGlitch(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float glitch_factor = (avg > 128) ? 0.8f : 1.2f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * glitch_factor);
    }
}

```

Filter #495 - EchoShift

Echo trails with a spatial shift-each echo layer is offset both in time and space.

Technique: Spatially-shifting temporal echo.

```
// Case 495 in unifiedFilterKernel:
processEchoShift(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processEchoShift(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for (int i = 0; i < min(3, params.numFrames); ++i) {
        unsigned char *frame = allFrames[i];
        if (frame) {
            float echo = 0.3f / (i + 1.0f);
            for (int j = 0; j < 3; ++j) {
                data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - echo) + frame[idx + j] * echo);
            }
        }
    }
}
```

Filter #496 - FractalNoise

Generates fractal noise (layered Perlin/simplex-style noise) and blends it with the image.

Technique: Multi-octave fractal noise blending.

```
// Case 496 in unifiedFilterKernel:
processFractalNoise(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFractalNoise(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float noise = 0.0f;
    float amplitude = 1.0f;
    float frequency = 0.01f;
    for (int i = 0; i < 3; ++i) {
        noise += amplitude * sinf(x * frequency + params.frame_count * 0.02f);
        noise += amplitude * cosf(y * frequency + params.frame_count * 0.015f);
        amplitude *= 0.5f;
        frequency *= 2.0f;
    }
    noise = (noise + 3.0f) / 6.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + 0.5f * noise));
    }
}
```

Filter #497 - GradientRotate

Rotates a color gradient overlay over time, creating a spinning color wheel effect.

Technique: Animated rotating gradient.

```
// Case 497 in unifiedFilterKernel:
processGradientRotate(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGradientRotate(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f;
    float cy = height / 2.0f;
    float angle = atan2f(y - cy, x - cx) + params.frame_count * 0.03f;
    float gradient = (sinf(angle) + 1.0f) / 2.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + 0.5f * gradient));
    }
}
```

Filter #498 - HarmonicShift

Shifts colors by harmonic intervals, creating musically-inspired color progressions.

Technique: Harmonic-interval color shifts.

```
// Case 498 in unifiedFilterKernel:
processHarmonicShift(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processHarmonicShift(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float harmonic = sinf(params.frame_count * 0.05f);
    float harmonic2 = cosf(params.frame_count * 0.03f);
    data[idx] = (unsigned char)fminf(255.0f, data[idx] * (1.0f + harmonic * 0.2f));
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * (1.0f + harmonic2 * 0.2f));
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * (1.0f - harmonic * 0.1f));
}
```

Filter #499 - AcidWarp

Chaotic, multi-layered sine/cosine warping that creates organic, acid-trip-like flowing distortion.

Technique: Multi-frequency sine/cosine coordinate warping.

```
// Case 499 in unifiedFilterKernel:
processAcidWarp(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processAcidWarp(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float warp = sinf(dist * 0.03f - params.frame_count * 0.1f) * 10.0f;
    int src_x = (int)(x + warp * dx / (dist + 1.0f)) % width;
    int src_y = (int)(y + warp * dy / (dist + 1.0f)) % height;
    if (src_x < 0)
        src_x += width;
    if (src_y < 0)
```

```

        src_y += height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Visual FX A (500-599)

Filter #500 - BlendDiagonal

Animates brightness along diagonal lines using a triangular wave pattern. Diagonal bands move across the frame creating a sweeping light/dark stripe effect.

Technique: Animated diagonal triangular-wave brightness modulation.

```

// Case 500 in unifiedFilterKernel:
processBlendDiagonal(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlendDiagonal(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int diag = (x + y + params.frame_count) % 32;
    float blend = (diag < 16) ? diag / 16.0f : (32 - diag) / 16.0f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + 0.5f * blend));
}

```

Filter #501 - ChromaFlash

Flashes one color channel at a time in rotation (R->G->B), boosting it by 50%-creates a cycling single-channel color flash.

Technique: Rotating single-channel brightness boost.

```

// Case 501 in unifiedFilterKernel:
processChromaFlash(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processChromaFlash(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int flash = (params.frame_count / 4) % 3;
    data[idx + flash] = (unsigned char)fminf(255.0f, data[idx + flash] * 1.5f);
}

```

Filter #502 - CircleWave

Concentric circular waves radiate from center, modulating brightness with a sine function of distance from center, animated outward over time.

Technique: Radial sine-wave brightness modulation from center.

```

// Case 502 in unifiedFilterKernel:
processCircleWave(x, y, data, width, height, step, params);

```

```
// Referenced __device__ function(s):
__device__ void processCircleWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float wave = sinf(dist * 0.1f - params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * (0.6f + 0.4f * wave));
}
```

Filter #503 - ColorCrush

Quantizes/crashes color values to a reduced palette that changes over time-animated posterization with a shifting crush factor.

Technique: Animated color quantization (posterize).

```
// Case 503 in unifiedFilterKernel:
processColorCrush(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorCrush(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int crush = 3 + (params.frame_count / 20) % 5;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)((data[idx + j] / crush) * crush);
}
```

Filter #504 - CrosshatchBlend

Overlays an animated crosshatch pattern-thin horizontal and vertical lines darken the image in a woven grid that moves over time.

Technique: Animated crosshatch darkening pattern.

```
// Case 504 in unifiedFilterKernel:
processCrosshatchBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCrosshatchBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int hatch = ((x + params.frame_count) % 8 < 2) || ((y + params.frame_count) % 8 < 2);
    if (hatch)
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.6f);
}
```

Filter #505 - CyberGlitch

Sparse random horizontal pixel displacement-2% of pixels get randomly shifted sideways by up to 250 pixels, creating sudden horizontal tears.

Technique: Sparse random horizontal pixel displacement.

```
// Case 505 in unifiedFilterKernel:
processCyberGlitch(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processCyberGlitch(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    if (r < 0.02f) {
        int offset = (int)(r * 500) - 250;
        int src_x = (x + offset + width) % width;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[y * step + src_x * 4 + j];
    }
}
```

Filter #506 - DarkPulse

Pulsing darkness on shadow areas-pixels below mid-brightness darken and brighten with a sine oscillation, while bright areas stay untouched.

Technique: Sine-pulsed shadow darkening.

```
// Case 506 in unifiedFilterKernel:
processDarkPulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDarkPulse(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pulse = 0.5f + 0.3f * sinf(params.frame_count * 0.08f);
    int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    if (avg < 128)
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * pulse);
}
```

Filter #507 - DiamondPatternX

Overlays an animated diamond (rhombus) pattern-Manhattan-distance-based diamonds tile the frame with brightness modulation.

Technique: Animated diamond/rhombus tiling brightness pattern.

```
// Case 507 in unifiedFilterKernel:
processDiamondPatternX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDiamondPatternX(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int dx = abs((x + params.frame_count) % 32 - 16);
    int dy = abs((y + params.frame_count) % 32 - 16);
    float diamond = 1.0f - (dx + dy) / 32.0f;
    for (int j = 0; j < 3; ++j)
```

```

    data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + 0.5f * diamond));
}

```

Filter #508 - DigitalRain

Matrix-style digital rain-vertical streaks of bright green fall down the frame, tinting nearby pixels green while dimming red and blue.

Technique: Vertical green streak overlay (Matrix-style).

```

// Case 508 in unifiedFilterKernel:
processDigitalRain(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processDigitalRain(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int rain = (y + params.frame_count * 3) % height;
    if (rain < 20) {
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * 1.5f);
        data[idx] = (unsigned char)(data[idx] * 0.7f);
        data[idx + 2] = (unsigned char)(data[idx + 2] * 0.7f);
    }
}

```

Filter #509 - DisplaceX

Sine-wave horizontal displacement-each row is shifted left/right by a sine function of its Y position, creating a gentle animated wobble.

Technique: Sine-modulated horizontal row displacement.

```

// Case 509 in unifiedFilterKernel:
processDisplaceX(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processDisplaceX(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int disp = (int)(sinf(y * 0.05f + params.frame_count * 0.05f) * 8);
    int src_x = (x + disp + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[src_idx + j] * 0.5f);
}

```

Filter #510 - DriftBlend

Blends current frame with a drifting (horizontally offset) version of the previous frame, creating a slow sideways ghosting trail.

Technique: Horizontally-drifting temporal blend.

```

// Case 510 in unifiedFilterKernel:
processDriftBlend(x, y, data, allFrames, width, step, params);

```

```

// Referenced __device__ function(s):
__device__ void processDriftBlend(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↳ width, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        int drift = (params.frame_count / 2) % 10;
        int src_x = (x + drift) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.6f + prev[src_idx + j] * 0.4f);
    }
}

```

Filter #511 - EdgePulse

Edge-detected pixels pulse brighter with a sine animation-detected edges glow and fade rhythmically while flat areas remain unchanged.

Technique: Pulsing edge brightness enhancement.

```

// Case 511 in unifiedFilterKernel:
processEdgePulse(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processEdgePulse(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↳ const FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0 && x < width - 1 && y > 0 && y < height - 1) {
        int edge = 0;
        for (int j = 0; j < 3; ++j)
            edge += abs(data[y * step + (x + 1) * 4 + j] - data[y * step + (x - 1) * 4 + j]);
        float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.1f);
        if (edge > 50)
            for (int j = 0; j < 3; ++j)
                data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * (1.0f + pulse * 0.5f));
    }
}

```

Filter #512 - FlameEffect

Simulates flames rising from the bottom-red/orange warming increases toward the bottom of the frame with animated sine flickering, while blue cools at the top.

Technique: Bottom-up warm color gradient with sine flicker.

```

// Case 512 in unifiedFilterKernel:
processFlameEffect(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processFlameEffect(int x, int y, unsigned char *data, int height, size_t step, const
    ↳ FilterParams &params) {
    int idx = y * step + x * 4;
    float flame = sinf(x * 0.1f + params.frame_count * 0.15f) * 0.5f + 0.5f;
    float yf = 1.0f - (float)y / height;
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * (1.0f + flame * yf * 0.5f));
}

```

```

    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * (1.0f + flame * yf * 0.3f));
    data[idx] = (unsigned char)(data[idx] * (1.0f - flame * yf * 0.2f));
}

```

Filter #513 - FlickerShift

Sparse random pixel inversion-every 5th pixel (animated pattern) gets color-inverted, creating an uneven rapid flicker.

Technique: Patterned sparse pixel inversion.

```

// Case 513 in unifiedFilterKernel:
processFlickerShift(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFlickerShift(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int flicker = (params.frame_count + x + y) % 5;
    if (flicker == 0)
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(255 - data[idx + j]);
}

```

Filter #514 - GhostLayer

Layers up to 4 history frames as transparent ghost overlays with decreasing opacity (15%, 7.5%, 5%, 3.75%).

Technique: Multi-frame diminishing ghost layering.

```

// Case 514 in unifiedFilterKernel:
processGhostLayer(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processGhostLayer(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for (int i = 0; i < min(4, params.numFrames); ++i) {
        unsigned char *frame = allFrames[i];
        if (frame) {
            float ghost = 0.15f / (i + 1.0f);
            for (int j = 0; j < 3; ++j)
                data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - ghost) + frame[idx + j] * ghost)
        }
    }
}

```

Filter #515 - GlitchBlockX

Block-based glitch-16x16 pixel blocks randomly get their pixels shifted horizontally, creating periodic block-level displacement.

Technique: Random 16x16 block horizontal displacement.

```
// Case 515 in unifiedFilterKernel:
processGlitchBlockX(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGlitchBlockX(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int bx = x / 16, by = y / 16;
    float r = gpu_rand(bx, by, params.seed + params.frame_count / 5);
    if (r < 0.1f) {
        int offset = (int)(r * 160) - 80;
        int src_x = (x + offset + width) % width;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[y * step + src_x * 4 + j];
    }
}
```

Filter #516 - GlowPulse

Bright areas (>150 brightness) pulse with a sine glow-already-bright pixels get brighter and dimmer rhythmically, creating a pulsing bloom effect.

Technique: Sine-pulsed brightness boost on highlights.

```
// Case 516 in unifiedFilterKernel:
processGlowPulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGlowPulse(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.07f);
    if (avg > 150)
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * (1.0f + pulse * 0.3f));
}
```

Filter #517 - GridDistort

Checkerboard grid distortion-alternating 8x8 grid cells shift pixels left or right based on an animated offset.

Technique: Checkerboard-alternating grid pixel shifts.

```
// Case 517 in unifiedFilterKernel:
processGridDistort(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGridDistort(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int gx = (x / 8) % 2, gy = (y / 8) % 2;
    int offset = ((gx + gy) % 2) * ((params.frame_count % 10) - 5);
    int src_x = (x + offset + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
```

```

        data[idx + j] = data[src_idx + j];
    }

```

Filter #518 - HexPattern

Overlays a hexagonal tile pattern that dims alternating hex regions, creating an animated honeycomb brightness pattern.

Technique: Animated hexagonal tile brightness modulation.

```

// Case 518 in unifiedFilterKernel:
processHexPattern(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processHexPattern(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int hx = (x + (y / 2) * 3 + params.frame_count) % 12;
    int hy = y % 6;
    float hex = (hx < 6 && hy < 3) ? 1.0f : 0.7f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * hex);
}

```

Filter #519 - HueRotate

True hue rotation-rotates the entire color wheel by an angle that continuously increases (full 360deg cycle), using a proper color matrix transformation.

Technique: Color-matrix hue rotation.

```

// Case 519 in unifiedFilterKernel:
processHueRotate(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processHueRotate(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float angle = (params.frame_count % 360) * 0.0174533f;
    float cs = cosf(angle), sn = sinf(angle);
    float r = data[idx + 2], g = data[idx + 1], b = data[idx];
    data[idx + 2] = (unsigned char)fminf(255.0f, fmaxf(0.0f, r * (0.213f + cs * 0.787f - sn * 0.213f) +
    ↪ g * (0.715f - cs * 0.715f - sn * 0.715f) + b * (0.072f - cs * 0.072f + sn * 0.928f)));
    data[idx + 1] = (unsigned char)fminf(255.0f, fmaxf(0.0f, r * (0.213f - cs * 0.213f + sn * 0.143f) +
    ↪ g * (0.715f + cs * 0.285f + sn * 0.140f) + b * (0.072f - cs * 0.072f - sn * 0.283f)));
    data[idx] = (unsigned char)fminf(255.0f, fmaxf(0.0f, r * (0.213f - cs * 0.213f - sn * 0.787f) + g *
    ↪ (0.715f - cs * 0.715f + sn * 0.715f) + b * (0.072f + cs * 0.928f + sn * 0.072f)));
}

```

Filter #520 - InterweaveX

Interweaves current and previous frames in alternating 2-pixel-wide vertical stripes that shift over time.

Technique: Alternating vertical stripe temporal interweave.

```
// Case 520 in unifiedFilterKernel:
processInterweaveX(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processInterweaveX(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *prev = allFrames[0];
    if (prev) {
        int weave = (x + params.frame_count) % 4;
        if (weave < 2)
            for (int j = 0; j < 3; ++j)
                data[idx + j] = prev[idx + j];
    }
}
```

Filter #521 - JitterBlend

Random per-pixel horizontal jitter (+-3 pixels) blended with the original-creates a subtle nervous/shaky displacement.

Technique: Random horizontal jitter with blend-back.

```
// Case 521 in unifiedFilterKernel:
processJitterBlend(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processJitterBlend(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    int jitter = (int)(r * 6) - 3;
    int src_x = (x + jitter + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * 0.7f + data[src_idx + j] * 0.3f);
}
```

Filter #522 - KaleidoScope4

4-way kaleidoscope mirror-folds the image into quadrants by mirroring X and Y around center, blended with the original for a semi-transparent kaleidoscope.

Technique: Quadrant-fold mirror blend.

```
// Case 522 in unifiedFilterKernel:
processKaleidoScope4(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processKaleidoScope4(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int mx = (x < width / 2) ? x : width - 1 - x;
    int my = (y < height / 2) ? y : height - 1 - y;
    int src_idx = my * step + mx * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[src_idx + j] * 0.5f);
}
```

```
}  
}
```

Filter #523 - LaserScan

A bright red laser scan line sweeps vertically down the frame, tinting a 3-pixel-wide band to bright red as it passes.

Technique: Vertical red scan line sweep.

```
// Case 523 in unifiedFilterKernel:  
processLaserScan(x, y, data, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processLaserScan(int x, int y, unsigned char *data, int width, int height, size_t step,  
    ↪ const FilterParams &params) {  
    int idx = y * step + x * 4;  
    int scan = (params.frame_count * 4) % height;  
    if (abs(y - scan) < 3) {  
        data[idx + 2] = 255;  
        data[idx + 1] = (unsigned char)(data[idx + 1] * 0.5f);  
        data[idx] = (unsigned char)(data[idx] * 0.5f);  
    }  
}
```

Filter #524 - LightLeak

Simulates photographic light leaks-warm orange/yellow glow that drifts across the frame as a sine-modulated lens-flare-like overlay.

Technique: Animated warm light leak overlay.

```
// Case 524 in unifiedFilterKernel:  
processLightLeak(x, y, data, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void processLightLeak(int x, int y, unsigned char *data, int width, int height, size_t step,  
    ↪ const FilterParams &params) {  
    int idx = y * step + x * 4;  
    float lx = (float)x / width, ly = (float)y / height;  
    float leak = sinf(lx * 3.14159f + params.frame_count * 0.05f) * sinf(ly * 3.14159f);  
    leak = fmaxf(0.0f, leak);  
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + leak * 80);  
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + leak * 60);  
}
```

Filter #525 - MeltDown

Vertical melt distortion-pixels are pulled from slightly above or below based on a sine wave, creating a dripping/melting effect.

Technique: Sine-wave vertical pixel displacement (melt).

```
// Case 525 in unifiedFilterKernel:  
processMeltDown(x, y, data, height, step, params);
```



```
// Referenced __device__ function(s):
__device__ void processMeltDown(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int melt = (int)(sinf(x * 0.1f + params.frame_count * 0.05f) * 5);
    int src_y = min(max(y + melt, 0), height - 1);
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * 0.4f + data[src_idx + j] * 0.6f);
}
```

Filter #526 - MirrorDiag

Diagonal mirror blend-alternating pixels (checkerboard) are blended with their diagonally-opposite counterpart, creating an animated diagonal symmetry.

Technique: Diagonal-opposite pixel blending.

```
// Case 526 in unifiedFilterKernel:
processMirrorDiag(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processMirrorDiag(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    if ((x + y) % 2 == params.frame_count % 2) {
        int mx = width - 1 - x, my = height - 1 - y;
        mx = min(max(mx, 0), width - 1);
        my = min(max(my, 0), height - 1);
        int src_idx = my * step + mx * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + data[src_idx + j] * 0.5f);
    }
}
```

Filter #527 - NeonGlow

Strong edges glow in neon colors-edge detection highlights are rendered as single bright channel values (R, G, or B rotating), creating neon-sign-style outlines.

Technique: Edge detection + rotating single-channel neon glow.

```
// Case 527 in unifiedFilterKernel:
processNeonGlow(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processNeonGlow(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0 && x < width - 1 && y > 0 && y < height - 1) {
        int edge = 0;
        for (int j = 0; j < 3; ++j) {
            edge += abs(data[(y + 1) * step + x * 4 + j] - data[(y - 1) * step + x * 4 + j]);
            edge += abs(data[y * step + (x + 1) * 4 + j] - data[y * step + (x - 1) * 4 + j]);
        }
        if (edge > 100) {
            int ch = params.frame_count % 3;

```

```

        data[idx + ch] = 255;
    }
}

```

Filter #528 - NoiseBlendX

Subtle per-pixel noise modulation-each pixel's brightness is randomly varied by up to +-30%, creating a gentle animated grain.

Technique: Per-pixel random brightness noise.

```

// Case 528 in unifiedFilterKernel:
processNoiseBlendX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processNoiseBlendX(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float noise = gpu_rand(x, y, params.seed + params.frame_count) * 0.3f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * (0.85f + noise));
}

```

Filter #529 - PixelDrift

Block-aligned horizontal drift-pixels shift sideways in 4-pixel groups with an animated ping-pong offset.

Technique: 4-pixel block horizontal drift.

```

// Case 529 in unifiedFilterKernel:
processPixelDrift(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processPixelDrift(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int drift = ((x / 4 + params.frame_count) % 8) - 4;
    int src_x = (x + drift + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #530 - PlasmaWave

Classic plasma effect-overlapping sine waves on X and Y axes create a flowing plasma pattern that modulates RGB channels differently.

Technique: Dual-axis sine plasma color modulation.

```

// Case 530 in unifiedFilterKernel:
processPlasmaWave(x, y, data, step, params);

```

```
// Referenced __device__ function(s):
__device__ void processPlasmaWave(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float plasma = sinf(x * 0.03f + params.frame_count * 0.05f) + sinf(y * 0.03f + params.frame_count *
    ↪ 0.04f);
    plasma = (plasma + 2.0f) / 4.0f;
    data[idx] = (unsigned char)(data[idx] * (0.5f + plasma * 0.5f));
    data[idx + 1] = (unsigned char)(data[idx + 1] * (1.0f - plasma * 0.3f));
    data[idx + 2] = (unsigned char)(data[idx + 2] * (0.5f + (1.0f - plasma) * 0.5f));
}

```

Filter #531 - PrismSplit

Chromatic prism split-the red channel is offset rightward and blue leftward by an animated amount, simulating light passing through a prism.

Technique: Animated opposing R/B channel offset (chromatic aberration).

```
// Case 531 in unifiedFilterKernel:
processPrismSplit(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processPrismSplit(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = 2 + (params.frame_count % 4);
    int rx = (x + offset) % width, bx = (x - offset + width) % width;
    data[idx + 2] = data[y * step + rx * 4 + 2];
    data[idx] = data[y * step + bx * 4];
}

```

Filter #532 - PulseRadial

Radial brightness pulse-concentric rings of brightness modulation pulse outward from center, like a radar ping expanding.

Technique: Outward-pulsing radial sine brightness rings.

```
// Case 532 in unifiedFilterKernel:
processPulseRadial(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPulseRadial(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float pulse = sinf(dist * 0.05f - params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * (0.7f + 0.3f * pulse));
}

```

Filter #533 - RainbowStrobe

Strobes through rainbow colors-cycles through boosting R, G, then B channels in a 6-phase sequence, creating a fast rainbow flash.

Technique: 6-phase RGB channel boost cycling.

```
// Case 533 in unifiedFilterKernel:
processRainbowStrobe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processRainbowStrobe(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int phase = (params.frame_count / 3) % 6;
    float boost = 1.3f;
    if (phase == 0 || phase == 5)
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * boost);
    if (phase == 1 || phase == 2)
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * boost);
    if (phase == 3 || phase == 4)
        data[idx] = (unsigned char)fminf(255.0f, data[idx] * boost);
}
```

Filter #534 - RefractionX

Refraction displacement-pixels are shifted horizontally by a sine function of (x+y), simulating light bending through an uneven medium.

Technique: Diagonal sine-refraction horizontal displacement.

```
// Case 534 in unifiedFilterKernel:
processRefractionX(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processRefractionX(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float refract = sinf((x + y) * 0.05f + params.frame_count * 0.03f) * 4.0f;
    int src_x = ((int)(x + refract) % width + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * 0.6f + data[src_idx + j] * 0.4f);
}
```

Filter #535 - ScanlineX

Animated CRT scanlines-every 4th row (animating downward) is darkened by 30%, simulating moving CRT scan lines.

Technique: Animated horizontal scanline darkening.

```
// Case 535 in unifiedFilterKernel:
processScanlineX(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processScanlineX(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
```

```

    int idx = y * step + x * 4;
    int scanline = (y + params.frame_count) % 4;
    if (scanline == 0)
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.7f);
}

```

Filter #536 - ShatterEffect

Block-shatter displacement-32x32 blocks are displaced by random offsets that increase over time, simulating the image shattering into drifting pieces.

Technique: Time-expanding random block displacement.

```

// Case 536 in unifiedFilterKernel:
processShatterEffect(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processShatterEffect(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int bx = x / 32, by = y / 32;
    float r = gpu_rand(bx, by, params.seed);
    int ox = (int)((r - 0.5f) * (params.frame_count % 20));
    int oy = (int)((r - 0.5f) * (params.frame_count % 20));
    int src_x = (x + ox + width) % width, src_y = (y + oy + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #537 - StaticNoise

TV static-5% of pixels are randomly replaced with monochrome noise values, creating scattered white noise dots.

Technique: Sparse random pixel static noise.

```

// Case 537 in unifiedFilterKernel:
processStaticNoise(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processStaticNoise(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    if (r < 0.05f) {
        unsigned char noise = (unsigned char)(r * 5000) % 256;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = noise;
    }
}

```

Filter #538 - TunnelVision

Animated vignette/tunnel effect-edges of the frame darken while center stays bright, with the vignette intensity pulsing over time.

Technique: Pulsing radial vignette.

```
// Case 538 in unifiedFilterKernel:
processTunnelVision(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTunnelVision(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float max_dist = sqrtf(cx * cx + cy * cy);
    float vignette = 1.0f - (dist / max_dist) * 0.7f;
    vignette = fmaxf(0.3f, vignette + 0.2f * sinf(params.frame_count * 0.05f));
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * vignette);
}
```

Filter #539 - AberrationPulse

Pulsing chromatic aberration-red and blue channels are offset in opposite horizontal directions by an amount that oscillates with a sine wave.

Technique: Sine-pulsed opposing R/B channel offset.

```
// Case 539 in unifiedFilterKernel:
processAberrationPulse(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processAberrationPulse(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = (int)(5.0f * sinf(params.frame_count * 0.1f));
    int rx = (x + offset + width) % width;
    int bx = (x - offset + width) % width;
    data[idx] = data[y * step + bx * 4];
    data[idx + 2] = data[y * step + rx * 4 + 2];
}
```

Filter #540 - AquaWave

Aquatic wave distortion with color tinting-vertical sine waves displace pixels while boosting green/cyan and reducing blue, creating an underwater look.

Technique: Vertical wave displacement + aqua color tint.

```
// Case 540 in unifiedFilterKernel:
processAquaWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processAquaWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
```

```

    int idx = y * step + x * 4;
    float wave = sinf(x * 0.05f + params.frame_count * 0.08f) * 10.0f;
    int src_y = ((int)(y + wave) + height) % height;
    int src_idx = src_y * step + x * 4;
    data[idx] = (unsigned char)fminf(255.0f, data[src_idx] * 0.8f);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[src_idx + 1] * 1.2f);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[src_idx + 2] * 1.1f);
}

```

Filter #541 - BinaryFlash

Animated binary threshold-converts to black and white with a threshold that oscillates via sine, creating a flashing high-contrast silhouette effect.

Technique: Oscillating-threshold binary conversion.

```

// Case 541 in unifiedFilterKernel:
processBinaryFlash(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBinaryFlash(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    int threshold = 128 + (int)(64.0f * sinf(params.frame_count * 0.15f));
    unsigned char val = (gray > threshold) ? 255 : 0;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = val;
}

```

Filter #542 - BloomGlow

Bloom/glow effect-all pixels pulse brighter with a sine wave, and highlights (>200) get an extra boost, creating a pulsing cinematic bloom.

Technique: Sine-pulsed brightness + highlight bloom.

```

// Case 542 in unifiedFilterKernel:
processBloomGlow(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processBloomGlow(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float bloom = 1.0f + 0.3f * sinf(params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j) {
        float val = data[idx + j] * bloom;
        if (data[idx + j] > 200)
            val += 30;
        data[idx + j] = (unsigned char)fminf(255.0f, val);
    }
}

```

Filter #543 - CellularNoise

Cellular/Voronoi-style pattern-divides space into 16x16 cells and dims pixels based on distance from cell center, creating a cellular shadow pattern.

Technique: Distance-from-cell-center brightness modulation.

```
// Case 543 in unifiedFilterKernel:
processCellularNoise(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCellularNoise(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int cx = (x / 16) * 16 + 8, cy = (y / 16) * 16 + 8;
    float dist = sqrtf((float)((x - cx) * (x - cx) + (y - cy) * (y - cy)));
    float factor = fmaxf(0.5f, 1.0f - dist / 16.0f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
}
```

Filter #544 - ChromaShift2

Chromatic aberration variant-fixed red-right and blue-left channel offset with animated shift distance (3-7 pixels).

Technique: Animated chromatic channel separation.

```
// Case 544 in unifiedFilterKernel:
processChromaShift2(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processChromaShift2(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = 3 + (params.frame_count % 5);
    int r_idx = y * step + ((x + shift) % width) * 4;
    int b_idx = y * step + ((x - shift + width) % width) * 4;
    data[idx + 2] = data[r_idx + 2];
    data[idx] = data[b_idx];
}
```

Filter #545 - ColorBands

Horizontal animated color bands-the frame is divided into horizontal thirds (scrolling) that boost red, green, or blue respectively.

Technique: Scrolling horizontal RGB band boost.

```
// Case 545 in unifiedFilterKernel:
processColorBands(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorBands(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int band = (y + params.frame_count) % 60;
    if (band < 20)
```



```

        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * 1.5f);
    else if (band < 40)
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * 1.5f);
    else
        data[idx] = (unsigned char)fminf(255.0f, data[idx] * 1.5f);
}

```

Filter #546 - ColorVortex

Color vortex-hue shifts based on angle from center and distance, creating a spinning color wheel overlay that rotates over time.

Technique: Angle+distance based color vortex rotation.

```

// Case 546 in unifiedFilterKernel:
processColorVortex(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processColorVortex(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float angle = atan2f(dy, dx) + params.frame_count * 0.02f;
    float dist = sqrtf(dx * dx + dy * dy);
    int hue_shift = (int)(angle * 40.0f + dist * 0.1f) % 256;
    data[idx] = (data[idx] + hue_shift) % 256;
    data[idx + 1] = (data[idx + 1] + hue_shift / 2) % 256;
}

```

Filter #547 - CrystalMosaic

Crystal/mosaic pixelation-samples from the center of each block (variable size 8-15), with block size animating over time.

Technique: Animated variable-size center-sampled mosaic.

```

// Case 547 in unifiedFilterKernel:
processCrystalMosaic(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processCrystalMosaic(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size = 8 + (params.frame_count % 8);
    int bx = (x / size) * size + size / 2;
    int by = (y / size) * size + size / 2;
    bx = bx < width ? bx : width - 1;
    by = by < height ? by : height - 1;
    int src_idx = by * step + bx * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #548 - CubicDistort

Cubic barrel/pincushion distortion-applies a cubic polynomial warp from center with oscillating strength, creating breathing lens distortion.

Technique: Oscillating cubic lens distortion.

```
// Case 548 in unifiedFilterKernel:
processCubicDistort(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processCubicDistort(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float nx = (x - width / 2.0f) / (width / 2.0f);
    float ny = (y - height / 2.0f) / (height / 2.0f);
    float distort = 0.1f * sinf(params.frame_count * 0.05f);
    nx = nx + distort * nx * nx * nx;
    ny = ny + distort * ny * ny * ny;
    int src_x = (int)((nx + 1.0f) * width / 2.0f) % width;
    int src_y = (int)((ny + 1.0f) * height / 2.0f) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #549 - DepthFade

Vertical depth fade-the image fades darker toward the top and brighter toward the bottom, with the gradient position pulsing via sine.

Technique: Pulsing vertical depth-gradient brightness.

```
// Case 549 in unifiedFilterKernel:
processDepthFade(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processDepthFade(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float depth = (float)y / height;
    depth = depth + 0.2f * sinf(params.frame_count * 0.03f);
    depth = fmaxf(0.3f, fminf(1.0f, depth));
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * depth);
}
```

Filter #550 - DiscoFlash

Disco flash-adds strong color (+100) to one channel at a time cycling R->G->B in 4-frame phases, creating a bold color strobe.

Technique: 3-phase channel additive color flash.

```
// Case 550 in unifiedFilterKernel:
processDiscoFlash(x, y, data, step, params);
```

```
// Referenced __device__ function(s):
__device__ void processDiscoFlash(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int phase = params.frame_count % 12;
    if (phase < 4)
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + 100);
    else if (phase < 8)
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + 100);
    else
        data[idx] = (unsigned char)fminf(255.0f, data[idx] + 100);
}
```

Filter #551 - DitherBlend

Animated checker dither-alternating pixels in a checkerboard are either brightened (x1.2) or darkened (x0.8), with the pattern flipping each frame.

Technique: Animated checkerboard brightness dithering.

```
// Case 551 in unifiedFilterKernel:
processDitherBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDitherBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int pattern = ((x + params.frame_count) % 2) ^ ((y + params.frame_count) % 2);
    float factor = pattern ? 1.2f : 0.8f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
}
```

Filter #552 - DoubleVision

Double vision displacement-the image is blended 50/50 with a horizontally-offset copy of itself, with the offset distance oscillating.

Technique: Oscillating horizontal double-image blend.

```
// Case 552 in unifiedFilterKernel:
processDoubleVision(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processDoubleVision(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = 10 + (int)(5.0f * sinf(params.frame_count * 0.1f));
    int x2 = (x + offset) % width;
    int idx2 = y * step + x2 * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)((data[idx + j] + data[idx2 + j]) / 2);
}
```

Filter #553 - DreamHaze

Dreamy haze-pixels are blended toward a bright haze with a spatially-varying sine pattern, creating soft, foggy dream-like brightness.

Technique: Spatially-varying sine haze blend.

```
// Case 553 in unifiedFilterKernel:
processDreamHaze(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDreamHaze(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float haze = 0.7f + 0.3f * sinf(params.frame_count * 0.02f + x * 0.01f + y * 0.01f);
    for (int j = 0; j < 3; ++j) {
        float val = data[idx + j] * haze + 50 * (1.0f - haze);
        data[idx + j] = (unsigned char)fminf(255.0f, val);
    }
}
```

Filter #554 - ElectricStorm

Sparse blue-white electric sparks-2% of pixels randomly flash to bright blue-white, simulating electric discharges.

Technique: Sparse random blue-white pixel flash.

```
// Case 554 in unifiedFilterKernel:
processElectricStorm(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processElectricStorm(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    if (r < 0.02f) {
        data[idx] = 200;
        data[idx + 1] = 200;
        data[idx + 2] = 255;
    }
}
```

Filter #555 - EmbossShift

Animated emboss effect-calculates difference from a diagonally-offset neighbor (offset distance animates 1-3), creating a shifting emboss relief.

Technique: Animated-offset diagonal emboss.

```
// Case 555 in unifiedFilterKernel:
processEmbossShift(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processEmbossShift(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int shift = 1 + (params.frame_count % 3);
```

```

    if (x >= shift && y >= shift && x < width - 1 && y < height - 1) {
        for (int j = 0; j < 3; ++j) {
            int diff = data[idx + j] - data[(y - shift) * step + (x - shift) * 4 + j];
            data[idx + j] = (unsigned char)(128 + diff);
        }
    }
}

```

Filter #556 - FiberOptic

Radial fiber-optic coloring-pixels are tinted toward a single channel (R, G, or B) based on their angle from center, creating radial fiber-optic color spokes.

Technique: Angle-based radial channel boost.

```

// Case 556 in unifiedFilterKernel:
processFiberOptic(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFiberOptic(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float angle = atan2f(y - cy, x - cx);
    int fiber = (int)((angle + 3.14159f) * 20) % 3;
    data[idx + fiber] = (unsigned char)fminf(255.0f, data[idx + fiber] * 1.5f);
}

```

Filter #557 - FilmGrain

Realistic film grain-adds random brightness noise (+-15 levels) to each pixel, simulating photographic film grain.

Technique: Per-pixel additive random brightness noise.

```

// Case 557 in unifiedFilterKernel:
processFilmGrain(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFilmGrain(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float grain = (gpu_rand(x, y, params.seed + params.frame_count) - 0.5f) * 30.0f;
    for (int j = 0; j < 3; ++j) {
        float val = data[idx + j] + grain;
        data[idx + j] = (unsigned char)fmaxf(0.0f, fminf(255.0f, val));
    }
}

```

Filter #558 - FireWorks

Firework burst-renders an expanding ring of bright red/orange from a fixed explosion center that grows outward and repeats.

Technique: Expanding radial ring burst animation.

```

// Case 558 in unifiedFilterKernel:
processFireWorks(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFireWorks(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = (params.seed % (width / 2)) + width / 4;
    int cy = (params.seed % (height / 2)) + height / 4;
    float dist = sqrtf((float)((x - cx) * (x - cx) + (y - cy) * (y - cy)));
    float ring = (float)(params.frame_count % 60);
    if (fabsf(dist - ring) < 3.0f) {
        data[idx + 2] = 255;
        data[idx + 1] = (unsigned char)(200 - ring * 3);
        data[idx] = 0;
    }
}

```

Filter #559 - FluidMotion

Fluid flow distortion-pixels are displaced by sinusoidal flow in both X and Y, creating a smooth, liquid/flowing warp.

Technique: Dual-axis sinusoidal flow displacement.

```

// Case 559 in unifiedFilterKernel:
processFluidMotion(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFluidMotion(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float t = params.frame_count * 0.05f;
    float dx = sinf(y * 0.02f + t) * 5.0f;
    float dy = cosf(x * 0.02f + t) * 5.0f;
    int src_x = ((int)(x + dx) + width) % width;
    int src_y = ((int)(y + dy) + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #560 - FogRoll

Rolling fog-a horizontal fog line oscillates vertically, with pixels below it gradually blending toward grey/white, simulating rolling ground fog.

Technique: Animated horizontal fog-line gradient blend.

```

// Case 560 in unifiedFilterKernel:
processFogRoll(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processFogRoll(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float fog_line = height * (0.5f + 0.3f * sinf(params.frame_count * 0.02f));
}

```

```

float fog = 0.0f;
if (y > fog_line)
    fog = fminf(0.7f, (y - fog_line) / (height * 0.3f));
for (int j = 0; j < 3; ++j) {
    float val = data[idx + j] * (1.0f - fog) + 200 * fog;
    data[idx + j] = (unsigned char)val;
}
}

```

Filter #561 - GlassRefract

Glass refraction distortion-pixels are displaced in both X and Y by a sinexcosine pattern, simulating light bending through wavy glass.

Technique: 2D sine-cosine glass refraction displacement.

```

// Case 561 in unifiedFilterKernel:
processGlassRefract(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGlassRefract(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float refract = sinf(x * 0.1f + params.frame_count * 0.1f) * cosf(y * 0.1f) * 8.0f;
    int src_x = ((int)(x + refract) + width) % width;
    int src_y = ((int)(y + refract * 0.5f) + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #562 - GlowTrails

Highlight glow trails-only bright pixels (>150) get a sine-pulsed brightness boost, creating glowing trails from bright objects.

Technique: Conditional highlight sine-pulsed glow.

```

// Case 562 in unifiedFilterKernel:
processGlowTrails(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGlowTrails(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float glow = 1.0f + 0.5f * sinf(params.frame_count * 0.1f + x * 0.02f);
    for (int j = 0; j < 3; ++j) {
        if (data[idx + j] > 150) {
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * glow);
        }
    }
}

```

Filter #563 - GridPulse

Pulsing grid overlay-grid lines (2px wide every 32px) pulse between their original color and a mid-grey, creating a breathing grid overlay.

Technique: Sine-pulsed grid line brightness.

```
// Case 563 in unifiedFilterKernel:
processGridPulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGridPulse(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int grid_x = x % 32, grid_y = y % 32;
    float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.1f);
    if (grid_x < 2 || grid_y < 2) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * pulse + 128 * (1.0f - pulse));
    }
}
```

Filter #564 - HalftoneBlend

Halftone printing effect-converts to dot pattern where dot radius is proportional to local brightness, blended with original colors. Dot size animates.

Technique: Animated halftone dot pattern blend.

```
// Case 564 in unifiedFilterKernel:
processHalftoneBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processHalftoneBlend(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    int dot_size = 4 + (params.frame_count % 4);
    int cx = (x / dot_size) * dot_size + dot_size / 2;
    int cy = (y / dot_size) * dot_size + dot_size / 2;
    float dist = sqrtf((float)((x - cx) * (x - cx) + (y - cy) * (y - cy)));
    float threshold = (gray / 255.0f) * dot_size;
    unsigned char val = (dist < threshold) ? 255 : 0;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (data[idx + j] + val) / 2;
}
```

Filter #565 - HeatDistort

Heat shimmer distortion-horizontal displacement that increases toward the top of the frame, simulating rising heat waves.

Technique: Height-proportional horizontal sine heat shimmer.

```
// Case 565 in unifiedFilterKernel:
processHeatDistort(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
```



```

__device__ void processHeatDistort(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float heat = sinf(y * 0.1f + params.frame_count * 0.2f) * (1.0f - (float)y / height) * 10.0f;
    int src_x = ((int)(x + heat) + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #566 - HoloGlitch

Holographic glitch scanlines-fast-moving thin horizontal bands (5px) get color-shifted by adding different fixed amounts to R, G, B channels.

Technique: Moving scanline RGB offset bands.

```

// Case 566 in unifiedFilterKernel:
processHoloGlitch(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processHoloGlitch(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int scanline = (y + params.frame_count * 2) % 100;
    if (scanline < 5) {
        data[idx] = (data[idx] + 50) % 256;
        data[idx + 1] = (data[idx + 1] + 100) % 256;
        data[idx + 2] = (data[idx + 2] + 150) % 256;
    }
}

```

Filter #567 - InfraredView

False-color infrared visualization-remaps brightness to a thermal-IR palette: hot whites->yellow->red->purple->blue for different intensity ranges, with pulsing intensity.

Technique: Brightness-to-infrared false color mapping.

```

// Case 567 in unifiedFilterKernel:
processInfraredView(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processInfraredView(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int intensity = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float pulse = 0.8f + 0.2f * sinf(params.frame_count * 0.05f);
    if (intensity > 180) {
        data[idx] = 255;
        data[idx + 1] = 255;
        data[idx + 2] = 200;
    } else if (intensity > 140) {
        data[idx] = 0;
        data[idx + 1] = (unsigned char)(255 * pulse);
        data[idx + 2] = 255;
    } else if (intensity > 100) {

```

```

        data[idx] = 0;
        data[idx + 1] = 100;
        data[idx + 2] = (unsigned char)(200 * pulse);
    } else {
        data[idx] = (unsigned char)(50 * pulse);
        data[idx + 1] = 0;
        data[idx + 2] = 100;
    }
}

```

Filter #568 - LavaLamp

Lava lamp blobs-two overlapping sinusoidal blob patterns create organic flowing shapes that shift between warm red/orange and cool green.

Technique: Dual-frequency sinusoidal blob color blending.

```

// Case 568 in unifiedFilterKernel:
processLavaLamp(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processLavaLamp(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float t = params.frame_count * 0.02f;
    float blob1 = sinf(x * 0.02f + t) * cosf(y * 0.02f + t * 0.7f);
    float blob2 = cosf(x * 0.03f - t * 0.5f) * sinf(y * 0.03f + t);
    float blend = (blob1 + blob2 + 2.0f) / 4.0f;
    data[idx + 2] = (unsigned char)(data[idx + 2] * blend + 200 * (1.0f - blend));
    data[idx + 1] = (unsigned char)(data[idx + 1] * (1.0f - blend) + 100 * blend);
}

```

Filter #569 - LensFlare

Simulated lens flare-a bright circular flare moves horizontally across the upper third of the frame, adding intense white where it overlaps.

Technique: Animated moving circular lens flare spot.

```

// Case 569 in unifiedFilterKernel:
processLensFlare(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processLensFlare(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int flare_x = width / 2 + (int)(100.0f * sinf(params.frame_count * 0.03f));
    int flare_y = height / 3;
    float dist = sqrtf((float)((x - flare_x) * (x - flare_x) + (y - flare_y) * (y - flare_y)));
    if (dist < 50) {
        float intensity = 1.0f - dist / 50.0f;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + 200 * intensity);
    }
}

```

Filter #570 - LightningBolt

Random lightning bolt-a jagged vertical bright line flashes briefly every 15 frames, with random horizontal jitter per row.

Technique: Periodic random vertical lightning flash.

```
// Case 570 in unifiedFilterKernel:
processLightningBolt(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processLightningBolt(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    if (params.frame_count % 15 < 2) {
        int bolt_x = width / 2 + (params.seed % 100) - 50;
        bolt_x += (int)(gpu_rand(y, params.seed, params.frame_count) * 30 - 15);
        if (abs(x - bolt_x) < 3) {
            data[idx] = 200;
            data[idx + 1] = 200;
            data[idx + 2] = 255;
        }
    }
}
```

Filter #571 - LiquidMetal

Liquid metal sheen-converts to metallic-looking surface using brightness-dependent silver/chrome coloring with animated reflections.

Technique: Brightness-based metallic color mapping.

```
// Case 571 in unifiedFilterKernel:
processLiquidMetal(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processLiquidMetal(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float shine = 0.5f + 0.5f * sinf(gray * 0.1f + params.frame_count * 0.1f);
    unsigned char metal = (unsigned char)(gray * shine + 128 * (1.0f - shine));
    data[idx] = metal;
    data[idx + 1] = metal;
    data[idx + 2] = (unsigned char)fminf(255.0f, metal * 1.1f);
}
```

Filter #572 - MatrixCode

Matrix code rain-columns of bright green characters rain downward with brighter leading edges and darker tails.

Technique: Vertical green code column animation.

```
// Case 572 in unifiedFilterKernel:
processMatrixCode(x, y, data, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processMatrixCode(int x, int y, unsigned char *data, int height, size_t step, const
↳ FilterParams &params) {
    int idx = y * step + x * 4;
    int col = x / 8;
    int drop = ((params.frame_count * 3 + col * 17) % (height + 50)) - 25;
    if (y > drop && y < drop + 25) {
        float intensity = 1.0f - (float)(y - drop) / 25.0f;
        data[idx] = 0;
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + 200 * intensity);
        data[idx + 2] = 0;
    }
}

```

Filter #573 - MirrorKaleid

Multi-mirror kaleidoscope-more complex mirror folding than KaleidoScope4, with additional angular reflections for richer symmetry.

Technique: Multi-angle mirror kaleidoscope.

```

// Case 573 in unifiedFilterKernel:
processMirrorKaleid(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processMirrorKaleid(int x, int y, unsigned char *data, int width, int height, size_t
↳ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int mx = (x < width / 2) ? x : width - 1 - x;
    int my = (y < height / 2) ? y : height - 1 - y;
    mx = (mx + params.frame_count) % (width / 2);
    int src_idx = my * step + mx * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #574 - NightVision

Night vision simulation-converts to bright green monochrome with intensity amplification and slight noise, mimicking night-vision goggles.

Technique: Green monochrome conversion + amplification + noise.

```

// Case 574 in unifiedFilterKernel:
processNightVision(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processNightVision(int x, int y, unsigned char *data, size_t step, const FilterParams &
↳ params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] * 30 + data[idx + 1] * 59 + data[idx + 2] * 11) / 100;
    float noise = (gpu_rand(x, y, params.seed + params.frame_count) - 0.5f) * 20.0f;
    gray = (int)fmaxf(0.0f, fminf(255.0f, gray + noise));
    data[idx] = 0;
    data[idx + 1] = (unsigned char)(gray);
    data[idx + 2] = 0;
}

```

Filter #575 - OilSlick

Oil slick rainbow-maps thin-film interference-style rainbow colors based on angle and distance, creating iridescent oil-on-water coloring.

Technique: Angle/distance thin-film interference coloring.

```
// Case 575 in unifiedFilterKernel:
processOilSlick(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processOilSlick(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float phase = x * 0.05f + y * 0.03f + params.frame_count * 0.1f;
    float r_shift = sinf(phase) * 50.0f;
    float g_shift = sinf(phase + 2.094f) * 50.0f;
    float b_shift = sinf(phase + 4.189f) * 50.0f;
    data[idx] = (unsigned char)fmaxf(0.0f, fminf(255.0f, data[idx] + b_shift));
    data[idx + 1] = (unsigned char)fmaxf(0.0f, fminf(255.0f, data[idx + 1] + g_shift));
    data[idx + 2] = (unsigned char)fmaxf(0.0f, fminf(255.0f, data[idx + 2] + r_shift));
}
```

Filter #576 - ParticleField

Sparse bright particle dots scattered across the frame that animate over time, creating a starfield/particle effect overlay.

Technique: Random sparse bright pixel particle overlay.

```
// Case 576 in unifiedFilterKernel:
processParticleField(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processParticleField(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 4, y / 4, params.seed);
    if (r < 0.1f) {
        int py = (y + params.frame_count * 2) % 4;
        if (py == 0) {
            for (int j = 0; j < 3; ++j)
                data[idx + j] = 255;
        }
    }
}
```

Filter #577 - PinwheelSpin

Spinning pinwheel pattern-alternating angular sectors are darkened/brightened with the pattern rotating over time, like a spinning pinwheel.

Technique: Rotating angular sector brightness pattern.

```
// Case 577 in unifiedFilterKernel:
processPinwheelSpin(x, y, data, width, height, step, params);
```

```

// Referenced __device__ function(s):
__device__ void processPinwheelSpin(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float angle = atan2f(y - cy, x - cx) + params.frame_count * 0.05f;
    int sector = (int)((angle + 3.14159f) / 0.785f) % 8;
    if (sector % 2 == 0) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.6f);
    }
}

```

Filter #578 - PixelStorm

Aggressive random pixel displacement-pixels are randomly relocated in a storm-like fashion, creating a raging pixel noise storm.

Technique: Heavy random pixel coordinate displacement.

```

// Case 578 in unifiedFilterKernel:
processPixelStorm(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPixelStorm(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    if (r < 0.03f) {
        int ox = (int)((r * 1000) - 500) % 20;
        int oy = (int)((r * 500) - 250) % 20;
        int src_x = (x + ox + width) % width;
        int src_y = (y + oy + height) % height;
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
    }
}

```

Filter #579 - PlaidPattern

Overlapping horizontal and vertical color bands create a plaid/tartan pattern over the image, animated with scrolling.

Technique: Scrolling overlapping orthogonal color bands.

```

// Case 579 in unifiedFilterKernel:
processPlaidPattern(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPlaidPattern(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int stripe_x = (x + params.frame_count) % 40;
    int stripe_y = (y + params.frame_count) % 40;
}

```

```

float blend = 1.0f;
if (stripe_x < 8)
    blend *= 1.3f;
if (stripe_y < 8)
    blend *= 1.3f;
for (int j = 0; j < 3; ++j)
    data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * blend);
}

```

Filter #580 - PolarInvert

Polar coordinate inversion-maps pixels using inverted radial distance from center, creating a polar inside-out transformation.

Technique: Inverted polar coordinate mapping.

```

// Case 580 in unifiedFilterKernel:
processPolarInvert(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPolarInvert(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float max_dist = sqrtf(cx * cx + cy * cy);
    float threshold = max_dist * (0.5f + 0.3f * sinf(params.frame_count * 0.05f));
    if (dist > threshold) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = 255 - data[idx + j];
    }
}

```

Filter #581 - PolychromeTint

Multiple overlapping color tints applied across the frame-different regions get different color washes, creating a polychrome mosaic of tints.

Technique: Multi-region color tint overlay.

```

// Case 581 in unifiedFilterKernel:
processPolychromeTint(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPolychromeTint(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int zone = ((x / 50) + (y / 50) + params.frame_count / 10) % 6;
    float tints[6][3] = {{1.2f, 0.9f, 0.9f}, {0.9f, 1.2f, 0.9f}, {0.9f, 0.9f, 1.2f}, {1.2f, 1.2f, 0.9f},
    ↪ {1.2f, 0.9f, 1.2f}, {0.9f, 1.2f, 1.2f}};
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * tints[zone][j]);
}

```

Filter #582 - PopArtDots

Pop-art style large colored dots-renders oversized halftone dots in bold primary colors, simulating Roy Lichtenstein-style pop art.

Technique: Large bold-colored halftone dots.

```
// Case 582 in unifiedFilterKernel:
processPopArtDots(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processPopArtDots(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int dot_size = 8;
    int cx = (x / dot_size) * dot_size + dot_size / 2;
    int cy = (y / dot_size) * dot_size + dot_size / 2;
    float dist = sqrtf((float)((x - cx) * (x - cx) + (y - cy) * (y - cy)));
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float radius = (gray / 255.0f) * dot_size * 0.6f;
    if (dist < radius) {
        int color = ((cx / dot_size) + (cy / dot_size) + params.frame_count / 5) % 3;
        data[idx] = (color == 0) ? 255 : 0;
        data[idx + 1] = (color == 1) ? 255 : 0;
        data[idx + 2] = (color == 2) ? 255 : 0;
    } else {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = 255;
    }
}
```

Filter #583 - PrismaticEdge

Edge detection rendered with prismatic/rainbow colors-edges cycle through spectrum colors rather than white, creating rainbow outlines.

Technique: Rainbow-colored edge detection.

```
// Case 583 in unifiedFilterKernel:
processPrismaticEdge(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPrismaticEdge(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0 && y > 0 && x < width - 1 && y < height - 1) {
        int edge = 0;
        for (int j = 0; j < 3; ++j) {
            int diff = abs(data[idx + j] - data[(y + 1) * step + (x + 1) * 4 + j]);
            edge += diff;
        }
        if (edge > 50) {
            int hue = (x + y + params.frame_count * 3) % 256;
            data[idx + 2] = hue;
            data[idx + 1] = (hue + 85) % 256;
            data[idx] = (hue + 170) % 256;
        }
    }
}
```

Filter #584 - PulseWarp

Pulsing spatial warp-the image warps inward/outward from center with a sine-pulsed intensity, creating a breathing zoom effect.

Technique: Sine-pulsed radial zoom warp.

```
// Case 584 in unifiedFilterKernel:
processPulseWarp(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processPulseWarp(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float pulse = sinf(dist * 0.05f - params.frame_count * 0.1f) * 10.0f;
    float angle = atan2f(y - cy, x - cx);
    int src_x = (int)(cx + (dist + pulse) * cosf(angle)) % width;
    int src_y = (int)(cy + (dist + pulse) * sinf(angle)) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #585 - QuantumNoise

High-frequency random noise with quantum-inspired patterns-rapid, dense per-pixel noise that changes every frame.

Technique: Dense per-pixel per-frame random noise.

```
// Case 585 in unifiedFilterKernel:
processQuantumNoise(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processQuantumNoise(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    if (r < 0.1f) {
        int channel = (int)(r * 30) % 3;
        data[idx + channel] = (unsigned char)((int)(r * 2550) % 256);
    }
}
```

Filter #586 - QuiltBlend

Quilt-pattern blending-square patches alternate between current and history frames in a quilt-like grid layout.

Technique: Block-grid alternating temporal quilt blend.

```
// Case 586 in unifiedFilterKernel:
```

```

processQuiltBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processQuiltBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int patch_x = (x / 32) % 2;
    int patch_y = (y / 32) % 2;
    int pattern = (patch_x + patch_y + params.frame_count / 15) % 4;
    float factors[4][3] = {{1.2f, 0.8f, 0.8f}, {0.8f, 1.2f, 0.8f}, {0.8f, 0.8f, 1.2f}, {1.1f, 1.1f, 0.8f}
    ↪ };
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factors[pattern][j]);
}

```

Filter #587 - RadarSweep

Radar sweep—a rotating bright line sweeps clockwise from center like a radar display, brightening pixels it passes over.

Technique: Rotating angular bright sweep line.

```

// Case 587 in unifiedFilterKernel:
processRadarSweep(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRadarSweep(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float angle = atan2f(y - cy, x - cx);
    float sweep_angle = fmodf(params.frame_count * 0.05f, 6.28318f) - 3.14159f;
    float diff = fabsf(angle - sweep_angle);
    if (diff > 3.14159f)
        diff = 6.28318f - diff;
    if (diff < 0.3f) {
        float intensity = 1.0f - diff / 0.3f;
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + 150 * intensity);
    }
}

```

Filter #588 - RaindropRipple

Simulates raindrop ripples—circular wave distortions appear at random positions, creating concentric ring displacement patterns.

Technique: Random radial ripple displacement.

```

// Case 588 in unifiedFilterKernel:
processRaindropRipple(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRaindropRipple(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int drop_x = (params.seed % width);
    int drop_y = (params.seed / 2) % height;
}

```

```

float dist = sqrtf((float)((x - drop_x) * (x - drop_x) + (y - drop_y) * (y - drop_y)));
float ring = fmodf(params.frame_count * 2.0f, 100.0f);
if (fabsf(dist - ring) < 5.0f) {
    float wave = sinf((dist - ring) * 0.5f) * 0.3f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * (1.0f + wave));
}
}

```

Filter #589 - RasterBars

Retro raster bars-horizontal bars of bright color scroll vertically, creating Amiga/C64-style raster bar overlays.

Technique: Scrolling horizontal color bar overlay.

```

// Case 589 in unifiedFilterKernel:
processRasterBars(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processRasterBars(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int bar = (y + params.frame_count * 2) % 100;
    if (bar < 20) {
        float intensity = sinf(bar * 0.157f);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * (1.0f + intensity));
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * (1.0f + intensity * 0.5f));
    }
}

```

Filter #590 - RetroTube

CRT TV simulation-adds scanlines, slight curvature, color bleeding, and phosphor glow to mimic an old CRT television.

Technique: CRT scanlines + curvature + color bleed.

```

// Case 590 in unifiedFilterKernel:
processRetroTube(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRetroTube(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = (x - cx) / cx, dy = (y - cy) / cy;
    float dist_sq = dx * dx + dy * dy;
    float curve = 1.0f + dist_sq * 0.2f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] / curve);
    if ((y + params.frame_count) % 3 == 0) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.95f);
    }
}

```

Filter #591 - RingWave

Expanding ring waves from center-concentric bright rings pulse outward, modulating brightness as they pass.

Technique: Outward-expanding concentric ring brightness pulse.

```
// Case 591 in unifiedFilterKernel:
processRingWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRingWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float wave = sinf(dist * 0.1f - params.frame_count * 0.15f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * (0.8f + wave * 0.2f));
}
```

Filter #592 - RippleTank

Ripple tank physics simulation-overlapping circular wave patterns create interference fringe brightness modulation.

Technique: Circular wave interference pattern.

```
// Case 592 in unifiedFilterKernel:
processRippleTank(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processRippleTank(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float wave1 = sinf(x * 0.05f + params.frame_count * 0.1f);
    float wave2 = sinf(y * 0.05f + params.frame_count * 0.08f);
    float combined = (wave1 + wave2) * 5.0f;
    int src_x = ((int)(x + combined) + width) % width;
    int src_y = ((int)(y + combined) + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #593 - RotatingPrism

A rotating prism overlay that shifts color channels at different angular offsets, creating a spinning prismatic color separation.

Technique: Angle-rotation-based channel separation.

```
// Case 593 in unifiedFilterKernel:
processRotatingPrism(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processRotatingPrism(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float angle = atan2f(y - cy, x - cx) + params.frame_count * 0.02f;
    int sector = (int)((angle + 3.14159f) / 1.047f) % 6;
    float hue_shift = sector * 42.0f;
    data[idx + 2] = (unsigned char)fmodf(data[idx + 2] + hue_shift, 256.0f);
    data[idx + 1] = (unsigned char)fmodf(data[idx + 1] + hue_shift * 0.5f, 256.0f);
}

```

Filter #594 - SandStorm

Sand storm noise-dense brownish/tan noise particles scattered across the frame, simulating a sandstorm overlay.

Technique: Dense warm-tinted noise particle overlay.

```

// Case 594 in unifiedFilterKernel:
processSandStorm(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSandStorm(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    if (r < 0.15f) {
        int ox = (int)((r - 0.075f) * 40);
        int src_x = (x + ox + width) % width;
        int src_idx = y * step + src_x * 4;
        unsigned char sand = (unsigned char)(200 + r * 50);
        data[idx] = (data[src_idx] + sand) / 2;
        data[idx + 1] = (data[src_idx + 1] + sand - 20) / 2;
        data[idx + 2] = (data[src_idx + 2] + sand - 40) / 2;
    }
}

```

Filter #595 - SaturationPulse

Color saturation pulses-saturation is boosted and reduced with a sine wave, making colors alternately vivid and muted.

Technique: Sine-modulated saturation amplification.

```

// Case 595 in unifiedFilterKernel:
processSaturationPulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSaturationPulse(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.08f);
    for (int j = 0; j < 3; ++j) {
        float diff = data[idx + j] - gray;
        data[idx + j] = (unsigned char)fminf(255.0f, fmaxf(0.0f, gray + diff * (1.0f + pulse)));
    }
}

```

```
}
```

Filter #596 - ScatterPixel

Pixel scattering-each pixel is displaced to a nearby random neighbor position, creating a dissolved/scattered look.

Technique: Random neighbor pixel coordinate scatter.

```
// Case 596 in unifiedFilterKernel:
processScatterPixel(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processScatterPixel(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    int scatter = 2 + (params.frame_count % 5);
    int ox = (int)((r - 0.5f) * scatter * 2);
    int oy = (int)((gpu_rand(y, x, params.seed) - 0.5f) * scatter * 2);
    int src_x = (x + ox + width) % width;
    int src_y = (y + oy + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #597 - ShadowPlay

Enhances shadows with animated movement-dark areas are emphasized and their boundaries shift with an animated offset.

Technique: Animated shadow enhancement and displacement.

```
// Case 597 in unifiedFilterKernel:
processShadowPlay(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processShadowPlay(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int shadow_off = 5 + (params.frame_count % 10);
    int sx = (x + shadow_off) % width;
    int sy = (y + shadow_off) % height;
    int shadow_idx = sy * step + sx * 4;
    int shadow_val = (data[shadow_idx] + data[shadow_idx + 1] + data[shadow_idx + 2]) / 3;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * 0.7f + shadow_val * 0.1f);
}
```

Filter #598 - ShimmerGlass

Shimmering glass overlay-pixels are displaced by a rapidly oscillating fine-grain sine pattern, like looking through textured glass.

Technique: Fine-grain oscillating displacement shimmer.

```
// Case 598 in unifiedFilterKernel:
processShimmerGlass(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processShimmerGlass(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float shimmer = sinf(x * 0.2f + params.frame_count * 0.2f) * cosf(y * 0.2f + params.frame_count *
    ↪ 0.15f) * 3.0f;
    int src_x = ((int)(x + shimmer) + width) % width;
    int src_y = ((int)(y + shimmer) + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)((data[idx + j] + data[src_idx + j]) / 2);
}
```

Filter #599 - SilhouetteBlend

Silhouette extraction blended with original-creates a dark silhouette outline and blends it back at variable opacity.

Technique: Silhouette extraction + variable blend.

```
// Case 599 in unifiedFilterKernel:
processSilhouetteBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSilhouetteBlend(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    int threshold = 100 + (int)(50.0f * sinf(params.frame_count * 0.05f));
    if (gray < threshold) {
        float fade = (float)gray / threshold;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * fade);
    }
}
```

Visual FX B (600-699)

Filter #600 - SketchOutline

Pencil sketch effect-extracts edges using neighboring pixel differences and renders them as dark pencil-like outlines on a lighter background.

Technique: Edge-difference pencil outline rendering.

```
// Case 600 in unifiedFilterKernel:
processSketchOutline(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSketchOutline(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0 && y > 0 && x < width - 1 && y < height - 1) {
```

```

    int edge = 0;
    for (int j = 0; j < 3; ++j) {
        int gx = data[(y)*step + (x + 1) * 4 + j] - data[(y)*step + (x - 1) * 4 + j];
        int gy = data[(y + 1) * step + (x) * 4 + j] - data[(y - 1) * step + (x) * 4 + j];
        edge += abs(gx) + abs(gy);
    }
    edge = edge / 3;
    unsigned char sketch = (edge > 30) ? 0 : 255;
    float blend = 0.5f + 0.5f * sinf(params.frame_count * 0.03f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - blend) + sketch * blend);
}
}

```

Filter #601 - SliceShift

Horizontal slice displacement-random horizontal strips shift left or right, creating a sliced/torn glitch effect.

Technique: Random horizontal slice offset.

```

// Case 601 in unifiedFilterKernel:
processSliceShift(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processSliceShift(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int slice = y / 20;
    int shift = (slice % 2 == 0) ? (params.frame_count % 30) : -(params.frame_count % 30);
    int src_x = (x + shift + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #602 - SmearMotion

Horizontal smear-pixels are pulled from a row of neighboring pixels to one side, creating a directional smearing/motion blur effect.

Technique: Horizontal neighbor-sampling directional smear.

```

// Case 602 in unifiedFilterKernel:
processSmearMotion(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processSmearMotion(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int smear_len = 5 + (params.frame_count % 10);
    int sum[3] = {0, 0, 0};
    for (int i = 0; i < smear_len; ++i) {
        int sx = (x + i) % width;
        int sidx = y * step + sx * 4;
        for (int j = 0; j < 3; ++j)
            sum[j] += data[sidx + j];
    }
}

```



```

    }
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(sum[j] / smear_len);
}

```

Filter #603 - SmokeWisp

Smoke wisp overlay-soft, flowing cloud-like patterns drift across the frame, blending with the image at low opacity.

Technique: Flowing noise-based smoke overlay.

```

// Case 603 in unifiedFilterKernel:
processSmokeWisp(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSmokeWisp(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float t = params.frame_count * 0.03f;
    float wisp = sinf(x * 0.03f + t) * cosf(y * 0.02f + t * 0.7f) * 15.0f;
    int src_y = ((int)(y + wisp) + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j) {
        float blend = (data[idx + j] + data[src_idx + j] + 50) / 2.5f;
        data[idx + j] = (unsigned char)fminf(255.0f, blend);
    }
}

```

Filter #604 - SnowDrift

Snow drift effect-white particles drift diagonally downward, accumulating a scattered snow-like overlay.

Technique: Diagonal-drifting white particle overlay.

```

// Case 604 in unifiedFilterKernel:
processSnowDrift(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processSnowDrift(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count);
    if (r < 0.02f) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = 255;
    } else {
        float bright = 1.0f + 0.1f * sinf(params.frame_count * 0.05f);
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * bright);
    }
}

```

Filter #605 - SolarFlare

Solar flare from center-a bright warm radial burst emanates from a point, creating a lens-flare-like solar bloom effect.

Technique: Radial warm glow burst from center.

```
// Case 605 in unifiedFilterKernel:
processSolarFlare(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSolarFlare(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int flare_x = width / 4, flare_y = height / 4;
    float dist = sqrtf((float)((x - flare_x) * (x - flare_x) + (y - flare_y) * (y - flare_y)));
    float flare_radius = 30.0f + 20.0f * sinf(params.frame_count * 0.1f);
    if (dist < flare_radius) {
        float intensity = 1.0f - dist / flare_radius;
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + 200 * intensity);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + 150 * intensity);
    }
}
```

Filter #606 - SparkShower

Shower of bright spark pixels that rain downward, creating scattered bright falling dots.

Technique: Downward-falling bright pixel sparks.

```
// Case 606 in unifiedFilterKernel:
processSparkShower(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processSparkShower(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed);
    if (r < 0.005f) {
        int fall = (params.frame_count * 5 + (int)(r * 1000)) % height;
        if (abs(y - fall) < 3) {
            data[idx] = 100;
            data[idx + 1] = 200;
            data[idx + 2] = 255;
        }
    }
}
```

Filter #607 - SpectrumWave

Spectrum-colored wave-a horizontal wave pattern colors the image through the visible spectrum, creating a scrolling rainbow wave.

Technique: Horizontal sine-wave spectrum coloring.

```
// Case 607 in unifiedFilterKernel:
processSpectrumWave(x, y, data, step, params);
```

```
// Referenced __device__ function(s):
__device__ void processSpectrumWave(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float wave = sinf(x * 0.02f + y * 0.02f + params.frame_count * 0.1f);
    int hue = (int)((wave + 1.0f) * 127.5f);
    data[idx] = (unsigned char)((data[idx] + hue) / 2);
    data[idx + 1] = (unsigned char)((data[idx + 1] + (hue + 85) % 256) / 2);
    data[idx + 2] = (unsigned char)((data[idx + 2] + (hue + 170) % 256) / 2);
}
```

Filter #608 - SpiralZoom

Spiral zoom distortion-pixels are displaced both radially and rotationally from center based on their distance, creating a spiraling zoom tunnel.

Technique: Radial + rotational spiral displacement.

```
// Case 608 in unifiedFilterKernel:
processSpiralZoom(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSpiralZoom(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float angle = atan2f(dy, dx) + dist * 0.01f + params.frame_count * 0.02f;
    float zoom = 0.95f + 0.05f * sinf(params.frame_count * 0.05f);
    int src_x = (int)(cx + dist * zoom * cosf(angle)) % width;
    int src_y = (int)(cy + dist * zoom * sinf(angle)) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #609 - SplitMirror

Split mirror-the frame is divided into halves (animated split point) with one half mirrored, revealing/hiding the mirror effect over time.

Technique: Animated split-point half-frame mirror.

```
// Case 609 in unifiedFilterKernel:
processSplitMirror(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSplitMirror(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int split = width / 2 + (int)(50.0f * sinf(params.frame_count * 0.03f));
    if (x > split) {
        int mirror_x = split - (x - split);
        if (mirror_x >= 0) {
```

```

        int src_idx = y * step + mirror_x * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
    }
}

```

Filter #610 - StarBurst

Star burst pattern-radiating lines from center create a star/starburst brightness pattern that rotates over time.

Technique: Rotating radial line starburst pattern.

```

// Case 610 in unifiedFilterKernel:
processStarBurst(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processStarBurst(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float angle = atan2f(y - cy, x - cx);
    float ray = sinf(angle * 8.0f + params.frame_count * 0.1f);
    if (ray > 0.7f) {
        float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
        float max_dist = sqrtf(cx * cx + cy * cy);
        float fade = 1.0f - dist / max_dist;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + 100 * fade * ray);
    }
}

```

Filter #611 - StaticPulse

TV static that pulses-static noise density oscillates with a sine wave, creating periods of heavy and light static.

Technique: Sine-density-modulated static noise.

```

// Case 611 in unifiedFilterKernel:
processStaticPulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processStaticPulse(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pulse = 0.8f + 0.2f * sinf(params.frame_count * 0.15f);
    float r = gpu_rand(x, y, params.seed + params.frame_count / 3);
    if (r < 0.1f) {
        unsigned char noise = (unsigned char)(r * 2550) % 256;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * pulse + noise * (1.0f - pulse));
    }
}

```

Filter #612 - StencilCut

Stencil cut-out effect-removes image data in shaped patterns, leaving stencil-like cut-outs that reveal black or an alternate pattern.

Technique: Pattern-based pixel masking/removal.

```
// Case 612 in unifiedFilterKernel:
processStencilCut(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processStencilCut(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    int levels = 4;
    int quantized = (gray / (256 / levels)) * (256 / levels);
    int threshold = 64 + (params.frame_count % 128);
    if (gray < threshold)
        quantized = 0;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)((data[idx + j] + quantized) / 2);
}
```

Filter #613 - StippleShade

Stipple shading-converts brightness to a density of small dots, creating an artistic stippled/pointillist shading effect.

Technique: Brightness-to-dot-density stippling.

```
// Case 613 in unifiedFilterKernel:
processStippleShade(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processStippleShade(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float r = gpu_rand(x, y, params.seed);
    float threshold = gray / 255.0f;
    if (r > threshold) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.3f);
    }
}
```

Filter #614 - StormCloud

Storm cloud overlay-dark, roiling cloud patterns pass over the frame, dimming and tinting areas beneath them.

Technique: Animated noise-based dark cloud shadow.

```
// Case 614 in unifiedFilterKernel:
processStormCloud(x, y, data, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processStormCloud(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float cloud = sinf(x * 0.02f + params.frame_count * 0.05f) * cosf(y * 0.03f + params.frame_count *
    ↪ 0.03f);
    if (y < height / 3 && cloud > 0.3f) {
        float gray_factor = 0.5f + cloud * 0.3f;
        int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(avg * gray_factor);
    }
}

```

Filter #615 - StreakBlur

Directional streak blur-applies a linear motion blur along a single direction, creating speed-line-like streaks.

Technique: Single-direction linear motion blur.

```

// Case 615 in unifiedFilterKernel:
processStreakBlur(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processStreakBlur(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int streak_len = 8 + (params.frame_count % 8);
    int sum[3] = {0, 0, 0};
    int count = 0;
    for (int i = -streak_len; i <= streak_len; ++i) {
        int sx = (x + i + width) % width;
        int sidx = y * step + sx * 4;
        for (int j = 0; j < 3; ++j)
            sum[j] += data[sidx + j];
        count++;
    }
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(sum[j] / count);
}

```

Filter #616 - StrobeEdge

Edge detection that strobes-alternates between showing detected edges and the normal image on a rapid cycle.

Technique: Alternating edge-detect/normal frame strobe.

```

// Case 616 in unifiedFilterKernel:
processStrobeEdge(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processStrobeEdge(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0 && y > 0 && x < width - 1 && y < height - 1) {
        int edge = 0;
    }
}

```

```

        for (int j = 0; j < 3; ++j) {
            int diff = abs(data[idx + j] - data[(y + 1) * step + (x + 1) * 4 + j]);
            edge += diff;
        }
        if (edge > 40 && (params.frame_count % 4) < 2) {
            for (int j = 0; j < 3; ++j)
                data[idx + j] = 255;
        }
    }
}

```

Filter #617 - SubpixelShift

Sub-pixel chromatic shift-each RGB channel is offset by a sub-pixel amount, creating a subtle LCD-like color fringing.

Technique: Per-channel sub-pixel offset.

```

// Case 617 in unifiedFilterKernel:
processSubpixelShift(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processSubpixelShift(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int r_shift = 1 + (params.frame_count % 3);
    int b_shift = -1 - (params.frame_count % 3);
    int r_x = (x + r_shift + width) % width;
    int b_x = (x + b_shift + width) % width;
    data[idx + 2] = data[y * step + r_x * 4 + 2];
    data[idx] = data[y * step + b_x * 4];
}

```

Filter #618 - SwimDistort

Swimming distortion-pixels wobble in both X and Y with slow sine oscillation, creating an underwater swimming effect.

Technique: Dual-axis slow sine coordinate wobble.

```

// Case 618 in unifiedFilterKernel:
processSwimDistort(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processSwimDistort(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float t = params.frame_count * 0.05f;
    float dx = sinf(y * 0.05f + t) * 8.0f;
    float dy = cosf(x * 0.05f + t * 0.8f) * 8.0f;
    int src_x = ((int)(x + dx) + width) % width;
    int src_y = ((int)(y + dy) + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #619 - TangentWarp

Tangent-based warp-uses tangent function for displacement, creating discontinuous warping at tangent poles for aggressive distortion.

Technique: Tangent-function coordinate warping.

```
// Case 619 in unifiedFilterKernel:
processTangentWarp(x, y, data, width, height, step, params);

// Referenced __device__ function(s):

__device__ void processTangentWarp(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float nx = (x - width / 2.0f) / (width / 2.0f);
    float ny = (y - height / 2.0f) / (height / 2.0f);
    float warp = tanf(nx * 0.5f) * tanf(ny * 0.5f) * 20.0f * sinf(params.frame_count * 0.05f);
    int src_x = ((int)(x + warp) + width) % width;
    int src_y = ((int)(y + warp) + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #620 - TapeGlitch

VHS tape glitch-horizontal strips shift sideways with noise, and occasional tracking bars roll through, simulating degraded VHS playback.

Technique: Horizontal strip offset + rolling tracking bar.

```
// Case 620 in unifiedFilterKernel:
processTapeGlitch(x, y, data, width, step, params);

// Referenced __device__ function(s):

__device__ void processTapeGlitch(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int glitch_line = (params.seed + params.frame_count * 7) % 50;
    if ((y % 50) < glitch_line && (y % 50) > glitch_line - 5) {
        int shift = (params.frame_count % 30) - 15;
        int src_x = (x + shift + width) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * 1.3f);
    }
}
```

Filter #621 - TechnoGrid

Techno grid overlay-bright neon-colored grid lines in a techy pattern overlay the image, pulsing to an animated rhythm.

Technique: Animated neon grid overlay.


```

// Case 621 in unifiedFilterKernel:
processTechnoGrid(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processTechnoGrid(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int gx = x % 16, gy = y % 16;
    int pulse = (params.frame_count % 32);
    if (gx == 0 || gy == 0) {
        float bright = (pulse < 16) ? pulse / 16.0f : (32 - pulse) / 16.0f;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] + 100 * bright);
    }
}

```

Filter #622 - TeleportPixel

Pixel teleportation-random pixels are swapped to distant positions, creating a teleportation/scatter effect.

Technique: Random long-distance pixel position swap.

```

// Case 622 in unifiedFilterKernel:
processTeleportPixel(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTeleportPixel(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 8, y / 8, params.seed + params.frame_count / 5);
    if (r < 0.1f) {
        int tx = (int)(r * width * 10) % width;
        int ty = (int)(r * height * 7) % height;
        int src_idx = ty * step + tx * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
    }
}

```

Filter #623 - TemporalBlur

Multi-frame temporal blur-averages multiple history frames to create a smooth motion blur over time.

Technique: Multi-frame temporal averaging.

```

// Case 623 in unifiedFilterKernel:
processTemporalBlur(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processTemporalBlur(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float blur = 0.7f + 0.3f * sinf(params.frame_count * 0.02f + x * 0.01f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * blur + 128 * (1.0f - blur));
}

```

```
}
```

Filter #624 - TerraFracture

Terra/ground fracture effect-irregular fracture lines cut across the frame with offset displacement on each side, like cracking earth.

Technique: Irregular fracture-line displacement.

```
// Case 624 in unifiedFilterKernel:
processTerraFracture(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTerraFracture(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int block = (x / 40) + (y / 40) * (width / 40);
    float r = gpu_rand(block, params.seed, params.frame_count / 10);
    if (r < 0.3f) {
        int ox = (int)((r - 0.15f) * 20);
        int oy = (int)((r - 0.15f) * 20);
        int src_x = (x + ox + width) % width;
        int src_y = (y + oy + height) % height;
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
    }
}
```

Filter #625 - TextureWave

Texture wave-applies a sine-wave distortion that follows the image's natural texture/brightness contours.

Technique: Brightness-contour-following sine wave.

```
// Case 625 in unifiedFilterKernel:
processTextureWave(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processTextureWave(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float wave = sinf(x * 0.1f + params.frame_count * 0.1f) * cosf(y * 0.1f);
    float texture = (wave + 1.0f) * 0.5f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * (0.7f + texture * 0.3f));
}
```

Filter #626 - ThresholdPulse

Pulsing threshold-converts to binary black/white with a threshold that sweeps from dark to bright and back, revealing different detail levels.

Technique: Sweeping binary threshold animation.

```
// Case 626 in unifiedFilterKernel:
processThresholdPulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processThresholdPulse(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    int threshold = 64 + (int)(128.0f * fabsf(sinf(params.frame_count * 0.05f)));
    if (gray > threshold) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * 1.3f);
    } else {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.7f);
    }
}
```

Filter #627 - TidalWave

Tidal wave-a large rolling wave distortion sweeps horizontally across the frame, pushing pixels sideways in a crest pattern.

Technique: Horizontal rolling wave displacement.

```
// Case 627 in unifiedFilterKernel:
processTidalWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTidalWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf(x * 0.02f - params.frame_count * 0.1f) * 30.0f * (1.0f - (float)y / height);
    int src_y = ((int)(y + wave) + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #628 - TintCycle

Color tint cycling-the overall color tint rotates through warm->cool->warm in a continuous smooth cycle.

Technique: Smooth continuous color temperature cycling.

```
// Case 628 in unifiedFilterKernel:
processTintCycle(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processTintCycle(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int cycle = params.frame_count % 90;
    float r_tint = (cycle < 30) ? 1.2f : 0.9f;
    float g_tint = (cycle >= 30 && cycle < 60) ? 1.2f : 0.9f;
    float b_tint = (cycle >= 60) ? 1.2f : 0.9f;
```

```

    data[idx] = (unsigned char)fminf(255.0f, data[idx] * b_tint);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * g_tint);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * r_tint);
}

```

Filter #629 - TraceEdge

Edge tracing-detects and draws thin, continuous edge lines like a pen tracing the image contours.

Technique: Thin continuous edge line rendering.

```

// Case 629 in unifiedFilterKernel:
processTraceEdge(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTraceEdge(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0 && y > 0 && x < width - 1 && y < height - 1) {
        int edge = 0;
        for (int j = 0; j < 3; ++j) {
            edge += abs(data[y * step + (x + 1) * 4 + j] - data[y * step + (x - 1) * 4 + j]);
            edge += abs(data[(y + 1) * step + x * 4 + j] - data[(y - 1) * step + x * 4 + j]);
        }
        float trace = fminf(1.0f, edge / 200.0f);
        float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.1f);
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - trace * pulse) + 255 * trace * pulse
    ↪ );
    }
}

```

Filter #630 - TriangleMosaic

Triangle mosaic-divides the image into triangular tiles and fills each with its average color, creating a geometric low-poly look.

Technique: Triangular tile average-color mosaic.

```

// Case 630 in unifiedFilterKernel:
processTriangleMosaic(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTriangleMosaic(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size = 20 + (params.frame_count % 20);
    int tx = x / size, ty = y / size;
    int lx = x % size, ly = y % size;
    int cx, cy;
    if (lx + ly < size) {
        cx = tx * size;
        cy = ty * size;
    } else {
        cx = tx * size + size;
        cy = ty * size + size;
    }
}

```

```

    cx = cx < width ? cx : width - 1;
    cy = cy < height ? cy : height - 1;
    int src_idx = cy * step + cx * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #631 - TripleSplit

Triple split-divides the frame into three horizontal or vertical strips, each from a different time point.

Technique: Three-strip temporal split display.

```

// Case 631 in unifiedFilterKernel:
processTripleSplit(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processTripleSplit(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int third = width / 3;
    int shift = (params.frame_count % 20) - 10;
    if (x < third) {
        int src_x = (x - shift + width) % width;
        data[idx + 2] = data[y * step + src_x * 4 + 2];
    } else if (x >= 2 * third) {
        int src_x = (x + shift + width) % width;
        data[idx] = data[y * step + src_x * 4];
    }
}

```

Filter #632 - TurbulentFlow

Turbulent flow field-complex multi-frequency noise drives pixel displacement, creating chaotic fluid-like distortion.

Technique: Multi-frequency noise flow displacement.

```

// Case 632 in unifiedFilterKernel:
processTurbulentFlow(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processTurbulentFlow(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float t = params.frame_count * 0.03f;
    float turb_x = sinf(y * 0.03f + t) * cosf(x * 0.02f + t * 0.7f) * 10.0f;
    float turb_y = cosf(x * 0.03f + t * 0.5f) * sinf(y * 0.02f + t) * 10.0f;
    int src_x = ((int)(x + turb_x) + width) % width;
    int src_y = ((int)(y + turb_y) + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #633 - UnderwaterCaustic

Underwater caustic lighting-bright, shifting caustic light patterns dance across the frame, simulating light shining through water.

Technique: Sine-interference caustic light pattern.

```
// Case 633 in unifiedFilterKernel:
processUnderwaterCaustic(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processUnderwaterCaustic(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float caustic = sinf(x * 0.1f + params.frame_count * 0.15f) * sinf(y * 0.1f + params.frame_count *
    ↪ 0.1f);
    caustic = (caustic + 1.0f) * 0.5f;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] * (0.8f + caustic * 0.4f));
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * (0.9f + caustic * 0.3f));
}
```

Filter #634 - UnsharpPulse

Pulsing unsharp mask-sharpening intensity oscillates, making the image alternately sharp and soft with a rhythmic pulse.

Technique: Oscillating unsharp-mask intensity.

```
// Case 634 in unifiedFilterKernel:
processUnsharpPulse(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processUnsharpPulse(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    if (x > 0 && y > 0 && x < width - 1 && y < height - 1) {
        float amount = 0.5f + 0.5f * sinf(params.frame_count * 0.08f);
        for (int j = 0; j < 3; ++j) {
            int center = data[idx + j] * 5;
            int neighbors = data[(y - 1) * step + x * 4 + j] + data[(y + 1) * step + x * 4 + j] +
                data[y * step + (x - 1) * 4 + j] + data[y * step + (x + 1) * 4 + j];
            float sharp = center - neighbors;
            data[idx + j] = (unsigned char)fmaxf(0.0f, fminf(255.0f, data[idx + j] + sharp * amount *
            ↪ 0.25f));
        }
    }
}
```

Filter #635 - VaporTrail

Vapor trail afterimages-moving bright areas leave fading white vapor-like trails behind them.

Technique: Bright-pixel temporal fading trail.

```
// Case 635 in unifiedFilterKernel:
processVaporTrail(x, y, data, width, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processVaporTrail(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int trail_len = 10 + (params.frame_count % 10);
    float sum[3] = {0, 0, 0};
    for (int i = 0; i < trail_len; ++i) {
        int sx = (x - i + width) % width;
        float weight = 1.0f - (float)i / trail_len;
        for (int j = 0; j < 3; ++j)
            sum[j] += data[y * step + sx * 4 + j] * weight;
    }
    float total_weight = trail_len * 0.5f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(sum[j] / total_weight);
}

```

Filter #636 - VectorField

Vector field visualization-pixel displacement follows a simulated vector field pattern, creating flowing directional distortion.

Technique: Vector-field-directed pixel displacement.

```

// Case 636 in unifiedFilterKernel:
processVectorField(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processVectorField(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float angle = sinf(x * 0.02f) * cosf(y * 0.02f) * 3.14159f;
    float mag = 5.0f + 3.0f * sinf(params.frame_count * 0.05f);
    int src_x = ((int)(x + mag * cosf(angle)) + width) % width;
    int src_y = ((int)(y + mag * sinf(angle)) + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #637 - VelocityBlur

Velocity-proportional blur-pixels that have moved more between frames get more blur, creating a realistic motion-dependent blur.

Technique: Motion-proportional directional blur.

```

// Case 637 in unifiedFilterKernel:
processVelocityBlur(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processVelocityBlur(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float blur_len = dist * 0.02f * (1.0f + sinf(params.frame_count * 0.05f));
}

```

```

float nx = dx / (dist + 0.001f), ny = dy / (dist + 0.001f);
int sum[3] = {0, 0, 0};
int count = 0;
for (int i = 0; i < 5; ++i) {
    int sx = (int)(x - nx * i * blur_len) % width;
    int sy = (int)(y - ny * i * blur_len) % height;
    sx = (sx + width) % width;
    sy = (sy + height) % height;
    for (int j = 0; j < 3; ++j)
        sum[j] += data[sy * step + sx * 4 + j];
    count++;
}
for (int j = 0; j < 3; ++j)
    data[idx + j] = (unsigned char)(sum[j] / count);
}

```

Filter #638 - VerticalMelt

Vertical melt-pixels drip/melt downward with animated speed, creating a vertical smearing melt effect.

Technique: Downward vertical pixel displacement melt.

```

// Case 638 in unifiedFilterKernel:
processVerticalMelt(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processVerticalMelt(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float melt = sinf(x * 0.05f + params.frame_count * 0.1f) * 20.0f;
    int src_y = ((int)(y - melt) + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)((data[idx + j] + data[src_idx + j]) / 2);
}

```

Filter #639 - VHSTracking

VHS tracking distortion-horizontal offset bands roll through the frame with color bleeding, simulating bad VHS tracking.

Technique: Rolling horizontal offset + color bleed bands.

```

// Case 639 in unifiedFilterKernel:
processVHSTracking(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processVHSTracking(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int band = (y + params.frame_count * 3) % 200;
    if (band < 10) {
        int shift = (int)(sinf(band * 0.5f + params.frame_count * 0.2f) * 30);
        int src_x = (x + shift + width) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j)

```



```

        data[idx + j] = data[src_idx + j];
    }
    if ((y + params.frame_count) % 3 == 0) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.95f);
    }
}

```

Filter #640 - VibrantPop

Vibrant color pop-saturates colors to maximum and increases contrast for a bold, eye-popping vivid look.

Technique: Saturation + contrast amplification.

```

// Case 640 in unifiedFilterKernel:
processVibrantPop(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processVibrantPop(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float pop = 1.2f + 0.3f * sinf(params.frame_count * 0.08f);
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    for (int j = 0; j < 3; ++j) {
        float diff = data[idx + j] - gray;
        data[idx + j] = (unsigned char)fmaxf(0.0f, fminf(255.0f, gray + diff * pop));
    }
}

```

Filter #641 - VignetteFlash

Vignette with flash-a dark vignette frames the image but periodically flashes bright, creating a camera-flash-within-vignette effect.

Technique: Pulsing vignette with periodic bright flash.

```

// Case 641 in unifiedFilterKernel:
processVignetteFlash(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processVignetteFlash(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float max_dist = sqrtf(cx * cx + cy * cy);
    float vignette = 1.0f - (dist / max_dist) * 0.5f;
    float flash = 0.8f + 0.2f * sinf(params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * vignette * flash);
}

```

Filter #642 - VoronoiShatter

Voronoi diagram shatter-divides the image into Voronoi cells that displace/shatter apart over time.

Technique: Voronoi-cell-based displacement shatter.

```
// Case 642 in unifiedFilterKernel:
processVoronoiShatter(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processVoronoiShatter(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cell_size = 30 + (params.frame_count % 20);
    int cx = ((x / cell_size) * cell_size + cell_size / 2) % width;
    int cy = ((y / cell_size) * cell_size + cell_size / 2) % height;
    float r = gpu_rand(cx, cy, params.seed);
    cx += (int)((r - 0.5f) * cell_size * 0.5f);
    cy += (int)((r - 0.5f) * cell_size * 0.5f);
    cx = (cx + width) % width;
    cy = (cy + height) % height;
    int src_idx = cy * step + cx * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #643 - WarpSpeed

Warp speed radial zoom-pixels streak radially from center as if traveling at warp speed, creating a hyperspace zoom effect.

Technique: Radial streak zoom from center.

```
// Case 643 in unifiedFilterKernel:
processWarpSpeed(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processWarpSpeed(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float warp = 1.0f + 0.3f * sinf(params.frame_count * 0.1f - dist * 0.02f);
    int src_x = (int)(cx + dx * warp) % width;
    int src_y = (int)(cy + dy * warp) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #644 - WaterColor

Watercolor painting effect-simplifies colors and softens edges to create a painted watercolor look.

Technique: Color simplification + edge softening.

```
// Case 644 in unifiedFilterKernel:
processWaterColor(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processWaterColor(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count / 5);
    int ox = (int)((r - 0.5f) * 6);
    int oy = (int)((gpu_rand(y, x, params.seed) - 0.5f) * 6);
    int src_x = (x + ox + width) % width;
    int src_y = (y + oy + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j) {
        float blend = (data[idx + j] + data[src_idx + j]) / 2.0f;
        data[idx + j] = (unsigned char)fminf(255.0f, blend * 1.1f);
    }
}
```

Filter #645 - WaveCollapse

Wave collapse animation-expanding wave patterns collapse inward then burst outward, creating a rhythmic wave interference pattern.

Technique: Alternating expand/collapse radial wave.

```
// Case 645 in unifiedFilterKernel:
processWaveCollapse(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processWaveCollapse(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float wave1 = sinf(x * 0.03f + params.frame_count * 0.1f);
    float wave2 = cosf(y * 0.03f + params.frame_count * 0.08f);
    float collapse = wave1 * wave2 * 15.0f;
    int src_x = ((int)(x + collapse) + width) % width;
    int src_y = ((int)(y + collapse) + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #646 - WebPattern

Spider web pattern overlay-radiating lines and concentric rings create a web-like geometric pattern.

Technique: Radial + concentric web grid pattern.

```
// Case 646 in unifiedFilterKernel:
processWebPattern(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processWebPattern(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
```

```

    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float angle = atan2f(y - cy, x - cx);
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    int radial = (int)(dist / 20 + params.frame_count * 0.1f) % 2;
    int angular = (int)((angle + 3.14159f) * 8 / 3.14159f) % 2;
    if (radial == angular) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.7f);
    }
}

```

Filter #647 - WhirlpoolSpin

Whirlpool spinning vortex-pixels spiral inward toward center with increasing angular velocity, creating a draining whirlpool effect.

Technique: Distance-proportional angular rotation vortex.

```

// Case 647 in unifiedFilterKernel:
processWhirlpoolSpin(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processWhirlpoolSpin(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float angle = atan2f(dy, dx) + dist * 0.01f * sinf(params.frame_count * 0.05f);
    int src_x = (int)(cx + dist * cosf(angle)) % width;
    int src_y = (int)(cy + dist * sinf(angle)) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #648 - WindBlast

Wind blast distortion-pixels are pushed strongly to one side with a turbulent offset, simulating a powerful wind.

Technique: Directional turbulent pixel push.

```

// Case 648 in unifiedFilterKernel:
processWindBlast(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processWindBlast(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int intensity = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    int wind = (255 - intensity) / 10 + (params.frame_count % 10);
    int src_x = (x - wind + width) % width;
    int src_idx = y * step + src_x * 4;
}

```

```

    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)((data[idx + j] + data[src_idx + j]) / 2);
}

```

Filter #649 - WireframePulse

Wireframe pulsing overlay-a wireframe grid is drawn over the image with line brightness that pulses.

Technique: Pulsing wireframe grid overlay.

```

// Case 649 in unifiedFilterKernel:
processWireframePulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processWireframePulse(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int grid = 20;
    int gx = x % grid, gy = y % grid;
    float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.1f);
    if (gx < 2 || gy < 2) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + pulse * 0.5f) + 128 * pulse);
    } else {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f);
    }
}

```

Filter #650 - XRayFlash

X-ray flash-periodically inverts to a blue-tinted negative reminiscent of X-ray film, then snaps back to normal.

Technique: Periodic blue-negative X-ray inversion.

```

// Case 650 in unifiedFilterKernel:
processXRayFlash(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processXRayFlash(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] * 30 + data[idx + 1] * 59 + data[idx + 2] * 11) / 100;
    float flash = 0.5f + 0.5f * sinf(params.frame_count * 0.1f);
    if ((params.frame_count / 10) % 2 == 0) {
        data[idx] = (unsigned char)(255 - gray);
        data[idx + 1] = (unsigned char)(255 - gray);
        data[idx + 2] = (unsigned char)fminf(255.0f, (255 - gray) * 1.2f);
    } else {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * flash);
    }
}

```

Filter #651 - ZebraStripe

Zebra stripe pattern-alternating black and white diagonal stripes scroll across the frame, blended with the image.

Technique: Scrolling diagonal zebra stripe blend.

```
// Case 651 in unifiedFilterKernel:
processZebraStripe(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processZebraStripe(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int stripe = (x + y + params.frame_count) % 20;
    float factor = (stripe < 10) ? 1.2f : 0.8f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * factor);
}
```

Filter #652 - ZenRipple

Gentle zen-garden ripples-very subtle, slow concentric ripples from center, creating a calm, meditative distortion.

Technique: Slow gentle radial ripple displacement.

```
// Case 652 in unifiedFilterKernel:
processZenRipple(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processZenRipple(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float ripple = sinf(dist * 0.05f - params.frame_count * 0.03f) * 0.2f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, fmaxf(0.0f, data[idx + j] * (1.0f + ripple)));
}
```

Filter #653 - ZigzagWave

Zigzag wave distortion-rows are displaced in a zigzag pattern that animates over time, creating a zigzag wave effect.

Technique: Animated zigzag row displacement.

```
// Case 653 in unifiedFilterKernel:
processZigzagWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processZigzagWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int period = 40;
    int phase = (y / period) % 2;
    int offset = (phase == 0) ? (params.frame_count % period) : -(params.frame_count % period);
```

```

    int src_x = (x + offset + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #654 - ZoomPulse

Zoom pulse-the image alternately zooms in and out from center with a pulsing sine animation, like a heartbeat zoom.

Technique: Sine-pulsed center zoom in/out.

```

// Case 654 in unifiedFilterKernel:
processZoomPulse(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processZoomPulse(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float zoom = 1.0f + 0.1f * sinf(params.frame_count * 0.08f);
    int src_x = (int)(cx + (x - cx) / zoom) % width;
    int src_y = (int)(cy + (y - cy) / zoom) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #655 - ZoneTint

Zone tinting-divides the frame into zones that each get a different color tint, with zone boundaries animating over time.

Technique: Variable-zone animated color tinting.

```

// Case 655 in unifiedFilterKernel:
processZoneTint(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processZoneTint(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int zone_x = (x * 3) / width;
    int zone_y = (y * 3) / height;
    int zone = zone_x + zone_y * 3;
    zone = (zone + params.frame_count / 20) % 9;
    float tints[9][3] = {{1.3f, 0.9f, 0.9f}, {0.9f, 1.3f, 0.9f}, {0.9f, 0.9f, 1.3f}, {1.2f, 1.2f, 0.8f},
    ↪ {1.2f, 0.8f, 1.2f}, {0.8f, 1.2f, 1.2f}, {1.1f, 1.0f, 0.9f}, {0.9f, 1.1f, 1.0f}, {1.0f, 0.9f, 1.1
    ↪ f}};
    data[idx] = (unsigned char)fminf(255.0f, data[idx] * tints[zone][2]);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * tints[zone][1]);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * tints[zone][0]);
}

```

Filter #656 - AcidDrip

Vertical dripping distortion-pixels drip downward in long streaks like acid melting, with animated drip speed.

Technique: Long-streak vertical drip displacement.

```
// Case 656 in unifiedFilterKernel:
processAcidDrip(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processAcidDrip(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float drip = sinf(x * 0.1f + params.frame_count * 0.05f) * 20.0f;
    int src_y = ((int)(y + drip * (float)y / height) + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * 1.2f);
}
```

Filter #657 - AuroraWave

Aurora borealis effect-shimmering curtains of green/pink/blue light wave across the top of the frame.

Technique: Horizontal curtain sine-wave color overlay.

```
// Case 657 in unifiedFilterKernel:
processAuroraWave(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processAuroraWave(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float wave = sinf(x * 0.02f + params.frame_count * 0.05f) * sinf(y * 0.01f + params.frame_count *
    ↪ 0.03f);
    float aurora = (wave + 1.0f) * 0.5f * (1.0f - (float)y / height);
    if (aurora > 0.3f) {
        data[idx] = (unsigned char)fminf(255.0f, data[idx] + 50 * aurora);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + 100 * aurora);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + 30 * aurora);
    }
}
```

Filter #658 - BandPass

Band-pass filter-only allows a narrow range of brightness values through, dimming everything outside the animated pass band.

Technique: Animated brightness band-pass gating.

```
// Case 658 in unifiedFilterKernel:
processBandPass(x, y, data, step, params);
```



```
// Referenced __device__ function(s):

__device__ void processBandPass(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int low = 60 + (params.frame_count % 60);
    int high = 180 + (params.frame_count % 60);
    for (int j = 0; j < 3; ++j) {
        if (data[idx + j] < low)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f);
        else if (data[idx + j] > high)
            data[idx + j] = (unsigned char)(255 - (data[idx + j] - high));
    }
}
```

Filter #659 - BilinearStretch

Bilinear stretch-stretches the image non-uniformly using bilinear interpolation with animated anchor points.

Technique: Non-uniform bilinear stretch with moving anchors.

```
// Case 659 in unifiedFilterKernel:
processBilinearStretch(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBilinearStretch(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float sx = 1.0f + 0.3f * sinf(params.frame_count * 0.05f);
    float sy = 1.0f + 0.3f * cosf(params.frame_count * 0.05f);
    int src_x = (int)(x / sx) % width;
    int src_y = (int)(y / sy) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #660 - BleedThrough

Color bleed-through-colors from adjacent pixels bleed outward as if printed on wet paper, creating a watercolor-bleed effect.

Technique: Directional neighbor color bleeding.

```
// Case 660 in unifiedFilterKernel:
processBleedThrough(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processBleedThrough(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int bleed = 3 + (params.frame_count % 5);
    for (int j = 0; j < 3; ++j) {
        int sum = data[idx + j];
    }
}
```

```

        for (int b = 1; b <= bleed; ++b) {
            int bx = (x + b) % width;
            sum += data[y * step + bx * 4 + j] / (b + 1);
        }
        data[idx + j] = (unsigned char)fminf(255.0f, (float)sum);
    }
}

```

Filter #661 - BlockShatter

Block-level shatter-rectangular blocks fly apart in random directions over time, like an exploding mosaic.

Technique: Random-direction block explosion displacement.

```

// Case 661 in unifiedFilterKernel:
processBlockShatter(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlockShatter(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int bsize = 16 + (params.frame_count % 16);
    int bx = x / bsize, by = y / bsize;
    float r = gpu_rand(bx, by, params.seed + params.frame_count / 10);
    if (r < 0.3f) {
        int ox = (int)((r - 0.15f) * bsize);
        int oy = (int)((r - 0.15f) * bsize);
        int src_x = (x + ox + width) % width;
        int src_y = (y + oy + height) % height;
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
    }
}

```

Filter #662 - BlurMask

Selective blur mask-blurs only certain regions (e.g., by brightness or position) while keeping others sharp.

Technique: Region-selective conditional blur.

```

// Case 662 in unifiedFilterKernel:
processBlurMask(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBlurMask(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    int threshold = 128 + (int)(64.0f * sinf(params.frame_count * 0.05f));
    if (gray > threshold && x > 1 && y > 1 && x < width - 2 && y < height - 2) {
        for (int j = 0; j < 3; ++j) {
            int sum = data[idx + j] * 4;
            sum += data[(y - 1) * step + x * 4 + j];
            sum += data[(y + 1) * step + x * 4 + j];
        }
    }
}

```

```

        sum += data[y * step + (x - 1) * 4 + j];
        sum += data[y * step + (x + 1) * 4 + j];
        data[idx + j] = (unsigned char)(sum / 8);
    }
}

```

Filter #663 - BokehBlur

Bokeh blur simulation-out-of-focus areas render as circular bokeh blur discs, simulating shallow depth-of-field.

Technique: Circular disc blur (bokeh simulation).

```

// Case 663 in unifiedFilterKernel:
processBokehBlur(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBokehBlur(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float max_dist = sqrtf(cx * cx + cy * cy);
    int blur_size = (int)(dist / max_dist * 5.0f * (1.0f + sinf(params.frame_count * 0.05f)));
    if (blur_size > 0 && x > blur_size && y > blur_size && x < width - blur_size && y < height -
    ↪ blur_size) {
        for (int j = 0; j < 3; ++j) {
            int sum = 0, count = 0;
            for (int dy = -blur_size; dy <= blur_size; dy += 2) {
                for (int dx = -blur_size; dx <= blur_size; dx += 2) {
                    sum += data[(y + dy) * step + (x + dx) * 4 + j];
                    count++;
                }
            }
            data[idx + j] = (unsigned char)(sum / count);
        }
    }
}

```

Filter #664 - BounceWave

Bouncing wave-a sine wave with hard bounces (absolute value) displaces pixels, creating a rubber-bounce wave pattern.

Technique: Absolute-sine bouncing wave displacement.

```

// Case 664 in unifiedFilterKernel:
processBounceWave(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBounceWave(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float bounce = fabsf(sinf(x * 0.03f + params.frame_count * 0.1f)) * 15.0f;
    int src_y = ((int)(y + bounce) + height) % height;
    int src_idx = src_y * step + x * 4;
}

```

```

    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #665 - BrokenGlass

Broken glass fragment overlay-the image appears shattered into irregular glass-like shards with edge highlights.

Technique: Irregular polygon shard fragmentation.

```

// Case 665 in unifiedFilterKernel:
processBrokenGlass(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBrokenGlass(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int shard_x = x / 25, shard_y = y / 25;
    float r = gpu_rand(shard_x, shard_y, params.seed);
    float angle = r * 6.28318f;
    float offset = (r - 0.5f) * 10.0f * sinf(params.frame_count * 0.05f);
    int src_x = (int)(x + offset * cosf(angle)) % width;
    int src_y = (int)(y + offset * sinf(angle)) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #666 - BubbleWarp

Bubble/sphere warping-circular regions inflate outward like bubbles, warping the image in spherical bumps.

Technique: Circular spherical inflation warp.

```

// Case 666 in unifiedFilterKernel:
processBubbleWarp(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processBubbleWarp(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float bubble = sinf(dist * 0.05f - params.frame_count * 0.1f) * 10.0f;
    float angle = atan2f(dy, dx);
    int src_x = (int)(x + bubble * cosf(angle)) % width;
    int src_y = (int)(y + bubble * sinf(angle)) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

```
}
```

Filter #667 - CRTCurvature

CRT screen curvature-applies barrel distortion to simulate the curved surface of a CRT monitor, with scanlines and vignetting.

Technique: Barrel distortion + scanlines + vignette.

```
// Case 667 in unifiedFilterKernel:
processCRTCurvature(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processCRTCurvature(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float nx = (x - width / 2.0f) / (width / 2.0f);
    float ny = (y - height / 2.0f) / (height / 2.0f);
    float curve = 0.1f * (1.0f + 0.5f * sinf(params.frame_count * 0.03f));
    nx = nx * (1.0f + curve * ny * ny);
    ny = ny * (1.0f + curve * nx * nx);
    int src_x = (int)((nx + 1.0f) * width / 2.0f) % width;
    int src_y = (int)((ny + 1.0f) * height / 2.0f) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #668 - CascadeBlend

Cascading multi-frame blend-blends a cascading series of history frames with progressively decreasing weights.

Technique: Multi-frame cascading weighted blend.

```
// Case 668 in unifiedFilterKernel:
processCascadeBlend(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCascadeBlend(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int cascade = (y + params.frame_count) % 60;
    float factor = (cascade < 30) ? cascade / 30.0f : (60 - cascade) / 30.0f;
    for (int j = 0; j < 3; ++j) {
        data[idx + j] = (unsigned char)(data[idx + j] * (0.5f + factor * 0.5f));
    }
}
```

Filter #669 - CelShade

Cel shade/toon shader-quantizes colors to a few flat levels with hard edge outlines, creating a cartoon/anime aesthetic.

Technique: Color quantization + edge outline (toon).

```
// Case 669 in unifiedFilterKernel:
processCelShade(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCelShade(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int levels = 4 + (params.frame_count % 4);
    int step_size = 256 / levels;
    for (int j = 0; j < 3; ++j) {
        int val = data[idx + j];
        val = (val / step_size) * step_size + step_size / 2;
        data[idx + j] = (unsigned char)fminf(255.0f, (float)val);
    }
}
```

Filter #670 - ChainReaction

Chain reaction spread-bright pixels trigger their neighbors to brighten in the next frame, spreading outward like a chain reaction.

Technique: Brightness-threshold spreading activation.

```
// Case 670 in unifiedFilterKernel:
processChainReaction(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processChainReaction(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int ring = params.frame_count % 100;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    if (fabsf(dist - ring * 3) < 5.0f) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * 1.5f);
    }
}
```

Filter #671 - ChannelDelay

Per-channel temporal delay-each RGB channel is sourced from a different time point, creating a temporal chromatic separation.

Technique: Per-channel different-frame sourcing.

```
// Case 671 in unifiedFilterKernel:
processChannelDelay(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processChannelDelay(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int r_delay = 5 + (params.frame_count % 10);
    int b_delay = -(5 + (params.frame_count % 10));
```

```

    data[idx + 2] = data[y * step + ((x + r_delay + width) % width) * 4 + 2];
    data[idx] = data[y * step + ((x + b_delay + width) % width) * 4];
}

```

Filter #672 - ChromaBleed

Chroma bleed-color information bleeds horizontally from saturated areas, simulating analog video chroma bleed.

Technique: Horizontal color saturation bleed.

```

// Case 672 in unifiedFilterKernel:
processChromaBleed(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processChromaBleed(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int bleed = 5 + (params.frame_count % 8);
    int sum_r = 0, sum_g = 0, sum_b = 0;
    for (int i = 0; i < bleed; ++i) {
        int bx = (x + i) % width;
        sum_r += data[y * step + bx * 4 + 2];
        sum_g += data[y * step + bx * 4 + 1];
        sum_b += data[y * step + bx * 4];
    }
    data[idx + 2] = (unsigned char)(sum_r / bleed);
    data[idx + 1] = data[idx + 1];
    data[idx] = (unsigned char)(sum_b / bleed);
}

```

Filter #673 - CircuitTrace

Circuit board trace pattern-draws right-angle line patterns reminiscent of PCB traces overlaid on the image.

Technique: Right-angle grid circuit pattern overlay.

```

// Case 673 in unifiedFilterKernel:
processCircuitTrace(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCircuitTrace(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int grid = 12;
    int gx = x % grid, gy = y % grid;
    int node_x = x / grid, node_y = y / grid;
    float r = gpu_rand(node_x, node_y, params.seed);
    int trace = (r < 0.5f) ? (gx < 2) : (gy < 2);
    if (trace) {
        float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.1f + node_x + node_y);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + 150 * pulse);
    }
}

```

Filter #674 - ClockWipe

Clock wipe transition-a radial wipe rotates around center like a clock hand, revealing the history frame behind the current frame.

Technique: Animated radial clock-hand wipe.

```
// Case 674 in unifiedFilterKernel:
processClockWipe(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processClockWipe(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float angle = atan2f(y - cy, x - cx) + 3.14159f;
    float wipe_angle = fmodf(params.frame_count * 0.05f, 6.28318f);
    if (angle < wipe_angle) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = 255 - data[idx + j];
    }
}
```

Filter #675 - CloudShadow

Cloud shadow overlay-dark shadow patches drift across the frame as if clouds are passing overhead.

Technique: Drifting noise-based shadow patch overlay.

```
// Case 675 in unifiedFilterKernel:
processCloudShadow(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCloudShadow(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float cloud = sinf(x * 0.01f + params.frame_count * 0.02f) * cosf(y * 0.015f + params.frame_count *
    ↪ 0.015f);
    float shadow = (cloud > 0.3f) ? 0.6f : 1.0f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * shadow);
}
```

Filter #676 - ColorBurn

Color burn blend mode-applies the Photoshop-style "color burn" blend between current and history frames, deepening dark colors.

Technique: Color burn composite blend.

```
// Case 676 in unifiedFilterKernel:
processColorBurn(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processColorBurn(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float intensity = (data[idx] + data[idx + 1] + data[idx + 2]) / 765.0f;
```



```

float burn = intensity * intensity;
int channel = params.frame_count % 3;
data[idx + channel] = (unsigned char)(data[idx + channel] * burn + 255 * (1.0f - burn) * 0.3f);
}

```

Filter #677 - ColorHalves

Color halves-splits the frame into two halves with different color processing (e.g., warm left, cool right) with animated split.

Technique: Animated split-frame dual color grading.

```

// Case 677 in unifiedFilterKernel:
processColorHalves(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processColorHalves(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int split = width / 2 + (int)(50.0f * sinf(params.frame_count * 0.03f));
    if (x < split) {
        data[idx] = (unsigned char)(data[idx] * 0.5f);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * 1.5f);
    } else {
        data[idx + 2] = (unsigned char)(data[idx + 2] * 0.5f);
        data[idx] = (unsigned char)fminf(255.0f, data[idx] * 1.5f);
    }
}

```

Filter #678 - ComicDots

Comic book Benday dots-converts to large visible halftone dots in CMYK-like notation for a comic book print look.

Technique: Large color halftone Benday dot pattern.

```

// Case 678 in unifiedFilterKernel:
processComicDots(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processComicDots(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int dot_size = 6;
    int cx = (x / dot_size) * dot_size + dot_size / 2;
    int cy = (y / dot_size) * dot_size + dot_size / 2;
    float dist = sqrtf((float)((x - cx) * (x - cx) + (y - cy) * (y - cy)));
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    float radius = (gray / 255.0f) * dot_size * 0.7f;
    if (dist < radius) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = 0;
    } else {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = 255;
    }
}

```

Filter #679 - ConcentricPulse

Concentric brightness pulse-expanding rings of alternating brightness pulse outward from center.

Technique: Alternating expanding concentric brightness rings.

```
// Case 679 in unifiedFilterKernel:
processConcentricPulse(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processConcentricPulse(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float pulse = sinf(dist * 0.05f - params.frame_count * 0.15f);
    float factor = 0.7f + pulse * 0.3f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * factor);
}
```

Filter #680 - CopperTone

Copper/bronze tone-maps colors to a warm copper-bronze monochrome palette, like sepia but warmer/more metallic.

Technique: Copper/bronze monochrome color mapping.

```
// Case 680 in unifiedFilterKernel:
processCopperTone(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCopperTone(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] * 30 + data[idx + 1] * 59 + data[idx + 2] * 11) / 100;
    float pulse = 0.8f + 0.2f * sinf(params.frame_count * 0.05f);
    data[idx] = (unsigned char)(gray * 0.4f * pulse);
    data[idx + 1] = (unsigned char)(gray * 0.6f * pulse);
    data[idx + 2] = (unsigned char)(gray * 0.9f * pulse);
}
```

Filter #681 - CornerStretch

Corner stretch-warps the image by stretching corners outward, creating a pillow/pin-cushion type distortion from corners.

Technique: Corner-anchored outward stretch distortion.

```
// Case 681 in unifiedFilterKernel:
processCornerStretch(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processCornerStretch(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
```

```

float nx = (float)x / width, ny = (float)y / height;
float stretch = 0.2f * sinf(params.frame_count * 0.05f);
float corner_dist = fminf(fminf(nx, 1.0f - nx), fminf(ny, 1.0f - ny));
float factor = 1.0f + stretch * (1.0f - corner_dist * 4.0f);
factor = fmaxf(0.5f, fminf(1.5f, factor));
int src_x = (int)(width / 2.0f + (x - width / 2.0f) * factor) % width;
int src_y = (int)(height / 2.0f + (y - height / 2.0f) * factor) % height;
src_x = (src_x + width) % width;
src_y = (src_y + height) % height;
int src_idx = src_y * step + src_x * 4;
for (int j = 0; j < 3; ++j)
    data[idx + j] = data[src_idx + j];
}

```

Filter #682 - CosmicDust

Cosmic dust particles-overlays twinkling colored particles that drift slowly, creating a starfield/cosmic-dust effect.

Technique: Drifting colored particle twinkle overlay.

```

// Case 682 in unifiedFilterKernel:
processCosmicDust(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCosmicDust(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x, y, params.seed + params.frame_count / 3);
    if (r < 0.01f) {
        float bright = r * 100.0f;
        data[idx] = (unsigned char)fminf(255.0f, data[idx] + 150 * bright);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + 100 * bright);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + 200 * bright);
    }
}

```

Filter #683 - CrossBlur

Cross-shaped blur-blurs in a cross/plus pattern (horizontal + vertical only), creating a star-filter-like soft focus.

Technique: Orthogonal cross-pattern blur.

```

// Case 683 in unifiedFilterKernel:
processCrossBlur(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processCrossBlur(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int size = 3 + (params.frame_count % 5);
    if (x > size && y > size && x < width - size && y < height - size) {
        for (int j = 0; j < 3; ++j) {
            int sum = data[idx + j];
            for (int i = 1; i <= size; ++i) {
                sum += data[y * step + (x - i) * 4 + j];
            }
        }
    }
}

```

```

        sum += data[y * step + (x + i) * 4 + j];
        sum += data[(y - i) * step + x * 4 + j];
        sum += data[(y + i) * step + x * 4 + j];
    }
    data[idx + j] = (unsigned char)(sum / (size * 4 + 1));
}
}
}

```

Filter #684 - CrossProcess

Cross-process film look-shifted color curves simulating cross-processed film (E-6 in C-41), with boosted greens and contrasted reds.

Technique: Cross-process color curve simulation.

```

// Case 684 in unifiedFilterKernel:
processCrossProcess(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCrossProcess(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float r = data[idx + 2] / 255.0f;
    float g = data[idx + 1] / 255.0f;
    float b = data[idx] / 255.0f;
    float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.05f);
    r = powf(r, 0.8f + pulse * 0.2f);
    g = powf(g, 1.0f);
    b = powf(b, 1.2f - pulse * 0.2f);
    data[idx + 2] = (unsigned char)(fminf(1.0f, r * 1.1f) * 255);
    data[idx + 1] = (unsigned char)(g * 255);
    data[idx] = (unsigned char)(fminf(1.0f, b * 0.9f) * 255);
}

```

Filter #685 - CrystalEdge

Crystal edge highlight-edges are highlighted with bright crystalline-colored outlines, creating a jewel-like edge effect.

Technique: Bright crystal-colored edge highlighting.

```

// Case 685 in unifiedFilterKernel:
processCrystalEdge(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processCrystalEdge(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int facet_size = 20 + (params.frame_count % 20);
    int fx = (x / facet_size) * facet_size;
    int fy = (y / facet_size) * facet_size;
    int edge_x = x - fx, edge_y = y - fy;
    if (edge_x < 2 || edge_y < 2 || edge_x > facet_size - 3 || edge_y > facet_size - 3) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * 1.5f);
    }
}

```

```
}
```

Filter #686 - CubeRotate

Cube rotation-the frame is mapped onto a 3D cube face that rotates over time, showing the image from different angles with perspective.

Technique: 3D perspective cube-face rotation mapping.

```
// Case 686 in unifiedFilterKernel:
processCubeRotate(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processCubeRotate(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float angle = params.frame_count * 0.02f;
    float cos_a = cosf(angle), sin_a = sinf(angle);
    int src_x = (int)(cx + dx * cos_a - dy * sin_a) % width;
    int src_y = (int)(cy + dx * sin_a + dy * cos_a) % height;
    src_x = (src_x + width) % width;
    src_y = (src_y + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #687 - CurtainReveal

Curtain reveal transition-the image opens from center like parting curtains, revealing a history frame behind.

Technique: Center-opening curtain wipe transition.

```
// Case 687 in unifiedFilterKernel:
processCurtainReveal(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processCurtainReveal(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int curtain = (params.frame_count * 3) % width;
    if (x < curtain) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = 255 - data[idx + j];
    }
}
```

Filter #688 - CyberPunk

Cyberpunk aesthetic-high contrast with neon magenta/cyan color grading, scanlines, and glitch artifacts for a cyberpunk look.

Technique: Neon color grade + scanlines + glitch.

```
// Case 688 in unifiedFilterKernel:
processCyberPunk(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processCyberPunk(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int scan = (y + params.frame_count * 2) % 4;
    float cyan = (scan < 2) ? 1.3f : 0.9f;
    float magenta = (scan >= 2) ? 1.3f : 0.9f;
    data[idx] = (unsigned char)fminf(255.0f, data[idx] * cyan);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * cyan);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * magenta);
}
```

Filter #689 - DataCorrupt

Data corruption simulation-randomly corrupts pixel data in block-sized chunks, mimicking file corruption artifacts.

Technique: Random block data corruption.

```
// Case 689 in unifiedFilterKernel:
processDataCorrupt(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processDataCorrupt(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 8, y / 4, params.seed + params.frame_count / 5);
    if (r < 0.05f) {
        int shift = (int)(r * 500) % 50 - 25;
        int src_x = (x + shift + width) % width;
        int src_idx = y * step + src_x * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
    }
}
```

Filter #690 - DebrisField

Debris field particles-chunks of pixel data scatter outward from random points like exploding debris.

Technique: Outward-scattering pixel chunk particles.

```
// Case 690 in unifiedFilterKernel:
processDebrisField(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDebrisField(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 10, y / 10, params.seed);
    if (r < 0.15f) {
```

```

    int ox = (int)((r - 0.075f) * 20 + params.frame_count * 0.5f) % 20 - 10;
    int oy = (int)((r - 0.075f) * 20) % 20 - 10;
    int src_x = (x + ox + width) % width;
    int src_y = (y + oy + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
}

```

Filter #691 - DeepFry

Deep fried meme aesthetic-extreme saturation boost, heavy sharpening, and JPEG-like compression artifacts for the "deep fried" meme look.

Technique: Extreme saturation + sharpening + artifact.

```

// Case 691 in unifiedFilterKernel:
processDeepFry(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDeepFry(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float intensity = 1.5f + 0.5f * sinf(params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j) {
        float val = (data[idx + j] - 128) * intensity + 128;
        val = fmaxf(0.0f, fminf(255.0f, val));
        if (val > 200)
            val = 255;
        else if (val < 55)
            val = 0;
        data[idx + j] = (unsigned char)val;
    }
}

```

Filter #692 - DesyncRGB

RGB desynchronization-each color channel is offset in a different direction and by different amounts, creating severe chromatic desync.

Technique: Multi-directional per-channel offset desync.

```

// Case 692 in unifiedFilterKernel:
processDesyncRGB(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDesyncRGB(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int r_off_x = (params.frame_count % 10) - 5;
    int g_off_y = (params.frame_count % 8) - 4;
    int b_off_x = -((params.frame_count % 10) - 5);
    int r_x = (x + r_off_x + width) % width;
    int g_y = (y + g_off_y + height) % height;
    int b_x = (x + b_off_x + width) % width;
    data[idx + 2] = data[y * step + r_x * 4 + 2];
}

```

```

    data[idx + 1] = data[g_y * step + x * 4 + 1];
    data[idx] = data[y * step + b_x * 4];
}

```

Filter #693 - DiagonalWipe

Diagonal wipe transition-a diagonal line sweeps across the frame, transitioning between current and history frames.

Technique: Diagonal line wipe transition.

```

// Case 693 in unifiedFilterKernel:
processDiagonalWipe(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDiagonalWipe(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int diag = x + y;
    int wipe = (params.frame_count * 5) % (width + height);
    if (diag < wipe) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = 255 - data[idx + j];
    }
}

```

Filter #694 - DigitalArtifact

Digital artifact blocks-randomly places colored artifact rectangles (macroblocks) across the frame, simulating digital compression artifacts.

Technique: Random colored macroblock artifact placement.

```

// Case 694 in unifiedFilterKernel:
processDigitalArtifact(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDigitalArtifact(int x, int y, unsigned char *data, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int block = (x / 8) + (y / 8) * 100;
    float r = gpu_rand(block, params.seed, params.frame_count / 10);
    if (r < 0.02f) {
        unsigned char artifact = (unsigned char)((int)(r * 12750) % 256);
        for (int j = 0; j < 3; ++j)
            data[idx + j] = artifact;
    }
}

```

Filter #695 - DimensionRift

Dimension rift-a tear/rift appears in the image showing a warped/inverted version behind it, like a tear in space-time.

Technique: Spatial tear with warped/inverted inner reveal.


```
// Case 695 in unifiedFilterKernel:
processDimensionRift(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processDimensionRift(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float rift = sinf(dist * 0.03f - params.frame_count * 0.1f);
    if (rift > 0.7f) {
        int src_x = (width - 1 - x);
        int src_y = (height - 1 - y);
        int src_idx = src_y * step + src_x * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (data[idx + j] + data[src_idx + j]) / 2;
    }
}
```

Filter #696 - DotCrawl

Dot crawl artifact-simulates the NTSC dot-crawl artifact where fine patterns create crawling dot interference along edges.

Technique: Edge-following crawling dot pattern overlay.

```
// Case 696 in unifiedFilterKernel:
processDotCrawl(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDotCrawl(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int crawl = ((x + y + params.frame_count) % 4);
    if (crawl == 0) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.8f);
    }
}
```

Filter #697 - DualTone

Dual-tone color grading-maps shadows to one color and highlights to another (duotone), creating a two-color stylized look.

Technique: Shadow/highlight dual-color mapping.

```
// Case 697 in unifiedFilterKernel:
processDualTone(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processDualTone(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int gray = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
    int phase = (params.frame_count / 30) % 3;
    if (gray < 128) {
```

```

        if (phase == 0) {
            data[idx] = gray;
            data[idx + 1] = 0;
            data[idx + 2] = gray;
        } else if (phase == 1) {
            data[idx] = 0;
            data[idx + 1] = gray;
            data[idx + 2] = gray;
        } else {
            data[idx] = gray;
            data[idx + 1] = gray;
            data[idx + 2] = 0;
        }
    } else {
        if (phase == 0) {
            data[idx] = 255 - gray;
            data[idx + 1] = 255;
            data[idx + 2] = 255 - gray;
        } else if (phase == 1) {
            data[idx] = 255;
            data[idx + 1] = 255 - gray;
            data[idx + 2] = 255 - gray;
        } else {
            data[idx] = 255 - gray;
            data[idx + 1] = 255 - gray;
            data[idx + 2] = 255;
        }
    }
}

```

Filter #698 - EchoFade

Echo fade-multiple temporal echo layers fade progressively to black, creating a dark-fading trail effect.

Technique: Multi-layer fade-to-black echo.

```

// Case 698 in unifiedFilterKernel:
processEchoFade(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processEchoFade(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float fade = 0.9f - (params.frame_count % 30) * 0.02f;
    unsigned char *prev = allFrames[3 % params.numFrames];
    if (prev) {
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - fade) + prev[idx + j] * fade);
    }
}

```

Filter #699 - EdgeMelt

Edge melt-detected edges blur and spread outward over time, as if the edges are melting and bleeding.

Technique: Edge-detected blur spreading.

```
// Case 699 in unifiedFilterKernel:
processEdgeMelt(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processEdgeMelt(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float melt = sinf(x * 0.02f + params.frame_count * 0.05f) * 10.0f;
    int src_y = ((int)(y + melt) % height + height) % height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Visual FX C (700-735)

Filter #700 - ElasticWarp

Elastic rubber-sheet warp-pixels are displaced as if the image is printed on a rubber sheet being stretched and pulled with animated anchor points.

Technique: Animated multi-point elastic displacement.

```
// Case 700 in unifiedFilterKernel:
processElasticWarp(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processElasticWarp(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float elastic = sinf(dist * 0.05f - params.frame_count * 0.08f) * 0.15f;
    int src_x = (int)(x + dx * elastic) % width;
    if (src_x < 0)
        src_x += width;
    int src_y = (int)(y + dy * elastic) % height;
    if (src_y < 0)
        src_y += height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #701 - EmberGlow

Glowing embers effect-bright orange/red points pulse and fade like glowing embers scattered across the frame.

Technique: Pulsing warm-colored point glow overlay.

```
// Case 701 in unifiedFilterKernel:
processEmberGlow(x, y, data, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processEmberGlow(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float glow = 0.5f + 0.5f * sinf(params.frame_count * 0.1f + x * 0.01f);
    data[idx] = (unsigned char)fminf(255.0f, data[idx] * 0.8f);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * (0.5f + glow * 0.3f));
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * (0.8f + glow * 0.4f));
}

```

Filter #702 - EntropyShift

Entropy-based color shift-regions with high local variance (detail/texture) get color-shifted more than smooth regions.

Technique: Local-variance-proportional color shifting.

```

// Case 702 in unifiedFilterKernel:
processEntropyShift(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processEntropyShift(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float entropy = gpu_rand(x, y, params.seed + params.frame_count);
    int shift = (int)(entropy * 50.0f) - 25;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, fmaxf(0.0f, data[idx + j] + shift * (j == params.
            ↪ frame_count % 3 ? 1.0f : 0.3f)));
}

```

Filter #703 - ErosionBlend

Morphological erosion blended with original-shrinks bright features to their minimum neighbor value, blended back for a rough-textured look.

Technique: Minimum-neighbor erosion blend.

```

// Case 703 in unifiedFilterKernel:
processErosionBlend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processErosionBlend(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    if (x <= 1 || x >= width - 2 || y <= 1 || y >= height - 2)
        return;
    int idx = y * step + x * 4;
    unsigned char minVal[3] = {255, 255, 255};
    for (int dy = -1; dy <= 1; ++dy) {
        for (int dx = -1; dx <= 1; ++dx) {
            int n_idx = (y + dy) * step + (x + dx) * 4;
            for (int j = 0; j < 3; ++j)
                if (data[n_idx + j] < minVal[j])
                    minVal[j] = data[n_idx + j];
        }
    }
    float blend = 0.5f + 0.3f * sinf(params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j)

```

```

        data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - blend) + minVal[j] * blend);
    }

```

Filter #704 - ExplosionBurst

Explosion burst-pixels fly radially outward from a center point in an animated burst, creating an explosion-scatter effect.

Technique: Radial outward pixel scatter burst.

```

// Case 704 in unifiedFilterKernel:
processExplosionBurst(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processExplosionBurst(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float burst = fmaxf(0.0f, 1.0f - dist / (width * 0.5f));
    float pulse = 0.5f + 0.5f * sinf(params.frame_count * 0.15f - dist * 0.02f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * (1.0f + burst * pulse * 0.5f));
}

```

Filter #705 - FacetMirror

Faceted mirror-divides the image into irregularly-shaped facets that each reflect/mirror a slightly different region, like a gem-cut mirror.

Technique: Irregular multi-facet region mirror.

```

// Case 705 in unifiedFilterKernel:
processFacetMirror(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFacetMirror(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int facet_size = 40 + (params.frame_count % 20);
    int fx = (x / facet_size) % 2;
    int fy = (y / facet_size) % 2;
    int src_x = fx ? (facet_size - 1 - (x % facet_size)) + (x / facet_size) * facet_size : x;
    int src_y = fy ? (facet_size - 1 - (y % facet_size)) + (y / facet_size) * facet_size : y;
    src_x = (src_x % width + width) % width;
    src_y = (src_y % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #706 - FadeStreak

Fade streak-directional fading streaks trail behind bright objects, creating comet-like tail trails that fade to black.

Technique: Directional brightness trailing fade streaks.

```
// Case 706 in unifiedFilterKernel:
processFadeStreak(x, y, data, width, step, params);

// Referenced __device__ function(s):
__device__ void processFadeStreak(int x, int y, unsigned char *data, int width, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float streak = (float)(x + params.frame_count * 3) / width;
    streak = fmodf(streak, 1.0f);
    float fade = 1.0f - streak * 0.5f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * fade);
}
```

Filter #707 - FeatherEdge

Feathered edge vignette-soft, feathered edges gradually blend to black around the frame border with animated feather width.

Technique: Animated soft-edge border feathering.

```
// Case 707 in unifiedFilterKernel:
processFeatherEdge(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFeatherEdge(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float edge_x = fminf((float)x / 50.0f, (float)(width - x) / 50.0f);
    float edge_y = fminf((float)y / 50.0f, (float)(height - y) / 50.0f);
    float feather = fminf(1.0f, fminf(edge_x, edge_y));
    float pulse = 0.8f + 0.2f * sinf(params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * feather * pulse);
}
```

Filter #708 - FlashFreeze

Flash freeze-periodically freezes the entire frame for several frames then resumes, simulating a freeze-frame strobe.

Technique: Periodic multi-frame freeze.

```
// Case 708 in unifiedFilterKernel:
processFlashFreeze(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFlashFreeze(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    int flash = (params.frame_count / 15) % 4;
```

```

    if (flash == 0) {
        int avg = (data[idx] + data[idx + 1] + data[idx + 2]) / 3;
        data[idx] = (unsigned char)fminf(255.0f, avg + 50);
        data[idx + 1] = (unsigned char)fminf(255.0f, avg + 60);
        data[idx + 2] = (unsigned char)fminf(255.0f, avg + 40);
    }
}

```

Filter #709 - FlipMirror

Animated flip mirror-alternates between horizontal and vertical flipping with blend transitions between orientations.

Technique: Alternating H/V flip with blend transition.

```

// Case 709 in unifiedFilterKernel:
processFlipMirror(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFlipMirror(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int phase = (params.frame_count / 20) % 4;
    int src_x = x, src_y = y;
    if (phase == 1)
        src_x = width - 1 - x;
    else if (phase == 2)
        src_y = height - 1 - y;
    else if (phase == 3) {
        src_x = width - 1 - x;
        src_y = height - 1 - y;
    }
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (data[idx + j] + data[src_idx + j]) / 2;
}

```

Filter #710 - FloatDrift

Floating drift-the entire image gently drifts/floats in X and Y with slow sine movement, creating a floating/bobbing effect.

Technique: Slow sine whole-frame position drift.

```

// Case 710 in unifiedFilterKernel:
processFloatDrift(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFloatDrift(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float drift_x = sinf(y * 0.02f + params.frame_count * 0.03f) * 8.0f;
    float drift_y = cosf(x * 0.02f + params.frame_count * 0.04f) * 8.0f;
    int src_x = ((int)(x + drift_x) % width + width) % width;
    int src_y = ((int)(y + drift_y) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)

```

```

        data[idx + j] = data[src_idx + j];
    }

```

Filter #711 - FlowField

Flow field visualization-renders pixel displacement guided by a procedural flow field, creating flowing organic distortion paths.

Technique: Procedural flow-field pixel displacement.

```

// Case 711 in unifiedFilterKernel:
processFlowField(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFlowField(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float angle = sinf(x * 0.01f) * cosf(y * 0.01f) * 3.14159f + params.frame_count * 0.02f;
    float flow = 5.0f;
    int src_x = ((int)(x + cosf(angle) * flow) % width + width) % width;
    int src_y = ((int)(y + sinf(angle) * flow) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (data[idx + j] + data[src_idx + j]) / 2;
}

```

Filter #712 - FoldWarp

Paper fold warp-simulates folding the image along animated fold lines, with perspective foreshortening on each fold.

Technique: Fold-line-based perspective warp.

```

// Case 712 in unifiedFilterKernel:
processFoldWarp(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFoldWarp(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int fold_x = (x < width / 2) ? x * 2 : (width - 1 - x) * 2;
    int fold_y = (y < height / 2) ? y * 2 : (height - 1 - y) * 2;
    fold_x = fold_x % width;
    fold_y = fold_y % height;
    int src_idx = fold_y * step + fold_x * 4;
    float blend = 0.5f + 0.3f * sinf(params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - blend) + data[src_idx + j] * blend);
}

```

Filter #713 - FragmentScatter

Fragment scatter-the image is broken into small fragments that scatter outward from their original positions.

Technique: Small fragment random scatter displacement.

```
// Case 713 in unifiedFilterKernel:
processFragmentScatter(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processFragmentScatter(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 8, y / 8, params.seed);
    int scatter_x = (int)(r * 16.0f) - 8;
    int scatter_y = (int)(gpu_rand(y / 8, x / 8, params.seed) * 16.0f) - 8;
    int src_x = ((x + scatter_x) % width + width) % width;
    int src_y = ((y + scatter_y) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #714 - FrequencyPulse

Frequency pulse-different spatial frequencies (detail levels) pulse at different rates, creating a shimmering multi-frequency effect.

Technique: Multi-frequency brightness pulsing.

```
// Case 714 in unifiedFilterKernel:
processFrequencyPulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFrequencyPulse(int x, int y, unsigned char *data, size_t step, const FilterParams
    ↪ &params) {
    int idx = y * step + x * 4;
    float freq1 = sinf(x * 0.05f + params.frame_count * 0.1f);
    float freq2 = sinf(y * 0.07f + params.frame_count * 0.08f);
    float pulse = 0.7f + 0.3f * freq1 * freq2;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * pulse);
}
```

Filter #715 - FrostBite

Frost/ice effect-adds a white crystalline frost overlay that spreads from edges inward, with animated frost-creep.

Technique: Edge-inward spreading white frost overlay.

```
// Case 715 in unifiedFilterKernel:
processFrostBite(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processFrostBite(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float frost = gpu_rand(x, y, params.seed + params.frame_count / 5);
    if (frost > 0.9f) {
        data[idx] = (unsigned char)fminf(255.0f, data[idx] + 80);
    }
}
```

```

        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + 90);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + 100);
    } else {
        float tint = 0.9f + frost * 0.1f;
        data[idx + 2] = (unsigned char)(data[idx + 2] * tint);
    }
}

```

Filter #716 - FuseBlend

Fuse blend-blends frames together with a "fuse/ignite" pattern that sweeps across the frame like a burning fuse.

Technique: Sweeping fuse-line blend transition.

```

// Case 716 in unifiedFilterKernel:
processFuseBlend(x, y, data, allFrames, step, params);

// Referenced __device__ function(s):
__device__ void processFuseBlend(int x, int y, unsigned char *data, unsigned char **allFrames, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    unsigned char *f1 = allFrames[2 % params.numFrames];
    unsigned char *f2 = allFrames[5 % params.numFrames];
    if (f1 && f2) {
        float fuse = 0.5f + 0.3f * sinf(params.frame_count * 0.05f);
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.4f + f1[idx + j] * fuse * 0.3f + f2[idx +
    ↪ j] * (1.0f - fuse) * 0.3f);
    }
}

```

Filter #717 - GalaxySpiral

Galaxy spiral distortion-pixels are displaced in a logarithmic spiral pattern from center, creating a spiral galaxy arm structure.

Technique: Logarithmic spiral displacement from center.

```

// Case 717 in unifiedFilterKernel:
processGalaxySpiral(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGalaxySpiral(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float angle = atan2f(dy, dx) + params.frame_count * 0.02f;
    float dist = sqrtf(dx * dx + dy * dy);
    float spiral = sinf(angle * 3.0f - dist * 0.03f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * (0.7f + 0.3f * spiral));
}

```

Filter #718 - GelWobble

Gel wobble-the image wobbles as if made of soft gel/jelly, with squishy multi-frequency wobble animation.

Technique: Multi-frequency soft wobble displacement.

```
// Case 718 in unifiedFilterKernel:
processGelWobble(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGelWobble(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float wobble_x = sinf(y * 0.03f + params.frame_count * 0.08f) * 6.0f;
    float wobble_y = sinf(x * 0.03f + params.frame_count * 0.06f) * 6.0f;
    int src_x = ((int)(x + wobble_x) % width + width) % width;
    int src_y = ((int)(y + wobble_y) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (data[idx + j] + data[src_idx + j]) / 2;
}
```

Filter #719 - GhostEcho

Ghost echo-multiple ghost copies of the image at different positions and opacities, like looking through haunted glass.

Technique: Multi-position offset ghost overlay.

```
// Case 719 in unifiedFilterKernel:
processGhostEcho(x, y, data, allFrames, width, step, params);

// Referenced __device__ function(s):
__device__ void processGhostEcho(int x, int y, unsigned char *data, unsigned char **allFrames, int width
    ↪ , size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset = 20 + (int)(sinf(params.frame_count * 0.05f) * 10.0f);
    int ghost_x = (x + offset) % width;
    unsigned char *prev = allFrames[4 % params.numFrames];
    if (prev) {
        int ghost_idx = y * step + ghost_x * 4;
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.7f + prev[ghost_idx + j] * 0.3f);
    }
}
```

Filter #720 - GlassShatter

Glass shatter animation-the image cracks into glass-like shards that separate and fall over time.

Technique: Irregular shard crack + separation animation.

```
// Case 720 in unifiedFilterKernel:
processGlassShatter(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void processGlassShatter(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int shard_size = 25 + (params.frame_count % 15);
    int sx = x / shard_size, sy = y / shard_size;
    float r = gpu_rand(sx, sy, params.seed);
    int offset_x = (int)((r - 0.5f) * shard_size * 0.5f);
    int offset_y = (int)((gpu_rand(sy, sx, params.seed) - 0.5f) * shard_size * 0.5f);
    int src_x = ((x + offset_x) % width + width) % width;
    int src_y = ((y + offset_y) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}

```

Filter #721 - GlimmerPulse

Glimmer pulse-scattered individual pixels glimmer/sparkle with bright flashes, creating a twinkling sparkle overlay.

Technique: Random sparse bright pixel sparkle flash.

```

// Case 721 in unifiedFilterKernel:
processGlimmerPulse(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGlimmerPulse(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float glimmer = gpu_rand(x, y, params.seed + params.frame_count / 3);
    float pulse = 0.8f + 0.4f * sinf(params.frame_count * 0.15f + glimmer * 6.28f);
    if (glimmer > 0.95f)
        pulse = 1.5f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * pulse);
}

```

Filter #722 - GlitchMosaic

Glitched mosaic-mosaic tiles with random glitchy offsets and color corruption within each tile.

Technique: Mosaic tiling + per-tile glitch corruption.

```

// Case 722 in unifiedFilterKernel:
processGlitchMosaic(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGlitchMosaic(int x, int y, unsigned char *data, int width, int height, size_t
    ↪ step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int tile = 16 + (params.frame_count % 16);
    int tx = (x / tile) * tile, ty = (y / tile) * tile;
    float r = gpu_rand(tx, ty, params.seed + params.frame_count / 10);
    if (r > 0.7f) {
        int src_x = (tx + (int)(r * tile * 3)) % width;
        int src_y = (ty + (int)(gpu_rand(ty, tx, params.seed) * tile * 3)) % height;
        int src_idx = src_y * step + src_x * 4;
    }
}

```

```

        for (int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
    }
}

```

Filter #723 - GlowEdge

Glowing edge detection-edges are detected and rendered with a bright, soft glow around them.

Technique: Edge detection + soft glow halo.

```

// Case 723 in unifiedFilterKernel:
processGlowEdge(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGlowEdge(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    if (x <= 1 || x >= width - 2 || y <= 1 || y >= height - 2)
        return;
    int idx = y * step + x * 4;
    float gx = 0, gy = 0;
    for (int dy = -1; dy <= 1; ++dy) {
        for (int dx = -1; dx <= 1; ++dx) {
            int n_idx = (y + dy) * step + (x + dx) * 4;
            float val = (data[n_idx] + data[n_idx + 1] + data[n_idx + 2]) / 3.0f;
            gx += val * dx;
            gy += val * dy;
        }
    }
    float edge = sqrtf(gx * gx + gy * gy) / 255.0f;
    float glow = 1.0f + edge * sinf(params.frame_count * 0.1f) * 0.5f;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, data[idx + j] * glow);
}

```

Filter #724 - GradientMelt

Gradient melt-pixels melt/flow downward at rates proportional to their brightness gradient, creating a waterfall-like flow.

Technique: Brightness-gradient-proportional downward flow.

```

// Case 724 in unifiedFilterKernel:
processGradientMelt(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processGradientMelt(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float melt = (float)y / height + sinf(x * 0.02f + params.frame_count * 0.05f) * 0.1f;
    float fade = fmaxf(0.3f, 1.0f - melt * 0.7f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * fade);
}

```

Filter #725 - GrainStorm

Heavy grain storm-intense film grain/noise that storms across the frame much more aggressively than normal film grain.

Technique: Heavy per-pixel random noise storm.

```
// Case 725 in unifiedFilterKernel:
processGrainStorm(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processGrainStorm(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float grain = gpu_rand(x, y, params.seed + params.frame_count) * 2.0f - 1.0f;
    float intensity = 30.0f + 20.0f * sinf(params.frame_count * 0.1f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)fminf(255.0f, fmaxf(0.0f, data[idx + j] + grain * intensity));
}
```

Filter #726 - GravityPull

Gravity pull-pixels are displaced downward by an amount proportional to their distance from top, simulating gravitational pull/sag.

Technique: Top-distance-proportional downward displacement.

```
// Case 726 in unifiedFilterKernel:
processGravityPull(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processGravityPull(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy) + 1.0f;
    float pull = 50.0f / dist * sinf(params.frame_count * 0.05f);
    int src_x = (int)(x - dx * pull / dist) % width;
    if (src_x < 0)
        src_x += width;
    int src_y = (int)(y - dy * pull / dist) % height;
    if (src_y < 0)
        src_y += height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #727 - GridWarp

Grid warp-divides the image into a grid and warps each grid cell independently, creating a patchwork distortion.

Technique: Per-grid-cell independent warp.

```
// Case 727 in unifiedFilterKernel:
processGridWarp(x, y, data, width, height, step, params);
```

```
// Referenced __device__ function(s):
__device__ void processGridWarp(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    int grid = 32;
    float warp_x = sinf((x / grid) * 0.5f + params.frame_count * 0.05f) * 8.0f;
    float warp_y = cosf((y / grid) * 0.5f + params.frame_count * 0.04f) * 8.0f;
    int src_x = ((int)(x + warp_x) % width + width) % width;
    int src_y = ((int)(y + warp_y) % height + height) % height;
    int src_idx = src_y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #728 - HazeLayer

Haze layer overlay-adds a semi-transparent uniform haze/fog layer over the image with animated density.

Technique: Animated semi-transparent haze overlay.

```
// Case 728 in unifiedFilterKernel:
processHazeLayer(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processHazeLayer(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    float haze = 0.3f + 0.2f * sinf(params.frame_count * 0.03f);
    float depth = (float)y / height;
    float fog = haze * depth;
    unsigned char fog_color = 180 + (unsigned char)(30 * sinf(params.frame_count * 0.05f));
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - fog) + fog_color * fog);
}
```

Filter #729 - HeatRipple

Heat ripple distortion-stronger sine-wave horizontal displacement concentrated toward the bottom, creating aggressive heat-haze ripples.

Technique: Bottom-concentrated horizontal heat ripple.

```
// Case 729 in unifiedFilterKernel:
processHeatRipple(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processHeatRipple(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
    int idx = y * step + x * 4;
    float ripple = sinf(y * 0.05f + params.frame_count * 0.15f) * (3.0f + 2.0f * sinf(params.frame_count
    ↪ * 0.02f));
    int src_x = ((int)(x + ripple) % width + width) % width;
    int src_idx = y * step + src_x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

```

    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * 1.1f);
}

```

Filter #730 - HexagonBlur

Hexagonal blur-applies blur in a hexagonal kernel shape rather than circular, creating a distinctive hex-shaped bokeh.

Technique: Hexagonal kernel blur.

```

// Case 730 in unifiedFilterKernel:
processHexagonBlur(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processHexagonBlur(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    int hex_size = 20 + (params.frame_count % 10);
    int hx = x / hex_size, hy = y / hex_size;
    int cx = hx * hex_size + hex_size / 2;
    int cy = hy * hex_size + hex_size / 2;
    cx = (cx % width + width) % width;
    cy = (cy % height + height) % height;
    int src_idx = cy * step + cx * 4;
    float blend = 0.6f + 0.2f * sinf(params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * (1.0f - blend) + data[src_idx + j] * blend);
}

```

Filter #731 - HighContrast

Animated high contrast-increases contrast with an animated curve that oscillates between moderate and extreme contrast.

Technique: Oscillating contrast curve amplification.

```

// Case 731 in unifiedFilterKernel:
processHighContrast(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processHighContrast(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float contrast = 1.5f + 0.5f * sinf(params.frame_count * 0.05f);
    for (int j = 0; j < 3; ++j) {
        float val = (data[idx + j] - 128) * contrast + 128;
        data[idx + j] = (unsigned char)fminf(255.0f, fmaxf(0.0f, val));
    }
}

```

Filter #732 - HologramScan

Hologram scan effect-scrolling horizontal bright bands with blue/cyan tinting, simulating a sci-fi holographic display scanning.

Technique: Scrolling cyan-tinted scan band overlay.

```
// Case 732 in unifiedFilterKernel:
processHologramScan(x, y, data, height, step, params);

// Referenced __device__ function(s):
__device__ void processHologramScan(int x, int y, unsigned char *data, int height, size_t step, const
    ↪ FilterParams &params) {
    int idx = y * step + x * 4;
    int scanline = (params.frame_count * 3) % height;
    float dist = fabsf(y - scanline);
    if (dist < 30) {
        float glow = 1.0f - dist / 30.0f;
        data[idx] = (unsigned char)fminf(255.0f, data[idx] + glow * 100);
        data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] + glow * 150);
        data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] + glow * 80);
    }
    if (y % 3 == 0)
        for (int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.8f);
}
```

Filter #733 - HorizonBend

Horizon bend-bends the horizon line of the image into a curve, creating a fisheye-like horizon warp.

Technique: Horizontal horizon-line curve distortion.

```
// Case 733 in unifiedFilterKernel:
processHorizonBend(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processHorizonBend(int x, int y, unsigned char *data, int width, int height, size_t step
    ↪ , const FilterParams &params) {
    int idx = y * step + x * 4;
    float bend = sinf(x * 0.01f + params.frame_count * 0.03f) * 30.0f;
    int src_y = (int)(y + bend * (1.0f - (float)y / height)) % height;
    if (src_y < 0)
        src_y += height;
    int src_idx = src_y * step + x * 4;
    for (int j = 0; j < 3; ++j)
        data[idx + j] = data[src_idx + j];
}
```

Filter #734 - HotSpot

Spotlight hot spot-a bright circular hot spot moves around the frame, brightening the area it covers.

Technique: Moving circular brightness spotlight.

```
// Case 734 in unifiedFilterKernel:
processHotSpot(x, y, data, width, height, step, params);

// Referenced __device__ function(s):
__device__ void processHotSpot(int x, int y, unsigned char *data, int width, int height, size_t step,
    ↪ const FilterParams &params) {
```

```

    int idx = y * step + x * 4;
    float cx = width / 2.0f + sinf(params.frame_count * 0.02f) * width * 0.3f;
    float cy = height / 2.0f + cosf(params.frame_count * 0.03f) * height * 0.3f;
    float dist = sqrtf((x - cx) * (x - cx) + (y - cy) * (y - cy));
    float hot = expf(-dist / 80.0f);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * (1.0f + hot * 0.8f));
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * (1.0f + hot * 0.4f));
}

```

Filter #735 - HueWobble

Hue wobble-the hue of the image oscillates/wobbles back and forth around its original value with sine modulation.

Technique: Sine-oscillated hue shift wobble.

```

// Case 735 in unifiedFilterKernel:
processHueWobble(x, y, data, step, params);

// Referenced __device__ function(s):
__device__ void processHueWobble(int x, int y, unsigned char *data, size_t step, const FilterParams &
    ↪ params) {
    int idx = y * step + x * 4;
    float wobble = sinf(x * 0.02f + y * 0.02f + params.frame_count * 0.1f) * 30.0f;
    int shift = (int)wobble;
    unsigned char temp[3] = {data[idx], data[idx + 1], data[idx + 2]};
    int ch = (params.frame_count / 10) % 3;
    data[idx + ch] = (unsigned char)fminf(255.0f, fmaxf(0.0f, temp[ch] + shift));
    data[idx + (ch + 1) % 3] = (unsigned char)fminf(255.0f, fmaxf(0.0f, temp[(ch + 1) % 3] - shift / 2))
    ↪ ;
}

```

Glitch Series (736-857)

Filter #736 - glitch_alpha_diamond1

Alpha diamond pattern glitch variant 1-displaces pixels in a diamond-shaped pattern with alpha blending from history frames.

Technique: Diamond-pattern displacement + alpha blend.

```

// Case 736 in unifiedFilterKernel:
process_glitch_AlphaDiamond1(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_AlphaDiamond1(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int index = ((params.frame_count % width) + width) % width;
    int offset = gpu_triangle_wave(params.frame_count, 7) + 1;
    int start_x = ((index + y) % width + width) % width;
    int src_x = ((start_x + x) % width + width) % width;
    unsigned char *ref = allFrames[offset % params.numFrames];
    if(ref) {
        int src_idx = y * step + src_x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[src_idx + j]);
    }
}

```

```
}
```

Filter #737 - glitch_alpha_diamond2

Alpha diamond variant 2-larger diamond blocks with different blending intensity than variant 1.

Technique: Larger diamond-pattern alpha blend.

```
// Case 737 in unifiedFilterKernel:
process_glitch_AlphaDiamond2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_AlphaDiamond2(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int index = ((params.frame_count * 3) % width + width) % width;
    int offset = gpu_triangle_wave(params.frame_count, 7) + 1;
    int src_x = ((index + y + x) % width + width) % width;
    unsigned char *ref = allFrames[offset % params.numFrames];
    if(ref) {
        int src_idx = y * step + src_x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[src_idx + j]);
    }
}
```

Filter #738 - glitch_alpha_diamond3

Alpha diamond variant 3-animated diamond size with increased displacement.

Technique: Animated-size diamond displacement blend.

```
// Case 738 in unifiedFilterKernel:
process_glitch_AlphaDiamond3(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_AlphaDiamond3(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 0.1f + 0.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.05f));
    int index = ((params.frame_count * 2) % width + width) % width;
    int offset = gpu_triangle_wave(params.frame_count, 7) + 1;
    int src_x = ((index + y + x) % width + width) % width;
    unsigned char *ref = allFrames[offset % params.numFrames];
    if(ref) {
        int src_idx = y * step + src_x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(alpha * data[idx + j] + (1.0f - alpha) * ref[src_idx + j]);
    }
}
```

Filter #739 - glitch_alpha_diamond4

Alpha diamond variant 4-combines diamond pattern with XOR blending for digital artifacts.

Technique: Diamond displacement + XOR blend.

```
// Case 739 in unifiedFilterKernel:
process_glitch_AlphaDiamond4(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_AlphaDiamond4(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 0.1f + 0.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.05f));
    int index = ((params.frame_count * 2) % width + width) % width;
    int offset = gpu_triangle_wave(params.frame_count, 7) + 1;
    int src_x = ((index + y + x) % width + width) % width;
    int fsel = (int)(gpu_rand(x, y, params.seed) * 9.0f) % params.numFrames;
    unsigned char *ref = allFrames[fsel];
    if(ref) {
        int src_idx = y * step + src_x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(alpha * data[idx + j] + (1.0f - alpha) * ref[src_idx + j]);
    }
}
```

Filter #740 - glitch_alpha_diamond5

Alpha diamond variant 5-maximum-intensity diamond pattern with aggressive displacement and heavy blending.

Technique: Aggressive diamond displacement + heavy blend.

```
// Case 740 in unifiedFilterKernel:
process_glitch_AlphaDiamond5(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_AlphaDiamond5(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 0.1f + 0.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.03f));
    int index = ((params.frame_count * 3) % width + width) % width;
    int src_x = ((index + y + x) % width + width) % width;
    int fsel = (int)(gpu_rand(x, y, params.seed + params.frame_count) * 9.0f) % params.numFrames;
    unsigned char *ref = allFrames[fsel];
    if(ref) {
        int src_idx = y * step + src_x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(alpha * data[idx + j] + (1.0f - alpha) * ref[src_idx + j]);
    }
}
```

Filter #741 - glitch_alphatrails

Alpha blended trails from history frames-smooth, translucent motion trails using alpha compositing for clean trailing.

Technique: Alpha-composited temporal trailing.

```
// Case 741 in unifiedFilterKernel:
process_glitch_Alphatrails(x, y, data, allFrames, width, height, step, params);
```

```

// Referenced __device__ function(s):
__device__ void process_glitch_Alphatrails(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 0.1f + 0.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.05f));
    int f_indices[3] = {1 % params.numFrames, (params.numFrames / 2) % params.numFrames, (params.
    ↪ numFrames - 1) % params.numFrames};
    for(int j = 0; j < 3; ++j) {
        unsigned char *ref = allFrames[f_indices[j]];
        if(ref) {
            data[idx + j] = gpu_wrap_cast(alpha * data[idx + j] + (1.0f - alpha) * ref[idx]);
        }
    }
}

```

Filter #742 - glitch_color_fade_ex1

Extended color fade glitch 1-colors fade through specific transitions with glitch interruptions and jumps.

Technique: Glitch-interrupted color fade transition v1.

```

// Case 742 in unifiedFilterKernel:
process_glitch_ColorFadeEx1(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_ColorFadeEx1(int x, int y, unsigned char *data, unsigned char **allFrames
    ↪ , int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float color_val = fmodf((float)(y * width + x + params.frame_count * 7 + j * 83), 254.0f);
        data[idx + j] = gpu_wrap_cast(data[idx + j] + color_val);
    }
}

```

Filter #743 - glitch_color_fade_ex2

Extended color fade glitch 2-alternative color fade path with different transition timing and more aggressive interruption.

Technique: Glitch-interrupted color fade transition v2.

```

// Case 743 in unifiedFilterKernel:
process_glitch_ColorFadeEx2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_ColorFadeEx2(int x, int y, unsigned char *data, unsigned char **allFrames
    ↪ , int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float color0 = fmodf(params.frame_count * 13.0f + 50.0f, 254.0f);
    float color1 = fmodf(params.frame_count * 17.0f + 100.0f, 254.0f);
    float color2 = fmodf(params.frame_count * 23.0f + 150.0f, 254.0f);
    data[idx] = gpu_wrap_cast(data[idx] + color0);
    data[idx + 1] = gpu_wrap_cast(data[idx + 1] + color1);
    data[idx + 2] = gpu_wrap_cast(data[idx + 2] + color2);
}

```

Filter #744 - glitch_color_shift

Global color channel shifting glitch-shifts all color channels by random offsets simultaneously, creating sudden color jumps.

Technique: Random global channel value shift.

```
// Case 744 in unifiedFilterKernel:
process_glitch_ColorShift(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_ColorShift(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset_r = ((x + params.seed) % width + width) % width;
    int offset_g = ((x + params.seed + width / 3) % width + width) % width;
    int offset_b = ((x + params.seed + 2 * width / 3) % width + width) % width;
    int f1 = 1 % params.numFrames;
    int f2 = (params.numFrames / 2) % params.numFrames;
    int f3 = (params.numFrames - 1) % params.numFrames;
    unsigned char *r1 = allFrames[f1];
    unsigned char *r2 = allFrames[f2];
    unsigned char *r3 = allFrames[f3];
    bool strobe = (params.frame_count % 2) == 0;
    if(r1 && r2 && r3) {
        if(strobe) {
            data[idx] = r1[y * step + offset_r * 4];
            data[idx + 1] = r2[y * step + offset_g * 4 + 1];
            data[idx + 2] = r3[y * step + offset_b * 4 + 2];
        } else {
            data[idx] = r1[y * step + offset_b * 4];
            data[idx + 1] = r2[y * step + offset_r * 4 + 1];
            data[idx + 2] = r3[y * step + offset_g * 4 + 2];
        }
    }
}
```

Filter #745 - glitch_colorxor01

Color XOR glitch variant 01-XORs pixel colors with an animated value pattern. Each variant in the series (01-20) uses different XOR patterns and timing.

Technique: Animated color XOR pattern v01.

```
// Case 745 in unifiedFilterKernel:
process_glitch_Colorxor01(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor01(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.1f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}
```

Filter #746 - glitch_colorxor02

Color XOR glitch variant 02-different XOR bitmask pattern producing alternate digital color artifacts.

Technique: Animated color XOR pattern v02.

```
// Case 746 in unifiedFilterKernel:
process_glitch_Colorxor02(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor02(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.07f + gpu_rand(x,y,params.
    ↪ seed)*0.5f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}
```

Filter #747 - glitch_colorxor03

Color XOR glitch variant 03-XOR with position-dependent masks for spatially-varying artifacts.

Technique: Position-dependent color XOR v03.

```
// Case 747 in unifiedFilterKernel:
process_glitch_Colorxor03(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor03(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.05f + j * 1.5f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}
```

Filter #748 - glitch_colorxor04

Color XOR glitch variant 04-frame-count-dependent XOR patterns for time-varying color distortion.

Technique: Time-varying color XOR v04.

```
// Case 748 in unifiedFilterKernel:
process_glitch_Colorxor04(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor04(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
```

```

        float a = 1.0f + fmodf(params.frame_count * 0.03f + j * 0.7f, 2.0f);
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}

```

Filter #749 - glitch_colorxor05

Color XOR glitch variant 05-combined position and time XOR for complex evolving patterns.

Technique: Position+time combined XOR v05.

```

// Case 749 in unifiedFilterKernel:
process_glitch_Colorxor05(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor05(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int op_type = (params.frame_count + (int)(gpu_rand(x, y, params.seed) * 3.0f)) % 3;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.08f + j * 2.1f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        if(op_type == 0) data[idx + j] = data[idx + j] ^ val;
        else if(op_type == 1) data[idx + j] = data[idx + j] & val;
        else data[idx + j] = data[idx + j] | val;
    }
}

```

Filter #750 - glitch_colorxor06

Color XOR glitch variant 06-per-channel independent XOR masks.

Technique: Per-channel independent XOR v06.

```

// Case 750 in unifiedFilterKernel:
process_glitch_Colorxor06(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor06(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int op_type = (params.frame_count + (int)(gpu_rand(x, y, params.seed) * 3.0f)) % 3;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * cosf(params.frame_count * 0.06f + j * 1.3f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        if(op_type == 0) data[idx + j] = data[idx + j] ^ val;
        else if(op_type == 1) data[idx + j] = data[idx + j] & val;
        else data[idx + j] = data[idx + j] | val;
    }
}

```

Filter #751 - glitch_colorxor07

Color XOR glitch variant 07-XOR with history frame values for temporal XOR artifacts.

Technique: History-frame XOR blend v07.

```
// Case 751 in unifiedFilterKernel:
process_glitch_Colorxor07(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor07(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.04f + j * 0.9f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}
```

Filter #752 - glitch_colorxor08

Color XOR glitch variant 08-high-frequency XOR pattern creating fine digital noise.

Technique: High-frequency XOR noise v08.

```
// Case 752 in unifiedFilterKernel:
process_glitch_Colorxor08(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor08(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.035f + j * 1.1f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}
```

Filter #753 - glitch_colorxor09

Color XOR glitch variant 09-low-frequency XOR pattern creating broad color bands.

Technique: Low-frequency XOR banding v09.

```
// Case 753 in unifiedFilterKernel:
process_glitch_Colorxor09(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor09(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.03f + j * 0.8f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}
```

Filter #754 - glitch_colorxor10

Color XOR glitch variant 10-diagonal XOR pattern creating diagonal digital stripes.

Technique: Diagonal XOR stripe pattern v10.

```
// Case 754 in unifiedFilterKernel:
process_glitch_Colorxor10(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor10(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * cosf(params.frame_count * 0.03f + j * 0.6f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}
```

Filter #755 - glitch_colorxor11

Color XOR glitch variant 11-radial XOR pattern from center.

Technique: Radial XOR pattern v11.

```
// Case 755 in unifiedFilterKernel:
process_glitch_Colorxor11(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor11(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 2.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.025f + j * 1.7f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}
```

Filter #756 - glitch_colorxor12

Color XOR glitch variant 12-block-based XOR with tile boundaries.

Technique: Block-tiled XOR pattern v12.

```
// Case 756 in unifiedFilterKernel:
process_glitch_Colorxor12(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor12(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.0f + 3.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.028f + j * 2.0f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}
```

```
}  
}
```

Filter #757 - glitch_colorxor13

Color XOR glitch variant 13-row-alternating XOR patterns.

Technique: Row-alternating XOR v13.

```
// Case 757 in unifiedFilterKernel:  
process_glitch_Colorxor13(x, y, data, allFrames, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void process_glitch_Colorxor13(int x, int y, unsigned char *data, unsigned char **allFrames,  
    ↪ int width, int height, size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    for(int j = 0; j < 3; ++j) {  
        float a = 1.1f + 1.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.04f + j * 1.2f));  
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);  
        data[idx + j] = data[idx + j] ^ val;  
    }  
}
```

Filter #758 - glitch_colorxor14

Color XOR glitch variant 14-column-alternating XOR patterns.

Technique: Column-alternating XOR v14.

```
// Case 758 in unifiedFilterKernel:  
process_glitch_Colorxor14(x, y, data, allFrames, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void process_glitch_Colorxor14(int x, int y, unsigned char *data, unsigned char **allFrames,  
    ↪ int width, int height, size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    for(int j = 0; j < 3; ++j) {  
        float a = 1.1f + 1.9f * (0.5f + 0.5f * cosf(params.frame_count * 0.035f + j * 1.4f));  
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);  
        data[idx + j] = data[idx + j] ^ val;  
    }  
}
```

Filter #759 - glitch_colorxor15

Color XOR glitch variant 15-spiral XOR pattern from center.

Technique: Spiral XOR pattern v15.

```
// Case 759 in unifiedFilterKernel:  
process_glitch_Colorxor15(x, y, data, allFrames, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void process_glitch_Colorxor15(int x, int y, unsigned char *data, unsigned char **allFrames,  
    ↪ int width, int height, size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;
```

```

    for(int j = 0; j < 3; ++j) {
        float a = 1.1f + 1.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.038f + j * 1.6f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}

```

Filter #760 - glitch_colorxor16

Color XOR glitch variant 16-checkerboard XOR pattern.

Technique: Checkerboard XOR pattern v16.

```

// Case 760 in unifiedFilterKernel:
process_glitch_Colorxor16(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor16(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.1f + 1.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.033f + j * 1.8f + x * 0.001f)
    ↪ );
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}

```

Filter #761 - glitch_colorxor17

Color XOR glitch variant 17-frame-difference-based XOR.

Technique: Frame-difference XOR v17.

```

// Case 761 in unifiedFilterKernel:
process_glitch_Colorxor17(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor17(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.1f + 1.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.042f + j * 2.2f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}

```

Filter #762 - glitch_colorxor18

Color XOR glitch variant 18-brightness-threshold XOR.

Technique: Brightness-threshold XOR v18.

```

// Case 762 in unifiedFilterKernel:

```

```

process_glitch_Colorxor18(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor18(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 1.1f + 1.9f * (0.5f + 0.5f * cosf(params.frame_count * 0.045f + j * 1.0f));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}

```

Filter #763 - glitch_colorxor19

Color XOR glitch variant 19-edge-weighted XOR pattern.

Technique: Edge-weighted XOR v19.

```

// Case 763 in unifiedFilterKernel:
process_glitch_Colorxor19(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor19(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {
        float a = 0.5f * (1.0f + 1.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.04f + j * 1.3f)));
        unsigned char val = gpu_wrap_cast(a * data[idx + j]);
        data[idx + j] = data[idx + j] ^ val;
    }
}

```

Filter #764 - glitch_colorxor20

Color XOR glitch variant 20-maximum complexity combined XOR from all previous patterns.

Technique: Multi-pattern combined XOR v20.

```

// Case 764 in unifiedFilterKernel:
process_glitch_Colorxor20(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Colorxor20(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq_x = (int)(gpu_rand(x, y, params.seed) * width);
    int sq_y = (int)(gpu_rand(y, x, params.seed) * height);
    int sq_w = (int)(gpu_rand(x + y, y - x, params.seed) * width);
    int sq_h = (int)(gpu_rand(x * y, x + y, params.seed) * height);
    if(x >= sq_x && x < sq_x + sq_w && y >= sq_y && y < sq_y + sq_h) {
        for(int j = 0; j < 3; ++j) {
            float a = 1.0f + 2.0f * (0.5f + 0.5f * sinf(params.frame_count * 0.05f + j * 1.5f));
            unsigned char val = gpu_wrap_cast(a * data[idx + j]);
            data[idx + j] = data[idx + j] ^ val;
        }
    }
}

```

Filter #765 - glitch_distort_picture

Picture distortion glitch-randomly warps and stretches rectangular regions of the frame, creating data-corruption-style picture distortion.

Technique: Random region stretch/warp distortion.

```
// Case 765 in unifiedFilterKernel:
process_glitch_DistortPicture(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_DistortPicture(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int rwidth = 1 + ((params.seed + y * 5) % width);
    int offx = gpu_AC_GetFX(width - 1, x, rwidth);
    int offset = (y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[offset];
    if(ref && offx >= 0 && offx < width) {
        int src_idx = y * step + offx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #766 - glitch_echo

Glitch echo-delayed frame echo with glitchy interruptions, creating stuttered temporal echoes with corruption.

Technique: Stuttered glitchy frame echo.

```
// Case 766 in unifiedFilterKernel:
process_glitch_Echo(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Echo(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = width / 2, cy = height / 2;
    int x2 = cx + (x - cx) / 2, y2 = cy + (y - cy) / 2;
    int x3 = cx + (x - cx) / 4, y3 = cy + (y - cy) / 4;
    if(x2 >= 0 && x2 < width && y2 >= 0 && y2 < height && x3 >= 0 && x3 < width && y3 >= 0 && y3 <
    ↪ height) {
        int idx2 = y2 * step + x2 * 4;
        int idx3 = y3 * step + x3 * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(0.4f * data[idx + j] + 0.3f * data[idx2 + j] + 0.3f * data[
    ↪ idx3 + j]);
    }
}
```

Filter #767 - glitch_fast_monocolortrails

Fast monochrome color trails-rapid single-color-channel motion trails that produce fast-moving colored ghost streaks.

Technique: Single-channel rapid temporal trailing.

```
// Case 767 in unifiedFilterKernel:
process_glitch_FastMonocolortrails(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FastMonocolortrails(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int f_indices[3] = {1 % params.numFrames, (params.numFrames / 2) % params.numFrames, (params.
    ↪ numFrames - 1) % params.numFrames};
    for(int j = 0; j < 3; ++j) {
        unsigned char *ref = allFrames[f_indices[j]];
        if(ref) data[idx + j] = ref[idx];
    }
}
```

Filter #768 - glitch_frame_pixels

Frame pixel glitch-individual pixels are randomly grabbed from different history frames, creating a per-pixel temporal scatter.

Technique: Per-pixel random history-frame sourcing.

```
// Case 768 in unifiedFilterKernel:
process_glitch_FramePixels(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FramePixels(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int bx = x / 64, by = y / 64;
    int sub_x = x % 64, sub_y = y % 64;
    int off = ((bx + by) * 8 + (sub_x / 8 + sub_y / 8)) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #769 - glitch_frame_resize_stretch

Frame resize stretch glitch-the image is stretched/squished to random sizes, creating rubber-band resize distortion.

Technique: Random resize/stretch distortion.

```
// Case 769 in unifiedFilterKernel:
process_glitch_FrameResizeStretch(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void process_glitch_FrameResizeStretch(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int frame_size = width / 32;
    if(frame_size < 1) frame_size = 1;
    int off = ((y / frame_size) + params.frame_count) % params.numFrames;
    int frame_x_off = (int)(gpu_rand(0, y / frame_size, params.seed) * (width / 16));
    int off_x = gpu_AC_GetFX(width - 1, x, width + frame_x_off);
    unsigned char *ref = allFrames[off];
    if(ref && off_x >= 0 && off_x < width) {
        int src_idx = y * step + off_x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #770 - glitch_frame_skip

Frame skip glitch-randomly skips to different history frames, creating a stuttering/jumping playback effect.

Technique: Random history-frame skip jump.

```

// Case 770 in unifiedFilterKernel:
process_glitch_FrameSkip(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameSkip(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool on = ((params.frame_count / 10) % 2) == 0;
    int off = params.frame_count % params.numFrames;
    if(on) {
        unsigned char *ref = allFrames[off];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}

```

Filter #771 - glitch_frame_skip_resize

Frame skip with resize-combines frame skipping with random resizing for a compounded skip+stretch glitch.

Technique: Combined frame skip + random resize.

```

// Case 771 in unifiedFilterKernel:
process_glitch_FrameSkipResize(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameSkipResize(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool on = ((params.frame_count / 12) % 2) == 0;
    int off = params.frame_count % params.numFrames;

```



```

    if(on) {
        int bsize = 8 + gpu_triangle_wave(params.frame_count, 16);
        int bx = (x / bsize) * bsize;
        int by = (y / bsize) * bsize;
        unsigned char *ref = allFrames[off];
        if(ref && bx < width && by < height) {
            int src_idx = by * step + bx * 4;
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[src_idx + j];
        }
    }
}

```

Filter #772 - glitch_frame_skip_shadow

Frame skip with shadow-frame skipping plus a dark ghost of the skipped frames trailing behind.

Technique: Frame skip + dark ghost trail.

```

// Case 772 in unifiedFilterKernel:
process_glitch_FrameSkipShadow(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameSkipShadow(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool on = ((params.frame_count / 15) % 2) == 0;
    int off = gpu_triangle_wave(params.frame_count, params.numFrames - 1);
    if(on) {
        unsigned char *ref = allFrames[off % params.numFrames];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
        }
    }
}

```

Filter #773 - glitch_frame_skip_shadow_x2

Frame skip with double shadow-frame skipping with two shadow layers at different opacities for deeper trailing.

Technique: Frame skip + double ghost trail.

```

// Case 773 in unifiedFilterKernel:
process_glitch_FrameSkipShadowX2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameSkipShadowX2(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool on = ((params.frame_count / 15) % 2) == 0;
    int off = gpu_triangle_wave(params.frame_count, params.numFrames - 1);
    if(on) {
        unsigned char *ref1 = allFrames[off % params.numFrames];
        unsigned char *ref2 = allFrames[(params.numFrames - 2) % params.numFrames];
        if(ref1 && ref2) {

```

```

        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.33f * data[idx + j] + 0.33f * ref1[idx + j] + 0.33f *
↪ ref2[idx + j]);
    }
}

```

Filter #774 - glitch_frame_slide01

Frame slide glitch 01-the current frame slides horizontally, revealing history frames beneath at the edge.

Technique: Horizontal frame slide reveal v1.

```

// Case 774 in unifiedFilterKernel:
process_glitch_FrameSlide01(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameSlide01(int x, int y, unsigned char *data, unsigned char **allFrames
↪ , int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int half_w = width / 2;
    int offset_x = gpu_triangle_wave(params.frame_count * 10, width - half_w - 1);
    int off = params.frame_count % params.numFrames;
    if(x >= offset_x && x < offset_x + half_w) {
        unsigned char *ref = allFrames[off];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}

```

Filter #775 - glitch_frame_slide02

Frame slide glitch 02-vertical sliding variant.

Technique: Vertical frame slide reveal v2.

```

// Case 775 in unifiedFilterKernel:
process_glitch_FrameSlide02(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameSlide02(int x, int y, unsigned char *data, unsigned char **allFrames
↪ , int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int half_h = height / 2;
    int offset_y = gpu_triangle_wave(params.frame_count * 10, height - half_h - 1);
    int off = params.frame_count % params.numFrames;
    if(y >= offset_y && y < offset_y + half_h) {
        unsigned char *ref = allFrames[off];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}

```

Filter #776 - glitch_frame_slide03

Frame slide glitch 03-diagonal sliding variant.

Technique: Diagonal frame slide reveal v3.

```
// Case 776 in unifiedFilterKernel:
process_glitch_FrameSlide03(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameSlide03(int x, int y, unsigned char *data, unsigned char **allFrames
    ↪ , int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int half_w = width / 2;
    int offset_x = gpu_triangle_wave(params.frame_count * 10, width - half_w - 1);
    int off = (int)(gpu_rand(0, 0, params.seed) * params.numFrames) % params.numFrames;
    if(x >= offset_x && x < offset_x + half_w) {
        unsigned char *ref = allFrames[off];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #777 - glitch_frame_slide04

Frame slide glitch 04-multi-direction slide with animated direction changes.

Technique: Multi-direction animated frame slide v4.

```
// Case 777 in unifiedFilterKernel:
process_glitch_FrameSlide04(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameSlide04(int x, int y, unsigned char *data, unsigned char **allFrames
    ↪ , int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int half_h = height / 2;
    int offset_y = gpu_triangle_wave(params.frame_count * 10, height - half_h - 1);
    int off = (int)(gpu_rand(0, 0, params.seed) * params.numFrames) % params.numFrames;
    if(y >= offset_y && y < offset_y + half_h) {
        unsigned char *ref = allFrames[off];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #778 - glitch_frame_square_col

Frame square column glitch-columns of square blocks sourced from different history frames, creating a columnar temporal mosaic.

Technique: Column-aligned block history mosaic.

```
// Case 778 in unifiedFilterKernel:
process_glitch_FrameSquareCol(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameSquareCol(int x, int y, unsigned char *data, unsigned char **
↳ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int bx = x / 34, by = y / 34;
    int inner_x = x % 34, inner_y = y % 34;
    if(inner_x < 30 && inner_y < 30) {
        int off = gpu_triangle_wave(bx + by * (width / 34), params.numFrames - 1);
        unsigned char *ref = allFrames[off % params.numFrames];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #779 - glitch_frame_stutter_filter

Frame stutter filter-the video stutters/repeats frames with a filtered blend between repeats.

Technique: Filtered frame stutter/repeat.

```
// Case 779 in unifiedFilterKernel:
process_glitch_FrameStutterFilter(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_FrameStutterFilter(int x, int y, unsigned char *data, unsigned char **
↳ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool stutter = ((params.frame_count / 8) % 3) != 0;
    if(stutter) {
        int off = (params.frame_count / 8) % params.numFrames;
        unsigned char *ref = allFrames[off];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #780 - glitch_glitch_x1

Meta glitch X1-combines multiple glitch techniques (offset, color shift, noise) into one compound glitch effect.

Technique: Multi-technique compound glitch.

```
// Case 780 in unifiedFilterKernel:
process_glitch_GlitchX1(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_GlitchX1(int x, int y, unsigned char *data, unsigned char **allFrames,
↳ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
```

```

    int divisor = height / 4;
    if(divisor < 1) divisor = 1;
    int band = y / divisor;
    float r = gpu_rand(0, band, params.seed + params.frame_count);
    if(r > 0.25f) {
        int off = ((int)(r * 100) + params.frame_count) % params.numFrames;
        unsigned char *ref = allFrames[off];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}

```

Filter #781 - glitch_gpt1

GPT-generated glitch 1-AI-designed glitch pattern with unique mathematical distortion formula.

Technique: Custom formula glitch v1.

```

// Case 781 in unifiedFilterKernel:
process_glitch_Gpt1(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Gpt1(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int col_shift = (int)(gpu_rand(0, 0, params.seed) * (width / 4));
    int row_shift = (int)(gpu_rand(1, 1, params.seed) * (height / 4));
    int sx = (x + col_shift) % width;
    int sy = (y + row_shift) % height;
    data[idx + 2] = data[y * step + sx * 4 + 2];
    data[idx + 1] = data[sy * step + x * 4 + 1];
    if(gpu_rand(x, y, params.seed) < 0.2f)
        data[idx] = (unsigned char)(gpu_rand(x, y, params.seed + 1) * 255.0f);
}

```

Filter #782 - glitch_gpt2

GPT-generated glitch 2-different AI-designed glitch with alternative mathematical transformation.

Technique: Custom formula glitch v2.

```

// Case 782 in unifiedFilterKernel:
process_glitch_Gpt2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Gpt2(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float amplitude = 25.0f;
    float frequency = 2.0f * 3.14159f / 180.0f;
    float phase = params.frame_count * 0.05f;
    int dx = (int)(amplitude * sinf(frequency * y + phase));
    int dy = (int)(amplitude * cosf(frequency * x + phase));
    int nx = x + dx;
    int ny = y + dy;
    if(nx >= 0 && nx < width && ny >= 0 && ny < height) {

```

```

        int src_idx = ny * step + nx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
    }
}

```

Filter #783 - glitch_gpt3

GPT-generated glitch 3-third AI-designed glitch pattern.

Technique: Custom formula glitch v3.

```

// Case 783 in unifiedFilterKernel:
process_glitch_Gpt3(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Gpt3(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float cx = width / 2.0f, cy = height / 2.0f;
    float dx = x - cx, dy = y - cy;
    float dist = sqrtf(dx * dx + dy * dy);
    float maxDist = sqrtf(cx * cx + cy * cy);
    float factor = 1.0f + 0.3f * sinf(params.frame_count * 0.05f) * (dist / maxDist);
    int nx = (int)(cx + dx * factor);
    int ny = (int)(cy + dy * factor);
    if(nx >= 0 && nx < width && ny >= 0 && ny < height) {
        int src_idx = ny * step + nx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = data[src_idx + j];
    }
}

```

Filter #784 - glitch_invertflash

Invert flash glitch-harsh periodic color inversion flash with glitch timing (not regular strobe).

Technique: Irregular periodic inversion flash.

```

// Case 784 in unifiedFilterKernel:
process_glitch_Invertflash(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Invertflash(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    if((params.frame_count / 4) % 2 == 0) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = 255 - data[idx + j];
    }
}

```

Filter #785 - glitch_line_across_side01

Line across side glitch 01-horizontal glitch lines shoot across from one side of the frame with displacement. Variants 01-06 differ in speed, direction, and frequency.

Technique: Horizontal shooting glitch line v01.

```
// Case 785 in unifiedFilterKernel:
process_glitch_LineAcrossSide01(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineAcrossSide01(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int row_col = gpu_triangle_wave(y + params.frame_count * 25, width) + x;
    int row_x = gpu_AC_GetFX(width - 1, x, row_col);
    int off = (y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && row_x >= 0 && row_x < width) {
        int src_idx = y * step + row_x * 4;
        for(int j = 0; j < 3; ++j) {
            if(1.0f < 1.0f)
                data[idx + j] = gpu_wrap_cast(1.0f * data[idx + j] + 1.0f * ref[src_idx + j]);
            else
                data[idx + j] = ref[src_idx + j];
        }
    }
}
```

Filter #786 - glitch_line_across_side02

Line across side variant 02-from opposite direction.

Technique: Reverse horizontal shooting glitch line v02.

```
// Case 786 in unifiedFilterKernel:
process_glitch_LineAcrossSide02(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineAcrossSide02(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int row_col = gpu_triangle_wave(y + params.frame_count * 25, width) + x;
    int row_x = gpu_AC_GetFX(width - 1, x, row_col);
    int off = (y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && row_x >= 0 && row_x < width) {
        int src_idx = y * step + row_x * 4;
        for(int j = 0; j < 3; ++j) {
            if(1.0f < 1.0f)
                data[idx + j] = gpu_wrap_cast(1.0f * data[idx + j] + 1.0f * ref[src_idx + j]);
            else
                data[idx + j] = ref[src_idx + j];
        }
    }
}
```

Filter #787 - glitch_line_across_side03

Line across side variant 03-faster speed.

Technique: Fast horizontal glitch line v03.

```
// Case 787 in unifiedFilterKernel:
process_glitch_LineAcrossSide03(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineAcrossSide03(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int row_col = gpu_triangle_wave(y + params.frame_count * 25, width) + x;
    int row_x = gpu_AC_GetFX(width - 1, x, row_col);
    int off = (y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && row_x >= 0 && row_x < width) {
        int src_idx = y * step + row_x * 4;
        for(int j = 0; j < 3; ++j) {
            if(0.5f < 1.0f)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[src_idx + j]);
            else
                data[idx + j] = ref[src_idx + j];
        }
    }
}
```

Filter #788 - glitch_line_across_side04

Line across side variant 04-with color shift.

Technique: Color-shifted horizontal glitch line v04.

```
// Case 788 in unifiedFilterKernel:
process_glitch_LineAcrossSide04(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineAcrossSide04(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int row_col = gpu_triangle_wave(y + params.frame_count * 25, width) + x;
    int row_x = gpu_AC_GetFX(width - 1, x, row_col);
    int off = (y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && row_x >= 0 && row_x < width) {
        int src_idx = y * step + row_x * 4;
        for(int j = 0; j < 3; ++j) {
            if(0.5f < 1.0f)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[src_idx + j]);
            else
                data[idx + j] = ref[src_idx + j];
            data[idx + j] ^= ref[src_idx + j];
        }
    }
}
```


Filter #789 - glitch_line_across_side05

Line across side variant 05-multiple simultaneous lines.

Technique: Multi-line horizontal glitch v05.

```
// Case 789 in unifiedFilterKernel:
process_glitch_LineAcrossSide05(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineAcrossSide05(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int row_col = gpu_triangle_wave(y + params.frame_count * 25, width) + x;
    int row_x = gpu_AC_GetFX(width - 1, x, row_col);
    int off = (y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && row_x >= 0 && row_x < width) {
        int src_idx = y * step + row_x * 4;
        for(int j = 0; j < 3; ++j) {
            if(0.5f < 1.0f)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[src_idx + j]);
            else
                data[idx + j] = ref[src_idx + j];
        }
    }
}
```

Filter #790 - glitch_line_across_side06

Line across side variant 06-with displacement wake.

Technique: Displacement-wake horizontal glitch line v06.

```
// Case 790 in unifiedFilterKernel:
process_glitch_LineAcrossSide06(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineAcrossSide06(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int row_col = gpu_triangle_wave(y + params.frame_count * 25, width) + x;
    int row_x = gpu_AC_GetFX(width - 1, x, row_col);
    int off = (y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && row_x >= 0 && row_x < width) {
        int src_idx = y * step + row_x * 4;
        for(int j = 0; j < 3; ++j) {
            if(1.0f < 1.0f)
                data[idx + j] = gpu_wrap_cast(1.0f * data[idx + j] + 1.0f * ref[src_idx + j]);
            else
                data[idx + j] = ref[src_idx + j];
        }
    }
}
```

Filter #791 - glitch_line_collection01

Line collection glitch 01-draws collections of displaced horizontal lines at various positions. Variants 01-07 use different arrangements and densities.

Technique: Displaced horizontal line collection v01.

```
// Case 791 in unifiedFilterKernel:
process_glitch_LineCollection01(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineCollection01(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int band = width / 32;
    if(band < 1) band = 1;
    int seg = x / band;
    int off = (seg + y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #792 - glitch_line_collection02

Line collection variant 02-denser line packing.

Technique: Dense horizontal line collection v02.

```
// Case 792 in unifiedFilterKernel:
process_glitch_LineCollection02(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineCollection02(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int band = width / 32;
    if(band < 1) band = 1;
    int seg = x / band;
    int off = (seg + y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #793 - glitch_line_collection03

Line collection variant 03-variable-width lines.

Technique: Variable-width line collection v03.

```
// Case 793 in unifiedFilterKernel:
process_glitch_LineCollection03(x, y, data, allFrames, width, height, step, params);
```

```
// Referenced __device__ function(s):
__device__ void process_glitch_LineCollection03(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int band = 2;
    if(band < 1) band = 1;
    int seg = x / band;
    int off = (seg + y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #794 - glitch_line_collection04

Line collection variant 04-with color tinting per line.

Technique: Color-tinted line collection v04.

```
// Case 794 in unifiedFilterKernel:
process_glitch_LineCollection04(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineCollection04(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int band = 10;
    if(band < 1) band = 1;
    int seg = x / band;
    int off = (seg + y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #795 - glitch_line_collection05

Line collection variant 05-animated line positions.

Technique: Animated position line collection v05.

```
// Case 795 in unifiedFilterKernel:
process_glitch_LineCollection05(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineCollection05(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int band = 50;
    if(band < 1) band = 1;
    int seg = x / band;
    int off = (seg + y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
```

```

        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```

Filter #796 - glitch_line_collection06

Line collection variant 06-crossing/intersecting lines.

Technique: Crossing line collection v06.

```

// Case 796 in unifiedFilterKernel:
process_glitch_LineCollection06(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineCollection06(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int band = 50;
    if(band < 1) band = 1;
    int seg = x / band;
    int off = (seg + y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```

Filter #797 - glitch_line_collection07

Line collection variant 07-maximum density combined.

Technique: Maximum density line collection v07.

```

// Case 797 in unifiedFilterKernel:
process_glitch_LineCollection07(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineCollection07(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int band = 50;
    if(band < 1) band = 1;
    int seg = x / band;
    int off = (seg + y + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```

Filter #798 - glitch_line_offset_inout

Line offset in/out-horizontal lines grow outward from center then retract inward, oscillating in/out.

Technique: Center-expanding/contracting line offset v1.

```
// Case 798 in unifiedFilterKernel:
process_glitch_LineOffsetInout(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineOffsetInout(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int num = 1 + (int)(gpu_rand(0, y, params.seed) * 50.0f);
    if(num < 1) num = 1;
    int off = (y / num + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #799 - glitch_line_offset_inout2

Line offset in/out variant 2-faster oscillation with more lines.

Technique: Fast expanding/contracting line offset v2.

```
// Case 799 in unifiedFilterKernel:
process_glitch_LineOffsetInout2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_LineOffsetInout2(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int num = 1 + (int)(gpu_rand(0, y, params.seed + 1) * 250.0f);
    if(num < 1) num = 1;
    int off = gpu_triangle_wave(y / num + params.frame_count, params.numFrames - 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #800 - glitch_line_offset_inout3

Line offset in/out variant 3-asymmetric expansion with trailing displacement.

Technique: Asymmetric expanding line offset v3.

```
// Case 800 in unifiedFilterKernel:
process_glitch_LineOffsetInout3(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void process_glitch_LineOffsetInout3(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int num = 1 + (int)(gpu_rand(0, y, params.seed + 2) * 250.0f);
    if(num < 1) num = 1;
    int off = gpu_triangle_wave(y / num + params.frame_count, params.numFrames - 1);
    int offx = gpu_AC_GetFX(width - 1, x, width + num);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref && offx >= 0 && offx < width) {
        int src_idx = y * step + offx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #801 - glitch_newrandblend

New random blend-randomly blends rectangular patches from different history frames, creating a patchwork of time points.

Technique: Random rectangular patch temporal blend.

```

// Case 801 in unifiedFilterKernel:
process_glitch_Newrandblend(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Newrandblend(int x, int y, unsigned char *data, unsigned char **allFrames
    ↪ , int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = (int)(gpu_rand(x, y, params.seed + params.frame_count) * params.numFrames) % params.
    ↪ numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
    }
}

```

Filter #802 - glitch_newstretchlines

New stretch lines-horizontal lines are randomly stretched or compressed, creating rubber-band-line distortion.

Technique: Random horizontal line stretch/compress.

```

// Case 802 in unifiedFilterKernel:
process_glitch_Newstretchlines(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Newstretchlines(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int stretch = width + gpu_triangle_wave(y + params.frame_count, width / 2);
    int sx = gpu_AC_GetFX(width - 1, x, stretch);
    int off = params.frame_count % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && sx >= 0 && sx < width) {

```

```

        int src_idx = y * step + sx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #803 - glitch_newvarlines

New variable lines-draws lines of varying thickness, color, and offset across the frame in glitchy patterns.

Technique: Variable-parameter random line drawing.

```

// Case 803 in unifiedFilterKernel:
process_glitch_Newvarlines(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Newvarlines(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int var = 2 + (int)(gpu_rand(0, y, params.seed) * 30.0f);
    int off = (x / var + params.frame_count) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```

Filter #804 - glitch_outoforder

Out of order-rows or blocks of the frame are shuffled into a random order, scrambling the vertical arrangement of the image.

Technique: Random row/block order shuffling.

```

// Case 804 in unifiedFilterKernel:
process_glitch_Outoforder(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Outoforder(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int scramble = (int)(gpu_rand(0, y / 4, params.seed) * params.numFrames);
    int off = scramble % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```

Filter #805 - glitch_pic_adjust1

Picture adjustment glitch 1-randomly adjusts brightness, contrast, and color balance per frame, creating fluctuating exposure.

Technique: Random per-frame exposure adjustment v1.

```
// Case 805 in unifiedFilterKernel:
process_glitch_PicAdjust1(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PicAdjust1(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float factor = 1.0f + 0.5f * sinf(params.frame_count * 0.05f);
    for(int j = 0; j < 3; ++j) {
        float val = data[idx + j] * factor;
        data[idx + j] = (unsigned char)fminf(255.0f, fmaxf(0.0f, val));
    }
}
```

Filter #806 - glitch_pic_adjust2

Picture adjustment glitch 2-more aggressive version with wider adjustment ranges.

Technique: Aggressive random exposure adjustment v2.

```
// Case 806 in unifiedFilterKernel:
process_glitch_PicAdjust2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PicAdjust2(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r_factor = 1.0f + 0.3f * sinf(params.frame_count * 0.03f);
    float g_factor = 1.0f + 0.3f * sinf(params.frame_count * 0.05f);
    float b_factor = 1.0f + 0.3f * sinf(params.frame_count * 0.07f);
    data[idx] = (unsigned char)fminf(255.0f, data[idx] * b_factor);
    data[idx + 1] = (unsigned char)fminf(255.0f, data[idx + 1] * g_factor);
    data[idx + 2] = (unsigned char)fminf(255.0f, data[idx + 2] * r_factor);
}
```

Filter #807 - glitch_picture_jump01

Picture jump glitch 01-the entire image jumps/teleports to a random position offset for a few frames then returns. Variants 01-31 use different jump patterns, distances, and timing.

Technique: Random position jump teleport v01.

```
// Case 807 in unifiedFilterKernel:
process_glitch_PictureJump01(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump01(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = height / 16 + y + params.frame_count * 1;
    if(size_y > height * 2) size_y = height / 16 + (size_y % (height * 2));
}
```



```

    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 1) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.2f * data[idx + j] + 0.8f * ref[src_idx + j]);
    }
}

```

Filter #808 - glitch_picture_jump02

Picture jump variant 02-horizontal-only jumps.

Technique: Horizontal position jump v02.

```

// Case 808 in unifiedFilterKernel:
process_glitch_PictureJump02(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump02(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = height / 16 + y + params.frame_count * 2;
    if(size_y > height * 2) size_y = height / 16 + (size_y % (height * 2));
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 2) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.2f * data[idx + j] + 0.8f * ref[src_idx + j]);
    }
}

```

Filter #809 - glitch_picture_jump03

Picture jump variant 03-vertical-only jumps.

Technique: Vertical position jump v03.

```

// Case 809 in unifiedFilterKernel:
process_glitch_PictureJump03(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump03(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = height / 16 + y + params.frame_count * 3;
    if(size_y > height * 2) size_y = height / 16 + (size_y % (height * 2));
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 3) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.2f * data[idx + j] + 0.8f * ref[src_idx + j]);
    }
}

```

```
}
```

Filter #810 - glitch_picture_jump04

Picture jump variant 04-diagonal jumps.

Technique: Diagonal position jump v04.

```
// Case 810 in unifiedFilterKernel:
process_glitch_PictureJump04(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump04(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_x = width / 16 + y + params.frame_count * 4;
    if(size_x > width * 2) size_x = width / 16 + (size_x % (width * 2));
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 4) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cx >= 0 && cx < width) {
        int src_idx = y * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #811 - glitch_picture_jump05

Picture jump variant 05-small rapid micro-jumps.

Technique: Rapid micro position jump v05.

```
// Case 811 in unifiedFilterKernel:
process_glitch_PictureJump05(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump05(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 1, height * 2);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 5) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #812 - glitch_picture_jump06

Picture jump variant 06-large infrequent jumps.

Technique: Infrequent large position jump v06.

```
// Case 812 in unifiedFilterKernel:
process_glitch_PictureJump06(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump06(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = height / 16 + y + params.frame_count * 6;
    if(size_y > height * 2) size_y = height / 16 + (size_y % (height * 2));
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 6) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.2f * data[idx + j] + 0.8f * ref[src_idx + j]);
    }
}
```

Filter #813 - glitch_picture_jump07

Picture jump variant 07-smooth slide transition between jumps.

Technique: Sliding position jump transition v07.

```
// Case 813 in unifiedFilterKernel:
process_glitch_PictureJump07(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump07(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = height / 16 + y + params.frame_count * 7;
    if(size_y > height * 2) size_y = height / 16 + (size_y % (height * 2));
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 7) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.2f * data[idx + j] + 0.8f * ref[src_idx + j]);
    }
}
```

Filter #814 - glitch_picture_jump08

Picture jump variant 08-with zoom during jump.

Technique: Position jump + zoom v08.

```
// Case 814 in unifiedFilterKernel:
process_glitch_PictureJump08(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump08(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
```

```

    int idx = y * step + x * 4;
    int size_y = height / 16 + y + params.frame_count * 8;
    if(size_y > height * 2) size_y = height / 16 + (size_y % (height * 2));
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 8) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.2f * data[idx + j] + 0.8f * ref[src_idx + j]);
    }
}

```

Filter #815 - glitch_picture_jump09

Picture jump variant 09-with rotation during jump.

Technique: Position jump + rotation v09.

```

// Case 815 in unifiedFilterKernel:
process_glitch_PictureJump09(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump09(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = height / 16 + y + params.frame_count * 9;
    if(size_y > height * 2) size_y = height / 16 + (size_y % (height * 2));
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 9) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.2f * data[idx + j] + 0.8f * ref[src_idx + j]);
    }
}

```

Filter #816 - glitch_picture_jump10

Picture jump variant 10-with color shift during jump.

Technique: Position jump + color shift v10.

```

// Case 816 in unifiedFilterKernel:
process_glitch_PictureJump10(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump10(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = (int)(gpu_rand(0, y / (height / 2 + 1), params.seed) * (height - 1)) + 1;
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 10) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)

```

```

        data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #817 - glitch_picture_jump11

Picture jump variant 11-bouncing jump with deceleration.

Technique: Bouncing deceleration jump v11.

```

// Case 817 in unifiedFilterKernel:
process_glitch_PictureJump11(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump11(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_x = width / 16 + y + params.frame_count * 11;
    if(size_x > width * 2) size_x = width / 16 + (size_x % (width * 2));
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 11) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cx >= 0 && cx < width) {
        int src_idx = y * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #818 - glitch_picture_jump12

Picture jump variant 12-elastic snap-back after jump.

Technique: Elastic snap-back jump v12.

```

// Case 818 in unifiedFilterKernel:
process_glitch_PictureJump12(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump12(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_x = width / 16 + y + params.frame_count * 12;
    if(size_x > width * 2) size_x = width / 16 + (size_x % (width * 2));
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 12) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cx >= 0 && cx < width) {
        int src_idx = y * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #819 - glitch_picture_jump13

Picture jump variant 13-spiral jump path.

Technique: Spiral path position jump v13.

```
// Case 819 in unifiedFilterKernel:
process_glitch_PictureJump13(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump13(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_x = width / 16 + y + params.frame_count * 13;
    if(size_x > width * 2) size_x = width / 16 + (size_x % (width * 2));
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 13) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cx >= 0 && cx < width) {
        int src_idx = y * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #820 - glitch_picture_jump14

Picture jump variant 14-multi-step sequential jumps.

Technique: Sequential multi-step jump v14.

```
// Case 820 in unifiedFilterKernel:
process_glitch_PictureJump14(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump14(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_x = width / 16 + y + params.frame_count * 14;
    if(size_x > width * 2) size_x = width / 16 + (size_x % (width * 2));
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 14) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cx >= 0 && cx < width) {
        int src_idx = y * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #821 - glitch_picture_jump15

Picture jump variant 15-X channel jumps separately from Y.

Technique: Independent X/Y channel jump v15.

```
// Case 821 in unifiedFilterKernel:
process_glitch_PictureJump15(x, y, data, allFrames, width, height, step, params);
```

```

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump15(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 11, height * 2);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 15) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #822 - glitch_picture_jump16

Picture jump variant 16-with ghost trail during jump.

Technique: Ghost-trailed position jump v16.

```

// Case 822 in unifiedFilterKernel:
process_glitch_PictureJump16(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump16(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 12, height * 2);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 16) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #823 - glitch_picture_jump17

Picture jump variant 17-frame-skip combined with jump.

Technique: Frame-skip + position jump v17.

```

// Case 823 in unifiedFilterKernel:
process_glitch_PictureJump17(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump17(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 13, height * 2);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 17) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {

```

```

        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #824 - glitch_picture_jump18

Picture jump variant 18-jump with XOR artifact.

Technique: XOR artifact position jump v18.

```

// Case 824 in unifiedFilterKernel:
process_glitch_PictureJump18(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump18(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 14, height * 2);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 18) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #825 - glitch_picture_jump19

Picture jump variant 19-sinusoidal jump path.

Technique: Sine-wave position jump path v19.

```

// Case 825 in unifiedFilterKernel:
process_glitch_PictureJump19(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump19(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 15, height * 2);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 19) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```


Filter #826 - glitch_picture_jump20

Picture jump variant 20-random walk jump sequence.

Technique: Random walk position jump v20.

```
// Case 826 in unifiedFilterKernel:
process_glitch_PictureJump20(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump20(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 16, height * 2);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int off = (params.frame_count + 20) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height) {
        int src_idx = cy * step + x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #827 - glitch_picture_jump21

Picture jump variant 21-accelerating jumps.

Technique: Accelerating position jump v21.

```
// Case 827 in unifiedFilterKernel:
process_glitch_PictureJump21(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump21(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 1, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 2, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 21) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #828 - glitch_picture_jump22

Picture jump variant 22-decelerating jumps.

Technique: Decelerating position jump v22.

```
// Case 828 in unifiedFilterKernel:
process_glitch_PictureJump22(x, y, data, allFrames, width, height, step, params);
```

```
// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump22(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 2, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 3, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 22) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #829 - glitch_picture_jump23

Picture jump variant 23-pulsating amplitude jumps.

Technique: Pulse-amplitude position jump v23.

```
// Case 829 in unifiedFilterKernel:
process_glitch_PictureJump23(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump23(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 3, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 4, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 23) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #830 - glitch_picture_jump24

Picture jump variant 24-corner-biased jumps.

Technique: Corner-targeted position jump v24.

```
// Case 830 in unifiedFilterKernel:
process_glitch_PictureJump24(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump24(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 4, height);

```

```

int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 5, width);
int cy = gpu_AC_GetFZ(height - 1, y, size_y);
int cx = gpu_AC_GetFX(width - 1, x, size_x);
int off = (params.frame_count + 24) % params.numFrames;
unsigned char *ref = allFrames[off];
if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
    int src_idx = cy * step + cx * 4;
    for(int j = 0; j < 3; ++j)
        data[idx + j] = ref[src_idx + j];
}
}

```

Filter #831 - glitch_picture_jump25

Picture jump variant 25-edge-biased jumps.

Technique: Edge-targeted position jump v25.

```

// Case 831 in unifiedFilterKernel:
process_glitch_PictureJump25(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump25(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 5, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 1, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 25) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #832 - glitch_picture_jump26

Picture jump variant 26-center-avoidant jumps.

Technique: Center-avoidant position jump v26.

```

// Case 832 in unifiedFilterKernel:
process_glitch_PictureJump26(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump26(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 6, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 2, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 26) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {

```

```

        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #833 - glitch_picture_jump27

Picture jump variant 27-mirrored double-jump.

Technique: Mirrored double position jump v27.

```

// Case 833 in unifiedFilterKernel:
process_glitch_PictureJump27(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump27(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 7, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 3, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 27) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #834 - glitch_picture_jump28

Picture jump variant 28-with scale distortion.

Technique: Scale-distorted position jump v28.

```

// Case 834 in unifiedFilterKernel:
process_glitch_PictureJump28(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump28(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 1, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 4, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 28) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #835 - glitch_picture_jump29

Picture jump variant 29-with wrap-around.

Technique: Wrap-around position jump v29.

```
// Case 835 in unifiedFilterKernel:
process_glitch_PictureJump29(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump29(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 2, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 5, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 29) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #836 - glitch_picture_jump30

Picture jump variant 30-with partial visibility.

Technique: Partial-visibility position jump v30.

```
// Case 836 in unifiedFilterKernel:
process_glitch_PictureJump30(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump30(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 3, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 1, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 30) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #837 - glitch_picture_jump31

Picture jump variant 31-maximum combined jump effects.

Technique: Maximum-complexity combined jump v31.

```
// Case 837 in unifiedFilterKernel:
process_glitch_PictureJump31(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_PictureJump31(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int size_y = 2 + gpu_triangle_wave(y + params.frame_count * 4, height);
    int size_x = 2 + gpu_triangle_wave(x + params.frame_count * 2, width);
    int cy = gpu_AC_GetFZ(height - 1, y, size_y);
    int cx = gpu_AC_GetFX(width - 1, x, size_x);
    int off = (params.frame_count + 31) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && cy >= 0 && cy < height && cx >= 0 && cx < width) {
        int src_idx = cy * step + cx * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}
```

Filter #838 - glitch_rect_size

Rectangle size glitch-randomly-sized rectangles from the image are resized (scaled up/down) and placed back, creating scaled-block artifacts.

Technique: Random rectangle resize-in-place.

```
// Case 838 in unifiedFilterKernel:
process_glitch_RectSize(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_RectSize(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int num = 8 + gpu_triangle_wave(params.frame_count * 2, 56);
    if(num < 1) num = 1;
    int row_size = height / num;
    int col_size = width / num;
    if(row_size < 1) row_size = 1;
    if(col_size < 1) col_size = 1;
    int bx = x / col_size, by = y / row_size;
    int off = (int)(gpu_rand(bx, by, params.seed) * params.numFrames) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
    }
}
```

Filter #839 - glitch_rsquare2

Random square glitch 2-randomly-positioned square blocks are filled or displaced, creating square artifact blocks.

Technique: Random square block displacement v2.

```
// Case 839 in unifiedFilterKernel:
process_glitch_Rsquare2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Rsquare2(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r1 = gpu_rand(x / 32, y / 32, params.seed + 0);
    float r2 = gpu_rand(x / 64, y / 64, params.seed + 0 + 100);
    int off = (int)(r1 * params.numFrames) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && r2 > 0.3f) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #840 - glitch_rsquare3

Random square glitch 3-larger blocks with color tinting.

Technique: Large tinted square block displacement v3.

```
// Case 840 in unifiedFilterKernel:
process_glitch_Rsquare3(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Rsquare3(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r1 = gpu_rand(x / 32, y / 32, params.seed + 1);
    float r2 = gpu_rand(x / 64, y / 64, params.seed + 1 + 100);
    int off = (int)(r1 * params.numFrames) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && r2 > 0.3f) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #841 - glitch_rsquare4

Random square glitch 4-with XOR color treatment.

Technique: XOR-colored square blocks v4.

```
// Case 841 in unifiedFilterKernel:
process_glitch_Rsquare4(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Rsquare4(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r1 = gpu_rand(x / 32, y / 32, params.seed + 2);
    float r2 = gpu_rand(x / 64, y / 64, params.seed + 2 + 100);
    int off = (int)(r1 * params.numFrames) % params.numFrames;
    unsigned char *ref = allFrames[off];
```

```

    if(ref && r2 > 0.3f) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```

Filter #842 - glitch_rsquare5

Random square glitch 5-overlapping squares with blend.

Technique: Overlapping blended square blocks v5.

```

// Case 842 in unifiedFilterKernel:
process_glitch_Rsquare5(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Rsquare5(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r1 = gpu_rand(x / 32, y / 32, params.seed + 3);
    float r2 = gpu_rand(x / 64, y / 64, params.seed + 3 + 100);
    int off = (int)(r1 * params.numFrames) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref && r2 > 0.3f) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```

Filter #843 - glitch_slice_frame

Frame slice glitch-slices the frame into horizontal strips and offsets them randomly, creating a horizontal-slice glitch.

Technique: Random horizontal frame slicing.

```

// Case 843 in unifiedFilterKernel:
process_glitch_SliceFrame(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SliceFrame(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool active = ((params.frame_count / 30) % 2) == 0;
    if(active) {
        int rand_start = (int)(gpu_rand(0, 0, params.seed) * (height / 8));
        int half = width / 2;
        int offset_y = (x < half) ? rand_start : (int)(gpu_rand(1, 1, params.seed) * (height / 8));
        int src_y = y - offset_y;
        if(src_y >= 0 && src_y < height) {
            int src_idx = src_y * step + x * 4;
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
        }
    }
}

```

Filter #844 - glitch_slice_frame_w

Frame slice width variant-vertical slicing variant that cuts the frame into vertical strips with random offsets.

Technique: Random vertical frame slicing.

```
// Case 844 in unifiedFilterKernel:
process_glitch_SliceFrameW(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SliceFrameW(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool active = ((params.frame_count / 30) % 2) == 0;
    if(active) {
        int rand_start = (int)(gpu_rand(0, 0, params.seed) * (width / 8));
        int half = height / 2;
        int offset_x = (y < half) ? rand_start : (int)(gpu_rand(1, 1, params.seed) * (width / 8));
        int src_x = x - offset_x;
        if(src_x >= 0 && src_x < width) {
            int src_idx = y * step + src_x * 4;
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
        }
    }
}
```

Filter #845 - glitch_square_block_v2a

Square block glitch v2a-square-tiled block manipulation variant A. The v2 series (a-h) each apply different per-block transformations.

Technique: Per-block transformation variant A.

```
// Case 845 in unifiedFilterKernel:
process_glitch_SquareBlockV2A(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SquareBlockV2A(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 2 + gpu_triangle_wave(params.frame_count * 2 + 0, 6);
    if(sq < 1) sq = 1;
    int row_band = y / sq;
    int off = gpu_triangle_wave(row_band + 0, params.numFrames - 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        if(1.0f < 1.0f) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(1.0f * data[idx + j] + 1.0f * ref[idx + j]);
        } else {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #846 - glitch_square_block_v2b

Square block variant B-per-block color inversion.

Technique: Per-block color inversion variant B.

```
// Case 846 in unifiedFilterKernel:
process_glitch_SquareBlockV2B(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SquareBlockV2B(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 2 + gpu_triangle_wave(params.frame_count * 2 + 3, 6);
    if(sq < 1) sq = 1;
    int row_band = y / sq;
    int off = gpu_triangle_wave(row_band + 1, params.numFrames - 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        if(1.0f < 1.0f) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(1.0f * data[idx + j] + 1.0f * ref[idx + j]);
        } else {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #847 - glitch_square_block_v2c

Square block variant C-per-block brightness shift.

Technique: Per-block brightness shift variant C.

```
// Case 847 in unifiedFilterKernel:
process_glitch_SquareBlockV2C(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SquareBlockV2C(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 2 + gpu_triangle_wave(params.frame_count * 2 + 6, 6);
    if(sq < 1) sq = 1;
    int row_band = y / sq;
    int off = gpu_triangle_wave(row_band + 2, params.numFrames - 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        if(0.5f < 1.0f) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
        } else {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #848 - glitch_square_block_v2d

Square block variant D-per-block horizontal flip.

Technique: Per-block horizontal mirror variant D.

```
// Case 848 in unifiedFilterKernel:
process_glitch_SquareBlockV2D(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SquareBlockV2D(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 2 + gpu_triangle_wave(params.frame_count * 2 + 9, 6);
    if(sq < 1) sq = 1;
    int row_band = y / sq;
    int off = gpu_triangle_wave(row_band + 3, params.numFrames - 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        if(0.5f < 1.0f) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
        } else {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #849 - glitch_square_block_v2e

Square block variant E-per-block temporal swap (from history).

Technique: Per-block history swap variant E.

```
// Case 849 in unifiedFilterKernel:
process_glitch_SquareBlockV2E(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SquareBlockV2E(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 2 + gpu_triangle_wave(params.frame_count * 2 + 12, 6);
    if(sq < 1) sq = 1;
    int row_band = y / sq;
    int off = gpu_triangle_wave(row_band + 4, params.numFrames - 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        if(0.5f < 1.0f) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
        } else {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #850 - glitch_square_block_v2f

Square block variant F-per-block XOR with neighbor block.

Technique: Per-block neighbor XOR variant F.

```
// Case 850 in unifiedFilterKernel:
process_glitch_SquareBlockV2F(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SquareBlockV2F(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 2 + gpu_triangle_wave(params.frame_count * 2 + 15, 6);
    if(sq < 1) sq = 1;
    int row_band = y / sq;
    int off = gpu_triangle_wave(row_band + 5, params.numFrames - 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        if(0.5f < 1.0f) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
        } else {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #851 - glitch_square_block_v2g

Square block variant G-per-block channel rotation.

Technique: Per-block channel rotation variant G.

```
// Case 851 in unifiedFilterKernel:
process_glitch_SquareBlockV2G(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SquareBlockV2G(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 2 + gpu_triangle_wave(params.frame_count * 2 + 18, 6);
    if(sq < 1) sq = 1;
    int row_band = y / sq;
    int off = gpu_triangle_wave(row_band + 6, params.numFrames - 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        if(0.5f < 1.0f) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
        } else {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #852 - glitch_square_block_v2h

Square block variant H-combined all block transformations randomly per block.

Technique: Random mixed per-block transforms variant H.

```
// Case 852 in unifiedFilterKernel:
process_glitch_SquareBlockV2H(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SquareBlockV2H(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int sq = 2 + gpu_triangle_wave(params.frame_count * 2 + 21, 6);
    if(sq < 1) sq = 1;
    int row_band = y / sq;
    int off = gpu_triangle_wave(row_band + 7, params.numFrames - 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        if(0.5f < 1.0f) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
        } else {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #853 - glitch_square_xor

Square XOR glitch-XORs square-shaped regions with their background, creating sharp-edged XOR artifact squares.

Technique: Square-region XOR.

```
// Case 853 in unifiedFilterKernel:
process_glitch_SquareXor(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_SquareXor(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float alpha = 1.0f + 24.0f * (0.5f + 0.5f * sinf((x + y + params.frame_count) * 0.05f));
    if(x + 1 < width && y + 1 < height) {
        unsigned char total[3] = {0, 0, 0};
        for(int dy = 0; dy <= 1 && y + dy < height; ++dy) {
            for(int dx = 0; dx <= 1 && x + dx < width; ++dx) {
                int sidx = (y + dy) * step + (x + dx) * 4;
                for(int j = 0; j < 3; ++j)
                    total[j] = (unsigned char)fminf(255.0f, total[j] + data[sidx + j] * 0.25f);
            }
        }
        for(int j = 0; j < 3; ++j)
            data[idx + j] ^= (unsigned char)(alpha * total[j]);
    }
}
```

Filter #854 - glitch_store_frame10

Store frame 10 glitch-stores every 10th frame and replays it unexpectedly, creating periodic stuck-frame flashes.

Technique: Periodic stored-frame playback.

```
// Case 854 in unifiedFilterKernel:
process_glitch_StoreFrame10(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_StoreFrame10(int x, int y, unsigned char *data, unsigned char **allFrames
    ↪ , int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = (params.frame_count / 4) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #855 - glitch_stuckframe

Stuck frame glitch-the frame gets "stuck" for a random number of frames before updating, simulating frozen/stuck playback.

Technique: Random-duration frame freeze.

```
// Case 855 in unifiedFilterKernel:
process_glitch_Stuckframe(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_Stuckframe(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int arr[] = {1, 2, 2, 3, 3, 3};
    int arr_off = ((params.frame_count % 6) + 6) % 6;
    int pixel_idx = y * width + x;
    int off = 0;
    for(int k = 0; k < pixel_idx % 16; ++k) {
        off += arr[(arr_off + k) % 6];
        if(off >= params.numFrames) off = off % params.numFrames;
    }
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #856 - glitch_stutter_long

Long stutter glitch-extends the stutter effect over many frames, creating a long, droning repetition of a single frame.

Technique: Extended duration frame stutter.

```
// Case 856 in unifiedFilterKernel:
process_glitch_StutterLong(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_StutterLong(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int period = 15 + (int)(gpu_rand(0, 0, params.seed) * 15.0f);
    if(period < 1) period = 1;
    bool show = (params.frame_count % period) == 0;
    if(show) {
        int off = gpu_triangle_wave(params.frame_count / period, params.numFrames - 1);
        unsigned char *ref = allFrames[off % params.numFrames];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #857 - glitch_stutter_sbrv

Stutter with sub-reverse-stutters frames with occasional short reverse playback segments, creating a glitchy forward-reverse stutter.

Technique: Stutter + short reverse playback.

```
// Case 857 in unifiedFilterKernel:
process_glitch_StutterSbrv(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void process_glitch_StutterSbrv(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int period = 10 + (int)(gpu_rand(0, 0, params.seed + 1) * 20.0f);
    if(period < 1) period = 1;
    bool show = (params.frame_count % period) < (period / 2);
    if(show) {
        int sq = 4 + gpu_triangle_wave(params.frame_count, 12);
        int bx = (x / sq) * sq;
        int by = (y / sq) * sq;
        int off = params.frame_count % params.numFrames;
        unsigned char *ref = allFrames[off];
        if(ref && bx < width && by < height) {
            int src_idx = by * step + bx * 4;
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[src_idx + j];
        }
    }
}
```

AC Glitch Library (858-904)

Filter #858 - acgl_glitch_AlphaBlendFive

AC Glitch Library: blends 5 history frames with decreasing alpha weights for a deep, smooth multi-frame ghosting.

Technique: 5-frame decreasing-alpha blend.

```
// Case 858 in unifiedFilterKernel:
acgl_glitch_AlphaBlendFive(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_AlphaBlendFive(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float sum[3] = {0, 0, 0};
    int valid = 0;
    for(int i = 0; i < 5; ++i) {
        int fi = i % params.numFrames;
        unsigned char *ref = allFrames[fi];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                sum[j] += ref[idx + j];
            valid++;
        }
    }
    if(valid > 0) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(sum[j] / valid);
    }
}
```

Filter #859 - acgl_glitch_AlphaBlendTri

AC Glitch Library: blends 3 history frames (triangle pattern) with alternating alpha weights.

Technique: 3-frame triangular alpha blend.

```
// Case 859 in unifiedFilterKernel:
acgl_glitch_AlphaBlendTri(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_AlphaBlendTri(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int f0 = 0 % params.numFrames;
    int f1 = 3 % params.numFrames;
    int f2 = (params.numFrames - 1) % params.numFrames;
    unsigned char *r0 = allFrames[f0], *r1 = allFrames[f1], *r2 = allFrames[f2];
    if(r0 && r1 && r2) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(0.33f * r0[idx + j] + 0.33f * r1[idx + j] + 0.33f * r2[idx +
    ↪ j]);
    }
}
```


Filter #860 - acgl_glitch_AlphaBlendExpand

AC Glitch Library: alpha blend with an expanding region-the blended area grows outward from center over time.

Technique: Center-expanding alpha blend region.

```
// Case 860 in unifiedFilterKernel:
acgl_glitch_AlphaBlendExpand(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_AlphaBlendExpand(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int maxN = params.numFrames < 10 ? params.numFrames : 10;
    if(maxN < 2) maxN = 2;
    int iter = 1 + gpu_triangle_wave(params.frame_count, maxN - 1);
    if(iter < 1) iter = 1;
    if(iter > params.numFrames) iter = params.numFrames;
    float weight = 1.0f / iter;
    float sum[3] = {0, 0, 0};
    for(int q = 0; q < iter; ++q) {
        unsigned char *ref = allFrames[q % params.numFrames];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                sum[j] += weight * ref[idx + j];
        }
    }
    for(int j = 0; j < 3; ++j)
        data[idx + j] = gpu_wrap_cast(sum[j]);
}
```

Filter #861 - acgl_glitch_BarsCol

AC Glitch Library: vertical color bars-columns of the image are replaced with solid color bars at regular intervals.

Technique: Regular vertical solid color bars.

```
// Case 861 in unifiedFilterKernel:
acgl_glitch_BarsCol(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_BarsCol(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int num_cols = 64 + gpu_triangle_wave(params.frame_count, 64);
    if(num_cols < 1) num_cols = 1;
    int band = x / num_cols;
    int fsel = (int)(gpu_rand(band, params.frame_count / 4, params.seed) * 8.0f) % params.numFrames;
    unsigned char *ref = allFrames[fsel];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
    }
}
```

Filter #862 - acgl_glitch_BarsColAlpha

AC Glitch Library: semi-transparent vertical color bars blended with the underlying image.

Technique: Alpha-blended vertical color bars.

```
// Case 862 in unifiedFilterKernel:
acgl_glitch_BarsColAlpha(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_BarsColAlpha(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int num_cols = 64 + gpu_triangle_wave(params.frame_count, 64);
    if(num_cols < 1) num_cols = 1;
    int band = x / num_cols;
    float alpha = 0.1f + 0.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.05f));
    int fsel = (int)(gpu_rand(band, params.frame_count / 4, params.seed) * 8.0f) % params.numFrames;
    unsigned char *ref = allFrames[fsel];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(alpha * data[idx + j] + (1.0f - alpha) * ref[idx + j]);
    }
}
```

Filter #863 - acgl_glitch_BarsHoriz

AC Glitch Library: horizontal color bars-rows are replaced with solid color bars.

Technique: Regular horizontal solid color bars.

```
// Case 863 in unifiedFilterKernel:
acgl_glitch_BarsHoriz(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_BarsHoriz(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int range = height / 4 > 25 ? height / 4 - 25 : 1;
    int band_h = 25 + gpu_triangle_wave(params.frame_count * 5, range);
    if(band_h < 5) band_h = 5;
    int band = y / band_h;
    int fsel = (int)(gpu_rand(band, params.frame_count / 3, params.seed) * 8.0f) % params.numFrames;
    unsigned char *ref = allFrames[fsel];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
    }
}
```

Filter #864 - acgl_glitch_BarsHorizAlpha

AC Glitch Library: semi-transparent horizontal color bars blended with the image.

Technique: Alpha-blended horizontal color bars.

```
// Case 864 in unifiedFilterKernel:
acgl_glitch_BarsHorizAlpha(x, y, data, allFrames, width, height, step, params);
```

```
// Referenced __device__ function(s):
__device__ void acgl_glitch_BarsHorizAlpha(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int range = height / 4 > 25 ? height / 4 - 25 : 1;
    int band_h = 25 + gpu_triangle_wave(params.frame_count * 5, range);
    if(band_h < 5) band_h = 5;
    int band = y / band_h;
    float alpha = 0.1f + 0.9f * (0.5f + 0.5f * sinf(params.frame_count * 0.05f));
    int fsel = (int)(gpu_rand(band, params.frame_count / 3, params.seed) * 8.0f) % params.numFrames;
    unsigned char *ref = allFrames[fsel];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(alpha * data[idx + j] + (1.0f - alpha) * ref[idx + j]);
    }
}
```

Filter #865 - acgl_glitch_BlackSquare

AC Glitch Library: random black squares appear across the frame, blocking out portions of the image.

Technique: Random black square block overlay.

```
// Case 865 in unifiedFilterKernel:
acgl_glitch_BlackSquare(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_BlackSquare(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int bx = (x / 32) * 32;
    int by = (y / 32) * 32;
    int inner_x = x - bx;
    int inner_y = y - by;
    if(inner_x >= 1 && inner_x < 31 && inner_y >= 1 && inner_y < 31) {
        float r = gpu_rand(bx, by, params.seed);
        if(r > 0.4f) {
            data[idx] = 0;
            data[idx + 1] = 0;
            data[idx + 2] = 0;
        }
    }
}
```

Filter #866 - acgl_glitch_BlendStuck

AC Glitch Library: blends with a stuck/frozen frame-a single captured frame is continuously blended with the live feed.

Technique: Stuck-frame continuous blend.

```
// Case 866 in unifiedFilterKernel:
acgl_glitch_BlendStuck(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
```

```

__device__ void acgl_glitch_BlendStuck(int x, int y, unsigned char *data, unsigned char **allFrames, int
↳ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = (int)(gpu_rand(y, 0, params.seed + params.frame_count / 50) * params.numFrames) % params.
↳ numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
    }
}

```

Filter #867 - acgl_glitch_ColorDistort

AC Glitch Library: per-channel color distortion-each RGB channel is independently warped/shifted for aggressive color corruption.

Technique: Independent per-channel color distortion.

```

// Case 867 in unifiedFilterKernel:
acgl_glitch_ColorDistort(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_ColorDistort(int x, int y, unsigned char *data, unsigned char **allFrames,
↳ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int pos = params.frame_count % 3;
    int offset = (params.frame_count / 3) % params.numFrames;
    unsigned char *ref = allFrames[offset];
    if(ref) {
        switch(pos) {
            case 0:
                data[idx] ^= ref[idx + 1];
                data[idx + 1] &= ref[idx + 2];
                data[idx + 2] |= ref[idx];
                break;
            case 1:
                data[idx] &= ref[idx + 2];
                data[idx + 1] ^= ref[idx];
                data[idx + 2] |= ref[idx + 1];
                break;
            case 2:
                data[idx] |= ref[idx];
                data[idx + 1] &= ref[idx + 1];
                data[idx + 2] ^= ref[idx + 2];
                break;
        }
    }
}

```

Filter #868 - acgl_glitch_ColorRect

AC Glitch Library: colored rectangles drawn at random positions and sizes, overlaid on the image.

Technique: Random colored rectangle overlay.

```
// Case 868 in unifiedFilterKernel:
acgl_glitch_ColorRect(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_ColorRect(int x, int y, unsigned char *data, unsigned char **allFrames, int
↳ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cy_start = (int)(gpu_rand(0, 0, params.seed) * (height - 1));
    int rect_h = 1 + (int)(gpu_rand(1, 0, params.seed) * (height / 8));
    int offset = (int)(gpu_rand(2, 0, params.seed) * params.numFrames) % params.numFrames;
    int off = (int)(gpu_rand(3, 0, params.seed) * 3.0f) % 3;
    if(y >= cy_start && y < cy_start + rect_h) {
        unsigned char *ref = allFrames[offset];
        if(ref) {
            data[idx] = ref[idx];
            data[idx + 1] = ref[idx + 1];
            data[idx + 2] = ref[idx + 2];
            data[idx + off] = 255;
        }
    }
}
```

Filter #869 - acgl_glitch_ColorOnOff

AC Glitch Library: rapid color/colorless switching-alternates between full color and desaturated/grayscale on each frame or few frames.

Technique: Rapid color/grayscale toggle.

```
// Case 869 in unifiedFilterKernel:
acgl_glitch_ColorOnOff(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_ColorOnOff(int x, int y, unsigned char *data, unsigned char **allFrames, int
↳ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = (y + params.frame_count) % params.numFrames;
    int dir = params.frame_count % 2;
    unsigned char *ref = allFrames[off];
    if(ref) {
        if(dir == 0) {
            if(data[idx] > 150) data[idx] = ref[idx] ^ data[idx];
            if(data[idx + 1] < 150) data[idx + 1] = ref[idx + 1] ^ data[idx + 1];
            if(data[idx + 2] > 215) data[idx + 2] = ref[idx + 2] ^ data[idx + 2];
        } else {
            if(data[idx + 2] > 150) data[idx] = ref[idx] ^ data[idx];
            if(data[idx + 1] < 150) data[idx + 1] = ref[idx + 1] ^ data[idx + 1];
            if(data[idx] > 215) data[idx + 2] = ref[idx + 2] ^ data[idx + 2];
        }
    }
}
```

Filter #870 - acgl_glitch_DEM

AC Glitch Library: DEM (digital elevation map) style visualization-maps brightness to a false-color elevation palette.

Technique: Brightness-to-elevation false color mapping.

```
// Case 870 in unifiedFilterKernel:
acgl_glitch_DEM(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_DEM(int x, int y, unsigned char *data, unsigned char **allFrames, int width,
    ↪ int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cycle = 1 + (int)(gpu_rand(0, 0, params.seed + params.frame_count / 10) * 10.0f);
    if(cycle < 1) cycle = 1;
    int off = (params.frame_count / cycle) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        if((off % 4) == 0) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(data[idx + j] + (data[idx + j] ^ ref[idx + j]));
        } else {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
        }
    }
}
```

Filter #871 - acgl_glitch_FrameMirror

AC Glitch Library: mirrors entire frames from history-alternates between the current frame and various mirrored history frames.

Technique: History-frame mirroring alternation.

```
// Case 871 in unifiedFilterKernel:
acgl_glitch_FrameMirror(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameMirror(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = gpu_triangle_wave(params.frame_count, params.numFrames > 1 ? params.numFrames - 1 : 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        int mirror_x = width - 1 - x;
        if(mirror_x >= 0 && mirror_x < width) {
            int src_idx = y * step + mirror_x * 4;
            for(int j = 0; j < 3; ++j)
                data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[src_idx + j]);
        }
    }
}
```

Filter #872 - acgl_glitch_FramePix

AC Glitch Library: pixelates using frame history-mosaics the image using large blocks sourced from different time points.

Technique: History-frame-sourced block mosaic.

```
// Case 872 in unifiedFilterKernel:
acgl_glitch_FramePix(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FramePix(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int half_h = height / 2;
    if(half_h < 1) half_h = 1;
    int offset_y = (y + params.frame_count) % half_h;
    if(offset_y < 1) offset_y = 1;
    unsigned char b = data[idx], g = data[idx + 1], r = data[idx + 2];
    data[idx] = gpu_wrap_cast((r / 2.0f) * offset_y);
    data[idx + 1] = gpu_wrap_cast((g / 3.0f) * offset_y);
    data[idx + 2] = gpu_wrap_cast((b / 4.0f) * offset_y);
}
```

Filter #873 - acgl_glitch_FrameReverse

AC Glitch Library: reverse frame playback-periodically plays history frames in reverse order, creating backward motion sequences.

Technique: Periodic reverse frame playback.

```
// Case 873 in unifiedFilterKernel:
acgl_glitch_FrameReverse(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameReverse(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = gpu_triangle_wave(params.frame_count, params.numFrames > 1 ? params.numFrames - 1 : 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
    }
}
```

Filter #874 - acgl_glitch_FrameReverse2

AC Glitch Library: reverse frame playback variant 2-longer reverse sequences with blending between forward and reverse.

Technique: Extended reverse playback + blend.

```
// Case 874 in unifiedFilterKernel:
acgl_glitch_FrameReverse2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameReverse2(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = gpu_triangle_wave(params.frame_count * 2, params.numFrames > 1 ? params.numFrames - 1 : 1)
    ↪ ;
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
```

```

        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref[idx + j]);
    }
}

```

Filter #875 - acgl_glitch_FrameReverseNoBlend

AC Glitch Library: hard-cut reverse playback without blending-abrupt switches between forward and reverse for a jarring stutter.

Technique: Hard-cut reverse playback without blend.

```

// Case 875 in unifiedFilterKernel:
acgl_glitch_FrameReverseNoBlend(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameReverseNoBlend(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = gpu_triangle_wave(params.frame_count, params.numFrames > 1 ? params.numFrames - 1 : 1);
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```

Filter #876 - acgl_glitch_FrameSepBand

AC Glitch Library: frame separation bands-horizontal bands alternate between current and different history frames, creating temporal banding.

Technique: Horizontal temporal separation bands.

```

// Case 876 in unifiedFilterKernel:
acgl_glitch_FrameSepBand(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameSepBand(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = (y / 240) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```

Filter #877 - acgl_glitch_FrameSwap

AC Glitch Library: frame swap-randomly swaps the current frame with a history frame for a few frames, creating time-jump flashes.

Technique: Random history-frame swap.


```
// Case 877 in unifiedFilterKernel:
acgl_glitch_FrameSwap(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameSwap(int x, int y, unsigned char *data, unsigned char **allFrames, int
↳ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = gpu_triangle_wave(params.frame_count, params.numFrames > 1 ? params.numFrames - 1 : 1);
    bool on = (params.frame_count % 2) == 0;
    if(on) {
        unsigned char *ref = allFrames[off % params.numFrames];
        if(ref) {
            for(int j = 0; j < 3; ++j)
                data[idx + j] = ref[idx + j];
        }
    }
}
```

Filter #878 - acgl_glitch_FrameXBlend

AC Glitch Library: horizontal frame blend-blends current frame with history using a horizontal gradient mask (more history on one side).

Technique: Horizontal gradient temporal blend.

```
// Case 878 in unifiedFilterKernel:
acgl_glitch_FrameXBlend(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameXBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
↳ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset_x = (y + params.frame_count) % width;
    int src_idx = y * step + offset_x * 4;
    for(int j = 0; j < 3; ++j)
        data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
}
```

Filter #879 - acgl_glitch_FrameXBlendXor

AC Glitch Library: horizontal frame XOR blend-combines horizontal gradient blending with XOR for digital artifacts along the gradient.

Technique: Horizontal gradient XOR blend.

```
// Case 879 in unifiedFilterKernel:
acgl_glitch_FrameXBlendXor(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameXBlendXor(int x, int y, unsigned char *data, unsigned char **allFrames,
↳ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset_x = (y + params.frame_count) % width;
    int src_idx = y * step + offset_x * 4;
    for(int j = 0; j < 3; ++j)
        data[idx + j] ^= data[src_idx + j];
}
```

Filter #880 - acgl_glitch_FrameYBlend

AC Glitch Library: vertical frame blend-blends current with history using a vertical gradient mask.

Technique: Vertical gradient temporal blend.

```
// Case 880 in unifiedFilterKernel:
acgl_glitch_FrameYBlend(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameYBlend(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset_y = (x + params.frame_count) % height;
    int src_idx = offset_y * step + x * 4;
    for(int j = 0; j < 3; ++j)
        data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * data[src_idx + j]);
}
```

Filter #881 - acgl_glitch_FrameYBlendXor

AC Glitch Library: vertical frame XOR blend-vertical gradient with XOR artifacts.

Technique: Vertical gradient XOR blend.

```
// Case 881 in unifiedFilterKernel:
acgl_glitch_FrameYBlendXor(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameYBlendXor(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int offset_y = (x + params.frame_count) % height;
    int src_idx = offset_y * step + x * 4;
    for(int j = 0; j < 3; ++j)
        data[idx + j] ^= data[src_idx + j];
}
```

Filter #882 - acgl_glitch_AddMulXor

AC Glitch Library: combined add, multiply, and XOR operations between current and history frames-triple-operation digital manipulation.

Technique: Add + multiply + XOR triple blend.

```
// Case 882 in unifiedFilterKernel:
acgl_glitch_AddMulXor(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_AddMulXor(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
```

```

int off = y % params.numFrames;
int r_ch = params.frame_count % 3;
unsigned char *ref = allFrames[off];
if(ref) {
    unsigned char color_val = gpu_wrap_cast(0.5f * data[idx + r_ch] + 0.5f * ref[idx + r_ch]);
    int div = width / 4;
    if(div < 1) div = 1;
    if((x % div) == 0)
        color_val ^= ref[idx + r_ch];
    for(int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(0.5f * data[idx + j] + 0.5f * color_val);
}
}

```

Filter #883 - acgl_glitch_AddXor

AC Glitch Library: addition followed by XOR between frames-adds then XORs for layered digital artifacts.

Technique: Add then XOR dual operation.

```

// Case 883 in unifiedFilterKernel:
acgl_glitch_AddXor(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_AddXor(int x, int y, unsigned char *data, unsigned char **allFrames, int
↳ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = ((x + y) / 4) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        unsigned char c0 = ref[idx] ^ data[idx];
        unsigned char c1 = ref[idx + 1] ^ data[idx + 1];
        unsigned char c2 = ref[idx + 2] ^ data[idx + 2];
        data[idx] = (unsigned char)(0.5f * data[idx] + 0.5f * c0);
        data[idx + 1] = (unsigned char)(0.5f * data[idx + 1] + 0.5f * c1);
        data[idx + 2] = (unsigned char)(0.5f * data[idx + 2] + 0.5f * c2);
    }
}

```

Filter #884 - acgl_glitch_ColorBarsRand

AC Glitch Library: random color bars-color bar positions, widths, and colors randomize each cycle, creating chaotic bar patterns.

Technique: Randomized color bar pattern.

```

// Case 884 in unifiedFilterKernel:
acgl_glitch_ColorBarsRand(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_ColorBarsRand(int x, int y, unsigned char *data, unsigned char **allFrames,
↳ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool strobe = (params.frame_count % 2) == 0;
    int offset = gpu_triangle_wave(params.frame_count / 6, 2);
    int off_x = (int)(gpu_rand(x, y, params.seed) * width) % width;

```

```

int off_b = (int)(gpu_rand(x + 2, y, params.seed) * width) % width;
int f1 = 1 % params.numFrames, f4 = 4 % params.numFrames, f7 = (params.numFrames - 1) % params.
↳ numFrames;
unsigned char *r1 = allFrames[f1], *r4 = allFrames[f4], *r7 = allFrames[f7];
if(r1 && r4 && r7) {
    unsigned char pix[3][3];
    int s0 = y * step + off_x * 4;
    int s2 = y * step + off_b * 4;
    pix[0][0] = r1[s0]; pix[0][1] = r1[s0 + 1]; pix[0][2] = r1[s0 + 2];
    pix[1][0] = r4[idx]; pix[1][1] = r4[idx + 1]; pix[1][2] = r4[idx + 2];
    pix[2][0] = r7[s2]; pix[2][1] = r7[s2 + 1]; pix[2][2] = r7[s2 + 2];
    for(int j = 0; j < 3; ++j) {
        if(!strobe) data[idx + j] = pix[offset][j];
        else data[idx + j] = pix[j][j];
    }
}
}

```

Filter #885 - acgl_glitch_ColorBarsX

AC Glitch Library: vertical color bars with animated scrolling-bars scroll horizontally across the frame.

Technique: Scrolling vertical color bars.

```

// Case 885 in unifiedFilterKernel:
acgl_glitch_ColorBarsX(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_ColorBarsX(int x, int y, unsigned char *data, unsigned char **allFrames, int
↳ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool strobe = (params.frame_count % 2) == 0;
    int offset = gpu_triangle_wave(params.frame_count / 6, 2);
    int off_x = ((x + params.frame_count) * 7) % width;
    int off_g = ((x + params.frame_count) * 13) % width;
    int off_b = ((x + params.frame_count) * 19) % width;
    int f1 = 1 % params.numFrames, f4 = 4 % params.numFrames, f7 = (params.numFrames - 1) % params.
↳ numFrames;
    unsigned char *r1 = allFrames[f1], *r4 = allFrames[f4], *r7 = allFrames[f7];
    if(r1 && r4 && r7) {
        unsigned char pix[3][3];
        pix[0][0] = r1[y * step + off_x * 4]; pix[0][1] = r1[y * step + off_x * 4 + 1]; pix[0][2] = r1[y
↳ * step + off_x * 4 + 2];
        pix[1][0] = r4[y * step + off_g * 4]; pix[1][1] = r4[y * step + off_g * 4 + 1]; pix[1][2] = r4[y
↳ * step + off_g * 4 + 2];
        pix[2][0] = r7[y * step + off_b * 4]; pix[2][1] = r7[y * step + off_b * 4 + 1]; pix[2][2] = r7[y
↳ * step + off_b * 4 + 2];
        for(int j = 0; j < 3; ++j) {
            if(!strobe) data[idx + j] = pix[offset][j];
            else data[idx + j] = pix[j][j];
        }
    }
}
}

```

Filter #886 - acgl_glitch_ColorBarsY

AC Glitch Library: horizontal color bars with animated scrolling-bars scroll vertically.

Technique: Scrolling horizontal color bars.

```
// Case 886 in unifiedFilterKernel:
acgl_glitch_ColorBarsY(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_ColorBarsY(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool strobe = (params.frame_count % 2) == 0;
    int offset = gpu_triangle_wave(params.frame_count / 6, 2);
    int off_r = ((y + params.frame_count) * 7) % height;
    int off_g = ((y + params.frame_count) * 13) % height;
    int f1 = 1 % params.numFrames, f4 = 4 % params.numFrames, f7 = (params.numFrames - 1) % params.
    ↪ numFrames;
    unsigned char *r1 = allFrames[f1], *r4 = allFrames[f4], *r7 = allFrames[f7];
    if(r1 && r4 && r7) {
        unsigned char pix[3][3];
        pix[0][0] = r1[off_r * step + x * 4]; pix[0][1] = r1[off_r * step + x * 4 + 1]; pix[0][2] = r1[
    ↪ off_r * step + x * 4 + 2];
        pix[1][0] = r4[off_g * step + x * 4]; pix[1][1] = r4[off_g * step + x * 4 + 1]; pix[1][2] = r4[
    ↪ off_g * step + x * 4 + 2];
        pix[2][0] = r7[idx]; pix[2][1] = r7[idx + 1]; pix[2][2] = r7[idx + 2];
        for(int j = 0; j < 3; ++j) {
            if(!strobe) data[idx + j] = pix[offset][j];
            else data[idx + j] = pix[j][j];
        }
    }
}
```

Filter #887 - acgl_glitch_ColorShiftY

AC Glitch Library: vertical color shifting-color channels shift vertically at different rates, creating vertical chromatic separation.

Technique: Vertical per-channel offset shift.

```
// Case 887 in unifiedFilterKernel:
acgl_glitch_ColorShiftY(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_ColorShiftY(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool strobe = (params.frame_count % 2) == 0;
    int offset = gpu_triangle_wave(params.frame_count / 6, 2);
    int off_r = ((y * 3 + x + params.frame_count) * 7) % height;
    int off_g = ((y * 5 + x + params.frame_count) * 11) % height;
    int f1 = 1 % params.numFrames, f4 = 4 % params.numFrames, f7 = (params.numFrames - 1) % params.
    ↪ numFrames;
    unsigned char *r1 = allFrames[f1], *r4 = allFrames[f4], *r7 = allFrames[f7];
    if(r1 && r4 && r7) {
        unsigned char pix[3][3];
        pix[0][0] = r1[off_r * step + x * 4]; pix[0][1] = r1[off_r * step + x * 4 + 1]; pix[0][2] = r1[
    ↪ off_r * step + x * 4 + 2];
        pix[1][0] = r4[off_g * step + x * 4]; pix[1][1] = r4[off_g * step + x * 4 + 1]; pix[1][2] = r4[
    ↪ off_g * step + x * 4 + 2];
    }
```

```

        pix[2][0] = r7[idx]; pix[2][1] = r7[idx + 1]; pix[2][2] = r7[idx + 2];
        for(int j = 0; j < 3; ++j) {
            if(!strobe) data[idx + j] = pix[offset][j];
            else data[idx + j] = pix[j][j];
        }
    }
}

```

Filter #888 - acgl_glitch_LineCollectionRGB

AC Glitch Library: RGB line collection-draws horizontal lines where each line is a single color channel (red, green, or blue) from the image.

Technique: Single-channel horizontal line extraction.

```

// Case 888 in unifiedFilterKernel:
acgl_glitch_LineCollectionRGB(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_LineCollectionRGB(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int r_count = 1 + (int)(gpu_rand(y, params.frame_count, params.seed) * 50.0f);
    if(r_count < 1) r_count = 1;
    int rgb = (params.frame_count + y / r_count) % 3;
    int segment = x / r_count;
    int fsel = (int)(gpu_rand(y, segment, params.seed + params.frame_count) * params.numFrames) % params
    ↪ .numFrames;
    unsigned char *ref = allFrames[fsel];
    if(ref) {
        data[idx + rgb] = ref[idx + rgb];
    }
}

```

Filter #889 - acgl_glitch_FrameX2

AC Glitch Library: double-frame blend-blends two history frames at once for a deeper temporal composite.

Technique: Dual history frame blend.

```

// Case 889 in unifiedFilterKernel:
acgl_glitch_FrameX2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_FrameX2(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int bx = x / 16, by = y / 16;
    int off = ((bx + by + params.frame_count) % params.numFrames);
    float r = gpu_rand(bx, by, params.seed);
    if(r > 0.5f) {
        int stretch = off + 1;
        if(stretch < 1) stretch = 1;
        int off_x = gpu_AC_GetFX(width - 1, x, width * stretch);
        unsigned char *ref = allFrames[off % params.numFrames];
        if(ref && off_x >= 0 && off_x < width) {

```

```

        int src_idx = y * step + off_x * 4;
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[src_idx + j];
    }
}

```

Filter #890 - acgl_glitch_NewBars

AC Glitch Library: new-style color bars-redesigned bar pattern with modern width and color choices.

Technique: Modern styled color bar pattern.

```

// Case 890 in unifiedFilterKernel:
acgl_glitch_NewBars(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_NewBars(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int nf = params.numFrames > 0 ? params.numFrames : 1;
    int max_band = height / nf;
    if(max_band < 1) max_band = 1;
    int off = (y / max_band) % nf;
    unsigned char *ref = allFrames[off];
    if(ref) {
        int rnd = (int)(gpu_rand(x, y, params.seed) * 3.0f) % 3;
        unsigned char b = data[idx], g = data[idx + 1], r = data[idx + 2];
        unsigned char rb = ref[idx], rg = ref[idx + 1], rr = ref[idx + 2];
        unsigned char ob, og, or_;
        switch(rnd) {
            case 0: ob = b & rb; og = g | rg; or_ = r ^ rr; break;
            case 1: ob = b | rb; og = g ^ rg; or_ = r & rr; break;
            default: ob = b ^ rb; og = g & rg; or_ = r | rr; break;
        }
        data[idx] = (unsigned char)(0.5f * b + 0.5f * ob);
        data[idx + 1] = (unsigned char)(0.5f * g + 0.5f * og);
        data[idx + 2] = (unsigned char)(0.5f * r + 0.5f * or_);
    }
}

```

Filter #891 - acgl_glitch_NewBars2

AC Glitch Library: new bars variant 2-alternative bar layout with different spacing and color assignment.

Technique: Alternative modern bar layout.

```

// Case 891 in unifiedFilterKernel:
acgl_glitch_NewBars2(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_NewBars2(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    for(int j = 0; j < 3; ++j) {

```

```

        int off = ((y * width + x) * 3 + j + params.frame_count) % params.numFrames;
        unsigned char *ref = allFrames[off];
        if(ref)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
    }
}

```

Filter #892 - acgl_glitch_NewBlendLines

AC Glitch Library: new blend lines-horizontal lines sourced from blended history frames, creating temporal line composites.

Technique: History-blended horizontal line rendering.

```

// Case 892 in unifiedFilterKernel:
acgl_glitch_NewBlendLines(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_NewBlendLines(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int mx = 1 + (int)(gpu_rand(0, 0, params.seed + params.frame_count) * params.numFrames);
    if(mx < 1) mx = 1;
    if(mx > params.numFrames) mx = params.numFrames;
    int off = y % mx;
    unsigned char *ref = allFrames[off % params.numFrames];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
    }
}

```

Filter #893 - acgl_glitch_NewLines

AC Glitch Library: new lines-horizontal displacement lines with updated generation algorithm for cleaner glitch lines.

Technique: Updated algorithm horizontal glitch lines.

```

// Case 893 in unifiedFilterKernel:
acgl_glitch_NewLines(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_NewLines(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int off = (int)(gpu_rand(y, 0, params.seed + params.frame_count) * params.numFrames) % params.
    ↪ numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}

```


Filter #894 - acgl_glitch_NewOne

AC Glitch Library: new experimental glitch-a novel combined glitch technique unique to this library.

Technique: Experimental combined glitch.

```
// Case 894 in unifiedFilterKernel:
acgl_glitch_NewOne(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_NewOne(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int fsel = (int)(gpu_rand(0, 0, params.seed + params.frame_count) * params.numFrames) % params.
    ↪ numFrames;
    int rnd_op = (int)(gpu_rand(1, 0, params.seed + params.frame_count) * 3.0f) % 3;
    unsigned char *ref = allFrames[fsel];
    if(ref) {
        float chance = gpu_rand(x, y, params.seed);
        if(chance < 0.077f) {
            for(int j = 0; j < 3; ++j) {
                switch(rnd_op) {
                    case 0: data[idx + j] &= ref[idx + j]; break;
                    case 1: data[idx + j] ^= ref[idx + j]; break;
                    case 2: data[idx + j] |= ref[idx + j]; break;
                }
            }
        }
    }
}
```

Filter #895 - acgl_glitch_StrobeCycle

AC Glitch Library: strobe cycle-cycles through multiple strobe patterns (white, black, color, invert) in a repeating sequence.

Technique: Multi-pattern strobe cycling.

```
// Case 895 in unifiedFilterKernel:
acgl_glitch_StrobeCycle(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_StrobeCycle(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool strobe = (params.frame_count % 2) == 0;
    int offset = (params.frame_count / 10) % 3;
    int f1 = 1 % params.numFrames, f4 = 4 % params.numFrames, f7 = (params.numFrames - 1) % params.
    ↪ numFrames;
    unsigned char *r1 = allFrames[f1], *r4 = allFrames[f4], *r7 = allFrames[f7];
    if(r1 && r4 && r7) {
        unsigned char pix[3][3];
        pix[0][0] = r1[idx]; pix[0][1] = r1[idx + 1]; pix[0][2] = r1[idx + 2];
        pix[1][0] = r4[idx]; pix[1][1] = r4[idx + 1]; pix[1][2] = r4[idx + 2];
        pix[2][0] = r7[idx]; pix[2][1] = r7[idx + 1]; pix[2][2] = r7[idx + 2];
        for(int j = 0; j < 3; ++j) {
            if(!strobe) data[idx + j] = pix[offset][j];
            else data[idx + j] = pix[j][j];
        }
    }
}
```

```
}  
}
```

Filter #896 - acgl_glitch_StuckFrame2

AC Glitch Library: stuck frame variant 2-holds a frame longer with gradual blend back to live, smoother than the original stuck frame.

Technique: Extended stuck frame with gradual resume.

```
// Case 896 in unifiedFilterKernel:  
acgl_glitch_StuckFrame2(x, y, data, allFrames, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void acgl_glitch_StuckFrame2(int x, int y, unsigned char *data, unsigned char **allFrames,  
    ↪ int width, int height, size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    int step_arr[] = {1, 2, 2, 3, 3, 3};  
    int arr_off = gpu_triangle_wave(params.frame_count, 5);  
    int step_val = step_arr[arr_off];  
    int pixel_id = y * width + x;  
    int period = params.numFrames > 1 ? params.numFrames - 1 : 1;  
    int off = gpu_triangle_wave(pixel_id * step_val, period);  
    unsigned char *ref = allFrames[off % params.numFrames];  
    if(ref) {  
        for(int j = 0; j < 3; ++j)  
            data[idx + j] = ref[idx + j];  
    }  
}
```

Filter #897 - acgl_glitch_StuckLine

AC Glitch Library: stuck line-individual horizontal lines get stuck (frozen) at random intervals while the rest of the image updates.

Technique: Random per-line freeze.

```
// Case 897 in unifiedFilterKernel:  
acgl_glitch_StuckLine(x, y, data, allFrames, width, height, step, params);  
  
// Referenced __device__ function(s):  
__device__ void acgl_glitch_StuckLine(int x, int y, unsigned char *data, unsigned char **allFrames, int  
    ↪ width, int height, size_t step, const FilterParams &params) {  
    int idx = y * step + x * 4;  
    int off = y % params.numFrames;  
    unsigned char *ref = allFrames[off];  
    if(ref) {  
        for(int j = 0; j < 3; ++j)  
            data[idx + j] = ref[idx + j];  
    }  
}
```

Filter #898 - acgl_glitch_StuckRow

AC Glitch Library: stuck row-entire rows of pixels freeze for varying durations, creating a row-by-row temporal desync.

Technique: Per-row variable-duration freeze.

```
// Case 898 in unifiedFilterKernel:
acgl_glitch_StuckRow(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_StuckRow(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int stride = 1 + (params.frame_count % 3);
    int off = (y * stride) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #899 - acgl_glitch_StuckRowLine

AC Glitch Library: stuck row + line combination-both individual lines and row groups freeze independently.

Technique: Combined line + row freeze.

```
// Case 899 in unifiedFilterKernel:
acgl_glitch_StuckRowLine(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_StuckRowLine(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int rsize = 25 + (int)(gpu_rand(y / 25, 0, params.seed + params.frame_count) * 25.0f);
    if(rsize < 1) rsize = 1;
    int segment = y / rsize;
    int off = (int)(gpu_rand(segment, 0, params.seed + params.frame_count) * params.numFrames) % params.
    ↪ numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #900 - acgl_glitch_SepBlocks

AC Glitch Library: separated blocks-block tiles separate with visible gaps between them, creating a tiled grid with spacing.

Technique: Block tile separation with gap.

```
// Case 900 in unifiedFilterKernel:
acgl_glitch_SepBlocks(x, y, data, allFrames, width, height, step, params);
```

```

// Referenced __device__ function(s):
__device__ void acgl_glitch_SepBlocks(int x, int y, unsigned char *data, unsigned char **allFrames, int
↳ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    bool strobe = (params.frame_count % 2) == 0;
    int offset = gpu_triangle_wave(params.frame_count / 6, params.numFrames > 1 ? params.numFrames - 1 :
↳ 1);
    if(strobe) {
        if(y < height / 4) {
            unsigned char *ref = allFrames[offset % params.numFrames];
            if(ref) {
                for(int j = 0; j < 3; ++j)
                    data[idx + j] = ref[idx + j];
            }
        }
        int start_y = (int)(gpu_rand(0, 0, params.seed + params.frame_count) * height);
        if(y >= start_y) {
            int f7 = (params.numFrames - 1) % params.numFrames;
            unsigned char *ref = allFrames[f7];
            if(ref) {
                for(int j = 0; j < 3; ++j)
                    data[idx + j] = ref[idx + j];
            }
        }
    }
}

```

Filter #901 - acgl_glitch_Plug1

AC Glitch Library: plugin slot 1-a general-purpose glitch filter template used as an extensible plugin point.

Technique: General-purpose glitch plugin.

```

// Case 901 in unifiedFilterKernel:
acgl_glitch_Plug1(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_Plug1(int x, int y, unsigned char *data, unsigned char **allFrames, int
↳ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    data[idx] = (unsigned char)(gpu_rand(x, y, params.seed) * 255.0f);
    data[idx + 1] = (unsigned char)(gpu_rand(x + 1, y, params.seed) * 255.0f);
    data[idx + 2] = (unsigned char)(gpu_rand(x, y + 1, params.seed) * 255.0f);
}

```

Filter #902 - acgl_glitch_OppositeDir

AC Glitch Library: opposite direction-alternating rows/blocks move in opposite directions, creating a push-pull shearing effect.

Technique: Alternating opposite-direction row displacement.

```

// Case 902 in unifiedFilterKernel:
acgl_glitch_OppositeDir(x, y, data, allFrames, width, height, step, params);

```

```
// Referenced __device__ function(s):
__device__ void acgl_glitch_OppositeDir(int x, int y, unsigned char *data, unsigned char **allFrames,
    ↪ int width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    int cx = 50 + gpu_triangle_wave(params.frame_count, 50);
    if(cx < 1) cx = 1;
    int off_fwd = (y / cx) % params.numFrames;
    unsigned char *ref_fwd = allFrames[off_fwd];
    if(ref_fwd) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref_fwd[idx + j]);
    }
    int off_rev = (params.numFrames - 1 - off_fwd) % params.numFrames;
    if(off_rev < 0) off_rev = 0;
    unsigned char *ref_rev = allFrames[off_rev];
    if(ref_rev) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = gpu_wrap_cast(0.5f * data[idx + j] + 0.5f * ref_rev[idx + j]);
    }
}
```

Filter #903 - acgl_glitch_OffStuck

AC Glitch Library: offset stuck-combines spatial offset with stuck-frame, showing a frozen frame at a spatial offset position.

Technique: Spatially-offset stuck frame.

```
// Case 903 in unifiedFilterKernel:
acgl_glitch_OffStuck(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_OffStuck(int x, int y, unsigned char *data, unsigned char **allFrames, int
    ↪ width, int height, size_t step, const FilterParams &params) {
    int idx = y * step + x * 4;
    float r = gpu_rand(x / 8, y / 8, params.seed + params.frame_count);
    int off = (int)(r * params.numFrames) % params.numFrames;
    unsigned char *ref = allFrames[off];
    if(ref) {
        for(int j = 0; j < 3; ++j)
            data[idx + j] = ref[idx + j];
    }
}
```

Filter #904 - acgl_glitch_NewVarBlendLines

AC Glitch Library: new variable blend lines-horizontal lines blended from various history frames with variable blend ratios and widths.

Technique: Variable-ratio history-blended horizontal lines.

```
// Case 904 in unifiedFilterKernel:
acgl_glitch_NewVarBlendLines(x, y, data, allFrames, width, height, step, params);

// Referenced __device__ function(s):
__device__ void acgl_glitch_NewVarBlendLines(int x, int y, unsigned char *data, unsigned char **
    ↪ allFrames, int width, int height, size_t step, const FilterParams &params) {
```

```

int idx = y * step + x * 4;
int r = 1 + (int)(gpu_rand(0, 0, params.seed + params.frame_count) * 64.0f);
if(r < 1) r = 1;
int off = params.numFrames - 1 - (y / r) % params.numFrames;
if(off < 0) off = 0;
if(off >= params.numFrames) off = params.numFrames - 1;
unsigned char *ref = allFrames[off];
if(ref) {
    for(int j = 0; j < 3; ++j)
        data[idx + j] = (unsigned char)(data[idx + j] * 0.5f + ref[idx + j] * 0.5f);
}
}

```

Part II - GLSL Shaders

This section documents all 1097 GLSL fragment shaders in the shaders/ directory. These shaders are designed for ACMX2 (AcidCam MX2), a real-time GPU-accelerated video effects processor. Each shader processes the input texture in real time - many respond to time, mouse position, and audio amplitude.

Vertex Shader

The vertex shader is shared by all fragment shaders in ACMX2. It transforms each vertex position by the model-view and projection matrices and passes through the texture coordinate (tc) to the fragment stage.

```

#version 410 core

layout (location=0) in vec3 pos;
layout (location=1) in vec2 texCoord;

out vec2 tc;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main() {
    gl_Position = proj_matrix * mv_matrix * vec4(pos,1.0);
    tc = texCoord;
}

```

Other Effects (225 shaders)

Shader #13 - addup cos

File: addup_cos.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;

```

```

uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 source = color;
    float val = cos((color[0]+color[1]+color[2])*alpha);
    for(int q = 0; q < 3; ++q)
        color[q] = color[q] * (val * color[q]/0.3);
    color = (0.5 * color) + (0.7 * source);
}

```

Shader #26 - alpha diamond

File: alpha_diamond.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc;
    ivec2 coords = ivec2(uv * iResolution);
    vec4 origColor = texture(samp, uv);
    float pos = mod(time_f, 7.0) + 1.0;
    int x = coords.x;
    int y = coords.y;
    vec3 newColor = origColor.rgb;
    if ((x % 2) == 0) {
        if ((y % 2) == 0) {
            newColor.r = (1.0 - pos * origColor.r);
            newColor.b = (float(x + y) * pos) / 255.0;
        } else {
            newColor.r = (pos * origColor.r - float(y)) / 255.0;
            newColor.b = (float(x - y) * pos) / 255.0;
        }
    } else {
        if ((y % 2) == 0) {
            newColor.r = (pos * origColor.r - float(x)) / 255.0;
            newColor.b = (float(x - y) * pos) / 255.0;
        } else {
            newColor.r = (pos * origColor.r - float(y)) / 255.0;
            newColor.b = (float(x + y) * pos) / 255.0;
        }
    }
}

```

```

    }
    float temp = newColor.r;
    newColor.r = newColor.b;
    newColor.b = temp;
    vec3 finalColor = (sin(time_f) > 0.0) ? vec3(1.0) - newColor : newColor;
    color = sin(vec4(finalColor, 1.0) * time_f);
    color.a = 1.0;
}

```

Shader #27 - alpha sin os

File: alpha_sin_os.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float value, float range) {
    float modValue = mod(value, 2.0 * range);
    return range - abs(modValue - range);
}
void main() {
    vec2 uv = tc;
    vec4 pixelColor = texture(samp, uv);
    float scale = 0.5 + 0.5 * pingPong(time_f, 8.0) + 1.5;
    vec4 scaledColor = mod(pixelColor * scale, 1.0);
    color = scaledColor;
    color.a = 1.0;
}

```

Shader #32 - apart

File: apart.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec2 pseudoRandomDirection(float time) {
    return normalize(vec2(sin(time * 1.3), cos(time * 1.7)));
}
void main(void) {
    vec2 centeredCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float dist = length(centeredCoord);

```



```

    vec2 spiralDir = pseudoRandomDirection(time_f + dist * 5.0);    float angle = atan(centeredCoord.y,
    ↪ centeredCoord.x) + time_f * 2.0;
    float radius = dist * (1.0 + 0.1 * sin(time_f * 3.0 + dist * 10.0));
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * radius * 0.5 + spiralDir * 0.1 * sin(time_f * 2.0)
    ↪ ;
    spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
    color = texture(samp, spiralCoord);
}

```

Shader #42 - aura3

File: aura3.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
void main(void){
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float radius = mix(0.8, 1.2, 0.5 + 0.5 * sin(time_f * 1.3));
    radius *= 2.0;
    float r = length(uv);
    float glow = smoothstep(radius, radius - 0.25, r);
    vec4 base = texture(samp, tc);
    vec3 pink = vec3(1.0, 0.2, 0.6);
    float pulse = 0.5 + 0.5 * sin(time_f * 3.0);
    vec3 aura = pink * glow * pulse * 1.5;
    vec3 blended = mix(base.rgb, aura + base.rgb * 0.6, glow);
    color = vec4(blended, base.a);
}

```

Shader #43 - aura4

File: aura4.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;

```

```

uniform float time_f;
const float PI = 3.1415926535897932384626433832795;
float hash(vec2 p){return fract(sin(dot(p,vec2(127.1,311.7)))*43758.5453123);}
mat2 rot(float a){float s=sin(a),c=cos(a);return mat2(c,-s,s,c);}
float sdEquilateralTriangle(vec2 p){
    const float k=1.7320508075688772;
    p.x=abs(p.x)-1.0;
    p.y=p.y+1.0/k;
    if(p.x+k*p.y>0.0)p=vec2(p.x-k*p.y,-k*p.x-p.y)/2.0;
    p.x-=clamp(p.x,-2.0,0.0);
    return -length(p)*sign(p.y);
}
float triTile(vec2 p, float scale, float edge){
    p*=scale;
    vec2 g=floor(p);
    vec2 f=fract(p)-0.5;
    f*=rot(0.15*(sin(time_f*0.6)+sin(dot(g,vec2(1.7,2.3)))));
    float d=abs(sdEquilateralTriangle(f*2.0));
    float m=smoothstep(edge,0.0,d);
    return m;
}
float triFractal(vec2 p){
    float t=0.0;
    float s=3.0;
    float e=0.18;
    for(int i=0;i<4;i++){
        t=max(t,triTile(p,s,e));
        p*=rot(0.35);
        s*=1.9;
        e*=0.8;
    }
    return clamp(t,0.0,1.0);
}
void main(void){
    vec2 uv=tc*2.0-1.0;
    uv.x*=iResolution.x/iResolution.y;
    float radius=mix(0.8,1.2,0.5+0.5*sin(time_f*1.3))*2.0;
    float r=length(uv);
    float glow=smoothstep(radius,radius-0.25,r);
    vec4 base=texture(samp,tc);
    vec3 pink=vec3(1.0,0.2,0.6);
    float pulse=0.5+0.5*sin(time_f*3.0);
    float auraAmp=1.4*pulse;
    float triMask=triFractal(uv*1.2+vec2(0.15*sin(time_f*0.7),0.12*cos(time_f*0.55)));
    float lines=pow(triMask,1.5);
    float flakes=smoothstep(0.6,1.0,triMask);
    vec3 aura=pink*(0.55+0.45*flakes)*glow*auraAmp;
    vec3 linesGlow=pink*(0.35+0.65*pulse)*lines*glow;
    vec3 col=base.rgb;
    col=mix(col, col + aura, glow*0.85);
    col+=linesGlow*0.75;
    color=vec4(col, base.a);
}

```

Shader #53 - balloon

File: balloon.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float dist = length(normCoord);
    float cycleDuration = 10.0;
    float cyclePhase = mod(time_f, cycleDuration) / cycleDuration;
    float effectPhase = cyclePhase < 0.5 ? cyclePhase * 2.0 : (1.0 - cyclePhase) * 2.0;
    float effectRadius = 0.75 * sin(effectPhase * 3.14159);
    if (dist < effectRadius) {
        float effectStrength = 10.0 - (dist / effectRadius);
        normCoord *= 1.0 + effectStrength * (effectRadius - dist) * sin(effectPhase * 3.14159);
    }
    vec2 finalCoord = (normCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
    color = texture(samp, finalCoord);
}
```

Shader #60 - bi cycle

File: bi_cycle.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    int numSpokes = 32;
    float spokeWidth = 0.05;
    float spokeAngle = radians(360.0 / float(numSpokes));
    float spokePattern = mod(angle + time_f * 2.0, spokeAngle) - spokeAngle / 2.0;
    float spoke = smoothstep(-spokeWidth, 0.0, spokePattern) - smoothstep(0.0, spokeWidth, spokePattern)
    ↵ ;
    float centerGlow = exp(-radius * 20.0);
    vec4 texColor = texture(samp, uv * 0.5 + 0.5);
    float glow = max(spoke, centerGlow);
    color = texColor * vec4(glow, glow, glow, 1.0);
}
```

Shader #63 - blackhole

File: blackhole.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv *= iResolution.x / iResolution.y;
    float dist = length(uv);
    float timeEffect = mod(time_f * 0.1, 3.0);
    float pullIntensity = pow(dist, 1.0 + 4.0 * timeEffect);
    vec2 pulledUV = uv * pullIntensity;
    pulledUV = (pulledUV / (iResolution.x / iResolution.y)) * 0.5 + 0.5;
    vec4 texColor = texture(samp, sin(pulledUV * timeEffect));
    color = vec4(texColor.rgb, 1.0);
}
```

Shader #64 - blank

File: blank.glsl

Renders a fractal pattern using iterative computation (100 iterations). Random number generator (hash-based, deterministic) Generates values in the range [0.0, 1.0] Draw Julia fractal at a given position with an animated seed

Technique: Iterative fractal computation with complex-plane mapping - Pseudo-random noise via fract(sin(dot())) hash function - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
float rand(vec2 pos) {
    return fract(sin(dot(pos, vec2(12.9898, 78.233))) * 43758.5453);
}
vec4 drawRandomJulia(vec2 uv, vec2 center, vec2 seed) {
    uv = (uv - center) * 3.0;
    vec2 z = uv;
    const int MAX_ITER = 100;
    float brightness = 0.0;
    for (int i = 0; i < MAX_ITER; i++) {
        z = vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + seed;
        if (dot(z, z) > 4.0) {
            brightness = float(i) / float(MAX_ITER);
            break;
        }
    }
    vec3 col = vec3(sin(brightness * 6.2831), brightness, 1.0 - brightness);
    return vec4(col, brightness);
}
void main(void) {
    color = texture(samp, tc);
}
```

```

    vec2 uv = 2.0 * tc - 1.0;
    float cycleDuration = 5.0;
    float cycleTime = mod(time_f, cycleDuration);
    float cycleIndex = floor(time_f / cycleDuration);
    vec2 randomPos = vec2(rand(vec2(cycleIndex, 1.0)), rand(vec2(cycleIndex, 2.0))) * 1.6 - 0.8;
    vec2 juliaSeed = vec2(sin(time_f * 0.3), cos(time_f * 0.3)) * 0.5;
    float alpha = smoothstep(0.0, 1.0, cycleTime) * smoothstep(cycleDuration, cycleDuration - 1.0,
    ↪ cycleTime);
    vec4 juliaColor = drawRandomJulia(uv, randomPos, juliaSeed);
    juliaColor.a *= alpha;
    color = mix(color, juliaColor, juliaColor.a);
}

```

Shader #67 - blue

File: blue.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] /= 2;
    color[1] /= 2;
    color[2] /= 2;
    color[0] = sin(color[0]*alpha_b);
    color[1] = (color[1]*alpha_g);
    color[2] = cos(color[2]*alpha_r);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

```
}
```

Shader #71 - blue strobe

File: blue_strobe.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = color[0]*(random_value[0]/255);
    color[1] = color[1]*(timeval * 0.05812);
    color[2] = color[2]*(alpha_r * timeval * (random_value[1]/255));
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #76 - bowl by time

File: bowl-by_time.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float h1(float n){ return fract(sin(n)*43758.5453123); }
vec2 h2(float n){ return fract(sin(vec2(n, n+1.0))*vec2(43758.5453,22578.1459)); }
void main(void){
    float rate = 0.6;
    float t = time_f * rate;
    float t0 = floor(t);
    float a = fract(t);
    float w = a*a*(3.0-2.0*a);
    vec2 p0 = vec2(0.15) + h2(t0)*0.7;
    vec2 p1 = vec2(0.15) + h2(t0+1.0)*0.7;
    vec2 center = mix(p0, p1, w);
    vec2 p = tc - center;
    float r = length(p);
    float ang = atan(p.y, p.x);
    float swirl = 2.2 + 0.8*sin(time_f*0.35);
    float spin = 0.6*sin(time_f*0.2);
    ang += swirl * r + spin;
    float bend = 0.35;
    float rp = r + bend * r * r;
    vec2 uv = center + vec2(cos(ang), sin(ang)) * rp;
    uv += 0.02 * vec2(sin((tc.y+time_f)*4.0), cos((tc.x-time_f)*3.5));
    uv = vec2(pingPong(uv.x, 1.0), pingPong(uv.y, 1.0));
    color = texture(samp, uv);
}
```

Shader #78 - breathe

File: breathe.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main(void) {
    vec2 normCoord = gl_FragCoord.xy / iResolution.xy;
```

```

vec2 centeredCoord = normCoord - vec2(0.5, 0.5);
centeredCoord.x *= iResolution.x / iResolution.y;
float audioScale = 0.1 + amp_low * 0.4;
float scale = 1.0 + audioScale * sin(time_f * 2.0);
vec2 scaledCoord = centeredCoord * scale;
scaledCoord.x *= iResolution.y / iResolution.x;
scaledCoord += vec2(0.5, 0.5);
scaledCoord = clamp(scaledCoord, 0.0, 1.0);
color = texture(samp, scaledCoord);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
color.rgb *= 1.0 + _ab * 0.6;
color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #79 - bright

File: bright.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    float x = gl_FragCoord.x;
    float y = gl_FragCoord.y;
    vec2 cord1 = vec2(tc[0]/x, tc[1]/y);
    vec2 cord2 = vec2(tc[0]/x, tc[1]/y);
    vec2 cord3 = vec2(tc[0]/x, tc[1]/y);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    color[0] = color[0]+col1[0];
    color[1] = color[1]+col2[1];
    color[2] = color[2]+col3[2];
}

```


Shader #81 - brighten

File: brighten.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 cb = color;
    color[0] = color[0]*1.5;
    color[1] = color[1]*1.5;
    color[2] = color[2]*1.5;
}
```

Shader #82 - brighten rev

File: brighten_rev.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
```

```

    color = texture(samp, tc);
    vec4 cb = color;
    color[0] = color[0]*cb[2]*2.3;
    color[1] = color[1]*cb[1]*2.3;
    color[2] = color[2]*cb[0]*2.3;
}

```

Shader #93 - cane

File: cane.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 7.0);
    vec2 uv = tc * 2.0 - 1.0;
    float zigzagAmount = 0.1;
    uv.x += pingPong(uv.y * 10.0 + time_t, 1.0) * zigzagAmount;
    uv.y += pingPong(uv.x * 10.0 + time_t, 1.0) * zigzagAmount;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    float swirlStrength = 3.0;
    angle += swirlStrength * exp(-radius * 2.0) * time_t;
    uv = vec2(cos(angle), sin(angle)) * radius;
    uv = (uv + 1.0) / 2.0;
    uv = clamp(uv, 0.0, 1.0);
    vec4 texColor = texture(samp, uv);
    color = texColor;
}

```

Shader #94 - cd

File: cd.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {

```

```

    vec2 centeredCoord = tc - 0.5;
    float angle = time_f;
    vec2 rotatedCoord;
    rotatedCoord.x = centeredCoord.x * cos(angle) - centeredCoord.y * sin(angle);
    rotatedCoord.y = centeredCoord.x * sin(angle) + centeredCoord.y * cos(angle);
    rotatedCoord += 0.5;
    color = texture(samp, rotatedCoord);
}

```

Shader #98 - cmod

File: cmod.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
void main(void) {
    vec2 uv = tc;
    vec4 texColor = blur(samp, uv, iResolution);
    texColor.r *= 0.5 + 0.5 * cos(time_f * 2.0);
}

```

```

    texColor.g *= 0.5 + 0.5 * cos(time_f * 2.0 + 2.0);
    texColor.b *= 0.5 + 0.5 * cos(time_f * 2.0 + 4.0);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = sin(texColor * time_t);
}

```

Shader #139 - comb3

File: comb3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec2 rotateUV(vec2 uv, float angle) {
    float s = sin(angle);
    float c = cos(angle);
    uv -= 0.5;
    uv = mat2(c, -s, s, c) * uv;
    uv += 0.5;
    return uv;
}

vec2 reflectUV(vec2 uv, float segments) {
    float angle = atan(uv.y - 0.5, uv.x - 0.5);
    float radius = length(uv - vec2(0.5));
    float segmentAngle = 2.0 * 3.14159265359 / segments;
    angle = mod(angle, segmentAngle);
    angle = abs(angle - segmentAngle * 0.5);
    return vec2(cos(angle), sin(angle)) * radius + 0.5;
}

vec2 fractalZoom(vec2 uv, float zoom, float time) {
    for (int i = 0; i < 5; i++) {
        uv = abs(uv * zoom) - 0.5;
        uv = rotateUV(uv, time * 0.1);
    }
    return uv;
}

void main() {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec4 originalTexture = texture(samp, tc);
    vec2 kaleidoUV = reflectUV(uv, 6.0);
    float zoom = 1.5 + 0.5 * sin(time_f * 0.5);
    kaleidoUV = fractalZoom(kaleidoUV, zoom, time_f);
    kaleidoUV = rotateUV(kaleidoUV, time_f * 0.2);
    vec4 kaleidoColor = texture(samp, kaleidoUV);
    float blendFactor = 0.6;
    vec4 blendedColor = mix(kaleidoColor, originalTexture, blendFactor);
    blendedColor.rgb *= 0.5 + 0.5 * sin(kaleidoUV.xy * time_f);
}

```

```

    color = sin(blendedColor * pingPong(time_f, 10.0));
    vec4 t = texture(samp, tc);
    color = color * t * 0.8;
    color = sin(color * pingPong(time_f, 15.0));
    color.a = 1.0;
}

```

Shader #144 - comp quality

File: comp_quality.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Hermite smoothstep interpolation
 - Distance field computation - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash12(vec2 p){return fract(sin(dot(p,vec2(127.1,311.7)))*43758.5453123);}
vec3 tonemap(vec3 x){
    x = x / (x + vec3(1.0));
    return pow(x, vec3(1.0 / 2.2));
}
vec3 compositeEffect(vec2 uv) {
    vec2 px = 1.0 / iResolution;
    float off = 1.5 * px.x;
    vec3 col;
    col.r = texture(samp, uv + vec2(off, 0.0)).r;
    col.g = texture(samp, uv).g;
    col.b = texture(samp, uv - vec2(off, 0.0)).b;
    float n = hash12(uv * iResolution + time_f * 123.7) - 0.5;
    col += n * 0.02;
    float scan = sin((uv.y * iResolution.y) * 1.5) * 0.06;
    col -= scan;
    float bleed = sin(uv.y * iResolution.y * 0.2 + time_f * 5.0) * 0.004;
    col += vec3(bleed * 0.001, 0.0, -bleed * 0.001);
    return col;
}
void main(void) {
    vec2 uv = tc;
    vec3 col = compositeEffect(uv);
    vec2 px = 1.0 / iResolution;
    vec3 blur =
        compositeEffect(uv + vec2( px.x,  px.y)) +
        compositeEffect(uv + vec2(-px.x,  px.y)) +
        compositeEffect(uv + vec2( px.x, -px.y)) +
        compositeEffect(uv + vec2(-px.x, -px.y));
    blur *= 0.25;
    float sharpAmt = 0.35;
    col += (col - blur) * sharpAmt;
    vec2 c = vec2(0.5);
    float v = smoothstep(0.95, 0.25, distance(uv, c));
    col *= v;
    col = tonemap(col);
    color = vec4(clamp(col,0.0,1.0), 1.0);
}

```

Shader #155 - cos

File: cos.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    for(int i = 0; i < 3; ++i) {
        color[i] += cos(color[i] * timeval);
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #156 - cos sin

File: cos_sin.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
```

```

uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    color[0] += timeval*cos(0.1489*tc[0]);
    color[1] += timeval*sin(0.1482*tc[1]);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #161 - crt

File: crt.glsl

Overlays CRT-style scanlines onto the image for a retro display effect.

Technique: Scanline overlay using modular y-coordinate - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    vec2 center = uv - 0.5;
    uv = uv + center * (0.1 * length(center));
    float scanline = sin(uv.y * iResolution.y * 0.75) * 0.05;
    vec3 offset = vec3(0.001, -0.001, 0.0) * scanline;
    vec3 texColor;
    texColor.r = texture(samp, uv + vec2(offset.r, 0.0)).r;
    texColor.g = texture(samp, uv + vec2(0.0, offset.g)).g;
    texColor.b = texture(samp, uv + vec2(0.0, offset.b)).b;
    texColor *= (1.0 - scanline);
    color = vec4(texColor, 1.0);
}

```

Shader #162 - crystal 2

File: crystal-2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 5.0) + 2.0;
    float radius = sin(0.5 * time_t);
    vec2 center = vec2(iResolution.x / 2.0, iResolution.y / 2.0);
    vec2 texCoord = tc * iResolution;
    vec2 delta = texCoord - center;
    float dist = length(delta);
    float maxRadius = min(iResolution.x, iResolution.y) * radius;
    vec2 newTexCoord = texCoord;
    if (dist < maxRadius) {
        float scaleFactor = 1.0 - sqrt(dist / maxRadius);
        newTexCoord = center + delta * scaleFactor;
    }
    newTexCoord = clamp(newTexCoord / iResolution, 0.0, 1.0);
    vec4 texColor = texture(samp, newTexCoord);
    color = texColor;
}
```

Shader #163 - crystal 3

File: crystal-3.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 5.0) + 2.0;
    float radius = sin(0.5 * time_t);
    vec2 center = vec2(iResolution.x / 2.0, iResolution.y / 2.0);
    vec2 texCoord = tc * iResolution;
    vec2 delta = texCoord - center;
    float dist = length(delta);
    float maxRadius = min(iResolution.x, iResolution.y) * radius;
    float angle = time_f;
    float cosAngle = cos(angle);
    float sinAngle = sin(angle);
    vec2 rotatedDelta = vec2(
```



```

        cosAngle * delta.x - sinAngle * delta.y,
        sinAngle * delta.x + cosAngle * delta.y
    );
    vec2 newTexCoord = texCoord;
    if (dist < maxRadius) {
        float scaleFactor = 1.0 - sqrt(dist / maxRadius);
        newTexCoord = center + rotatedDelta * scaleFactor;
    }
    newTexCoord = clamp(newTexCoord / iResolution, 0.0, 1.0);
    vec4 texColor = texture(samp, newTexCoord);
    color = texColor;
}

```

Shader #164 - crystal 4

File: crystal-4.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 5.0) + 2.0;
    float radius = 1.0;
    vec2 center = vec2(iResolution.x / 2.0, iResolution.y / 2.0);
    vec2 texCoord = tc * iResolution;
    vec2 delta = texCoord - center;
    float dist = length(delta);
    float maxRadius = min(iResolution.x, iResolution.y) * radius;
    float angle = time_f;
    float cosAngle = cos(angle);
    float sinAngle = sin(angle);
    vec2 rotatedDelta = vec2(
        cosAngle * delta.x - sinAngle * delta.y,
        sinAngle * delta.x + cosAngle * delta.y
    );
    vec2 newTexCoord = texCoord;
    if (dist < maxRadius) {
        float scaleFactor = 1.0 - sqrt(dist / maxRadius);
        newTexCoord = center + rotatedDelta * scaleFactor;
    }
    newTexCoord = clamp(newTexCoord / iResolution, 0.0, 1.0);
    vec4 texColor = texture(samp, newTexCoord);
    color = texColor;
}

```

Shader #165 - crystal

File: crystal.glsl

Processes the input texture to produce the "crystal" visual effect. Example hardcoded radius
Hardcoded center to the middle of the screen

Technique: Driven by: resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    float radius = 0.5;
    vec2 center = vec2(iResolution.x / 2.0, iResolution.y / 2.0);
    vec2 texCoord = tc * iResolution;
    vec2 delta = texCoord - center;
    float dist = length(delta);
    float maxRadius = min(iResolution.x, iResolution.y) * radius;
    vec2 newTexCoord = texCoord;
    if (dist < maxRadius) {
        float scaleFactor = 1.0 - sqrt(dist / maxRadius);
        newTexCoord = center + delta * scaleFactor;
    }
    newTexCoord = clamp(newTexCoord / iResolution, 0.0, 1.0);
    vec4 texColor = texture(samp, newTexCoord);
    color = texColor;
}
```

Shader #166 - crystalball

File: crystalball.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    float aspectRatio = iResolution.x / iResolution.y;
    vec2 correctedCenter = vec2(0.5, 0.5 / aspectRatio);
    vec2 adjustedTC = tc;
    adjustedTC.y /= aspectRatio;
    vec2 toCenter = adjustedTC - correctedCenter;
    float dist = length(toCenter) * aspectRatio;
    float zoomRange = 0.1 + (0.5 + 0.5 * sin(time_f * 2.0)) * 6.0;
    float dynamicRadius = 0.2 * zoomRange;
    if (dist < dynamicRadius) {
        float refractStrength = 0.5 * zoomRange;
        vec2 refractTC = toCenter * (1.0 - refractStrength * sqrt(1.0 - (dist / dynamicRadius))) / dist;
        refractTC.y *= aspectRatio;
        color = texture(samp, correctedCenter + refractTC);
    } else {
        color = texture(samp, adjustedTC);
    }
}
```

```
}  
}
```

Shader #176 - cyclone

File: cyclone.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core  
out vec4 color;  
uniform sampler2D samp;  
uniform vec2 iResolution;  
uniform float time_f;  
uniform float amp_peak;  
uniform float amp_rms;  
uniform float amp_smooth;  
uniform float amp_low;  
uniform float amp_mid;  
uniform float amp_high;  
uniform float iamp;  
void main(void) {  
    vec2 normCoord = gl_FragCoord.xy / iResolution.xy;  
    vec2 centeredCoord = normCoord - vec2(0.5, 0.5);  
    centeredCoord.x *= iResolution.x / iResolution.y;  
    float r = length(centeredCoord);  
    float theta = atan(centeredCoord.y, centeredCoord.x);  
    float twistAmount = 15.0 + amp_low * 20.0;  
    theta += (1.0 - r) * twistAmount * sin(time_f);  
    vec2 twistedCoord = vec2(cos(theta), sin(theta)) * r;  
    twistedCoord.x *= iResolution.y / iResolution.x;  
    twistedCoord += vec2(0.5, 0.5);  
    color = texture(samp, twistedCoord);  
    float _ab = clamp(amp_peak, 0.0, 1.0);  
    float _abass = clamp(amp_low, 0.0, 1.0);  
    color.rgb *= 1.0 + _ab * 0.6;  
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(  
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);  
}
```

Shader #181 - dark

File: dark.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330  
in vec2 tc;  
out vec4 color;  
uniform float alpha_r;  
uniform float alpha_g;
```

```

uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    float x = gl_FragCoord.x;
    float y = gl_FragCoord.y;
    vec2 cord1 = vec2(tc[0]/x, tc[1]/y);
    vec2 cord2 = vec2(tc[0]/x, tc[1]/y);
    vec2 cord3 = vec2(tc[0]/x, tc[1]/y);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    color[0] = (color[0]+col1[0]) * 0.9;
    color[1] = (color[1]+col2[1]) * 0.9;
    color[2] = (color[2]+col3[2]) * 0.9;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #183 - dark psych

File: dark_psych.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

```

```

}
void main(void) {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float time = time_f * 0.5;
    float sine1 = sin(uv.x * 10.0 + time) * cos(uv.y * 10.0 + time);
    float sine2 = sin(uv.y * 15.0 - time) * cos(uv.x * 15.0 - time);
    float sine3 = sin(sqrt(uv.x * uv.x + uv.y * uv.y) * 20.0 + time);
    float pattern = sine1 + sine2 + sine3;
    float colorIntensity = pattern * 0.5 + 0.5;
    vec3 psychedelicColor = vec3(
        sin(colorIntensity * 6.28318 + 0.0) * 0.5 + 0.5,
        sin(colorIntensity * 6.28318 + 2.09439) * 0.5 + 0.5,
        sin(colorIntensity * 6.28318 + 4.18879) * 0.5 + 0.5
    );
    vec3 baseColor = psychedelicColor;
    uv = tc * 2.0 - 1.0;
    float t = mod(time_f, 20);
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + t * 20.0;
    vec3 rainbowColor = rainbow(angle / (2.0 * 3.14159));
    vec4 originalColor = texture(samp, tc);
    vec3 blendedColor = mix(originalColor.rgb, rainbowColor, 0.5);
    color = vec4(sin(blendedColor * t), originalColor.a);
    color = mix(color, vec4(baseColor, 1.0), 0.5);
}

```

Shader #189 - dark square

File: dark_square.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
}

```

```

    }
    color[0] = color[0]*sin(tc[0] * timeval);
    color[1] = color[1]*cos(tc[1] * timeval);
    color[2] = color[2]*tan(tc[0] * timeval);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #190 - darkb

File: darkb.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    color[2] = 0.1;
}

```

Shader #191 - darkg

File: darkg.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;

```

```

uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    color[1] = 0.1;
}

```

Shader #192 - darkr

File: darkr.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    color[0] = 0.1;
}

```

Shader #193 - darkred

File: darkred.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330

```

```

in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[2] = (color[0]*timeval*0.05)*alpha_b;
    color[1] = (color[1]*timeval*0.08)*alpha_g;
    color[0] = 1.0;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #209 - delic

File: delic.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    uv.x += sin(uv.y * 10.0 + time_f) * 0.1;
    uv.y += cos(uv.x * 10.0 + time_f) * 0.1;
    float modFactor = pingPong(time_f * 0.5, 1.0);

```



```

    vec4 texColor = texture(samp, uv);
    vec3 psychedelicColor = vec3(
        texColor.r * sin(time_f + uv.y * 10.0),
        texColor.g * cos(time_f + uv.x * 10.0),
        texColor.b * sin(time_f + uv.y * 5.0)
    );
    psychedelicColor = mix(texColor.rgb, psychedelicColor, modFactor);
    color = vec4(psychedelicColor, texColor.a);
}

```

Shader #211 - diagsquare

File: diagsquare.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
void main() {
    vec2 uv = tc;
    float squareSize = 0.2;
    vec2 gridCoord = floor(uv / squareSize);
    vec2 localCoord = fract(uv / squareSize);
    float diag = mod(gridCoord.x + gridCoord.y, 2.0);
    vec4 texColor = texture(samp, uv);
    if (diag == 0.0) {
        float shift = sin(time_f + gridCoord.x * 0.5) * 0.1;
        localCoord.x += shift;
        uv = fract(gridCoord * squareSize + localCoord);
        texColor = texture(samp, uv);
    }
    color = texColor;
}

```

Shader #212 - diagsquarecircle

File: diagsquarecircle.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - 2D rotation matrix -

Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

```

```

float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 rotate(vec2 p, float angle) {
    float s = sin(angle);
    float c = cos(angle);
    return vec2(c * p.x - s * p.y, s * p.x + c * p.y);
}
void main() {
    vec2 uv = tc;
    float squareSize = sin(0.2 * time_f);
    vec2 gridCoord = floor(uv / squareSize);
    vec2 localCoord = fract(uv / squareSize);
    float diag = mod(gridCoord.x + gridCoord.y, 2.0);
    if (diag == 0.0) {
        float angle = time_f + gridCoord.x * 0.5 + gridCoord.y * 0.5;
        localCoord = rotate(localCoord - 0.5, angle) + 0.5;
        uv = fract(gridCoord * squareSize + localCoord);
    }
    vec4 texColor = texture(samp, uv);
    color = texColor;
}

```

Shader #217 - disk

File: disk.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
 - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float spinningSpeed = mix(5.0, 1.0, radius);
    float spinningAngle = angle + time_f * spinningSpeed;
    vec2 spinningUV = vec2(cos(spiningAngle), sin(spiningAngle)) * radius * 0.5 + 0.5;
    vec4 texColor = texture(samp, spinningUV);
    float centerGlow = exp(-radius * 10.0);
    float outerGlow = smoothstep(0.4, 0.5, radius) * 0.5 + 0.5;
    float glow = max(centerGlow, outerGlow);
    color = texColor * vec4(glow, glow, glow, 1.0);
}

```

Shader #232 - dmdc

File: dmdc.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look.

Technique: Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending -
Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float pingPong(float t, float l) {
    return abs(mod(t, 2.0 * l) - l);
}
mat2 rotate(float angle) {
    float c = cos(angle);
    float s = sin(angle);
    return mat2(c, -s, s, c);
}
void main() {
    vec2 uv = tc * iResolution.xy;
    vec2 centerUV = (floor(uv / 50.0) + 0.5) * 50.0;
    vec2 localUV = uv - centerUV;
    float t = pingPong(time_f, 4.0) * 2.0;
    float rotation = atan(localUV.y, localUV.x) + length(localUV) * 0.1 + t;
    localUV = rotate(rotation) * localUV;
    vec2 diamondMask = abs(mod(localUV, 100.0) - 50.0);
    float mask = smoothstep(25.0, 30.0, length(diamondMask - vec2(25.0)));
    vec2 texCoord = (centerUV + localUV) / iResolution.xy;
    vec4 sampledColor = texture(samp, texCoord);
    color = mix(sampledColor, vec4(0.0), mask);
}
```

Shader #233 - drain

File: drain.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralSpeed = 2.0 + amp_low * 4.0;
```

```

float inwardSpeed = 0.5;
float drainRadius = 8.0;
float loopDuration = 20.0;
float currentTime = mod(time_f, loopDuration);
angle += (1.0 - smoothstep(0.0, drainRadius, dist)) * currentTime * spiralSpeed;
    dist *= mix(1.0, 0.0, currentTime * inwardSpeed);
vec2 spiralCoord = vec2(cos(angle), sin(angle)) * dist;
spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
color = texture(samp, spiralCoord);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
color.rgb *= 1.0 + _ab * 0.6;
color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #241 - drain reset

File: drain_reset.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
 - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    float loopDuration = 5.0;
    float currentTime = mod(time_f, loopDuration);
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralSpeed = 5.0;
    float inwardSpeed = currentTime / loopDuration;    angle += (1.0 - smoothstep(0.0, 8.0, dist)) *
    ↪ currentTime * spiralSpeed;
    dist *= 1.0 - inwardSpeed;
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * tan(dist);
    spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
    color = texture(samp, spiralCoord);
}

```

Shader #324 - empty1

File: empty1.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    color = texture(samp, tc);
}

```

Shader #333 - fat blue

File: fat-blue.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float stretch_factor = 1.0 + 0.5 * abs(sin(time_f * 3.14));
    vec2 new_tc = center + dir;
    vec4 original_color = texture(samp, tc);
    vec4 stretched_color = texture(samp, center + vec2(dir.x / stretch_factor, dir.y));
    color = vec4(original_color.r, original_color.g, stretched_color.b, original_color.a);
}

```

Shader #334 - fat green

File: fat-green.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float stretch_factor = 1.0 + 0.5 * abs(sin(time_f * 3.14));
    vec2 new_tc = center + dir;
    vec4 original_color = texture(samp, tc);
    vec4 stretched_color = texture(samp, center + vec2(dir.x / stretch_factor, dir.y));
    color = vec4(original_color.r, stretched_color.g, original_color.b, original_color.a);
}

```

Shader #335 - fat red

File: fat-red.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float stretch_factor = 1.0 + 0.5 * abs(sin(time_f * 3.14));
    vec2 new_tc = center + dir;
    vec4 original_color = texture(samp, tc);
    vec4 stretched_color = texture(samp, center + vec2(dir.x / stretch_factor, dir.y));
    color = vec4(stretched_color.r, original_color.g, original_color.b, original_color.a);
}
```

Shader #337 - fat slow

File: fat-slow.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float stretch_factor = 1.0 + 0.5 * abs(sin(time_f * 3.14));
    dir.x /= stretch_factor;
    vec2 new_tc = center + dir;
    color = texture(samp, new_tc);
}
```

Shader #338 - fat

File: fat.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
```

```

uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float factor = 0.5;
    dir.x *= factor;
    vec2 new_tc = center + dir;
    color = texture(samp, new_tc);
}

```

Shader #345 - focus

File: focus.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float noise(vec2 st) {
    return fract(sin(dot(st.xy, vec2(12.9898,78.233))) * 43758.5453123);
}
void main(void) {
    vec2 uv = tc;
    float time = time_f * 0.5;
    float noiseValue = noise(uv * 10.0 + time);
    vec2 heatWaveOffset = vec2(noiseValue - 0.5, 0.0) * 0.02;
    float time_t = mod(time_f, 10);
    vec2 nuv = uv + sin(heatWaveOffset * time_t);
    vec4 texColor = texture(samp, nuv);
    color = texColor;
}

```

Shader #354 - frac mir

File: frac_mir.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform vec2 iResolution;
uniform float time_f;
uniform sampler2D samp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;

```

```

uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec2 kaleidoscope(vec2 uv, float segments) {
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    angle = mod(angle, 2.0 * 3.14159265359 / segments);
    return vec2(cos(angle), sin(angle)) * radius;
}
float fractalSpiral(vec2 uv) {
    float scale = 1.0;
    float intensity = 0.0;
    float rotation = time_f * 0.5;
    for (int i = 0; i < 8; i++) {
        uv = mat2(cos(rotation), -sin(rotation), sin(rotation), cos(rotation)) * uv;
        uv = abs(uv) / dot(uv, uv) - 0.5;
        intensity += 1.0 / scale * sin(uv.x * 10.0 + uv.y * 10.0 + time_f);
        scale *= 1.8;
    }
    return intensity;
}
vec3 blendMaterials(vec3 texColor, vec3 fractalColor, vec2 uv) {
    float fresnel = pow(1.0 - dot(normalize(uv), vec2(0.0, 0.0)), 2.0);
    vec3 specularHighlight = vec3(1.0) * pow(max(0.0, fractalColor.r), 20.0);
    return mix(texColor, fractalColor + specularHighlight, fresnel);
}
float pingPong(float x, float length) {
    return length - abs(mod(x, 2.0 * length) - length);
}
void main() {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float segments = 6.0 + mod(time_f, 10.0);
    uv = kaleidoscope(uv, segments);
    float fractalValue = fractalSpiral(uv * 2.0);
    vec3 fractalColor = vec3(fractalValue * 0.8, fractalValue * 0.6, fractalValue * 0.4);
    vec3 texColor = texture(samp, uv * 0.5 + 0.5).rgb;
    vec3 blendedColor = blendMaterials(texColor, fractalColor, uv);
    blendedColor += vec3(0.3);
    blendedColor.r += sin(uv.x * 2.0 + time_f) * 0.2;
    blendedColor.g += sin(uv.y * 2.0 + time_f * 0.5) * 0.2;
    blendedColor.b += sin((uv.x + uv.y) * 0.5 + time_f) * 0.2;
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    blendedColor *= 1.0 + _ab * 0.6;
    blendedColor = mix(blendedColor, blendedColor * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 +
↵ clamp(amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(blendedColor, 1.0);
    color = mix(color, texture(samp, tc), 0.5);
    float time_t = pingPong(time_f, 5.0) + 1.5;
    color = color * sin(vec4(blendedColor, 1.0) * time_t);
    color.a = 1.0;
}

```

Shader #458 - g ufo

File: g_ufo.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright.

Technique: 2D rotation matrix - Distance field computation - Radial vignette falloff - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec2 rotate(vec2 pos, float angle) {
    float s = sin(angle);
    float c = cos(angle);
    pos -= vec2(0.5);
    pos = mat2(c, -s, s, c) * pos;
    pos += vec2(0.5);
    return pos;
}
void main(void) {
    vec2 pos = tc;
    float aspectRatio = iResolution.x / iResolution.y;
    pos.x *= aspectRatio;
    float spinSpeed = time_f * 0.5;
    pos = rotate(pos, spinSpeed);
    float dist = distance(pos, vec2(0.5 * aspectRatio, 0.5));
    float scale = 1.0 + 0.2 * sin(dist * 15.0 - time_f * 2.0);
    pos = (pos - vec2(0.5)) * scale + vec2(0.5);
    color = texture(samp, pos);
}
```

Shader #461 - gaura

File: gaura.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec4 texColor = texture(samp, uv);
    float pulse = 0.5 + 0.5 * sin(time_f * 3.0);
    vec2 offset1 = vec2(0.01, 0.02) * sin(time_f + length(uv - 0.5) * 20.0);
    vec2 offset2 = vec2(-0.02, 0.01) * cos(time_f * 1.5 + length(uv - 0.5) * 15.0);
    vec2 offset3 = vec2(0.03, -0.03) * sin(time_f * 5.0);
    vec3 aura1 = texture(samp, uv + offset1).rgb * 0.6;
    vec3 aura2 = texture(samp, uv + offset2).rgb * 0.4;
    vec3 aura3 = texture(samp, uv + offset3).rgb * 0.3;
    vec3 strobingGradient = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 10.0),
        0.5 + 0.5 * cos(time_f + uv.y * 10.0),

```

```

        0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 10.0)
    );
    vec3 auraGlow = mix(aura1 + aura2 + aura3, strobingGradient, 0.5);
    float auraStrength = 0.4 + 0.3 * sin(time_f * 4.0 + length(uv - 0.5) * 10.0);
    vec3 finalAura = auraGlow * auraStrength;
    color = vec4(sin(finalAura * (pingPong(time_f, 8.0) + 2.0)), 0.5 * texColor.a);
}

```

Shader #473 - gem pong

File: gem-pong.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns. Standard Ping-Pong function Neon Palette Generator Mask for the sphere shape

Technique: Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 neonGradient(float t) {
    return 0.5 + 0.5 * cos(6.28318 * (t + vec3(0.0, 0.33, 0.67)));
}
void main(void) {
    vec2 uv = (tc * 2.0 - 1.0);
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float d = length(uv);
    float radius = 0.85;
    float mask = smoothstep(radius, radius - 0.01, d);
    float z = sqrt(max(0.0, radius * radius - d * d));
    vec3 normal = normalize(vec3(uv, z));
    float t1 = pingPong(time_f * 0.4, 4.0);
    float t2 = pingPong(time_f * 0.2, 5.0) - 2.5;
    float angle = atan(uv.y, uv.x);
    float spiral = angle + (t2 * log(d + 0.1)) + t1;
    vec3 spiralColor = neonGradient(spiral * 0.5);
    float diffuse = max(dot(normal, normalize(vec3(1.0, 1.0, 1.0))), 0.0);
    float specular = pow(max(dot(reflect(vec3(-0.5, -0.5, -1.0), normal), vec3(0,0,1)), 0.0), 20.0);
    float blurAmt = pingPong(time_f * 0.1, 1.0);
    vec4 tex = texture(samp, tc + (normal.xy * 0.05 * blurAmt));
    vec3 finalRGB = spiralColor * (diffuse + 0.4) + (specular * 0.5);
    vec3 background = tex.rgb * 0.3;
    vec3 result = mix(background, finalRGB, mask);
    float edgeGlow = smoothstep(radius + 0.05, radius - 0.05, d) * (1.0 - mask);
    result += neonGradient(time_f * 0.2) * edgeGlow * 0.5;
    color = vec4(result, 1.0);
}

```

Shader #478 - gem frac

File: gem_frac.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Store original UV for distance effects Space folding (The "Fractal" part) Use a variation of your wave/pingpong logic for contrast

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    vec2 uv0 = uv;
    vec3 finalColor = vec3(0.0);
    float t = time_f * 0.2;
    for (float i = 0.0; i < 4.0; i++) {
        uv = fract(uv * 1.5) - 0.5;
        float d = length(uv) * exp(-length(uv0));
        float angle = atan(uv.y, uv.x) + (t + i);
        vec3 col = vec3(
            sin(angle * 3.0 + i * 1.2),
            sin(angle * 4.0 + i * 1.5),
            sin(angle * 5.0 + i * 1.8)
        );
        d = sin(d * 8.0 + t) / 8.0;
        d = abs(d);
        d = pow(0.01 / d, 1.2);
        finalColor += col * d;
    }
    vec3 texColor = texture(samp, tc).rgb;
    vec3 result = mix(finalColor, texColor, 0.7);
    color = vec4(result, alpha);
}
```

Shader #489 - genergy

File: genergy.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec4 texColor = texture(samp, uv);
    float pulse = 0.5 + 0.5 * sin(time_f * 4.0);
    vec2 trailOffset1 = vec2(0.02, 0.03) * sin(time_f * 3.0 + length(uv - 0.5) * 15.0) * pulse;
```

```

vec2 trailOffset2 = vec2(-0.03, 0.02) * cos(time_f * 2.0 + length(uv - 0.5) * 20.0) * pulse;
vec2 trailOffset3 = vec2(0.01, -0.01) * sin(time_f * 5.0) * (1.0 - pulse);
vec3 trail1 = texture(samp, uv + trailOffset1).rgb * 0.7;
vec3 trail2 = texture(samp, uv + trailOffset2).rgb * 0.5;
vec3 trail3 = texture(samp, uv + trailOffset3).rgb * 0.3;
vec3 energyGlow = vec3(
    0.5 + 0.5 * sin(time_f + uv.x * 25.0),
    0.5 + 0.5 * cos(time_f * 1.2 + uv.y * 25.0),
    0.5 + 0.5 * sin(time_f * 0.8 + uv.x * uv.y * 25.0)
);
vec3 finalColor = mix(trail1 + trail2 + trail3, energyGlow, 0.6);
finalColor *= 0.8 + 0.2 * sin(time_f * 6.0 + length(uv - 0.5) * 10.0);
color = vec4(finalColor, texColor.a);
}

```

Shader #490 - geo pi

File: geo-pi.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    float time = pingPong(time_f * PI, 10) * 0.5;
    float time_t = mod(time_f, 10.0);
    float angle = time;
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 toCenter = uv - center;
    float radius = length(toCenter);
    float theta = atan(toCenter.y, toCenter.x) + time;
    float pattern = abs(sin(12.0 * theta) * cos(12.0 * radius * (pingPong(time_f * PI, 20.0))));
    vec3 colorShift = vec3(0.5 * sin(time) + 0.5, 0.5 * cos(time) + 0.5, sin(radius - (pingPong(time *
    ↪ PI, 10.0))));
    vec3 finalColor = (0.2 + 0.8 * pattern) + colorShift * pattern;
    vec4 text_color = texture(samp, tc);
    vec4 fc = vec4(finalColor, 1.0);
    color = mix(text_color, fc, 0.3);
}

```

Shader #491 - geometric

File: geometric.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    float time = time_f * 0.5;
    float time_t = mod(time_f, 10.0);
    float angle = time;
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 toCenter = uv - center;
    float radius = length(toCenter);
    float theta = atan(toCenter.y, toCenter.x) + time;
    float pattern = abs(sin(12.0 * theta) * cos(12.0 * radius));
    vec3 colorShift = vec3(0.5 * sin(time) + 0.5, 0.5 * cos(time) + 0.5, sin(radius - time));
    vec3 finalColor = (0.2 + 0.8 * pattern) + colorShift * pattern;
    vec4 text_color = texture(samp, tc);
    vec4 fc = vec4(finalColor, 1.0);
    color = mix(text_color, fc, 0.3);
}
```

Shader #492 - geometric2

File: geometric2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    float time = time_f * 0.5;
    float time_t = mod(time_f, 10.0);
    float angle = time;
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 toCenter = uv - center;
    float radius = length(toCenter);
    float theta = atan(toCenter.y, toCenter.x) + time;
    float pattern = abs(sin(12.0 * theta) * cos(12.0 * radius));
    vec3 colorShift = vec3(0.5 * sin(time) + 0.5, 0.5 * cos(time) + 0.5, sin(radius - time));
    vec3 finalColor = (0.2 + 0.8 * pattern) + colorShift * pattern;
    vec4 text_color = texture(samp, tc);
    vec4 fc = vec4(finalColor, 1.0);
    color = sin(mix(text_color, fc, 0.3) * time);
}
```

```
    color.a = 1.0;
}
```

Shader #493 - geometric3

File: geometric3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    float time = time_f * 0.5;
    float time_t = mod(time_f, 15.0);
    float angle = time;
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 toCenter = uv - center;
    float radius = length(toCenter);
    float theta = atan(toCenter.y, toCenter.x) + time;
    float pattern = abs(sin(12.0 * theta) * cos(12.0 * radius));
    vec3 colorShift = vec3(0.5 * sin(time) + 0.5, 0.5 * cos(time) + 0.5, sin(radius - time));
    vec3 finalColor = (0.2 + 0.8 * pattern) + colorShift * pattern;
    vec4 text_color = texture(samp, tc);
    vec4 fc = vec4(finalColor, 1.0);
    color = sin(mix(text_color, fc, 0.3) * length(time_t));
    color.a = 1.0;
}
```

Shader #494 - geometric4

File: geometric4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    float time = time_f * 0.5;
    float angle = time;
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
```

```

    vec2 toCenter = uv - center;
    float radius = length(toCenter);
    float theta = atan(toCenter.y, toCenter.x) + time;
    float pattern = abs(sin(12.0 * theta) * cos(12.0 * radius));
    vec4 texColor = texture(samp, tc);
    vec2 modUv = tc + (0.1 * pattern) * vec2(cos(theta), sin(theta));
    vec4 modTexColor = texture(samp, modUv);
    color = mix(texColor, modTexColor, pattern);
}

```

Shader #495 - geometric5

File: geometric5.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = uv - center;
    pos *= 2.0;
    float angle = time_f;
    float cosA = cos(angle);
    float sinA = sin(angle);
    mat2 rot = mat2(cosA, -sinA, sinA, cosA);
    pos = rot * pos;
    float depth = mod(pos.x + pos.y, 0.5) - 0.25;
    float height = sin(depth * 3.14159) * 0.25;
    vec3 baseColor = vec3(0.5 + 0.5 * sin(time_f), 0.5 + 0.5 * sin(time_f + 2.0), 0.5 + 0.5 * sin(time_f
    ↵ + 4.0));
    float light = 0.5 + 0.5 * sin(depth * 3.14159 + time_f);
    vec3 color3D = height * light * baseColor;
    color3D = sin(color3D * time_f);
    vec4 texColor = texture(samp, uv);
    color = mix(texColor, vec4(color3D, 1.0), 0.3);
}

```

Shader #496 - gfs

File: gfs.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform vec2 iResolution;

```

```

uniform float time_f;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = (uv - 0.5) * 2.0;
    float zoom = 1.0 + 0.5 * sin(time_f);
    centeredUV *= zoom;
    float angle = time_f * 1.5 * zoom;
    float cosA = cos(angle);
    float sinA = sin(angle);
    centeredUV = vec2(
        cosA * centeredUV.x - sinA * centeredUV.y,
        sinA * centeredUV.x + cosA * centeredUV.y
    );
    float prismAngle = atan(centeredUV.y, centeredUV.x);
    float radius = length(centeredUV);
    float prismEffect = sin(prismAngle * 3.0) * 0.1;
    vec2 refractedUV = vec2(
        centeredUV.x + prismEffect * cos(prismAngle),
        centeredUV.y + prismEffect * sin(prismAngle)
    );
    refractedUV = refractedUV * 0.5 + 0.5;
    vec4 texColorPrism = texture(samp, refractedUV);
    vec4 texColorOriginal = texture(samp, uv);
    float blendFactor = smoothstep(0.5, 0.7, radius);
    color = mix(texColorOriginal, texColorPrism, 1.0 - blendFactor);
}

```

Shader #498 - ggrad

File: ggrad.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 7.0);
    vec3 neonPurple = vec3(199.0/255.0, 0.0, 1.0);
    vec3 neonPink = vec3(1.0, 0.0, 127.0/255.0);
    vec3 gradientColor = mix(neonPink, neonPurple, tc.y);
    color = mix(vec4(sin(gradientColor * time_t), 1.0), texture(samp, tc), 0.5);
}

```

Shader #499 - gkale

File: gkale.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv) * 1.4142;
    float segments = 8.0;
    angle = mod(angle, 6.28318 / segments);
    angle = abs(angle - 3.14159 / segments);
    uv = vec2(cos(angle), sin(angle)) * radius;
    uv = uv * 0.5 + 0.5;
    uv = clamp(uv, 0.0, 1.0);
    float time_t = pingPong(time_f * 0.5, 7.0) + 2.0;
    vec4 texColor = texture(samp, uv);
    float pattern = cos(radius * 10.0 - time_t * 5.0);
    vec3 colorShift = vec3(
        0.5 + 0.5 * cos(pattern + time_t + 0.0),
        0.5 + 0.5 * cos(pattern + time_t + 2.094),
        0.5 + 0.5 * cos(pattern + time_t + 4.188)
    );
    vec3 finalColor = texColor.rgb * colorShift;
    color = vec4(finalColor, texColor.a);
    color = mix(color, texture(samp, tc), 0.5);
    color = sin(color * time_t);
    color.a = 1.0;
}
```

Shader #526 - gmir2

File: gmir2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = uv * 2.0 - 1.0;
```

```

float angle = atan(centeredUV.y, centeredUV.x);
float radius = length(centeredUV);
float spin = time_f * 0.5;
angle += floor(mod(angle + 3.14159, 3.14159 / 4.0)) * spin;
vec2 rotatedUV = vec2(cos(angle), sin(angle)) * radius;
rotatedUV = abs(mod(rotatedUV, 2.0) - 1.0);
vec4 texColor = texture(samp, rotatedUV * 0.5 + 0.5);
vec3 gradientEffect = vec3(
    0.5 + 0.5 * sin(time_f + uv.x * 15.0),
    0.5 + 0.5 * cos(time_f + uv.y * 15.0),
    0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 15.0)
);
vec3 finalColor = mix(texColor.rgb, gradientEffect, 0.3);
color = vec4(finalColor, texColor.a);
}

```

Shader #545 - green

File: green.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] /= 2;
    color[1] /= 2;
    color[2] /= 2;
    color[0] = sin(color[0]*alpha_b);
    color[1] = cos(color[1]*alpha_g);
    color[2] = color[2]*alpha_r;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)

```

```

        int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #546 - green aura

File: green_aura.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    float x = gl_FragCoord.x / iResolution_.x / 3;
    float y = gl_FragCoord.y / iResolution_.y / 3;
    color[0] = color[0]*asin(fract(x/3) * 0.3421);
    color[1] = color[1]*acos(fract(y/3) * 0.1249);
    color[2] = color[2]*atan(fract(x+y/3) * 0.2912);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #550 - green move

File: green_move.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    vec2 pos = gl_FragCoord.xy / iResolution_.xy;
    vec4 l = vec4(0.2, 0.8, 0.2, 1.0);
    color = vec4(pos, 1, 1) * l * color * alpha;
}
```

Shader #551 - green strobe

File: green_strobe.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
```

```

uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[1] = color[1]*(random_value[0]/255);
    color[0] = color[0]*(timeval * 0.05812);
    color[2] = color[2]*(alpha_r * timeval * (random_value[1]/255));
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #552 - greenblue

File: greenblue.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = (color[0]/8)*alpha_r * timeval;
    color[1] = (color[1]/4)*alpha_g * timeval;
    color[2] = (color[2]/2)*alpha_b * timeval;
    ivec3 int_color;

```

```

    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #564 - gvortex

File: gvortex.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = (uv - 0.5) * 2.0;
    float radius = length(centeredUV);
    float angle = atan(centeredUV.y, centeredUV.x) + time_f * 0.3;
    float mirrors = 6.0;
    angle = mod(angle, 2.0 * 3.14159 / mirrors) * mirrors;
    float refractAmount = pow(1.0 - radius, 2.0) * 0.5;
    vec2 refractedUV = vec2(
        centeredUV.x + sin(angle) * refractAmount,
        centeredUV.y + cos(angle) * refractAmount
    );
    vec2 texUV = refractedUV * 0.5 + 0.5;
    vec4 texColor = texture(samp, mod(texUV, 1.0));
    float pulse = 0.5 + 0.5 * sin(time_f * 3.0);
    vec3 dynamicColor = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 10.0),
        0.5 + 0.5 * cos(time_f + uv.y * 10.0),
        0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 10.0)
    );
    color = vec4(texColor.rgb * 0.6 + dynamicColor * 0.4 * pulse, texColor.a);
}

```

Shader #572 - heartthrob

File: heartthrob.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform float time_f;
uniform vec2 iResolution;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main(void) {
    vec2 uv = tc - vec2(0.5);
    float radius = length(uv) * 2.0;
    float frequency = 10.0;
    float amplitude = 0.1;
    float audioAmp = amplitude + amp_low * 0.3;
    float pulsate = audioAmp * sin(time_f * frequency);
    float adjustedRadius = clamp(radius + pulsate, 0.0, 1.0);
    vec3 neonBlue = vec3(0.0, 1.0, 1.0);
    vec3 neonPink = vec3(1.0, 0.0, 0.5);
    vec3 gradientColor = mix(neonBlue, neonPink, adjustedRadius);\
    vec4 texColor = texture(samp, tc);
    vec3 finalColor = texColor.rgb * gradientColor;
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalColor *= 1.0 + _ab * 0.6;
    finalColor = mix(finalColor, finalColor * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalColor, texColor.a);
}

```

Shader #573 - heartthrob2

File: heartthrob2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc - vec2(0.5);
    float radius = length(uv) * 2.0;
    float frequency = 10.0;
    float amplitude = 0.1;
    float pulsate = amplitude * sin(time_f * frequency);
    float adjustedRadius = clamp(radius + pulsate, 0.0, 1.0);
    vec3 neonBlue = vec3(0.0, 1.0, 1.0);
    vec3 neonPink = vec3(1.0, 0.0, 0.5);
    vec3 gradientColor = mix(neonBlue, neonPink, adjustedRadius);\
    vec4 texColor = texture(samp, tc);
    vec3 finalColor = texColor.rgb * gradientColor;
    color = vec4(sin(finalColor * time_f), texColor.a);
}

```

Shader #575 - heat

File: heat.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float noise(vec2 st) {
    return fract(sin(dot(st.xy, vec2(12.9898,78.233))) * 43758.5453123);
}
void main(void) {
    vec2 uv = tc;
    float time = time_f * 0.5;
    float noiseValue = noise(uv * 10.0 + time);
    vec2 heatWaveOffset = vec2(noiseValue - 0.5, 0.0) * 0.02;
    vec2 nuv = uv + heatWaveOffset;
    vec4 texColor = texture(samp, nuv);
    color = texColor;
}
```

Shader #590 - index purple

File: index_purple.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 back_color = color;
    float x = gl_FragCoord.x / 16;
    float y = gl_FragCoord.y / 16;
    color[0] += (color[0] * 255 / x) / 255;
    color[1] = 0;
}
```



```

    color[2] += (color[2] * 255 / y) / 255;
    color[0] = color[0] * 0.8;
    color[2] = color[2] * 0.8;
}

```

Shader #592 - inflate

File: inflate.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float inflate = abs(sin(time_f * 10.0)) * 0.5 + 0.5;
    vec2 adjusted_tc = (tc - 0.5) * vec2(inflate, inflate) + 0.5;
    color = texture(samp, adjusted_tc);
}

```

Shader #595 - kale

File: kale.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = uv - center;
    float r = length(pos);
    float angle = atan(pos.y, pos.x);
    float numSegments = 6.0;
    float segmentAngle = 2.0 * 3.14159 / numSegments;
    angle = mod(angle, segmentAngle);
    angle = abs(angle - segmentAngle / 2.0);
    pos.x = r * cos(angle);
    pos.y = r * sin(angle);
    float rotationSpeed = time_f * 0.7;
    float cosA = cos(rotationSpeed);
    float sinA = sin(rotationSpeed);
    mat2 rot = mat2(cosA, -sinA, sinA, cosA);
    pos = rot * pos;
}

```

```

    vec4 texColor = texture(samp, pos + center);
    color = texColor;
}

```

Shader #596 - kale2

File: kale2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = uv - center;
    float r = length(pos);
    float angle = atan(pos.y, pos.x);
    float numSegments = 6.0;
    float segmentAngle = 2.0 * 3.14159 / numSegments;
    angle = mod(angle, segmentAngle);
    angle = abs(angle - segmentAngle / 2.0);
    pos.x = r * cos(angle);
    pos.y = r * sin(angle);
    float rotationSpeed = time_f * 0.7;
    float cosA = cos(rotationSpeed);
    float sinA = sin(rotationSpeed);
    mat2 rot = mat2(cosA, -sinA, sinA, cosA);
    pos = rot * pos;
    vec4 texColor = texture(samp, sin(pos * time_f) + center);
    color = texColor;
}

```

Shader #597 - kale3

File: kale3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {

```

```

    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = uv - center;
    float r = length(pos);
    float angle = atan(pos.y, pos.x);
    float numSegments = 6.0;
    float segmentAngle = 2.0 * 3.14159 / numSegments;
    angle = mod(angle, segmentAngle);
    angle = abs(angle - segmentAngle / 2.0);
    pos.x = r * cos(angle);
    pos.y = r * sin(angle);
    float rotationSpeed = time_f * 0.7;
    float cosA = cos(rotationSpeed);
    float sinA = sin(rotationSpeed);
    mat2 rot = mat2(cosA, -sinA, sinA, cosA);
    pos = rot * pos;
    vec4 texColor = texture(samp, tan(cos(pos * time_f)) + center);
    color = texColor;
}

```

Shader #598 - kale4

File: kale4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = uv - center;
    float r = length(pos);
    float angle = atan(pos.y, pos.x);
    float numSegments = 6.0;
    float segmentAngle = 2.0 * 3.14159 / numSegments;
    angle = mod(angle, segmentAngle);
    angle = abs(angle - segmentAngle / 2.0);
    pos.x = r * cos(angle);
    pos.y = r * sin(angle);
    float rotationSpeed = time_f * 0.3;
    float cosA = cos(rotationSpeed);
    float sinA = sin(rotationSpeed);
    mat2 rot = mat2(cosA, -sinA, sinA, cosA);
    pos = rot * sin(pos * time_f);
    vec4 texColor = texture(samp, pos + center);
    color = mix(texture(samp, tc), texColor, 0.5);
}

```

Shader #609 - lineinlineout

File: lineinlineout.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc;
    float centerY = 0.5 + 0.25 * sin(time_f);
    float shiftAmount = 0.1 * sin(time_f);
    if (uv.y > centerY) {
        uv.x += shiftAmount;
    } else {
        uv.x -= shiftAmount;
    }
    vec4 baseColor = texture(samp, uv);
    color = baseColor;
}
```

Shader #623 - magic 2

File: magic-2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec2 uv = tc;
    float x = uv.x + sin(uv.y * 10.0 + time_f * 5.0) * 0.05;
    float y = uv.y + cos(uv.x * 10.0 + time_f * 5.0) * 0.05;
    vec2 displacedUV = vec2(x, y);
    vec4 texColor = texture(samp, displacedUV);
    float haze = sin(uv.y * 20.0 + time_f * 10.0) * 0.5 + 0.5;
    vec4 hazeColor = vec4(0.8, 0.3, 1.0, 1.0) * haze;
    color = sin((texColor + hazeColor)*time_t);
}
```

Shader #624 - magic

File: magic.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float x = uv.x + sin(uv.y * 10.0 + time_f * 5.0) * 0.05;
    float y = uv.y + cos(uv.x * 10.0 + time_f * 5.0) * 0.05;
    vec2 displacedUV = vec2(x, y);
    vec4 texColor = texture(samp, displacedUV);
    float haze = sin(uv.y * 20.0 + time_f * 10.0) * 0.5 + 0.5;
    vec4 hazeColor = vec4(0.8, 0.3, 1.0, 1.0) * haze;
    color = texColor + hazeColor;
}
```

Shader #626 - matrix

File: matrix.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    float time_t = mod(time_f, 10.0);
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float refractionIndex = time_t * 0.5;
    float radius = 1.0 * tan(time_t);
    float distSquared = dot(normCoord, normCoord);
    normCoord *= 1.0 + refractionIndex * distSquared;
    vec2 coord = tan(normCoord * time_t) / vec2(iResolution.x / iResolution.y, 1.0) * 0.5 + 0.5;
    color = texture(samp, coord);
}
```

Shader #632 - mb

File: mb.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc * 2.0 - 1.0;
    uv *= iResolution.y / iResolution.x;
    vec2 center = vec2(0.0, 0.0);
    vec2 offset = uv - center;
    float dist = length(offset);
    float angle = atan(offset.y, offset.x);
    float pulse = sin(time_f * 2.0 + dist * 10.0) * 0.15;
    float spiral = sin(angle * 5.0 + time_f * 2.0) * 0.15;
    float stretch = cos(time_f * 3.0 - dist * 15.0) * 0.2;
    vec2 spiralOffset = vec2(cos(angle), sin(angle)) * spiral;
    vec2 stretchOffset = normalize(offset) * (pulse + stretch);
    vec2 morphUV = uv + spiralOffset + stretchOffset;
    vec4 texColor = texture(samp, morphUV * 0.5 + 0.5);
    vec3 blended = vec3(0.0);
    float weight = 0.0;
    for (float x = -2.0; x <= 2.0; x++) {
        for (float y = -2.0; y <= 2.0; y++) {
            vec2 sampleUV = morphUV + vec2(x, y) * 0.005;
            vec4 sampleColor = texture(samp, sampleUV * 0.5 + 0.5);
            blended += sampleColor.rgb;
            weight += 1.0;
        }
    }
    blended /= weight;
    blended = mix(texColor.rgb, blended, 0.5);
    float shimmer = sin(dist * 20.0 - time_f * 5.0) * 0.1 + 1.0;
    color = vec4(blended * shimmer, texColor.a);
}

```

Shader #633 - minimize

File: minimize.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright.

Technique: Distance field computation - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float ripple(vec2 pos, float time, float speed, float frequency) {
    float aspectRatio = iResolution.x / iResolution.y;
    pos.x *= aspectRatio;
    float d = distance(pos, vec2(0.5 * aspectRatio, 0.5));
    return sin(d * frequency - time * speed) * exp(-d * 3.0);
}
void main(void) {
    vec2 pos = tc;
    float time_t = mod(time_f, 3.5);

```

```

float x = ripple(pos, time_f, 0.5, 12.0);
pos = (pos * x) + (pos * time_t);
color = texture(samp, pos);
}

```

Shader #634 - mirdmd

File: mirdmd.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: UV mirroring/folding for symmetry - Driven by: resolution

```

#version 330 core
uniform sampler2D samp;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = uv * 2.0 - 1.0;
    centeredUV = abs(centeredUV);
    vec2 mirroredUV = mod(centeredUV, 1.0);
    vec4 texColor = texture(samp, mirroredUV);
    color = texColor;
}

```

Shader #635 - mirdmd1

File: mirdmd1.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = uv * 2.0 - 1.0;
    float speedFactorX = sin(time_f * 0.5) * 0.5 + 0.5;
    float speedFactorY = cos(time_f * 0.5) * 0.5 + 0.5;
    float angleX = time_f * (0.2 + speedFactorX * 0.8);
    float angleY = time_f * (0.2 + speedFactorY * 0.8);
    centeredUV.x = abs(centeredUV.x);
    centeredUV.y = abs(centeredUV.y);
    mat2 rotationX = mat2(cos(angleX), -sin(angleX), sin(angleX), cos(angleX));
    mat2 rotationY = mat2(cos(angleY), -sin(angleY), sin(angleY), cos(angleY));
    vec2 rotatedUV = centeredUV;
    rotatedUV.x = (rotationX * vec2(rotatedUV.x, 0.0)).x;
    rotatedUV.y = (rotationY * vec2(0.0, rotatedUV.y)).y;
    vec2 mirroredUV = mod(rotatedUV, 1.0);
    vec4 texColor = texture(samp, mirroredUV);
    float pulse = 0.5 + 0.5 * sin(time_f * 3.0);
    vec3 pulsatingColor = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 10.0),

```

```

        0.5 + 0.5 * cos(time_f + uv.y * 10.0),
        0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 10.0)
    ) * pulse;
    color = vec4(texColor.rgb * 0.7 + pulsatingColor * 0.3, texColor.a);
}

```

Shader #636 - mirdmdr

File: mirdmdr.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = uv * 2.0 - 1.0;
    float angle1 = time_f * 0.5;
    float angle2 = -time_f * 0.5;
    if (centeredUV.x >= 0.0) {
        mat2 rotation1 = mat2(cos(angle1), -sin(angle1), sin(angle1), cos(angle1));
        centeredUV = rotation1 * centeredUV;
    } else {
        mat2 rotation2 = mat2(cos(angle2), -sin(angle2), sin(angle2), cos(angle2));
        centeredUV = rotation2 * centeredUV;
    }
    centeredUV = abs(centeredUV);
    vec2 mirroredUV = mod(centeredUV, 1.0);
    vec4 texColor = texture(samp, mirroredUV);
    color = texColor;
}

```

Shader #640 - mirdmdt

File: mirdmdt.glsl

Applies geometric transformations including rotation and scaling to distort the image.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = uv * 2.0 - 1.0;
    float angle = time_f * 0.5;
    mat2 rotation = mat2(cos(angle), -sin(angle), sin(angle), cos(angle));
    if (centeredUV.x < 0.0) {
        centeredUV.x = -centeredUV.x;
    }
    centeredUV = rotation * centeredUV;
}

```



```

    } else {
        centeredUV = rotation * centeredUV;
    }
    centeredUV = mod(centeredUV, 1.0);
    vec2 mirroredUV = centeredUV * 0.5 + 0.5;
    vec4 texColor = texture(samp, mirroredUV);
    color = texColor;
}

```

Shader #716 - mix diag

File: mix_diag.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Linear mix/lerp blending - Driven by: time

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main() {
    color = texture(samp, tc);
    vec2 uv = tc;
    uv.x -= 0.05;
    uv.y -= 0.05;
    vec4 color2 = texture(samp, uv);
    color = mix(color, color2, 0.5);
}

```

Shader #717 - move

File: move.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 uv = tc - center;
    float dist = length(uv);
    float ripple = sin(10.0 * dist - time_f * 6.28318) * 0.1;
    uv += uv * sin(ripple * time_f);
    uv += center;
    color = texture(samp, uv);
}

```

Shader #718 - moveGL

File: moveGL.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 uv = tc - center;
    float dist = length(uv);
    float ripple = sin(10.0 * dist - time_f * 6.28318) * 0.1;
    uv += uv * sin(ripple * time_f);
    uv += center * atan(dist * time_f);
    color = texture(samp, uv);
}
```

Shader #720 - negative

File: negative.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec4 texColor = texture(samp, tc);
    color = vec4(1.0 - texColor.rgb, texColor.a);
}
```

Shader #728 - nofilter

File: nofilter.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
```

```

uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
uniform vec2 image_pos;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}
void main(void)
{
    color = texture(samp, tc);
}

```

Shader #729 - nothing

File: nothing.glsl

Processes the input texture to produce the "nothing" visual effect.

Technique: Direct texture color manipulation with arithmetic operations

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
void main(void) {
    color = texture(samp, tc);
}

```

Shader #738 - old film

File: old-film.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
 Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co){
    return fract(sin(dot(co.xy ,vec2(12.9898,78.233))) * 43758.5453);
}
void main(void) {

```

```

    vec2 uv = tc;
    vec4 texColor = texture(samp, uv);
    float noise = rand(uv * time_f) * 0.1;
    float line = step(0.995, rand(vec2(uv.y * 100.0, time_f * 0.1)));
    texColor.rgb += noise;
    texColor.rgb = mix(texColor.rgb, vec3(0.0), line * 0.5);
    color = texColor;
}

```

Shader #739 - onoffsep

File: onoffsep.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
void main() {
    vec2 uv = tc;
    float lineWidth = 0.01 + 0.005 * sin(time_f * 2.0);
    float splitOffset = 0.05 * sin(time_f * 1.5);
    float linePosition = abs(fract(uv.y * 10.0 + time_f) - 0.5);
    float frameSep = mod(floor(uv.x * 5.0) + floor(uv.y * 5.0), 2.0);
    vec2 displacedUV = uv;
    if (frameSep == 0.0) {
        displacedUV.x += splitOffset;
    } else {
        displacedUV.x -= splitOffset;
    }
    vec4 texColor = texture(samp, displacedUV);
    if (linePosition < lineWidth) {
        texColor.rgb += vec3(1.0, 0.0, 0.0) * abs(sin(time_f));
    }
    color = texColor;
}

```

Shader #740 - open shadow

File: open_shadow.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float trailLength = 1.0;
    float trailOffset = 0.05 * pingPong(time_f, trailLength);
    vec2 offset = vec2(sin(time_f), cos(time_f)) * trailOffset;
    vec4 shadowColor = texture(samp, tc - offset);
    vec4 currentColor = texture(samp, tc);
    color = mix(currentColor, shadowColor, 0.5);
}

```

Shader #741 - open shadow jump

File: open_shadow_jump.glsl

Blends multiple texture samples together using interpolation for a composite effect.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float trailLength = sin(8.0 * time_f);
    vec4 currentColor = texture(samp, tc);
    vec4 shadowColor1 = texture(samp, tc - vec2(sin(time_f), cos(time_f)) * 0.02 * pingPong(time_f,
↪ trailLength));
    vec4 shadowColor2 = texture(samp, tc - vec2(sin(time_f + 1.0), cos(time_f + 1.0)) * 0.04 * pingPong(
↪ time_f + 1.0, trailLength));
    vec4 shadowColor3 = texture(samp, tc - vec2(sin(time_f + 2.0), cos(time_f + 2.0)) * 0.06 * pingPong(
↪ time_f + 2.0, trailLength));
    vec4 shadowColor = (shadowColor1 + shadowColor2 + shadowColor3) / 3.0;
    color = mix(currentColor, shadowColor, 0.5);
}

```

Shader #742 - open shadow multi

File: open_shadow_multi.glsl

Blends multiple texture samples together using interpolation for a composite effect.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float trailLength = 8.0;
    vec4 currentColor = texture(samp, tc);
    vec4 shadowColor1 = texture(samp, tc - vec2(sin(time_f), cos(time_f)) * 0.02 * pingPong(time_f,
    ↪ trailLength));
    vec4 shadowColor2 = texture(samp, tc - vec2(sin(time_f + 1.0), cos(time_f + 1.0)) * 0.04 * pingPong(
    ↪ time_f + 1.0, trailLength));
    vec4 shadowColor3 = texture(samp, tc - vec2(sin(time_f + 2.0), cos(time_f + 2.0)) * 0.06 * pingPong(
    ↪ time_f + 2.0, trailLength));
    vec4 shadowColor = (shadowColor1 + shadowColor2 + shadowColor3) / 3.0;
    color = mix(currentColor, shadowColor, 0.5);
}

```

Shader #743 - open shadow multi x2

File: open_shadow_multi_x2.glsl

Blends multiple texture samples together using interpolation for a composite effect.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float trailLength = 16.0;
    vec4 currentColor = texture(samp, tc);
    vec4 shadowColor1 = texture(samp, tc - vec2(sin(time_f), cos(time_f)) * 0.02 * pingPong(time_f,
    ↪ trailLength));
    vec4 shadowColor2 = texture(samp, tc - vec2(sin(time_f + 1.0), cos(time_f + 1.0)) * 0.04 * pingPong(
    ↪ time_f + 1.0, trailLength));
    vec4 shadowColor3 = texture(samp, tc - vec2(sin(time_f + 2.0), cos(time_f + 2.0)) * 0.06 * pingPong(
    ↪ time_f + 2.0, trailLength));
    vec4 shadowColor = (shadowColor1 + shadowColor2 + shadowColor3) / 3.0;
    color = mix(currentColor, shadowColor, 0.9);
}

```

Shader #744 - open up

File: open_up.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float dist = length(uv);
    float angle = atan(uv.y, uv.x);
    float centerSize = 0.5 + 0.4 * sin(time_f * 3.0);
    float bloomEffect = sin(time_f + dist * 6.0) * 0.5 + 0.5;
    vec2 centerUV = uv * mix(1.0, bloomEffect, 0.3);
    vec2 outerUV = uv * (1.0 + 0.5 * (1.0 - dist));
    centerUV = centerUV * 0.5 + 0.5;
    outerUV = outerUV * 0.5 + 0.5;
    vec4 centerColor = texture(samp, centerUV);
    vec4 outerColor = texture(samp, outerUV);
    float mixFactor = smoothstep(centerSize - 0.1, centerSize, dist);
    color = mix(centerColor, outerColor, mixFactor);
    color = mix(color, texture(samp, tc), 0.5);
}
```

Shader #745 - optxttime

File: optxttime.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
}
```

```

    color[0] = color[0]+(timeval*optx[0]);
    color[1] = color[1]+(timeval*optx[1]);
    color[2] = color[2]*tc[0]+(timeval*optx[2]);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #746 - optxtime cos

File: optxtime_cos.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = color[0] + (cos(optx[0])*timeval);
    color[1] = color[1] + (cos(optx[1])*timeval);
    color[2] = color[2] + (cos(optx[2])*timeval);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #747 - optxtime tex

File: optxtime_tex.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = color[0]*(0.142*vpos[0])+(timeval*optx[0]);
    color[1] = color[1]*(0.341*vpos[1])+(timeval*optx[1]);
    color[2] = color[2]*(0.451*vpos[2])+(timeval*optx[2]);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #748 - page turn

File: page_turn.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float foldAmount = sin(time_f) * 0.5 + 0.5;
    vec2 uv = tc;
    if (uv.x > foldAmount) {
        float offset = (uv.x - foldAmount) / (1.0 - foldAmount);
        uv.x = foldAmount + offset * foldAmount;
    }
    color = texture(samp, uv);
}

```

Shader #749 - page turn2

File: page_turn2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0) + 2.0;
    float foldAmount = sin(time_t) * 0.5 + 0.5;
    vec2 uv = tc;
    if (uv.x > foldAmount) {
        float offset = (uv.x - foldAmount) / (1.0 - foldAmount);
        uv.x = foldAmount + offset * foldAmount;
    }
    color = texture(samp, uv);
}

```

Shader #750 - page turn3

File: page_turn3.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);

```

```

    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0) + 2.0;
    float foldAmount = sin(time_t) * 0.5 + 0.5;
    vec2 uv = tc;
    if (uv.x > foldAmount) {
        float offset = (uv.x - foldAmount) / (1.0 - foldAmount);
        uv.x = sin(foldAmount + offset * foldAmount * time_t);
    }
    color = texture(samp, uv);
}

```

Shader #751 - particle 1

File: particle_1.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec2 hash(vec2 p) {
    p = vec2(dot(p, vec2(127.1, 311.7)), dot(p, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(p) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f*f*(3.0-2.0*f);
    return mix(mix(dot(hash(i + vec2(0.0, 0.0))), f - vec2(0.0, 0.0)), dot(hash(i + vec2(1.0, 0.0)), f -
    ↪ vec2(1.0, 0.0)), u.x), mix(dot(hash(i + vec2(0.0, 1.0))), f - vec2(0.0, 1.0)), dot(hash(i + vec2
    ↪ (1.0, 1.0)), f - vec2(1.0, 1.0)), u.x), u.y);
}
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    vec3 col = vec3(0.0);
    float numParticles = 50.0;
    for (float i = 0.0; i < numParticles; i += 1.0) {
        float t = time_f * 0.1 + i * 6.28 / numParticles;
        vec2 pos = vec2(sin(t + noise(vec2(i, t)) * 10.0), cos(t + noise(vec2(i, t)) * 10.0)) * 0.3 +
        ↪ 0.5;
        float d = length(uv - pos);
        col += vec3(0.1, 0.5, 0.8) * (1.0 - d) * exp(-d * 10.0);
    }
    color = vec4(col, 1.0);
    color = mix(texture(samp, tc), color, 0.5);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = sin(color * time_t);
    color.a = 1.0;
}

```

```
}
```

Shader #752 - pastel

File: pastel.glsl

Applies an iterative, time-animated effect that evolves the image over time.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rgb2hsb(vec3 c) {
    vec4 K = vec4(0.0, -1.0/3.0, 2.0/3.0, -1.0);
    vec4 p = mix(vec4(c.bg, K.wz), vec4(c.gb, K.xy), step(c.b, c.g));
    vec4 q = mix(vec4(p.xyw, c.r), vec4(c.r, p.yzx), step(p.x, c.r));
    float d = q.x - min(q.w, q.y);
    float e = 1.0e-10;
    return vec3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e), q.x);
}
vec3 hsb2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main() {
    vec2 uv = tc;
    vec4 texColor = texture(samp, uv);
    vec2 texelSize = 1.0 / iResolution;
    vec3 result = vec3(0.0);
    float weight[5];
    weight[0] = 0.227027;
    weight[1] = 0.1945946;
    weight[2] = 0.1216216;
    weight[3] = 0.054054;
    weight[4] = 0.016216;
    for (int i = -4; i <= 4; i++) {
        result += texture(samp, uv + vec2(float(i) * texelSize.x, 0.0)).rgb * weight[abs(i)];
    }
    for (int j = -4; j <= 4; j++) {
        result += texture(samp, uv + vec2(0.0, float(j) * texelSize.y)).rgb * weight[abs(j)];
    }
    vec3 hsb = rgb2hsb(result / 2.0);
    hsb.x += time_f * 0.1;
    hsb.y = 0.5;
    hsb.z = 0.9;
    vec3 rgb = hsb2rgb(hsb);
    color = vec4(rgb, texColor.a);
}
```

Shader #753 - pebble

File: pebble.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float dist = length(normCoord);
    float maxRippleRadius = 8.0;
    float rippleSpeed = 0.5;
    float phase = mod(time_f * rippleSpeed, maxRippleRadius);
    float ripple = sin((dist - phase) * 10.0) * exp(-dist * 3.0);
    vec2 displacedCoord = vec2(tc.x, tc.y + ripple * sin(time_f));
    color = texture(samp, displacedCoord);
}
```

Shader #754 - pebble fast

File: pebble_fast.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float dist = length(normCoord);
    float maxRippleRadius = 25.0;
    float rippleSpeed = 2.0;
    float phase = mod(time_f * rippleSpeed, maxRippleRadius);
    float ripple = sin((dist - phase) * 10.0) * exp(-dist * 3.0);
    vec2 displacedCoord = vec2(tc.x, tc.y + ripple * sin(time_f));
    color = texture(samp, displacedCoord);
}
```

Shader #760 - pinch

File: pinch.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
```

```

uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    uv -= center;
    float dist = length(uv);
    float time_t = pingPong(time_f, 10.0);
    float pinchAmount = 1.5 * sin(time_t);
    float radius = 0.5;
    if (dist < radius) {
        uv *= 1.0 - (pinchAmount * (1.0 - dist / radius));
    }
    uv += center;
    color = texture(samp, uv);
}

```

Shader #762 - plasma

File: plasma.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
float PI = 3.1415926535897932384626433832795;
uniform sampler2D samp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main() {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float plasma = 0.0;
    float pFreq = 5.0 + amp_mid * 10.0;
    plasma += sin((uv.x + time_f) * pFreq);
    plasma += sin((uv.y + time_f) * pFreq);
    plasma += sin((uv.x + uv.y + time_f) * pFreq);
    plasma += cos(length(uv + time_f) * (10.0 + amp_low * 15.0));
    plasma *= 0.25;
    vec3 col;
    col.r = cos(plasma * PI + time_f * 0.2) * 0.5 + 0.5;
    col.g = sin(plasma * PI + time_f * 0.2) * 0.5 + 0.5;
    col.b = sin(plasma * PI + time_f * 0.4) * 0.5 + 0.5;
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    col *= 1.0 + _ab * 0.6;
}

```

```

    col = mix(col, col * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high, 0.0, 1.0) *
    ↪ 0.25), _ab);
    color = vec4(col, 1.0);
    color = mix(color, texture(samp, tc), 0.5);
}

```

Shader #763 - plasma2

File: plasma2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
float PI = 3.1415926535897932384626433832795;
void main() {
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 p = (tc - 0.5) * ar;
    float r = max(length(p), 1e-4);
    float a = atan(p.y, p.x);
    float t = time_f;
    float pulse = 0.5 + 0.5 * sin(t * 2.0);
    float zoomSpeed = 0.35 + 0.15 * sin(t * 0.7);
    float swirl = 0.05 * sin(6.0 * a + t * 2.0 + r * 10.0);
    float wobble = 0.10 * sin(a * 8.0 + t * 1.5);
    float lr = log(r);
    lr += zoomSpeed * t;
    lr = fract(lr * (1.0 + 0.25 * pulse));
    lr = (lr - 0.5) * 2.0 + wobble;
    a += swirl;
    float nr = exp(lr);
    vec2 q = vec2(cos(a), sin(a)) * nr;
    vec2 baseUV = q / ar + 0.5;
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= ar.x;
    float plasma = 0.0;
    plasma += sin((uv.x + t) * 5.0);
    plasma += sin((uv.y + t) * 5.0);
    plasma += sin((uv.x + uv.y + t) * 5.0);
    plasma += cos(length(uv + t) * 10.0);
    plasma *= 0.25;
    vec3 baseColor;
    baseColor.r = cos(plasma * PI + t * 0.2) * 0.5 + 0.5;
    baseColor.g = sin(plasma * PI + t * 0.2) * 0.5 + 0.5;
    baseColor.b = sin(plasma * PI + t * 0.4) * 0.5 + 0.5;
    vec2 dir = normalize(q + 1e-5);
    float disp = 0.002 + 0.01 * pulse;
    float rC = texture(samp, baseUV + dir * disp).r;
    float gC = texture(samp, baseUV).g;
    float bC = texture(samp, baseUV - dir * disp).b;
    vec3 prismColor = vec3(rC, gC, bC);
    float morph = 0.5 + 0.5 * sin(t * 1.1);
}

```

```

    vec3 mixed = mix(baseColor, prismColor, 0.6 + 0.4 * morph);
    float breathe = 0.5 + 0.5 * sin(t * 3.0 + length(q) * 12.0);
    mixed *= 0.9 + 0.2 * breathe;
    color = vec4(mixed, 1.0);
}

```

Shader #764 - plasma3

File: plasma3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main() {
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 p = (tc - 0.5) * ar;
    float r = max(length(p), 1e-4);
    float a = atan(p.y, p.x);
    float t = time_f;
    float pulse = 0.5 + 0.5 * sin(t * 2.0);
    float zoomSpeed = 0.35 + 0.15 * sin(t * 0.7);
    float swirl = 0.05 * sin(6.0 * a + t * 2.0 + r * 10.0);
    float wobble = 0.10 * sin(a * 8.0 + t * 1.5);
    float lr = log(r);
    lr += zoomSpeed * t;
    lr = fract(lr * (1.0 + 0.25 * pulse));
    lr = (lr - 0.5) * 2.0 + wobble;
    a += sin(swirl * (pingPong(time_f, 10.0) * PI));
    float nr = exp(lr);
    vec2 q = vec2(cos(a), sin(a)) * nr;
    vec2 baseUV = q / ar + 0.5;
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= ar.x;
    float plasma = 0.0;
    plasma += sin((uv.x + t) * 5.0);
    plasma += sin((uv.y + t) * 5.0);
    plasma += sin((uv.x + uv.y + t) * 5.0);
    plasma += cos(length(uv + t) * 10.0);
    plasma *= 0.25;
    vec3 baseColor;
    baseColor.r = cos(plasma * PI + t * 0.2) * 0.5 + 0.5;
    baseColor.g = sin(plasma * PI + t * 0.2) * 0.5 + 0.5;
    baseColor.b = sin(plasma * PI + t * 0.4) * 0.5 + 0.5;
    vec2 dir = normalize(q + 1e-5);
    float disp = 0.002 + 0.01 * pulse;
    float rC = texture(samp, baseUV + dir * disp).r;
}

```



```

float gC = texture(samp, baseUV).g;
float bC = texture(samp, baseUV - dir * disp).b;
vec3 prismColor = vec3(rC, gC, bC);
float morph = 0.5 + 0.5 * sin(t * 1.1);
vec3 mixed = mix(baseColor, prismColor, 0.6 + 0.4 * morph);
float breathe = 0.5 + 0.5 * sin(t * 3.0 + length(q) * 12.0);
mixed *= 0.9 + 0.2 * sin(breathe * pingPong(time_f, 10.0));
color = vec4(mixed, 1.0);
}

```

Shader #765 - plasma prism

File: plasma_prism.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
float PI = 3.1415926535897932384626433832795;
void main() {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float plasma = 0.0;
    plasma += sin((uv.x + time_f) * 5.0);
    plasma += sin((uv.y + time_f) * 5.0);
    plasma += sin((uv.x + uv.y + time_f) * 5.0);
    plasma += cos(length(uv + time_f) * 10.0);
    plasma *= 0.25;
    vec3 baseColor;
    baseColor.r = cos(plasma * PI + time_f * 0.2) * 0.5 + 0.5;
    baseColor.g = sin(plasma * PI + time_f * 0.2) * 0.5 + 0.5;
    baseColor.b = sin(plasma * PI + time_f * 0.4) * 0.5 + 0.5;
    float dispersion = 0.02;
    vec3 prismColor;
    prismColor.r = texture(samp, tc + vec2(dispersion, 0.0)).r;
    prismColor.g = texture(samp, tc).g;
    prismColor.b = texture(samp, tc - vec2(dispersion, 0.0)).b;
    color = vec4(mix(baseColor, prismColor, 0.6), 1.0);
}

```

Shader #768 - pong ataan ex

File: pong-ataan-ex.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;

```

```

in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float h1(float n){ return fract(sin(n)*43758.5453123); }
vec2 h2(float n){ return fract(sin(vec2(n,n+1.0))*vec2(43758.5453,22578.1459)); }
void main(void) {
    float rate = 0.35;
    float t = time_f*rate;
    float k = floor(t);
    float a = fract(t);
    a = a*a*(3.0-2.0*a);
    vec2 p0 = h2(k)*2.0-1.0;
    vec2 p1 = h2(k+1.0)*2.0-1.0;
    vec2 shift = mix(p0,p1,a)*0.12;
    float s0 = 0.9 + 0.2*h1(k+10.0);
    float s1 = 0.9 + 0.2*h1(k+11.0);
    float scale = mix(s0,s1,a);
    float dir = sign(h1(k+20.0)-0.5);
    float modulatedTime = pingPong(time_f + h1(k+30.0)*2.0, 5.0);
    float rot = dir * modulatedTime;
    vec2 center = vec2(0.5) + shift;
    vec2 d = tc - center;
    float ang = atan(d.y, d.x) + rot;
    float r = length(d)/scale;
    vec2 rotatedTC = center + r*vec2(cos(ang), sin(ang));
    rotatedTC = fract(rotatedTC);
    color = texture(samp, rotatedTC);
}

```

Shader #769 - pong atan

File: pong-atan.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float angle = atan(tc.y - 0.5, tc.x - 0.5);
    float modulatedTime = pingPong(time_f, 5.0);
    angle += modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle) * (tc.x - 0.5) - sin(angle) * (tc.y - 0.5) + 0.5;
    rotatedTC.y = sin(angle) * (tc.x - 0.5) + cos(angle) * (tc.y - 0.5) + 0.5;
    color = texture(samp, rotatedTC);
}

```

```
}
```

Shader #770 - pong atan2

File: pong-atan2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float angle1 = atan(tc.y - 0.5, tc.x - 0.5);
    float modulatedTime1 = pingPong(time_f, 3.0);
    angle1 += modulatedTime1;
    float angle2 = atan(tc.x - 0.5, tc.y - 0.5);
    float modulatedTime2 = pingPong(time_f * 0.5, 2.5);
    angle2 += modulatedTime2;
    float angle3 = atan(tc.y - 0.5 + modulatedTime2, tc.x - 0.5 + modulatedTime1);
    float modulatedTime3 = pingPong(time_f * 1.5, 4.0);
    angle3 += modulatedTime3;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle3) * (tc.x - 0.5) - sin(angle3) * (tc.y - 0.5) + 0.5;
    rotatedTC.y = sin(angle3) * (tc.x - 0.5) + cos(angle3) * (tc.y - 0.5) + 0.5;
    rotatedTC = sin(rotatedTC * (modulatedTime1 * modulatedTime2));
    color = texture(samp, rotatedTC);
}
```

Shader #771 - pong atan3

File: pong-atan3.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float angle1 = atan(tc.y - 0.5, tc.x - 0.5);
    float modulatedTime1 = pingPong(time_f, 3.0);
```

```

angle1 += modulatedTime1;
float angle2 = atan(tc.x - 0.5, tc.y - 0.5);
float modulatedTime2 = pingPong(time_f * 0.5, 2.5);
angle2 += modulatedTime2;
float angle3 = atan(tc.y - 0.5 + modulatedTime2, tc.x - 0.5 + modulatedTime1);
float modulatedTime3 = pingPong(time_f * 1.5, 4.0);
angle3 += modulatedTime3;
vec2 rotatedTC;
rotatedTC.x = cos(angle3) * (tc.x - 0.5) - sin(angle3) * (tc.y - 0.5) + 0.5;
rotatedTC.y = sin(angle3) * (tc.x - 0.5) + cos(angle3) * (tc.y - 0.5) + 0.5;
rotatedTC = sin(rotatedTC * (modulatedTime1 * modulatedTime2 * modulatedTime3));
color = texture(samp, rotatedTC);
}

```

Shader #772 - pong pi.mp4

File: pong_pi.mp4.glsl

Renders a fractal pattern using iterative computation (many iterations).

Technique: Iterative fractal computation with complex-plane mapping - Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (tc - 0.5) * ar;
    float base = 1.65;
    float period = log(base);
    float t = time_f * 0.45;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float k = fract((log(r) - t) / period);
    float rw = exp(k * period);
    a += t * 0.4;
    vec2 z = vec2(cos(a), sin(a)) * rw;
    float swirl = sin(pingPong(time_f * PI, 10.0) * 0.6 + r * 5.0) * 0.1;
    z += swirl * normalize(p);
    vec2 uv = z / ar + 0.5;
    uv = fract(uv);
    vec4 tex = texture(samp, uv);
    color = tex;
}

```

Shader #773 - pong tex

File: pong_tex.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0;
    uv.x = pingPong(uv.x, 1.0);
    uv.y = pingPong(uv.y, 1.0);
    float frequency = 10.0;
    float amplitude = 0.05;
    uv += sin(uv * frequency + time_f) * amplitude;
    vec4 texColor = texture(samp, uv);
    color = texColor;
}
```

Shader #774 - pool

File: pool.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float time_t = mod(time_f, 100);
    float scale = time_t;
    float speed = 16.0;
    float offset = sin(time_f * speed + tc.x * scale) * 0.05;
    vec2 tcOffset = vec2(tc.x, tc.y + offset);
    color = texture(samp, tcOffset);
}
```

Shader #776 - prisim glass triangle

File: prisim_glass_triangle.glsl

Applies an iterative, time-animated effect that evolves the image over time. High-fidelity spectral helper for the rainbow Apex is at (0.5, 1.0) -> Top Center Base spans from (0.0, 0.0) to (1.0, 0.0) -> Bottom corners

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 spectrum(float w) {
    vec3 c;
    if (w < 0.2) c = mix(vec3(1.0, 0.0, 1.0), vec3(0.0, 0.0, 1.0), w * 5.0);
    else if (w < 0.4) c = mix(vec3(0.0, 0.0, 1.0), vec3(0.0, 1.0, 0.0), (w - 0.2) * 5.0);
    else if (w < 0.6) c = mix(vec3(0.0, 1.0, 0.0), vec3(1.0, 1.0, 0.0), (w - 0.4) * 5.0);
    else if (w < 0.8) c = mix(vec3(1.0, 1.0, 0.0), vec3(1.0, 0.0, 0.0), (w - 0.6) * 5.0);
    else c = vec3(1.0, 0.0, 0.0);
    return c;
}
void main(void) {
    vec2 uv = tc;
    float wedge = 1.0 - 2.0 * abs(uv.x - 0.5);
    if (uv.y < wedge && uv.y > 0.0) {
        vec3 accumulatedColor = vec3(0.0);
        float dispersion = 0.12;
        int samples = 24;
        vec2 refractDir = normalize(uv - vec2(0.5, 1.0));
        for (int i = 0; i < samples; i++) {
            float w = float(i) / float(samples - 1);
            float shift = (w - 0.5) * dispersion;
            vec2 sampleUV = uv + (refractDir * shift);
            accumulatedColor += texture(samp, clamp(sampleUV, 0.0, 1.0)).rgb * spectrum(w);
        }
        vec3 finalRGB = accumulatedColor / (float(samples) * 0.5);
        finalRGB *= 1.1;
        finalRGB += vec3(0.03, 0.04, 0.05);
        color = vec4(finalRGB, 1.0);
        float edge = smoothstep(0.015, 0.0, abs(uv.y - wedge));
        color.rgb += edge * 0.4;
    } else {
        color = texture(samp, uv);
    }
}
```

Shader #777 - prism

File: prism.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    float radius = 1.0;
    vec2 center = iResolution * 0.5;
```

```

vec2 texCoord = tc * iResolution;
vec2 delta = texCoord - center;
float dist = length(delta);
float maxRadius = min(iResolution.x, iResolution.y) * radius;
if (dist < maxRadius) {
    float scaleFactor = 1.0 - pow(dist / maxRadius, 2.0);
    vec2 direction = normalize(delta);
    float offsetR = 0.015;
    float offsetG = 0.0;
    float offsetB = -0.015;
    vec2 texCoordR = center + delta * scaleFactor + direction * offsetR * maxRadius;
    vec2 texCoordG = center + delta * scaleFactor + direction * offsetG * maxRadius;
    vec2 texCoordB = center + delta * scaleFactor + direction * offsetB * maxRadius;
    texCoordR /= iResolution;
    texCoordG /= iResolution;
    texCoordB /= iResolution;
    texCoordR = clamp(texCoordR, 0.0, 1.0);
    texCoordG = clamp(texCoordG, 0.0, 1.0);
    texCoordB = clamp(texCoordB, 0.0, 1.0);
    float r = texture(samp, texCoordR).r;
    float g = texture(samp, texCoordG).g;
    float b = texture(samp, texCoordB).b;
    color = vec4(r, g, b, 1.0);
} else {
    vec2 newTexCoord = clamp(tc, 0.0, 1.0);
    color = texture(samp, newTexCoord);
}
}

```

Shader #778 - prism glass

File: prism_glass.glsl

Applies an iterative, time-animated effect that evolves the image over time. Signed Distance Function for an equilateral triangle Prism Parameters How much the light "spreads"

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 spectrum(float w) {
    vec3 c;
    if (w < 0.2) c = mix(vec3(1,0,1), vec3(0,0,1), w * 5.0);
    else if (w < 0.4) c = mix(vec3(0,0,1), vec3(0,1,0), (w-0.2) * 5.0);
    else if (w < 0.6) c = mix(vec3(0,1,0), vec3(1,1,0), (w-0.4) * 5.0);
    else if (w < 0.8) c = mix(vec3(1,1,0), vec3(1,0,0), (w-0.6) * 5.0);
    else c = vec3(1,0,0);
    return c;
}
float sdTriangle(vec2 p, float r) {
    const float k = sqrt(3.0);
    p.x = abs(p.x) - r;
    p.y = p.y + r/k;
    if (p.x + k*p.y > 0.0) p = vec2(p.x - k*p.y, -k*p.x - p.y) / 2.0;
    p.x -= clamp(p.x, -2.0*r, 0.0);
}

```

```

    return -length(p) * sign(p.y);
}
void main(void) {
    vec2 uv = tc;
    vec2 p = (tc - 0.5) * iResolution.xy / iResolution.y;
    float prismSize = 0.3;
    float dispersion = 0.08;
    int samples = 12;
    float d = sdTriangle(p, prismSize);
    if (d < 0.0) {
        vec3 accumulatedColor = vec3(0.0);
        float totalWeight = 0.0;
        vec2 refractDir = normalize(p - vec2(0.0, -0.1));
        for (int i = 0; i < samples; i++) {
            float w = float(i) / float(samples - 1);
            float shift = (w - 0.5) * dispersion;
            vec2 sampleUV = uv + refractDir * shift;
            vec3 spec = spectrum(w);
            accumulatedColor += texture(samp, clamp(sampleUV, 0.0, 1.0)).rgb * spec;
            totalWeight += 1.3;
        }
        color = vec4(accumulatedColor / (float(samples) * 0.5), 1.0);
        color.rgb += vec3(0.05, 0.07, 0.1);
    } else {
        color = texture(samp, uv);
        float edge = smoothstep(0.01, 0.0, abs(d));
        color.rgb += edge * 0.4;
    }
}

```

Shader #780 - prism glass wall

File: prism_glass_wall.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections.

High-fidelity spectral helper Line from (0,0) to (0.5, 1) Line from (1,0) to (0.5, 1)

Technique: Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 spectrum(float w) {
    vec3 c;
    if (w < 0.2) c = mix(vec3(1.0, 0.0, 1.0), vec3(0.0, 0.0, 1.0), w * 5.0);
    else if (w < 0.4) c = mix(vec3(0.0, 0.0, 1.0), vec3(0.0, 1.0, 0.0), (w - 0.2) * 5.0);
    else if (w < 0.6) c = mix(vec3(0.0, 1.0, 0.0), vec3(1.0, 1.0, 0.0), (w - 0.4) * 5.0);
    else if (w < 0.8) c = mix(vec3(1.0, 1.0, 0.0), vec3(1.0, 0.0, 0.0), (w - 0.6) * 5.0);
    else c = vec3(1.0, 0.0, 0.0);
    return c;
}
void main(void) {
    vec2 uv = tc;
    vec2 refractDir;
    float edge1 = 2.0 * uv.x;
    float edge2 = -2.0 * uv.x + 2.0;
}

```



```

    if (uv.y > edge1) {
        refractDir = normalize(uv - vec2(0.0, 1.0));
    } else if (uv.y > edge2) {
        refractDir = normalize(uv - vec2(1.0, 1.0));
    } else {
        refractDir = normalize(uv - vec2(0.5, 0.0));
    }
    vec3 accumulatedColor = vec3(0.0);
    float dispersion = 0.07;
    int samples = 18;
    for (int i = 0; i < samples; i++) {
        float w = float(i) / float(samples - 1);
        float shift = (w - 0.5) * dispersion;
        vec2 sampleUV = uv + (refractDir * shift);
        accumulatedColor += texture(samp, clamp(sampleUV, 0.0, 1.0)).rgb * spectrum(w);
    }
    vec3 finalRGB = accumulatedColor / (float(samples) * 0.5);
    finalRGB += vec3(0.02, 0.03, 0.04);
    float line1 = smoothstep(0.005, 0.0, abs(uv.y - edge1));
    float line2 = smoothstep(0.005, 0.0, abs(uv.y - edge2));
    finalRGB += (line1 + line2) * 0.5;
    color = vec4(finalRGB, 1.0);
}

```

Shader #782 - prism quad

File: prism_quad.glsl

Applies an iterative, time-animated effect that evolves the image over time. High-fidelity spectral helper for smooth rainbow gradients Instead of triangle edges, we use the center of the screen (0.5, 0.5) as the "apex" of the refraction.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 spectrum(float w) {
    vec3 c;
    if (w < 0.2) c = mix(vec3(1.0, 0.0, 1.0), vec3(0.0, 0.0, 1.0), w * 5.0);
    else if (w < 0.4) c = mix(vec3(0.0, 0.0, 1.0), vec3(0.0, 1.0, 0.0), (w - 0.2) * 5.0);
    else if (w < 0.6) c = mix(vec3(0.0, 1.0, 0.0), vec3(1.0, 1.0, 0.0), (w - 0.4) * 5.0);
    else if (w < 0.8) c = mix(vec3(1.0, 1.0, 0.0), vec3(1.0, 0.0, 0.0), (w - 0.6) * 5.0);
    else c = vec3(1.0, 0.0, 0.0);
    return c;
}
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = uv - center;
    float dist = length(dir);
    vec2 refractDir = normalize(dir);
    vec3 accumulatedColor = vec3(0.0);
    float dispersionBase = 0.09;
    float dispersion = dispersionBase * dist;
    int samples = 24;
    for (int i = 0; i < samples; i++) {

```

```

        float w = float(i) / float(samples - 1);
        float shift = (w - 0.5) * dispersion;
        vec2 sampleUV = uv + (refractDir * shift);
        accumulatedColor += texture(samp, clamp(sampleUV, 0.0, 1.0)).rgb * spectrum(w);
    }
    vec3 finalRGB = accumulatedColor / (float(samples) * 0.5);
    finalRGB *= 1.05;
    finalRGB *= 1.0 - (dist * 0.1);
    color = vec4(finalRGB, 1.0);
}

```

Shader #783 - prism quad by time

File: prism_quad_by_time.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright. Spectral helper for the rainbow smear Change '1.5' to adjust the speed of the oscillation Remap pulse to dispersion range (0.02 is subtle, 0.25 is extreme)

Technique: Linear mix/lerp blending - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 spectrum(float w) {
    vec3 c;
    if (w < 0.2) c = mix(vec3(1.0, 0.0, 1.0), vec3(0.0, 0.0, 1.0), w * 5.0);
    else if (w < 0.4) c = mix(vec3(0.0, 0.0, 1.0), vec3(0.0, 1.0, 0.0), (w - 0.2) * 5.0);
    else if (w < 0.6) c = mix(vec3(0.0, 1.0, 0.0), vec3(1.0, 1.0, 0.0), (w - 0.4) * 5.0);
    else if (w < 0.8) c = mix(vec3(1.0, 1.0, 0.0), vec3(1.0, 0.0, 0.0), (w - 0.6) * 5.0);
    else c = vec3(1.0, 0.0, 0.0);
    return pow(c, vec3(0.8));
}
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = uv - center;
    float dist = length(dir);
    vec2 refractDir = normalize(dir);
    float pulse = sin(time_f * 1.5) * 0.5 + 0.5;
    float dispersionBase = mix(0.02, 0.25, pulse);
    float dispersion = dispersionBase * dist;
    int samples = 28;
    vec3 accumulatedColor = vec3(0.0);
    for (int i = 0; i < samples; i++) {
        float w = float(i) / float(samples - 1);
        float shift = (w - 0.5) * dispersion;
        vec2 sampleUV = uv + (refractDir * shift);
        vec3 tex = texture(samp, clamp(sampleUV, 0.0, 1.0)).rgb;
        accumulatedColor += tex * spectrum(w);
    }
    vec3 finalRGB = accumulatedColor / (float(samples) * 0.45);
    finalRGB = pow(finalRGB, vec3(0.9));
    vec2 lightPos = vec2(0.2 + (pulse * 0.1), 0.8);
    float lightDist = length(uv - lightPos);
    float reflection = pow(max(1.0 - lightDist, 0.0), 15.0) * (0.4 + pulse * 0.4);
    float streak = pow(max(1.0 - abs(uv.x + uv.y - 1.0), 0.0), 50.0) * (0.2 + pulse * 0.3);
}

```

```

    finalRGB += (reflection + streak);
    finalRGB *= 1.1 - (dist * 0.2);
    color = vec4(finalRGB, 1.0);
}

```

Shader #784 - prism quad by time fast

File: prism_quad_by_time_fast.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright. Spectral helper for the rainbow smear Change '1.5' to adjust the speed of the oscillation Remap pulse to dispersion range (0.02 is subtle, 0.25 is extreme)

Technique: Linear mix/lerp blending - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 spectrum(float w) {
    vec3 c;
    if (w < 0.2) c = mix(vec3(1.0, 0.0, 1.0), vec3(0.0, 0.0, 1.0), w * 5.0);
    else if (w < 0.4) c = mix(vec3(0.0, 0.0, 1.0), vec3(0.0, 1.0, 0.0), (w - 0.2) * 5.0);
    else if (w < 0.6) c = mix(vec3(0.0, 1.0, 0.0), vec3(1.0, 1.0, 0.0), (w - 0.4) * 5.0);
    else if (w < 0.8) c = mix(vec3(1.0, 1.0, 0.0), vec3(1.0, 0.0, 0.0), (w - 0.6) * 5.0);
    else c = vec3(1.0, 0.0, 0.0);
    return pow(c, vec3(0.8));
}
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = uv - center;
    float dist = length(dir);
    vec2 refractDir = normalize(dir);
    float pulse = tan(time_f * 1.1) * 0.5 + 0.5;
    float dispersionBase = mix(0.02, 0.25, pulse);
    float dispersion = dispersionBase * dist;
    int samples = 28;
    vec3 accumulatedColor = vec3(0.0);
    for (int i = 0; i < samples; i++) {
        float w = float(i) / float(samples - 1);
        float shift = (w - 0.5) * dispersion;
        vec2 sampleUV = uv + (refractDir * shift);
        vec3 tex = texture(samp, clamp(sampleUV, 0.0, 1.0)).rgb;
        accumulatedColor += tex * spectrum(w);
    }
    vec3 finalRGB = accumulatedColor / (float(samples) * 0.45);
    finalRGB = pow(finalRGB, vec3(0.9));
    vec2 lightPos = vec2(0.2 + (pulse * 0.1), 0.8);
    float lightDist = length(uv - lightPos);
    float reflection = pow(max(1.0 - lightDist, 0.0), 15.0) * (0.4 + pulse * 0.4);
    float streak = pow(max(1.0 - abs(uv.x + uv.y - 1.0), 0.0), 50.0) * (0.2 + pulse * 0.3);
    finalRGB += (reflection + streak);
    finalRGB *= 1.1 - (dist * 0.2);
    color = vec4(finalRGB, 1.0);
}

```

Shader #785 - prism quad strong

File: prism_quad_strong.glsl

Applies an iterative, time-animated effect that evolves the image over time. Enhanced spectral helper for more "neon" brilliance Boost saturation Increased for a wider rainbow

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 spectrum(float w) {
    vec3 c;
    if (w < 0.2) c = mix(vec3(1.0, 0.0, 1.0), vec3(0.0, 0.0, 1.0), w * 5.0);
    else if (w < 0.4) c = mix(vec3(0.0, 0.0, 1.0), vec3(0.0, 1.0, 0.0), (w - 0.2) * 5.0);
    else if (w < 0.6) c = mix(vec3(0.0, 1.0, 0.0), vec3(1.0, 1.0, 0.0), (w - 0.4) * 5.0);
    else if (w < 0.8) c = mix(vec3(1.0, 1.0, 0.0), vec3(1.0, 0.0, 0.0), (w - 0.6) * 5.0);
    else c = vec3(1.0, 0.0, 0.0);
    return pow(c, vec3(0.8));
}

void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = uv - center;
    float dist = length(dir);
    vec2 refractDir = normalize(dir);
    float dispersionBase = 0.18;
    float dispersion = dispersionBase * dist;
    int samples = 30;
    vec3 accumulatedColor = vec3(0.0);
    float maxBrightness = 0.0;
    for (int i = 0; i < samples; i++) {
        float w = float(i) / float(samples - 1);
        float shift = (w - 0.5) * dispersion;
        vec2 sampleUV = uv + (refractDir * shift);
        vec3 tex = texture(samp, clamp(sampleUV, 0.0, 1.0)).rgb;
        maxBrightness = max(maxBrightness, length(tex));
        accumulatedColor += tex * spectrum(w);
    }
    vec3 finalRGB = accumulatedColor / (float(samples) * 0.45);
    finalRGB = pow(finalRGB, vec3(0.9));
    vec2 lightPos = vec2(0.2, 0.8);
    float lightDist = length(uv - lightPos);
    float reflection = pow(max(1.0 - lightDist, 0.0), 12.0) * 0.6;
    float streak = pow(max(1.0 - abs(uv.x + uv.y - 1.0), 0.0), 40.0) * 0.3;
    finalRGB += (reflection + streak);
    finalRGB *= 1.1 - (dist * 0.2);
    color = vec4(finalRGB, 1.0);
}
```

Shader #786 - psych

File: psych.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
void main() {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float time = time_f * 0.5;
    float sine1 = sin(uv.x * 10.0 + time) * cos(uv.y * 10.0 + time);
    float sine2 = sin(uv.y * 15.0 - time) * cos(uv.x * 15.0 - time);
    float sine3 = sin(sqrt(uv.x * uv.x + uv.y * uv.y) * 20.0 + time);
    float pattern = sine1 + sine2 + sine3;
    float colorIntensity = pattern * 0.5 + 0.5;
    vec3 psychedelicColor = vec3(
        sin(colorIntensity * 6.28318 + 0.0) * 0.5 + 0.5,
        sin(colorIntensity * 6.28318 + 2.09439) * 0.5 + 0.5,
        sin(colorIntensity * 6.28318 + 4.18879) * 0.5 + 0.5
    );
    color = vec4(psychedelicColor, 1.0);
    color = mix(color, texture(samp, tc), 0.5);
}
```

Shader #793 - pull

File: pull.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 centeredCoord = (tc * 2.0) - vec2(1.0, 1.0);
    float stretchFactor = 1.0 + (1.0 - abs(centeredCoord.y)) * 0.5;
    centeredCoord.x *= sin(stretchFactor * time_f);
    vec2 stretchedCoord = (centeredCoord + vec2(1.0, 1.0)) / 2.0;
    color = texture(samp, stretchedCoord);
}
```

Shader #794 - pull out

File: pull_out.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
```

```

out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 centeredCoord = (tc * 2.0) - vec2(1.0, 1.0);
    float stretchFactor = 1.0 + (1.0 - abs(centeredCoord.y)) * 0.5;
    centeredCoord.x *= sin(stretchFactor * sin(time_f));
    vec2 stretchedCoord = (centeredCoord + vec2(1.0, 1.0)) / 2.0;
    color = texture(samp, stretchedCoord);
}

```

Shader #796 - purple fade

File: purple_fade.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 l = vec4(0.8, 0.6, 1.0, 1.0);
    color = color * l * alpha;
}

```

Shader #798 - purple strobe

File: purple_strobe.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;

```

```

uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
in vec2 iResolution;
uniform vec2 iResolution;
uniform float restore_black;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}
void main(void)
{
    color = texture(samp, tc);
    vec2 color_value1 = gl_FragCoord.xy / iResolution.xy;
    vec4 col = vec4(0.2, 0.4, 0.8, 1.0);
    float rnd = random(color_value1);
    vec4 value = vec4(gl_FragCoord.x/255, gl_FragCoord.y/255, gl_FragCoord.x/255/rnd,1.0);
    col += value * alpha_r;
    color = color * col;
    color = color / 3.0;
}

```

Shader #800 - radial

File: radial.glsl

Applies an iterative, time-animated effect that evolves the image over time.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc - 0.5;
    float blurAmount = length(uv) * 0.2;
    vec4 sum = vec4(0.0);
    for (int i = -4; i <= 4; i++) {
        for (int j = -4; j <= 4; j++) {
            vec2 offset = vec2(i, j) * blurAmount / 10.0;
            sum += texture(samp, tc + offset);
        }
    }
    color = sum / 81.0;
}

```

Shader #853 - react12

File: react12.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 uv = tc - center;
    float dist = length(uv);
    float ripple = sin(10.0 * dist - time_f * 6.28318) * 0.1;
    uv += uv * ripple;
    uv += center;
    color = texture(samp, uv);
}
```

Shader #854 - react13

File: react13.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 uv = tc - center;
    float dist = length(uv);
    float angle = time_f * 2.0 + dist * 5.0;
    float s = sin(angle);
    float c = cos(angle);
    uv = vec2(
        uv.x * c - uv.y * s,
        uv.x * s + uv.y * c
    );
    uv += center;
    color = texture(samp, uv);
}
```

Shader #855 - react14

File: react14.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: UV mirroring/folding for symmetry - Linear mix/lerp blending - Driven by: time, resolution


```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float t = pingPong(time_f, 15.0);
    float wave = sin(uv.x * 10.0 + t * 2.0) * 0.1;
    float expand = 0.5 + 0.5 * sin(t * 2.0);
    vec2 spiral_uv = uv * expand + vec2(cos(t), sin(t)) * 0.2;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    vec3 shifted_color = sin(blended_color * t * 2.0);
    vec4 color_result = vec4(shifted_color, original_color.a);
    vec4 color2 = texture(samp, tc / 2);
    vec2 tc1 = tc;
    vec2 tc2 = tc;
    tc1[0] = 1.0 - tc1[0];
    tc2[1] = 1.0 - tc2[1];
    vec4 color3 = texture(samp, tc1 / 4);
    vec4 color4 = texture(samp, tc2 / 4);
    color_result = (color_result * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4);
    color = color_result;
}

```

Shader #856 - react15

File: react15.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;

```

```

    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

void main() {
    vec2 uv = tc;
    float t = time_f * 0.1;
    vec3 rainbowColor = rainbow(uv.x + t);
    vec4 texColor = texture(samp, uv);
    color = mix(texColor, vec4(rainbowColor, 1.0), 0.5);
}

```

Shader #861 - react2

File: react2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 uv = tc - 0.5;
    float amplitude = sin(time_f) * 0.5 + 0.5;
    float angle = time_f + amplitude * 10.0;
    mat2 rotation = mat2(cos(angle), -sin(angle), sin(angle), cos(angle));
    vec2 rotatedUV = rotation * uv;
    color = texture(samp, rotatedUV + 0.5);
}

```

Shader #862 - react3

File: react3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 uv = tc - 0.5;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x) + radius * 10.0 + time_f * (0.5 + sin(time_f) * 0.5);
    vec2 spiralUV = vec2(cos(angle), sin(angle)) * radius;
    color = texture(samp, spiralUV + 0.5);
}

```

Shader #863 - react4

File: react4.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 uv = tc - 0.5;
    float drumEffect = sin(time_f * 10.0) * 0.1;
    vec2 drumUV = uv * (1.0 + drumEffect);
    color = texture(samp, drumUV + 0.5);
}
```

Shader #866 - react7

File: react7.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 redOffset = vec2(sin(time_f * 5.0), cos(time_f * 5.0)) * 0.02;
    vec2 greenOffset = vec2(cos(time_f * 7.0), sin(time_f * 7.0)) * 0.02;
    vec2 blueOffset = vec2(sin(time_f * 3.0), cos(time_f * 3.0)) * 0.02;
    vec4 redChannel = texture(samp, tc + redOffset);
    vec4 greenChannel = texture(samp, tc + greenOffset);
    vec4 blueChannel = texture(samp, tc + blueOffset);
    color = vec4(redChannel.r, greenChannel.g, blueChannel.b, 1.0);
}
```

Shader #868 - react9

File: react9.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
vec3 getRainbowColor(float t) {
    float r = 0.5 + 0.5 * cos(6.28318 * (t + 0.0));
    float g = 0.5 + 0.5 * cos(6.28318 * (t + 0.333));
```

```

    float b = 0.5 + 0.5 * cos(6.28318 * (t + 0.666));
    return vec3(r, g, b);
}

void main(void) {
    vec2 redOffset = vec2(sin(time_f * 5.0), cos(time_f * 5.0)) * 0.02;
    vec2 greenOffset = vec2(cos(time_f * 7.0), sin(time_f * 7.0)) * 0.02;
    vec2 blueOffset = vec2(sin(time_f * 3.0), cos(time_f * 3.0)) * 0.02;
    vec4 redChannel = texture(samp, tc + redOffset);
    vec4 greenChannel = texture(samp, tc + greenOffset);
    vec4 blueChannel = texture(samp, tc + blueOffset);
    vec4 originalColor = vec4(redChannel.r, greenChannel.g, blueChannel.b, 1.0);
    vec3 rainbowColor = getRainbowColor(time_f * 0.1);
    vec3 mixedColor = mix(originalColor.rgb, rainbowColor, 0.5);
    float beat = abs(sin(time_f * 10.0));
    color = vec4(mix(originalColor.rgb, mixedColor, beat), 1.0);
}

```

Shader #869 - recep 2

File: recep-2.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float noise(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}

void main(void) {
    vec2 uv = tc;
    float n = noise(uv * time_f);
    float offset = sin(uv.y * 10.0 + time_f) * 0.1;
    uv.x += offset * n;
    vec4 texColor = texture(samp, uv);
    float stripe = step(0.5, fract(uv.y * 80.0 + time_f));
    texColor.rgb *= stripe;
    texColor.rgb += n * 0.2;
    color = 1.0 - texColor;
    color.a = 1.0;
}

```

Shader #870 - recep

File: recep.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float noise(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void) {
    vec2 uv = tc;
    float n = noise(uv * time_f);
    float offset = sin(uv.y * 10.0 + time_f) * 0.1;
    uv.x += offset * n;
    vec4 texColor = texture(samp, uv);
    float stripe = step(0.5, fract(uv.y * 40.0 + time_f));
    texColor.rgb *= stripe;
    texColor.rgb += n * 0.2;
    color = texColor;
}

```

Shader #871 - rect1

File: rect1.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float loop_dur = 15.0;
    float current_t = mod(time_f, loop_dur);
    vec2 uv = (tc - 0.5) * (iResolution.x / iResolution.y, 1.0);
    vec2 cord = tan(uv * current_t);
    color = texture(samp, cord);
}

```

Shader #872 - rect spiral

File: rect_spiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float loop_dur = 20.0;
    float current_t = mod(time_f, loop_dur);

```

```

    vec2 uv = (tc - 0.5) * (iResolution.x / iResolution.y, 1.0);
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    float spiralSpeed = 1.0;
    angle += current_t * spiralSpeed;
    radius *= current_t;    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * radius;
    vec2 transformedCoord = sin(spiralCoord * radius/2);
    vec2 finalCoord = transformedCoord * 0.5 + 0.5;
    color = texture(samp, finalCoord);
}

```

Shader #873 - red

File: red.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] /= 2;
    color[1] /= 2;
    color[2] /= 2;
    color[0] = cos(color[0]*alpha_b);
    color[1] = sin(color[1]*alpha_g);
    color[2] = (color[2]*alpha_r);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #874 - red strobe

File: red_strobe.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[2] = color[2]*(random_value[0]/255);
    color[1] = color[1]*(timeval * 0.05812);
    color[0] = color[0]*(alpha_r * timeval * (random_value[1]/255));
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #876 - reverse

File: reverse.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```
#version 330
in vec2 tc;
```

```

out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    vec2 vpos = tc;
    vpos[0] = 1.0-tc[0];
    color = texture(samp, vpos);
}

```

Shader #877 - reverse alpha

File: reverse_alpha.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    vec2 vpos = tc;
    vpos[0] = 1.0-tc[0];
    color = texture(samp, tc);
    vec4 color2 = texture(samp, vpos);
    color = (0.5 * color/alpha) + (0.5 * color2);
    color *= 2;
}

```

Shader #878 - reverse vert

File: reverse_vert.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    vec2 vpos = tc;
    vpos[1] = 1.0-tc[1];
    color = texture(samp, vpos);
}
```

Shader #880 - reverse xy

File: reverse_xy.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
```

```

uniform float time_f;
uniform float restore_black;
void main(void)
{
    vec2 vpos = 1.0-tc;
    color = texture(samp, vpos);
}

```

Shader #911 - sbrv

File: sbrv.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc;
    float blockSize = 0.1 + 0.05 * sin(time_f * 2.0);
    float blockY = floor(uv.y / blockSize) * blockSize;
    vec2 resizedUV = vec2(uv.x, blockY + mod(uv.y, blockSize) * (0.5 + 0.5 * sin(time_f + blockY * 10.0)
↪ ));
    vec4 texColor = texture(samp, resizedUV);
    color = texColor;
}

```

Shader #913 - scramble 2

File: scramble-2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    float time_t = pingPong(time_f, 10);
    float scrambleAmount = sin(time_f * 10.0) * 0.5 + 0.5;
    scrambleAmount = cos(scrambleAmount * time_t);
    uv.x += sin(uv.y * 50.0 + time_f * 10.0) * scrambleAmount * 0.05;
    uv.y += cos(uv.x * 50.0 + time_f * 10.0) * scrambleAmount * 0.05;
    color = texture(samp, tan(uv * time_t));
}

```

```
}
```

Shader #914 - scramble 3

File: scramble-3.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float scrambleAmountX = sin(time_f * 20.0) * 0.5 + 0.5;
    float scrambleAmountY = cos(time_f * 20.0) * 0.5 + 0.5;
    uv.x += sin(uv.y * 100.0 + time_f * 20.0) * scrambleAmountX * 0.1;
    uv.y += cos(uv.x * 100.0 + time_f * 20.0) * scrambleAmountY * 0.1;
    vec2 noise = vec2(
        sin(time_f + uv.y * 50.0) * 0.05,
        cos(time_f + uv.x * 50.0) * 0.05
    );
    uv += noise;
    color = texture(samp, uv);
}
```

Shader #915 - scramble

File: scramble.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float scrambleAmount = sin(time_f * 10.0) * 0.5 + 0.5;
    uv.x += sin(uv.y * 50.0 + time_f * 10.0) * scrambleAmount * 0.05;
    uv.y += cos(uv.x * 50.0 + time_f * 10.0) * scrambleAmount * 0.05;
    color = texture(samp, uv);
}
```

Shader #923 - shake

File: shake.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
void main(void) {
    vec2 pos = tc;
    pos = pos + value_alpha_r;
    color = texture(samp, pos);
}
```

Shader #924 - sheet

File: sheet.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec2 start = gl_FragCoord.xy / 64;
    vec2 val = fract(start);
    for(int i = 0; i < 3; ++i) {
        color[i] += color[i]*(val[0]+val[1]);
    }
}
```

Shader #925 - shift

File: shift.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: 2D rotation matrix - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 colorRotate(vec4 color) {
    ivec4 col = ivec4(color * 255);
    for(int i = 0; i < 3; ++i) {
        col[i] = col[i]%0xFF;
    }
    return vec4(col/255.0);
}

void main() {
    vec4 col = blur(samp, tc, iResolution);
    float value = pingPong(time_f, 10.0) + 2.0;
    color = colorRotate(col * value);
}
```

Shader #926 - shift dark

File: shift_dark.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: 2D rotation matrix - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 colorRotate(vec4 color) {
    ivec4 col = ivec4(color * 255);
    for(int i = 0; i < 3; ++i) {
        col[i] = col[i]%0xFF;
    }
    return vec4(col/255.0);
}

void main() {
    vec4 col = blur(samp, tc, iResolution);
    float value = pingPong(time_f, 10.0) + 2.0;
    float time_t = pingPong(time_f, 7.0) + 2.0;
    color = sin(colorRotate(col * value) * time_t);
    color.a = 1.0;
}
```

Shader #927 - shrink texture

File: shrink_texture.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    vec2 pos = uv - 0.5;
    float len = length(pos);
    vec2 ripple = pos * (0.6 / len) * sin(len * 2.0 - time_f * 1.0);
    uv += ripple * sin(0.3 * time_f);
    color = texture(samp, uv);
}
```

Shader #928 - shrink texture 4

File: shrink_texture_4.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    vec2 pos = uv - 0.5;
    float len = length(pos);
    vec2 ripple = pos * (0.6 / len) * sin(len * 2.0 - time_f * 1.0);
    uv += ripple * 0.4;
    color = texture(samp, uv);
}
```

Shader #929 - sin

File: sin.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
```

```

uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    for(int i = 0; i < 3; ++i) {
        color[i] += sin(color[i] * timeval);
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #930 - sin delic

File: sin_delic.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    uv.x += sin(uv.y * 10.0 + time_f) * 0.1;
    uv.y += cos(uv.x * 10.0 + time_f) * 0.1;
    float modFactor = pingPong(time_f * 0.5, 1.0);
    vec4 texColor = texture(samp, uv);
    vec3 psychedelicColor = vec3(
        texColor.r * sin(time_f + uv.y * 10.0),
        texColor.g * cos(time_f + uv.x * 10.0),
        texColor.b * sin(time_f + uv.y * 5.0)
    );
    psychedelicColor = mix(texColor.rgb, psychedelicColor, modFactor);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = vec4(sin(psychedelicColor * time_t), texColor.a);
}

```


Shader #933 - sine grad

File: sine_grad.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 rainbow(float t) {
    float r = sin(6.28318 * t + 0.0);
    float g = sin(6.28318 * t + 2.09439);
    float b = sin(6.28318 * t + 4.18878);
    return vec3(r, g, b) * 0.5 + 0.5;
}
void main(void) {
    float scale = 5.0;
    float speed = 0.2;
    float gradient_pos = 0.5 + 0.5 * sin((tc.x * scale + tc.y * scale) + time_f * speed);
    vec3 color_gradient = rainbow(gradient_pos);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec4 ctx = texture(samp, tc);
    color = vec4(sin(ctx.rgb * time_t), ctx.a);
    color = mix(color, vec4(color_gradient, 1.0), 0.5);
}
```

Shader #934 - sine grad2

File: sine_grad2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 rainbow(float t) {
    float r = sin(6.28318 * t + 0.0);
    float g = sin(6.28318 * t + 2.09439);
    float b = sin(6.28318 * t + 4.18878);
    return vec3(r, g, b) * 0.5 + 0.5;
}
void main(void) {
    float swirl_strength = 0.1;
```

```

float swirl_speed = 1.0;
float angle = swirl_speed * time_f + length(tc - 0.5);
vec2 swirl_offset = vec2(cos(angle), sin(angle)) * swirl_strength;
vec2 swirl_tc = tc + swirl_offset;
swirl_tc = clamp(swirl_tc, vec2(0.0), vec2(1.0));
float scale = 5.0;
float speed = 0.2;
float gradient_pos = 0.5 + 0.5 * sin((swirl_tc.x * scale + swirl_tc.y * scale) + time_f * speed);
vec3 color_gradient = rainbow(gradient_pos);
vec4 ctx = texture(samp, swirl_tc);
color = vec4(sin(ctx.rgb * time_f), ctx.a);
color = mix(color, vec4(color_gradient, 1.0), 0.5);
}

```

Shader #935 - sine grad3

File: sine_grad3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 rainbow(float t) {
    float r = sin(6.28318 * t + 0.0);
    float g = sin(6.28318 * t + 2.09439);
    float b = sin(6.28318 * t + 4.18878);
    return vec3(r, g, b) * 0.5 + 0.5;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x) + time_f * 2.0;
    float radius = dist + sin(time_f * 2.0 + dist * 10.0) * 0.05;
    vec2 spiral_tc = center + vec2(cos(angle), sin(angle)) * radius;
    spiral_tc = clamp(spiral_tc, vec2(0.0), vec2(1.0));
    float gradient_pos = mod(dist * 3.0 - time_f * 0.5, 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec4 ctx = texture(samp, spiral_tc);
    color = mix(ctx, vec4(color_gradient, 1.0), 0.5);
}

```

Shader #936 - sine grad4

File: sine_grad4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 rainbow(float t) {
    float r = sin(6.28318 * t + 0.0);
    float g = sin(6.28318 * t + 2.09439);
    float b = sin(6.28318 * t + 4.18878);
    return vec3(r, g, b) * 0.5 + 0.5;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float radius = dist + sin(time_f * 2.0 + dist * 10.0) * 0.05;
    vec2 spiral_offset = vec2(cos(angle), sin(angle)) * radius;
    vec2 spiral_tc = center + spiral_offset;
    spiral_tc = clamp(spiral_tc, vec2(0.0), vec2(1.0));
    float gradient_pos = mod(dist * 3.0 - time_f * 0.5, 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec4 ctx = texture(samp, tc);
    color = mix(ctx, vec4(color_gradient, 1.0), 0.5);
}
```

Shader #937 - sine_grad5

File: sine_grad5.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 rainbow(float t) {
    float r = sin(6.28318 * t + 0.0);
    float g = sin(6.28318 * t + 2.09439);
```

```

    float b = sin(6.28318 * t + 4.18878);
    return vec3(r, g, b) * 0.5 + 0.5;
}

void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float radius = dist + sin(time_f * 2.0 + dist * 10.0) * 0.05;
    vec2 spiral_offset = vec2(cos(angle), sin(angle)) * radius;
    vec2 spiral_tc = center + spiral_offset;
    spiral_tc = clamp(spiral_tc, vec2(0.0), vec2(1.0));
    float gradient_pos = mod(dist * 3.0 - time_f * 0.5, 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec4 ctx = texture(samp, tc);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = mix(sin(ctx * time_t), vec4(color_gradient, 1.0), 0.5);
    color.a = 1.0;
}

```

Shader #942 - sine time

File: sine_time.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

void main(void) {
    vec4 color_solid = vec4(0.3, 0.1, 0.8, 1.0);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec4 ctx = texture(samp, tc);
    color = vec4(sin(ctx.rgb * time_t), ctx.a);
    color = mix(color, color_solid, 0.5);
}

```

Shader #943 - sing

File: sing.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = (tc * 2.0) - 1.0;
    float distance = length(uv);
    float phase = sin(time_f * 3.14159265);
    float scale = smoothstep(0.0, 1.0, phase);
    vec2 scaled_uv = uv / scale;
    scaled_uv = (scaled_uv + 1.0) * 0.5;
    scaled_uv = clamp(scaled_uv, 0.0, 1.0);
    color = texture(samp, scaled_uv);
}

```

Shader #944 - single glittch

File: single-glittch.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Texture coordinates passed from vertex shader Single texture unit for all mappings and manipulations Screen resolution, useful for dynamic effects

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float alpha_r;
const float PI = 3.1415926535897932384626433832795;
float noise(vec2 p) {
    vec2 ip = floor(p);
    vec2 fp = fract(p);
    float a = dot(sin(ip), vec2(1.0, 17.0));
    float b = dot(cos(ip), vec2(1.0, 17.0));
    float n_x = mix(mix(a, b, fp.x), mix(sin(a), cos(b), fp.x), fp.y);
    return n_x;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main() {
    vec2 uv = tc;
    int OCTAVES = int(alpha_r * pingPong(time_f*PI, 10.0));
    float detailNoise = 0.0;
    for (int i = 0; i < OCTAVES; ++i) {
        float freq = pow(2.0, float(i));
        float amp = pow(0.5, float(OCTAVES - i));
        detailNoise += noise(uv * freq) * freq * sin(amp * time_f);
    }
    uv += vec2(detailNoise) * 0.1;
    color = texture(samp, uv);
}

```

Shader #946 - skinny

File: skinny.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float factor = 2.0;
    dir.x *= factor;
    vec2 new_tc = center + dir;
    color = texture(samp, new_tc);
}
```

Shader #947 - slither

File: slither.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float frequency = 10.0;
    float amplitude = 0.02;
    float speed = 2.0;
    float rowHeight = 1.0 / 4.0;
    vec2 snake;
    if (tc.y < rowHeight) {
        snake = tc + vec2(amplitude * sin(tc.y * frequency + time_f * speed), amplitude * sin(tc.x *
        ↪ frequency + time_f * speed));
    } else if (tc.y < 2.0 * rowHeight) {
        snake = tc + vec2(amplitude * sin((tc.y - rowHeight) * frequency + time_f * speed), amplitude *
        ↪ sin(tc.x * frequency + time_f * speed));
    } else if (tc.y < 3.0 * rowHeight) {
        snake = tc + vec2(amplitude * sin((tc.y - 2.0 * rowHeight) * frequency + time_f * speed),
        ↪ amplitude * sin(tc.x * frequency + time_f * speed));
    } else {
        snake = tc + vec2(amplitude * sin((tc.y - 3.0 * rowHeight) * frequency + time_f * speed),
        ↪ amplitude * sin(tc.x * frequency + time_f * speed));
    }
    vec4 texColor = texture(samp, snake);
    color = texColor;
}
```

Shader #948 - slither vert

File: slither_vert.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float frequency = 10.0;
    float amplitude = 0.02;
    float speed = 2.0;
    float columnWidth = 1.0 / 4.0;
    vec2 snake;
    if (tc.x < columnWidth) {
        snake = tc + vec2(amplitude * sin(tc.y * frequency + time_f * speed), amplitude * sin(tc.x *
        ↪ frequency + time_f * speed));
    } else if (tc.x < 2.0 * columnWidth) {
        snake = tc + vec2(amplitude * sin(tc.y * frequency + time_f * speed), amplitude * sin((tc.x -
        ↪ columnWidth) * frequency + time_f * speed));
    } else if (tc.x < 3.0 * columnWidth) {
        snake = tc + vec2(amplitude * sin(tc.y * frequency + time_f * speed), amplitude * sin((tc.x -
        ↪ 2.0 * columnWidth) * frequency + time_f * speed));
    } else {
        snake = tc + vec2(amplitude * sin(tc.y * frequency + time_f * speed), amplitude * sin((tc.x -
        ↪ 3.0 * columnWidth) * frequency + time_f * speed));
    }
    vec4 texColor = texture(samp, snake);
    color = texColor;
}
```

Shader #953 - snake

File: snake.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc * iResolution;
    float wave = sin(uv.x * 0.05 + time_f * 2.0) * 0.05;
    vec2 shiftedUV = vec2(uv.x, uv.y + wave * iResolution.y);
    vec4 texColor = texture(samp, shiftedUV / iResolution);
    color = texColor;
}
```

Shader #954 - snake dir

File: snake_dir.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
vec4 snake1() {
    vec2 uv = tc * iResolution;
    float wave = sin(uv.x * 0.05 + time_f * 2.0) * 0.05;
    vec2 shiftedUV = vec2(uv.x, uv.y + wave * iResolution.y);
    vec4 texColor = texture(samp, shiftedUV / iResolution);
    return texColor;
}
vec4 snake2() {
    vec2 uv = tc * iResolution;
    float wave = sin(uv.y * 0.05 + time_f * 2.0) * 0.05;
    vec2 shiftedUV = vec2(uv.x + wave * iResolution.x, uv.y);
    vec4 texColor = texture(samp, shiftedUV / iResolution);
    return texColor;
}
void main(void) {
    float time_t = mod(time_f, 10);
    color = sin(mix(snake1(), snake2(), 0.5) * (alpha * time_t));
    color.a = 1.0;
}
```

Shader #955 - snake updown

File: snake_updown.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * iResolution;
    float wave = sin(uv.y * 0.05 + time_f * 2.0) * 0.05;
    vec2 shiftedUV = vec2(uv.x + wave * iResolution.x, uv.y);
    vec4 texColor = texture(samp, shiftedUV / iResolution);
    color = texColor;
}
```

Shader #956 - snes

File: snes.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic.

Technique: Indexed color palette look-up table - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float pixelSize = 0.01;
    uv = floor(uv / pixelSize) * pixelSize;
    vec4 texColor = texture(samp, uv);
    vec3 paletteColor = floor(texColor.rgb * 31.0) / 31.0;
    color = vec4(paletteColor, texColor.a);
}
```

Shader #957 - source strobe

File: source_strobe.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    ivec4 r;
    for(int i = 0; i < 3; ++i) {
        r[i] = int(random_value[i]);
        source[i] = ((source[i]/2)*r[i])/12;
    }
}
```

```

    }
    color[0] = tc[0]*color[0];
    color[1] = tc[1]*color[1];
    color[2] = tc[1]*color[2];
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #959 - spiral aura

File: spiral-aura.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc * 2.0 - 1.0;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float spiral = angle + radius * 5.0 - time_f;
    float reflection = mod(spiral, 3.14159 * 2.0);
    vec2 spiralUV = vec2(cos(spiral), sin(spiral)) * radius + 0.5;
    vec2 reflectUV = vec2(cos(reflection), sin(reflection)) * radius + 0.5;
    vec4 gradientColor = vec4(0.5 + 0.5 * cos(time_f + radius * 10.0), 0.5 + 0.5 * sin(time_f + radius *
    ↪ 10.0), 0.5 + 0.5 * cos(time_f - radius * 10.0), 1.0);
    vec4 textureColor = texture(samp, reflectUV);
    color = mix(gradientColor, textureColor, 0.5);
}

```

Shader #969 - spiral square

File: spiral_square.glsl

Applies geometric transformations including rotation and scaling to distort the image.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;

```

```

vec2 rotate(vec2 v, float a) {
    float s = sin(a);
    float c = cos(a);
    mat2 m = mat2(c, -s, s, c);
    return m * v;
}

void main() {
    float time_t = mod(time_f, 15.0);
    vec2 uv = tc;
    uv = uv * 2.0 - 1.0;
    uv = rotate(uv, time_t);
    uv += 2.0;
    uv = fract(uv * time_t);
    color = texture(samp, uv);
}

```

Shader #972 - splash x

File: splash-x.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void)
{
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    float distance = length(uv - center);
    float ripple = sin(10.0 * distance - time_f * 3.0);
    ripple *= sin(time_f) / distance;
    uv += (uv - center) * ripple;
    color = texture(samp, uv);
    color = mix(color, vec4(0.0, 0.5, 1.0, 1.0), 0.2);
}

```

Shader #973 - splash y

File: splash-y.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void)
{

```

```

    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    float distance = length(uv - center);
    float ripple = sin(10.0 * distance - time_f * 3.0);
    ripple *= tan(time_f) / distance;
    uv += (uv - center) * ripple;
    color = texture(samp, uv);
    color = mix(color, vec4(0.0, 0.5, 1.0, 1.0), 0.2);
}

```

Shader #974 - splash

File: splash.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void)
{
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    float distance = length(uv - center);
    float ripple = sin(10.0 * distance - time_f * 3.0);
    ripple *= 0.03 / distance;
    uv += (uv - center) * ripple;
    color = texture(samp, uv);
    color = mix(color, vec4(0.0, 0.5, 1.0, 1.0), 0.2);
}

```

Shader #978 - star

File: star.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform vec2 iResolution;
in vec2 iResolution_;
in float timeval;
uniform sampler2D samp;
uniform float alpha;
void main() {
    float time = timeval;
    vec2 resolution = iResolution_;
    vec2 uv = gl_FragCoord.xy / resolution.xy;
}

```

```

    uv = uv * 2.0 - 1.0;
    uv.x *= resolution.x / resolution.y;
    float angleRotation = time;
    float cosA = cos(angleRotation);
    float sinA = sin(angleRotation);
    mat2 rotationMat = mat2(cosA, -sinA, sinA, cosA);
    uv = rotationMat * uv;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle = mod(angle, 2.0 * 3.14159 / 5.0);
    float sharpness = 0.5;
    bool inStar = radius < (cos(sharpness) / cos(angle - sharpness));
    if (inStar) {
        color = texture(samp, tc);
        color = color * (alpha * 0.5);
    } else {
        color = texture(samp, tc);
    }
}

```

Shader #979 - star5

File: star5.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = (uv - center) * 2.0;
    float angle = time_f;
    float cosA = cos(angle);
    float sinA = sin(angle);
    mat2 rotate = mat2(cosA, -sinA, sinA, cosA);
    pos = rotate * pos;
    float r = length(pos);
    float a = atan(pos.y, pos.x);
    float n = 5.0;
    float t = mod(a + 3.14159 / n, 2.0 * 3.14159 / n) * n / 2.0 / 3.14159;
    t = abs(t - 0.5);
    float starMask = smoothstep(0.02, 0.04, t + r * 0.5 - 0.3) * smoothstep(0.9, 0.7, r);
    vec4 texColor = texture(samp, tc);
    vec3 darkenedTexture = texColor.rgb * 0.5;
    vec3 finalColor = mix(darkenedTexture, texColor.rgb, starMask);
    float time_t = mod(time_f, 10);
    if(starMask < 0.5)
        color = texture(samp, sin(tc * cos(time_t * a)));
    else
        color = vec4(finalColor, 1.0);
}

```

Shader #980 - stardust

File: stardust.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation - Radial vignette falloff - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void) {
    vec2 uv = tc;
    vec4 texColor = texture(samp, uv);
    float starDensity = 0.01;
    float starSize = 0.03;
    vec2 gridPos = floor(uv * iResolution.xy * starDensity);
    vec2 starPos = fract(uv * iResolution.xy * starDensity);
    float starNoise = rand(gridPos);
    float starIntensity = smoothstep(starSize, starSize * 0.5, distance(starPos, vec2(starNoise, fract(
        ↪ sin(time_f + starNoise) * 0.5 + 0.5))));
    float twinkleSpeed = 10.0 + amp_high * 30.0;
    starIntensity *= 0.5 + 0.5 * sin(time_f * twinkleSpeed + starNoise * 100.0);
    vec4 starColor = vec4(vec3(starIntensity), starIntensity);
    color = mix(texColor, starColor, starColor.a);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}
```

Shader #981 - starX

File: starX.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. color = texture(samp, sin(tc * cos(time_t * a)));

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
```

```

in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = (uv - center) * 2.0;
    float angle = time_f;
    float cosA = cos(angle);
    float sinA = sin(angle);
    mat2 rotate = mat2(cosA, -sinA, sinA, cosA);
    pos = rotate * pos;
    float r = length(pos);
    float a = atan(pos.y, pos.x);
    float n = 5.0;
    float t = mod(a + 3.14159 / n, 2.0 * 3.14159 / n) * n / 2.0 / 3.14159;
    t = abs(t - 0.5);
    float time_t = mod(time_f, 10);
    color = texture(samp, sin(tc * t));
}

```

Shader #982 - starX2

File: starX2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = (uv - center) * 2.0;
    float angle = time_f;
    float cosA = cos(angle);
    float sinA = sin(angle);
    mat2 rotate = mat2(cosA, -sinA, sinA, cosA);
    pos = rotate * pos;
    float r = length(pos);
    float a = atan(pos.y, pos.x);
    float n = 5.0;
    float t = mod(a + 3.14159 / n, 2.0 * 3.14159 / n) * n / 2.0 / 3.14159;
    t = abs(t - 0.5);
    float time_t = mod(time_f, 10);
    color = texture(samp, sin(tc * cos(time_t * a)));
}

```

Shader #983 - starX3

File: starX3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. `vec2(tc[0] * time_t * r, tc[1] * time_t * t)); color = texture(samp, sin(tc * t));`

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = (uv - center) * 2.0;
    float angle = time_f;
    float cosA = cos(angle);
    float sinA = sin(angle);
    mat2 rotate = mat2(cosA, -sinA, sinA, cosA);
    pos = rotate * pos;
    float r = length(pos);
    float a = atan(pos.y, pos.x);
    float n = 5.0;
    float t = mod(a + 3.14159 / n, 2.0 * 3.14159 / n) * n / 2.0 / 3.14159;
    t = abs(t - 0.5);
    float time_t = mod(time_f, 10);
    color = texture(samp, tc * r * time_t * t);
}
```

Shader #984 - starX4

File: starX4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 pos = (uv - center) * 2.0;
    float angle = time_f;
    float cosA = cos(angle);
    float sinA = sin(angle);
    mat2 rotate = mat2(cosA, -sinA, sinA, cosA);
    pos = rotate * pos;
    float r = length(pos);
    float a = atan(pos.y, pos.x);
```



```

float n = 5.0;
float t = mod(a + 3.14159 / n, 2.0 * 3.14159 / n) * n / 2.0 / 3.14159;
t = abs(t - 0.5);
float time_t = time_f;
color = texture(samp, sin(vec2(tc[0] * t * time_t, tc[1] * a * time_t)));
}

```

Shader #985 - strobe

File: strobe.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    color[0] = color[0] * color[0]/alpha_r;
    color[1] = color[1] * color[1]/alpha_g;
    color[2] = color[2] * color[2]/alpha_b;
}

```

Shader #987 - strobe in out

File: strobe_in_out.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;

```

```

uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    ivec3 values;
    float num_val = timeval * (alpha*alpha_r);
    if(num_val < 1) num_val = 1;
    for(int i = 0; i < 3; ++i) {
        values[i] = values[i]^int(num_val);
        color[i] = (float(values[i])/255)*alpha_g;
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #989 - strobe x1

File: strobe_x1.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)

```

```

{
    color = texture(samp, tc);
    float value = fract(timeval);
    if(value >= 0 && value <= 0.3) {
        color[0] *= value;
    } else if (value >= 0.3 && value <= 0.6) {
        color[1] *= value;
    } else if(value >= 0.6 && value <= 1.0) {
        color[2] *= value;
    }
    color = color * (alpha/2);
}

```

Shader #991 - surround

File: surround.glsl

Blends multiple texture samples together using interpolation for a composite effect.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;

```

```

    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec4 colorLeft = texture(samp, tc + vec2(-1.0 / iResolution.x, 0.0));
    vec4 colorRight = texture(samp, tc + vec2(1.0 / iResolution.x, 0.0));
    vec4 colorUp = texture(samp, tc + vec2(0.0, 1.0 / iResolution.y));
    vec4 colorDown = texture(samp, tc + vec2(0.0, -1.0 / iResolution.y));
    vec4 avgColor = (original_color + colorLeft + colorRight + colorUp + colorDown) / 5.0;
    vec3 blended_color = mix(avgColor.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_t), original_color.a);
}

```

Shader #999 - tan

File: tan.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    for(int i = 0; i < 3; ++i) {
        color[i] += tan(color[i] * timeval);
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        if(int_color[i] > 255)
            int_color[i] = int_color[i] % 255;
        color[i] = float(int_color[i]) / 255;
    }
}

```

Shader #1002 - tear

File: tear.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void){
    vec2 normCoord = tc;
    float tearLine = sin(time_f) * 0.5 + 0.5;
    float distFromTear = abs(normCoord.x - tearLine);
    float displacement = 0.202 * (1.0 - smoothstep(0.0, 0.1, distFromTear));
    normCoord.y += sin(displacement);
    color = texture(samp, normCoord);
}
```

Shader #1003 - tearing

File: tearing.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float tear1 = sin(tc.x * 10.0 + time_f * 5.0) * 0.05;
    float tear2 = sin(tc.x * 20.0 + time_f * 7.0) * 0.03;
    float tear3 = sin(tc.x * 30.0 + time_f * 9.0) * 0.02;
    float combined_tear = tear1 + tear2 + tear3;
    vec2 new_tc = vec2(tc.x + combined_tear, tc.y + combined_tear);
    color = texture(samp, new_tc);
}
```

Shader #1004 - tearing max

File: tearing_max.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
```

```

float tear1 = sin(tc.x * 10.0 + time_f * 5.0) * 0.05;
float tear2 = sin(tc.x * 20.0 + time_f * 7.0) * 0.03;
float tear3 = sin(tc.x * 30.0 + time_f * 9.0) * 0.02;
float tear4 = sin(tc.x * 40.0 + time_f * 11.0) * 0.015;
float tear5 = sin(tc.x * 50.0 + time_f * 13.0) * 0.01;
float tear6 = sin(tc.x * 60.0 + time_f * 15.0) * 0.007;
float tear7 = sin(tc.x * 70.0 + time_f * 17.0) * 0.005;
float tear8 = sin(tc.x * 80.0 + time_f * 19.0) * 0.003;
float tear9 = sin(tc.x * 90.0 + time_f * 21.0) * 0.002;
float combined_tear = tear1 + tear2 + tear3 + tear4 + tear5 + tear6 + tear7 + tear8 + tear9;
vec2 new_tc = vec2(tc.x + combined_tear, tc.y + combined_tear);
color = texture(samp, new_tc);
}

```

Shader #1005 - tearing single

File: tearing_single.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float tear = sin(tc.x * 10.0 + time_f * 5.0) * 0.05;
    vec2 new_tc = vec2(tc.x + tear, tc.y + tear);
    color = texture(samp, new_tc);
}

```

Shader #1006 - tearing spiral

File: tearing_spiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float tear1 = sin(tc.x * 10.0 + time_f * 5.0) * 0.05;
    float tear2 = sin(tc.x * 20.0 + time_f * 7.0) * 0.03;
    float tear3 = sin(tc.x * 30.0 + time_f * 9.0) * 0.02;
    float tear4 = sin(tc.x * 40.0 + time_f * 11.0) * 0.015;
    float tear5 = sin(tc.x * 50.0 + time_f * 13.0) * 0.01;
    float tear6 = sin(tc.x * 60.0 + time_f * 15.0) * 0.007;
    float tear7 = sin(tc.x * 70.0 + time_f * 17.0) * 0.005;
    float tear8 = sin(tc.x * 80.0 + time_f * 19.0) * 0.003;
    float tear9 = sin(tc.x * 90.0 + time_f * 21.0) * 0.002;
    float combined_tear = tear1 + tear2 + tear3 + tear4 + tear5 + tear6 + tear7 + tear8 + tear9;
}

```

```

    vec2 center = vec2(0.5, 0.5);
    vec2 vec = tc - center;
    float distance = length(vec);
    float angle = atan(vec.y, vec.x);
    angle += time_f * 2.0;
    vec2 new_vec = vec2(cos(angle), sin(angle)) * distance;
    vec2 new_tc = center + new_vec + vec2(combined_tear, combined_tear);
    color = texture(samp, new_tc);
}

```

Shader #1012 - time frac

File: time_frac.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
uniform vec4 inc_value;
void main(void)
{
    color = texture(samp, tc);
    color = (0.5 * color) + (0.5 * fract(inc_value/255 * time_f));
}

```

Shader #1016 - tornado

File: tornado.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;

```

```

uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main(void) {
    vec2 normCoord = tc;
    vec2 centeredCoord = normCoord * 2.0 - vec2(1.0, 1.0);
    centeredCoord.x *= iResolution.x / iResolution.y;
    float angle = atan(centeredCoord.y, centeredCoord.x);
    float dist = length(centeredCoord);
    float spiralFactor = 5.0 + amp_low * 5.0;
    angle += dist * spiralFactor + time_f;
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * dist;
    spiralCoord.x *= iResolution.y / iResolution.x;
    spiralCoord = (spiralCoord + vec2(1.0, 1.0)) / 2.0;
    color = texture(samp, spiralCoord);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #1017 - tremor1

File: tremor1.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 tremorEffect(vec2 uv, float frequency, float amplitude) {
    float tremorX = sin(uv.y * frequency + time_f * 5.0) * amplitude;
    float tremorY = cos(uv.x * frequency + time_f * 5.0) * amplitude;
    return uv + vec2(tremorX, tremorY);
}
void main() {
    vec2 uv = tc;
    float frequency = 20.0;
    float amplitude = 0.01 + 0.01 * sin(time_f * 3.0);
    uv = tremorEffect(uv, frequency, amplitude);
    vec4 texColor = texture(samp, fract(uv));
    color = texColor;
}

```


Shader #1018 - tremor2

File: tremor2.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
float pingPong(float t, float maxT) {
    return abs(mod(t, 2.0 * maxT) - maxT);
}
vec2 tremorEffect(vec2 uv, float frequency, float maxAmplitude, float interval) {
    float intervalFactor = pingPong(time_f, interval) / interval;
    float amplitude = maxAmplitude * intervalFactor;
    float tremorX = sin(uv.y * frequency + time_f * 5.0) * amplitude;
    float tremorY = cos(uv.x * frequency + time_f * 5.0) * amplitude;
    return uv + vec2(tremorX, tremorY);
}
void main() {
    vec2 uv = tc;
    float frequency = 20.0;
    float maxAmplitude = 0.02;
    float interval = 2.0;
    uv = tremorEffect(uv, frequency, maxAmplitude, interval);
    vec4 texColor = texture(samp, fract(uv));
    color = texColor;
}
```

Shader #1019 - tremor3

File: tremor3.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 tremorEffect(vec2 uv, float frequency, float maxAmplitude) {
    float amplitude = maxAmplitude * (0.5 + 0.5 * sin(time_f * 2.0));
    float tremorX = sin(uv.y * frequency + time_f * 10.0) * amplitude;
    float tremorY = cos(uv.x * frequency + time_f * 10.0) * amplitude;
    return uv + vec2(tremorX, tremorY);
}
void main() {
```

```

    vec2 uv = tc;
    float frequency = 30.0;
    float maxAmplitude = 0.02;
    uv = tremorEffect(uv, frequency, maxAmplitude);
    vec4 texColor = texture(samp, fract(uv));
    color = texColor;
}

```

Shader #1020 - tremor4

File: tremor4.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 tremorEffect(vec2 uv, float baseFrequency, float baseAmplitude, float timeFactor) {
    float duration = mod(time_f, timeFactor);
    float frequency = baseFrequency * (1.0 + 0.5 * duration / timeFactor);
    float amplitude = baseAmplitude * (1.0 + 0.5 * duration / timeFactor);
    float tremorX = sin(uv.y * frequency + time_f * 10.0) * amplitude;
    float tremorY = cos(uv.x * frequency + time_f * 10.0) * amplitude;
    return uv + vec2(tremorX, tremorY);
}
void main() {
    vec2 uv = tc;
    float baseFrequency = 20.0;
    float baseAmplitude = 0.01;
    float timeFactor = 5.0;
    uv = tremorEffect(uv, baseFrequency, baseAmplitude, timeFactor);
    vec4 texColor = texture(samp, fract(uv));
    color = texColor;
}

```

Shader #1025 - tv show

File: tv_show.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Computes luminance for brightness-based effects.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

```

```

float noise(float x) {
    return fract(sin(x) * 43758.5453);
}
void main(void) {
    vec2 uv = tc;
    vec2 center = uv - 0.5;
    uv = uv + center * (0.02 * sin(time_f * 1.5));
    float timeBasedNoise = noise(time_f * 10.0) * 0.05;
    float spatialNoise = noise(uv.y * 10.0 + time_f * 5.0) * 0.05;
    float combinedNoise = timeBasedNoise + spatialNoise;
    vec3 texColor = texture(samp, uv).rgb;
    float gray = dot(texColor, vec3(0.299, 0.587, 0.114));
    texColor = vec3(gray);
    texColor = texColor * (1.0 - combinedNoise);
    color = vec4(texColor, 1.0);
}

```

Shader #1028 - twirl tex

File: twirl_tex.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 normCoord = tc - center;
    normCoord.x *= iResolution.x / iResolution.y;
    float distance = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float twirlAmount = 5.0;
    angle += distance * twirlAmount * sin(time_f);
    vec2 twirledCoord = vec2(cos(angle), sin(angle)) * distance;
    twirledCoord.x *= iResolution.y / iResolution.x;
    color = texture(samp, twirledCoord + center);
}

```

Shader #1059 - weirdlines

File: weirdlines.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;

```

```

uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec3 st1 = gl_FragCoord.xyz;
    for(int i = 0; i < 3; ++i) {
        color[i] = color[i]*timeval*st1[i];
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #1060 - whirlex

File: whirlex.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
in vec2 tc;
out vec4 color;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main() {

```

```

    vec2 uv = tc - 0.5;
    uv.y *= iResolution.y / iResolution.x;
    float intensity = pingPong(time_f, 10.0);
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float swirl = sin(time_f * 0.5) * 0.5 + 0.5;
    angle += intensity * swirl * sin(radius * 10.0 + time_f);
    uv = vec2(cos(angle), sin(angle)) * radius;
    vec4 texColor = texture(samp, uv + 0.5);
    color = texColor;
}

```

Shader #1064 - wrap around

File: wrap_around.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(color * 255);
    ivec4 inc = ivec4(inc_valuex);
    ivec4 inc_v = ivec4(inc_value);
    source += inc + inc_v;
    for(int i = 0; i < 3; ++i) {
        source[i] = source[i]%255;
        color[i] = float(source[i])/255;
    }
}

```

Shader #1066 - xcordstrobe

File: xcordstrobe.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = sin(color[0]+(tc[0]))*(timeval*alpha_r);
    color[1] = cos(color[1]+(tc[1]))*(timeval*alpha_g);
    color[2] = tan(color[2]+(tc[0]))*(timeval*alpha_b);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #1089 - yin yang

File: yin-yang.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Distance field computation - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
```

```

    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + time_f * 20.0;
    float radius = length(uv);
    float circle = step(0.5, radius) * step(radius, 0.75);
    float dot1 = step(distance(uv, vec2(0.25 * cos(time_f), 0.25 * sin(time_f))), 0.1);
    float dot2 = step(distance(uv, vec2(-0.25 * cos(time_f), -0.25 * sin(time_f))), 0.1);
    float yin_yang = circle + dot1 + dot2;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, yin_yang);
    color = vec4(sin(blended_color * time_f), original_color.a);
}

```

Shader #1090 - zigzag

File: zigzag.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 normCoord = tc;
    float timeAdjustedFrequency = 10.0 + sin(time_f) * 5.0;
    float timeAdjustedAmplitude = 0.01 + (sin(time_f * 0.5) * 0.5 + 0.5) * 0.05;
    float zigzagFactor = abs(fract(normCoord.y * timeAdjustedFrequency) - 0.5) * 2.0;
    float xDisplacement = zigzagFactor * timeAdjustedAmplitude;
    vec2 displacedCoord = vec2(normCoord.x + xDisplacement, normCoord.y);
    displacedCoord = clamp(displacedCoord, 0.0, 1.0);
    color = texture(samp, displacedCoord);
}

```

Color Manipulation (118 shaders)

Shader #25 - all colors

File: all_colors.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;

```

```

in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
vec4 xor_RGB(vec4 a, vec4 b) {
    ivec3 ai;
    ivec4 bi = ivec4(clamp(b, 0.0, 1.0) * 255.0 + 0.5);
    for (int i = 0; i < 3; ++i) {
        ai[i] = int(clamp(a[i], 0.0, 1.0) * 255.0 + 0.5);
        ai[i] = ai[i] ^ bi[i];
        ai[i] = ai[i] % 256;
        a[i] = float(ai[i]) / 255.0;
    }
    a.a = 1.0;
    return a;
}
vec4 blur(sampler2D image, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    float k[100];
    float v[100] = float[](
        0.5,1.0,1.5,2.0,2.5,2.5,2.0,1.5,1.0,0.5,
        1.0,2.0,2.5,3.0,3.5,3.5,3.0,2.5,2.0,1.0,
        1.5,2.5,3.0,3.5,4.0,4.0,3.5,3.0,2.5,1.5,
        2.0,3.0,3.5,4.0,4.5,4.5,4.0,3.5,3.0,2.0,
        2.5,3.5,4.0,4.5,5.0,5.0,4.5,4.0,3.5,2.5,
        2.5,3.5,4.0,4.5,5.0,5.0,4.5,4.0,3.5,2.5,
        2.0,3.0,3.5,4.0,4.5,4.5,4.0,3.5,3.0,2.0,
        1.5,2.5,3.0,3.5,4.0,4.0,3.5,3.0,2.5,1.5,
        1.0,2.0,2.5,3.0,3.5,3.5,3.0,2.5,2.0,1.0,
        0.5,1.0,1.5,2.0,2.5,2.5,2.0,1.5,1.0,0.5
    );
    for (int i = 0; i < 100; ++i) k[i] = v[i];
    float s = 0.0;
    for (int i = 0; i < 100; ++i) s += k[i];
    vec4 r = vec4(0.0);
    for (int y = -5; y <= 4; ++y) {
        for (int x = -5; x <= 4; ++x) {
            int idx = (y + 5) * 10 + (x + 5);
            r += texture(image, uv + vec2(float(x), float(y)) * ts) * k[idx];
        }
    }
    return r / s;
}
float pingPong(float x, float len) {
    float m = mod(x, len * 2.0);
    return m <= len ? m : (len * 2.0 - m);
}
void main(void) {
    vec3 col = blur(samp, tc, iResolution).rgb;
    float t = mod(time_f, 6.0);
    float mcv = 1.0;
    if (t < 1.0) {
        col.r = mix(0.0, mcv, t);
    } else if (t < 2.0) {
        col.r = mcv;
        col.g = mix(0.0, mcv, t - 1.0);
    } else if (t < 3.0) {
        col.r = mcv;
        col.g = mcv;
    }
}

```



```

        col.b = mix(0.0, mcv, t - 2.0);
    } else if (t < 4.0) {
        col = vec3(mcv);
        col.b = mix(mcv, alpha_b, t - 3.0);
    } else if (t < 5.0) {
        col = vec3(mcv, mcv, alpha_b);
        col.g = mix(mcv, alpha_g, t - 4.0);
    } else {
        col = vec3(mcv, alpha_g, alpha_b);
        col.r = mix(mcv, alpha_r, t - 5.0);
    }
    vec4 cyc = vec4(col, 1.0);
    vec4 xr = xor_RGB(blur(samp, tc, iResolution), cyc);
    float tt = pingPong(time_f, 20.0) + 2.0;
    vec3 s1 = 0.5 + 0.5 * sin(xr.rgb * tt);
    vec3 s2 = 0.5 + 0.5 * sin((xr.rgb * 6.28318) + vec3(0.0, 2.0943951, 4.1887902));
    vec3 allColors = mix(s1, s2, 0.65);
    color = vec4(clamp(allColors, 0.0, 1.0), 1.0);
}

```

Shader #39 - audiocolor

File: audiocolor.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Texcoords passed from vertex shader Time (in seconds, typically) The texture we're sampling

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
vec2 swirl(vec2 uv, float centerRadius, float swirlAmount) {
    float r = length(uv);
    float a = atan(uv.y, uv.x);
    float f = smoothstep(centerRadius, 0.0, r);
    a += swirlAmount * f;
    return r * vec2(cos(a), sin(a));
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 rainbow(float t) {
    return vec3(
        sin(2.0 * 3.14159 * t),
        sin(2.0 * 3.14159 * t + 2.0943951),
        sin(2.0 * 3.14159 * t + 4.18879)
    ) * 0.5 + 0.5;
}
void main(void)
{

```

```

float A = uamp * amp / time_f;
vec2 uv = tc * 2.0 - 1.0;
float aspect = iResolution.x / iResolution.y;
uv.x *= aspect;
uv = swirl(uv, 0.8, 2.0 * A * sin(time_f * 0.5));
uv.x /= aspect;
uv = uv * 0.5 + 0.5;
vec4 baseColor = texture(samp, uv);
float rainbowFactor = time_f * 0.2 + uv.x + uv.y + A * 2.0;
vec3 rainbowColor = rainbow(fract(rainbowFactor));
baseColor.rgb = mix(baseColor.rgb, rainbowColor, 0.75);
uv = swirl(uv - 0.5, 1.0, A * 10.0) + 0.5;
baseColor *= texture(samp, uv);
float t_amp = pingPong(amp, 8.0) + 1.0;
baseColor.rgb *= (t_amp + uamp);
baseColor = clamp(baseColor, 0.0, 1.0);
float time_t = pingPong(time_f, 6.0) + 1.0;
color = sin(baseColor * time_t);
color.a = 1.0;
}

```

Shader #40 - aura

File: aura.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
void main(void){
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float radius = mix(0.8, 1.2, 0.5 + 0.5 * sin(time_f * 1.3));
    radius *= 2;
    float r = length(uv);
    float glow = smoothstep(radius, radius - 0.25, r) * pingPong(time_f, 5.0);
    float ang = atan(uv.y, uv.x);
    vec3 grad = hsv2rgb(vec3(fract(ang / (2.0 * PI) + time_f * 0.05), 0.8, 1.0));
    vec4 base = texture(samp, tc);
    vec3 aura = grad * glow * (0.8 + 0.2 * sin(time_f * 2.0));
    color = vec4(base.rgb + aura * (1.0 - base.rgb), base.a);
}

```

Shader #61 - bit rainbow

File: bit_rainbow.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Indexed color palette look-up table - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float NESQuantize(float val) {
    return floor(val * 16.0) / 15.0;
}
vec3 NESPalette(vec3 col) {
    col.r = NESQuantize(col.r);
    col.g = NESQuantize(col.g);
    col.b = NESQuantize(col.b);
    return col;
}
vec3 rainbowGradient(float t) {
    float r = 0.5 + 0.5 * cos(6.28318 * t + 0.0);
    float g = 0.5 + 0.5 * cos(6.28318 * t + 2.09439);
    float b = 0.5 + 0.5 * cos(6.28318 * t + 4.18878);
    return vec3(r, g, b);
}
void main(void) {
    vec2 uv = tc * iResolution.xy / min(iResolution.x, iResolution.y);
    vec2 center = vec2(0.5, 0.5);
    float angle = atan(uv.y - center.y, uv.x - center.x) + time_f;
    float radius = length(uv - center);
    vec3 gradientColor = rainbowGradient(angle / (2.0 * 3.14159));
    gradientColor *= smoothstep(0.4, 0.6, radius);
    vec4 texColor = texture(samp, tc);
    vec3 blendedColor = mix(texColor.rgb, gradientColor, 0.5);
    vec3 nesColor = NESPalette(blendedColor);
    color = vec4(nesColor, texColor.a);
}
```

Shader #80 - bright rainbow

File: bright_rainbow.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
```

```

    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + time_f * 20.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(blended_color, original_color.a);
}

```

Shader #97 - chue

File: chue.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Computes luminance for brightness-based effects. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 adjustHue(vec4 color, float angle) {
    float U = cos(angle);
    float W = sin(angle);
    mat3 rotationMatrix = mat3(
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114
    ) + mat3(
        0.701, -0.587, -0.114,
        -0.299, 0.413, -0.114,
        -0.3, -0.588, 0.886
    ) * U + mat3(
        0.168, 0.330, -0.497,
        -0.328, 0.035, 0.292,
        1.25, -1.05, -0.203
    ) * W;
    return vec4(rotationMatrix * color.rgb, color.a);
}

void main() {
    vec2 uv = tc;
    vec4 sampledColor = texture(samp, uv);
    float hueShift = sin(uv.x * 10.0 + uv.y * 10.0 + time_f);
    vec4 adjustedColor = adjustHue(sampledColor, hueShift);
    color = adjustedColor;
}

```

Shader #105 - color f diag

File: color-f-diag.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    vec4 ctx = texture(samp, tc);
    float adjusted_time = time_f * 0.1;
    float hue = mod((tc.x + tc.y) / sqrt(2.0) + adjusted_time, 1.0);
    vec3 hsvColor = vec3(hue, 1.0, 1.0);
    vec3 smoothColor = hsv2rgb(hsvColor);
    ctx.rgb = mix(ctx.rgb, smoothColor, 0.5);
    color = ctx;
}
```

Shader #106 - color f

File: color-f.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    vec4 ctx = texture(samp, tc);
    float adjusted_time = time_f * 0.1;
    float hue = mod(tc.x + adjusted_time, 1.0);
    vec3 hsvColor = vec3(hue, 1.0, 1.0);
    vec3 smoothColor = hsv2rgb(hsvColor);
    ctx.rgb = mix(ctx.rgb, smoothColor, 0.5);
    color = ctx;
}
```

Shader #107 - color f2

File: color-f2.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    vec4 ctx = texture(samp, tc);
    float adjusted_time = time_f * 0.09;
    float hue = mod((tc.x + tc.y) / sqrt(2.0) + adjusted_time, 1.0);
    vec3 hsvColor = vec3(hue, 1.0, 1.0);
    vec3 smoothColor = hsv2rgb(hsvColor);
    ctx.rgb = mix(ctx.rgb, smoothColor, 0.5);
    float modulation = (sin(time_f) + 1.0) / 2.0;
    vec4 modulatedColor = mix(ctx, sin(ctx * time_f), modulation);
    color = modulatedColor;
}
```

Shader #108 - color max

File: color-max.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
}
```

```

    icolor.a = 1.0;
return icolor;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec3 colorValue = blur(samp, tc, iResolution).rgb;
    float time_t = pingPong(time_f, 20.0) + 2.0;
    float maxColorValue = 1.0;
    float timeCycle = mod(time_f, 6.0);
    if (timeCycle < 1.0) {
        colorValue.r = mix(0.0, maxColorValue, timeCycle);
    } else if (timeCycle < 2.0) {
        colorValue.r = maxColorValue;
        colorValue.g = mix(0.0, maxColorValue, timeCycle - 1.0);
    } else if (timeCycle < 3.0) {
        colorValue.r = maxColorValue;
        colorValue.g = maxColorValue;
        colorValue.b = mix(0.0, maxColorValue, timeCycle - 2.0);
    } else if (timeCycle < 4.0) {
        colorValue.r = maxColorValue;
        colorValue.g = maxColorValue;
        colorValue.b = maxColorValue;
        colorValue.b = mix(maxColorValue, alpha_b, timeCycle - 3.0);
    } else if (timeCycle < 5.0) {
        colorValue.r = maxColorValue;
        colorValue.g = maxColorValue;
        colorValue.b = alpha_b;
        colorValue.g = mix(maxColorValue, alpha_g, timeCycle - 4.0);
    } else {
        colorValue.r = maxColorValue;
    }
}

```

```

        colorValue.g = alpha_g;
        colorValue.b = alpha_b;
        colorValue.r = mix(maxColorValue, alpha_r, timeCycle - 5.0);
    }
    color = vec4(colorValue, 1.0);
    color = xor_RGB(blur(samp, tc, iResolution), color);
    color = sin(color * time_t);
    color.a = 1.0;
}

```

Shader #109 - color mod dark

File: color-mod-dark.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    vec4 ctx = texture(samp, tc);
    float adjusted_time = time_f * 0.09;
    float hue = mod((tc.x + tc.y) / sqrt(2.0) + adjusted_time, 1.0);
    vec3 hsvColor = vec3(hue, 1.0, 1.0);
    vec3 smoothColor = hsv2rgb(hsvColor);
    vec3 blendedColor = mix(ctx.rgb, smoothColor, 0.5);
    float modulation = (sin(time_f) + 1.0) / 2.0;
    vec3 darkColor = blendedColor * 0.5;
    vec3 brightColor = blendedColor * 1.5;
    vec3 finalColor = mix(darkColor, brightColor, modulation);
    color = vec4(finalColor, ctx.a);
}

```

Shader #110 - color mod

File: color-mod.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;

```



```

uniform float time_f;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    vec4 ctx = texture(samp, tc);
    float adjusted_time = time_f * 0.09;
    float hue = mod((tc.x + tc.y) / sqrt(2.0) + adjusted_time, 1.0);
    vec3 hsvColor = vec3(hue, 1.0, 1.0);
    vec3 smoothColor = hsv2rgb(hsvColor);
    vec3 blendedColor = mix(ctx.rgb, smoothColor, 0.5);
    float modulation = (sin(time_f) + 1.0) / 2.0;
    vec4 modulatedColor = mix(ctx, vec4(blendedColor, ctx.a), modulation);
    color = modulatedColor;
}

```

Shader #111 - color o

File: color-o.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    vec4 ctx = texture(samp, tc);
    float adjusted_time = time_f * 0.05;
    float hue = mod(tc.x + tc.y * 0.5 + adjusted_time, 1.0);
    vec3 hsvColor = vec3(hue, 1.0, 1.0);
    vec3 smoothColor = hsv2rgb(hsvColor);
    ctx.rgb = mix(ctx.rgb, smoothColor, 0.7);
    color = ctx;
}

```

Shader #112 - color effect

File: color_effect.glsl

Generates procedural noise patterns combined with sinusoidal distortion. + feedback * 0.01 warp
/+ feedback 0.01*/;

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
void main(void) {
    vec2 uv = tc;
    vec2 warp = uv + vec2(
        sin(uv.y * 10.0 + time_f) * 0.1,
        sin(uv.x * 10.0 + time_f) * 0.1
    );
    vec3 colorShift = vec3(
        0.5 * sin(time_f * 0.5) + 0.5,
        0.5 * sin(time_f * 0.7 + 2.0) + 0.5,
        0.5 * sin(time_f * 0.3 + 4.0) + 0.5
    );
    float feedback = rand(uv + time_f);
    vec2 feedbackUv = tc;
    float time_t = mod(time_f, 50);
    vec4 texColor = texture(samp, feedbackUv);
    vec3 finalColor = texColor.rgb + colorShift;
    color = tan(vec4(finalColor, texColor.a) * time_t);
    color.a = 1.0;
}

```

Shader #114 - color flux

File: color_flux.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
in vec2 tc;
out vec4 color;

```

```

vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i] ^ isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i] % 255;
        icolor[i] = float(int_color[i]) / 255;
    }
    icolor.a = 1.0;
return icolor;
}

void main() {
    vec4 texColor = texture(samp, tc);
    float fluctuation = sin(time_f * (2.0 + amp_mid * 4.0)) * 0.5 + 0.5;
    vec3 extremeColor1 = vec3(1.0, 0.0, 0.0);
    vec3 extremeColor2 = vec3(0.0, 0.0, 1.0);
    vec3 fluctuatedColor = mix(extremeColor1, extremeColor2, fluctuation);
    vec4 fluctuatedVec = vec4(fluctuatedColor, 1.0);
    color = xor_RGB(texColor, fluctuatedVec);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #115 - color flux whirl

File: color_flux_whirl.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
in vec2 tc;
out vec4 color;

vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for (int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i] ^ isource[i];
        if (int_color[i] > 255)
            int_color[i] = int_color[i] % 255;
        icolor[i] = float(int_color[i]) / 255;
    }
    icolor.a = 1.0;
return icolor;
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

```

```

}
void main() {
    vec2 uv = tc - 0.5;
    uv.y *= iResolution.y / iResolution.x;
    float intensity = pingPong(time_f, 10.0);
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float swirl = sin(time_f * 0.5) * 0.5 + 0.5;
    angle += intensity * swirl * sin(radius * 10.0 + time_f);
    uv = vec2(cos(angle), sin(angle)) * radius;
    vec4 texColor = texture(samp, uv + 0.5);
    float fluctuation = sin(time_f * 2.0) * 0.5 + 0.5;
    vec4 fluctuatedColor = vec4(mix(vec3(1.0, 0.0, 0.0), vec3(0.0, 0.0, 1.0), fluctuation), 1.0);
    vec4 xorResult = xor_RGB(texColor, fluctuatedColor);
    color = mix(texColor, xorResult, 0.5);
}

```

Shader #116 - color g

File: color_g.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)),
u.x),
                mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)),
u.x), u.y);
}
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float t = time_f * 0.7;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t * 0.5;
    float radMod = pingPong(radius + t * 0.3, 0.5);
    float wave = sin(radius * 10.0 - t * 6.0) * 0.5 + 0.5;
    float noiseEffect = noise(uv * 10.0 + t * 0.5) * 0.2;
    float r = sin(angle * 3.0 + radMod * 8.0 + wave * 6.2831 +
noiseEffect);
}

```

```

    float g = sin(angle * 4.0 - radMod * 6.0 + wave * 4.1230 +
noiseEffect);
    float b = sin(angle * 5.0 + radMod * 10.0 - wave * 3.4560 -
noiseEffect);
    float blur = exp(-radius * 10.0) * 0.8 + 0.2;
    vec3 col = vec3(r, g, b) * blur * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc + sin(t * 0.2 + radius) * 0.01).rgb;
    col = mix(col, texColor, 0.4);
    color = vec4(col, alpha);
}

```

Shader #118 - color grad1

File: color_grad1.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
uniform vec4 inc_valuex;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void)
{
    color = texture(samp, tc);
    vec2 pos = gl_FragCoord.xy / iResolution;
    vec4 s = color * sin(inc_valuex / 255.0 * time_f);
    color[0] += s[0] * pos[0];
    color[1] += s[1] * pos[1];
    float time_t = pingPong(time_f, 20) + 2.0;
    color = sin(color * time_t);
    color.a = 1.0;
}

```

Shader #119 - color grad rainbow

File: color_grad_rainbow.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

```

```

uniform float restore_black;
uniform vec4 inc_valuex;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_t), original_color.a);
    vec2 pos = gl_FragCoord.xy / iResolution;
    vec4 s = color * sin(inc_valuex / 255.0 * time_f);
    color[0] += s[0] * pos[0];
    color[1] += s[1] * pos[1];
    time_t = pingPong(time_f, 20) + 2.0;
    color = sin(color * time_t);
    color.a = 1.0;
}

```

Shader #120 - color increase

File: color_increase.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
```

```

in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    color = (color * 0.5) + (0.5 * fract(color * time_f));
}

```

Shader #121 - color 1

File: color_1.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)),
u.x),
                mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)),
u.x), u.y);
}
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float t = time_f * 0.7;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t * 0.5;
}

```

```

    float radMod = pingPong(radius + t * 0.3, 0.5);
    float wave = sin(radius * 10.0 - t * 6.0) * 0.5 + 0.5;
    float noiseEffect = noise(uv * 10.0 + t * 0.5) * 0.2;
    float r = sin(angle * 3.0 + radMod * 8.0 + wave * 6.2831 +
noiseEffect);
    float g = sin(angle * 4.0 - radMod * 6.0 + wave * 4.1230 +
noiseEffect);
    float b = sin(angle * 5.0 + radMod * 10.0 - wave * 3.4560 -
noiseEffect);
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc).rgb;
    col = mix(col, texColor, 0.6);
    vec3 lightDir = normalize(vec3(0.5, 0.5, 1.0));
    vec3 norm = normalize(vec3(uv, sqrt(1.0 - dot(uv, uv))));
    float light = dot(norm, lightDir) * 0.5 + 0.5;
    col *= light * 1.2;
    color = vec4(col, alpha);
}

```

Shader #122 - color pool

File: color_pool.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void) {
    float time_t = mod(time_f, 100);
    float scale = time_t;
    float speed = 16.0;
    float offset = sin(time_f * speed + tc.x * scale) * 0.05;
    vec2 tcOffset = vec2(tc.x, tc.y + offset);
    color = texture(samp, tcOffset);
    vec4 color2 = vec4(0.8, 0.2, 0.8, 1.0);
    color = sin(xor_RGB(color, color2) * time_t);
    color.a = 1.0;
}

```


Shader #123 - color swirl o1

File: color_swirl-o1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float t = time_f * 0.5;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t;
    float radMod = pingPong(radius + t * 0.5, 0.5);
    float wave = sin(radius * 10.0 - t * 5.0) * 0.5 + 0.5;
    float r = sin(angle * 3.0 + radMod * 10.0 + wave * 6.2831);
    float g = sin(angle * 4.0 - radMod * 8.0 + wave * 4.1230);
    float b = sin(angle * 5.0 + radMod * 12.0 - wave * 3.4560);
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc).rgb;
    col = mix(col, texColor, 0.3);
    color = vec4(col, alpha);
}
```

Shader #124 - color swirl beautiful

File: color_swirl_beautiful.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Calculate radius and angle for wave effect Add ping-pong to create a pulsating effect around the center Sine wave for additional distortion and movement

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
```

```

    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float t = time_f * 0.5;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t;
    float radMod = pingPong(radius + t * 0.5, 0.5);
    float wave = sin(radius * 10.0 - t * 5.0) * 0.5 + 0.5;
    float r = sin(angle * 3.0 + radMod * 10.0 + wave * 6.2831);
    float g = sin(angle * 4.0 - radMod * 8.0 + wave * 4.1230);
    float b = sin(angle * 5.0 + radMod * 12.0 - wave * 3.4560);
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc).rgb;
    col = mix(col, texColor, 0.3);
    color = vec4(col, alpha);
}

```

Shader #125 - color swirl c

File: color_swirl_c.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float smoothNoise(vec2 uv) {
    return sin(uv.x * 12.0 + uv.y * 14.0 + time_f * 0.8) * 0.5 + 0.5;
}
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    float swirl = sin(time_f * 0.4) * 2.0;
    angle += swirl * radius * 1.5;
    float modRadius = pingPong(radius + time_f * 0.3, 0.8);
    float wave = sin(radius * 12.0 - time_f * 3.0) * 0.5 + 0.5;
    float cloudNoise = smoothNoise(uv * 3.0 + vec2(modRadius, angle * 0.5));
    cloudNoise += smoothNoise(uv * 6.0 - vec2(time_f * 0.2, time_f * 0.1));
    cloudNoise = pow(cloudNoise, 1.5);
    float r = sin(angle * 3.0 + modRadius * 8.0 + wave * 2.0) * cloudNoise;
    float g = sin(angle * 5.0 - modRadius * 6.0 + wave * 4.0) * cloudNoise;
    float b = sin(angle * 7.0 + modRadius * 10.0 - wave * 3.0) * cloudNoise;
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc).rgb;
    col = mix(col, texColor, 0.25);
    color = vec4(sin(col * pingPong(time_f * 1.5, 6.0) + 1.5), alpha);
}

```

Shader #126 - color swirl g

File: color_swirl_g.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;

float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}

float smoothNoise(vec2 uv) {
    return sin(uv.x * 12.0 + uv.y * 14.0 + time_f * 0.8) * 0.5 + 0.5;
}

void main(void) {
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 c = (m * iResolution - 0.5 * iResolution) / iResolution.y;
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y - (c - vec2(0.0));
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    float mx = (iMouse.z > 0.5) ? (iMouse.x / iResolution.x) : 0.5;
    float my = (iMouse.z > 0.5) ? (iMouse.y / iResolution.y) : 0.5;
    float swirl = sin(time_f * mix(0.25, 0.8, mx)) * mix(1.2, 3.0, my);
    angle += swirl * radius * 1.5;
    float modRadius = pingPong(radius + time_f * mix(0.15, 0.45, mx), 0.8);
    float wave = sin(radius * mix(8.0, 18.0, my) - time_f * mix(2.0, 5.0, mx)) * 0.5 + 0.5;
    float cloudNoise = smoothNoise(uv * 3.0 + vec2(modRadius, angle * 0.5));
    cloudNoise += smoothNoise(uv * 6.0 - vec2(time_f * 0.2, time_f * 0.1));
    cloudNoise = pow(cloudNoise, 1.5);
    float r = sin(angle * 3.0 + modRadius * 8.0 + wave * 2.0) * cloudNoise;
    float g = sin(angle * 5.0 - modRadius * 6.0 + wave * 4.0) * cloudNoise;
    float b = sin(angle * 7.0 + modRadius * 10.0 - wave * 3.0) * cloudNoise;
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc).rgb;
    col = mix(col, texColor, 0.5);
    float t = pingPong(time_f * 1.5, 6.0);
    color = vec4(sin(col * t + 1.5), alpha);
}
```

Shader #128 - color swirl sin

File: color_swirl_sin.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float t = time_f * 0.5;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t;
    float radMod = pingPong(radius + t * 0.5, 0.5);
    float wave = sin(radius * 10.0 - t * 5.0) * 0.5 + 0.5;
    float r = sin(angle * 3.0 + radMod * 10.0 + wave * 6.2831);
    float g = sin(angle * 4.0 - radMod * 8.0 + wave * 4.1230);
    float b = sin(angle * 5.0 + radMod * 12.0 - wave * 3.4560);
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc).rgb;
    col = mix(col, texColor, 0.3);
    color = vec4(sin(col * pingPong(time_f, 8.0) + 2.0), alpha);
}

```

Shader #130 - color timeval

File: color_timeval.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;

```

```

    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec2 cord1 = vec2(tc[0]/2, tc[1]/2);
    vec2 cord2 = vec2(tc[0]/4, tc[1]/4);
    vec2 cord3 = vec2(tc[0]/8, tc[1]/8);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    color[0] = (color[0] * col1[0]) * timeval;
    color[1] = (color[1] * col2[1]) * timeval;
    color[2] = (color[2] * col3[2]) * timeval;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #131 - color w

File: color_w.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float t = time_f * 0.7;
    float beat = abs(sin(time_f * 3.14159)) * 0.2 + 0.8;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t * 0.5;
    float radMod = pingPong(radius + t * 0.3, 0.5);
}

```

```

float wave = sin(radius * 10.0 - t * 6.0) * 0.5 + 0.5;
float distortion = sin((radius + t * 0.5) * 8.0) * beat * 0.1;
uv += vec2(cos(angle), sin(angle)) * distortion;
vec3 texColor = texture(samp, tc + distortion).rgb;
float noiseEffect = noise(uv * 10.0 + t * 0.5) * 0.2;
float r = sin(angle * 3.0 + radMod * 8.0 + wave * 6.2831 + noiseEffect);
float g = sin(angle * 4.0 - radMod * 6.0 + wave * 4.1230 + noiseEffect);
float b = sin(angle * 5.0 + radMod * 10.0 - wave * 3.4560 - noiseEffect);
vec3 col = vec3(r, g, b) * 0.5 + 0.5;
col = mix(col, texColor, 0.6);
vec3 lightDir = normalize(vec3(0.5, 0.5, 1.0));
vec3 norm = normalize(vec3(uv, sqrt(1.0 - dot(uv, uv))));
float light = dot(norm, lightDir) * 0.5 + 0.5;
col *= light * 1.2;
col *= beat;
color = vec4(col, alpha);
}

```

Shader #132 - color w2

File: color_w2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float t = time_f * 0.7;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    float swirlAmount = sin(t) * 2.0;
    angle += swirlAmount * exp(-radius * 4.0);
    uv = vec2(cos(angle), sin(angle)) * radius;
    float wave = sin(radius * 10.0 + t * 6.0) * 0.05;
    uv += vec2(cos(angle), sin(angle)) * wave;
    vec3 texColor = texture(samp, uv * 0.5 + 0.5).rgb;
    float modAngle = angle + t * 0.5;
    vec3 swirlColor = vec3(
        sin(modAngle * 3.0),
        sin(modAngle * 2.0 + 1.0),
        sin(modAngle * 1.0 + 2.0)
    ) * 0.5 + 0.5;
    vec3 finalColor = mix(swirlColor, texColor, 0.7);
    float pulse = abs(sin(time_f * 3.14159)) * 0.2 + 0.8;
    finalColor *= pulse;
    color = vec4(finalColor, alpha);
}

```

```
}
```

Shader #135 - colorsheet

File: colorsheet.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec2 cord1 = vec2(tc[0]/2, tc[0]/2);
    vec2 cord2 = vec2(tc[0]/4, tc[0]/4);
    vec2 cord3 = vec2(tc[0]/8, tc[0]/8);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec2 p = gl_FragCoord.xy / 32;
    vec2 d = fract(p);
    color[1] = (color[1]+col1[1])*(d[0]+d[1]);
    color[2] = (color[2]+col2[2])*(d[0]+d[1]);
}
```

Shader #154 - coord color

File: coord_color.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.
Applies matrix transformations.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
```

```

uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
in vec2 iResolution_;
uniform float restore_black;
uniform vec4 inc_valuex;
void main(void)
{
    color = texture(samp, tc);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec4 s = color * sin(inc_valuex/255 * timeval);
    color[0] += s[0] * pos[0];
    color[1] += s[1] * pos[1];
}

```

Shader #185 - dark rainbow

File: dark_rainbow.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + time_f * 20.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_f), original_color.a);
}

```

Shader #186 - dark rainbow2

File: dark_rainbow2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + pingPong(time_f * PI, 5.0) * 20.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_f), original_color.a);
}
```

Shader #187 - dark rainbow limit

File: dark_rainbow_limit.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float t = pingPong(time_f, 20) + 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + t * 20.0;
```

```

    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * t), original_color.a);
}

```

Shader #188 - dark rainbow rubber soul

File: dark_rainbow_rubber_soul.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + time_f * 20.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    float radius = length(uv);
    float bulge_strength = 0.2;
    float distortion = pow(radius, 2.0) * bulge_strength;
    vec2 distorted_uv = uv * (1.0 + distortion);
    vec4 original_color = texture(samp, distorted_uv * 0.5 + 0.5);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_f), original_color.a);
}

```

Shader #201 - deepcolor

File: deepcolor.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Normalized pixel coordinates Create dynamic wave distortion Add rotating distortion

Technique: RGB <-> HSV color space conversion - Per-channel UV offset for chromatic aberration - Scanline overlay using modular y-coordinate - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;

```

```

uniform sampler2D samp;
uniform vec2 iResolution;
#define PI 3.14159265359
#define TAU 6.28318530718
mat2 rotate(float angle) {
    float s = sin(angle);
    float c = cos(angle);
    return mat2(c, -s, s, c);
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 waveDistort(vec2 uv, float intensity) {
    vec2 d = vec2(0.0);
    d.x = intensity * sin(uv.y * 5.0 + time_f * 2.0) * cos(uv.x * 3.0 + time_f);
    d.y = intensity * cos(uv.x * 4.0 + time_f * 1.5) * sin(uv.y * 2.0 + time_f);
    return uv + d * 0.1;
}
vec3 chromaticAberration(sampler2D tex, vec2 uv, float amount) {
    float r = texture(tex, uv + vec2(amount, 0.0)).r;
    float g = texture(tex, uv).g;
    float b = texture(tex, uv - vec2(amount, 0.0)).b;
    return vec3(r, g, b);
}
void main() {
    vec2 uv = tc;
    vec2 distortedUV = waveDistort(uv, 1.0 + sin(time_f * 0.5));
    distortedUV -= vec2(0.5);
    distortedUV *= rotate(time_f * 0.3);
    distortedUV += vec2(0.5);
    vec3 texColor = chromaticAberration(samp, distortedUV, 0.005 * (1.0 + sin(time_f)));
    float hueShift = 0.5 * sin(time_f * 0.3) + 0.5 * sin(uv.x * 10.0 + time_f);
    vec3 rainbow = hsv2rgb(vec3(fract(time_f * 0.1 + uv.x * 0.5 + uv.y * 0.3), 0.8, 0.6));
    float wave = 0.5 + 0.5 * sin(uv.x * 20.0 + time_f * 5.0) *
        cos(uv.y * 15.0 + time_f * 4.0);
    vec3 finalColor = texColor * (1.0 + wave * 0.5) + rainbow * wave * 0.3;
    float scanline = 0.9 + 0.1 * sin(uv.y * iResolution.y * 1.5 + time_f * 10.0);
    finalColor *= scanline;
    vec2 vigUV = uv * (1.0 - uv) * 2.0;
    float vignette = pow(vigUV.x * vigUV.y * 15.0, 0.3);
    finalColor *= vignette;
    color = vec4(finalColor, 1.0);
}

```

Shader #210 - dhue

File: dhue.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Computes luminance for brightness-based effects. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;

```

```

uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 adjustHue(vec4 color, float angle) {
    float U = cos(angle);
    float W = sin(angle);
    mat3 rotationMatrix = mat3(
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114
    ) + mat3(
        0.701, -0.587, -0.114,
        -0.299, 0.413, -0.114,
        -0.3, -0.588, 0.886
    ) * U + mat3(
        0.168, 0.330, -0.497,
        -0.328, 0.035, 0.292,
        1.25, -1.05, -0.203
    ) * W;
    return vec4(rotationMatrix * color.rgb, color.a);
}
void main() {
    vec2 uv = (tc - 0.5) * iResolution / min(iResolution.x,
iResolution.y);
    float dist = length(uv);
    float ripple = sin(dist * 12.0 - pingPong(time_f, 10.0) * 10.0) * exp(-dist * 4.0);
    vec4 sampledColor = texture(samp, tc + ripple * 0.01);
    float hueShift = pingPong(time_f, 10.0) * ripple * 2.0;
    color = adjustHue(sampledColor, hueShift);
}

```

Shader #240 - drain rainbow

File: drain_rainbow.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 rainbow(float x) {
    float r = clamp(abs(x*6.0-3.0)-1.0,0.0,1.0);
    float g = clamp(2.0-abs(x*6.0-2.0),0.0,1.0);
    float b = clamp(2.0-abs(x*6.0-4.0),0.0,1.0);
    return vec3(r,g,b);
}
void main(void) {
    float loopDuration = 25.0;
    float currentTime = mod(time_f, loopDuration);
    vec2 aspect = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 normCoord = (tc * 2.0 - 1.0) * aspect;

```

```

    normCoord.x = abs(normCoord.x);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralSpeed = 5.0;
    float inwardSpeed = currentTime / loopDuration;
    angle += (1.0 - smoothstep(0.0, 8.0, dist)) * currentTime * spiralSpeed;
    dist *= 1.0 - inwardSpeed;
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * tan(dist);
    spiralCoord = (spiralCoord / aspect + 1.0) * 0.5;
    vec2 uv = spiralCoord;
    vec2 c = uv - 0.5;
    float r = length(c);
    float t = time_f;
    float k1 = 0.25 * sin(t * 0.6);
    float k2 = 0.10 * cos(t * 0.4);
    float rd = 1.0 + k1 * r*r + k2 * r*r*r*r;
    c *= rd;
    c.x += 0.015 * sin(8.0 * c.y + t * 1.3);
    c.y += 0.015 * sin(8.0 * c.x - t * 1.1);
    uv = clamp(c + 0.5, 0.0, 1.0);
    vec2 px = 1.0 / iResolution;
    vec3 s = texture(samp, uv).rgb * 0.29411765;
    s += texture(samp, uv + vec2(px.x,0)).rgb * 0.17647059;
    s += texture(samp, uv - vec2(px.x,0)).rgb * 0.17647059;
    s += texture(samp, uv + vec2(0,px.y)).rgb * 0.17647059;
    s += texture(samp, uv - vec2(0,px.y)).rgb * 0.17647059;
    float hue = fract(atan(c.y, c.x)/6.28318 + 0.5 + 0.07*time_f) ;
    vec3 rb = rainbow(hue);
    float rbAmt = smoothstep(0.15, 0.85, 0.5 + 0.5*sin(time_f*0.7)) * 0.8;
    vec3 outCol = mix(s, s * rb, rbAmt);
    color = vec4(outCol, 1.0);
}

```

Shader #261 - echo alpha three rgb

File: echo_alpha_three_rgb.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{

```

```

    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / (0.9 + alpha));
    vec4 color3 = texture(samp, tc / (1.5 + alpha));
    vec4 color4 = texture(samp, tc / (2.0 + alpha));
    color[0] = (color[0] * 0.5) + (color2[0] * 0.5);
    color[1] = (color[1] * 0.5) + (color3[1] * 0.5);
    color[2] = (color[2] * 0.5) + (color4[2] * 0.5);
}

```

Shader #267 - echo color

File: echo_color.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec4 inc_value;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc / 4);
    vec4 color4 = texture(samp, tc / 8);
    vec4 color_val = color * inc_value / 255 * alpha;
    color = (color_val * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```

Shader #268 - echo color1

File: echo_color1.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;

```

```

uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    color[1] = 0;
}

```

Shader #269 - echo color2

File: echo_color2.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    color[0] = 0;
}

```

Shader #270 - echo color3

File: echo_color3.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    color[2] = 0;
}
```

Shader #271 - echo color4

File: echo_color4.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
```



```

{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    color[0] = 0;
    color[2] = 0;
}

```

Shader #272 - echo color5

File: echo_color5.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    color[2] = 0;
    color[1] = 0;
}

```

Shader #275 - echo div4 rainbow

File: echo_div4_rainbow.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: UV mirroring/folding for symmetry - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;

```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float t = pingPong(time_f, 15.0);
    float wave = sin(uv.x * 10.0 + t * 2.0) * 0.1;
    float expand = 0.5 + 0.5 * sin(t * 2.0);
    vec2 spiral_uv = uv * expand + vec2(cos(t), sin(t)) * 0.2;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    vec4 color_result = vec4(sin(blended_color * t), original_color.a);
    vec4 color2 = texture(samp, tc / 2);
    vec2 tc1 = tc;
    vec2 tc2 = tc;
    tc1[0] = 1.0 - tc1[0];
    tc2[1] = 1.0 - tc2[1];
    vec4 color3 = texture(samp, tc1 / 4);
    vec4 color4 = texture(samp, tc2 / 4);
    color_result = (color_result * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4);
    color = color_result;
}

```

Shader #283 - echo mix colors

File: echo_mix_colors.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;

```

```

uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color[0] = (0.4 * color[2]) + (0.4 * color2[1]) + (0.4 * color3[1]) + (0.4 * color4[0]);
    color[1] = (0.4 * color[1]) + (0.4 * color2[1]) + (0.4 * color3[2]) + (0.4 * color4[0]);
    color[2] = (0.4 * color[0]) + (0.4 * color2[2]) + (0.4 * color3[2]) + (0.4 * color4[1]);
}

```

Shader #284 - echo mix colors rev

File: echo_mix_colors_rev.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color[2] = (0.4 * color[2]) + (0.4 * color2[1]) + (0.4 * color3[1]) + (0.4 * color4[0]);
    color[1] = (0.4 * color[1]) + (0.4 * color2[1]) + (0.4 * color3[2]) + (0.4 * color4[0]);
    color[0] = (0.4 * color[0]) + (0.4 * color2[2]) + (0.4 * color3[2]) + (0.4 * color4[1]);
}

```

Shader #287 - echo rainbow spin

File: echo_rainbow_spin.glsl

Blends multiple texture samples together using interpolation for a composite effect.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + time_f * 20.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    vec4 color_result = vec4(blended_color, original_color.a);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc / 4);
    vec4 color4 = texture(samp, tc / 8);
    color_result = (color_result * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4);
    color = color_result;
}

```

Shader #288 - echo rainbow swirl

File: echo_rainbow_swirl.glsl

Blends multiple texture samples together using interpolation for a composite effect.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float t = mod(time_f, 15);
    float wave = sin(uv.x * 10.0 + t * 2.0) * 0.1;
    float expand = 0.5 + 0.5 * sin(t * 2.0);
    vec2 spiral_uv = uv * expand + vec2(cos(t), sin(t)) * 0.2;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
}

```

```

    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    vec4 color_result = vec4(sin(blended_color * t), original_color.a);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc / 4);
    vec4 color4 = texture(samp, tc / 8);
    color_result = (color_result * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4);
    color = color_result;
}

```

Shader #293 - echo rgb

File: echo_rgb.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.
 color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc / 4);
    vec4 color4 = texture(samp, tc / 8);
    color[0] = (color[0] * 0.5) + (color2[0] * 0.5);
    color[1] = (color[1] * 0.5) + (color3[1] * 0.5);
    color[2] = (color[2] * 0.5) + (color4[2] * 0.5);
}

```

Shader #294 - echo rgb1

File: echo_rgb1.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Color quantization via floor rounding
 - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;

```

```

in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc;
    float timeOffset = 0.05 * sin(time_f);
    vec3 redLayer = texture(samp, uv + vec2(timeOffset, 0.0)).rgb;
    vec3 greenLayer = texture(samp, uv).rgb;
    vec3 blueLayer = texture(samp, uv - vec2(timeOffset, 0.0)).rgb;
    bool strobe = mod(floor(time_f * 2.0), 2.0) > 0.5;
    vec3 rgbStrobe = vec3(
        strobe ? redLayer.r : blueLayer.r,
        greenLayer.g,
        strobe ? blueLayer.b : redLayer.b
    );
    vec4 echoEffect = texture(samp, uv + vec2(0.01 * sin(time_f), 0.01 * cos(time_f)));
    color = mix(vec4(rgbStrobe, 1.0), echoEffect, 0.5);
}

```

Shader #297 - echo s rgb

File: echo_s_rgb.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void)
{
    vec2 center = vec2(0.5, 0.5);
    float angle = time_f;
    vec2 tc1 = tc;
    vec2 tc2 = tc - center;
    tc2 *= 0.8;
    tc2 = vec2(
        tc2.x * cos(angle + 1.0) - tc2.y * sin(angle + 1.0),
        tc2.x * sin(angle + 1.0) + tc2.y * cos(angle + 1.0)
    );
    tc2 += center;
    tc2 = fract(tc2);
    vec2 tc3 = tc - center;
    tc3 *= 0.6;
    tc3 = vec2(
        tc3.x * cos(angle + 2.0) - tc3.y * sin(angle + 2.0),
        tc3.x * sin(angle + 2.0) + tc3.y * cos(angle + 2.0)
    );
    tc3 += center;
    tc3 = fract(tc3);
    vec4 color1 = texture(samp, tc1);
    vec4 color2 = texture(samp, tc2);
    vec4 color3 = texture(samp, tc3);
    float pi = 3.14159265;
    float offset = 2.0 * pi / 3.0;
    float w0 = sin(time_f) * 0.5 + 0.5;

```

```

float w1 = sin(time_f + offset) * 0.5 + 0.5;
float w2 = sin(time_f + 2.0 * offset) * 0.5 + 0.5;
vec3 weights1 = vec3(w0, 0.0, 0.0);
vec3 weights2 = vec3(0.0, w1, 0.0);
vec3 weights3 = vec3(0.0, 0.0, w2);
vec3 finalColor = color1.rgb * weights1 + color2.rgb * weights2 + color3.rgb * weights3;
color = vec4(finalColor, 1.0);
}

```

Shader #325 - es1

File: es1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
 - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 rainbow(float x) {
    float r = clamp(abs(x*6.0-3.0)-1.0,0.0,1.0);
    float g = clamp(2.0-abs(x*6.0-2.0),0.0,1.0);
    float b = clamp(2.0-abs(x*6.0-4.0),0.0,1.0);
    return vec3(r,g,b);
}
void main(void) {
    float loopDuration = 25.0;
    float currentTime = mod(time_f, loopDuration);
    vec2 aspect = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 normCoord = (tc * 2.0 - 1.0) * aspect;
    normCoord.x = abs(normCoord.x);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralSpeed = 5.0;
    float inwardSpeed = currentTime / loopDuration;
    angle += (1.0 - smoothstep(0.0, 8.0, dist)) * currentTime * spiralSpeed;
    dist *= 1.0 - inwardSpeed;
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * tan(dist);
    spiralCoord = (spiralCoord / aspect + 1.0) * 0.5;
    vec2 uv = spiralCoord;
    vec2 c = uv - 0.5;
    float r = length(c);
    float t = time_f;
    float k1 = 0.25 * sin(t * 0.6);
    float k2 = 0.10 * cos(t * 0.4);
    float rd = 1.0 + k1 * r*r + k2 * r*r*r*r;
    c *= rd;
    c.x += 0.015 * sin(8.0 * c.y + t * 1.3);
    c.y += 0.015 * sin(8.0 * c.x - t * 1.1);
    uv = clamp(c + 0.5, 0.0, 1.0);
    vec2 px = 1.0 / iResolution;
    vec3 s = texture(samp, uv).rgb * 0.29411765;
    s += texture(samp, uv + vec2(px.x,0)).rgb * 0.17647059;
    s += texture(samp, uv - vec2(px.x,0)).rgb * 0.17647059;
}

```

```

s += texture(samp, uv + vec2(0,px.y)).rgb * 0.17647059;
s += texture(samp, uv - vec2(0,px.y)).rgb * 0.17647059;
float hue = fract(atan(c.y, c.x)/6.28318 + 0.5 + 0.07*time_f) ;
vec3 rb = rainbow(hue);
float rbAmt = smoothstep(0.15, 0.85, 0.5 + 0.5*sin(time_f*0.7)) * 0.8;
vec3 outCol = mix(s, sin(s * time_f) * rb, rbAmt);
color = vec4(outCol, 1.0);
}

```

Shader #336 - fat rgb

File: fat-rgb.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float stretch_factor = 1.0 + 0.5 * abs(sin(time_f * 3.14));
    vec2 new_tc = center + dir;
    vec4 original_color = texture(samp, tc);
    vec4 stretched_color = texture(samp, center + vec2(dir.x / stretch_factor, dir.y));
    int channel = int(mod(time_f, 3.0));
    if (channel == 0) {
        color = vec4(stretched_color.r, original_color.g, original_color.b, original_color.a);
    } else if (channel == 1) {
        color = vec4(original_color.r, stretched_color.g, original_color.b, original_color.a);
    } else {
        color = vec4(original_color.r, original_color.g, stretched_color.b, original_color.a);
    }
}

```

Shader #466 - gem color spiral

File: gem-color-spiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Discards pure black pixels for transparency. Controls the size of the ball Discard pixels outside the sphere to keep it a perfect circle Optional: render background texture dimmed

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

```



```

uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

void main(void) {
    vec2 uv = (tc * 2.0 - 1.0);
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float d = length(uv);
    float sphereRadius = 0.8;
    if (d > sphereRadius) {
        color = vec4(texture(samp, tc).rgb * 0.2, alpha);
        return;
    }
    float z = sqrt(sphereRadius * sphereRadius - d * d);
    vec3 normal = normalize(vec3(uv, z));
    float fisheyeRadius = atan(d, z);
    vec2 sphereUV = normalize(uv) * fisheyeRadius;
    float t = time_f * 0.8;
    float r_dist = length(sphereUV);
    float angle = atan(sphereUV.y, sphereUV.x);
    float spiral = angle + (log(r_dist + 0.1) * 3.0) - t * 2.0;
    float r = sin(spiral * 2.0 + t);
    float g = sin(spiral * 2.0 + t + 2.094);
    float b = sin(spiral * 2.0 + t + 4.188);
    vec3 spiralCol = vec3(r, g, b) * 0.5 + 0.5;
    vec3 lightDir = normalize(vec3(1.0, 1.0, 1.0));
    float diff = max(dot(normal, lightDir), 0.0);
    float spec = pow(max(dot(reflect(-lightDir, normal), vec3(0,0,1))), 0.0), 32.0);
    vec3 texColor = texture(samp, tc).rgb;
    vec3 finalCol = spiralCol * (diff + 0.3) + spec;
    finalCol = mix(finalCol, texColor, 0.2);
    finalCol *= smoothstep(sphereRadius, sphereRadius - 0.02, d);
    color = vec4(finalCol, alpha);
}

```

Shader #467 - gem color spsiral

File: gem-color-spsiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Discards pure black pixels for transparency. Controls the size of the ball Discard pixels outside the sphere to keep it a perfect circle Optional: render background texture dimmed

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

```

```

void main(void) {
    vec2 uv = (tc * 2.0 - 1.0);
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float d = length(uv);
    float sphereRadius = 0.8;
    if (d > sphereRadius) {
        color = vec4(texture(samp, tc).rgb * 0.2, alpha);
        return;
    }
    float z = sqrt(sphereRadius * sphereRadius - d * d);
    vec3 normal = normalize(vec3(uv, z));
    float fisheyeRadius = atan(d, z);
    vec2 sphereUV = normalize(uv) * fisheyeRadius;
    float t = time_f * 0.8;
    float r_dist = length(sphereUV);
    float angle = atan(sphereUV.y, sphereUV.x);
    float spiral = angle + (log(r_dist + 0.1) * 3.0) - t * 2.0;
    float r = sin(spiral * 2.0 + t);
    float g = sin(spiral * 2.0 + t + 2.094);
    float b = sin(spiral * 2.0 + t + 4.188);
    vec3 spiralCol = vec3(r, g, b) * 0.5 + 0.5;
    vec3 lightDir = normalize(vec3(1.0, 1.0, 1.0));
    float diff = max(dot(normal, lightDir), 0.0);
    float spec = pow(max(dot(reflect(-lightDir, normal), vec3(0,0,1)), 0.0), 32.0);
    vec3 texColor = texture(samp, tc).rgb;
    vec3 finalCol = spiralCol * (diff + 0.3) + spec;
    finalCol = mix(finalCol, texColor, 0.2);
    finalCol *= smoothstep(sphereRadius, sphereRadius - 0.02, d);
    color = vec4(finalCol, alpha);
}

```

Shader #470 - gem hue frac

File: gem-hue-frac.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Optimized Hue adjustment for smooth transitions Store original UV for global reference Slow down time for a trippier effect

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 hueShift(vec3 col, float hue) {
    const vec3 k = vec3(0.57735, 0.57735, 0.57735);
    float cosAngle = cos(hue);
    return col * cosAngle + cross(k, col) * sin(hue) + k * dot(k, col) * (1.0 - cosAngle);
}
void main() {
    vec2 uv = (tc - 0.5) * iResolution / min(iResolution.x, iResolution.y);
    vec2 uv0 = uv;

```

```

    vec3 finalCol = vec3(0.0);
    float t = time_f * 0.2;
    for (float i = 0.0; i < 4.0; i++) {
        uv = fract(uv * 1.5) - 0.5;
        float d = length(uv) * exp(-length(uv));
        vec3 col = vec3(0.5, 0.8, 0.9);
        d = sin(d * 8.0 + t) / 8.0;
        d = abs(d);
        d = pow(0.01 / d, 1.2);
        finalCol += col * d;
    }
    float distortion = length(finalCol.rgb) * 0.05;
    vec4 sampledColor = texture(samp, tc + distortion);
    float shiftAmt = pingPong(time_f, 5.0);
    vec3 shiftedColor = hueShift(sampledColor.rgb + (finalCol * 0.5), shiftAmt);
    color = vec4(shiftedColor, 1.0);
}

```

Shader #486 - gem rainbow metal

File: gem_rainbow_metal.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. peak absolute sample value in current audio buffer RMS energy of current audio buffer exponentially smoothed amplitude for gradual transitions

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec3 spectrum(float t) {
    return vec3(0.5 + 0.5 * cos(6.28318 * (t + vec3(0.0, 0.33, 0.67))));
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float r = length(uv);
    float angle = atan(uv.y, uv.x);
    float ripple = sin(angle * 10.0 + time_f) * 0.03;
    ripple += sin(angle * 25.0 - time_f * 2.0) * 0.01;
    float wave = sin(r * 20.0 - time_f * 4.0 + ripple * 10.0);
    float shift = ripple * 0.5 + wave * 0.01;
    float r_chan = texture(samp, tc + vec2(shift, 0.0)).r;
    float g_chan = texture(samp, tc).g;
    float b_chan = texture(samp, tc - vec2(shift, 0.0)).b;
    vec3 baseTex = vec3(r_chan, g_chan, b_chan);
    vec3 rainbow = spectrum(r - time_f * 0.5 + ripple);
}

```

```

float glowMask = smoothstep(0.5, 1.0, wave);
float center = exp(-r * 6.0);
vec3 coreGlow = vec3(1.0, 0.98, 0.9) * center * 1.5;
vec3 finalColor = mix(baseTex, rainbow, glowMask * 0.35);
finalColor += coreGlow;
finalColor += (wave * ripple * 2.0);
color = vec4(finalColor, 1.0);
}

```

Shader #487 - gem rainbow metal audio

File: gem_rainbow_metal_audio.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. peak absolute sample value in current audio buffer RMS energy of current audio buffer exponentially smoothed amplitude for gradual transitions

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec3 spectrum(float t) {
    return vec3(0.5 + 0.5 * cos(6.28318 * (t + vec3(0.0, 0.33, 0.67))));
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float r = length(uv);
    float angle = atan(uv.y, uv.x);
    float ripple = sin(angle * 10.0 + time_f) * (0.03 + amp_mid * 0.08);
    ripple += sin(angle * 25.0 - time_f * 2.0) * (0.01 + amp_high * 0.04);
    float wave = sin(r * 20.0 - time_f * 4.0 - (amp_low * 6.0) + ripple * 10.0);
    float shift = ripple * 0.5 + wave * 0.01 + (amp_high * 0.015);
    float r_chan = texture(samp, tc + vec2(shift, 0.0)).r;
    float g_chan = texture(samp, tc).g;
    float b_chan = texture(samp, tc - vec2(shift, 0.0)).b;
    vec3 baseTex = vec3(r_chan, g_chan, b_chan);
    vec3 rainbow = spectrum(r - time_f * 0.5 - (amp_smooth * 3.0) + ripple);
    float glowMask = smoothstep(0.5, 1.0, wave);
    float center = exp(-r * 6.0) + (exp(-r * 2.5) * amp_low * 0.6);
    vec3 coreGlow = vec3(1.0, 0.98, 0.9) * center * (1.5 + amp_peak * 1.5);
    vec3 finalColor = mix(baseTex, rainbow, glowMask * (0.35 + amp_peak * 0.5));
    finalColor += coreGlow;
    finalColor += (wave * ripple * (2.0 + amp_rms * 8.0));
    color = vec4(finalColor, 1.0);
}

```

Shader #502 - gkalei

File: gkalei.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

float noise(vec2 p) {
    return fract(sin(dot(p, vec2(12.9898, 78.233))) * 43758.5453);
}

float smoothNoise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    float a = noise(i);
    float b = noise(i + vec2(1.0, 0.0));
    float c = noise(i + vec2(0.0, 1.0));
    float d = noise(i + vec2(1.0, 1.0));
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(a, b, u.x) + (c - a) * u.y * (1.0 - u.x) + (d - b) * u.x * u.y;
}

float fractalNoise(vec2 p) {
    float value = 0.0;
    float amplitude = 0.5;
    for (int i = 0; i < 4; i++) {
        value += amplitude * smoothNoise(p);
        p *= 2.0;
        amplitude *= 0.5;
    }
    return value;
}

vec3 hsvToRgb(float h, float s, float v) {
    vec3 p = abs(fract(vec3(5.0, 3.0, 1.0) + h * 6.0) * 2.0 - 1.0);
    return v * mix(vec3(1.0), clamp((p - 1.0 + s), 0.0, 1.0), s);
}

void main(void) {
    vec2 uv = tc * 2.0 - vec2(1.0);
    uv *= iResolution.x / iResolution.y;
    uv *= 2.0;
    float displacement = fractalNoise(uv * 5.0 + time_f * 0.5);
    vec2 dispCoords = tc + vec2(displacement * 0.05, displacement * 0.1);
    vec4 texColor = texture(samp, dispCoords);
    float radius = length(uv);
    float angle = atan(uv.y, uv.x) + time_f * 1.5;
    float hue = mod(angle / (2.0 * 3.14159) + time_f * 0.3, 1.0);
    vec3 prismColor = hsvToRgb(hue, 1.0, 1.0 - pingPong(radius * 0.8 + time_f * 0.5, 1.0));
    color = vec4(sin(prismColor * time_f) * texColor.rgb, texColor.a);
    color.a = 1.0;
}
```

Shader #539 - gradient

File: gradient.glsl

Applies an iterative, time-animated effect that evolves the image over time. Discards pure black pixels for transparency. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in float restore_black_value;
void main(void)
{
    if(restore_black_value == 1.0 && texture(samp, tc) == vec4(0, 0, 0, 1))
        discard;
    color = texture(samp, tc);
    color[0] += (color[0]*alpha_r);
    color[1] += (color[1]*alpha_g);
    color[2] += (color[2]*alpha_b);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        color[i] += time_f * tc[0] * alpha_r;
        int_color[i] = int(255 * color[i]);
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #540 - gradient color

File: gradient_color.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
```

```

out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    for(int i = 0; i < 3; ++i) {
        color[i] += cos(color[i] * timeval);
        color[i] = color[i]+tc[0];
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #541 - gradient position

File: gradient_position.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;

```

```

void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = (0.321*timeval)+(color[0]*(0.1 * vpos.x));
    color[1] = (color[1]*(0.1 * vpos.y));
    color[2] = (0.4212*timeval)+(color[2]*(0.1 * vpos.x+vpos.y));
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #542 - gradient shift

File: gradient_shift.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
vec3 hue(float t){
    return clamp(vec3(abs(t*6.0-3.0)-1.0, 2.0-abs(t*6.0-2.0), 2.0-abs(t*6.0-4.0)),0.0,1.0);
}
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
void main(void){
    vec2 m = iMouse.z > 0.0 ? iMouse.xy / iResolution.xy : vec2(0.5);
    float a = time_f*0.6;
    vec2 dir = normalize(vec2(cos(a), sin(a)));
    float span = 0.2 + 0.15*sin(time_f*0.7);
    vec2 startP = m - dir*span;
    vec2 endP = m + dir*span;
    vec2 ab = endP - startP;
    float L = max(length(ab), 1e-4);
    vec2 u = ab / L;
    float t = clamp(dot(tc - startP, u) / L, 0.0, 1.0);
    float edge = smoothstep(0.0, 0.08, t) * (1.0 - smoothstep(0.92, 1.0, t));
    float w = (t - 0.5);
    float pull = (0.02 + 0.12*uamp) * (0.5 + 0.5*sin(time_f*1.1));

```



```

    vec2 uv = tc + u * w * pull * (0.5 + edge);
    vec4 base = texture(samp, uv);
    vec3 c0 = hue(fract(time_f*0.12));
    vec3 c1 = hue(fract(time_f*0.12 + 0.5));
    vec3 grad = mix(c0, c1, t);
    float k = clamp(amp, 0.0, 1.0);
    vec3 mixed = mix(base.rgb, base.rgb * grad, k);
    color = vec4(sin(mixed * pingPong(time_f, 10.0)), base.a);
}

```

Shader #543 - grainbow

File: grainbow.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float dispersion = 0.02;
    vec2 dir = normalize(uv);
    vec2 offset_r = tc + dir * dispersion;
    vec2 offset_g = tc;
    vec2 offset_b = tc - dir * dispersion;
    float r = texture(samp, offset_r).r;
    float g = texture(samp, offset_g).g;
    float b = texture(samp, offset_b).b;
    vec3 prism_color = vec3(r, g, b);
    float t = fract((angle / (2.0 * 3.14159)) + time_f * 0.1);
    vec3 rainbow_color = rainbow(t);
    float rainbow_factor = 0.5;
    vec3 final_color = mix(prism_color, rainbow_color, rainbow_factor);
    color = vec4(final_color, 1.0);
}

```

Shader #544 - grayscale

File: grayscale.glsl

Applies a time-animated visual transformation to the input texture. Computes luminance for brightness-based effects.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec4 texColor = texture(samp, tc);
    float gray = dot(texColor.rgb, vec3(0.299, 0.587, 0.114));
    color = vec4(vec3(gray), texColor.a);
}
```

Shader #548 - green gradient

File: green_gradient.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations. Assign a random value based on the integer coord

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec3 st = gl_FragCoord.xyz;
    for(int i = 0; i < 3; ++i) {
        float f = 2560/st[i];
        color[i] = color[i]*f;
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
```

```

        int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #549 - green gradient strobe

File: green_gradient_strobe.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations. Assign a random value based on the integer coord

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec3 st = gl_FragCoord.xyz;
    for(int i = 0; i < 3; ++i) {
        float f = 255/st[i];
        color[i] = color[i]*f*alpha_r+alpha_g+alpha_b;
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #577 - huei af2

File: huei_af2.glsl

Performs edge detection on the input image, highlighting boundaries and contours. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Sobel/Laplacian convolution kernel for edge detection - Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;

float pingPong(float x, float l){float m=mod(x,l*2.0);return m<=l?m:l*2.0-m;}
float hash(vec2 p){return fract(sin(dot(p,vec2(127.1,311.7))))*43758.5453);}
float noise(vec2 p){vec2 i=floor(p),f=fract(p);vec2 u=f*f*(3.0-2.0*f);float a=hash(i),b=hash(i+vec2(1,0)
    ↪ ),c=hash(i+vec2(0,1)),d=hash(i+vec2(1,1));return mix(mix(a,b,u.x),mix(c,d,u.x),u.y);}
vec3 rainbow(float t){t=fract(t);float r=abs(t*6.0-3.0)-1.0;float g=2.0-abs(t*6.0-2.0);float b=2.0-abs(t
    ↪ *6.0-4.0);return clamp(vec3(r,g,b),0.0,1.0);}
vec2 kaleido(vec2 uv, vec2 c, float seg, vec2 ar){
    vec2 p=(uv-c)*ar;
    float ang=atan(p.y,p.x);
    float ra=length(p);
    float a=6.2831853/seg;
    ang=mod(ang,a);
    ang=abs(ang-a*0.5);
    p=vec2(cos(ang),sin(ang))*ra;
    return c+p/ar;
}
vec2 vortex(vec2 uv, vec2 c, float t){
    vec2 d=uv-c;
    float r=length(d);
    float a=atan(d.y,d.x);
    a+=sin(t*0.9+r*12.0)*0.6+0.3*sin(r*24.0-t*1.7);
    float z=0.15*sin(t*0.6*pow(r,0.6)*10.0);
    r=clamp(r*(1.0-z),0.0,1.0);
    return c+vec2(cos(a),sin(a))*r;
}
vec4 blur9(sampler2D img, vec2 uv, vec2 res){
    vec2 ts=1.0/res;
    vec4 s=vec4(0.0);
    s+=texture(img,uv+ts*vec2(-1,-1));
    s+=texture(img,uv+ts*vec2( 0,-1));
    s+=texture(img,uv+ts*vec2( 1,-1));
    s+=texture(img,uv+ts*vec2(-1, 0));
    s+=texture(img,uv+ts*vec2( 0, 0));
    s+=texture(img,uv+ts*vec2( 1, 0));
    s+=texture(img,uv+ts*vec2(-1, 1));
    s+=texture(img,uv+ts*vec2( 0, 1));
    s+=texture(img,uv+ts*vec2( 1, 1));
    return s/9.0;
}
float luma(vec3 c){return dot(c,vec3(0.299,0.587,0.114));}
void main(){
```

```

vec2 ar=vec2(1.0,iResolution.y/iResolution.x);
vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
vec3 orig=texture(samp, tc).rgb;
vec2 uv=tc;
uv=kaleido(uv,m,6.0,ar);
uv=vortex(uv,m,time_f);
float r1=0.003+0.003*sin(time_f*1.1);
vec2 dir=normalize(uv-m+1e-6);
vec2 tw=dir*(0.004*(noise(uv*vec2(iResolution.x/iResolution.y,1.0)*5.0+time_f*0.7)-0.5));
vec4 b=blur9(samp,uv,iResolution);
vec4 cR=texture(samp, uv+dir*r1+tw);
vec4 cG=texture(samp, uv+tw);
vec4 cB=texture(samp, uv-dir*r1+tw);
vec3 texRGB=mix(b.rgb, vec3(cR.r,cG.g,cB.b), 0.85);
vec2 p=(uv-m)*ar;
float rad=length(p);
float ang=atan(p.y,p.x);
float beat=0.5+0.5*sin(time_f*2.1);
vec3 base=rainbow(ang/6.2831853+time_f*0.12);
vec3 glow=base*smoothstep(0.85,0.0,rad)*beat;
float strobe=0.5+0.5*sin(time_f*10.0+rad*40.0);
float w=0.28+0.22*strobe;
vec3 fractal=mix(texRGB, texRGB*rainbow(time_f*0.08+rad*2.0), w)+glow*0.7;
float L=luma(orig);
float gx=abs(dFdx(L));
float gy=abs(dFdy(L));
float edge=smoothstep(0.02,0.15,gx+gy);
float radial=smoothstep(0.05,0.65,rad);
float band=1.0-smoothstep(0.015,0.035,abs(fract(ang*3.0/3.14159+time_f*0.08)-0.5));
float n=0.35+0.65*noise(tc*vec2(iResolution.x/iResolution.y,1.0)*3.0+time_f*0.4);
float strength=clamp(0.55*radial+0.25*strobe,0.0,1.0);
strength*=mix(1.0,0.4,edge);
strength*=mix(1.0,0.6,band);
strength=mix(strength,strength*n,0.6);
vec3 screenMix=2.5-(1.0-fractal)*(1.0-orig);
vec3 outRGB=mix(orig, screenMix, strength);
float vign=1.0-smoothstep(0.8,1.08,length((tc-m)*ar));
color=vec4(outRGB*vign,1.0);
}

```

Shader #621 - list rainbow enter

File: list_rainbow_enter.glsl

Applies a time-animated visual transformation to the input texture. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec2 iMouse;
vec3 rainbow(float t){
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;

```

```

    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

void main(){
    vec2 mouse = iMouse / iResolution;
    mouse.y = 1.0 - mouse.y;
    vec2 uv = tc - mouse;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + time_f * 8.0;
    float radius = length(uv);
    vec3 rc = rainbow(angle / (2.0 * 3.14159));
    vec4 tex = texture(samp, tc);
    float falloff = smoothstep(0.5, 0.0, radius);
    vec3 outc = mix(tex.rgb, rc, falloff);
    color = vec4(outc, tex.a);
}

```

Shader #637 - mirdmdrcolor

File: mirdmdrcolor.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = uv * 2.0 - 1.0;
    float angle1 = time_f * 0.5;
    float angle2 = -time_f * 0.5;
    if (centeredUV.x >= 0.0) {
        mat2 rotation1 = mat2(cos(angle1), -sin(angle1), sin(angle1), cos(angle1));
        centeredUV = rotation1 * centeredUV;
    } else {
        mat2 rotation2 = mat2(cos(angle2), -sin(angle2), sin(angle2), cos(angle2));
        centeredUV = rotation2 * centeredUV;
    }
    centeredUV = abs(centeredUV);
    vec2 mirroredUV = mod(centeredUV, 1.0);
    vec4 texColor = texture(samp, mirroredUV);
    float pulse = 0.5 + 0.5 * sin(time_f * 3.0);
    vec3 pulsatingColor = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 10.0),
        0.5 + 0.5 * cos(time_f + uv.y * 10.0),
        0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 10.0)
    ) * pulse;
    color = vec4(texColor.rgb * 0.7 + pulsatingColor * 0.3, texColor.a);
}

```

Shader #719 - moving gradient

File: moving_gradient.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] += tc[0];
    color[1] += tc[1];
    color[2] += tc[0]+tc[1];
    color[0] += 0.1*timeval;
    color[1] += 0.3*timeval;
    color[2] += 0.7*timeval;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #730 - o1 hue

File: o1-hue.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Computes luminance for brightness-based effects. Applies matrix transformations.

Technique: Multi-tap convolution blur (5 texture samples) - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
const int SEGMENTS = 6;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 adjustHue(vec4 c, float angle) {
    float U = cos(angle);
    float W = sin(angle);
    mat3 rotationMatrix = mat3(
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114
    ) + mat3(
        0.701, -0.587, -0.114,
        -0.299, 0.413, -0.114,
        -0.300, -0.588, 0.886
    ) * U
    + mat3(
        0.168, 0.330, -0.497,
        -0.328, 0.035, 0.292,
        1.250, -1.050, -0.203
    ) * W;
    return vec4(rotationMatrix * c.rgb, c.a);
}
void main() {
    vec2 uv = (tc - 0.5) * iResolution / min(iResolution.x, iResolution.y);
    float r = length(uv);
    float angle = atan(uv.y, uv.x);
    float swirlStrength = 2.5;
    float swirl = time_f * 0.5;
    angle += swirlStrength * sin(swirl + r * 4.0);
    float segmentAngle = 2.0 * 3.14159265359 / float(SEGMENTS);
    angle = mod(angle, segmentAngle);
    angle = abs(angle - segmentAngle * 0.5);
    vec2 kaleidoUV = vec2(cos(angle), sin(angle)) * r;
    float ripple = sin(r * 12.0 - pingPong(time_f, 10.0) * 10.0) * exp(-r * 4.0);
    kaleidoUV += ripple * 0.01;
    vec2 scale = vec2(1.0) / (iResolution / min(iResolution.x, iResolution.y));
    vec2 sampleTC = kaleidoUV * scale + 0.5;
    float offsetAmount = 0.003 * sin(time_f * 0.5);
    vec4 col = texture(samp, sampleTC);
    col += texture(samp, sampleTC + vec2(offsetAmount, 0.0));
    col += texture(samp, sampleTC + vec2(-offsetAmount, offsetAmount));
    col += texture(samp, sampleTC + vec2(offsetAmount * 0.5, -offsetAmount));
    col /= 4.0;
    float hueSpeed = 2.0;
    float hueShift = (time_f * hueSpeed + ripple * 2.0);
    color = adjustHue(col, hueShift);
    float saturationBoost = 1.5;
    float brightnessBoost = 1.1;
    vec3 hsv = color.rgb;
    float avg = (hsv.r + hsv.g + hsv.b) / 3.0;
    hsv = mix(vec3(avg), hsv, saturationBoost);
    hsv *= brightnessBoost;
    color.rgb = hsv;
    color = mix(clamp(color, 0.0, 1.0), texture(samp, tc), 0.5);
    color = sin(color * pingPong(time_f, 10.0) + 2.0);
    color.a = 1.0;
}

```

Shader #758 - picolor

File: picolor.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    color += color / sin(3.1415926535/alpha);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #766 - plasma rainbow

File: plasma_rainbow.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
float PI = 3.1415926535897932384626433832795;
vec4 xor_RGB(vec4 icolor, vec4 isource) {
```

```

    ivec4 isource = ivec4(isourcex * 255);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
return icolor;
}

float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}

vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)), dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}

vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}

vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

void main() {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float plasma = 0.0;
    plasma += sin((uv.x + time_f) * 5.0);
    plasma += sin((uv.y + time_f) * 5.0);
    plasma += sin((uv.x + uv.y + time_f) * 5.0);
    plasma += cos(length(uv + time_f) * 10.0);
    plasma *= 0.25;
    vec3 col;
    col.r = cos(plasma * PI + time_f * 0.2) * 0.5 + 0.5;
    col.g = sin(plasma * PI + time_f * 0.2) * 0.5 + 0.5;
    col.b = sin(plasma * PI + time_f * 0.4) * 0.5 + 0.5;
    vec4 plasma_color = vec4(col, 1.0);
    vec4 tex_color = texture(samp, tc);
    vec4 xor_color = xor_RGB(plasma_color, tex_color);
    vec2 random_direction = smoothRandom2(time_f) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_f * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float angle = atan(spiral_uv.y + sin(uv.x * 10.0 + time_f * 2.0) * 0.1, spiral_uv.x) + time_f * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * PI));
    vec3 blended_color = mix(xor_color.rgb, rainbow_color, 0.5);
    float time_t = mod(time_f, 30.0);
    color = vec4(sin(blended_color * time_t), tex_color.a);
}

```

Shader #789 - psych wave rainbow

File: psych_wave_rainbow.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {
    ivec4 isource = ivec4(isourcex * 255);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)), dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 normPos = (gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0;
    float dist = length(normPos);
    float phase = sin(dist * 10.0 - time_f * 4.0);
    vec2 tcAdjusted = tc + (normPos * 0.305 * phase);
    vec4 texColor = texture(samp, tcAdjusted);
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float time = time_f * 0.5;
    float sine1 = sin(uv.x * 10.0 + time) * cos(uv.y * 10.0 + time);
    float sine2 = sin(uv.y * 15.0 - time) * cos(uv.x * 15.0 - time);
    float sine3 = sin(sqrt(uv.x * uv.x + uv.y * uv.y) * 20.0 + time);
```

```

float pattern = sine1 + sine2 + sine3;
float colorIntensity = pattern * 0.5 + 0.5;
vec3 psychedelicColor = vec3(
    sin(colorIntensity * 6.28318 + 0.0) * 0.5 + 0.5,
    sin(colorIntensity * 6.28318 + 2.09439) * 0.5 + 0.5,
    sin(colorIntensity * 6.28318 + 4.18879) * 0.5 + 0.5
);
vec4 finalColor = vec4(psychedelicColor, 1.0);
vec4 xorColor = xor_RGB(finalColor, texColor);
vec2 uv2 = tc * 2.0 - 1.0;
uv2.y *= iResolution.y / iResolution.x;
float wave = sin(uv2.x * 10.0 + time_f * 2.0) * 0.1;
vec2 random_direction = smoothRandom2(time_f) * 0.5;
float expand = 0.5 + 0.5 * sin(time_f * 2.0);
vec2 spiral_uv = uv2 * expand + random_direction;
float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_f * 2.0;
vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
vec4 original_color = texture(samp, tc);
vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
float time_t = mod(time_f, 30.0);
vec4 finalBlendedColor = vec4(sin(blended_color * time_t), original_color.a);
float speed = 5.0;
float amplitude = 0.03;
float wavelength = 10.0;
float ripple = sin(tc.x * wavelength + time_f * speed) * amplitude;
ripple += sin(tc.y * wavelength + time_f * speed) * amplitude;
vec2 rippleTC = tc + vec2(ripple, ripple);
vec4 rippleColor = texture(samp, rippleTC);
vec4 combinedColor = mix(xorColor, finalBlendedColor, 0.5);
color = mix(combinedColor, rippleColor, 0.5);
}

```

Shader #804 - rainbow touch

File: rainbow-touch.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(vec2 uv, float offset) {
    float rainbowFactor = sin(offset + uv.x * 10.0) * 0.5 + 0.5;
    return vec3(
        sin(rainbowFactor * 3.0 + 0.0),
        sin(rainbowFactor * 3.0 + 2.0),
        sin(rainbowFactor * 3.0 + 4.0)
    );
}
void main(void) {
    vec4 texColor = texture(samp, tc);
    float phase = mod(time_f, 6.0);
    float t = fract(time_f / 2.0);
    vec2 uv = tc;
    vec3 rainbowColor;

```

```

float angle = mix(0.0, 6.28318, t);
uv += vec2(sin(angle + length(uv) * 10.0), cos(angle + length(uv) * 10.0)) * 0.1;
rainbowColor = rainbow(uv, time_f);
color = vec4(texColor.rgb + rainbowColor * 0.5, texColor.a);
}

```

Shader #805 - rainbow.banding

File: rainbow.banding.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Scanline overlay using modular y-coordinate - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbowBanding(vec2 uv) {
    float banding = sin(uv.y * 200.0 + time_f * 5.0);
    vec3 col;
    col.r = sin(banding + 0.0) * 0.05 + 0.95;
    col.g = sin(banding + 2.0) * 0.05 + 0.95;
    col.b = sin(banding + 4.0) * 0.05 + 0.95;
    return col;
}
vec3 compositeEffect(vec2 uv) {
    vec3 col = texture(samp, uv).rgb;
    vec3 banding = rainbowBanding(uv);
    col *= banding;
    float noise = fract(sin(dot(uv.xy, vec2(12.9898, 78.233)))) * 43758.5453;
    col += noise * 0.02;
    float scanline = sin(uv.y * iResolution.y * 3.0) * 0.1;
    col -= scanline;
    return col;
}
void main(void) {
    vec2 uv = tc;
    vec3 col = compositeEffect(uv);
    color = vec4(col, 1.0);
}

```

Shader #807 - rainbow bright

File: rainbow_bright.glsl

Applies a time-animated visual transformation to the input texture. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Pixels darker than this become rainbow Speed of the color shift Size of the rainbow waves

Technique: Indexed color palette look-up table - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core

```

```

in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float DARKNESS_THRESHOLD = 0.2;
const float RAINBOW_SPEED = 1.0;
const float RAINBOW_SCALE = 3.0;
const float BLEND_SOFTNESS = 0.1;
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.00, 0.33, 0.67);
    return a + b * cos(6.28318 * (c * t + d));
}
float getLuma(vec3 c) {
    return dot(c, vec3(0.299, 0.587, 0.114));
}
void main(void) {
    vec4 texColor = texture(samp, tc);
    float brightness = getLuma(texColor.rgb);
    float mask = 1.0 - smoothstep(DARKNESS_THRESHOLD, DARKNESS_THRESHOLD + BLEND_SOFTNESS, brightness);
    vec3 rainbow = palette((tc.x + tc.y * 0.5) * RAINBOW_SCALE + iTime * RAINBOW_SPEED);
    vec3 finalRGB = mix(texColor.rgb, rainbow, mask);
    color = vec4(finalRGB, 1.0);
}

```

Shader #808 - rainbow cd

File: rainbow_cd.glsl

Applies geometric transformations including rotation and scaling to distort the image.

Technique: 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending -
Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),

```

```

        dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}

vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}

vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float pingPongTime = pingPong(time_f, 9.0);
    float rotations = floor(pingPongTime / 3.0);
    float direction = mod(rotations, 2.0) == 0.0 ? 1.0 : -1.0;
    float localTime = mod(pingPongTime, 3.0);
    float rotationAngle = localTime * 2.0 * 3.14159 * direction;
    float cosAngle = cos(rotationAngle);
    float sinAngle = sin(rotationAngle);
    vec2 rotatedUV = vec2(
        cosAngle * uv.x - sinAngle * uv.y,
        sinAngle * uv.x + cosAngle * uv.y
    );
    uv = rotatedUV;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_t), original_color.a);
}

```

Shader #809 - rainbow cd curtain

File: rainbow_cd_curtain.glsl

Applies geometric transformations including rotation and scaling to distort the image.

Technique: 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending -
Driven by: time, resolution

```

#version 330
in vec2 tc;

```

```

out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float value_alpha_r;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t, float seed) {
    t = fract(t + seed);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float pingPongTime = pingPong(time_f, 9.0);
    float rotations = floor(pingPongTime / 3.0);
    float direction = mod(rotations, 2.0) == 0.0 ? 1.0 : -1.0;
    float localTime = mod(pingPongTime, 3.0);
    float rotationAngle = localTime * 2.0 * 3.14159 * direction;
    float cosAngle = cos(rotationAngle);
    float sinAngle = sin(rotationAngle);
    vec2 rotatedUV = vec2(
        cosAngle * uv.x - sinAngle * uv.y,
        sinAngle * uv.x + cosAngle * uv.y
    );
    uv = rotatedUV;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_t * 2.0;
    float curtain_wave = sin(uv.y * 10.0 + time_f * 2.0) * 0.1;
    vec3 rainbow_color = rainbow((angle / (2.0 * 3.14159)) + curtain_wave, value_alpha_r);
    vec4 original_color = texture(samp, (rotatedUV + 1.0) / 2.0);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_t), original_color.a);
}

```


Shader #810 - rainbow cd spin

File: rainbow_cd_spin.glsl

Applies geometric transformations including rotation and scaling to distort the image.

Technique: 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending -

Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float value_alpha_r;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t, float seed) {
    t = fract(t + seed);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float pingPongTime = pingPong(time_f, 9.0);
    float rotations = floor(pingPongTime / 3.0);
    float direction = mod(rotations, 2.0) == 0.0 ? 1.0 : -1.0;
    float localTime = mod(pingPongTime, 3.0);
    float rotationAngle = localTime * 2.0 * 3.14159 * direction;
    float cosAngle = cos(rotationAngle);
    float sinAngle = sin(rotationAngle);
    vec2 rotatedUV = vec2(
        cosAngle * uv.x - sinAngle * uv.y,
        sinAngle * uv.x + cosAngle * uv.y
    );
    uv = rotatedUV;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
```

```

float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_t * 2.0;
vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159), value_alpha_r);
vec4 original_color = texture(samp, (rotatedUV + 1.0) / 2.0);
vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
color = vec4(sin(blended_color * time_t), original_color.a);
}

```

Shader #811 - rainbow cd strobe

File: rainbow_cd_strobe.glsl

Applies geometric transformations including rotation and scaling to distort the image.

Technique: 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending -

Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float value_alpha_r;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
        dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t, float seed) {
    t = fract(t + seed);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float pingPongTime = pingPong(time_f, 9.0);
    float rotations = floor(pingPongTime / 3.0);
    float direction = mod(rotations, 2.0) == 0.0 ? 1.0 : -1.0;
    float localTime = mod(pingPongTime, 3.0);
    float rotationAngle = localTime * 2.0 * 3.14159 * direction;
    float cosAngle = cos(rotationAngle);
    float sinAngle = sin(rotationAngle);
}

```

```

    vec2 rotatedUV = vec2(
        cosAngle * uv.x - sinAngle * uv.y,
        sinAngle * uv.x + cosAngle * uv.y
    );
    uv = rotatedUV;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159), value_alpha_r);
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_t), original_color.a);
}

```

Shader #812 - rainbow color1

File: rainbow_color1.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + time_f * 20.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(tan(blended_color * time_f), original_color.a);
}

```

Shader #813 - rainbow color2

File: rainbow_color2.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;

```

```

out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + time_f * 20.0;
    vec3 rainbow_color = rainbow(angle * time_f / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(cos(blended_color * mod(time_f, 25.0)), original_color.a);
}

```

Shader #814 - rainbow color3

File: rainbow_color3.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + time_f * 20.0;
    vec3 rainbow_color = rainbow(angle * time_f / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, sin(rainbow_color * time_f), 0.5);
    color = vec4(blended_color, original_color.a);
}

```

Shader #815 - rainbow color swirl

File: rainbow_color_swirl.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float waveFreq = 10.0 + amp_mid * 15.0;
    float wave = sin(uv.x * waveFreq + time_f * 2.0) * (0.1 + amp_low * 0.2);
    float angle = atan(uv.y + wave, uv.x) + time_f * (2.0 + amp_low * 3.0);
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_f), original_color.a);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}
```

Shader #816 - rainbow expand

File: rainbow_expand.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
```

```

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float wave = sin(uv.x * 10.0 + time_f * 2.0) * 0.1;
    float expand = 0.5 + 0.5 * sin(time_f * 2.0);
    vec2 spiral_uv = uv * expand + vec2(cos(time_f), sin(time_f)) * 0.2;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_f * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    float time_t = mod(time_f, 30.0);
    color = vec4(sin(blended_color * time_t), original_color.a);
}

```

Shader #822 - rainbow ink

File: rainbow_ink.glsl

Generates procedural noise patterns combined with sinusoidal distortion. color = vec4(blended_color, original_color.a);

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float random_seed;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float noise(vec2 p) {
    return texture(samp, p).r;
}
vec2 swirl(vec2 p, float t, float seed) {
    float radius = length(p);
    float angle = atan(p.y, p.x) + sin(t + radius * 10.0 + noise(p * 10.0 + seed) * 10.0) * 0.5;
    return vec2(cos(angle), sin(angle)) * radius;
}
void main(void) {
    float mod_time = mod(time_f, 25.0);
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    uv = swirl(uv, mod_time * 0.1, random_seed);
    uv *= 0.5;
    float n = noise(uv * 0.5 + mod_time * 0.05);
    vec3 rainbow_color = rainbow(n + mod_time * 0.1);
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(rainbow_color, 1.0);
}

```

Shader #823 - rainbow ink t

File: rainbow_ink_t.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float random_seed;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float noise(vec2 p) {
    return texture(samp, p).r;
}
vec2 swirl(vec2 p, float t, float seed) {
    float radius = length(p);
    float angle = atan(p.y, p.x) + sin(t + radius * 10.0 + noise(p * 10.0 + seed) * 10.0) * 0.5;
    return vec2(cos(angle), sin(angle)) * radius;
}
void main(void) {
    float mod_time = mod(time_f, 25.0);
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    uv = swirl(uv, mod_time * 0.1, random_seed);
    uv *= 0.5;
    float n = noise(uv * 0.5 + mod_time * 0.05);
    vec3 rainbow_color = rainbow(n + mod_time * 0.1);
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(blended_color, original_color.a);
}
```

Shader #824 - rainbow left

File: rainbow_left.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
```

```

vec3 hsv2rgb(vec3 c) {
    c.x = fract(c.x);      vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

void main() {
    vec2 uv = tc;
    float timeScale = 0.2;
    float spatialScale = 3.0;
    float hue = uv.x * spatialScale + time_f * timeScale;
    vec3 colorHue = hsv2rgb(vec3(hue, 1.0, 1.0));
    vec4 texColor = texture(samp, uv);
    vec4 finalColor = vec4(texColor.rgb * colorHue, texColor.a);
    color = finalColor;
}

```

Shader #828 - rainbow pi01

File: rainbow_pi01.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}

vec3 rainbow(vec2 uv, float offset) {
    float rainbowFactor = sin(offset + uv.x * 10.0) * 0.5 + 0.5;
    return vec3(
        sin(rainbowFactor * 3.0 + 0.0),
        sin(rainbowFactor * 3.0 + 2.0),
        sin(rainbowFactor * 3.0 + 4.0)
    );
}

void main(void) {
    vec4 texColor = texture(samp, tc);
    float phase = mod(time_f, 6.0);
    float t = fract(time_f / 2.0);
    vec2 uv = tc;
    vec3 rainbowColor;
    float angle = mix(0.0, 6.28318, t);
    uv += vec2(sin(angle + length(uv) * 10.0), cos(angle + length(uv) * 10.0)) * 0.1;
    rainbowColor = rainbow(uv, time_f);
    color = vec4(texColor.rgb + rainbowColor * 0.5, texColor.a);
    color = mix(color, vec4(rainbowColor, 1.0) * 0.5+pingPong((time_f * PI), 1.2), 0.5);
}

```


Shader #829 - rainbow pi neoin

File: rainbow_pi_neoin.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec3 rainbow(vec2 uv, float offset) {
    float rainbowFactor = sin(offset + uv.x * 10.0) * 0.5 + 0.5;
    return vec3(
        sin(rainbowFactor * 3.0 + 0.0),
        sin(rainbowFactor * 3.0 + 2.0),
        sin(rainbowFactor * 3.0 + 4.0)
    );
}
void main(void) {
    vec4 texColor = texture(samp, tc);
    float phase = mod(time_f, 6.0);
    float t = fract(time_f / 2.0);
    vec2 uv = tc;
    vec3 rainbowColor;
    float angle = mix(0.0, 6.28318, t);
    angle = angle * time_f * PI;
    uv += vec2(sin(angle + length(uv) * 10.0), cos(angle + length(uv) * 10.0)) * 0.1;
    rainbowColor = rainbow(uv, pingPong(sin(time_f), 25.0));
    color = vec4(texColor.rgb + rainbowColor * 0.5, texColor.a);
    color = mix(color, 0.5+vec4(rainbowColor, 1.0) * 0.5, 0.5);
}
```

Shader #830 - rainbow prisim

File: rainbow_prisim.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
vec3 hsv2rgb(vec3 c){
```

```

    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
float pingPong(float x,float length){
    float m=mod(x,length*2.0);
    return m<=length?m:length*2.0-m;
}
void main(void){
    float A=clamp(amp,0.0,1.0);
    float U=clamp(uamp,0.0,1.0);
    float time_z=pingPong(time_f,4.0)+0.5;
    vec2 uv=tc;
    vec2 center=(iMouse.z>0.5||iMouse.w>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 normPos=(uv-center)*vec2(iResolution.x/iResolution.y,1.0);
    float dist=length(normPos);
    float phase=sin(dist*10.0-time_f*4.0);
    float phaseGain=mix(0.6,1.4,A);
    vec2 tcAdjusted=uv+(normPos*0.305*phase*phaseGain);
    float dispersionScale=0.02*(0.5+U);
    vec2 dispersionOffset=normPos*dist*dispersionScale;
    vec2 tcAdjustedR=tcAdjusted-dispersionOffset;
    vec2 tcAdjustedG=tcAdjusted;
    vec2 tcAdjustedB=tcAdjusted+dispersionOffset;
    float r=texture(samp,tcAdjustedR).r;
    float g=texture(samp,tcAdjustedG).g;
    float b=texture(samp,tcAdjustedB).b;
    vec3 texColor=vec3(r,g,b);
    float angle=atan(normPos.y,normPos.x)+time_f;
    float hue=mod(angle/(2.0*3.14159265),1.0);
    vec3 rainbowColor=hsv2rgb(vec3(hue,1.0,1.0));
    vec3 modColor=mix(texColor,texColor*rainbowColor,0.4+0.6*A);
    float time_t=pingPong(time_f,8.0)+2.0+time_z*0.1*(0.5+U);
    color=vec4(sin(modColor*time_t),1.0);
}

```

Shader #831 - rainbow rip prism

File: rainbow_rip_prism.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
- Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

```

```

}
void main(void) {
    float time_z = pingPong(time_f, 4.0) + 0.5;
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    vec2 normPos = uv * 2.0 - 1.0 * time_z;
    float dist = length(normPos);
    float phase = sin(dist * 10.0 - time_f * 4.0);
    vec2 tcAdjusted = tc + (normPos * 0.305 * phase);
    float dispersionScale = 0.02;
    vec2 dispersionOffset = normPos * dist * dispersionScale;
    vec2 tcAdjustedR = tcAdjusted + dispersionOffset * (-1.0);
    vec2 tcAdjustedG = tcAdjusted;
    vec2 tcAdjustedB = tcAdjusted + dispersionOffset * 1.0;
    float r = texture(samp, tcAdjustedR).r;
    float g = texture(samp, tcAdjustedG).g;
    float b = texture(samp, tcAdjustedB).b;
    vec3 texColor = vec3(r, g, b);
    float angle = atan(normPos.y, normPos.x);
    float spiral = angle + time_f * 1.0;
    float hue = mod(spiral / (2.0 * 3.14159265), 1.0);
    vec3 rainbowColor = hsv2rgb(vec3(hue, 1.0, 1.0));
    vec3 finalColor = texColor * rainbowColor;
    float time_t = pingPong(time_f, 8.0) + 2.0;
    color = vec4(sin(finalColor * time_t), 1.0);
}

```

Shader #832 - rainbow spiral

File: rainbow_spiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);

```

```

    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

vec4 colorRotate(vec4 color) {
    ivec4 col = ivec4(color * 255);
    for (int i = 0; i < 3; ++i) {
        col[i] = col[i] % 0xFF;
    }
    return vec4(col / 255.0);
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float r = length(uv);
    float theta = atan(uv.y, uv.x);
    theta += time_f * 5.0 + r * 15.0;
    uv = vec2(cos(theta), sin(theta)) * r;
    vec3 rainbow_color = rainbow(uv.x + uv.y + time_f);
    vec4 blurred_color = blur(samp, tc, iResolution);
    vec3 blended_color = mix(blurred_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_t), blurred_color.a);
}

```

Shader #833 - rainbow spiral loose

File: rainbow_spiral_loose.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
}
```

```

    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

vec4 colorRotate(vec4 color) {
    ivec4 col = ivec4(color * 255);
    for (int i = 0; i < 3; ++i) {
        col[i] = col[i] % 0xFF;
    }
    return vec4(col / 255.0);
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float r = length(uv);
    float time_t = pingPong(time_f, 15.0) + 2.0;
    float theta = atan(uv.y, uv.x);
    theta += time_f * 2.0 + r * 5.0;
    uv = vec2(cos(theta), sin(theta)) * r;
    vec3 rainbow_color = rainbow(uv.x + uv.y + time_f);
    vec4 blurred_color = blur(samp, tc, iResolution);
    vec3 blended_color = mix(blurred_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_t), blurred_color.a);
}

```

Shader #834 - rainbow spiral rev

File: rainbow_spiral_rev.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}

vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}

vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);

```

```

    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
vec4 colorRotate(vec4 color) {
    ivec4 col = ivec4(color * 255);
    for (int i = 0; i < 3; ++i) {
        col[i] = col[i] % 0xFF;
    }
    return vec4(col / 255.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float r = length(uv);
    float interval = abs(sin(time_f * 0.1)) * 10.0 + 5.0;
    float time_t = pingPong(time_f, interval) + 2.0;
    float direction = sign(sin(time_f * 0.05));
    float theta = atan(uv.y, uv.x);
    theta += direction * (time_f * 2.0 + r * 5.0);
    vec2 spiralDirection = vec2(sin(time_f * 0.1), cos(time_f * 0.1));
    uv = mat2(cos(theta), -sin(theta), sin(theta), cos(theta)) * uv;
    uv += spiralDirection * time_f * 0.1;
    vec3 rainbow_color = rainbow(uv.x + uv.y + time_f);
}

```

```

    vec4 blurred_color = blur(samp, tc, iResolution);
    vec3 blended_color = mix(blurred_color.rgb, rainbow_color, 0.5);
    vec3 final_color = sin(blended_color * time_t) * 0.5 + 0.5;
    color = vec4(final_color, blurred_color.a);
}

```

Shader #835 - rainbow spiral rev2

File: rainbow_spiral_rev2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,

```



```

        1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
        1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
        0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float r = length(uv);
    float interval = abs(sin(time_f * 0.1)) * 10.0 + 5.0;
    float time_t = pingPong(time_f, interval) + 2.0;
    float theta = atan(uv.y, uv.x);
    theta = tan(theta * time_t);
    float direction = sign(sin(time_f * 0.25));
    theta += direction * (time_f * 2.0 + r * 5.0);
    vec2 spiralUV = mat2(cos(theta), -sin(theta), sin(theta), cos(theta)) * uv;
    spiralUV = sin(spiralUV * time_t);
    float mixFactor = (sin(time_f * 0.5) + 1.0) * 0.5;
    uv = mix(uv, spiralUV, mixFactor);
    vec3 rainbow_color = rainbow(uv.x + uv.y + time_f);
    vec4 blurred_color = blur(samp, tc, iResolution);
    vec3 blended_color = mix(blurred_color.rgb, rainbow_color, 0.5);
    vec3 final_color = sin(blended_color * time_t);
    color = vec4(final_color, 1.0);
}

```

Shader #836 - rainbow swirl limit

File: rainbow_swirl_limit.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

```

```

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float t = pingPong(time_f, 15) + 1.0;
    float wave = sin(uv.x * 10.0 + t * 2.0) * 0.1;
    float expand = 0.5 + 0.5 * sin(t * 2.0);
    vec2 spiral_uv = uv * expand + vec2(cos(t), sin(t)) * 0.2;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * t), original_color.a);
}

```

Shader #837 - rainbow swirl limit discard

File: rainbow_swirl_limit_discard.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Discards pure black pixels for transparency.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float t = pingPong(time_f, 15) + 1.0;
    float wave = sin(uv.x * 10.0 + t * 2.0) * 0.1;
    float expand = 0.5 + 0.5 * sin(t * 2.0);
    vec2 spiral_uv = uv * expand + vec2(cos(t), sin(t)) * 0.2;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    if (original_color.rgb == vec3(0.0, 0.0, 0.0)) {
        discard;
    }
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * t), original_color.a);
}

```

Shader #838 - rainbow top

File: rainbow_top.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution; vec3 hsv2rgb(vec3 c) {
    c.x = fract(c.x);
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main() {
    vec2 uv = tc;
    float timeScale = 0.2;
    float spatialScale = 3.0;
    float hue = (uv.x + uv.y) * spatialScale + time_f * timeScale;
    vec3 colorHue = hsv2rgb(vec3(hue, 1.0, 1.0));
    vec4 texColor = texture(samp, uv);
    vec4 finalColor = vec4(texColor.rgb * colorHue, texColor.a);
    color = finalColor;
}
```

Shader #883 - rgb control

File: rgb_control.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
in vec4 optx_val;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
```

```

void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = (optx_val[0]*timeval)*(color[0]*alpha_r);
    color[1] = (optx_val[1]*timeval)*(color[1]*alpha_g);
    color[2] = (optx_val[2]*timeval)*(color[2]*alpha_b);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #886 - RGBColorTrails cache

File: RGBColorTrails_cache.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. 8 History Buffers Clamp index to valid range 0-7 just in case Controls how fast the color assignments cycle through the history buffer

Technique: Multi-tap convolution blur (9 texture samples) - Temporal feedback using cached frame buffers - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform sampler2D samp1;
uniform sampler2D samp2;
uniform sampler2D samp3;
uniform sampler2D samp4;
uniform sampler2D samp5;
uniform sampler2D samp6;
uniform sampler2D samp7;
uniform sampler2D samp8;
uniform vec2 iResolution;
uniform float time_f;
vec4 getHistoryFrame(int index, vec2 uv) {
    int safe_idx = index % 8;
    if (safe_idx == 0) return texture(samp1, uv);
    if (safe_idx == 1) return texture(samp2, uv);
    if (safe_idx == 2) return texture(samp3, uv);
    if (safe_idx == 3) return texture(samp4, uv);
    if (safe_idx == 4) return texture(samp5, uv);
    if (safe_idx == 5) return texture(samp6, uv);
    if (safe_idx == 6) return texture(samp7, uv);
    return texture(samp8, uv);
}
void main(void) {
    int speed = 10;
    int strobe = int(time_f * float(speed));
    vec4 current = texture(samp, tc);

```

```

    int r_idx = (strobe + 0) % 8;
    int g_idx = (strobe + 3) % 8;
    int b_idx = (strobe + 6) % 8;
    vec4 histR = getHistoryFrame(r_idx, tc);
    vec4 histG = getHistoryFrame(g_idx, tc);
    vec4 histB = getHistoryFrame(b_idx, tc);
    float newR = (0.5 * current.r) + (0.5 * histR.r);
    float newG = (0.5 * current.g) + (0.5 * histG.g);
    float newB = (0.5 * current.b) + (0.5 * histB.b);
    color = vec4(newR, newG, newB, 1.0);
}

```

Shader #887 - rgbt

File: rgbt.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc;
    float trailStrength = 0.05;
    float strobeSpeed = sin(time_f * 10.0);
    vec2 trailOffsetR = vec2(trailStrength * strobeSpeed, 0.0);
    vec2 trailOffsetG = vec2(-trailStrength * strobeSpeed, trailStrength * strobeSpeed);
    vec2 trailOffsetB = vec2(0.0, -trailStrength * strobeSpeed);
    vec3 rgbTrail = vec3(
        texture(samp, uv + trailOffsetR).r,
        texture(samp, uv + trailOffsetG).g,
        texture(samp, uv + trailOffsetB).b
    );
    float strobe = 0.5 + 0.5 * sin(time_f * 20.0);
    vec3 baseColor = texture(samp, uv).rgb;
    color = vec4(mix(baseColor, rgbTrail, strobe), 1.0);
}

```

Shader #888 - rgbvar inc

File: rgbvar_inc.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;

```

```

uniform float alpha_b;
in float timeval;
uniform float alpha;
in vec4 optx_val;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = (color[0]*timeval*alpha_r)*optx_val[0];
    color[1] = (color[1]*timeval*alpha_g)*optx_val[1];
    color[2] = (color[2]*timeval*alpha_b)*optx_val[2];
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int(tc[0])^int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #889 - rhue

File: rhue.glsl

Applies geometric transformations including rotation and scaling to distort the image. Computes luminance for brightness-based effects.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec2 rotateCoords(vec2 coords, float angle) {
    float s = sin(angle);
    float c = cos(angle);
    return vec2(c * coords.x - s * coords.y, s * coords.x + c * coords.y);
}
vec4 adjustHue(vec4 color, float angle) {
    float U = cos(angle);
    float W = sin(angle);
    mat3 rotationMatrix = mat3(
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114,

```

```

        0.299, 0.587, 0.114
    ) + mat3(
        0.701, -0.587, -0.114,
        -0.299, 0.413, -0.114,
        -0.3, -0.588, 0.886
    ) * U + mat3(
        0.168, 0.330, -0.497,
        -0.328, 0.035, 0.292,
        1.25, -1.05, -0.203
    ) * W;
    return vec4(rotationMatrix * color.rgb, color.a);
}

void main() {
    vec2 uv = (tc - 0.5) * 2.0;
    uv = rotateCoords(uv, time_f);
    vec4 sampledColor = texture(samp, uv * 0.5 + 0.5);
    vec4 rotatedColor = adjustHue(sampledColor, length(uv) * time_f);
    color = rotatedColor;
}

```

Shader #890 - rightintwo

File: rightintwo.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec2 cord = vec2(tc[0]/2, tc[1]/2);
    vec4 clr = texture(samp, cord);
    vec2 p = vec2(gl_FragCoord.x/2, gl_FragCoord.y/4);
    vec2 left = fract(p);
    for(int i = 0; i < 3; ++i) {
        color[i] = (color[i] * clr[i]) * (left[0]+left[1]);
        color[i] += 2;
    }
}

```

```

    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #910 - sac geo

File: sac-geo.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
 - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec3 sacredGeometry(vec2 uv, float time) {
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float frequency = 5.0;
    float amplitude = 0.5;
    float sacredPattern = sin(radius * frequency + time) * amplitude + 0.5;
    float hue = fract(sacredPattern + time * 0.1);
    float saturation = 1.0;
    float value = 1.0;
    return hsv2rgb(vec3(hue, saturation, value));
}
void main() {
    vec2 uv = tc * 2.0 - 1.0;
    uv *= vec2(iResolution.x / iResolution.y, 1.0);
    float time = time_f * 0.1;
    vec3 sacredColor = sacredGeometry(uv, time);
    vec2 complexCoord = uv * vec2(cos(time), sin(time));
    vec3 complexColor = sacredGeometry(complexCoord, time);
    vec4 textureColor = texture(samp, tc);
    vec3 combinedColor = mix(sacredColor, complexColor, 0.5);
    color = vec4(mix(textureColor.rgb, combinedColor, 0.5), 1.0);
}

```


Shader #920 - sepia

File: sepia.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec4 texColor = texture(samp, tc);
    float r = texColor.r;
    float g = texColor.g;
    float b = texColor.b;
    color = vec4(
        dot(vec3(r, g, b), vec3(0.393, 0.769, 0.189)),
        dot(vec3(r, g, b), vec3(0.349, 0.686, 0.168)),
        dot(vec3(r, g, b), vec3(0.272, 0.534, 0.131)),
        texColor.a
    );
}
```

Shader #931 - sine1

File: sine1.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    vec2 uv = tc * iResolution.xy / iResolution.y;
    float pattern = sin(uv.x * 10.0) * cos(uv.y * 10.0);
    pattern = (pattern + 1.0) * 0.5;
    float reveal = pingPong(time_f * 0.5, 1.0);
    float cracks = step(pattern, reveal);
    float hue = pattern * 6.0;
    vec3 colorRGB = hsv2rgb(vec3(hue, 1.0, 1.0));
    color = vec4(colorRGB * cracks, 1.0);
    color = mix(color, texture(samp, tc), 0.5);
}
```

```
    color.a = 1.0;
}
```

Shader #932 - sine2

File: sine2.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    vec2 uv = tc * iResolution.xy / iResolution.y;
    float pattern = sin(uv.x * 10.0) * cos(uv.y * 10.0);
    pattern = (pattern + 1.0) * 0.5;
    float reveal = 1.0;
    float cracks = step(pattern, reveal);
    float hue = pattern * 6.0;
    vec3 colorRGB = hsv2rgb(vec3(sin(hue * time_f), 1.0, 1.0));
    float time_t = pingPong(time_f, 8.0) + 2.0;
    color = vec4(colorRGB * cracks * time_t, 1.0);
    color = mix(sin(color * time_t), color + tan(texture(samp,tc) * time_t), 0.5);
    color.a = 1.0;
}
```

Shader #938 - sine_grad6

File: sine_grad6.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
```

```

    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 rainbow(float t) {
    float hue = mod(t, 1.0) * 6.0;
    float c = 1.0;
    float x = 1.0 - abs(mod(hue, 2.0) - 1.0);
    vec3 rgb = hue < 1.0 ? vec3(c, x, 0.0) :
                hue < 2.0 ? vec3(x, c, 0.0) :
                hue < 3.0 ? vec3(0.0, c, x) :
                hue < 4.0 ? vec3(0.0, x, c) :
                hue < 5.0 ? vec3(x, 0.0, c) :
                vec3(c, 0.0, x);

    return rgb;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float radius = dist + sin(time_f * 2.0 + dist * 10.0) * 0.05;
    vec2 spiral_offset = vec2(cos(angle), sin(angle)) * radius;
    vec2 spiral_tc = center + spiral_offset;
    spiral_tc = clamp(spiral_tc, vec2(0.0), vec2(1.0));
    float gradient_pos = mod(dist * 3.0 - time_f * 0.5, 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec4 ctx = texture(samp, tc);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = mix(sin(ctx * time_t), vec4(color_gradient, 1.0), 0.5);
}

```

Shader #939 - sine_grad7

File: sine_grad7.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 rainbow(float t) {
    float hue = mod(t, 1.0) * 6.0;
    float c = 1.0;
    float x = 1.0 - abs(mod(hue, 2.0) - 1.0);
    vec3 rgb = hue < 1.0 ? vec3(c, x, 0.0) :
                hue < 2.0 ? vec3(x, c, 0.0) :
                hue < 3.0 ? vec3(0.0, c, x) :
                hue < 4.0 ? vec3(0.0, x, c) :
                hue < 5.0 ? vec3(x, 0.0, c) :
                vec3(c, 0.0, x);
}

```

```

        vec3(c, 0.0, x);

    return rgb;
}

void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float spiralTime = time_f * 0.5;
    float gradient_pos = mod(dist * 3.0 + angle / (2.0 * 3.14159) + spiralTime, 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec2 spiral_offset = vec2(cos(angle), sin(angle)) * dist;
    vec2 spiral_tc = center + spiral_offset;
    spiral_tc = clamp(spiral_tc, vec2(0.0), vec2(1.0));
    vec4 ctx = texture(samp, tc);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = mix(sin(ctx * time_t), vec4(color_gradient, 1.0), 0.5);
    color.a = 1.0;
}

```

Shader #977 - srainbow

File: srainbow.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float dispersion = 0.02;
    vec2 dir = normalize(uv);
    vec2 offset_r = tc + dir * dispersion;
    vec2 offset_g = tc;
    vec2 offset_b = tc - dir * dispersion;
    float r = texture(samp, offset_r).r;
    float g = texture(samp, offset_g).g;
    float b = texture(samp, offset_b).b;
}

```

```

    vec3 prism_color = vec3(r, g, b);
    float t = fract((angle / (2.0 * 3.14159)) + time_f * 0.1);
    vec3 rainbow_color = rainbow(t);
    float rainbow_factor = 0.5;
    vec3 final_color = mix(prism_color, rainbow_color, rainbow_factor);
    float time_t = pingPong(time_f, 4.0) + 2.0;
    color = vec4(sin(final_color * time_t), 1.0);
}

```

Shader #986 - strobe colors

File: strobe_colors.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    color[0] = color[0] * sin(fract(alpha * timeval));
    color[1] = color[1] * cos(fract(alpha * timeval));
    color[2] = color[2] * tan(fract(alpha * timeval));
}

```

Cursor & Mouse (117 shaders)

Shader #22 - air full mouse

File: air_full_mouse.glsl

Uses distance fields with sinusoidal modulation to generate animated visual patterns. Responds to mouse position for interactive control.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation
- Driven by: time, mouse, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution.xy) : vec2(0.5);
    vec2 uv = tc;
    float dist = distance(uv, m);
    float falloff = smoothstep(0.5, 0.0, dist);
    float distortionStrength = 0.03 * falloff;
    float distortionFrequency = 20.0;
    float distortion1 = sin((uv.y + dist * 5.0) * distortionFrequency + time_f) * distortionStrength;
    float distortion2 = cos((uv.x + uv.y + dist * 5.0) * distortionFrequency + time_f) *
    ↪ distortionStrength;
    vec2 distortedTC1 = uv + vec2(distortion1, distortion2);
    vec2 distortedTC2 = uv + vec2(distortion2, -distortion1);
    vec4 texColor1 = texture(samp, distortedTC1);
    vec4 texColor2 = texture(samp, distortedTC2);
    color = mix(texColor1, texColor2, 0.5 + 0.5 * sin(time_f));
}
```

Shader #23 - air mouse

File: air_mouse.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
float h1(float n){return fract(sin(n*91.345+37.12)*43758.5453123);}
vec2 h2(vec2 p){return fract(sin(vec2(dot(p,vec2(127.1,311.7)),dot(p,vec2(269.5,183.3))))*43758.5453);}
vec2 rot(vec2 v,float a){float c=cos(a),s=sin(a);return vec2(c*v.x-s*v.y,s*v.x+c*v.y);}
void main(void){
    vec2 uv=tc;
```

```

float a=clamp(amp,0.0,1.0);
float ua=clamp(uamp,0.0,1.0);
vec2 m = iMouse.z>0.5 ? (iMouse.xy/iResolution) : fract(vec2(0.37+0.11*sin(time_f*0.63),0.42+0.13*
↳ cos(time_f*0.57)));
float baseStr = mix(0.002,0.018,a);
float freq = mix(6.0,22.0,ua);
float t = time_f;
float s1 = sin((uv.y+0.07*sin(uv.x*3.0)) * freq + t*1.3);
float s2 = cos((uv.x+uv.y*0.5) * (freq*0.7) + t*1.7);
vec2 wv = vec2(s1,s2)*baseStr;
vec2 d = uv - m;
float r = length(d)+1e-5;
float fall = smoothstep(0.5,0.0,r);
float swirl = (0.15+0.45*ua+0.25*a) * fall * (0.7+0.3*sin(t*2.1+seed));
vec2 tang = rot(normalize(d), 1.5707963);
vec2 air = tang * swirl * (0.02+0.18*a) * (0.6+0.4*cos(uv.y*40.0+t*3.0));
vec2 nJit = (h2(uv*vec2(233.3,341.9)+t+seed)-0.5);
vec2 jitter = nJit * (0.0005 + 0.004*ua) * (0.5+0.5*sin(t*5.0+seed*3.0));
vec2 uv0 = uv + wv + air + jitter;
float ca = 0.0015 + 0.004*a;
vec4 c;
c.r = texture(samp, uv0 + vec2( ca,0.0)).r;
c.g = texture(samp, uv0 + vec2(-ca,0.0)).g;
c.b = texture(samp, uv0 + vec2(-ca,0.0)).b;
float pulse = 0.005*(0.5+0.5*sin(t*3.7+seed));
c.rgb += vec3(pulse)*ua;
color = vec4(c.rgb,1.0);
}

```

Shader #33 - apart mouse

File: apart_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0,-s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
vec2 pseudoRandomDirection(float t){
    vec2 v = vec2(sin(t * 1.3), cos(t * 1.7));
    return normalize(v);
}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    float zoom = mix(0.1, 1.4, 0.5 + 0.5 * sin(time_f * 0.6));
    vec2 p = (tc - m) * ar / zoom;
    float ax = 0.3 * sin(time_f * 0.7);
}

```

```

float ay = 0.25 * cos(time_f * 0.6);
float az = 0.4 * time_f;
mat3 R = rotZ(az) * rotY(ay) * rotX(ax);
vec3 q = R * vec3(p, 1.0);
float zf = 1.0 / (1.0 + q.z * 0.8);
vec2 pr = q.xy * zf;
float dist = length(pr);
float ang = atan(pr.y, pr.x) + time_f * 2.0;
float radius = dist * (1.0 + 0.1 * sin(time_f * 3.0 + dist * 10.0));
vec2 dir = pseudoRandomDirection(time_f + dist * 5.0);
vec2 spiral = vec2(cos(ang), sin(ang)) * radius * 0.5 + dir * 0.1 * sin(time_f * 2.0);
vec2 uv = spiral / ar + m;
uv = clamp(uv, 0.0, 1.0);
color = texture(samp, uv);
}

```

Shader #38 - audio mouse

File: audio_mouse.glsl

Uses distance fields with sinusoidal modulation to generate animated visual patterns. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation
 - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float uamp;
uniform float amp;
uniform vec4 iMouse;
void main(void) {
    vec2 mousePos = iMouse.xy / iResolution.xy;
    float dist = distance(tc, mousePos);
    float glitchStrength = exp(-dist * 10.0) * (uamp) * 4.0;
    vec2 glitchOffset = vec2(
        sin(time_f * 10.0 + tc.y * 20.0) * glitchStrength,
        cos(time_f * 15.0 + tc.x * 25.0) * glitchStrength
    );
    vec2 distortedTc = tc + glitchOffset;
    color = texture(samp, distortedTc);
    vec4 originalColor = texture(samp, tc);
    float blendFactor = smoothstep(0.0, 0.1, glitchStrength);
    color = mix(originalColor, color, blendFactor);
}

```

Shader #41 - aura2

File: aura2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
 - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation -
 Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*0.35+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t*0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a*k1 + b*k2 + c*k3) * 1.05;
}
```

```

vec3 softTone(vec3 c){
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299,0.587,0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res){
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts*vec2(-1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts*vec2( 0.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts*vec2( 1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts*vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts*vec2( 1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts*vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts*vec2( 0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts*vec2( 1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0*s10 + s20 + 2.0*s01 + 4.0*s11 + 2.0*s21 + s02 + 2.0*s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv){
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 gdir = normalize(vec2(cos(time_f*0.27), sin(time_f*0.31)));
    float s = dot((uv-0.5)*vec2(iResolution.x/iResolution.y,1.0), gdir);
    float w = max(fwidth(s)*4.0, 0.002);
    float band = smoothstep(-0.5-w, -0.5+w, sin(s*2.2 + time_f*0.9));
    vec3 neon = neonPalette(time_f);
    vec3 grad = mix(tex, mix(tex, neon, 0.6), 0.35 + 0.25*band);
    grad = mix(grad, tex, 0.10);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void){
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float radius = mix(0.8, 1.2, 0.5 + 0.5 * sin(time_f * 1.3));
    radius *= 2.0;
    float r = length(uv);
    float glow = smoothstep(radius, radius - 0.25, r);
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + 2.0*sin(time_f*0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * 0.23, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
}

```

```

vec2 q = abs(p); if(q.y > q.x) q = q.yx;
float base = 1.82 + 0.18*sin(time_f*0.2);
float period = log(base);
float tz = time_f * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + time_f*0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap*1.045) / ar + m);
vec2 u2 = fract((pwrap*0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * (0.0015 + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
vign = mix(0.9, 1.15, vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD*1e-3)*9.5 + time_f*1.2));
float pulse = 0.5 + 0.5*sin(time_f*2.0 + rD*28.0 + k*12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = outCol*outCol*0.18 + pow(max(outCol-0.6,0.0), vec3(2.0))*0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #56 - bend2mouse

File: bend2mouse.glsl

Processes the input texture to produce the "bend2mouse" visual effect. Responds to mouse position for interactive control. Only when mouse is dragged Convert mouse coordinates to UV space
Calculate drag vector components

Technique: Hermite smoothstep interpolation - Driven by: mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5);
    vec2 dragEffect = vec2(0.0);
    if (iMouse.z > 0.0) {
        vec2 clickPos = iMouse.zw / iResolution.xy;
        vec2 currentPos = iMouse.xy / iResolution.xy;
        vec2 dragVec = currentPos - clickPos;
        float dragStrength = length(dragVec) * 2.0;
        vec2 dragDir = normalize(dragVec + vec2(0.0001));
        vec2 uvOffset = uv - clickPos;
        float distFromClick = length(uvOffset);
    }
}

```

```

float parallel = dot(uvOffset, dragDir);
float perp = dot(uvOffset, vec2(-dragDir.y, dragDir.x));
parallel *= 1.0 + dragStrength;
float wave = sin(parallel * 20.0 - dragStrength * 10.0) * 0.02;
float vortex = tan(perp * 15.0 + dragStrength * 5.0) * 0.01;
dragEffect = vec2(
    parallel * dragDir.x - perp * dragDir.y,
    parallel * dragDir.y + perp * dragDir.x
) * (1.0 + wave + vortex);
float bubble = smoothstep(0.2, 0.1, distFromClick);
dragEffect += bubble * dragVec * 0.5;
}
color = texture(samp, uv + dragEffect);
}

```

Shader #84 - brot zoom mouse

File: brot-zoom-mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
void main(void){
    float aspect=iResolution.x/iResolution.y;
    vec2 ar=vec2(aspect,1.0);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 p=(tc-m)*ar;
    vec3 v=vec3(p,1.0);
    float ax=0.25*sin(time_f*0.7);
    float ay=0.25*cos(time_f*0.6);
    float az=0.4*time_f;
    mat3 R=rotZ(az)*rotY(ay)*rotX(ax);
    vec3 r=R*v;
    float persp=0.6;
    float zf=1.0/(1.0+r.z*persp);
    vec2 q=r.xy*zf;
    float eps=1e-6;
    float base=1.72;
    float period=log(base);
    float t=time_f*0.5;
    float rad=length(q)+eps;
    float ang=atan(q.y,q.x)+t*0.3;
    float k=fract((log(rad)-t)/period);
    float rw=exp(k*period);
    vec2 qwrap=vec2(cos(ang),sin(ang))*rw;
    float N=8.0;
    float stepA=6.28318530718/N;
    float a=atan(qwrap.y,qwrap.x)+time_f*0.05;
}

```

```

    a=mod(a,stepA);
    a=abs(a-stepA*0.5);
    vec2 kdir=vec2(cos(a),sin(a));
    vec2 kaleido=kdir*length(qwrap);
    vec2 uv=kaleido/ar+m;
    uv=fract(uv);
    color=texture(samp,uv);
}

```

Shader #85 - bubble 2

File: bubble-2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float len = length(uv);
    float time_t = pingPong(time_f, 10.0);
    float bubble = smoothstep(0.8, 1.0, 1.0 - len);
    bubble = sin(bubble * time_t);
    vec2 distort = uv * (1.0 + 0.1 * sin(time_f + len * 20.0));
    distort = sin(distort * time_t);
    vec4 texColor = texture(samp, distort * 0.5 + 0.5);
    color = mix(texColor, vec4(1.0, 1.0, 1.0, 1.0), bubble);
}

```

Shader #86 - bubble move

File: bubble-move.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float h1(float n){return fract(sin(n)*43758.5453123);}
vec2 h2(float n){return fract(sin(vec2(n,n+1.0))*vec2(43758.5453,22578.1459));}
void main(void){

```

```

float rate = 0.4;
float t = time_f*rate;
float t0 = floor(t);
float a = fract(t);
float s = a*a*(3.0-2.0*a);
vec2 p0 = 0.2 + h2(t0)*0.6;
vec2 p1 = 0.2 + h2(t0+1.0)*0.6;
vec2 center = mix(p0,p1,s);
float r0 = 0.35 + 0.1*h1(t0*1.7);
float r1 = 0.35 + 0.1*h1((t0+1.0)*1.7);
float radius = mix(r0,r1,s);
vec2 uv = tc;
vec2 d = uv - center;
float len = length(d)/radius;
float wob = 0.12 + 0.06*h1(t0*2.3);
vec2 distort = d * (1.0 + wob * sin(time_f*1.6 + len*22.0));
vec2 coord = clamp(center + distort, 0.001, 0.999);
vec4 texColor = texture(samp, coord);
float bubble = smoothstep(0.8, 1.0, 1.0 - len);
color = mix(texColor, vec4(1.0), bubble);
}

```

Shader #87 - bubble zoom mouse

File: bubble-zoom-mouse.glsl

Renders a fractal pattern using iterative computation (many iterations). Responds to mouse position for interactive control.

Technique: Iterative fractal computation with complex-plane mapping - Polar coordinate transformation ($\text{atan2} + \text{length}$) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p = (tc - m) * ar;
    float base = 1.65;
    float period = log(base);
    float tz = time_f * 0.45;
    float r0 = length(p) + 1e-6;
    float a0 = atan(p.y, p.x) + tz * 0.35;
    float k = fract((log(r0) - tz) / period);
    float rw = exp(k * period);
    vec2 pz = vec2(cos(a0), sin(a0)) * rw;
    float time_t = pingPong(time_f, 10.0);
    float len_iso = length(pz / ar);
    float bubble = smoothstep(0.8, 1.0, 1.0 - len_iso);
}

```

```

    bubble = sin(bubble * time_t);
    vec2 distort = pz * (1.0 + 0.1 * sin(time_f + length(pz) * 20.0));
    distort = sin(distort * time_t);
    vec2 uv = fract(distort / ar + m);
    vec4 texColor = texture(samp, uv);
    color = mix(texColor, vec4(1.0), bubble);
}

```

Shader #88 - bubble

File: bubble.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float len = length(uv);
    float bubble = smoothstep(0.8, 1.0, 1.0 - len);
    vec2 distort = uv * (1.0 + 0.1 * sin(time_f + len * 20.0));
    vec4 texColor = texture(samp, distort * 0.5 + 0.5);
    color = mix(texColor, vec4(1.0, 1.0, 1.0, 1.0), bubble);
}

```

Shader #89 - bubble amp

File: bubble_amp.glsl

Processes the input texture to produce the "bubble amp" visual effect. Responds to audio input (amplitude/volume) for interactive control. Audio amplitude (0.0 to 1.0) Transform texture coordinates to [-1, 1] Distance from the center

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float uamp;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float len = length(uv);
    float bubble = smoothstep(0.7, 1.0, 1.0 - len);
    float beat = 1.0 + 0.2 * sin(uamp * 100.0);
    vec2 distort = uv * (1.0 + 0.15 * sin(len * 20.0 * beat));
    vec4 texColor = texture(samp, distort * 0.5 + 0.5);
    color = mix(texColor, vec4(1.0, 1.0, 1.0, 1.0), bubble * uamp);
}

```

Shader #90 - bubbleGL

File: bubbleGL.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float h1(float n){return fract(sin(n)*43758.5453123);}
vec2 h2(float n){return fract(sin(vec2(n,n+1.0))*vec2(43758.5453,22578.1459));}
void main(void){
    float rate = 0.4;
    float t = time_f*rate;
    float t0 = floor(t);
    float a = fract(t);
    float s = a*a*(3.0-2.0*a);
    vec2 p0 = 0.2 + h2(t0)*0.6;
    vec2 p1 = 0.2 + h2(t0+1.0)*0.6;
    vec2 center = mix(p0,p1,s);
    float r0 = 0.35 + 0.1*h1(t0*1.7);
    float r1 = 0.35 + 0.1*h1((t0+1.0)*1.7);
    float radius = mix(r0,r1,s);
    vec2 uv = tc;
    vec2 d = uv - center;
    float len = length(d)/radius;
    float wob = 0.12 + 0.06*h1(t0*2.3);
    vec2 distort = d * (1.0 + wob * sin(time_f*1.6 + len*22.0));
    vec2 coord = clamp(center + sin(distort * time_f), 0.001, 0.999);
    vec4 texColor = texture(samp, coord);
    float bubble = smoothstep(0.8, 1.0, 1.0 - len);
    color = mix(texColor, vec4(1.0), bubble);
}
```

Shader #91 - BubbleMouse

File: BubbleMouse.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control. Convert to [-1, 1] range Default to no distortion Check if the mouse is being dragged (zw is initial click position)

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
```



```

vec2 uv = tc * 2.0 - 1.0;
vec2 distort = uv;
float bubble = 0.0;
if (iMouse.z != 0.0 || iMouse.w != 0.0) {
    vec2 clickPos = (iMouse.zw / iResolution.xy) * 2.0 - 1.0;
    vec2 currentPos = (iMouse.xy / iResolution.xy) * 2.0 - 1.0;
    vec2 dragVec = currentPos - clickPos;
    float dragLen = length(dragVec);
    vec2 dragDir = normalize(dragVec + vec2(0.0001));
    vec2 uv_centered = uv - clickPos;
    float parallel = dot(uv_centered, dragDir);
    vec2 perp = uv_centered - parallel * dragDir;
    parallel *= 1.0 + dragLen * 3.0;
    vec2 stretchedUV = perp + parallel * dragDir;
    float lenStretched = length(stretchedUV);
    bubble = smoothstep(0.6, 0.9, 1.0 - lenStretched);
    distort = clickPos + stretchedUV * (1.0 + 0.1 * sin(time_f + lenStretched * 20.0));
}
vec2 texCoord = distort * 0.5 + 0.5;
vec4 texColor = texture(samp, texCoord);
color = mix(texColor, vec4(1.0), bubble);
}

```

Shader #99 - code flux mouse

File: code_flux_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
uniform vec4 iMouse;
in vec2 tc;
out vec4 color;
vec4 xor_RGB(vec4 icolor, vec4 source){
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255.0);
    for(int i=0;i<3;++i){
        int_color[i] = int(255.0*icolor[i]);
        int_color[i] = int_color[i] ^ isource[i];
        if(int_color[i]>255) int_color[i] = int_color[i] % 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x,float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
void main(){
    vec2 center = (iMouse.z>0.5 || iMouse.w>0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p = (tc - center) * vec2(1.0, iResolution.y / iResolution.x);
    float intensity = pingPong(time_f, 10.0);

```

```

float angle = atan(p.y, p.x);
float radius = length(p);
float swirl = sin(time_f*0.5)*0.5+0.5;
angle += intensity * swirl * sin(radius*10.0 + time_f);
vec2 q = vec2(cos(angle), sin(angle)) * radius;
vec2 uv = q * vec2(1.0, iResolution.x / iResolution.y) + center;
vec4 texColor = texture(samp, uv);
float fluctuation = sin(time_f*2.0)*0.5+0.5;
vec4 fluctuatedColor = vec4(mix(vec3(1.0,0.0,0.0), vec3(0.0,0.0,1.0), fluctuation), 1.0);
vec4 xorResult = xor_RGB(texColor, fluctuatedColor);
color = mix(texColor, xorResult, 0.5);
}

```

Shader #100 - code rev3

File: code_rev3.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control. Mouse-based control: use normalized mouse Y to influence wave amplitude

Technique: Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
    float yPos = 1.0 - (gl_FragCoord.y / iResolution.y);
    float mouseY = (iMouse.z > 0.5) ? (iMouse.y / iResolution.y) : 0.5;
    float waveAmplitude = mix(0.05, 0.35, mouseY);
    float wavePhase = sin(yPos * 10.0 + time_f * 2.0 + mouseY * 6.2831);
    vec2 tcAdjusted = tc + vec2(0.0, wavePhase * waveAmplitude);
    color = texture(samp, tcAdjusted);
}

```

Shader #127 - color swirl o1 mouse

File: color_swirl_o1_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;
float pingPong(float x, float length) {

```

```

    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 m = (iMouse.z > 0.5 ? iMouse.xy : 0.5 * iResolution) / iResolution;
    vec2 uv = (tc - m) * vec2(iResolution.x / iResolution.y, 1.0);
    float t = time_f * 0.5;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x) + t;
    float radMod = pingPong(radius + t * 0.5, 0.5);
    float wave = sin(radius * 10.0 - t * 5.0) * 0.5 + 0.5;
    float r = sin(angle * 3.0 + radMod * 10.0 + wave * 6.2831);
    float g = sin(angle * 4.0 - radMod * 8.0 + wave * 4.1230);
    float b = sin(angle * 5.0 + radMod * 12.0 - wave * 3.4560);
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc).rgb;
    col = mix(col, texColor, 0.3);
    color = vec4(col, alpha);
}

```

Shader #129 - color swirl sin mouse

File: color_swirl_sin_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 m = (iMouse.z > 0.5 ? iMouse.xy : 0.5 * iResolution) / iResolution;
    vec2 uv = (tc - m) * vec2(iResolution.x / iResolution.y, 1.0);
    float t = time_f * 0.5;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t;
    float radMod = pingPong(radius + t * 0.5, 0.5);
    float wave = sin(radius * 10.0 - t * 5.0) * 0.5 + 0.5;
    float r = sin(angle * 3.0 + radMod * 10.0 + wave * 6.2831);
    float g = sin(angle * 4.0 - radMod * 8.0 + wave * 4.1230);
    float b = sin(angle * 5.0 + radMod * 12.0 - wave * 3.4560);
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc).rgb;
    col = mix(col, texColor, 0.3);
    color = vec4(sin(col * pingPong(time_f, 8.0) + 2.0), alpha);
}

```

Shader #133 - color w mouse

File: color_w_mouse.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}

float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}

void main(void) {
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = (tc - m) * vec2(iResolution.x / iResolution.y, 1.0);
    float t = time_f * 0.7;
    float beat = abs(sin(time_f * 3.14159)) * 0.2 + 0.8;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t * 0.5;
    float radMod = pingPong(radius + t * 0.3, 0.5);
    float wave = sin(radius * 10.0 - t * 6.0) * 0.5 + 0.5;
    float distortion = sin((radius + t * 0.5) * 8.0) * beat * 0.1;
    vec2 dir = vec2(cos(angle), sin(angle));
    vec2 tcSample = tc + dir * distortion;
    vec3 texColor = texture(samp, tcSample).rgb;
    float noiseEffect = noise(uv * 10.0 + t * 0.5) * 0.2;
    float r = sin(angle * 3.0 + radMod * 8.0 + wave * 6.2831 + noiseEffect);
    float g = sin(angle * 4.0 - radMod * 6.0 + wave * 4.1230 + noiseEffect);
    float b = sin(angle * 5.0 + radMod * 10.0 - wave * 3.4560 - noiseEffect);
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    col = mix(col, texColor, 0.6);
    vec3 lightDir = normalize(vec3(0.5, 0.5, 1.0));
    float d = dot(uv, uv);
    float z = sqrt(max(0.0, 1.0 - d));
    vec3 norm = normalize(vec3(uv, z));
    float light = dot(norm, lightDir) * 0.5 + 0.5;
    col *= light * 1.2;
    col *= beat;
    color = vec4(col, alpha);
}
```

Shader #137 - comb3 frac mouse

File: comb3-frac-mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec2 rotateUV(vec2 uv, float angle, vec2 center) {
    float s = sin(angle);
    float c = cos(angle);
    uv -= center;
    uv = mat2(c, -s, s, c) * uv;
    uv += center;
    return uv;
}

vec2 reflectUV(vec2 uv, float segments, vec2 center) {
    float angle = atan(uv.y - center.y, uv.x - center.x);
    float radius = length(uv - center);
    float segmentAngle = 2.0 * 3.14159265359 / segments;
    angle = mod(angle, segmentAngle);
    angle = abs(angle - segmentAngle * 0.5);
    return vec2(cos(angle), sin(angle)) * radius + center;
}

vec2 fractalZoom(vec2 uv, float zoom, float time, vec2 center) {
    for (int i = 0; i < 5; i++) {
        uv = abs((uv - center) * zoom) - 0.5 + center;
        uv = rotateUV(uv, sin(time * pingPong(time_f, 10.0)) * 0.1, center);
    }
    return uv;
}

void main() {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 center = m * iResolution / vec2(iResolution.y);
    vec4 originalTexture = texture(samp, tc);
    vec2 kaleidoUV = reflectUV(uv, 6.0, center);
    float zoom = 1.5 + 0.5 * sin(time_f * 0.5);
    kaleidoUV = fractalZoom(kaleidoUV, zoom, time_f, center);
    kaleidoUV = rotateUV(kaleidoUV, time_f * 0.2, center);
    vec4 kaleidoColor = texture(samp, kaleidoUV);
    float blendFactor = 0.6;
    vec4 blendedColor = mix(kaleidoColor, originalTexture, blendFactor);
    blendedColor.rgb *= 0.5 + 0.5 * sin(kaleidoUV.xyx + time_f);
    color = sin(blendedColor * pingPong(time_f, 10.0));
}
```

```

    vec4 t = texture(samp, tc);
    color = color * t * 0.8;
    color = sin(color * pingPong(time_f, 15.0));
    color.a = 1.0;
}

```

Shader #138 - comb3 frac mouse2

File: comb3-frac-mouse2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec2 rotateUV(vec2 uv, float angle, vec2 center) {
    float s = sin(angle);
    float c = cos(angle);
    uv -= center;
    uv = mat2(c, -s, s, c) * uv;
    uv += center;
    return uv;
}

vec2 reflectUV(vec2 uv, float segments, vec2 center) {
    float angle = atan(uv.y - center.y, uv.x - center.x);
    float radius = length(uv - center);
    float segmentAngle = 2.0 * 3.14159265359 / segments;
    angle = mod(angle, segmentAngle);
    angle = abs(angle - segmentAngle * 0.5);
    return vec2(cos(angle), sin(angle)) * radius + center;
}

vec2 fractalZoom(vec2 uv, float zoom, float time, vec2 center) {
    for (int i = 0; i < 5; i++) {
        uv = abs((uv - center) * zoom) - 0.5 + center;
        uv = rotateUV(uv, sin(time * pingPong(time_f, 10.0)) * 0.1, center);
    }
    return uv;
}

void main() {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 center = m * iResolution / vec2(iResolution.y);
    vec4 originalTexture = texture(samp, tc);
    vec2 kaleidoUV = reflectUV(uv, 6.0, center);
    float zoom = 1.5 + 0.5 * sin(time_f * 0.5);
    kaleidoUV = fractalZoom(kaleidoUV, zoom, time_f, center);
    kaleidoUV = rotateUV(kaleidoUV, time_f * 0.2, center);
    vec4 kaleidoColor = texture(samp, kaleidoUV);
}

```

```

float blendFactor = 0.6;
vec4 blendedColor = mix(kaleidoColor, originalTexture, blendFactor);
blendedColor.rgb *= 0.5 + 0.5 * sin(kaleidoUV.xy * time_f);
color = sin(blendedColor * pingPong(time_f, 10.0));
vec4 t = texture(samp, tc);
color = color * t * 0.8;
color = sin(color * pingPong(time_f, 15.0));
color.a = 1.0;
}

```

Shader #140 - comb3 geo mouse

File: comb3_geo_mouse.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float h1(float n){return fract(sin(n)*43758.5453123);}
vec2 h2(float n){return fract(sin(vec2(n, n+1.0))*vec2(43758.5453,22578.1459));}
vec2 rotateUV(vec2 uv, float a, vec2 c, float aspect) {
    float s = sin(a), cv = cos(a);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cv, -s, s, cv) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float r = length(p);
    float seg = 6.28318530718 / segments;
    ang = mod(ang, seg);
    ang = abs(ang - seg * 0.5);
    vec2 q = vec2(cos(ang), sin(ang)) * r;
    q.x /= aspect;
    return q + c;
}
vec2 tileMirror(vec2 uv, float tiles, vec2 c) {
    vec2 p = (uv - c) * tiles;
    p = abs(fract(p) * 2.0 - 1.0);
    return p / tiles + c;
}
vec2 swirl(vec2 uv, vec2 c, float aspect, float k) {

```

```

    vec2 p = uv - c;
    p.x *= aspect;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x) + k * r;
    vec2 q = vec2(cos(a), sin(a)) * r;
    q.x /= aspect;
    return q + c;
}

vec2 fractalZoom(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 5; i++) {
        p = abs((p - c) * zoom) - 0.5 + c;
        p = rotateUV(p, t * 0.1, c, aspect);
    }
    return p;
}

void main() {
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    float t = time_f;
    float modef = mod(floor(t * 0.1666667), 4.0);
    int mode = int(modef);
    float seed = floor(t * 0.1666667);
    float seg = 4.0 + floor(h1(seed) * 8.0);
    float tiles = 2.0 + floor(h1(seed + 3.1) * 6.0);
    float k = (h1(seed + 7.7) * 2.0 - 1.0) * (0.6 + 0.6 * sin(t * 0.37));
    float zoom = 1.3 + 0.6 * sin(t * 0.5 + h1(seed + 1.7) * 6.2831853);
    vec2 duv = uv;
    if (mode == 0) duv = reflectUV(uv, seg, m, aspect);
    if (mode == 1) duv = tileMirror(rotateUV(uv, 0.4 * sin(t * 0.6), m, aspect), tiles, m);
    if (mode == 2) duv = swirl(uv, m, aspect, k * 6.0);
    if (mode == 3) duv = fractalZoom(uv, zoom, t, m, aspect);
    vec2 d = uv - m;
    float dist = length(d);
    float r = 0.45;
    float w = 1.0 - smoothstep(0.0, r, dist);
    vec2 warp = normalize(d + 1e-5) * sin(dist * (18.0 + seg) - t * (2.0 + h1(seed)*3.0)) * 0.12 * w;
    duv += warp;
    duv = vec2(pingPong(duv.x + 0.05 * sin(t), 1.0), pingPong(duv.y + 0.05 * cos(t), 1.0));
    vec4 base = texture(samp, uv);
    vec4 fx = texture(samp, duv);
    float blend = 0.45 + 0.25 * sin(t * 0.8);
    vec4 mixedCol = mix(base, fx, blend);
    mixedCol.rgb *= 0.5 + 0.5 * sin(duv.xy * (2.0 + seg * 0.1) + t);
    vec4 c1 = sin(mixedCol * pingPong(t, 10.0));
    vec4 t0 = texture(samp, tc);
    vec4 c2 = sin(c1 * t0 * 0.8 * pingPong(t, 15.0));
    color = c2;
    color.a = 1.0;
}

```

Shader #141 - comb3 mouse

File: comb3_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear

mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle);
    float cosv = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cosv, -s, s, cosv) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float angle = atan(p.y, p.x);
    float radius = length(p);
    float seg = 6.28318530718 / segments;
    angle = mod(angle, seg);
    angle = abs(angle - seg * 0.5);
    vec2 r = vec2(cos(angle), sin(angle)) * radius;
    r.x /= aspect;
    return r + c;
}

vec2 fractalZoom(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 5; i++) {
        p = abs((p - c) * zoom) - 0.5 + c;
        p = rotateUV(p, t * 0.1, c, aspect);
    }
    return p;
}

void main() {
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    vec2 kaleidoUV = reflectUV(uv, 6.0, m, aspect);
    float zoom = 1.5 + 0.5 * sin(time_f * 0.5);
    kaleidoUV = fractalZoom(kaleidoUV, zoom, time_f, m, aspect);
    kaleidoUV = rotateUV(kaleidoUV, time_f * 0.2, m, aspect);
    vec4 kaleidoColor = texture(samp, kaleidoUV);
    float blendFactor = 0.6;
    vec4 blendedColor = mix(kaleidoColor, originalTexture, blendFactor);
    blendedColor.rgb *= 0.5 + 0.5 * sin(kaleidoUV.xyz + time_f);
    color = sin(blendedColor * pingPong(time_f, 10.0));
    vec4 t = texture(samp, tc);
    color = color * t * 0.8;
    color = sin(color * pingPong(time_f, 15.0));
    color.a = 1.0;
}
```

Shader #143 - comp mouse

File: comp_mouse.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Scanline overlay using modular y-coordinate - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
vec3 compositeEffect(vec2 uv) {
    float offset = 0.01;
    vec3 col;
    col.r = texture(samp, uv + vec2(offset, 0.0)).r;
    col.g = texture(samp, uv).g;
    col.b = texture(samp, uv - vec2(offset, 0.0)).b;
    float noise = fract(sin(dot(uv.xy, vec2(12.9898, 78.233))) * 43758.5453);
    col += noise * 0.05;
    float scanline = sin(uv.y * iResolution.y * 1.5) * 0.1;
    col -= scanline;
    float bleed = sin(uv.y * iResolution.y * 0.2 + time_f * 5.0) * 0.005;
    col.r += bleed * 0.002;
    col.b -= bleed * 0.002;
    return col;
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p = (tc - m) * ar;
    vec3 q = vec3(p, 1.0);
    float ax = 0.25 * sin(time_f * 0.7);
    float ay = 0.25 * cos(time_f * 0.6);
    float az = 0.15 * sin(time_f * 0.9);
    mat3 R = rotZ(az) * rotY(ay) * rotX(ax);
    q = R * q;
    float persp = 0.7;
    float zf = 1.0 / (1.0 + q.z * persp);
    vec2 uv = (q.xy * zf) / ar + m;
    uv = clamp(uv, 0.0, 1.0);
    vec3 col = compositeEffect(uv);
    color = vec4(col, 1.0);
}
```

Shader #170 - CursorMove

File: CursorMove.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float h1(float n){return fract(sin(n)*43758.5453123);}
vec2 h2(float n){return fract(sin(vec2(n, n+1.0))*vec2(43758.5453,22578.1459));}
void main(void) {
    vec2 uv = tc;
    float rate = 0.8;
    float t = time_f*rate;
    float t0 = floor(t);
    float a = fract(t);
    vec2 p0 = vec2(0.1) + h2(t0)*0.8;
    vec2 p1 = vec2(0.1) + h2(t0+1.0)*0.8;
    float w = a*a*(3.0-2.0*a);
    vec2 m_auto = mix(p0, p1, w);
    vec2 m = (iMouse.z>0.5 || iMouse.w>0.5) ? (iMouse.xy/iResolution) : m_auto;
    vec2 d = uv - m;
    float dist = length(d);
    float a1 = clamp(amp, 0.0, 1.0);
    float ua = clamp(uamp, 0.0, 1.0);
    float r = mix(0.06, 0.35, a1) * 1.5;
    float s = smoothstep(r, 0.0, dist);
    float k = 0.6 + 0.4 * ua;
    float swirl = (0.8 * ua + 0.2 * a1) * s * (r - dist) * 8.0;
    float ang = atan(d.y, d.x) + swirl;
    vec2 drot = vec2(cos(ang), sin(ang)) * dist;
    float lens = 1.0 - k * s * (1.0 - dist / r);
    vec2 warped = m + drot * lens;
    float wob = sin(time_f * 3.0 + dist * 20.0) * 0.005 * ua * s;
    vec2 n = normalize(drot + vec2(1e-6));
    warped += n * wob;
    vec4 warpedCol = texture(samp, clamp(warped, vec2(0.0), vec2(1.0)));
    color = warpedCol;
}
```

Shader #171 - CursorWarp2

File: CursorWarp2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
```

```

uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float h1(float n){return fract(sin(n)*43758.5453123);}
vec2 h2(float n){return fract(sin(vec2(n, n+1.0))*vec2(43758.5453,22578.1459));}
float tri(float x){float f=fract(x); return 1.0-abs(f*2.0-1.0);}
void main(void) {
    vec2 uv = tc;
    float rate = 0.8;
    float t = time_f*rate;
    float t0 = floor(t);
    float a = fract(t);
    vec2 p0 = vec2(0.1) + h2(t0)*0.8;
    vec2 p1 = vec2(0.1) + h2(t0+1.0)*0.8;
    float w = a*a*(3.0-2.0*a);
    vec2 m_auto = mix(p0, p1, w);
    vec2 m = (iMouse.z>0.5 || iMouse.w>0.5) ? (iMouse.xy/iResolution) : m_auto;
    vec2 d = uv - m;
    float dist = length(d);
    float a1 = clamp(amp, 0.0, 1.0);
    float ua = clamp(uamp, 0.0, 1.0);
    float r0 = mix(0.06, 0.35, a1) * 1.5;
    float pr = tri(time_f*0.6);
    float r = r0 * mix(0.75, 1.35, pr*(0.3+0.7*ua));
    float s = smoothstep(r, 0.0, dist);
    float k = 0.6 + 0.4 * ua;
    float swirl = (0.8 * ua + 0.2 * a1) * s * (r - dist) * 8.0;
    float ang = atan(d.y, d.x) + swirl;
    vec2 drot = vec2(cos(ang), sin(ang)) * dist;
    float lens = 1.0 - k * s * (1.0 - dist / r);
    vec2 warped = m + drot * lens;
    float wob = sin(time_f * 3.0 + dist * 20.0) * 0.005 * ua * s;
    vec2 n = normalize(drot + vec2(1e-6));
    warped += n * wob;
    vec4 warpedCol = texture(samp, clamp(warped, vec2(0.0), vec2(1.0)));
    color = warpedCol;
}

```

Shader #178 - cyclone zoom mouse

File: cyclone_zoom_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
void main(void){
    vec2 normCoord = gl_FragCoord.xy / iResolution.xy;
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);

```

```

    vec2 p = normCoord - m;
    p.x *= aspect;
    float eps = 1e-6;
    float r = length(p) + eps;
    float theta = atan(p.y, p.x);
    float base = 1.7;
    float period = log(base);
    float t = time_f * 0.5;
    float k = fract((log(r) - t) / period);
    float rw = exp(k * period);
    float twistAmount = 15.0;
    theta += (1.0 - clamp(rw, 0.0, 1.0)) * twistAmount * sin(time_f) + t * 1.57079632679;
    vec2 q = vec2(cos(theta), sin(theta)) * rw;
    q.x /= aspect;
    vec2 uv = fract(q + m);
    color = texture(samp, uv);
}

```

Shader #214 - diamond pattern 1

File: diamond_pattern_1.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.
Responds to mouse position for interactive control. warp coordinates based on mouse influence

Technique: Hermite smoothstep interpolation - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main() {
    vec2 uv = tc;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 d = uv - m;
    float dist = length(d);
    float pos = 1.0 + 6.0 * (1.0 - smoothstep(0.0, 0.35, dist)) + 0.25 * sin(time_f * 1.5);
    vec2 warp = d * sin(time_f * 0.8 + dist * 8.0) * 0.15 * pos;
    uv += warp;
    ivec2 coords = ivec2(uv * iResolution);
    vec4 origColor = texture(samp, uv);
    int x = coords.x;
    int y = coords.y;
    vec3 newColor = origColor.rgb;
    if ((x % 2) == 0) {
        if ((y % 2) == 0) {
            newColor.r = (1.0 - pos * origColor.r);
            newColor.b = (float(x + y) * pos) / 255.0;
        } else {
            newColor.r = (pos * origColor.r - float(y)) / 255.0;
            newColor.b = (float(x - y) * pos) / 255.0;
        }
    }
    } else {

```

```

    if ((y % 2) == 0) {
        newColor.r = (pos * origColor.r - float(x)) / 255.0;
        newColor.b = (float(x - y) * pos) / 255.0;
    } else {
        newColor.r = (pos * origColor.r - float(y)) / 255.0;
        newColor.b = (float(x + y) * pos) / 255.0;
    }
}
float temp = newColor.r;
newColor.r = newColor.b;
newColor.b = temp;
vec3 finalColor = (sin(time_f) > 0.0) ? vec3(1.0) - newColor : newColor;
color = sin(vec4(finalColor, 1.0) * pingPong(time_f, 10));
color.a = 1.0;
}

```

Shader #239 - drain mouse

File: drain_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
void main(void) {
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    float spiralSpeed = 2.0;
    float inwardSpeed = 0.5;
    float drainRadius = 8.0;
    float loopDuration = 20.0;
    float currentTime = mod(time_f, loopDuration);
    float progress = clamp(currentTime / loopDuration * inwardSpeed, 0.0, 1.0);
    vec2 normCoord = (tc - m) * ar;
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    angle += (1.0 - smoothstep(0.0, drainRadius, dist)) * currentTime * spiralSpeed;
    dist *= mix(1.0, 0.0, progress);
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * dist;
    vec2 uv = spiralCoord / ar + m;
    color = texture(samp, uv);
}

```

Shader #245 - dseek

File: dseek.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position for interactive control. Normalize mouse coordinates Calculate mouse movement

direction and speed Prevent division by zero

Technique: Driven by: time, mouse, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 mouse = iMouse.xy / iResolution.xy;
    vec2 prevMouse = iMouse.zw / iResolution.xy;
    vec2 mouseDelta = mouse - prevMouse;
    float mouseSpeed = length(mouseDelta);
    mouseDelta = normalize(mouseDelta + 0.000001);
    float dist = length(mouse - tc);
    float influence = exp(-dist * 2.0);
    influence *= mouseSpeed;
    vec2 displacement = mouseDelta * influence * 0.15;
    displacement += 0.03 * vec2(
        sin(time_f * 4.0 + tc.y * 12.0),
        cos(time_f * 4.0 + tc.x * 12.0)
    ) * influence;
    vec2 displacedTc = tc + sin(displacement * pingPong(time_f, 7.0));
    color = texture(samp, displacedTc);
}
```

Shader #295 - echo rgb mouse

File: echo_rgb_mouse.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position for interactive control.

Technique: Multi-tap convolution blur (5 texture samples) - Color quantization via floor rounding - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main() {
    vec2 uv = tc;
    vec2 m = (iMouse.z > 0.5 ? iMouse.xy : 0.5 * iResolution) / iResolution;
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 p = (uv - m);
    float r = length(p * ar);
    float mask = smoothstep(0.5, 0.0, r);
    float timeOffset = 0.05 * sin(time_f);
    vec3 redLayer = texture(samp, uv + vec2(timeOffset, 0.0)).rgb;
    vec3 greenLayer = texture(samp, uv).rgb;
```

```

    vec3 blueLayer = texture(samp, uv - vec2(timeOffset, 0.0)).rgb;
    bool strobe = mod(floor(time_f * 2.0), 2.0) > 0.5;
    vec3 rgbStrobe = vec3(
        strobe ? redLayer.r : blueLayer.r,
        greenLayer.g,
        strobe ? blueLayer.b : redLayer.b
    );
    vec2 dir = normalize(p + 1e-6);
    vec4 echoEffect = texture(samp, uv + dir * (0.015 + 0.005 * sin(time_f * 2.0)));
    vec4 fx = mix(vec4(rgbStrobe, 1.0), echoEffect, 0.5);
    vec4 base = texture(samp, uv);
    color = mix(base, fx, mask);
}

```

Shader #318 - elastic2

File: elastic2.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position for interactive control. Intense twirl effect Chaotic displacement Radial wave distortion

Technique: Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - Scanline overlay using modular y-coordinate - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec4 iMouse;
float random(vec2 st) {
    return fract(sin(dot(st.xy, vec2(12.9898,78.233))) * 43758.5453);
}
float noise(vec2 p) {
    vec2 ip = floor(p);
    vec2 fp = fract(p);
    float a = random(ip);
    float b = random(ip + vec2(1.0, 0.0));
    float c = random(ip + vec2(0.0, 1.0));
    float d = random(ip + vec2(1.0, 1.0));
    fp = fp * fp * (3.0 - 2.0 * fp);
    return mix(a, b, fp.x) + (c - a) * fp.y * (1.0 - fp.x) + (d - b) * fp.x * fp.y;
}
float fractalNoise(vec2 p) {
    float v = 0.0;
    v += noise(p*1.0)*1.0;
    v += noise(p*2.0)*0.5;
    v += noise(p*4.0)*0.25;
    v += noise(p*8.0)*0.125;
    return v / 1.875;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 uv = tc;
    vec2 mouse = iMouse.xy;
    float mouseInfluence = 0.0;
    if (iMouse.z > 0.0) {

```



```

        mouseInfluence = 1.0 - smoothstep(0.0, 0.5, length(mouse - uv));
    }
    float time = time_f * 2.0;
    vec2 tcFromCenter = uv - center;
    float distance = length(tcFromCenter);
    float angle = atan(tcFromCenter.y, tcFromCenter.x);
    float twirl = 10.0 * (1.0 - smoothstep(0.0, 0.8, distance)) + time * 5.0;
    angle += twirl * (0.5 + sin(time * 0.7) * 0.5 + mouseInfluence * 2.0);
    vec2 displacement = vec2(
        fractalNoise(uv * 10.0 + time * 2.0) - 0.5,
        fractalNoise(uv * 10.0 + time * 1.5) - 0.5
    ) * 0.2 * (0.5 + mouseInfluence * 2.0);
    float radialWave = sin(distance * 30.0 - time * 10.0) * 0.1;
    radialWave += sin(angle * 5.0 + time * 5.0) * 0.05;
    float radius = distance * (1.0 + radialWave * 2.0 + mouseInfluence);
    vec2 distortedTC = center + vec2(cos(angle), sin(angle)) * radius;
    distortedTC += displacement * (1.0 + sin(time * 10.0) * 0.5);
    float glitchIntensity = (sin(time * 3.0) * 0.5 + 0.5) * 0.3 + mouseInfluence * 0.5;
    if (random(vec2(time)) > 0.7) {
        distortedTC.x += (random(vec2(time)) - 0.5) * glitchIntensity;
        distortedTC.y += (random(vec2(time * 0.7)) - 0.5) * glitchIntensity;
    }
    vec4 color1 = texture(samp, distortedTC + vec2(glitchIntensity * 0.02, 0.0));
    vec4 color2 = texture(samp, distortedTC - vec2(glitchIntensity * 0.02, 0.0));
    color = vec4(color1.r, color2.g, color1.b, 1.0);
    float scanLine = sin(uv.y * 800.0 + time * 10.0) * 0.1;
    color.rgb += scanLine * glitchIntensity;
    vec2 chromaOffset = vec2(
        fractalNoise(uv * 50.0 + time) * 0.02,
        fractalNoise(uv * 50.0 + time * 1.2) * 0.02
    );
    color.r = texture(samp, distortedTC + chromaOffset).r;
    color.b = texture(samp, distortedTC - chromaOffset).b;
    color.rgb = mix(color.rgb, smoothstep(0.0, 1.0, color.rgb * 1.2), 0.5);
    if (random(vec2(time * 0.3)) > 0.97) {
        color.rgb *= 0.1;
    }
    float vignette = 1.0 - smoothstep(0.4, 0.8, distance);
}

```

Shader #319 - elastic mouse

File: elastic_mouse.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec4 iMouse;
float random(vec2 st){return fract(sin(dot(st,vec2(12.9898,78.233)))*43758.5453);}
float noise(vec2 p){

```

```

    vec2 ip=floor(p),fp=fract(p);
    float a=random(ip);
    float b=random(ip+vec2(1.0,0.0));
    float c=random(ip+vec2(0.0,1.0));
    float d=random(ip+vec2(1.0,1.0));
    fp=fp*fp*(3.0-2.0*fp);
    return mix(a,b,fp.x)+(c-a)*fp.y*(1.0-fp.x)+(d-b)*fp.x*fp.y;
}
float fractalNoise(vec2 p){
    float v=0.0;
    v+=noise(p*1.0)*1.0;
    v+=noise(p*2.0)*0.5;
    v+=noise(p*4.0)*0.25;
    v+=noise(p*8.0)*0.125;
    return v/1.875;
}
void main(){
    vec2 center=vec2(0.5);
    vec2 uv=tc;
    vec2 m=(iMouse.z>0.0)?iMouse.xy:center;
    float mouseInfluence=1.0-smoothstep(0.0,0.5,length(m-uv));
    float t=time_f*2.0;
    vec2 d=uv-center;
    float dist=length(d);
    float ang=atan(d.y,d.x);
    float twirl=10.0*(1.0-smoothstep(0.0,0.8,dist))+t*5.0;
    ang+=twirl*(0.5+sin(t*0.7)*0.5+mouseInfluence*2.0);
    vec2 disp=vec2(fractalNoise(uv*10.0+t*2.0)-0.5,fractalNoise(uv*10.0+t*1.5)-0.5)*0.2*(0.5+
    ↪ mouseInfluence*2.0);
    float radialWave=sin(dist*30.0-t*10.0)*0.1+sin(ang*5.0+t*5.0)*0.05;
    float radius=dist*(1.0+radialWave*2.0+mouseInfluence);
    vec2 tc2=center+vec2(cos(ang),sin(ang))*radius;
    tc2+=disp*(1.0+sin(t*10.0)*0.5);
    float rgate=random(vec2(floor(t*3.0)));
    float gInt=(sin(t*3.0)*0.5+0.5)*0.3+mouseInfluence*0.5;
    if(rgate>0.7){
        tc2.x+= (random(vec2(t))-0.5)*gInt;
        tc2.y+= (random(vec2(t*0.7))-0.5)*gInt;
    }
    tc2=clamp(tc2,0.0,1.0);
    vec2 chroma=vec2(fractalNoise(uv*50.0+t)*0.02,fractalNoise(uv*50.0+t*1.2)*0.02);
    vec4 cR=texture(samp,clamp(tc2+vec2(gInt*0.02,0.0),0.0,1.0));
    vec4 cG=texture(samp,clamp(tc2-vec2(gInt*0.02,0.0),0.0,1.0));
    vec3 col=vec3(cR.r,cG.g,cR.b);
    float scan=sin(uv.y*800.0+t*10.0)*0.1;
    col+=scan*gInt;
    col.r=texture(samp,clamp(tc2+chroma,0.0,1.0)).r;
    col.b=texture(samp,clamp(tc2-chroma,0.0,1.0)).b;
    col=mix(col,smoothstep(0.0,1.0,col*1.2),0.5);
    if(random(vec2(floor(t*0.3)))>0.97) col*=0.1;
    float vignette=1.0-smoothstep(0.4,0.8,dist);
    color=vec4(col*vignette,1.0);
}

```

Shader #340 - fish pos mouse

File: fish_pos_mouse.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control.

Technique: Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : (length * 2.0 - m);
}
float h1(float n){ return fract(sin(n)*43758.5453123); }
vec2 h2(float n){ return fract(sin(vec2(n,n+1.0))*vec2(43758.5453,22578.1459)); }
void main(void){
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec2 m = iMouse.xy / iResolution;
    bool hasMouse = iMouse.z > 0.5 || iMouse.w > 0.5;
    float rate = 0.8;
    float t = time_f * rate;
    float t0 = floor(t);
    float a = fract(t);
    float w = a*a*(3.0-2.0*a);
    vec2 p0 = vec2(0.15) + h2(t0) * 0.7;
    vec2 p1 = vec2(0.15) + h2(t0 + 1.0) * 0.7;
    vec2 autoCenter = mix(p0, p1, w);
    vec2 center = hasMouse ? m : autoCenter;
    vec2 uv = tc - center;
    float lenv = length(uv);
    float factor = sqrt(lenv) * 0.5;
    float s = 1.0 + sin(factor * time_t);
    uv *= s;
    color = texture(samp, uv + center);
}
```

Shader #342 - fisheye mouse

File: fisheye_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Driven by: time, mouse, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
```

```

    vec2 p = (tc - m) * ar;
    float ax = 0.28 * sin(time_f * 0.7);
    float ay = 0.28 * cos(time_f * 0.6);
    float az = 0.5 * time_f;
    mat3 R = rotZ(az) * rotY(ay) * rotX(ax);
    vec3 q3 = R * vec3(p, 1.0);
    float persp = 0.7;
    float zf = 1.0 / (1.0 + q3.z * persp);
    vec2 q = q3.xy * zf;
    float d = length(q);
    float w = 1.0 - smoothstep(0.0, 0.55, length((tc - m) * ar));
    float s = 0.5 + 4.0 * sin(time_f);
    float ang = atan(q.y, q.x);
    float radius = d * (1.0 + s * w * (d * d));
    vec2 r = vec2(cos(ang), sin(ang)) * sin(radius * time_f);
    vec2 uv = r / ar + m;
    uv = clamp(uv, 0.0, 1.0);
    color = texture(samp, uv);
}

```

Shader #351 - frac wrap

File: frac-wrap.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;

```

```

    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;

```

```

    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    vec4 baseTex = texture(samp, uv);
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = pingPong(time_f * PI/4, 3.0) * 0.65;
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + pingPong(time_f * PI/2, 3.0) * 0.6);
    ↵ ;
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + pingPong(time_f * PI, 5.0) * 1.2));

```

```

ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(pingPong(time_f * PI, 5.0) * 2.0 + rD * 28.0 + k *
↳ 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #353 - frac cosine

File: frac_cosine.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending -
Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 rainbow(float t){
    float hue = mod(t, 1.0) * 6.0;
    float c = 1.0;
    float x = 1.0 - abs(mod(hue, 2.0) - 1.0);
    vec3 rgb = (hue < 1.0) ? vec3(c, x, 0.0) :
                (hue < 2.0) ? vec3(x, c, 0.0) :
                (hue < 3.0) ? vec3(0.0, c, x) :
                (hue < 4.0) ? vec3(0.0, x, c) :
                (hue < 5.0) ? vec3(x, 0.0, c) :
                vec3(c, 0.0, x);
    return rgb;
}

```

```

}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {

```



```

    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uvRad = tc * 2.0 - 1.0;
    uvRad.x *= aspect;
    float r0 = pingPong(sin(length(uvRad) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - 0.25, r0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float baseF = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float periodF = log(baseF) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / periodF);
    float rw = exp(k * periodF);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dirRGB = normalize(pwrap + 1e-6);
    vec2 off = dirRGB * (0.0015 + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);

```

```

vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
vec2 pS = (tc - m) * ar;
float baseS = 1.68;
float periodS = log(baseS);
float tS = time_f * 0.5;
float rS = length(pS) + 1e-6;
float aS = atan(pS.y, pS.x) + tS * 0.35;
float kS = fract((log(rS) - tS) / periodS);
float rwS = exp(kS * periodS);
vec2 zS = vec2(cos(aS), sin(aS)) * rwS;
vec2 uvS = fract(zS / ar + m);
vec2 dirS = (uvS - m);
dirS.x *= aspect;
float distS = length(dirS);
float angleS = atan(dirS.y, dirS.x);
float frequency = 1.0;
float amplitude = 3.0;
float spiralTime = sin(time_f * frequency) * amplitude;
float scale = 1.0;
float gradient_pos = mod(distS * scale + (angleS + spiralTime) / (2.0 * PI), 1.0);
vec3 color_gradient = rainbow(gradient_pos);
vec4 ctxS = texture(samp, uvS);
float time_t = mod(time_f, 10.0) + 2.0;
vec3 spirallayer = mix(cos(ctxS.rgb * time_t), color_gradient, 0.5);
float spiralMask = smoothstep(0.12, 0.65, distS);
spiralMask *= 0.6 + 0.4 * sin(time_f * 0.3 + rS * 8.0);
spiralMask = clamp(spiralMask, 0.0, 1.0);
vec3 combined = mix(outCol, spirallayer, spiralMask);
combined *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = combined * combined * 0.18 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.12;
combined += bloom;
combined = mix(combined, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
combined = clamp(combined, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, combined, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #355 - frac mirror mouse zoom

File: frac_mirror_mouse_zoom.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<5;i++){
        p = abs((p - c) * zoom) - 0.5 + c;
        p = rotateUV(p, t*0.1, c, aspect);
    }
    return p;
}
void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    vec2 kUV = reflectUV(uv, 6.0, m, aspect);
    float foldZoom = 1.5 + (0.5 + amp_low * 0.4) * sin(time_f * 0.5);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * 0.2, m, aspect);
    vec2 p = (kUV - m) * ar;
    float base = 1.72;
    float period = log(base);
    float tz = time_f * 0.5;
    float r = length(p) + 1e-6;
    float ang = atan(p.y, p.x) + tz * 0.3;
    float k = fract((log(r) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;

```

```

    vec2 zoomUV = fract(pwrap / ar + m);
    vec4 kaleidoColor = texture(samp, zoomUV);
    float blendFactor = 0.6;
    vec4 blendedColor = mix(kaleidoColor, originalTexture, blendFactor);
    blendedColor.rgb *= 0.5 + 0.5 * sin(zoomUV.xy * time_f);
    color = sin(blendedColor * pingPong(time_f, 10.0));
    vec4 t = texture(samp, tc);
    color = color * t * 0.8;
    color = sin(color * pingPong(time_f, 15.0));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    if(color[0] > 0.8)
        color[0] = 0.8;
    if(color[1] > 0.8)
        color[1] = 0.8;
    if(color[2] > 0.8)
        color[2] = 0.8;
    if(color[0] < 0.1)
        color[0] = 0.1;
    if(color[1] < 0.1)
        color[1] = 0.1;
    if(color[2] < 0.1)
        color[2] = 0.1;
    color.a = 1.0;
}

```

Shader #356 - frac mouse z

File: frac_mouse_z.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);

```

```

    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}

vec2 warp(vec2 uv, float t){
    float s1=sin((uv.x+uv.y)*8.0+t*0.8);
    float s2=cos((uv.x-uv.y)*9.0-t*0.6);
    float s3=sin((uv.x*2.0)*5.0+t*1.1)+cos((uv.y*2.0)*7.0-t*1.3);
    return uv + 0.02*vec2(s1+s3*0.5, s2-s3*0.5);
}

vec2 logPolarWrap(vec2 uv, vec2 c, vec2 ar, float tz, float base){
    vec2 p=(uv-c)*ar;
    float r=length(p)+1e-6;
    float a=atan(p.y,p.x)+tz*0.3;
    float per=log(base);
    float k=fract((log(r)-tz)/per);
    float rw=exp(k*per);
    vec2 q=vec2(cos(a),sin(a))*rw;
    return fract(q/ar+c);
}

vec2 fold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p=uv;
    for(int i=0;i<7;i++){
        p = abs((p - c) * zoom) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.05, c, aspect);
        p = warp(p, t + float(i)*0.37);
    }
    return p;
}

void main(){
    float aspect=iResolution.x/iResolution.y;
    vec2 ar=vec2(aspect,1.0);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 uv=tc;
    vec4 baseTex=texture(samp, tc);
    float seg=mix(5.0, 12.0, 0.5+0.5*sin(time_f*0.35));
    vec2 kUV=reflectUV(uv, seg, m, aspect);
    float mouseAmp=(iMouse.z>0.5)?(0.3+0.4*abs(sin((iMouse.x+iMouse.y)*0.001+time_f*0.2))):0.45;
    float foldZoom=1.35 + (0.45 + amp_low * 0.4)*sin(time_f*0.45) + mouseAmp*0.25;
    kUV=fold(kUV, foldZoom, time_f, m, aspect);
    kUV=rotateUV(kUV, time_f*0.22, m, aspect);
    float tz=time_f*0.6;
    float base=1.72 + 0.2*sin(time_f*0.2);

```

```

vec2 zp0=logPolarWrap(kUV, m, ar, tz, base);
vec2 zp1=logPolarWrap(kUV+0.003*vec2(1.0,0.0), m, ar, tz+0.08, base);
vec2 zp2=logPolarWrap(kUV+0.003*vec2(0.0,1.0), m, ar, tz+0.16, base);
vec2 zpw0=warp(zp0, time_f);
vec2 zpw1=warp(zp1, time_f+0.7);
vec2 zpw2=warp(zp2, time_f+1.4);
vec4 s0=texture(samp, zpw0);
vec4 s1=texture(samp, zpw1);
vec4 s2=texture(samp, zpw2);
vec3 chroma=vec3(s0.r, s1.g, s2.b);
float ring=sin(10.0*length((kUV-m)*ar)-time_f*3.0)*0.5+0.5;
float swirl=sin(6.2831853*(zpw0.x+zpw0.y)+time_f*1.4)*0.5+0.5;
float mask=smoothstep(0.0,1.0,ring*0.6+swirl*0.4);
vec3 hsv=vec3(fract(time_f*0.07 + ring*0.25 + swirl*0.25), 0.8, 0.9);
vec3 rainbow=hsv2rgb(hsv);
vec3 kaleido = mix(chroma, chroma*rainbow, 0.65);
vec4 kaleidoTex=vec4(kaleido,1.0);
float blendFactor=0.55 + 0.15*sin(time_f*0.33);
vec4 mixA = mix(kaleidoTex, baseTex, blendFactor*(0.6+0.4*mask));
vec3 pulse = 0.5 + 0.5*sin(vec3(1.7,2.3,3.1)*time_f + (zpw0.yyx+zpw0.yxx)*9.0);
vec4 glow = vec4(mixA.rgb * pulse, 1.0);
vec4 mod1 = sin(glow * (1.0 + 0.7*pingPong(time_f, 9.0)));
vec4 mod2 = sin(mod1 * (1.0 + 0.9*pingPong(time_f, 13.0)));
vec4 t = texture(samp, tc);
vec4 outc = mod2 * t * (0.7 + 0.3*mask);
outc.rgb = clamp(outc.rgb, 0.08, 0.92);
color = vec4(outc.rgb, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
color.rgb *= 1.0 + _ab * 0.6;
color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #357 - frac mouse z2

File: frac_mouse_z2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){

```

```

    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}

vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    float seg = 6.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
    vec2 kUV = reflectUV(uv, seg, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    vec2 p = (kUV - m) * ar;
    float base = 1.82 + 0.18*sin(time_f*0.2);
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float r = length(p) + 1e-6;
    float ang = atan(p.y, p.x) + tz * 0.35 + 0.35*sin(r*9.0 + time_f*0.8);
    float k = fract((log(r) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap*1.045) / ar + m);
    vec2 u2 = fract((pwrap*0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
    float hue = fract(ang*0.15 + time_f*0.08 + k*0.5);

```

```

float sat = 0.8 - 0.2*cos(time_f*0.7 + r*6.0);
float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
vec3 tint = hsv2rgb(vec3(hue, sat, val));
float ring = smoothstep(0.0, 0.7, sin(log(r+1e-3)*8.0 + time_f*1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + r*18.0 + k*12.0);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
vign = mix(0.85, 1.2, vign);
float blendFactor = 0.58;
float rC = texture(samp, u0 + off).r;
float gC = texture(samp, u1).g;
float bC = texture(samp, u2 - off).b;
vec3 kaleidoRGB = vec3(rC, gC, bC);
vec4 kaleidoColor = vec4(kaleidoRGB, 1.0) * vec4(tint, 1.0);
vec4 merged = mix(kaleidoColor, originalTexture, blendFactor);
merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
vec3 outCol = merged.rgb + bloom;
float wob = 0.9 + 0.1*sin(time_f + r*12.0 + k*9.0);
outCol *= wob;
vec4 t = texture(samp, tc);
outCol = mix(outCol, outCol*t.rgb, 0.8);
outCol = sin(outCol * (0.5 + 0.5*pingPong(time_f, 12.0)));
outCol = clamp(outCol, vec3(0.08), vec3(0.96));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
outCol *= 1.0 + _ab * 0.6;
outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
↳ 0.0, 1.0) * 0.25), _ab);
color = vec4(outCol, 1.0);
}

```

Shader #359 - frac shader01

File: frac_shader01.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}

```



```

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}

vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    vec2 d = normalize(vec2(cos(t*0.27), sin(t*0.31)));
    float s = dot(p, d);
    float band = 0.5 + 0.5*sin(s*6.28318530718*0.35 + t*0.9);
    float h = fract(s*0.22 + t*0.07 + 0.15*sin(t*0.33));
    float S = 0.75 + 0.25*sin(t*0.21 + s*2.0);
    float V = 0.75 + 0.25*band;
    vec3 base = hsv2rgb(vec3(h, S, V));
    float edge = smoothstep(0.2, 0.8, band);
    return mix(base*0.6, base, edge);
}

void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    float seg = 6.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
    vec2 kUV = reflectUV(uv, seg, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    vec2 p = (kUV - m) * ar;
    float base = 1.82 + 0.18*sin(time_f*0.2);
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float r = length(p) + 1e-6;
}

```

```

float ang = atan(p.y, p.x) + tz * 0.35 + 0.35*sin(r*9.0 + time_f*0.8);
float k = fract((log(r) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap*1.045) / ar + m);
vec2 u2 = fract((pwrap*0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
float hue = fract(ang*0.15 + time_f*0.08 + k*0.5);
float sat = 0.8 - 0.2*cos(time_f*0.7 + r*6.0);
float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
vec3 tint = hsv2rgb(vec3(hue, sat, val));
float ring = smoothstep(0.0, 0.7, sin(log(r+1e-3)*8.0 + time_f*1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + r*18.0 + k*12.0);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
vign = mix(0.85, 1.2, vign);
float blendFactor = 0.58;
float rC = texture(samp, u0 + off).r;
float gC = texture(samp, u1).g;
float bC = texture(samp, u2 - off).b;
vec3 kaleidoRGB = vec3(rC, gC, bC);
vec4 kaleidoColor = vec4(kaleidoRGB, 1.0) * vec4(tint, 1.0);
vec4 merged = mix(kaleidoColor, originalTexture, blendFactor);
merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
vec3 outCol = merged.rgb + bloom;
float wob = 0.9 + 0.1*sin(time_f + r*12.0 + k*9.0);
outCol *= wob;
vec3 grad = movingGradient(tc, m, time_f, aspect);
float gradAmt = 0.35 + 0.25*sin(time_f*0.5 + r*5.0 + k*9.0);
vec3 screenBlend = 1.0 - (1.0 - outCol) * (1.0 - grad);
outCol = mix(outCol, screenBlend, gradAmt);
vec4 t = texture(samp, tc);
outCol = mix(outCol, outCol*t.rgb, 0.8);
outCol = sin(outCol * (0.5 + 0.5*pingPong(time_f, 12.0)));
outCol = clamp(outCol, vec3(0.08), vec3(0.96));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
outCol *= 1.0 + _ab * 0.6;
outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
↵ 0.0, 1.0) * 0.25), _ab);
color = vec4(outCol, 1.0);
}

```

Shader #360 - frac shader01 dark

File: frac_shader01_dark.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    vec2 d = normalize(vec2(cos(t*0.27), sin(t*0.31)));
    float s = dot(p, d);
    float band = 0.5 + 0.5*sin(s*6.28318530718*0.35 + t*0.9);
    float h = fract(s*0.22 + t*0.07 + 0.15*sin(t*0.33));
    float S = 0.75 + 0.25*sin(t*0.21 + s*2.0);
    float V = 0.6 + 0.125*band;
    vec3 base = hsv2rgb(vec3(h, S, V));
    float edge = smoothstep(0.2, 0.8, band);
    return mix(base*0.6, base, edge);
}
vec2 triFold(vec2 uv, vec2 c, float aspect){

```

```

    vec2 p = (uv - c) * vec2(aspect, 1.0);
    vec2 b1 = vec2(1.0, 0.0);
    vec2 b2 = vec2(0.5, 0.86602540378);
    float u = dot(p, b1);
    float v = dot(p, b2);
    vec2 f = fract(vec2(u, v));
    if(f.x + f.y > 1.0) f = vec2(1.0) - f;
    vec2 q = f.x*b1 + f.y*b2;
    q.x /= aspect;
    return q + c;
}
float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}
float luma(vec3 c){ return dot(c, vec3(0.299,0.587,0.114)); }
vec3 bilateral9(vec2 uv, float radiusScale, float sigma_r){
    vec2 texel = 1.0 / iResolution;
    vec2 o = texel * radiusScale;
    vec3 c0 = texture(samp, uv).rgb;
    float L0 = luma(c0);
    vec2 offs[9] = vec2[](
        vec2(0,0),
        vec2( o.x, 0), vec2(-o.x, 0),
        vec2(0, o.y), vec2(0, -o.y),
        vec2( o.x, o.y), vec2(-o.x, o.y),
        vec2( o.x, -o.y), vec2(-o.x, -o.y)
    );
    float wsum = 0.0;
    vec3 acc = vec3(0.0);
    for(int i=0;i<9;i++){
        vec3 c = texture(samp, uv + offs[i]).rgb;
        float dl = luma(c) - L0;
        float wr = exp(-(dl*dl)/(2.0*sigma_r*sigma_r));
        float dsq = dot(offs[i]/texel, offs[i]/texel);
        float ws = exp(-dsq/2.0);
        float w = wr*ws;
        acc += c*w;
        wsum += w;
    }
    return acc / max(wsum, 1e-6);
}
float localVar9(vec2 uv, float radiusScale){
    vec2 texel = 1.0 / iResolution;
    vec2 o = texel * radiusScale;
    float m = 0.0;
    float s = 0.0;
    int n = 0;
    for(int y=-1;y<=1;y++){
        for(int x=-1;x<=1;x++){
            vec3 c = texture(samp, uv + vec2(x,y)*o).rgb;
            float L = luma(c);
            m += L;
            s += L*L;
            n++;
        }
    }
    m /= float(n);
    s /= float(n);
    return max(s - m*m, 0.0);
}
void main(){

```

```

float aspect = iResolution.x / iResolution.y;
vec2 ar = vec2(aspect, 1.0);
vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
vec2 uv = tc;
vec4 originalTexture = texture(samp, tc);
float seg = 3.0;
vec2 kUV = reflectUV(uv, seg, m, aspect);
kUV = triFold(kUV, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = triFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if(q.y > q.x) q = q.yx;
float base = 1.82 + 0.18*sin(time_f*0.2);
float period = log(base);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + time_f*0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap*1.045) / ar + m);
vec2 u2 = fract((pwrap*0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
float hue = fract(ang*0.25 + time_f*0.08 + k*0.5);
float sat = 0.75 - 0.25*cos(time_f*0.7 + rD*10.0);
float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
vec3 tint = hsv2rgb(vec3(hue, sat, val));
float ring = smoothstep(0.0, 0.7, sin(log(rD+1e-3)*9.5 + time_f*1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + rD*28.0 + k*12.0);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
vign = mix(0.85, 1.2, vign);
float blendFactor = 0.58;
float v0 = localVar9(u0, 1.0);
float v1 = localVar9(u1, 1.0);
float v2 = localVar9(u2, 1.0);
float pix = (v0+v1+v2)/3.0;
float sStrong = smoothstep(0.02, 0.20, pix);
float radA = mix(0.75, 2.25, sStrong);
float sigma_r = mix(0.10, 0.22, sStrong);
vec3 bc0 = bilateral9(u0 + off, radA, sigma_r);
vec3 bc1 = bilateral9(u1, radA, sigma_r);
vec3 bc2 = bilateral9(u2 - off, radA, sigma_r);
vec3 kaleidoRGB = vec3(bc0.r, bc1.g, bc2.b);
vec4 kaleidoColor = vec4(kaleidoRGB, 1.0) * vec4(tint, 1.0);
vec4 merged = mix(kaleidoColor, originalTexture, blendFactor);
merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
vec3 outCol = merged.rgb + bloom;
float wob = 0.9 + 0.1*sin(time_f + rD*14.0 + k*9.0);
outCol *= wob;
vec3 grad = movingGradient(tc, m, time_f, aspect) * 0.5;
float gradBase = 0.35 + 0.25*sin(time_f*0.5 + rD*7.0 + k*9.0);
float gradAmt = 0.5 * mix(gradBase*0.5, 0.95, sStrong);
vec3 screenBlend = 1.0 - (1.0 - outCol) * (1.0 - grad);
outCol = mix(outCol, screenBlend, gradAmt);
vec4 t = texture(samp, tc);
outCol = mix(outCol, outCol*t.rgb, 0.8);

```

```

    outCol = sin(outCol * (0.5 + 0.5*pingPong(time_f, 12.0)));
    outCol = clamp(outCol, vec3(0.08), vec3(0.96));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    outCol *= 1.0 + _ab * 0.6;
    outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
    ↪ 0.0, 1.0) * 0.25), _ab);
    color = vec4(outCol, 1.0);
}

```

Shader #363 - frac shader02 dmd

File: frac_shader02_dmd.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

```

```

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}

vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    vec2 d = normalize(vec2(cos(t*0.27), sin(t*0.31)));
    float s = dot(p, d);
    float band = 0.5 + 0.5*sin(s*6.28318530718*0.35 + t*0.9);
    float h = fract(s*0.22 + t*0.07 + 0.15*sin(t*0.33));
    float S = 0.75 + 0.25*sin(t*0.21 + s*2.0);
    float V = 0.75 + 0.25*band;
    vec3 base = hsv2rgb(vec3(h, S, V));
    float edge = smoothstep(0.2, 0.8, band);
    return mix(base*0.6, base, edge);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}

void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
    vec2 kUV = reflectUV(uv, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = p;
    q = abs(q);
    if(q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18*sin(time_f*0.2);
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + time_f*0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
}

```

```

    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap*1.045) / ar + m);
    vec2 u2 = fract((pwrap*0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
    float hue = fract(ang*0.25 + time_f*0.08 + k*0.5);
    float sat = 0.75 - 0.25*cos(time_f*0.7 + rD*10.0);
    float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
    vec3 tint = hsv2rgb(vec3(hue, sat, val));
    float ring = smoothstep(0.0, 0.7, sin(log(rD+1e-3)*9.5 + time_f*1.2));
    float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + rD*28.0 + k*12.0);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
    vign = mix(0.85, 1.2, vign);
    float blendFactor = 0.58;
    float rC = texture(samp, u0 + off).r;
    float gC = texture(samp, u1).g;
    float bC = texture(samp, u2 - off).b;
    vec3 kaleidoRGB = vec3(rC, gC, bC);
    vec4 kaleidoColor = vec4(kaleidoRGB, 1.0) * vec4(tint, 1.0);
    vec4 merged = mix(kaleidoColor, originalTexture, blendFactor);
    merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
    vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
    vec3 outCol = merged.rgb + bloom;
    float wob = 0.9 + 0.1*sin(time_f + rD*14.0 + k*9.0);
    outCol *= wob;
    vec3 grad = movingGradient(tc, m, time_f, aspect);
    float gradAmt = 0.35 + 0.25*sin(time_f*0.5 + rD*7.0 + k*9.0);
    vec3 screenBlend = 1.0 - (1.0 - outCol) * (1.0 - grad);
    outCol = mix(outCol, screenBlend, gradAmt);
    vec4 t = texture(samp, tc);
    outCol = mix(outCol, outCol*t.rgb, 0.8);
    outCol = sin(outCol * (0.5 + 0.5*pingPong(-time_f * PI, 5.0)));
    outCol = clamp(outCol, vec3(0.08), vec3(0.96));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    outCol *= 1.0 + _ab * 0.6;
    outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
    ↪ 0.0, 1.0) * 0.25), _ab);
    color = mix(t, vec4(outCol, 1.0), -pingPong(time_f * PI, 5.0));
}

```

Shader #364 - frac shader02 dmd2

File: frac_shader02_dmd2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;

```



```

uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
}

```

```

    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = abs(p);
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = length(uv);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * sin(time_f * 0.2);

```

```

float period = log(base);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #366 - frac shader02 dmd3

File: frac_shader02_dmd3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;

```

```

uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
}

```

```

    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = abs(p);
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = length(uv);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * sin(time_f * 0.2);
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);

```

```

float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #367 - frac shader02 dmd4

File: frac_shader02_dmd4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);

```

```

    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;

```

```

    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = abs(p);
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = length(uv);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * sin(time_f * 0.2);
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    rw = pingPong(sin(rw * (time_f * PI)), 5.0);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
}

```



```

vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * (PI * time_f), 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #368 - frac shader02 dmd5

File: frac_shader02_dmd5.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);

```

```

    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

```

```

}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}
float diamondRadius(vec2 p) {
    p = abs(p);
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = length(uv);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * sin(time_f * 0.2);
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
    ↵ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));

```

```

    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    pulse = (pulse * PI) * pingPong(time_f, 5.0);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
    vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
    outCol += bloom;
    outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #369 - frac shader02 dmd6i

File: frac_shader02_dmd6i.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {

```

```

    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;

```

```

    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);

```

```

vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #370 - frac shader02 dmd6i1

File: frac_shader02_dmd6i1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
}

```

```

    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);

```



```

    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * (PI/2)), 5.0);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);

```

```

vec3 outCol = vec3(kaleidoRGB.r * pingPong(time_f * PI, 2.0), kaleidoRGB.g * pingPong(time_f * PI,
↪ 2.0), kaleidoRGB.b * pingPong(time_f * PI, 2.0));
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI*2, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #376 - frac shader02 dmd6i bubble

File: frac_shader02_dmd6i_bubble.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
}

```

```

    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);

```

```

    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}
float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    float audio = clamp(amp * 0.8 + uamp * 0.6, 0.0, 5.0);
    float audioNorm = clamp(audio * 0.5, 0.0, 2.5);
    float t = time_f * (1.0 + audioNorm * 0.25);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 rel = tc - m;
    float dist = length(rel);
    float bubSize = 0.38 + 0.12 * audioNorm;
    float bubEdge = 0.28 + 0.10 * audioNorm;
    float bubMask = 1.0 - smoothstep(bubSize, bubSize + bubEdge, dist);
    float zoomBase = 1.3 + 0.7 * audioNorm;
    float zoomAnim = 0.5 + 0.5 * sin(t * 1.6 + audioNorm * 2.3);
    float zoom = mix(1.0, zoomBase, zoomAnim);
    vec2 tcZoom = m + rel / zoom;
    vec2 tcF = mix(tc, tcZoom, bubMask);
    vec4 baseTex = texture(samp, tcF);
    vec2 uv = tcF * 2.0 - 1.0;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * t), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec3 baseCol = preBlendColor(tcF);
    float seg = 4.0 + 2.0 * sin(t * 0.33);
    vec2 kUV = reflectUV(tcF, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(t * 0.42);
    kUV = fractalFold(kUV, foldZoom, t, m, aspect);
    kUV = rotateUV(kUV, t * 0.23, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(t * 0.2) * (PI * t), 5.0);
    float period = log(base) * pingPong(t * PI, 5.0);
    float tz = t * 0.65;
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + t * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
}

```

```

vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * (0.0015 + 0.001 * sin(t * 1.3)) * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tcF - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float angDir = atan(p.y, p.x) / (2.0 * PI);
angDir = angDir * 0.5 + 0.5;
float radN = clamp(length(p) * 0.9, 0.0, 1.0);
float hueShift = pingPong(time_f * 0.18 + audioNorm * 0.12, 1.0);
float hue = fract(angDir + (radN - 0.5) * 0.4 + (hueShift - 0.5) * 0.6);
float sat = 0.4 + 0.4 * radN;
float val = 0.45 + 0.45 * (1.0 - radN);
vec3 gradCol = hsv2rgb(vec3(hue, sat, val));
float gMix = (0.18 + 0.25 * audioNorm) * (0.3 + 0.7 * bubMask);
vec3 outCol = mix(kaleidoRGB, kaleidoRGB * gradCol, gMix);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + t * 1.2));
ring = ring * pingPong((t * PI), 5.0);
float pulse = 0.5 + 0.5 * sin(t * 2.0 + rD * 28.0 + k * 12.0 + audioNorm * 3.0);
outCol *= (0.60 + 0.20 * ring) * (0.80 + 0.10 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.08;
outCol += bloom * 0.3;
outCol = mix(outCol, baseTex.rgb, 0.18 + audioNorm * 0.04);
float rim = smoothstep(bubSize + 0.02, bubSize - 0.02, dist);
vec3 bubbleHighlight = hsv2rgb(vec3(hue, 0.7, 0.9)) * rim * (0.10 + 0.20 * audioNorm);
outCol += bubbleHighlight * bubMask * 0.5;
outCol = clamp(outCol, vec3(0.05), vec3(0.90));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.7 + 0.1);
float maxC = max(max(finalRGB.r, finalRGB.g), finalRGB.b);
float targetMax = 0.97;
if (maxC > targetMax) finalRGB *= targetMax / maxC;
finalRGB = clamp(finalRGB, vec3(0.03), vec3(0.97));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #378 - frac shader02 dmd6i wrap

File: frac_shader02_dmd6i_wrap.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;

```

```

uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));

```

```

    c = mix(vec3(1), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec2 xuv = 1.0 - abs(1.0 - 2.0 * tc);
    xuv = xuv - floor(xuv);
    vec4 baseTex = texture(samp, xuv);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
}

```

```

    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
    ↪ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
    vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
    outCol += bloom;
    outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #380 - frac shader02 dmd7i

File: frac_shader02_dmd7i.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;

```



```

uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

```

```

}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p * (pingPong(time_f * PI, 5.0))));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
}

```

```

float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #381 - frac shader02 dmd8i

File: frac_shader02_dmd8i.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;

```

```

uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
}

```

```

    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p * (pingPong(time_f * PI, 5.0))));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(pingPong(time_f * PI, 25.0) * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);

```

```

float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #382 - frac shader02 dmd9i

File: frac_shader02_dmd9i.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {

```

```

    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / sin(segments * pingPong(time_f, 5.0));
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (sin(pingPong(zoom * time_f * PI, 5.0)) + 0.15 * sin(t * (0.35 + amp_low *
        ↪ 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;

```

```

    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p * (pingPong(time_f * PI, 5.0))));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(pingPong(time_f * PI, 25.0) * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);

```



```

vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #396 - frac shader02 octo

File: frac_shader02_octo.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}

```

```

vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;

```

```

    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float octaRadius(vec2 p) {
    p = abs(p);
    return p.x + p.y;
}

vec2 octaFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = octaFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = octaFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = octaRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)

```

```

↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
float level = fract((log(rD) - tz) / period);
float distToEdge = min(level, 1.0 - level);
float edge = 1.0 - smoothstep(0.0, 0.06, distToEdge);
vec3 darkCol = outCol * 0.25;
outCol = mix(outCol, darkCol, edge);
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #401 - frac shader03 size

File: frac_shader03_size.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position for interactive control.

Technique: Multi-tap convolution blur (8 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending
- Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}

```

```

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}

vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    vec2 d = normalize(vec2(cos(t*0.27), sin(t*0.31)));
    float s = dot(p, d);
    float band = 0.5 + 0.5*sin(s*6.28318530718*0.35 + t*0.9);
    float h = fract(s*0.22 + t*0.07 + 0.15*sin(t*0.33));
    float S = 0.75 + 0.25*sin(t*0.21 + s*2.0);
    float V = 0.75 + 0.25*band;
    vec3 base = hsv2rgb(vec3(h, S, V));
    float edge = smoothstep(0.2, 0.8, band);
    return mix(base*0.6, base, edge);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}

float hash(vec2 p){
    return fract(sin(dot(p, vec2(127.1,311.7))) * 43758.5453123);
}

float starWeight(vec2 uv, float grid, float t, float aspect, float cls){
    vec2 g = uv * grid;

```

```

    vec2 id = floor(g);
    vec2 f = fract(g) - 0.5;
    float r = hash(id + cls*13.37);
    float sz = mix(0.35, 0.9, r);
    float d = max(abs(f.x)*aspect, abs(f.y));
    float fall = smoothstep(sz*0.9, 0.0, d);
    float tw = 0.5 + 0.5*sin(t*(2.0+6.0*r) + r*6.2831853);
    return fall * tw;
}

void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
    vec2 kUV = reflectUV(uv, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if(q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18*sin(time_f*0.2);
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + time_f*0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap*1.045) / ar + m);
    vec2 u2 = fract((pwrap*0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
    float hue = fract(ang*0.25 + time_f*0.08 + k*0.5);
    float sat = 0.75 - 0.25*cos(time_f*0.7 + rD*10.0);
    float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
    vec3 tint = hsv2rgb(vec3(hue, sat, val));
    float ring = smoothstep(0.0, 0.7, sin(log(rD+1e-3)*9.5 + time_f*1.2));
    float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + rD*28.0 + k*12.0);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
    vign = mix(0.85, 1.2, vign);
    float blendFactor = 0.58;
    float rC = texture(samp, u0 + off).r;
    float gC = texture(samp, u1).g;
    float bC = texture(samp, u2 - off).b;
    vec3 baseRGB = vec3(rC, gC, bC);
    vec2 s1p = vec2(cos(ang), sin(ang)) * (rw*0.55);
    vec2 s2p = vec2(cos(ang), sin(ang)) * (rw*1.10);
    vec2 s3p = vec2(cos(ang), sin(ang)) * (rw*1.85);
    vec3 sample1 = texture(samp, fract(s1p/ar + m)).rgb;
    vec3 sample2 = texture(samp, fract(s2p/ar + m)).rgb;
    vec3 sample3 = texture(samp, fract(s3p/ar + m)).rgb;
    float wSmall = starWeight(tc, 38.0, time_f, aspect, 1.0);
    float wMedium = starWeight(tc, 22.0, time_f, aspect, 2.0);
    float wLarge = starWeight(tc, 12.0, time_f, aspect, 3.0);
    float wSum = max(1e-4, wSmall + wMedium + wLarge);
    vec3 twinkleMix = (sample1*wSmall + sample2*wMedium + sample3*wLarge) / wSum;

```

```

    vec3 kaleido = baseRGB * tint;
    vec3 merged = mix(kaleido, originalTexture.rgb, blendFactor);
    merged *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
    vec3 bloom = merged * merged * 0.18 + pow(max(merged-0.6,0.0), vec3(2.0))*0.12;
    vec3 outCol = merged + bloom;
    outCol = mix(outCol, twinkleMix, 0.35 + 0.25*sin(time_f*0.6 + rD*7.0 + k*11.0));
    float wob = 0.9 + 0.1*sin(time_f + rD*14.0 + k*9.0);
    outCol *= wob;
    vec3 grad = movingGradient(tc, m, time_f, aspect);
    float gradAmt = 0.25 + 0.25*sin(time_f*0.5 + rD*7.0 + k*9.0);
    vec3 screenBlend = 1.0 - (1.0 - outCol) * (1.0 - grad);
    outCol = mix(outCol, screenBlend, gradAmt);
    vec4 t = texture(samp, tc);
    outCol = mix(outCol, outCol*t.rgb, 0.8);
    outCol = sin(outCol * (0.5 + 0.5*pingPong(time_f, 12.0)));
    outCol = clamp(outCol, vec3(0.08), vec3(0.96));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    outCol *= 1.0 + _ab * 0.6;
    outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
    ↪ 0.0, 1.0) * 0.25), _ab);
    color = vec4(outCol, 1.0);
}

```

Shader #403 - frac shader03 wormhole

File: frac_shader03_wormhole.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

```

```

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
}

```



```

float aspect = iResolution.x / iResolution.y;
vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
float r = length(p);
float t = time_f;
vec3 neon = neonPalette(t + r * 1.3);
float neonAmt = smoothstep(0.1, 0.8, r);
neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
vec3 grad = mix(tex, neon, neonAmt);
grad = mix(grad, tex, 0.2);
grad = softTone(grad);
return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 wormholeUV(vec2 uv, vec2 c, float aspect, float t) {
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float swirl = 0.22 / r;
    a += t * 0.35 + swirl;
    float z = 0.65 + 0.35 * sin(t * 0.6) + 0.15 * sin(r * 12.0 - t * 3.0);
    float rr = 1.0 / (r * 3.5 + 0.06) + 0.08 * z;
    vec2 q = vec2(cos(a), sin(a)) * rr;
    q /= ar;
    return fract(q + c);
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);

```

```

float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
vec2 wh0 = wormholeUV(tc, m, aspect, time_f);
vec2 wh1 = wormholeUV(tc + vec2(0.0009, 0.0), m, aspect, time_f + 0.03);
vec2 wh2 = wormholeUV(tc - vec2(0.0009, 0.0), m, aspect, time_f - 0.03);
vec3 whR = preBlendColor(wh0 + off);
vec3 whG = preBlendColor(wh1);
vec3 whB = preBlendColor(wh2 - off);
vec3 wormRGB = vec3(whR.r, whG.g, whB.b);
float rCenter = length((tc - m) * ar);
float throat = smoothstep(0.38, 0.06, rCenter);
float swirlGate = smoothstep(0.9, 1.6, time_f * 0.25 + 0.35 * sin(time_f * 0.7));
float gate = clamp(throat * (0.65 + 0.35 * pingPong(time_f * PI, 5.0)) + swirlGate * 0.15, 0.0, 1.0)
↳ ;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = mix(outCol, wormRGB, gate * 0.85);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #404 - frac shader03 wormhole2

File: frac_shader03_wormhole2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;

```

```

in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

```

```

}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 wormholeUV(vec2 uv, vec2 c, float aspect, float t) {
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float swirl = 0.22 / r;
    a += t * 0.35 + swirl;
    float z = 0.65 + 0.35 * sin(t * 0.6) + 0.15 * sin(r * 12.0 - t * 3.0);
    float rr = 1.0 / (r * 3.5 + 0.06) + 0.08 * z;
    vec2 q = vec2(cos(a), sin(a)) * rr;
    q /= ar;
    return fract(q + c);
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;

```

```

uv.x *= aspect;
float r = pingPong(sin(length(uv) * time_f), 5.0);
float radius = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
vec2 ar = vec2(aspect, 1.0);
vec3 baseCol = preBlendColor(tc);
float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
↪ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
vec2 wh0 = wormholeUV(tc, m, aspect, pingPong((time_f * PI), 5.0));
vec2 wh1 = wormholeUV(tc + vec2(0.0009, 0.0), m, aspect, time_f + 0.03);
vec2 wh2 = wormholeUV(tc - vec2(0.0009, 0.0), m, aspect, time_f - 0.03);
vec3 whR = preBlendColor(wh0 + off);
vec3 whG = preBlendColor(wh1);
vec3 whB = preBlendColor(wh2 - off);
vec3 wormRGB = vec3(whR.r, whG.g, whB.b);
float rCenter = length((tc - m) * ar);
float d00 = length(ar * (vec2(0.0, 0.0) - m));
float d10 = length(ar * (vec2(1.0, 0.0) - m));
float d01 = length(ar * (vec2(0.0, 1.0) - m));
float d11 = length(ar * (vec2(1.0, 1.0) - m));
float cornerR = max(max(d00, d10), max(d01, d11));
float phase = pingPong(time_f * 0.6, 1.0);
float holeR = mix(max(1.5 / max(iResolution.x, iResolution.y), 0.0008), cornerR, phase);
float rimW = mix(0.02, 0.12, smoothstep(0.0, 1.0, phase)) * cornerR;
float gate = smoothstep(holeR, holeR - rimW, rCenter);
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);

```

```

    outCol = mix(outCol, wormRGB, gate * 0.95);
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #405 - frac shader03 wormhole3

File: frac_shader03_wormhole3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);

```

```

    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(1), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {

```

```

    p = sin(abs(p));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
vec2 wormholeUV(vec2 uv, vec2 c, float aspect, float t) {
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float swirl = 0.22 / r;
    a += t * 0.35 + swirl;
    float z = 0.65 + 0.35 * sin(t * 0.6) + 0.15 * sin(r * 12.0 - t * 3.0);
    float rr = 1.0 / (r * 3.5 + 0.06) + 0.08 * z;
    vec2 q = vec2(cos(a), sin(a)) * rr;
    q /= ar;
    return fract(q + c);
}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);

```



```

vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
vec2 wh0 = wormholeUV(tc, m, aspect, pingPong((time_f * PI), 5.0));
vec2 wh1 = wormholeUV(tc + vec2(0.0009, 0.0), m, aspect, time_f + 0.03);
vec2 wh2 = wormholeUV(tc - vec2(0.0009, 0.0), m, aspect, time_f - 0.03);
vec3 whR = preBlendColor(wh0 + off);
vec3 whG = preBlendColor(wh1);
vec3 whB = preBlendColor(wh2 - off);
vec3 wormRGB = vec3(whR.r, whG.g, whB.b);
outCol += bloom;
outCol = wormRGB;
vec3 finalRGB = outCol;
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #406 - frac shader03 wormhole4

File: frac_shader03_wormhole4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);

```

```

    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

```

```

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 wormholeUV(vec2 uv, vec2 c, float aspect, float t) {
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float swirl = 0.22 / r;
    a += t * 0.35 + swirl;
    float z = 0.65 + 0.35 * sin(t * 0.6) + 0.15 * sin(r * 12.0 - t * 3.0);
    float rr = 1.0 / (r * 3.5 + 0.06) + 0.08 * z;
    vec2 q = vec2(cos(a), sin(a)) * rr;
    q /= ar;
    return fract(q + c);
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);

```

```

float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(sin(rC.r * pingPong(time_f * PI, 3.0)), sin(gC.g * pingPong(time_f * PI, 3.0)
↳ ), sin(bC.b * pingPong(time_f * PI, 3.0)));
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
vec2 wh0 = wormholeUV(tc, m, aspect, time_f);
vec2 wh1 = wormholeUV(tc + vec2(0.0009, 0.0), m, aspect, time_f + 0.03);
vec2 wh2 = wormholeUV(tc - vec2(0.0009, 0.0), m, aspect, time_f - 0.03);
vec3 whR = preBlendColor(wh0 + off);
vec3 whG = preBlendColor(wh1);
vec3 whB = preBlendColor(wh2 - off);
vec3 wormRGB = vec3(sin(whR.r * pingPong(time_f * PI, 1.3)), sin(whG.g * pingPong(time_f * PI, 1.4))
↳ , sin(whB.b * pingPong(time_f * PI, 1.5)));
float rCenter = length((tc - m) * ar);
float throat = sin(smoothstep(0.38, 0.06, rCenter) * pingPong(time_f * PI, 5.0));
float swirlGate = smoothstep(0.9, 1.6, time_f * 0.25 + 0.35 * sin(time_f * 0.7)) * pingPong(time_f *
↳ PI, 4.0);
float gate = clamp(throat * (0.65 + 0.35 * pingPong(time_f * PI, 5.0)) + pingPong(swirlGate * PI,
↳ 8.0) * 0.15, 0.0, 1.0);
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = mix(outCol, wormRGB, gate * 0.85);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #413 - frac shader05

File: frac_shader05.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: RGB <-> HSV color space conversion - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar=vec2(aspect,1.0);
    vec2 p=(uv-c)*ar;
    vec2 d=normalize(vec2(cos(t*0.27),sin(t*0.31)));
    float s=dot(p,d);
    float band=0.5+0.5*sin(s*6.28318530718*0.35+t*0.9);
    float h=fract(s*0.22+t*0.07+0.15*sin(t*0.33));
    float S=0.75+0.25*sin(t*0.21+s*2.0);
    float V=0.75+0.25*band;
    vec3 base=hsv2rgb(vec3(h,S,V));
    float edge=smoothstep(0.2,0.8,band);
    return mix(base*0.6,base,edge);
}
mat2 rot2(float a){float s=sin(a),c=cos(a);return mat2(c,-s,s,c);}
void main(){
    float aspect=iResolution.x/iResolution.y;
    vec2 ar=vec2(aspect,1.0);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 p=(tc-m)*ar;
    float T=8.0;
    float s=pingPong(time_f,T)/T;
    float e=s*s*(3.0-2.0*s);
    float z=min(1.0,mix(0.55,2.8,e));
    float spin=0.35*(e-0.5);
    p=rot2(spin)*p;
    vec2 uvTex=p/z/ar+m;
    vec4 baseTex=texture(samp,uvTex);
    vec3 grad=movingGradient(tc,m,time_f,aspect);
    float vign=1.0-smoothstep(0.78,1.15,length((tc-m)*ar));
    vign=mix(0.86,1.18,vign);
    float chroma=0.0025*(0.2+abs(e-0.5)*1.8);
    vec3 texRGB;
    texRGB.r=texture(samp,uvTex+vec2(chroma,0.0)).r;
    texRGB.g=texture(samp,uvTex).g;
    texRGB.b=texture(samp,uvTex-vec2(chroma,0.0)).b;
    float growMask=smoothstep(0.0,0.6,e)*smoothstep(1.0,0.6,e);
    float pulse=0.5+(0.5 + amp_rms * 0.3)*sin(time_f*2.0+length(p)*24.0);
    float mixTex=0.55+0.35*growMask;

```

```

float mixGrad=0.35+0.25*(1.0-growMask);
vec3 screenBlend=1.0-(1.0-texRGB)*(1.0-grad);
vec3 col=mix(texRGB,screenBlend,mixGrad);
col=mix(col,baseTex.rgb,mixTex);
col*=vign*(0.85+0.15*pulse);
vec3 bloom=col*col*0.18+pow(max(col-0.6,0.0),vec3(2.0))*0.12;
col+=bloom;
col=sin(col*(0.5+0.5*pingPong(time_f,12.0)));
col=clamp(col,vec3(0.08),vec3(0.96));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
col *= 1.0 + _ab * 0.6;
col = mix(col, col * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high, 0.0, 1.0) *
↪ 0.25), _ab);
color=vec4(col,1.0);
}

```

Shader #414 - frac shader diamond

File: frac_shader_diamond.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;

```

```

    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}

vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    vec2 d = normalize(vec2(cos(t*0.27), sin(t*0.31)));
    float s = dot(p, d);
    float band = 0.5 + 0.5*sin(s*6.28318530718*0.35 + t*0.9);
    float h = fract(s*0.22 + t*0.07 + 0.15*sin(t*0.33));
    float S = 0.75 + 0.25*sin(t*0.21 + s*2.0);
    float V = 0.75 + 0.25*band;
    vec3 base = hsv2rgb(vec3(h, S, V));
    float edge = smoothstep(0.2, 0.8, band);
    return mix(base*0.6, base, edge);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}

void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
    vec2 kUV = reflectUV(uv, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = p;
    q = abs(q);
    if(q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18*sin(time_f*0.2);
    float period = log(base);

```

```

float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + time_f*0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap*1.045) / ar + m);
vec2 u2 = fract((pwrap*0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
float hue = fract(ang*0.25 + time_f*0.08 + k*0.5);
float sat = 0.75 - 0.25*cos(time_f*0.7 + rD*10.0);
float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
vec3 tint = hsv2rgb(vec3(hue, sat, val));
float ring = smoothstep(0.0, 0.7, sin(log(rD+1e-3)*9.5 + time_f*1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + rD*28.0 + k*12.0);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
vign = mix(0.85, 1.2, vign);
float blendFactor = 0.58;
float rC = texture(samp, u0 + off).r;
float gC = texture(samp, u1).g;
float bC = texture(samp, u2 - off).b;
vec3 kaleidoRGB = vec3(rC, gC, bC);
vec4 kaleidoColor = vec4(kaleidoRGB, 1.0) * vec4(tint, 1.0);
vec4 merged = mix(kaleidoColor, originalTexture, blendFactor);
merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
vec3 outCol = merged.rgb + bloom;
float wob = 0.9 + 0.1*sin(time_f + rD*14.0 + k*9.0);
outCol *= wob;
vec3 grad = movingGradient(tc, m, time_f, aspect);
float gradAmt = 0.35 + 0.25*sin(time_f*0.5 + rD*7.0 + k*9.0);
vec3 screenBlend = 1.0 - (1.0 - outCol) * (1.0 - grad);
outCol = mix(outCol, screenBlend, gradAmt);
vec4 t = texture(samp, tc);
outCol = mix(outCol, outCol*t.rgb, 0.8);
outCol = sin(outCol * (0.5 + 0.5*pingPong(time_f, 12.0)));
outCol = clamp(outCol, vec3(0.08), vec3(0.96));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
outCol *= 1.0 + _ab * 0.6;
outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
↳ 0.0, 1.0) * 0.25), _ab);
color = vec4(outCol, 1.0);
}

```

Shader #415 - frac shader dmd4 glass

File: frac_shader_dmd4_glass.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB. Responds to mouse position for interactive control.

Technique: RGB <-> HSV color space conversion - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```



```

uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float hash21(vec2 p){
    p=fract(p*vec2(123.34,456.21));
    p+=dot(p,p+45.32);
    return fract(p.x*p.y);
}
vec2 hash22(vec2 p){
    float n=hash21(p);
    float m=hash21(p+17.7);
    return vec2(n,m);
}
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
float n2(vec2 p){
    vec2 i=floor(p),f=fract(p);
    vec2 u=f*f*(3.0-2.0*f);
    float a=hash21(i);
    float b=hash21(i+vec2(1,0));
    float c=hash21(i+vec2(0,1));
    float d=hash21(i+vec2(1,1));
    return mix(mix(a,b,u.x),mix(c,d,u.x),u.y);
}
float fbm(vec2 p){
    float s=0.0,a=0.5;
    for(int i=0;i<5;i++){
        s+=a*n2(p); p*=2.02; a*=0.5;
    }
    return s;
}
struct VOut{ float d; vec2 id; vec2 gv; vec2 nearest; };
VOut voronoi(vec2 p){
    vec2 i=floor(p), f=fract(p);
    float md=1e9; vec2 mid=vec2(0.0), mcell=vec2(0.0), mpt=vec2(0.0);
    for(int y=-1;y<=1;y++){
        for(int x=-1;x<=1;x++){
            vec2 g=vec2(x,y);
            vec2 id=i+g;
            vec2 rnd=hash22(id);
            vec2 o=sin((rnd*2.0-1.0)*6.28318 + vec2(time_f*0.35, time_f*0.27))*0.25 + rnd*0.5;
            vec2 r=g+o-f;
            float d=dot(r,r);
            if(d<md){md=d; mid=id; mcell=g; mpt=r;}
        }
    }
}

```

```

float edge=1e9;
for(int y=-1;y<=1;y++){
    for(int x=-1;x<=1;x++){
        vec2 g=vec2(x,y);
        if(all(equal(g,mcell))) continue;
        vec2 id=i+g;
        vec2 rnd=hash22(id);
        vec2 o=sin((rnd*2.0-1.0)*6.28318 + vec2(time_f*0.35, time_f*0.27))*0.25 + rnd*0.5;
        vec2 r=g+o-f;
        vec2 dif=0.5*(mpt+r);
        float d=(dot(dif,dif)-dot(0.5*(r-mpt),0.5*(r-mpt)))/max(length(dif),1e-4);
        edge=min(edge,abs(d));
    }
}
VOut v; v.d=sqrt(md); v.id=mid; v.gv=f; v.nearest=mpt; return VOut(edge,mid,f,mpt);
}

void main(){
    vec2 res=iResolution;
    float aspect=res.x/res.y;
    vec2 m=(iMouse.z>0.5)?iMouse.xy/res:vec2(0.5);
    vec2 uv=(tc-m)*vec2(aspect,1.0);
    float gzoom=exp(sin(time_f*0.6)*1.1);
    float zt=mix(0.65,1.85,0.5+0.5*sin(time_f*0.43));
    vec2 warp=vec2(fbm(uv*0.9+time_f*0.15), fbm(uv*0.9-vec2(3.1,1.7)+time_f*0.12));
    float localScale=mix(0.75,2.4,fbm(uv*0.6 + warp*0.8));
    vec2 p=uv*(gzoom*zt*localScale);
    vec2 flow=vec2(sin(time_f*0.23),cos(time_f*0.19))*0.6;
    p+=flow*fbm(uv*0.8+warp*0.5);
    VOut v=voronoi(p);
    float px=length(vec2(dFdx(tc.x),dFdy(tc.y)));
    float lead=0.006/(gzoom*localScale);
    float edge=1.0 - smoothstep(lead - px, lead + px, v.d);
    vec2 pid=v.id;
    float hue=fract(hash21(pid)*0.97 + 0.15*sin(time_f*0.2+hash21(pid+7.3)));
    float sat=mix(0.6,0.95,hash21(pid+3.7));
    float val=mix(0.7,1.0,hash21(pid+11.1));
    vec3 pane=hsv2rgb(vec3(hue,sat,val));
    vec2 nrm=normalize(v.nearest+1e-6);
    float gloss=0.55+0.45*pow(max(dot(nrm,normalize(vec2(cos(time_f*0.27),sin(time_f*0.31))))),0.0),1.2);
    pane*=gloss;
    vec3 base=texture(samp,tc).rgb;
    float baseMix=pingPong(0.12+0.15*hash21(pid+2.1) * (time_f * PI), 5.0);
    vec3 glass=mix(pane,base,baseMix);
    vec3 leadCol=vec3(0.08,0.09,0.10);
    vec3 col=mix(glass,leadCol,edge);
    float b=0.18;
    col+=pow(max(col-0.72,0.0),vec3(2.0))*b;
    float vign=1.0 - smoothstep(0.78,1.18,length((tc-m)*vec2(aspect,1.0)));
    col*=mix(0.92,1.12,vign);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    col *= 1.0 + _ab * 0.6;
    col = mix(col, col * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high, 0.0, 1.0) *
    ↪ 0.25), _ab);
    color=vec4(clamp(col,0.0,1.0),1.0);
}

```

Shader #420 - frac_star1

File: frac_star1.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float hash1(vec2 p){ return fract(sin(dot(p, vec2(127.1,311.7)))) * 43758.5453); }
vec2 hash2(vec2 p){ return fract(sin(vec2(dot(p, vec2(127.1,311.7)), dot(p, vec2(269.5,183.3))))) *
    ↪ 43758.5453); }
float pingPong(float x,float l){ float m=mod(x,l*2.0); return m<=l?m:l*2.0-m; }
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
vec3 neonPalette(float t){
    vec3 a=vec3(1.0,0.15,0.75), b=vec3(0.10,0.55,1.0), c=vec3(0.10,1.0,0.45);
    float ph=fract(t*0.08);
    vec3 k1=mix(a,b,smoothstep(0.00,0.33,ph));
    vec3 k2=mix(b,c,smoothstep(0.33,0.66,ph));
    vec3 k3=mix(c,a,smoothstep(0.66,1.00,ph));
    float A=step(ph,0.33), B=step(0.33,ph)*step(ph,0.66), C=step(0.66,ph);
    return normalize(A*k1+B*k2+C*k3)*1.05;
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res){
    vec2 ts=1.0/res;
    vec3 s00=textureGrad(img, uv+ts*vec2(-1,-1), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10=textureGrad(img, uv+ts*vec2( 0,-1), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20=textureGrad(img, uv+ts*vec2( 1,-1), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01=textureGrad(img, uv+ts*vec2(-1, 0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11=textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21=textureGrad(img, uv+ts*vec2( 1, 0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02=textureGrad(img, uv+ts*vec2(-1, 1), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12=textureGrad(img, uv+ts*vec2( 0, 1), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22=textureGrad(img, uv+ts*vec2( 1, 1), dFdx(uv), dFdy(uv)).rgb;
    return (s00+2.0*s10+s20+2.0*s01+4.0*s11+2.0*s21+s02+2.0*s12+s22)/16.0;
}
vec3 softTone(vec3 c){
    c=pow(max(c,0.0), vec3(0.95));
    float l=dot(c, vec3(0.299,0.587,0.114));
    c=mix(vec3(1), c, 0.9);
    return clamp(c,0.0,1.0);
}
```

```

}
vec2 sitePos(vec2 id, float grid){
    vec2 j=(hash2(id)*2.0-1.0)*(0.33+0.10*sin(time_f*0.37+6.2831*hash1(id+3.71)));
    return (id+0.5+j)/grid;
}
float starMetric(vec2 d, float n, float amp){
    float ang=atan(d.y,d.x);
    float r=length(d);
    float mod=1.0+amp*cos(n*ang);
    return r/max(0.001,mod);
}
void main(){
    vec2 res=iResolution;
    float aspect=res.x/res.y;
    vec2 focus=(iMouse.z>0.5)?(iMouse.xy/res):vec2(0.5);
    float t=pingPong(time_f*0.35,1.0);
    float zoom=mix(0.7,2.6,t);
    vec2 uvZ=(tc-focus)*zoom+focus;
    float grid=mix(18.0,28.0,0.5+0.5*sin(time_f*0.11));
    vec2 gtc=uvZ*grid;
    vec2 gid0=floor(gtc);
    float best1=1e9,best2=1e9;
    vec2 bestId=vec2(0.0), bestSite=vec2(0.0);
    float bestN=7.0, bestA=0.3;
    for(int j=-1;j<=1;j++){
        for(int i=-1;i<=1;i++){
            vec2 nid=gid0+vec2(i,j);
            vec2 s=sitePos(nid,grid);
            vec2 d=uvZ-s; d.x*=aspect;
            float nPts=floor(7.0+5.0*hash1(nid+7.7));
            float amp=0.28+0.22*hash1(nid+4.2);
            float dist=starMetric(d,nPts,amp);
            if(dist<best1){
                best2=best1;
                best1=dist;
                bestId=nid;
                bestSite=s;
                bestN=nPts;
                bestA=amp;
            }else if(dist<best2){
                best2=dist;
            }
        }
    }
    float edge=0.5*(best2-best1);
    float px=1.0/min(res.x,res.y);
    float border=1.0-smoothstep(px*0.75, px*0.75+fwidth(edge), edge);
    vec2 rel=uvZ-bestSite; rel.x*=aspect;
    float rot=6.2831853*hash1(bestId+11.9)+time_f*(0.10+0.08*hash1(bestId+2.2));
    float cs=cos(rot), sn=sin(rot);
    vec2 rrel=mat2(cs,-sn,sn,cs)*rel;
    vec2 pieceUV=rrel; pieceUV.x/=aspect; pieceUV+=bestSite;
    vec4 texCol=texture(samp,pieceUV);
    float ang=atan(rrel.y,rrel.x);
    float flare=pow(0.5+0.5*cos(bestN*ang+time_f*0.7+6.2831*hash1(bestId+3.4)),3.0);
    vec3 glow=HSV2rgb(vec3(fract(hash1(bestId+9.3)+time_f*0.05),0.85,1.0));
    vec3 neon=neonPalette(time_f+hash1(bestId+1.1)*8.0);
    vec3 tint=mix(neon,glow,0.45)*(0.55+0.45*flare);
    float rad=length(rrel);
    float vign=smoothstep(0.9,0.2,rad/(0.55+0.05*sin(time_f*0.5+hash1(bestId+5.5))));
    vec3 pieceCol=mix(texCol.rgb, texCol.rgb*tint, 0.25*vign);
    vec3 pre=tentBlur3(samp, tc, res);

```

```

    vec3 inside=mix(pre, pieceCol, 0.85);
    vec3 borderCol=mix(vec3(0.02,0.02,0.03), neonPalette(time_f*0.5), 0.15);
    vec3 outRGB=mix(inside, sin(borderCol * (time_f * PI)), border);
    outRGB=softTone(outRGB);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    outRGB *= 1.0 + _ab * 0.6;
    outRGB = mix(outRGB, outRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
    ↪ 0.0, 1.0) * 0.25), _ab);
    color=vec4(outRGB, texCol.a);
}

```

Shader #421 - frac texture

File: frac_texture.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Basic sharpening/blurring to keep texture quality high MODIFIED: No neon mixing. Just returns the texture, slightly dimmed. Optional: Desaturate slightly to look more "serious" or keep as is.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
}

```

```

    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    return tex * 0.7;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec2 xuv = 1.0 - abs(1.0 - 2.0 * tc);
    xuv = xuv - floor(xuv);
    vec4 baseTex = texture(samp, xuv);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.5 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);

```

```

    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
    ↪ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.6 + 0.4 * pulse);
    outCol *= vign;
    outCol = clamp(outCol, vec3(0.0), vec3(1.0));
    vec3 finalRGB = mix(baseTex.rgb * 0.8, outCol, pingPong(glow * PI, 5.0) * 0.6);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #430 - fractl shader04 tri

File: fractl_shader04_tri.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;

```

```

uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;

```



```

    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 gdir = normalize(vec2(cos(time_f * 0.27), sin(time_f * 0.31)));
    float s = dot((uv - 0.5) * vec2(iResolution.x / iResolution.y, 1.0), gdir);
    float w = max(fwidth(s) * 4.0, 0.002);
    float band = smoothstep(-0.5 - w, -0.5 + w, sin(s * 2.2 + time_f * 0.9));
    vec3 neon = neonPalette(time_f);
    vec3 grad = mix(tex, mix(tex, neon, 0.6), 0.35 + 0.25 * band);
    grad = mix(grad, tex, 0.10);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p * (pingPong(time_f * PI, 5.0))));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

mat2 rot2(float a) {
    float s = sin(a), c = cos(a);
    return mat2(c, -s, s, c);
}

float sdEquilateralTriangle(vec2 p) {
    const float k = 1.73205080757;
    p.x = abs(p.x) - 1.0;
    p.y = p.y + 1.0 / k;
    if (p.x + k * p.y > 0.0) p = vec2(p.x - k * p.y, -k * p.x - p.y) * 0.5;
    p.x = p.x + 1.0;
    return -length(p) * sign(p.y);
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(pingPong(time_f * PI, 25.0) * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
}

```

```

kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
float triSpin = time_f * 0.8 + k * 6.0;
float triScale = 8.0 + 3.0 * sin(time_f * 0.3);
vec2 tcoord = pwrap * triScale;
tcoord = rot2(triSpin) * tcoord;
vec2 cell = fract(tcoord) - 0.5;
float triDist = sdEquilateralTriangle(cell * 2.1);
float triMask = smoothstep(0.10, 0.0, triDist);
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 triColor = neonPalette(time_f + k * 3.0);
outCol = mix(outCol, outCol + triColor * 1.3, triMask * 0.8);
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #456 - g drum by mouse

File: g_drum_by_mouse.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control.

Technique: Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 d = tc - m;
    float r = length(d);
    vec2 dir = d / max(r, 1e-6);
    float waveLength = 0.05;
    float amplitude = 0.02;
    float speed = 2.0;
    float ripple = sin((r / waveLength - time_f * speed) * 6.2831853);
    vec2 uv = tc + dir * ripple * amplitude;
    color = texture(samp, uv);
}

```

Shader #459 - g_ufo_3d

File: g_ufo_3d.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p2 = (tc - m) * ar;
    float ax = 0.25 * sin(time_f * 0.7);
    float ay = 0.25 * cos(time_f * 0.6);
    float az = time_f * 0.5;
    vec3 p3 = vec3(p2, 1.0);
    mat3 R = rotZ(az) * rotY(ay) * rotX(ax);
    vec3 r = R * p3;
    float k = 0.6;
    float zf = 1.0 / (1.0 + r.z * k);
    vec2 q = r.xy * zf;
    float dist = length(p2);
    float scale = 1.0 + 0.2 * sin(dist * 15.0 - time_f * 2.0);
    q *= scale;
    vec2 uv = q / ar + m;
    uv = clamp(uv, 0.0, 1.0);
    color = texture(samp, uv);
}

```

```
}
```

Shader #460 - g_ufo 3d v2

File: g_ufo_3d_v2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position for interactive control. Rotate around X-axis with ping pong effect and direction based on texture coordinate x Rotate around Y-axis with ping pong effect and direction based on texture coordinate y Rotate around Z-axis over time

Technique: 2D rotation matrix - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p2 = (tc - m) * ar;
    float ax = 0.25 * sin(time_f * 0.7 + tc.x * 10.0);
    float ay = 0.25 * cos(time_f * 0.6 - tc.y * 10.0);
    float az = time_f * 0.5;
    vec3 p3 = vec3(p2, 1.0);
    mat3 R = rotZ(az) * rotY(ay) * rotX(ax);
    vec3 r = R * p3;
    float k = 0.6;
    float zf = 1.0 / (1.0 + r.z * k);
    vec2 q = r.xy * zf;
    float dist = length(p2);
    float scale_factor = 0.2 * sin(dist * 15.0 - time_f * 2.0 + tc.x * 5.0 + tc.y * 5.0);
    q *= (1.0 + scale_factor);
    vec2 uv = q / ar + m;
    uv = clamp(uv, 0.0, 1.0);
    color = texture(samp, uv);
}
```

Shader #465 - gem aura

File: gem-aura.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Reduced iterations slightly for clarity on the sphere surface Strobing pulse logic Spherical bulging effect

Technique: Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<4;i++){
        p = abs((p - c) * (zoom + 0.1 * sin(t * 0.3 + float(i)))) - 0.4 + c;
        p = rotateUV(p, t * 0.1 + float(i) * 0.05, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.2, 0.8);
    vec3 cyan = vec3(0.0, 0.9, 1.0);
    vec3 purple = vec3(0.6, 0.1, 1.0);
    float ph = fract(t * 0.1);
    if(ph < 0.33) return mix(pink, cyan, ph * 3.0);
    if(ph < 0.66) return mix(cyan, purple, (ph - 0.33) * 3.0);
    return mix(purple, pink, (ph - 0.66) * 3.0);
}
vec3 softTone(vec3 c){
    return pow(clamp(c, 0.0, 1.0), vec3(0.85));
}
vec3 preBlendColor(vec2 uv){
    vec3 tex = texture(samp, uv).rgb;
    float brightness = dot(tex, vec3(0.299, 0.587, 0.114));
    vec3 neon = neonPalette(time_f + brightness);
    float strobe = 0.9 + 0.1 * sin(time_f * 20.0);
    return mix(tex, neon, 0.5) * strobe;
}
float diamondRadius(vec2 p){
    p = abs(p);

```

```

    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void){
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p_centered = (tc - 0.5) * 2.0;
    p_centered.x *= aspect;
    float d = length(p_centered);
    float sphereRadius = 1.0;
    float z = sqrt(max(0.0, sphereRadius * sphereRadius - d * d));
    float fisheye = atan(d, z) / (PI * 0.5);
    vec2 sphereUV = (d > 0.0) ? (p_centered / d) * fisheye : vec2(0.0);
    sphereUV.x /= aspect;
    sphereUV = sphereUV * 0.5 + 0.5;
    vec4 baseTex = texture(samp, tc);
    float mask = smoothstep(1.0, 0.95, d);
    float outerGlow = smoothstep(1.3, 0.0, d) * 0.4;
    float seg = 6.0 + 2.0 * sin(time_f * 0.5);
    vec2 kUV = reflectUV(sphereUV, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.2 + 0.3 * sin(time_f * 0.4);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    vec2 p = (kUV - m) * vec2(aspect, 1.0);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(p.y, p.x) + time_f * 0.3;
    float k = fract(log(rD) - time_f * 0.5);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * exp(k);
    vec2 u0 = fract(pwrap + m);
    vec3 kaleidoRGB = preBlendColor(u0);
    float shading = dot(normalize(vec3(p_centered, z)), normalize(vec3(1.0, 1.0, 1.0)));
    shading = smoothstep(-0.2, 1.0, shading);
    vec3 sphereCol = kaleidoRGB * shading * 1.5;
    sphereCol += pow(shading, 10.0) * 0.5;
    vec3 finalRGB = mix(baseTex.rgb * 0.5, sphereCol, mask);
    finalRGB += neonPalette(time_f * 0.5) * outerGlow * (0.5 + 0.5 * sin(time_f * 5.0));
    color = vec4(finalRGB, 1.0);
}

```

Shader #475 - gem spiral frac

File: gem-spiral-frac.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Rotation matrix for the fractal folding Shift coordinates to be relative to the mouse (or center) Correct aspect ratio for a circular look

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation - Driven by: time, mouse, resolution

```

#version 330
in vec2 tc;

```

```

out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
vec3 neon(float t) {
    return 0.5 + 0.5 * cos(6.28318 * (t + vec3(0.0, 0.33, 0.67)));
}
mat2 rotate2d(float a) {
    return mat2(cos(a), -sin(a), sin(a), cos(a));
}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution.xy) : vec2(0.5);
    vec2 uv = tc;
    float dist = distance(uv, m);
    float falloff = smoothstep(0.5, 0.0, dist);
    float distortionStrength = 0.03 * falloff;
    float distortionFrequency = 20.0;
    float d1 = sin((uv.y + dist * 5.0) * distortionFrequency + time_f) * distortionStrength;
    float d2 = cos((uv.x + uv.y + dist * 5.0) * distortionFrequency + time_f) * distortionStrength;
    vec2 distortedUV = uv + vec2(d1, d2);
    vec2 p = (distortedUV - m);
    p.x *= aspect;
    float p_dist = length(p);
    float p_angle = atan(p.y, p.x);
    float fractalLayer = 0.0;
    vec2 iterP = p;
    float zoom = 1.5 + 0.5 * sin(time_f * 0.5);
    for(int i = 0; i < 5; i++) {
        iterP = abs(iterP) * zoom - 0.8;
        iterP *= rotate2d(time_f * 0.2 + float(i));
        fractalLayer += exp(-length(iterP) * 2.0);
    }
    float spiral = p_angle + log(p_dist + 0.001) * 2.0 - time_f * 3.0;
    float spiralPattern = smoothstep(0.1, 0.9, sin(spiral * 5.0));
    vec3 fractalCol = neon(p_dist - time_f * 0.5 + fractalLayer * 0.1);
    fractalCol *= fractalLayer * spiralPattern;
    vec4 tex = texture(samp, distortedUV);
    vec3 finalRGB = mix(tex.rgb, fractalCol, falloff * 0.7);
    finalRGB += fractalCol * (0.5 + 0.5 * sin(time_f * 10.0)) * 0.2;
    color = vec4(finalRGB, 1.0);
}

```

Shader #481 - gem kale large

File: gem_kale_large.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Standard cosine gradient for neon colors (Pink, Blue, Cyan, Purple) Fixed Blur: Using standard texture() to leverage Mipmaps for anti-aliasing Convert to neon spectrum based on brightness and time

Technique: Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;

```

```

in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.1415926535897932384626433832795;
vec3 getNeon(float t) {
    return 0.5 + 0.5 * cos(6.28318 * (vec3(1.0, 1.0, 1.0) * t + vec3(0.0, 0.33, 0.67)));
}
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * 0.35 + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 smoothSample(sampler2D img, vec2 uv) {
    return texture(img, uv).rgb;
}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = smoothSample(samp, uv);
    float brightness = dot(tex, vec3(0.299, 0.587, 0.114));
    vec3 neon = getNeon(brightness + time_f * 0.5);
    return mix(tex, neon, 0.6) * 1.2;
}
float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {

```



```

vec2 xuv = 1.0 - abs(1.0 - 2.0 * tc);
vec4 baseTex = texture(samp, xuv);
vec2 uv = tc * 2.0 - 1.0;
float aspect = iResolution.x / iResolution.y;
uv.x *= aspect;
float r = pingPong(sin(length(uv) * time_f), 5.0);
float radius = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radius, radius - 0.5, r);
vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
vec2 ar = vec2(aspect, 1.0);
float seg = 4.0 + 2.0 * sin(time_f * 0.33);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.15 + 0.15 * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * 0.23, m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0);
float tz = time_f * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw * 0.5;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.02) / ar + m);
vec2 u2 = fract((pwrap * 0.98) / ar + m);
vec3 rC = preBlendColor(u0);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float strobe = 0.8 + 0.2 * sin(time_f * 25.0);
float pulse = 0.5 + 0.5 * sin(time_f * 4.0 + rD * 10.0);
vec3 outCol = kaleidoRGB * strobe * (0.7 + 0.5 * pulse);
outCol = pow(outCol, vec3(0.8));
outCol *= 1.3;
float vign = 1.0 - smoothstep(0.7, 1.5, length((tc - m) * ar));
outCol *= vign;
vec3 finalRGB = mix(baseTex.rgb * 0.3, outCol, 0.8);
color = vec4(finalRGB, 1.0);
}

```

Shader #503 - glass mouse

File: glass_mouse.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position for interactive control.

Technique: Hermite smoothstep interpolation - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;

```

```

uniform vec4 iMouse;
void main(void) {
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    vec2 c = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    float distortionStrength = 0.2;
    float noiseFactor = sin(uv.x * 10.0 + time_f) * cos(uv.y * 10.0 + time_f);
    float w = 1.0 - smoothstep(0.0, 0.45, length(uv - c));
    vec2 dir = normalize(uv - c + 1e-5);
    vec2 distortedCoord = uv + distortionStrength * noiseFactor * w * dir;
    distortedCoord = clamp(distortedCoord, 0.0, 1.0);
    color = texture(samp, distortedCoord);
}

```

Shader #504 - glass mouse rad

File: glass_mouse_rad.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position for interactive control.

Technique: Hermite smoothstep interpolation - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
void main(void) {
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    vec2 c = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    float distortionStrength = 0.25;
    float noiseFactor = sin(uv.x * 10.0 + time_f) * cos(uv.y * 10.0 + time_f);
    float radius = 0.75;
    float w = 1.0 - smoothstep(0.0, radius, length(uv - c));
    vec2 dir = normalize(uv - c + 1e-5);
    vec2 distortedCoord = uv + distortionStrength * noiseFactor * w * dir;
    distortedCoord = clamp(distortedCoord, 0.0, 1.0);
    color = texture(samp, distortedCoord);
}

```

Shader #505 - glass PI

File: glass_PI.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position for interactive control.

Technique: Hermite smoothstep interpolation - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {

```

```

    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void) {
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    vec2 c = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    float distortionStrength = 0.25;
    float noiseFactor = sin(uv.x * 10.0 + pingPong(time_f * PI, 10.0)) * cos(uv.y * 10.0 + time_f);
    float radius = 0.75;
    float w = 1.0 - smoothstep(0.0, radius, length(uv - c));
    vec2 dir = normalize(uv - c + 1e-5);
    vec2 distortedCoord = uv + distortionStrength * noiseFactor * w * dir;
    distortedCoord = clamp(distortedCoord, 0.0, 1.0);
    color = texture(samp, distortedCoord);
}

```

Shader #506 - glitch no noise mouse

File: glitch-no-noise-mouse.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5,1.0,1.5,2.0,2.5,2.5,2.0,1.5,1.0,0.5,
                                     1.0,2.0,2.5,3.0,3.5,3.5,3.0,2.5,2.0,1.0,
                                     1.5,2.5,3.0,3.5,4.0,4.0,3.5,3.0,2.5,1.5,
                                     2.0,3.0,3.5,4.0,4.5,4.5,4.0,3.5,3.0,2.0,
                                     2.5,3.5,4.0,4.5,5.0,5.0,4.5,4.0,3.5,2.5,
                                     2.5,3.5,4.0,4.5,5.0,5.0,4.5,4.0,3.5,2.5,
                                     2.0,3.0,3.5,4.0,4.5,4.5,4.0,3.5,3.0,2.0,
                                     1.5,2.5,3.0,3.5,4.0,4.0,3.5,3.0,2.5,1.5,
                                     1.0,2.0,2.5,3.0,3.5,3.5,3.0,2.5,2.0,1.0,
                                     0.5,1.0,1.5,2.0,2.5,2.5,2.0,1.5,1.0,0.5);
    for (int i = 0; i < 100; i++) kernel[i] = kernelVals[i];
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) kernelSum += kernel[i];
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {

```

```

        vec2 offset = vec2(float(x), float(y)) * texelSize;
        result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
    }
}
return result / kernelSum;
}

void main(void) {
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    float dist = length((tc - m) * ar);
    float radius = 0.45;
    float w = 1.0 - smoothstep(0.0, radius, dist);
    float baseGlitch = 0.001;
    float glitchStrength = mix(baseGlitch, baseGlitch * 4.0, w);
    vec2 uv = tc;
    uv.x += (rand(uv + time_f) - 0.5) * glitchStrength;
    uv.y += (rand(uv + time_f * 1.5) - 0.5) * glitchStrength;
    float band = (rand(vec2(floor(tc.y * iResolution.y) + floor(time_f * 30.0))) - 0.5) * glitchStrength
    ↪ * 6.0 * w;
    uv.x += band;
    vec4 texColor = blur(samp, uv, iResolution);
    vec4 colorShift = vec4(texColor.r,
                           texColor.g * 0.5 + 0.5,
                           texColor.b * 0.5 + 0.5,
                           texColor.a);
    colorShift = sin(colorShift * time_t);
    float glitchNoise = rand(uv + time_f);
    vec4 glitchColor = vec4(vec3(glitchNoise), 1.0) * glitchStrength;
    color = mix(colorShift, glitchColor, glitchStrength * glitchNoise);
    color = sin(color * time_t);
    color.a = 1.0;
}

```

Shader #529 - gpt halluc

File: gpt_halluc.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Triangle wave wrap function: turns 0..1 into a seamless mirror pattern. If mapping to a mesh with square texture, you might want to force aspect to 1.0. This makes the UVs "bounce" back and forth (0->1->0) ensuring seamless edges on the mesh.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - 2D rotation matrix - UV mirroring/folding for symmetry - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;

```

```

const float iSaturation = 1.0;
const float iHueShift   = 0.0;
const float iZoom       = 1.0;
const float iRotation   = 0.0;
vec3 adjustBrightness(vec3 col, float b) {
    return col * b;
}
vec3 adjustContrast(vec3 col, float c) {
    return (col - 0.5) * c + 0.5;
}
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 rotateHue(vec3 col, float angle) {
    float U = cos(angle);
    float W = sin(angle);
    mat3 R = mat3(
        0.299 + 0.701*U + 0.168*W,
        0.587 - 0.587*U + 0.330*W,
        0.114 - 0.114*U - 0.497*W,
        0.299 - 0.299*U - 0.328*W,
        0.587 + 0.413*U + 0.035*W,
        0.114 - 0.114*U + 0.292*W,
        0.299 - 0.300*U + 1.250*W,
        0.587 - 0.588*U - 1.050*W,
        0.114 + 0.886*U - 0.203*W
    );
    return clamp(R * col, 0.0, 1.0);
}
vec3 applyColorAdjustments(vec3 col) {
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    col = rotateHue(col, iHueShift);
    return clamp(col, 0.0, 1.0);
}
vec2 applyZoomRotation(vec2 uv, vec2 center) {
    vec2 p = uv - center;
    float c = cos(iRotation);
    float s = sin(iRotation);
    p = mat2(c, -s, s, c) * p;
    float z = max(abs(iZoom), 0.001);
    p /= z;
    return p + center;
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
    vec2 deriv = fwidth(tc);
    if (deriv.x > 0.0 || deriv.y > 0.0) {
        lod = log2(max(max(deriv.x, deriv.y) * max(ts.x, ts.y), 1.0));
    }
    return textureLod(tex, sampleUV, lod);
}
float hash(vec2 p) {

```

```

    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 5; i++) {
        v += a * noise(p);
        p *= 2.1;
        a *= 0.55;
    }
    return v;
}
vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}
vec3 sampleWarp(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float freqControl = clamp(iFrequency, 0.0, 2.0);
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    float f1 = fbm(p * 1.8 + t * 0.3);
    float f2 = fbm(p.yx * 2.3 - t * 0.25);
    float f3 = fbm(p * 3.1 + vec2(1.3, -0.7) * t * 0.2);
    vec2 swirl = p;
    float r = length(swirl);
    float a = atan(swirl.y, swirl.x);
    a += (f1 * 4.0 + f2 * 2.0) * strength * 0.6;
    swirl = vec2(cos(a), sin(a)) * r;
    float sliceBase = 8.0;
    float sliceRange = 8.0;
    float slices = sliceBase + sliceRange * (0.3 + 0.7 * ampControl) + 4.0 * sin(t * 0.17);
    vec2 k = kaleido(swirl + vec2(f2, f3) * 0.4 * strength, slices);
    vec2 flow = k;
    flow.x += f1 * 0.8 * strength;
    flow.y += (f2 - f3) * 0.8 * strength;
    vec2 base = flow / vec2(aspect, 1.0) + center;
    base = fract(base);
    float chromaBoost = 0.5 + 0.5 * ampControl;
    vec2 chromaShift = 0.0035 * strength * chromaBoost *
        vec2(sin(t + f1 * 6.0), cos(t * 1.3 + f2 * 6.0));
    float rC = mxTexture(samp, base + chromaShift).r;
    float gC = mxTexture(samp, base).g;
    float bC = mxTexture(samp, base - chromaShift).b;
    vec3 col = vec3(rC, gC, bC);
    float bright = 0.7 + 0.6 * f3 + 0.4 * sin(t * 0.6 + f1 * 3.0);
    bright *= (0.6 + 0.8 * freqControl);
}

```

```

    col *= bright;
    float sat = 1.3 + 0.7 * sin(t * 0.43 + f2 * 5.0);
    sat *= (0.6 + 0.8 * freqControl);
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    col = mix(vec3(gray), col, sat);
    return col;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    uv = applyZoomRotation(uv, vec2(0.5));
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float freqControl = clamp(iFrequency, 0.0, 2.0);
    float tSpeed = 0.3 + 1.7 * (freqControl * 0.5);
    float t = time_f * tSpeed;
    float strength = 0.6 + 1.6 * (ampControl * 0.5);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec3 colA = sampleWarp(uv, t, strength, center, iResolution);
    vec3 colB = sampleWarp(uv + vec2(0.01, -0.007),
        t + 3.14,
        strength * 0.9,
        center,
        iResolution);
    float blend = 0.5 + 0.5 * sin(t * 0.25);
    vec3 col = mix(colA, colB, blend);
    color.rgb = applyColorAdjustments(col);
    color.a = 1.0;
}

```

Shader #554 - grid by mouse

File: grid_by_mouse.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;

float sstep(float a, float b, float x){x=clamp((x-a)/(b-a),0.0,1.0);return x*x*(3.0-2.0*x);}

void main(void) {
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec4 texColor = texture(samp, tc);
    float sparkle = abs(sin(time_f * 10.0 + tc.x * 100.0) * cos(time_f * 15.0 + tc.y * 100.0));
    vec2 d = tc - m;
    float dist = length(d);
    float r = mix(0.08, 0.35, clamp(amp,0.0,1.0));
    float fall = 1.0 - sstep(0.0, r, dist);
    float pulse = 0.5 + 0.5 * sin(time_f * 6.0 + dist * 40.0);
    float mask = fall * pulse;
}

```

```

    vec3 magicalColor = vec3(sin(time_f * 2.0) * 0.5 + 0.5, cos(time_f * 3.0) * 0.5 + 0.5, sin(time_f *
↪ 4.0) * 0.5 + 0.5);
    vec3 glow = magicalColor * sparkle * mask;
    color = vec4(texColor.rgb + glow, texColor.a);
}

```

Shader #567 - halluc gem

File: halluc_gem.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Triangle wave wrap function: turns 0..1 into a seamless mirror pattern Hard geometric rotation Replaces the liquid warp with a hard iterative fold

Technique: Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - Per-channel UV offset for chromatic aberration - 2D rotation matrix - UV mirroring/folding for symmetry - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;
const float iSaturation = 1.0;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
vec3 adjustBrightness(vec3 col, float b) {
    return col * b;
}
vec3 adjustContrast(vec3 col, float c) {
    return (col - 0.5) * c + 0.5;
}
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 rotateHue(vec3 col, float angle) {
    float U = cos(angle);
    float W = sin(angle);
    mat3 R = mat3(
        0.299 + 0.701*U + 0.168*W,
        0.587 - 0.587*U + 0.330*W,
        0.114 - 0.114*U - 0.497*W,
        0.299 - 0.299*U - 0.328*W,
        0.587 + 0.413*U + 0.035*W,
        0.114 - 0.114*U + 0.292*W,
        0.299 - 0.300*U + 1.250*W,
        0.587 - 0.588*U - 1.050*W,
        0.114 + 0.886*U - 0.203*W
    );
    return clamp(R * col, 0.0, 1.0);
}

```



```

vec3 applyColorAdjustments(vec3 col) {
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    col = rotateHue(col, iHueShift);
    return clamp(col, 0.0, 1.0);
}

vec2 applyZoomRotation(vec2 uv, vec2 center) {
    vec2 p = uv - center;
    float c = cos(iRotation);
    float s = sin(iRotation);
    p = mat2(c, -s, s, c) * p;
    float z = max(abs(iZoom), 0.001);
    p /= z;
    return p + center;
}

vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}

vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
    vec2 deriv = fwidth(tc);
    if (deriv.x > 0.0 || deriv.y > 0.0) {
        lod = log2(max(max(deriv.x, deriv.y) * max(ts.x, ts.y), 1.0));
    }
    return textureLod(tex, sampleUV, lod);
}

vec2 rotate2D(vec2 p, float a) {
    float c = cos(a);
    float s = sin(a);
    return mat2(c, -s, s, c) * p;
}

vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}

vec3 sampleFractal(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float freqControl = clamp(iFrequency, 0.0, 2.0);
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    float slices = 6.0 + floor(ampControl * 4.0);
    p = kaleido(p, slices);
    int iterations = 4 + int(ampControl * 2.0);
    float scale = 1.2 + (freqControl * 0.5);
    float shift = 0.1 * strength;
    float angle = t * 0.1;
    for(int i = 0; i < iterations; i++) {
        p = abs(p);
        p -= shift;
        p *= scale;
        p = rotate2D(p, angle + float(i)*0.5);
    }
}

```

```

    vec2 finalUV = p * 0.5 + center;
    float chroma = 0.005 * strength * sin(t);
    float r = mxTexture(samp, finalUV + vec2(chroma, 0.0)).r;
    float g = mxTexture(samp, finalUV).g;
    float b = mxTexture(samp, finalUV - vec2(chroma, 0.0)).b;
    return vec3(r, g, b);
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    uv = applyZoomRotation(uv, vec2(0.5));
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float tSpeed = 0.2;
    float t = time_f * tSpeed;
    float strength = 0.5 + (ampControl * 0.5);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec3 col = sampleFractal(uv, t, strength, center, iResolution);
    color.rgb = applyColorAdjustments(col);
    color.a = 1.0;
}

```

Shader #568 - halluc gem2

File: halluc_gem2.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Reduced iterations slightly for speed Hybrid: Liquid movement + Geometric Folding Add liquid wobble to the coordinates

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;
const float iSaturation = 1.0;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 rotateHue(vec3 col, float angle) {
    float U = cos(angle); float W = sin(angle);

```

```

    mat3 R = mat3(
        0.299+0.701*U+0.168*W, 0.587-0.587*U+0.330*W, 0.114-0.114*U-0.497*W,
        0.299-0.299*U-0.328*W, 0.587+0.413*U+0.035*W, 0.114-0.114*U+0.292*W,
        0.299-0.300*U+1.250*W, 0.587-0.588*U-1.050*W, 0.114+0.886*U-0.203*W
    );
    return clamp(R * col, 0.0, 1.0);
}

vec3 applyColorAdjustments(vec3 col) {
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    col = rotateHue(col, iHueShift);
    return clamp(col, 0.0, 1.0);
}

vec2 applyZoomRotation(vec2 uv, vec2 center) {
    vec2 p = uv - center;
    float c = cos(iRotation); float s = sin(iRotation);
    p = mat2(c, -s, s, c) * p;
    float z = max(abs(iZoom), 0.001);
    p /= z;
    return p + center;
}

vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}

vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
    vec2 deriv = fwidth(tc);
    if (deriv.x > 0.0 || deriv.y > 0.0) lod = log2(max(max(deriv.x, deriv.y) * max(ts.x, ts.y), 1.0));
    return textureLod(tex, sampleUV, lod);
}

float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}

float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}

float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        v += a * noise(p);
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}

vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}

```

```

vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}

vec3 sampleLiquidFractal(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float freqControl = clamp(iFrequency, 0.0, 2.0);
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    float n1 = fbm(p * 3.0 + t * 0.4);
    float n2 = fbm(p * 2.0 - t * 0.3);
    vec2 flow = vec2(cos(n1 * 6.0), sin(n2 * 6.0));
    p += flow * (0.05 + 0.1 * strength);
    float slices = 6.0 + floor(ampControl * 4.0);
    p = kaleido(p, slices);
    int iterations = 4 + int(ampControl * 2.0);
    float scale = 1.2 + (freqControl * 0.5);
    float shift = 0.1 * strength;
    float angle = t * 0.1;
    for(int i = 0; i < iterations; i++) {
        p = abs(p);
        p -= shift;
        p *= scale;
        p = rotate2D(p, angle + float(i)*0.5 + n1*0.2);
    }
    vec2 finalUV = p * 0.5 + center + (flow * 0.02);
    float chroma = 0.005 * strength * (1.0 + n1);
    float r = mxTexture(samp, finalUV + vec2(chroma, 0.0)).r;
    float g = mxTexture(samp, finalUV).g;
    float b = mxTexture(samp, finalUV - vec2(chroma, 0.0)).b;
    return vec3(r, g, b);
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    uv = applyZoomRotation(uv, vec2(0.5));
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float tSpeed = 0.3;
    float t = time_f * tSpeed;
    float strength = 0.5 + (ampControl * 0.5);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec3 col = sampleLiquidFractal(uv, t, strength, center, iResolution);
    color.rgb = applyColorAdjustments(col);
    color.a = 1.0;
}

```

Shader #570 - halluc pop

File: halluc_pop.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Applies matrix transformations. Bumped default contrast for "Pop" Correct aspect ratio for calculations Initial Zoom base

Technique: Indexed color palette look-up table - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.2;
const float iSaturation = 1.2;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d + iHueShift));
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    return textureLod(tex, sampleUV, 0.0);
}
mat2 rot(float a) {
    float c = cos(a), s = sin(a);
    return mat2(c, -s, s, c);
}
vec3 fractalLayer(vec2 uv, vec2 center, float t) {
    vec3 finalCol = vec3(0.0);
    vec2 p = uv - center;
    p.x *= iResolution.x / iResolution.y;
    float zoomFactor = 0.5 + (iZoom * 2.0);
    p /= zoomFactor;
    float d = 100.0;
    vec3 accTexture = vec3(0.0);
    float accLight = 0.0;
    float iter = 4.0;
    float scale = 1.5 + (iAmplitude * 0.5);
    for (float i = 0.0; i < iter; i++) {
        p = abs(p);
        p -= 0.25 * iAmplitude;
        p *= rot(t * 0.2 + iRotation + i);
        p *= scale;
        vec2 texUV = p * 0.5 + 0.5;
        texUV += vec2(sin(t + i), cos(t + i)) * 0.05;
        vec3 tex = mxTexture(samp, texUV).rgb;
        float weight = 1.0 / (i + 1.0);
```

```

        accTexture += tex * weight;
        float dist = length(p);
        d = min(d, dist);
        accLight += exp(-10.0 * abs(dist - 0.5));
    }
    accTexture /= 1.5;
    vec3 pal = palette(length(p) * 0.1 + d + t * 0.2);
    finalCol = mix(accTexture, pal, 0.4 * iSaturation);
    finalCol += pal * accLight * 0.5 * iContrast;
    finalCol = pow(finalCol, vec3(1.1));
    return finalCol;
}

void main() {
    vec2 uv = tc;
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    float t = time_f * (0.1 + iFrequency * 0.2);
    vec3 col = fractalLayer(uv, center, t);
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    col = mix(vec3(gray), col, iSaturation);
    col = (col - 0.5) * iContrast + 0.5;
    col *= iBrightness;
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = pow(vUV.x * vUV.y * 15.0, 0.15);
    col *= vig;
    color = vec4(col, 1.0);
}

```

Shader #571 - halluc pop2

File: halluc_pop2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Applies matrix transformations. Controls how much "fractal" mixes in Create a centered UV system for calculations Fix aspect for math

Technique: Indexed color palette look-up table - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;
const float iSaturation = 1.0;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
}

```

```

    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d + iHueShift));
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    return textureLod(tex, sampleUV, 0.0);
}
mat2 rot(float a) {
    float c = cos(a), s = sin(a);
    return mat2(c, -s, s, c);
}
void main() {
    vec2 uv = tc;
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec2 p = uv - center;
    p.x *= iResolution.x / iResolution.y;
    float zoom = 1.0 + iZoom;
    p /= zoom;
    float t = time_f * (0.1 + iFrequency * 0.1);
    vec2 fractalUV = p;
    float accum = 0.0;
    float iter = 3.0;
    for (float i = 0.0; i < iter; i++) {
        fractalUV = abs(fractalUV);
        fractalUV -= 0.1 * iAmplitude;
        fractalUV *= rot(t * 0.5 + iRotation + i);
        fractalUV *= 1.2;
        accum += length(fractalUV);
    }
    vec2 anchorUV = (p * rot(iRotation * 0.5)) + center;
    vec3 colBase = mxTexture(samp, anchorUV).rgb;
    vec2 fracSampUV = fractalUV + center;
    vec3 colFrac = mxTexture(samp, fracSampUV).rgb;
    float mask = smoothstep(0.2, 1.0, sin(accum * 5.0 - t * 2.0));
    vec3 finalCol = mix(colBase, colFrac, 0.3 * iAmplitude);
    vec3 popColor = palette(accum + t);
    float edge = pow(mask, 4.0);
    finalCol += popColor * edge * 0.5 * iContrast;
    float gray = dot(finalCol, vec3(0.299, 0.587, 0.114));
    finalCol = mix(vec3(gray), finalCol, iSaturation);
    finalCol = (finalCol - 0.5) * iContrast + 0.5;
    finalCol *= iBrightness;
    color = vec4(finalCol, 1.0);
}

```

Shader #576 - hue mouse

File: hue-mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling

effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
const int SEGMENTS = 6;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec4 adjustHue(vec4 c, float a) {
    float U = cos(a);
    float W = sin(a);
    mat3 R = mat3(
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114
    ) + mat3(
        0.701, -0.587, -0.114,
        -0.299, 0.413, -0.114,
        -0.300, -0.588, 0.886
    ) * U
    + mat3(
        0.168, 0.330, -0.497,
        -0.328, 0.035, 0.292,
        1.250, -1.050, -0.203
    ) * W;
    return vec4(R * c.rgb, c.a);
}
void main() {
    vec2 m = (iMouse.z > 0.5 || iMouse.w > 0.5) ? iMouse.xy / iResolution : vec2(0.5);
    vec2 uvn = (tc - m) * iResolution / min(iResolution.x, iResolution.y);
    float r = length(uvn);
    float ang = atan(uvn.y, uvn.x);
    float seg = 6.28318530718 / float(SEGMENTS);
    float swirlBase = 2.5;
    float swirlTime = time_f * 0.5;
    float swirlMouse = mix(0.0, 3.0, smoothstep(0.0, 0.6, r));
    ang += (swirlBase + swirlMouse) * sin(swirlTime + r * 4.0);
    ang = mod(ang, seg);
    ang = abs(ang - seg * 0.5);
    vec2 kUV = vec2(cos(ang), sin(ang)) * r;
    float rip = sin(r * 12.0 - pingPong(time_f, 10.0) * 10.0) * exp(-r * 4.0);
    kUV += rip * 0.01;
    vec2 scale = vec2(1.0) / (iResolution / min(iResolution.x, iResolution.y));
    vec2 st = kUV * scale + m;
    float off = 0.003 * sin(time_f * 0.5);
    vec4 c = texture(samp, st);
    c += texture(samp, st + vec2(off, 0.0));
    c += texture(samp, st + vec2(-off, off));
    c += texture(samp, st + vec2(off * 0.5, -off));
    c *= 0.25;
    float hueShift = time_f * 2.0 + rip * 2.0;
    vec4 hc = adjustHue(c, hueShift);
}
```



```

    vec3 rgb = hc.rgb;
    float avg = (rgb.r + rgb.g + rgb.b) / 3.0;
    rgb = mix(vec3(avg), rgb, 1.5);
    rgb *= 1.1;
    vec4 outc = vec4(rgb, 1.0);
    outc = mix(clamp(outc, 0.0, 1.0), texture(samp, tc), 0.5);
    outc = sin(outc * pingPong(time_f, 10.0) + 2.0);
    outc.a = 1.0;
    color = outc;
}

```

Shader #582 - huri af

File: huri_af.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float pingPong(float x, float l){float m=mod(x,l*2.0);return m<=l?m:l*2.0-m;}
float hash(vec2 p){return fract(sin(dot(p,vec2(127.1,311.7))))*43758.5453);}
float noise(vec2 p){vec2 i=floor(p),f=fract(p);vec2 u=f*f*(3.0-2.0*f);float a=hash(i),b=hash(i+vec2(1,0)
    ↪ ),c=hash(i+vec2(0,1)),d=hash(i+vec2(1,1));return mix(mix(a,b,u.x),mix(c,d,u.x),u.y);}
vec3 rainbow(float t){t=fract(t);float r=abs(t*6.0-3.0)-1.0;float g=2.0-abs(t*6.0-2.0);float b=2.0-abs(t
    ↪ *6.0-4.0);return clamp(vec3(r,g,b),0.0,1.0);}
vec2 kaleido(vec2 uv, vec2 c, float seg, vec2 ar){
    vec2 p=(uv-c)*ar;
    float ang=atan(p.y,p.x);
    float ra=length(p);
    float a=6.2831853/seg;
    ang=mod(ang,a);
    ang=abs(ang-a*0.5);
    p=vec2(cos(ang),sin(ang))*ra;
    return c+p/ar;
}
vec2 vortex(vec2 uv, vec2 c, float t){
    vec2 d=uv-c;
    float r=length(d);
    float a=atan(d.y,d.x);
    a=sin(t*0.9+r*12.0)*0.6+0.3*sin(r*24.0-t*1.7);
    float zoom=0.15*sin(t*0.6+pow(r,0.6)*10.0);
    r=clamp(r*(1.0-zoom),0.0,1.0);
    return c+vec2(cos(a),sin(a))*r;
}
vec4 blur9(sampler2D img, vec2 uv, vec2 res){
    vec2 ts=1.0/res;
    vec4 s=vec4(0.0);
    s+=texture(img,uv+ts*vec2(-1,-1));
    s+=texture(img,uv+ts*vec2( 0,-1));
    s+=texture(img,uv+ts*vec2( 1,-1));

```

```

    s+=texture(img,uv+ts*vec2(-1, 0));
    s+=texture(img,uv+ts*vec2( 0, 0));
    s+=texture(img,uv+ts*vec2( 1, 0));
    s+=texture(img,uv+ts*vec2(-1, 1));
    s+=texture(img,uv+ts*vec2( 0, 1));
    s+=texture(img,uv+ts*vec2( 1, 1));
    return s/9.0;
}
void main(){
    vec2 ar=vec2(1.0,iResolution.y/iResolution.x);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 uv=tc;
    uv=kaleido(uv,m,6.0,ar);
    uv=vortex(uv,m,time_f);
    float r1=0.003+0.003*sin(time_f*1.1);
    float r2=-r1;
    vec2 dir=normalize(uv-m+1e-6);
    vec2 o1=dir*r1, o2=dir*r2;
    float g1=noise(uv*vec2(iResolution.x/iResolution.y,1.0)*5.0+time_f*0.7);
    vec2 tw=dir*(0.004*(g1-0.5));
    vec4 b=blur9(samp,uv,iResolution);
    vec4 cR=texture(samp, uv+o1+tw);
    vec4 cG=texture(samp, uv+tw);
    vec4 cB=texture(samp, uv+o2+tw);
    vec3 texRGB=mix(b.rgb, vec3(cR.r,cG.g,cB.b), 0.85);
    vec2 p=(uv-m)*ar;
    float rad=length(p);
    float ang=atan(p.y,p.x);
    float beat=0.5+0.5*sin(time_f*2.1);
    vec3 base=rainbow(ang/6.2831853+time_f*0.12);
    vec3 glow=base*smoothstep(0.85,0.0,rad)*beat;
    float strobe=0.5+0.5*sin(time_f*10.0+rad*40.0);
    float w=0.35+0.25*strobe;
    vec3 neon=mix(texRGB, texRGB*rainbow(time_f*0.08+rad*2.0), w);
    neon+=glow*0.7;
    float vign=1.0-smoothstep(0.7,1.05,rad);
    vec3 grad=mix(neon, mix(base, neon, 0.6), 0.4);
    color=vec4(grad*vign,1.0);
}

```

Shader #583 - huri create mouse

File: huri_create_mouse.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float pingPong(float x, float l){float m=mod(x,l*2.0);return m<=l?m:l*2.0-m;}
float hash(vec2 p){return fract(sin(dot(p,vec2(127.1,311.7)))*43758.5453);}

```

```

float noise(vec2 p){vec2 i=floor(p),f=fract(p);vec2 u=f*f*(3.0-2.0*f);float a=hash(i),b=hash(i+vec2(1,0)
    ↪ ),c=hash(i+vec2(0,1)),d=hash(i+vec2(1,1));return mix(mix(a,b,u.x),mix(c,d,u.x),u.y);}
vec3 rainbow(float t){t=fract(t);float r=abs(t*6.0-3.0)-1.0;float g=2.0-abs(t*6.0-2.0);float b=2.0-abs(t
    ↪ *6.0-4.0);return clamp(vec3(r,g,b),0.0,1.0);}
vec2 kaleido(vec2 uv, vec2 c, float seg, vec2 ar){
    vec2 p=(uv-c)*ar;
    float ang=atan(p.y,p.x);
    float ra=length(p);
    float a=3.14159265*2.0/seg;
    ang=mod(ang,a);
    ang=abs(ang-a*0.5);
    p=vec2(cos(ang),sin(ang))*ra;
    return c+p/ar;
}
vec2 vortex(vec2 uv, vec2 c, float t){
    vec2 d=uv-c;
    float r=length(d);
    float a=atan(d.y,d.x);
    a+=sin(t*0.9+r*12.0)*0.6+0.3*sin(r*24.0-t*1.7);
    float zoom=0.15*sin(t*0.6+pow(r,0.6)*10.0);
    r=clamp(r*(1.0-zoom),0.0,1.0);
    return c+vec2(cos(a),sin(a))*r;
}
void main(){
    vec2 ar=vec2(1.0,iResolution.y/iResolution.x);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 uv=tc;
    uv=kaleido(uv,m,6.0,ar);
    uv=vortex(uv,m,time_f);
    float r1=0.002+0.002*sin(time_f*1.3);
    float r2=-r1;
    vec2 dir=normalize(uv-m+1e-6);
    vec2 o1=dir*r1, o2=dir*r2;
    float g1=noise(uv*vec2(iResolution.x/iResolution.y,1.0)*5.0+time_f*0.7);
    float tw=0.002*(g1-0.5);
    vec4 cR=texture(samp, uv+o1+tw);
    vec4 cG=texture(samp, uv+tw);
    vec4 cB=texture(samp, uv+o2+tw);
    vec3 texRGB=vec3(cR.r,cG.g,cB.b);
    vec2 p=(uv-m)*ar;
    float rad=length(p);
    float ang=atan(p.y,p.x);
    float beat=0.5+0.5*sin(time_f*2.1);
    vec3 glow=rainbow(ang/(6.28318)+time_f*0.15)*smoothstep(0.9,0.0,rad)*beat;
    float strobe=smoothstep(0.0,1.0,0.5+0.5*sin(time_f*10.0+rad*40.0));
    vec3 neon=mix(texRGB, texRGB*rainbow(time_f*0.1+rad*2.0), 0.35+0.25*strobe);
    neon += glow*0.7;
    float vign=1.0-smoothstep(0.7,1.05,rad);
    color=vec4(neon*vign,1.0);
}

```

Shader #593 - inthesky

File: inthesky.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position and audio input (amplitude/volume) for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Color quantization via floor

rounding - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
float pp(float x, float len){float m=mod(x,len*2.0);return m<=len?m:len*2.0-m;}
vec2 rot(vec2 uv, float a, vec2 c, float asp){
    float s=sin(a),co=cos(a);
    vec2 p=uv-c; p.x*=asp;
    p=mat2(co,-s,s,co)*p;
    p.x/=asp; return p+c;
}
vec2 reflectUV(vec2 uv, float seg, vec2 c, float asp){
    vec2 p=uv-c; p.x*=asp;
    float ang=atan(p.y,p.x);
    float r=length(p);
    float stepA=6.28318530718/seg;
    ang=mod(ang,stepA);
    ang=abs(ang-stepA*0.5);
    vec2 q=vec2(cos(ang),sin(ang))*r; q.x/=asp;
    return q+c;
}
vec2 fractalZoom(vec2 uv, float zoom, float t, vec2 c, float asp){
    vec2 p=uv;
    for(int i=0;i<5;i++){
        p=abs((p-c)*zoom)-0.5+c;
        p=rot(p,t*0.1,c,asp);
    }
    return p;
}
void main(){
    float asp=iResolution.x/iResolution.y;
    vec2 m0=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 center=vec2(0.5)+(m0-vec2(0.5))*(1.0+amp*0.5*sin(time_f));
    vec2 uv=tc;
    uv += (uv-center)*0.05*amp*sin(time_f*1.3);
    vec4 originalTexture=texture(samp,uv);
    float seg=6.0+floor(amp*2.0*sin(time_f*0.7));
    vec2 kUV=reflectUV(uv,seg,center,asp);
    float zoom=1.5+0.5*sin(time_f*0.5)+0.2*amp*sin(time_f*1.7);
    kUV=fractalZoom(kUV,zoom,time_f,center,asp);
    kUV=rot(kUV,time_f*0.2*(1.0+0.6*amp),center,asp);
    vec4 kaleidoColor=texture(samp,clamp(kUV,0.0,1.0));
    float blendFactor=0.6;
    vec4 blendedColor=mix(kaleidoColor,originalTexture,blendFactor);
    blendedColor.rgb*=0.5+0.5*sin(vec3(kUV.x,kUV.y,kUV.x)+time_f*(1.0+0.5*amp));
    vec4 t0=texture(samp,uv);
    vec4 c0=sin(blendedColor*pp(time_f*(1.0+0.3*amp),10.0));
    vec4 c1=c0*t0*(0.8+0.15*amp);
    color=sin(c1*pp(time_f*(1.0+0.2*amp),15.0));
    color.a=1.0;
}
```

Shader #599 - kale mouse

File: kale_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0,c,-s,0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s,0,1,0,-s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0,s,c,0,0,0,1);}
void main(void) {
    float asp=iResolution.x/iResolution.y;
    vec2 ar=vec2(asp,1.0);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 p=(tc-m)*ar;
    vec3 v=vec3(p,1.0);
    float ax=0.28*sin(time_f*0.7);
    float ay=0.24*cos(time_f*0.6);
    float az=time_f*0.7;
    vec3 r=(rotZ(az)*rotY(ay)*rotX(ax))*v;
    float zf=1.0/(1.0+0.7*r.z);
    vec2 q=r.xy*zf;
    float N=6.0;
    float stepA=6.28318530718/N;
    float ang=atan(q.y,q.x);
    float rad=length(q);
    ang=mod(ang,stepA);
    ang=abs(ang-stepA*0.5);
    vec2 k=vec2(cos(ang),sin(ang))*rad;
    vec2 uv=k/ar+m;
    uv=clamp(uv,0.0,1.0);
    color=texture(samp,uv);
}
```

Shader #601 - kscopic 3d mouse

File: kscopic_3d_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
```

```

mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
void main(void){
    float aspect=iResolution.x/iResolution.y;
    vec2 ar=vec2(aspect,1.0);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 p=(tc-m)*ar;
    vec3 v=vec3(p,1.0);
    float ax=0.25*sin(time_f*0.7);
    float ay=0.25*cos(time_f*0.6);
    float az=0.4*time_f;
    mat3 R=rotZ(az)*rotY(ay)*rotX(ax);
    vec3 r=R*v;
    float persp=0.6;
    float zf=1.0/(1.0+r.z*persp);
    vec2 q=r.xy*zf;
    float eps=1e-6;
    float base=1.72;
    float period=log(base);
    float t=time_f*0.5;
    float rad=length(q)+eps;
    float ang=atan(q.y,q.x)+t*0.3;
    float k=fract((log(rad)-t)/period);
    float rw=exp(k*period);
    vec2 qwrap=vec2(cos(ang),sin(ang))*rw;
    float N=8.0;
    float stepA=6.28318530718/N;
    float a=atan(qwrap.y,qwrap.x)+time_f*0.05;
    a=mod(a,stepA);
    a=abs(a-stepA*0.5);
    vec2 kdir=vec2(cos(a),sin(a));
    vec2 kaleido=kdir*length(qwrap);
    vec2 uv=kaleido/ar+m;
    uv=fract(uv);
    color=texture(samp,uv);
}

```

Shader #627 - matrix mouse

File: matrix_mouse.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
float hash21(vec2 p){p=fract(p*vec2(123.34,456.21));p+=dot(p,p+34.45);return fract(p.x*p.y);}
mat2 rot(float a){float c=cos(a),s=sin(a);return mat2(c,-s,s,c);}
void main(){
    vec2 ar = vec2(iResolution.x/iResolution.y,1.0);
    vec2 m = (iMouse.z>0.5)? (iMouse.xy/iResolution) : vec2(0.5);
    vec2 p = (tc - m)*ar;
    float r = length(p);

```

```

float base = mix(0.015,0.08,smoothstep(0.0,0.9,r));
float wob = 0.012*sin(time_f*0.9+5.0*r);
float ts = base + wob;
vec2 gid = floor(p/ts);
vec2 cellCenter = (gid+0.5)*ts;
float ang = 0.6*sin(time_f*0.7 + hash21(gid)*6.2831);
vec2 local = rot(ang)*(p-cellCenter);
vec2 jitter = (hash21(gid+13.7)-0.5)*0.25*ts*vec2(cos(time_f+gid.x),sin(time_f*1.3+gid.y));
vec2 uv = (cellCenter + jitter)/ar + m;
vec4 tex = texture(samp, uv + local*0.35);
vec2 edge = abs(fract(p/ts)-0.5);
float border = smoothstep(0.01,0.0,min(edge.x,edge.y));
vec3 tint = mix(tex.rgb, vec3(0.0,1.0,0.2), 0.25+0.25*sin(time_f+hash21(gid)*3.7));
vec3 col = mix(vec3(0.08,0.12,0.08), tint, border);
float vig = 1.0 - 0.65*smoothstep(0.7,1.1,length((tc-0.5)*vec2(ar.x,1.0)));
color = vec4(col*vig,1.0);
color = mix(color, texture(samp, tc), 0.5);
}

```

Shader #646 - mirror bubble zoom mouse

File: mirror-bubble-zoom-mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
void main(void){
    float aspect=iResolution.x/iResolution.y;
    vec2 ar=vec2(aspect,1.0);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 cv = 1.0 - abs(1.0 - 2.0 * tc);
    cv = cv - floor(cv);
    vec2 p=(cv-m)*ar;
    vec3 v=vec3(p,1.0);
    float ax=0.25*sin(time_f*0.7);
    float ay=0.25*cos(time_f*0.6);
    float az=0.4*time_f;
    mat3 R=rotZ(az)*rotY(ay)*rotX(ax);
    vec3 r=R*v;
    float persp=0.6;
    float zf=1.0/(1.0+r.z*persp);
    vec2 q=r.xy*zf;
    float eps=1e-6;
    float base=1.72;
    float period=log(base);
    float t=time_f*0.5;
    float rad=length(q)+eps;
    float ang=atan(q.y,q.x)+t*0.3;
}

```

```

float k=fract((log(rad)-t)/period);
float rw=exp(k*period);
vec2 qwrap=vec2(cos(ang),sin(ang))*rw;
float N=8.0;
float stepA=6.28318530718/N;
float a=atan(qwrap.y,qwrap.x)+time_f*0.05;
a=mod(a,stepA);
a=abs(a-stepA*0.5);
vec2 kdir=vec2(cos(a),sin(a));
vec2 kaleido=kdir*length(qwrap);
vec2 uv=kaleido/ar+m;
uv=fract(uv);
color=texture(samp,uv);
}

```

Shader #647 - mirror bubble

File: mirror-bubble.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    float len = length(uv);
    float time_t = pingPong(time_f, 10.0);
    float bubble = smoothstep(0.8, 1.0, 1.0 - len);
    bubble = sin(bubble * time_t);
    vec2 distort = uv * (1.0 + 0.1 * sin(time_f + len * 20.0));
    distort = sin(distort * time_t);
    vec4 texColor = texture(samp, distort * 0.5 + 0.5);
    color = mix(texColor, vec4(1.0, 1.0, 1.0, 1.0), bubble);
}

```

Shader #650 - mirror color mouse

File: mirror-color-mouse.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core

```



```

out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
void main(void) {
    vec2 cv = 1.0 - abs(1.0 - 2.0 * tc);
    cv = cv - floor(cv);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = (cv - m) * vec2(iResolution.x / iResolution.y, 1.0);
    float t = time_f * 0.7;
    float beat = abs(sin(time_f * 3.14159)) * 0.2 + 0.8;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t * 0.5;
    float radMod = pingPong(radius + t * 0.3, 0.5);
    float wave = sin(radius * 10.0 - t * 6.0) * 0.5 + 0.5;
    float distortion = sin((radius + t * 0.5) * 8.0) * beat * 0.1;
    vec2 dir = vec2(cos(angle), sin(angle));
    vec2 tcSample = tc + dir * distortion;
    vec3 texColor = texture(samp, tcSample).rgb;
    float noiseEffect = noise(uv * 10.0 + t * 0.5) * 0.2;
    float r = sin(angle * 3.0 + radMod * 8.0 + wave * 6.2831 + noiseEffect);
    float g = sin(angle * 4.0 - radMod * 6.0 + wave * 4.1230 + noiseEffect);
    float b = sin(angle * 5.0 + radMod * 10.0 - wave * 3.4560 - noiseEffect);
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    col = mix(col, texColor, 0.6);
    vec3 lightDir = normalize(vec3(0.5, 0.5, 1.0));
    float d = dot(uv, uv);
    float z = sqrt(max(0.0, 1.0 - d));
    vec3 norm = normalize(vec3(uv, z));
    float light = dot(norm, lightDir) * 0.5 + 0.5;
    col *= light * 1.2;
    col *= beat;
    color = vec4(col, alpha);
}

```

Shader #653 - mirror comb mouse

File: mirror-comb-mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec2 rotateUV(vec2 uv, float angle, vec2 center) {
    float s = sin(angle);
    float c = cos(angle);
    uv -= center;
    uv = mat2(c, -s, s, c) * uv;
    uv += center;
    return uv;
}

vec2 reflectUV(vec2 uv, float segments, vec2 center) {
    float angle = atan(uv.y - center.y, uv.x - center.x);
    float radius = length(uv - center);
    float segmentAngle = 2.0 * 3.14159265359 / segments;
    angle = mod(angle, segmentAngle);
    angle = abs(angle - segmentAngle * 0.5);
    return vec2(cos(angle), sin(angle)) * radius + center;
}

vec2 fractalZoom(vec2 uv, float zoom, float time, vec2 center) {
    for (int i = 0; i < 5; i++) {
        uv = abs((uv - center) * zoom) - 0.5 + center;
        uv = rotateUV(uv, sin(time * pingPong(time_f, 10.0)) * 0.1, center);
    }
    return uv;
}

void main() {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    uv = uv * iResolution / vec2(iResolution.y);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 center = m * iResolution / vec2(iResolution.y);
    vec4 originalTexture = texture(samp, tc);
    vec2 kaleidoUV = reflectUV(uv, 6.0, center);
    float zoom = 1.5 + 0.5 * sin(time_f * 0.5);
    kaleidoUV = fractalZoom(kaleidoUV, zoom, time_f, center);
    kaleidoUV = rotateUV(kaleidoUV, time_f * 0.2, center);
    vec4 kaleidoColor = texture(samp, kaleidoUV);
    float blendFactor = 0.6;
    vec4 blendedColor = mix(kaleidoColor, originalTexture, blendFactor);
    blendedColor.rgb *= 0.5 + 0.5 * sin(kaleidoUV.xyx + time_f);
    color = sin(blendedColor * pingPong(time_f, 10.0));
    vec4 t = texture(samp, tc);
    color = color * t * 0.8;
    color = sin(color * pingPong(time_f, 15.0));
    color.a = 1.0;
}
```

Shader #654 - mirror fish mouse

File: mirror-fish-mouse.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control.

Technique: Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : (length * 2.0 - m);
}
float h1(float n){ return fract(sin(n)*43758.5453123); }
vec2 h2(float n){ return fract(sin(vec2(n,n+1.0))*vec2(43758.5453,22578.1459)); }
void main(void){
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec2 m = iMouse.xy / iResolution;
    bool hasMouse = iMouse.z > 0.5 || iMouse.w > 0.5;
    float rate = 0.8;
    float t = time_f * rate;
    float t0 = floor(t);
    float a = fract(t);
    float w = a*a*(3.0-2.0*a);
    vec2 p0 = vec2(0.15) + h2(t0) * 0.7;
    vec2 p1 = vec2(0.15) + h2(t0 + 1.0) * 0.7;
    vec2 autoCenter = mix(p0, p1, w);
    vec2 center = hasMouse ? m : autoCenter;
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    uv = uv - center;
    float lenv = length(uv);
    float factor = sqrt(lenv) * 0.5;
    float s = 1.0 + sin(factor * time_t);
    uv *= s;
    color = texture(samp, uv + center);
}
```

Shader #655 - mirror frac mouse z2

File: mirror-frac-mouse-z2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
```

```

uniform float time_f;
uniform vec4 iMouse;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*0.35+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec4 originalTexture = texture(samp, tc);
    float seg = 6.0 + 2.0*sin(time_f*0.33);
    vec2 kUV = reflectUV(uv, seg, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * 0.23, m, aspect);
    vec2 p = (kUV - m) * ar;
    float base = 1.82 + 0.18*sin(time_f*0.2);
    float period = log(base);
    float tz = time_f * 0.65;
    float r = length(p) + 1e-6;
    float ang = atan(p.y, p.x) + tz * 0.35 + 0.35*sin(r*9.0 + time_f*0.8);
    float k = fract((log(r) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap*1.045) / ar + m);
}

```

```

    vec2 u2 = fract((pwrap*0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * (0.0015 + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
    float hue = fract(ang*0.15 + time_f*0.08 + k*0.5);
    float sat = 0.8 - 0.2*cos(time_f*0.7 + r*6.0);
    float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
    vec3 tint = hsv2rgb(vec3(hue, sat, val));
    float ring = smoothstep(0.0, 0.7, sin(log(r+1e-3)*8.0 + time_f*1.2));
    float pulse = 0.5 + 0.5*sin(time_f*2.0 + r*18.0 + k*12.0);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
    vign = mix(0.85, 1.2, vign);
    float blendFactor = 0.58;
    float rC = texture(samp, u0 + off).r;
    float gC = texture(samp, u1).g;
    float bC = texture(samp, u2 - off).b;
    vec3 kaleidoRGB = vec3(rC, gC, bC);
    vec4 kaleidoColor = vec4(kaleidoRGB, 1.0) * vec4(tint, 1.0);
    vec4 merged = mix(kaleidoColor, originalTexture, blendFactor);
    merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
    vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
    vec3 outCol = merged.rgb + bloom;
    float wob = 0.9 + 0.1*sin(time_f + r*12.0 + k*9.0);
    outCol *= wob;
    vec4 t = texture(samp, tc);
    outCol = mix(outCol, outCol*t.rgb, 0.8);
    outCol = sin(outCol * (0.5 + 0.5*pingPong(time_f, 12.0)));
    outCol = clamp(outCol, vec3(0.08), vec3(0.96));
    color = vec4(outCol, 1.0);
}

```

Shader #722 - neon mouse

File: neon_mouse.glsl

Applies a time-animated visual transformation to the input texture. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform vec4 iMouse;
vec3 overlayBlend(vec3 base, vec3 blend, float opacity) {
    vec3 c2 = blend * 2.0;
    vec3 c1 = 1.0 - 2.0 * (1.0 - blend);
    vec3 result = mix(base * c2, c1, step(0.5, base));
    return mix(base, result, opacity);
}
void main(void) {
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec2 p = (uv - m) * ar;
    float dist = length(p);

```

```

float angle = atan(p.y, p.x) + time_f * 5.0;
float spiral = cos(10.0 * dist - angle);
float ring = smoothstep(0.1, 0.2, abs(spiral) - dist * 0.5);
float falloff = 1.0 - smoothstep(0.0, 0.6, dist);
float mask = clamp(ring * falloff, 0.0, 1.0);
vec3 neon = vec3(value_alpha_r, value_alpha_g, value_alpha_b);
vec3 base = texture(samp, uv).rgb;
vec3 blended = overlayBlend(base, neon, mask);
color = vec4(blended, 1.0);
}

```

Shader #757 - PI x1.mp4

File: PI_x1.mp4.gls1

Performs edge detection on the input image, highlighting boundaries and contours. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Sobel/Laplacian convolution kernel for edge detection - Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float hash(vec2 p){return fract(sin(dot(p,vec2(127.1,311.7))))*43758.5453);}
float noise(vec2 p){vec2 i=floor(p),f=fract(p);vec2 u=f*f*(3.0-2.0*f);float a=hash(i),b=hash(i+vec2(1,0)
    ↪ ),c=hash(i+vec2(0,1)),d=hash(i+vec2(1,1));return mix(mix(a,b,u.x),mix(c,d,u.x),u.y);}
vec3 rainbow(float t){t=fract(t);float r=abs(t*6.0-3.0)-1.0;float g=2.0-abs(t*6.0-2.0);float b=2.0-abs(t
    ↪ *6.0-4.0);return clamp(vec3(r,g,b),0.0,1.0);}
vec2 kaleido(vec2 uv, vec2 c, float seg, vec2 ar){
    vec2 p=(uv-c)*ar;
    float ang=atan(p.y,p.x);
    float ra=length(p);
    float a=6.2831853/seg;
    ang=mod(ang,a);
    ang=abs(ang-a*0.5);
    p=vec2(cos(ang),sin(ang))*ra;
    return c+p/ar;
}
vec2 vortex(vec2 uv, vec2 c, float t){
    vec2 d=uv-c;
    float r=length(d);
    float a=atan(d.y,d.x);
    a+=sin(t*0.9+r*12.0)*0.6+0.3*sin(r*24.0-t*1.7);
    float z=0.15*sin(t*0.6+pow(r,0.6)*10.0);
    r=clamp(r*(1.0-z),0.0,1.0);
    return c+vec2(cos(a),sin(a))*r;
}
vec4 blur9(sampler2D img, vec2 uv, vec2 res){

```

```

    vec2 ts=1.0/res;
    vec4 s=vec4(0.0);
    s+=texture(img,uv+ts*vec2(-1,-1));
    s+=texture(img,uv+ts*vec2( 0,-1));
    s+=texture(img,uv+ts*vec2( 1,-1));
    s+=texture(img,uv+ts*vec2(-1, 0));
    s+=texture(img,uv+ts*vec2( 0, 0));
    s+=texture(img,uv+ts*vec2( 1, 0));
    s+=texture(img,uv+ts*vec2(-1, 1));
    s+=texture(img,uv+ts*vec2( 0, 1));
    s+=texture(img,uv+ts*vec2( 1, 1));
    return s/9.0;
}
float luma(vec3 c){return dot(c,vec3(0.299,0.587,0.114));}
void main(){
    vec2 ar=vec2(1.0,iResolution.y/iResolution.x);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec3 orig=texture(samp, tc).rgb;
    vec2 uv=tc;
    uv=kaleido(uv,m,6.0,ar);
    uv=vortex(uv,m,time_f);
    float r1=0.003+0.003*sin(time_f*1.1);
    vec2 dir=normalize(uv-m+1e-6);
    vec2 tw=dir*(0.004*(noise(uv*vec2(iResolution.x/iResolution.y,1.0)*5.0+time_f*0.7)-0.5));
    vec4 b=blur9(samp,uv,iResolution);
    vec4 cR=texture(samp, uv+dir*r1+tw);
    vec4 cG=texture(samp, uv+tw);
    vec4 cB=texture(samp, uv-dir*r1+tw);
    vec3 texRGB=mix(b.rgb, vec3(cR.r,cG.g,cB.b), 0.85);
    vec2 p=(uv-m)*ar;
    float rad=length(p);
    float ang=atan(p.y,p.x);
    float beat=0.5+0.5*sin(time_f*2.1);
    vec3 base=rainbow(ang/6.2831853+time_f*0.12);
    vec3 glow=base*smoothstep(0.85,0.0,rad)*beat;
    float strobe=0.5+0.5*sin(time_f*10.0+rad*40.0);
    float w=0.28+0.22*strobe;
    vec3 fractal=mix(texRGB, texRGB*rainbow(time_f*0.08+rad*2.0), w)+glow*0.7;
    float L=luma(orig);
    float gx=abs(dFdx(L));
    float gy=abs(dFdy(L));
    float edge=smoothstep(0.02,0.15,gx+gy);
    float radial=smoothstep(0.05,0.65,rad);
    float band=1.0-smoothstep(0.015,0.035,abs(fract(ang*3.0/3.14159+time_f*0.08)-0.5));
    float n=0.35+0.65*noise(tc*vec2(iResolution.x/iResolution.y,1.0)*3.0+time_f*0.4);
    float strength=clamp(0.55*radial+0.25*strobe,0.0,1.0);
    strength*=mix(1.0,0.4,edge);
    strength*=mix(1.0,0.6,band);
    strength*=mix(strength,strength*n,0.6);
    vec3 screenMix=1.5-(1.0-fractal)*(1.0-orig);
    vec3 outRGB=mix(orig, sin(screenMix * pingPong(time_f * PI, 10.0)), strength);
    float vign=1.0-smoothstep(0.8,1.08,length((tc-m)*ar));
    color=vec4(outRGB*vign,1.0);
}

```

Shader #827 - rainbow mouse x1

File: rainbow_mouse_x1.glsl

Applies a time-animated visual transformation to the input texture. Responds to mouse position

for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec2 iMouse;
vec3 rainbow(float t){
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(){
    vec2 mouse = (iMouse.x + iMouse.y > 0.0) ? (iMouse / iResolution) : vec2(0.5);
    vec2 ar = vec2(1.0, iResolution.y / iResolution.x);
    vec2 p = (tc - mouse) * ar;
    float angle = atan(p.y, p.x) + time_f * 8.0;
    float r = length(p);
    vec3 rc = rainbow(angle / (2.0 * 3.14159265359));
    vec4 tex = texture(samp, tc);
    float falloff = smoothstep(0.5, 0.0, r);
    vec3 outc = mix(tex.rgb, rc, falloff);
    color = vec4(outc, tex.a);
}
```

Shader #846 - random soul by mouse

File: random_soul_by_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
vec3 rainbow(float t){
    t=fract(t);
    float r=abs(t*6.0-3.0)-1.0;
    float g=2.0-abs(t*6.0-2.0);
    float b=2.0-abs(t*6.0-4.0);
    return clamp(vec3(r,g,b),0.0,1.0);
}
void main(){
    vec2 res=iResolution;
    vec2 uv=tc*2.0-1.0;
    uv.y*=res.y/res.x;
```



```

    vec2 m=iMouse.xy/res*2.0-1.0;
    m.y*=-res.y/res.x;
    vec2 d=uv-m;
    float r=length(d);
    float tight=6.0;
    float angle=atan(d.y,d.x)+r*tight+time_f*2.0;
    float ct=cos(angle), st=sin(angle);
    vec2 rot=vec2(d.x*ct-d.y*st,d.x*st+d.y*ct)+m;
    vec3 rain=rainbow((atan(rot.y,rot.x)+time_f*2.0)/(6.28318530718));
    float bulge=0.2;
    float k=pow(length(rot),2.0)*bulge;
    vec2 tex=rot*(1.0+k)*0.5+0.5;
    vec4 base=texture(samp,tex);
    vec3 mixc=mix(base.rgb,rain,0.5);
    color=vec4(sin(mixc*time_f),base.a);
}

```

Shader #900 - ripple mouse2

File: ripple_mouse2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control.

Technique: Hermite smoothstep interpolation - Driven by: time, mouse, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float ripplePulse(vec2 uv, vec2 c, float t0){
    float d = length(uv - c);
    float k = 28.0;
    float w = 8.0;
    float a = sin(d * k - (time_f - t0) * w);
    float falloff = exp(-d * 5.5);
    float gate = smoothstep(0.0, 0.4, time_f - t0) * step(time_f - t0, 2.6);
    return a * falloff * gate;
}
void main(){
    vec2 uv = tc;
    vec2 mouse = iMouse.xy / iResolution.xy;
    float interval = 1.25;
    float baseStart = floor(time_f / interval) * interval;
    float r = 0.0;
    r += ripplePulse(uv, mouse, baseStart - 0.0*interval);
    r += ripplePulse(uv, mouse, baseStart - 1.0*interval);
    r += ripplePulse(uv, mouse, baseStart - 2.0*interval);
    vec2 dir = normalize(uv - mouse + 1e-6);
    float amp = 0.08;
    vec2 offset = dir * r * amp;
    vec4 col = texture(samp, uv + offset);
    color = col;
}

```

Shader #902 - ripple outMouse

File: ripple_outMouse.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control. Mouse interaction parameters Only when mouse is dragged Dynamic wave parameters based on drag

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
    vec2 normPos = (gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0;
    float dist = length(normPos);
    vec2 dragVec = vec2(0.0);
    float dragSpeed = 0.0;
    vec2 dragDir = vec2(0.0);
    float verticalFactor = 0.0;
    if(iMouse.z > 0.0) {
        vec2 clickPos = (iMouse.zw / iResolution.xy) * 2.0 - 1.0;
        vec2 currentPos = (iMouse.xy / iResolution.xy) * 2.0 - 1.0;
        dragVec = currentPos - clickPos;
        dragSpeed = length(dragVec);
        dragDir = normalize(dragVec);
        verticalFactor = dragVec.y;
    }
    float speedFactor = 1.0 + verticalFactor * 2.0;
    float waveSize = 0.105 + dragSpeed * 0.2;
    float phase = sin(dist * 6.0 - time_f * speedFactor);
    vec2 displacement = mix(
        normPos * waveSize * phase,
        dragDir * waveSize * phase * (1.0 + dragSpeed * 2.0),
        smoothstep(0.0, 0.5, dragSpeed)
    );
    vec2 tcAdjusted = tc + displacement;
    color = texture(samp, tcAdjusted);
}
```

Shader #916 - seek

File: seek.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position for interactive control. Base wave distortion (original intensity) Mouse interaction Radial twist effect (original strength)

Technique: Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation - Radial vignette falloff - Driven by: time, mouse, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
```

```

uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
    vec2 uv = tc;
    vec2 mouse = iMouse.xy / iResolution.xy;
    float mouseDown = step(0.5, iMouse.z);
    vec2 wave = vec2(
        sin(uv.y * 10.0 + time_f * 3.0) * 0.03,
        cos(uv.x * 8.0 + time_f * 2.5) * 0.02
    );
    vec2 toMouse = uv - mouse;
    float mouseDist = length(toMouse);
    vec2 mouseOffset = toMouse * sin(mouseDist * 20.0 - time_f * 4.0) * 0.1 * mouseDown;
    float angle = sin(time_f * 2.0 + mouseDist * 10.0) * 0.4 * mouseDown;
    mat2 rot = mat2(cos(angle), -sin(angle), sin(angle), cos(angle));
    vec2 twistedUV = mix(uv, rot * (uv - mouse) + mouse, mouseDown);
    uv = twistedUV + wave + mouseOffset;
    float chromaIntensity = 0.003;
    vec4 texR = texture(samp, uv + vec2(chromaIntensity, -chromaIntensity));
    vec4 texG = texture(samp, uv);
    vec4 texB = texture(samp, uv - vec2(chromaIntensity, -chromaIntensity));
    float ripple = 0.0;
    if (mouseDown > 0.5) {
        float dist = distance(uv, mouse);
        ripple = sin(dist * 30.0 - time_f * 10.0) * 0.08 * smoothstep(0.3, 0.0, dist);
    }
    color = vec4(texR.r, texG.g, texB.b, 1.0) + abs(ripple);
    vec2 vigUV = uv * (1.0 - uv.xy);
    float vignette = pow(vigUV.x * vigUV.y * 5.0, 0.7);
    color *= mix(1.0, vignette, 0.2);
    color *= 0.95 + 0.05 * sin(time_f * 2.0);
    color = clamp(color, 0.8, 1.2);
}

```

Shader #917 - seek2

File: seek2.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections.
 Responds to mouse position for interactive control. 1 when mouse pressed Base wave distortion
 Mouse interaction distortion

Technique: Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep
 interpolation - Linear mix/lerp blending - Distance field computation - Radial vignette falloff
 - Driven by: time, mouse, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
    vec2 uv = tc;
    vec2 mouse = iMouse.xy / iResolution.xy;
    float mouseDown = step(0.5, iMouse.z);
    vec2 wave = vec2(
        sin(uv.y * 10.0 + time_f * 3.0) * 0.03,

```

```

        cos(uv.x * 8.0 + time_f * 2.5) * 0.02
    );
    wave = (wave * (time_f * 0.25));
    vec2 toMouse = uv - mouse;
    float mouseDist = length(toMouse);
    vec2 mouseOffset = toMouse * mouseDist * 20.0 - time_f * 4.0 * 0.1 * mouseDown;
    float angle = sin(time_f * 2.0 + mouseDist * 10.0) * 0.4 * mouseDown;
    mat2 rot = mat2(cos(angle), -sin(angle), sin(angle), cos(angle));
    vec2 twistedUV = mix(uv, rot * (uv - mouse) + mouse, mouseDown);
    uv = cos(twistedUV * (0.1 * time_f)) + wave + mouseOffset;
    float chromaIntensity = 0.005 + sin(time_f) * 0.003;
    vec4 texR = texture(samp, uv + vec2(chromaIntensity, -chromaIntensity));
    vec4 texG = texture(samp, uv + vec2(-chromaIntensity, 0.0));
    vec4 texB = texture(samp, uv + vec2(0.0, chromaIntensity));
    float ripple = 0.0;
    if (mouseDown > 0.5) {
        float dist = distance(uv, mouse);
        ripple = sin(dist * 30.0 - time_f * 10.0) * 0.1 * smoothstep(0.3, 0.0, dist);
    }
    color = vec4(texR.r, texG.g, texB.b, 1.0) * (1.0 - ripple);
    vec2 vigUV = uv * (1.0 - uv.xy);
    float vignette = vigUV.x * vigUV.y * 15.0;
    vignette = pow(vignette, 0.3);
}

```

Shader #961 - spiral mouse

File: spiral-mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 mouseUV = iMouse.xy / iResolution;
    mouseUV = mouseUV * 2.0 - 1.0;
    mouseUV.x *= aspect;
    float mouseDown = step(0.0, iMouse.z);
    vec2 spiralCenter = mix(vec2(0.0), mouseUV, mouseDown);
    vec2 localUV = uv - spiralCenter;
    float d = length(uv);
    float lensStrength = 1.5;
    vec3 normal = normalize(vec3(uv, 1.0 / lensStrength));
}

```

```

float localD = length(localUV);
float fisheyeRadius = atan(localD, 1.0);
vec2 distortedUV = normalize(localUV + 1e-6) * fisheyeRadius;
float t = time_f * 0.8;
float pTime = pingPong(time_f * 0.5, 2.0);
float r_dist = length(distortedUV);
float angle = atan(distortedUV.y, distortedUV.x);
float spiral = angle + (log(r_dist + 0.1) * (2.0 + pTime)) - t * 1.5;
float r = sin(spiral * 3.0 + t);
float g = sin(spiral * 3.0 + t + 2.094);
float b = sin(spiral * 3.0 + t + 4.188);
vec3 spiralCol = vec3(r, g, b) * 0.5 + 0.5;
vec3 lightDir = normalize(vec3(sin(time_f), cos(time_f), 1.0));
float diff = max(dot(normal, lightDir), 0.0);
float spec = pow(max(dot(reflect(-lightDir, normal), vec3(0.0, 0.0, 1.0)), 0.0), 16.0);
vec3 texColor = texture(samp, tc).rgb;
float spiralMask = 1.0 - smoothstep(0.45, 1.2, localD);
vec3 spiralLit = spiralCol * (diff + 0.5) + spec;
vec3 finalCol = mix(texColor, spiralLit, 0.7 * spiralMask);
finalCol *= smoothstep(2.0, 0.5, d);
color = vec4(finalCol, alpha);
}

```

Shader #993 - swirl by mouse

File: swirl_by_mouse.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float t = time_f * 0.7;
    float beat = abs(sin(time_f * 3.14159)) * 0.2 + 0.8;
}

```

```

float radius = length(uv);
float angle = atan(uv.y, uv.x);
angle += t * 0.5;
float radMod = pingPong(radius + t * 0.3, 0.5);
float wave = sin(radius * 10.0 - t * 6.0) * 0.5 + 0.5;
float distortion = sin((radius + t * 0.5) * 8.0) * beat * 0.1;
vec2 m = (iMouse.z > 0.5 ? iMouse.xy : 0.5 * iResolution) / iResolution;
vec2 d = tc - m;
float dist = length(d);
float r = mix(0.12, 0.35, beat);
float s = smoothstep(r, 0.0, dist);
float k = 6.0 * (0.6 + 0.4 * beat);
float ang = atan(d.y, d.x) + s * (r - dist) * k;
vec2 swirlUV = m + vec2(cos(ang), sin(ang)) * dist;
vec3 texBase = texture(samp, tc + distortion).rgb;
vec3 texSwirl = texture(samp, swirlUV).rgb;
vec3 texColor = mix(texBase, texSwirl, s);
float noiseEffect = noise(uv * 10.0 + t * 0.5) * 0.2;
float rC = sin(angle * 3.0 + radMod * 8.0 + wave * 6.2831 + noiseEffect);
float gC = sin(angle * 4.0 - radMod * 6.0 + wave * 4.1230 + noiseEffect);
float bC = sin(angle * 5.0 + radMod * 10.0 - wave * 3.4560 - noiseEffect);
vec3 col = vec3(rC, gC, bC) * 0.5 + 0.5;
col = mix(col, texColor, 0.6);
vec3 lightDir = normalize(vec3(0.5, 0.5, 1.0));
vec3 norm = normalize(vec3(uv, sqrt(max(0.0, 1.0 - dot(uv, uv)))));
float light = dot(norm, lightDir) * 0.5 + 0.5;
col *= light * 1.2;
col *= beat;
color = vec4(col, alpha);
}

```

Shader #994 - swirl by mouse2

File: swirl_by_mouse2.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
float hash(vec2 p){
    return fract(sin(dot(p, vec2(127.1,311.7))) * 43758.5453123);
}
float noise(vec2 p){
    vec2 i = floor(p), f = fract(p);
    vec2 u = f*f*(3.0-2.0*f);

```

```

    float a = hash(i+vec2(0.0,0.0));
    float b = hash(i+vec2(1.0,0.0));
    float c = hash(i+vec2(0.0,1.0));
    float d = hash(i+vec2(1.0,1.0));
    return mix(mix(a,b,u.x), mix(c,d,u.x), u.y);
}

void main(void){
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    float distanceC = length(uv - center);
    float core = tan(distanceC * 20.0 - time_f * 2.0) * 0.02;
    vec2 uvTan = uv + vec2(core, core);
    vec2 uvN = (tc * iResolution - 0.5 * iResolution) / iResolution.y;
    float t = time_f * 0.7;
    float beat = abs(sin(time_f * 3.14159)) * 0.2 + 0.8;
    float radius = length(uvN);
    float angle = atan(uvN.y, uvN.x);
    float radMod = pingPong(radius + t * 0.3, 0.5);
    float wave = sin(radius * 10.0 - t * 6.0) * 0.5 + 0.5;
    float distortion = sin((radius + t * 0.5) * 8.0) * beat * 0.1;
    vec2 m = (iMouse.z > 0.5 ? iMouse.xy : 0.5 * iResolution) / iResolution;
    vec2 d = tc - m;
    float dist = length(d);
    float r = mix(0.12, 0.35, beat);
    float s = smoothstep(r, 0.0, dist);
    float k = 6.0 * (0.6 + 0.4 * beat);
    float ang = atan(d.y, d.x) + s * (r - dist) * k;
    vec2 swirlUV = m + vec2(cos(ang), sin(ang)) * dist;
    vec3 texTan = texture(samp, uvTan + distortion * 0.02).rgb;
    vec3 texSwirl = texture(samp, swirlUV).rgb;
    vec3 texMix = mix(texTan, texSwirl, s);
    float n = noise(uvN * 10.0 + t * 0.5) * 0.2;
    float rC = sin(angle * 3.0 + radMod * 8.0 + wave * 6.2831 + n);
    float gC = sin(angle * 4.0 - radMod * 6.0 + wave * 4.1230 + n);
    float bC = sin(angle * 5.0 + radMod * 10.0 - wave * 3.4560 - n);
    vec3 col = vec3(rC, gC, bC) * 0.5 + 0.5;
    col = mix(col, texMix, 0.6);
    vec3 lightDir = normalize(vec3(0.5, 0.5, 1.0));
    vec3 norm = normalize(vec3(uvN, sqrt(max(0.0, 1.0 - dot(uvN, uvN)))));
    float light = dot(norm, lightDir) * 0.5 + 0.5;
    col *= light * 1.2;
    col *= beat;
    color = vec4(col, alpha);
}

```

Shader #995 - swirl by mouse full

File: swirl_by_mouse_full.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;
float pingPong(float x,float len){float m=mod(x,len*2.0);return m<=len?m:len*2.0-m;}
float hash(vec2 p){return fract(sin(dot(p,vec2(127.1,311.7)))*43758.5453123);}
float noise(vec2 p){vec2 i=floor(p),f=fract(p);vec2 u=f*f*(3.0-2.0*f);return mix(mix(hash(i),hash(i+vec2
    ↪ (1.0,0.0)),u.x),mix(hash(i+vec2(0.0,1.0)),hash(i+vec2(1.0,1.0)),u.x),u.y);}
void main(){
    vec2 uv=(tc*iResolution-0.5*iResolution)/iResolution.y;
    float t=time_f*0.7;
    float beat=abs(sin(time_f*3.14159))*0.2+0.8;
    float radius=length(uv);
    float angle=atan(uv.y,uv.x)+t*0.5;
    float radMod=pingPong(radius+t*0.3,0.5);
    float wave=sin(radius*10.0-t*6.0)*0.5+0.5;
    float distortion=sin((radius+t*0.5)*8.0)*beat*0.1;
    vec2 m=(iMouse.z>0.5?iMouse.xy:0.5*iResolution)/iResolution;
    vec2 d=tc-m;
    float dist=length(d);
    float r=mix(0.12,0.35,beat);
    float s=smoothstep(r,0.0,dist);
    float k=6.0*(0.6+0.4*beat);
    float ang=atan(d.y,d.x)+s*(r-dist)*k;
    vec2 swirlUV=m+vec2(cos(ang),sin(ang))*dist;
    vec3 base=texture(samp,swirlUV+distortion).rgb;
    float n=noise(uv*10.0+t*0.5)*0.2;
    vec3 modv=vec3(sin(angle*3.0+radMod*8.0+wave*6.2831+n),
        sin(angle*4.0-radMod*6.0+wave*4.1230+n),
        sin(angle*5.0+radMod*10.0-wave*3.4560-n))*0.5+0.5;
    vec3 lightDir=normalize(vec3(0.5,0.5,1.0));
    vec3 norm=normalize(vec3(uv,sqrt(max(0.0,1.0-dot(uv,uv)))));
    float light=dot(norm,lightDir)*0.5+0.5;
    vec3 col=base*modv*light*1.2*beat;
    color=vec4(col,alpha);
}

```

Shader #996 - swirl by mouse pos

File: swirl_by_mouse_pos.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

```



```

}
void main(void) {
    vec2 ar = vec2(1.0, iResolution.y / iResolution.x);
    vec2 iar = vec2(1.0, iResolution.x / iResolution.y);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p = (tc - m) * ar;
    float r = length(p);
    float r0 = 0.35;
    float k = 2.5 + 1.5 * sin(time_f * 0.5);
    float t = k * (r0 - r);
    float c = cos(t), s = sin(t);
    vec2 pr = mat2(c, -s, s, c) * p;
    vec2 tcoord = m + pr * iar;
    vec2 uv = tcoord * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float wave = sin(uv.x * 10.0 + time_f * 2.0) * 0.1;
    float angle = atan(uv.y + wave, uv.x) + time_f * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tcoord);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_f), original_color.a);
}

```

Shader #998 - swirlMouse

File: swirlMouse.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Calculate mouse delta and apply rotation Normalize by resolution Rotation based on horizontal drag

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)),
        hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)),
        hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
void main(void) {
    vec2 uv = (tc * iResolution - 0.5 * iResolution) / iResolution.y;

```

```

float t = time_f * 0.7;
float radius = length(uv);
float angle = atan(uv.y, uv.x);
vec2 mouseDelta = iMouse.xy - iMouse.zw;
mouseDelta /= iResolution.xy;
angle -= mouseDelta.x * 10.0;
float radMod = pingPong(radius + t * 0.3, 0.5);
float wave = sin(radius * 10.0 - t * 6.0) * 0.5 + 0.5;
float noiseEffect = noise(uv * 10.0 + t * 0.5) * 0.2;
float r = sin(angle * 3.0 + radMod * 8.0 + wave * 6.2831 + noiseEffect);
float g = sin(angle * 4.0 - radMod * 6.0 + wave * 4.1230 + noiseEffect);
float b = sin(angle * 5.0 + radMod * 10.0 - wave * 3.4560 - noiseEffect);
vec3 col = vec3(r, g, b) * 0.5 + 0.5;
vec3 texColor = texture(samp, tc).rgb;
col = mix(col, texColor, 0.6);
vec3 lightDir = normalize(vec3(0.5, 0.5, 1.0));
vec3 norm = normalize(vec3(uv, sqrt(1.0 - dot(uv, uv))));
float light = dot(norm, lightDir) * 0.5 + 0.5;
col *= light * 1.2;
color = vec4(col, alpha);
}

```

Shader #1044 - warp ppong by mouse

File: warp_ppong_by_mouse.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control.

Technique: Hermite smoothstep interpolation - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 d = uv - m;
    float dist = length(d);
    float r = 0.35;
    float w = 1.0 - smoothstep(0.0, r, dist);
    vec2 warp = normalize(d * 1e-5) * sin(dist * 24.0 - time_f * 4.0) * 0.25 * w;
    vec2 warpedCoords = uv + warp;
    warpedCoords.x = pingPong(warpedCoords.x + time_f * 0.05, 1.0);
    warpedCoords.y = pingPong(warpedCoords.y + time_f * 0.05, 1.0);
    color = texture(samp, warpedCoords);
}

```

Shader #1045 - warp ppong mouse

File: warp_ppong_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;

float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalZoom(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<5;i++){
        p = abs((p - c) * zoom) - 0.5 + c;
        p = rotateUV(p, t*0.1, c, aspect);
    }
    return p;
}

void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    vec2 kaleidoUV = reflectUV(uv, 6.0, m, aspect);
    float zoom = 1.5 + 0.5 * sin(time_f * 0.5);
    kaleidoUV = fractalZoom(kaleidoUV, zoom, time_f, m, aspect);
    kaleidoUV = rotateUV(kaleidoUV, time_f * 0.2, m, aspect);
    vec2 p = (kaleidoUV - m) * ar;
    float base = 1.72;
    float period = log(base);
    float tz = time_f * 0.5;
```

```

float r = length(p) + 1e-6;
float ang = atan(p.y, p.x) + tz * 0.3;
float k = fract((log(r) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 zoomUV = fract(pwrap / ar + m);
vec4 kaleidoColor = texture(samp, zoomUV);
float blendFactor = 0.6;
vec4 blendedColor = mix(kaleidoColor, originalTexture, blendFactor);
blendedColor.rgb *= 0.5 + 0.5 * sin(zoomUV.xy * time_f);
color = sin(blendedColor * pingPong(time_f, 8.0));
vec4 t = texture(samp, tc);
color = sin(color * pingPong(time_f, 8.0)) * t * 0.8;
vec4 color2 = tan(color * pingPong(time_f, 8.0));
color = ((0.5 * color2) + (0.5 * color));
if(color[0] < 0.1 && color[1] < 0.1 && color[2] < 0.1)
    color = texture(samp, tc);
color.a = 1.0;
}

```

Shader #1047 - warpcursor

File: warpcursor.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. radius scaled up 3x color = mix(orig, warpedCol, s);

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
void main(void) {
    vec2 uv = tc;
    vec2 m = iMouse.xy / iResolution;
    vec2 d = uv - m;
    float dist = length(d);
    float a = clamp(amp, 0.0, 1.0);
    float ua = clamp(uamp, 0.0, 1.0);
    float r = mix(0.06, 0.35, a) * 1.5;
    float s = smoothstep(r, 0.0, dist);
    float k = 0.6 + 0.4 * ua;
    float swirl = (0.8 * ua + 0.2 * a) * s * (r - dist) * 8.0;
    float ang = atan(d.y, d.x) + swirl;
    vec2 drot = vec2(cos(ang), sin(ang)) * dist;
    float lens = 1.0 - k * s * (1.0 - dist / r);
    vec2 warped = m + drot * lens;
    float wob = sin(time_f * 3.0 + dist * 20.0) * 0.005 * ua * s;
    vec2 n = normalize(drot + vec2(1e-6));
    warped += n * wob;
    vec4 orig = texture(samp, uv);
    vec4 warpedCol = texture(samp, clamp(warped, vec2(0.0), vec2(1.0)));
}

```

```
    color = warpedCol;
}
```

Shader #1048 - water cursor

File: water-cursor.glsl

Uses distance fields with sinusoidal modulation to generate animated visual patterns. Responds to mouse position for interactive control. Added for screen size Easy-to-edit radius variable Convert mouse position to texture coordinates [0-1] range

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation
- Driven by: time, mouse, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec4 iMouse;
uniform vec2 iResolution;
float u_radius = 0.3;
void main(void) {
    vec2 mouseUV = iMouse.xy / iResolution.xy;
    float dist = distance(tc, mouseUV);
    float influence = 1.0 - smoothstep(u_radius * 0.8, u_radius, dist);
    float speed = 5.0;
    float amplitude = 0.03 * influence;
    float wavelength = 10.0;
    float rippleX = sin(tc.x * wavelength + time_f * speed) * amplitude;
    float rippleY = sin(tc.y * wavelength + time_f * speed) * amplitude;
    vec2 rippleTC = mix(tc, tc + vec2(rippleX, rippleY), influence);
    vec4 originalColor = texture(samp, tc);
    vec4 rippleColor = texture(samp, rippleTC);
    color = mix(originalColor, rippleColor, influence);
}
```

Shader #1077 - xor rgb rainbow mouse

File: xor_rgb_rainbow_mouse.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position for interactive control.

Technique: Color quantization via floor rounding - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
```

```

vec4 blur5(sampler2D image, vec2 uv, vec2 res) {
    float k1[5] = float[](1.0, 4.0, 6.0, 4.0, 1.0);
    vec2 ts = 1.0 / res;
    vec4 s = vec4(0.0);
    float wsum = 0.0;
    for (int j = -2; j <= 2; ++j) {
        for (int i = -2; i <= 2; ++i) {
            float w = k1[i+2] * k1[j+2];
            s += texture(image, uv + vec2(float(i), float(j)) * ts) * w;
            wsum += w;
        }
    }
    return s / wsum;
}

vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}

vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 r0 = random2(vec2(t0));
    vec2 r1 = random2(vec2(t1));
    float a = fract(t);
    a = a * a * (3.0 - 2.0 * a);
    return mix(r0, r1, a);
}

vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

vec4 xor_RGB(vec4 a, vec4 b) {
    uvec3 ua = uvec3(clamp(floor(a.rgb * 255.0 + 0.5), 0.0, 255.0));
    uvec3 ub = uvec3(clamp(floor(b.rgb * 255.0 + 0.5), 0.0, 255.0));
    uvec3 ux = ua ^ ub;
    return vec4(vec3(ux) / 255.0, 1.0);
}

void main(void) {
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec4 tcolor = blur5(samp, tc, iResolution);
    float tA = pingPong(time_f, 10.0) + 2.0;
    float tB = pingPong(time_f, 5.0) + 2.0;
    float tw = pingPong(time_f, 15.0) + 1.0;
    vec2 uvn = (tc - m) * ar;
    float wave = sin(uvn.x * 10.0 + tw * 2.0) * 0.1;
    vec2 rnd = smoothRandom2(tw) * 0.5;
    float expand = 0.5 + 0.5 * sin(tw * 2.0);
    vec2 suv = uvn * expand + rnd;
    float ang = atan(suv.y + wave, suv.x) + tw * 2.0;
    vec3 rb = rainbow(ang / 6.2831853);
    vec3 base = tcolor.rgb;
    vec3 mixc = mix(base, rb, 0.5);
    color = xor_RGB(sin(vec4(mixc, 1.0) * tA), tcolor * tB);
}

```

Shader #1081 - xor rgb smoother mouse

File: xor_rgb_smoother_mouse.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns. Responds to mouse position for interactive control.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5,1.0,1.5,2.0,2.5,2.5,2.0,1.5,1.0,0.5,
                                     1.0,2.0,2.5,3.0,3.5,3.5,3.0,2.5,2.0,1.0,
                                     1.5,2.5,3.0,3.5,4.0,4.0,3.5,3.0,2.5,1.5,
                                     2.0,3.0,3.5,4.0,4.5,4.5,4.0,3.5,3.0,2.0,
                                     2.5,3.5,4.0,4.5,5.0,5.0,4.5,4.0,3.5,2.5,
                                     2.5,3.5,4.0,4.5,5.0,5.0,4.5,4.0,3.5,2.5,
                                     2.0,3.0,3.5,4.0,4.5,4.5,4.0,3.5,3.0,2.0,
                                     1.5,2.5,3.0,3.5,4.0,4.0,3.5,3.0,2.5,1.5,
                                     1.0,2.0,2.5,3.0,3.5,3.5,3.0,2.5,2.0,1.0,
                                     0.5,1.0,1.5,2.0,2.5,2.5,2.0,1.5,1.0,0.5);
    for (int i = 0; i < 100; i++) kernel[i] = kernelVals[i];
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
void main(void) {
    vec2 m = (iMouse.z > 0.5 ? iMouse.xy : 0.5 * iResolution) / iResolution;
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    float d = length((tc - m) * ar);
    float focus = 1.0 - smoothstep(0.15, 0.45, d);
```

```

    vec4 base = texture(samp, tc);
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec4 fx = xor_RGB(tcolor, tcolor * time_t);
    color = mix(base, fx, focus);
}

```

Shader #1085 - xorMouse

File: xorMouse.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position for interactive control. Calculate drag vector and strength Calculate color shift parameters Animate return when released

Technique: Multi-tap convolution blur (4 texture samples) - Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
vec3 hueShift(vec3 color, float hue) {
    vec3 k = vec3(0.57735, 0.57735, 0.57735);
    float cosAngle = cos(hue);
    return color * cosAngle + cross(k, color) * sin(hue) + k * dot(k, color) * (1.0 - cosAngle);
}
void main() {
    vec4 baseColor = texture(samp, tc);
    vec2 mouseNorm = iMouse.xy / iResolution.xy;
    vec2 clickNorm = iMouse.zw / iResolution.xy;
    vec2 dragVec = mouseNorm - clickNorm;
    float dragStrength = smoothstep(0.0, 0.5, length(dragVec));
    vec2 dragDir = normalize(dragVec + vec2(0.0001));
    float hueAngle = atan(dragDir.y, dragDir.x);
    float shiftAmount = dragStrength * 2.0;
    float returnSpeed = 2.0;
    float timeDecay = exp(-time_f * returnSpeed * (1.0 - step(0.5, iMouse.z)));
    shiftAmount *= mix(timeDecay, 1.0, step(0.5, iMouse.z));
    vec3 shiftedColor = hueShift(baseColor.rgb, hueAngle * shiftAmount);
    vec2 redOffset = dragDir * shiftAmount * 0.02;
    vec2 greenOffset = dragDir * shiftAmount * 0.01;
    vec3 finalColor = vec3(
        texture(samp, tc - redOffset).r,
        texture(samp, tc - greenOffset).g,
        texture(samp, tc).b
    );
    finalColor = mix(finalColor, shiftedColor, dragStrength);
    color = vec4(finalColor, baseColor.a);
}

```


Shader #1092 - zoom in out mouse

File: zoom_in_out_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, mouse, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution.xy) : vec2(0.5);
    vec2 normPos = ((gl_FragCoord.xy - iMouse.xy) / iResolution.xy) * 2.0 - 1.0;
    float distScr = length(normPos);
    float phase = sin(distScr * 10.0 - time_f * 4.0);
    float ar = iResolution.x / iResolution.y;
    vec2 p = tc - m;
    p.x *= ar;
    float r = length(p);
    float a = atan(p.y, p.x) + 6.0 * r + time_f * 0.6;
    p = r * vec2(cos(a), sin(a));
    float zoom = 0.95 + 0.25 * sin(time_f * 0.9);
    p /= zoom;
    p.x /= ar;
    vec2 coord = m + p;
    coord += normPos * 0.025 * phase;
    color = texture(samp, fract(coord));
}
```

Distortion & Warp (113 shaders)

Shader #0 - 3high

File: 3high.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. A helper function to generate pseudo-random values based on a 2D input. Start with the original texture coordinates. Adjust roll speed as desired.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Per-channel UV offset for chromatic aberration - Driven by: time, mouse, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
float rand(vec2 co) {
    return fract(sin(dot(co, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void) {
    vec2 uv = tc;
    float rollSpeed = 0.2;
```

```

    uv.y = mod(uv.y + time_f * rollSpeed, 1.0);
    float bendAmplitude = 0.02;
    float bendFrequency = 10.0;
    uv.x += sin(uv.y * 3.1415 * bendFrequency + time_f * 2.0) * bendAmplitude;
    float jitterChance = rand(vec2(time_f, uv.y));
    if(jitterChance > 0.95) {
        float jitterAmount = (rand(vec2(time_f * 2.0, uv.y * 3.0)) - 0.5) * 0.1;
        uv.x += jitterAmount;
    }
    float chromaShift = 0.005;
    float r = texture(samp, uv + vec2(chromaShift, 0.0)).r;
    float g = texture(samp, uv).g;
    float b = texture(samp, uv - vec2(chromaShift, 0.0)).b;
    color = vec4(r, g, b, 1.0);
}

```

Shader #5 - 8huri

File: 8huri.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic.

Technique: Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec2 vortexDistortion(vec2 uv, float time) {
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
    float distance = length(offset);
    float angle = atan(offset.y, offset.x) + sin(time + distance * 10.0) * 0.5;
    return center + vec2(cos(angle), sin(angle)) * distance;
}
void main(void) {
    vec2 uv = tc;
    uv = vortexDistortion(uv, time_f);
    uv.x = mod(uv.x + sin(time_f) * 0.2, 1.0);
    uv.y = mod(uv.y + cos(time_f) * 0.2, 1.0);
    vec4 texColor = texture(samp, uv);
    texColor.r += sin(time_f * 10.0) * 0.1;
    texColor.g += cos(time_f * 15.0) * 0.1;
    texColor.b += sin(time_f * 20.0) * 0.1;
    float[16] rVals = float[16](0.0, 0.26, 0.5, 0.74, 1.0, 0.88, 0.62, 0.38, 0.14, 0.08, 0.22, 0.54,
    ↪ 0.78, 0.92, 0.96, 0.1);
    float[16] gVals = float[16](0.0, 0.13, 0.25, 0.37, 0.5, 0.44, 0.31, 0.19, 0.08, 0.13, 0.37, 0.44,
    ↪ 0.56, 0.69, 0.82, 0.96);
    float[16] bVals = float[16](0.0, 0.06, 0.13, 0.25, 0.38, 0.44, 0.56, 0.69, 0.82, 0.96, 0.74, 0.5,
    ↪ 0.25, 0.13, 0.06, 0.0);
    int idxR = int(floor(texColor.r * 15.0 + 0.5));
    int idxG = int(floor(texColor.g * 15.0 + 0.5));
    int idxB = int(floor(texColor.b * 15.0 + 0.5));
    vec3 quantizedColor = vec3(rVals[idxR], gVals[idxG], bVals[idxB]);
    color = vec4(quantizedColor, texColor.a);
}

```

Shader #20 - air

File: air.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float distortionStrength = 0.02;
    float distortionFrequency = 15.0;
    float distortion1 = sin(tc.y * distortionFrequency + time_f) * distortionStrength;
    float distortion2 = cos((tc.x + tc.y) * distortionFrequency + time_f) * distortionStrength;
    vec2 distortedTC1 = tc + vec2(distortion1, distortion2);
    vec2 distortedTC2 = tc + vec2(distortion2, -distortion1);
    vec4 texColor1 = texture(samp, distortedTC1);
    vec4 texColor2 = texture(samp, distortedTC2);
    color = mix(texColor1, texColor2, 0.5);
}
```

Shader #21 - air full

File: air_full.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float distortionStrength = 0.02;
    float distortionFrequency = 15.0;
    float distortion1 = sin(tc.y * distortionFrequency + time_f) * distortionStrength;
    float distortion2 = cos((tc.x + tc.y) * distortionFrequency + time_f) * distortionStrength;
    vec2 distortedTC1 = tc + vec2(distortion1, distortion2);
    vec2 distortedTC2 = tc + vec2(distortion2, -distortion1);
    vec4 texColor1 = texture(samp, distortedTC1);
    vec4 texColor2 = texture(samp, distortedTC2);
    color = texColor2;
}
```

Shader #34 - atan bowl

File: atan-bowl.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = tc - center;
    float maxRadius = length(vec2(0.5, 0.5));
    float radius = length(offset);
    float normalizedRadius = radius / maxRadius;
    float angle = atan(offset.y, offset.x);
    float distortion = 0.5;
    float distortedRadius = normalizedRadius + distortion * pow(normalizedRadius, 2.0);
    distortedRadius = clamp(distortedRadius, 0.0, 1.0);
    distortedRadius *= maxRadius;
    vec2 distortedCoords = center + distortedRadius * vec2(cos(angle), sin(angle));
    float modulatedTime = pingPong(time_f, 5.0);
    angle += modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle) * (distortedCoords.x - center.x) - sin(angle) * (distortedCoords.y - center
↵ .y) + center.x;
    rotatedTC.y = sin(angle) * (distortedCoords.x - center.x) + cos(angle) * (distortedCoords.y - center
↵ .y) + center.y;
    vec2 warpedCoords;
    warpedCoords.x = pingPong(rotatedTC.x + time_f * 0.1, 1.0);
    warpedCoords.y = pingPong(rotatedTC.y + time_f * 0.1, 1.0);
    color = texture(samp, warpedCoords);
}
```

Shader #54 - bend

File: bend.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to audio input (amplitude/volume) for interactive control. Current amplitude (0.0 - 1.0) Normalize texture coordinates to range [0.0, 1.0] Calculate a time-based parameter

Technique: 2D rotation matrix - Driven by: time, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float amp;
uniform float uamp;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
```

```

void main(void) {
    vec2 uv = tc;
    float t = mod(time_f, 10.0) * 0.1;
    float angle = sin(t * 3.14159265) * 0.5;
    vec2 center = vec2(0.5, 0.5);
    vec2 centeredUV = uv - center;
    float dist = length(centeredUV);
    float bend = sin(dist * 6.0 - t * 2.0 * 3.14159265) * 0.05 * amp * uamp;
    mat2 rotation = mat2(cos(angle), -sin(angle),
                        sin(angle), cos(angle));
    vec2 rotatedUV = rotation * centeredUV;
    vec2 finalUV = rotatedUV + center;
    float time_t = pingPong(time_f, 10.0);
    finalUV += sin(bend * rotatedUV * time_t) * amp * uamp;
    color = texture(samp, finalUV);
    color = vec4(sin(color.rgb * (time_t+2.0)), 1.0);
}

```

Shader #55 - bend2

File: bend2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    float t = mod(time_f, 10.0) * 0.1;
    float angle = sin(t * 3.14159265) * 0.5;
    vec2 center = vec2(0.5, 0.5);
    uv -= center;
    float dist = length(uv);
    float bend = sin(dist * 6.0 - t * 2.0 * 3.14159265) * 0.05;
    uv = mat2(cos(angle), -sin(angle), sin(angle), cos(angle)) * uv;
    uv += center;
    float time_t = pingPong(time_f, 10.0);
    uv += sin(time_t * bend) + tan(bend * uv * time_t) ;
    color = texture(samp, uv);
}

```

Shader #57 - bend dir

File: bend_dir.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void) {
    vec2 center = vec2(0.5);
    vec2 uv = tc - center;
    float r = length(uv);
    float t = time_f;
    float s = pingPong(t, 10.0) * 0.1;
    float bendR = 0.15 + 0.1*sin(t*0.5);
    float swirl = (0.35 + 0.25*sin(t*0.33)) * (1.0 - smoothstep(0.0, 0.707, r));
    float ang = atan(uv.y, uv.x) + swirl;
    float rb = r * (1.0 + bendR * sin(r*12.0 + t*1.7));
    vec2 n1 = vec2(cos(t*0.37), sin(t*0.37));
    vec2 n2 = vec2(cos(t*0.53+1.7), sin(t*0.53+1.7));
    float w1 = sin(dot(uv, n1)*18.0 + t*1.3);
    float w2 = sin(dot(uv, n2)*14.0 - t*1.1);
    vec2 dirBend = normalize(n1)*w1 + normalize(n2)*w2;
    vec2 uvb = vec2(cos(ang), sin(ang)) * rb;
    uvb += dirBend * (0.025 + 0.02*sin(t*0.21)) * (0.5 + 0.5*sin(r*10.0 + t));
    float rot = sin(t*3.14159265*0.2) * 0.6;
    mat2 R = mat2(cos(rot), -sin(rot), sin(rot), cos(rot));
    uvb = R * uvb;
    uv = uvb + center;
    uv -= sin(uv*6.28318 + t) * (0.01 + 0.01*s);
    color = texture(samp, uv);
}
```

Shader #58 - bend rev

File: bend_rev.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    float t = mod(time_f, 10.0) * 0.1;
    float angle = sin(t * 3.14159265) * -0.5;
```

```

    vec2 center = vec2(0.5, 0.5);
    uv -= center;
    float dist = length(uv);
    float bend = sin(dist * 6.0 + t * 2.0 * 3.14159265) * 0.05;
    uv = mat2(cos(angle), -sin(angle), sin(angle), cos(angle)) * uv;
    uv += center;
    float time_t = pingPong(time_f, 10.0);
    uv -= sin(bend * uv * time_t);
    color = texture(samp, uv);
}

```

Shader #59 - bend twist

File: bend_twist.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
void main(void) {
    vec2 uv = tc - 0.5;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float dist = length(uv);
    float twistStrength = (time_f * 10.0);
    float angle = amp + (dist * twistStrength);
    float s = sin(angle);
    float c = cos(angle);
    vec2 twistedUV = vec2(
        uv.x * c - uv.y * s,
        uv.x * s + uv.y * c
    );
    float bendStrength = (time_f * uamp * 0.5);
    twistedUV.x += sin(twistedUV.y * 10.0 + time_f) * bendStrength;
    twistedUV.x /= aspect;
    vec2 finalUV = twistedUV + 0.5;
    color = texture(samp, finalUV);
}

```

Shader #77 - bowl

File: bowl.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look.

Technique: Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float len) {
    float m = mod(x, len * 2.0);
    return abs(m - len);
}
float rand(float n) {
    return fract(sin(n) * 43758.5453123);
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 off = tc - center;
    float maxR = length(vec2(0.5));
    float r = length(off);
    float nr = r / maxR;
    float ang = atan(off.y, off.x);
    float distortion = 0.5;
    float dr = nr + distortion * (nr * nr);
    dr = clamp(dr, 0.0, 1.0) * maxR;
    vec2 d = center + dr * vec2(cos(ang), sin(ang));
    float dir = sign(rand(floor(time_f * 0.2)) - 0.5);
    float phase = rand(floor(time_f * 0.1)) * 6.2831853;
    float rotateSpeed = 1.0;
    float warpSpeed = 0.1;
    ang += dir * (phase + time_f * rotateSpeed);
    vec2 dp = d - center;
    float ca = cos(ang), sa = sin(ang);
    vec2 rot = vec2(ca * dp.x - sa * dp.y, sa * dp.x + ca * dp.y) + center;
    float t = time_f * warpSpeed;
    vec2 w = vec2(pingPong(rot.x + t, 1.0), pingPong(rot.y + t, 1.0));
    color = texture(samp, w);
}
```

Shader #83 - brokenglass

File: brokenglass.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Simple hash for pseudo-random crack generation Voronoi for crack-like cells Crack mask that spreads with time

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
```



```

uniform vec2 iResolution;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453);
}
float voronoi(vec2 uv) {
    vec2 g = floor(uv);
    vec2 f = fract(uv);
    float res = 8.0;
    for (int y=-1; y<=1; y++) {
        for (int x=-1; x<=1; x++) {
            vec2 lattice = vec2(x,y);
            vec2 p = hash(g+lattice) * vec2(0.5+0.5*sin(time_f*0.2));
            vec2 diff = lattice + p - f;
            float d = length(diff);
            res = min(res, d);
        }
    }
    return res;
}
void main(void) {
    vec2 uv = tc * iResolution.xy / min(iResolution.x, iResolution.y);
    float cracks = smoothstep(0.05, 0.15, voronoi(uv * (2.0 + 0.5*sin(time_f*0.1))));
    vec2 offset = vec2(cos(uv.y*20.0 + time_f*0.5), sin(uv.x*20.0 - time_f*0.5))
        * (1.0 - cracks) * 0.02;
    vec4 texCol = texture(samp, tc + offset);
    vec3 crackColor = mix(vec3(0.0), texCol.rgb, cracks);
    color = vec4(crackColor, 1.0);
}

```

Shader #136 - comb

File: comb.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    float maxTime = pingPong(time_f, 10.0);
    float time = pingPong(time_f, maxTime);
    float gForceIntensity = 0.1;
    float waveFrequency = 10.0;
    float waveAmplitude = 0.03;
    float gravityPull = gForceIntensity * time * uv.y;
    float waveDistortion = waveAmplitude * sin(waveFrequency * uv.x + time * 2.0);
    vec2 centeredUV = uv - vec2(0.5, 0.5);
    float angle = time + length(centeredUV) * 4.0;
    float spiralAmount = 0.5 * (1.0 - uv.y);
    mat2 rotation = mat2(cos(angle), -sin(angle), sin(angle), cos(angle));
}

```

```

    vec2 spiraledUV = rotation * centeredUV * (1.0 + spiralAmount);
    vec2 distortedUV = spiraledUV + vec2(0.5, 0.5);
    distortedUV.y += gravityPull + waveDistortion;
    distortedUV = clamp(distortedUV, vec2(0.0), vec2(1.0));
    color = texture(samp, distortedUV);
}

```

Shader #157 - cosine swirl

File: cosine_swirl.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    float hue = mod(t, 1.0) * 6.0;
    float c = 1.0;
    float x = 1.0 - abs(mod(hue, 2.0) - 1.0);
    vec3 rgb = (hue < 1.0 ? vec3(c, x, 0.0) :
                (hue < 2.0 ? vec3(x, c, 0.0) :
                (hue < 3.0 ? vec3(0.0, c, x) :
                (hue < 4.0 ? vec3(0.0, x, c) :
                (hue < 5.0 ? vec3(x, 0.0, c) :
                vec3(c, 0.0, x);
    return rgb;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float frequency = 1.0;
    float amplitude = 3.0;
    float spiralTime = sin(time_f * frequency) * amplitude;
    float scale = 1.0;
    float gradient_pos = mod(dist * scale + (angle + spiralTime) / (2.0 * 3.14159), 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec4 ctx = texture(samp, tc);
    float time_t = mod(time_f, 10.0) + 2.0;
    color = mix(cos(ctx * time_t), vec4(color_gradient, 1.0), 0.5);
    color.a = 1.0;
}

```

Shader #158 - cosine swirl two

File: cosine_swirl_two.glsl

Renders a fractal pattern using iterative computation (many iterations). Responds to mouse position for interactive control.

Technique: Iterative fractal computation with complex-plane mapping - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
vec3 rainbow(float t){
    float hue = mod(t, 1.0) * 6.0;
    float c = 1.0;
    float x = 1.0 - abs(mod(hue, 2.0) - 1.0);
    vec3 rgb = (hue < 1.0) ? vec3(c, x, 0.0) :
                (hue < 2.0) ? vec3(x, c, 0.0) :
                (hue < 3.0) ? vec3(0.0, c, x) :
                (hue < 4.0) ? vec3(0.0, x, c) :
                (hue < 5.0) ? vec3(x, 0.0, c) :
                vec3(c, 0.0, x);
    return rgb;
}
void main(void){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p = (tc - m) * ar;
    float base = 1.68;
    float period = log(base);
    float t = time_f * 0.5;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x) + t * 0.35;
    float k = fract((log(r) - t) / period);
    float rw = exp(k * period);
    vec2 z = vec2(cos(a), sin(a)) * rw;
    vec2 uv = fract(z / ar + m);
    vec2 dir = (uv - m);
    dir.x *= aspect;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float frequency = 1.0;
    float amplitude = 3.0;
    float spiralTime = sin(time_f * frequency) * amplitude;
    float scale = 1.0;
    float gradient_pos = mod(dist * scale + (angle + spiralTime) / (2.0 * 3.14159), 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec4 ctx = texture(samp, uv);
    float time_t = mod(time_f, 10.0) + 2.0;
    color = mix(cos(ctx * time_t), vec4(color_gradient, 1.0), 0.5);
    color.a = 1.0;
}
```

Shader #167 - crystalbend

File: crystalbend.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Color quantization via floor

rounding - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp
blending - Driven by: time, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp;
uniform float uamp;
float PI = 3.141592653589793;
float mirror1(float x, float m){
    float p = mod(x, m*2.0);
    return p > m ? (2.0*m - p) : p;
}
vec2 kaleido(vec2 p, float seg){
    vec2 c = vec2(0.5);
    vec2 d = p - c;
    float a = atan(d.y, d.x);
    float r = length(d);
    float m = PI / max(2.0, seg);
    a = mirror1(a, m);
    return c + vec2(cos(a), sin(a)) * max(r, 1e-6);
}
void main(){
    vec2 uv = tc;
    float drive = clamp(amp + uamp, 0.0, 2.0);
    float seg = floor(mix(5.0, 12.0, clamp(drive*0.5, 0.0, 1.0)));
    vec2 k = kaleido(uv, seg);
    float r = length(k - 0.5);
    vec2 dir = normalize(k - 0.5 + vec2(1e-6, 0.0));
    float bend = (0.06 + 0.14 * uamp) / (r + 0.04);
    vec2 wob = 0.012 * (0.3 + amp) * vec2(sin(32.0*r - 1.7*time_f), cos(28.0*r - 1.3*time_f));
    vec2 base = k + dir * bend + wob;
    float disp = (0.001 + 0.004 * amp) / (r + 0.04);
    vec2 tR = base + dir * disp;
    vec2 tG = base;
    vec2 tB = base - dir * disp;
    vec3 warped;
    warped.r = texture(samp, tR).r;
    warped.g = texture(samp, tG).g;
    warped.b = texture(samp, tB).b;
    float wedge = PI / max(2.0, seg);
    float a = atan((k - 0.5).y, (k - 0.5).x);
    float edge = abs(mod(a + wedge, 2.0*wedge) - wedge) / wedge;
    float edgeGlow = pow(1.0 - edge, 6.0) * mix(0.10, 0.35, clamp(amp, 0.0, 1.0));
    warped += edgeGlow;
    vec3 orig = texture(samp, uv).rgb;
    float clarityBias = 0.45;
    float effectAmt = clamp(clarityBias + 0.4 * clamp(drive*0.5, 0.0, 1.0), 0.0, 1.0);
    float spatial = smoothstep(0.0, 0.85, r);
    float mixAmt = clamp(mix(effectAmt*0.7, effectAmt, spatial), 0.0, 1.0);
    vec3 outc = mix(orig, warped, mixAmt);
    color = vec4(outc, 1.0);
}
```

Shader #172 - cwrap

File: cwrap.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float h1(float n){return fract(sin(n)*43758.5453123);}
vec2 h2(float n){return fract(sin(vec2(n, n+1.0))*vec2(43758.5453,22578.1459));}
float tri(float x){float f=fract(x); return 1.0-abs(f*2.0-1.0);}
void main(void) {
    vec2 uv = tc;
    float rate = 0.8;
    float t = time_f*rate;
    float t0 = floor(t);
    float a = fract(t);
    vec2 p0 = vec2(0.1) + h2(t0)*0.8;
    vec2 p1 = vec2(0.1) + h2(t0+1.0)*0.8;
    float w = a*a*(3.0-2.0*a);
    vec2 m_auto = mix(p0, p1, w);
    vec2 m = (iMouse.z>0.5 || iMouse.w>0.5) ? (iMouse.xy/iResolution) : m_auto;
    vec2 d = uv - m;
    float dist = length(d);
    float a1 = clamp(amp, 0.0, 1.0);
    float ua = clamp(uamp, 0.0, 1.0);
    float r0 = mix(0.06, 0.35, a1) * 1.5;
    float pr = tri(time_f*0.6);
    float r = r0 * mix(0.75, 1.35, pr*(0.3+0.7*ua));
    float s = smoothstep(r, 0.0, dist);
    float k = 0.6 + 0.4 * ua;
    float swirl = (0.8 * ua + 0.2 * a1) * s * (r - dist) * 8.0;
    float ang = atan(d.y, d.x) + swirl;
    vec2 drot = vec2(cos(ang), sin(ang)) * dist;
    float lens = 1.0 - k * s * (1.0 - dist / r);
    vec2 warped = m + drot * lens;
    warped = warped * fract(time_f);
    float wob = sin(time_f * 3.0 + dist * 20.0) * 0.005 * ua * s;
    vec2 n = normalize(drot + vec2(1e-6));
    warped += n * wob;
    vec4 warpedCol = texture(samp, clamp(warped, vec2(0.0), vec2(1.0)));
    color = warpedCol;
}
```

Shader #179 - d1

File: d1.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.
Create distortion parameters Generate horizontal and vertical distortions Apply distortion to

texture coordinates

Technique: Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float waveSpeed = 1.1;
    float waveIntensity = 0.26;
    float waveDensity = 8.0;
    float horizontalDistortion = sin(uv.y * waveDensity + time_f * waveSpeed) * waveIntensity;
    float verticalDistortion = sin(uv.x * (waveDensity * 0.8) + time_f * waveSpeed * 1.2) *
        ↪ waveIntensity;
    uv.x += horizontalDistortion;
    uv.y += verticalDistortion;
    vec2 center = vec2(0.5);
    float distanceFromCenter = length(uv - center);
    float radialDistortion = sin(distanceFromCenter * 15.0 - time_f * 3.0) * 0.015;
    uv += (uv - center) * radialDistortion;
    color = texture(samp, uv);
}
```

Shader #180 - damaged vcr

File: damaged_vcr.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Random functions for chaos Fixed name collision Channel separation madness

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Per-channel UV offset for chromatic aberration - Scanline overlay using modular y-coordinate - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) { return fract(sin(dot(co.xy, vec2(12.9898,78.233))) * 43758.5453); }
float rand1(float x) { return fract(sin(x) * 43758.5453); }
vec3 brutalColorDistortion(vec3 col, vec2 uv, float t) {
    col.r = texture(samp, uv + vec2(sin(t*0.7)*0.03, cos(t*0.3)*0.02)).r;
    col.g = texture(samp, uv + vec2(sin(t*0.5)*0.02, cos(t*0.4)*0.03)).g;
    col.b = texture(samp, uv + vec2(sin(t*0.9)*0.04, cos(t*0.6)*0.01)).b;
    if (mod(t, 3.0) > 2.0) col = col.bgr;
    if (mod(t*0.7, 2.0) > 1.3) col = col.grb;
    return col;
}
void main(void) {
    vec2 uv = tc;
    float t = time_f * 3.0;
    uv.x += sin(uv.y * 50.0 + t * 10.0) * 0.1 * rand1(t);
    uv.y += cos(uv.x * 40.0 + t * 8.0) * 0.08 * rand1(t+0.3);
```

```

uv.x += sin(t * 5.0 + uv.y * 50.0) * 0.05;
uv.y += cos(t * 3.0 + uv.x * 60.0) * 0.03;
uv += vec2(rand1(t) - 0.5, rand1(t+0.5) - 0.5) * 0.1;
vec2 crtUV = uv * 2.0 - 1.0;
crtUV *= 1.0 + pow(length(crtUV), 3.0) * 0.5;
uv = crtUV * 0.5 + 0.5;
float split = step(0.5 + sin(t)*0.2, uv.x);
uv.y += split * (sin(t*2.0) * 0.1 * rand1(t));
float scanLine = fract(t * 0.3);
if(abs(uv.y - scanLine) < 0.02) {
    uv.x += (rand1(uv.y + t) - 0.5) * 0.3;
    uv.y += sin(uv.x * 100.0) * 0.1;
}
vec3 col = brutalColorDistortion(texture(samp, uv).rgb, uv, t);
float damageZone = step(0.9, rand(vec2(floor(uv.y * 20.0 + t * 0.5))));
col = mix(col, vec3(0.0), damageZone * 0.7);
float staticFlash = step(0.997, rand1(t * 0.1));
col += vec3(staticFlash * 2.0);
col *= 0.7 + 0.3 * sin(uv.x * 300.0 + t * 10.0);
vec3 noise = vec3(rand(uv * t), rand(uv * t + 0.3), rand(uv * t + 0.7)) * 0.8;
col += noise * smoothstep(0.3, 0.7, rand1(t * 0.3));
float crease = sin(uv.y * 30.0 + t * 5.0) * 0.1;
col *= 1.0 - smoothstep(0.3, 0.7, abs(crease));
col.r += sin(t * 5.0) * 0.3 + rand1(uv.x) * 0.2;
float osd = step(0.98, rand1(uv.x * 10.0 + t * 0.1));
col = mix(col, vec3(0.0, 1.0, 0.0), osd * 0.3);
color = vec4(col, 1.0);
color *= 0.8 + 0.2 * sin(uv.y * 800.0 + t * 10.0);
color *= 0.7 + 0.3 * rand1(t * 0.1);
color *= smoothstep(0.0, 0.2, uv.y) * smoothstep(1.0, 0.8, uv.y);
color *= smoothstep(0.0, 0.1, uv.x) * smoothstep(1.0, 0.9, uv.x);
}

```

Shader #194 - data

File: data.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. texture coordinates passed from vertex shader resolution (width, height) mouse data (not used in this example)

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, mouse, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
void main(void)
{
    vec2 uv = tc;
    float blockSize = 0.02;
    vec2 blockIndex = floor(uv / blockSize);
    float blockHash = hash(blockIndex + floor(time_f));
}

```

```

    if (blockHash > 0.9) {
        float xOffset = 0.1 * sin(time_f * 10.0);
        uv.x += xOffset;
    }
    float waveFreq = 20.0;
    float waveAmp = 0.01;
    uv.x += waveAmp * sin((uv.y + time_f) * waveFreq);
    color = texture(samp, uv);
}

```

Shader #202 - deeps1

File: deeps1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Create fractal layers Create multiple displacement layers Combine texture samples with color modulation

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
const float PI = 3.14159265359;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec2 polarDistort(vec2 uv) {
    vec2 center = uv - 0.5;
    float angle = atan(center.y, center.x);
    float radius = length(center) * 2.0;
    float fractal = sin(angle * 5.0 + time_f) * 0.2;
    fractal += sin(angle * 10.0 + time_f * 2.0) * 0.1;
    fractal += sin(angle * 20.0 + time_f * 0.5) * 0.05;
    radius += fractal * 0.3;
    angle += sin(time_f + radius * 5.0) * 0.5;
    vec2 distorted = vec2(cos(angle), sin(angle)) * radius;
    return distorted + 0.5;
}
vec4 fractalColor(vec2 uv) {
    vec2 uv1 = uv + vec2(sin(time_f * 0.7 + uv.y * 5.0), cos(time_f * 0.6 + uv.x * 5.0)) * 0.1;
    vec2 uv2 = uv + vec2(cos(time_f * 0.5 + uv.y * 10.0), sin(time_f * 0.4 + uv.x * 10.0)) * 0.05;
    vec2 uv3 = uv + vec2(sin(time_f * 0.3 + uv.y * 20.0), cos(time_f * 0.2 + uv.x * 20.0)) * 0.025;
    vec4 col1 = texture(samp, uv1);
    vec4 col2 = texture(samp, uv2);
    vec4 col3 = texture(samp, uv3);
    return vec4(col1.r, col2.g, col3.b, 1.0) * (1.0 + sin(time_f * 2.0) * 0.3);
}
void main(void) {
    vec2 uv = tc;
    uv = polarDistort(uv);
    uv.x += sin(uv.y * 10.0 + time_f * 2.0) * 0.03;
    uv.y += cos(uv.x * 8.0 + time_f * 1.5) * 0.03;
    float fractalScale = 5.0;
    uv.x += sin(uv.y * fractalScale + time_f) * 0.1;
}

```



```

    uv.y += cos(uv.x * fractalScale + time_f) * 0.1;
    float layer1 = pingPong(uv.x + time_f * 0.1, 1.0);
    float layer2 = pingPong(uv.y + time_f * 0.15, 1.0);
    uv = mix(uv, vec2(layer1, layer2), 0.3);
    vec4 finalColor = fractalColor(uv);
    vec4 originalColor = texture(samp, tc);
    vec2 centerVec = tc - 0.5;
    float radius = length(centerVec) * 2.0;
    float mixFactor = smoothstep(0.3, 0.7, radius);
    finalColor = mix(finalColor, originalColor, mixFactor * 0.5);
    color = finalColor;
}

```

Shader #203 - deeps3

File: deeps3.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float angle = atan(tc.y - 0.5, tc.x - 0.5);
    float modulatedTime = pingPong(time_f, 5.0);
    angle += modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle) * (tc.x - 0.5) - sin(angle) * (tc.y - 0.5) + 0.5;
    rotatedTC.y = sin(angle) * (tc.x - 0.5) + cos(angle) * (tc.y - 0.5) + 0.5;
    float warpFactor = sin(time_f) * 0.5 + 0.5;
    vec2 warpedCoords;
    warpedCoords.x = pingPong(rotatedTC.x + time_f * 0.1, 1.0);
    warpedCoords.y = pingPong(rotatedTC.y + time_f * 0.1, 1.0);
    vec2 warpedCoords2;
    warpedCoords2.x = pingPong(rotatedTC.x + time_f * 0.3, 1.0);
    warpedCoords2.y = pingPong(rotatedTC.y + time_f * 0.3, 1.0);
    vec2 warpedCoords3;
    warpedCoords3.x = pingPong(rotatedTC.x + time_f * 0.6, 1.0);
    warpedCoords3.y = pingPong(rotatedTC.y + time_f * 0.6, 1.0);
    color.r = texture(samp, warpedCoords).r;
    color.g = texture(samp, warpedCoords2).g;
    color.b = texture(samp, warpedCoords3).b;
    color.a = 1.0;
}

```

Shader #204 - deepseek3

File: deepseek3.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections.
Horizontal wave parameters Vertical wave parameters Diagonal wave parameters

Technique: Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation -
Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec3 rainbow(float t) {
    return 0.5 + 0.5 * cos(6.28318 * (t + vec3(0.0, 0.333, 0.666)));
}

vec4 waveEffect(vec2 uv, vec4 original) {
    float hSpeed = 1.5;
    float hFreq = 2.0;
    float hTime = pingPong(time_f * hSpeed, 2.0);
    float vSpeed = 1.2;
    float vFreq = 1.5;
    float vTime = pingPong(time_f * vSpeed, 2.0);
    float dSpeed = 2.0;
    float dFreq = 3.0;
    float dTime = pingPong(time_f * dSpeed, 2.0);
    float wave1 = sin((uv.x * hFreq + hTime) * 6.28318);
    float wave2 = sin((uv.y * vFreq + vTime) * 6.28318);
    float wave3 = sin((uv.x + uv.y) * dFreq + dTime * 3.14159);
    float combinedWave = (wave1 + wave2 + wave3) / 3.0;
    vec3 waveColor = rainbow(combinedWave * 0.5 + 0.5);
    float waveAlpha = smoothstep(0.3, 0.7, abs(combinedWave));
    float travelingWave =
        sin(uv.x * 20.0 + time_f * 5.0) *
        sin(uv.y * 20.0 - time_f * 3.0) *
        sin(time_f * 2.0);
    vec4 finalColor = original;
    finalColor.rgb = mix(
        finalColor.rgb,
        waveColor * (1.0 + travelingWave * 0.5),
        waveAlpha * 0.7
    );
    float edgeGlow = smoothstep(0.8, 0.0, length(uv - 0.5));
    finalColor.rgb += waveColor * edgeGlow * 0.3;
    return finalColor;
}

void main(void) {
    vec2 uv = tc;
    vec2 offset = vec2(
        sin(time_f + uv.y * 5.0) * 0.02,
        cos(time_f * 0.8 + uv.x * 4.0) * 0.02
    );
    vec4 original = texture(samp, uv + offset);
    color = waveEffect(uv, original);
    color.r = texture(samp, uv + offset * 0.3).r;
    color.b = texture(samp, uv - offset * 0.3).b;
    color = sin(color * pingPong(time_f, 10.0));
    color.a = original.a;
}
```

Shader #205 - deepseek4

File: deepseek4.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Wave timing parameters (6 second cycles) Horizontal wave movement Vertical wave movement

Technique: Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

float ramp(float t, float cycle) {
    float phase = mod(t, cycle * 2.0);
    return smoothstep(0.0, 1.0, phase/cycle) -
        smoothstep(1.0, 2.0, phase/cycle);
}

vec3 rainbow(float t) {
    return 0.5 + 0.5 * cos(6.28318 * (t + vec3(0.0, 0.333, 0.666)));
}

void main() {
    vec2 uv = tc;
    float hCycle = 6.0;
    float vCycle = 5.5;
    float dCycle = 6.5;
    float hPhase = time_f * 0.5;
    float hWave = ramp(hPhase, hCycle) * 2.0 - 1.0;
    float vPhase = time_f * 0.45;
    float vWave = ramp(vPhase, vCycle) * 2.0 - 1.0;
    float dPhase = time_f * 0.6;
    float dWave = ramp(dPhase, dCycle) * 2.0 - 1.0;
    float waveX = sin(uv.x * 8.0 + hWave * 20.0);
    float waveY = sin(uv.y * 6.0 + vWave * 15.0);
    float waveD = sin((uv.x + uv.y) * 10.0 + dWave * 25.0);
    float combined = (waveX + waveY + waveD) / 3.0;
    vec3 waveColor = rainbow(combined * 0.5 + 0.5 + time_f * 0.05);
    vec2 distort = vec2(
        combined * 0.02 * hWave,
        combined * 0.02 * vWave
    );
    vec4 tex = texture(samp, uv + distort);
    float colorBand = smoothstep(0.3, 0.7,
        sin(uv.x * 3.0 - hPhase * 0.5) *
        sin(uv.y * 2.0 + vPhase * 0.3) *
        sin((uv.x - uv.y) * 4.0 + dPhase * 0.4)
    );
    color = mix(tex, vec4(waveColor, 1.0), colorBand * 0.6);
    vec2 flowDir = normalize(vec2(hWave, vWave));
    float flowMask = dot(uv - 0.5, flowDir);
    color.rgb += waveColor * smoothstep(-0.5, 0.5, flowMask) * 0.2;
    color.r = texture(samp, uv + distort * 0.3).r;
    color.b = texture(samp, uv - distort * 0.3).b;
    color = sin(color * pingPong(time_f, 25.0));
}
```

```
    color.a = tex.a;
}
```

Shader #206 - deform

File: deform.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = tc;
    float warpAmount = sin(time_f);
    vec2 warp = vec2(
        sin(normCoord.y * 10.0 + time_f) * warpAmount,
        cos(normCoord.x * 10.0 + time_f) * warpAmount
    );
    vec2 warpedCoord = normCoord + warp;
    color = texture(samp, warpedCoord);
}
```

Shader #207 - deform cos

File: deform_cos.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = tc;
    float warpAmount = cos(time_f);
    vec2 warp = vec2(
        sin(normCoord.y * 10.0 + time_f) * warpAmount,
        cos(normCoord.x * 10.0 + time_f) * warpAmount
    );
    vec2 warpedCoord = normCoord + warp;
    color = texture(samp, warpedCoord);
}
```

Shader #208 - deform tan

File: deform_tan.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = tc;
    float warpAmount = tan(time_f);
    vec2 warp = vec2(
        sin(normCoord.y * 10.0 + time_f) * warpAmount,
        cos(normCoord.x * 10.0 + time_f) * warpAmount
    );
    vec2 warpedCoord = normCoord + warp;
    color = texture(samp, warpedCoord);
}
```

Shader #219 - Dispersion

File: Dispersion.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Applies matrix transformations. Slowed down slightly Rotation speeds derived from Shader 1 Perspective projection

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.14159265359;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        n = 1.0 - abs(n * 2.0 - 1.0);
```

```

        v += a * n;
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.2);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.2;
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p2 = (tc - m) * ar;
    float ax = 0.25 * sin(time_f * 0.5);
    float ay = 0.25 * cos(time_f * 0.4);
    float az = time_f * 0.2;
    vec3 p3 = vec3(p2, 1.0);
    mat3 R = rotZ(az) * rotY(ay) * rotX(ax);
    vec3 r = R * p3;
    float k = 0.6;
    float zf = 1.0 / (1.0 + r.z * k);
    vec2 projectedUV = r.xy * zf;
    float elect = fbm(projectedUV * 4.0 - time_f * 1.5);
    float dist = length(projectedUV);
    float ripple = sin(dist * 20.0 - time_f * 3.0);
    float scale = 1.0 + 0.15 * ripple * elect;
    projectedUV *= scale;
    vec2 tiledUV = 1.0 - abs(1.0 - 2.0 * (projectedUV + 0.5));
    tiledUV = tiledUV - floor(tiledUV);
    float dispersion = 0.02 * (0.5 + elect);
    vec2 dispOffset = normalize(projectedUV) * dispersion;
    vec2 uvR = tiledUV - dispOffset;
    vec2 uvG = tiledUV;
    vec2 uvB = tiledUV + dispOffset;
    float rChannel = texture(samp, clamp(uvR, 0.0, 1.0)).r;
    float gChannel = texture(samp, clamp(uvG, 0.0, 1.0)).g;
    float bChannel = texture(samp, clamp(uvB, 0.0, 1.0)).b;
    vec3 texColor = vec3(rChannel, gChannel, bChannel);
    vec3 neon = neonPalette(time_f + dist);
    float glowMask = smoothstep(0.4, 0.9, elect);
    vec3 finalColor = texColor + (neon * glowMask * 0.8);
    finalColor = finalColor / (1.0 + finalColor * 0.3);
    color = vec4(finalColor, 1.0);
}

```

Shader #220 - DispersionRotate

File: DispersionRotate.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Applies matrix transformations. Slowed down slightly Rotation speeds derived from Shader 1, including rotation around the Y-axis Add rotation around Y-axis and offset slightly for movement

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.14159265359;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.2);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.2;
}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p2 = (tc - m) * ar;
    float ax = 0.25 * sin(time_f * 0.5);
    float ay = 0.25 * cos(time_f * 0.4) + time_f * 0.1;
```

```

float az = time_f * 0.2;
vec3 p3 = vec3(p2, 1.0);
mat3 R = rotZ(az) * rotY(ay) * rotX(ax);
vec3 r = R * p3;
float k = 0.6;
float zf = 1.0 / (1.0 + r.z * k);
vec2 projectedUV = r.xy * zf;
float elect = fbm(projectedUV * 4.0 - time_f * 1.5);
float dist = length(projectedUV);
float ripple = sin(dist * 20.0 - time_f * 3.0);
float scale = 1.0 + 0.15 * ripple * elect;
projectedUV *= scale;
vec2 tiledUV = 1.0 - abs(1.0 - 2.0 * (projectedUV + 0.5));
tiledUV = tiledUV - floor(tiledUV);
float dispersion = 0.02 * (0.5 + elect);
vec2 dispOffset = normalize(projectedUV) * dispersion;
vec2 uvR = tiledUV - dispOffset;
vec2 uvG = tiledUV;
vec2 uvB = tiledUV + dispOffset;
float rChannel = texture(samp, clamp(uvR, 0.0, 1.0)).r;
float gChannel = texture(samp, clamp(uvG, 0.0, 1.0)).g;
float bChannel = texture(samp, clamp(uvB, 0.0, 1.0)).b;
vec3 texColor = vec3(rChannel, gChannel, bChannel);
vec3 neon = neonPalette(time_f + dist);
float glowMask = smoothstep(0.4, 0.9, elect);
vec3 finalColor = texColor + (neon * glowMask * 0.8);
finalColor = finalColor / (1.0 + finalColor * 0.3);
color = vec4(finalColor, 1.0);
}

```

Shader #221 - DispersionX

File: DispersionX.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Indexed color palette look-up table - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.14159265359;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}

```



```

float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.2);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.2;
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p2 = (tc - m) * ar;
    float ax = time_f * 0.5;
    mat3 R = rotX(ax);
    vec3 p3 = vec3(p2, 1.0);
    vec3 r = R * p3;
    float k = 0.6;
    float zf = 1.0 / (1.0 + r.z * k);
    vec2 projectedUV = r.xy * zf;
    float elect = fbm(projectedUV * 4.0 - time_f * 1.5);
    float dist = length(projectedUV);
    float ripple = sin(dist * 20.0 - time_f * 3.0);
    float scale = 1.0 + 0.15 * ripple * elect;
    projectedUV *= scale;
    vec2 tiledUV = 1.0 - abs(1.0 - 2.0 * (projectedUV + 0.5));
    tiledUV = tiledUV - floor(tiledUV);
    float dispersion = 0.02 * (0.5 + elect);
    vec2 dispOffset = normalize(projectedUV) * dispersion;
    vec2 uvR = tiledUV - dispOffset;
    vec2 uvG = tiledUV;
    vec2 uvB = tiledUV + dispOffset;
    float rChannel = texture(samp, clamp(uvR, 0.0, 1.0)).r;
    float gChannel = texture(samp, clamp(uvG, 0.0, 1.0)).g;
    float bChannel = texture(samp, clamp(uvB, 0.0, 1.0)).b;
    vec3 texColor = vec3(rChannel, gChannel, bChannel);
    vec3 neon = neonPalette(time_f + dist);
    float glowMask = smoothstep(0.4, 0.9, elect);
    vec3 finalColor = texColor + (neon * glowMask * 0.8);
    finalColor = finalColor / (1.0 + finalColor * 0.3);
    color = vec4(finalColor, 1.0);
}

```

Shader #222 - DispersionX Audio

File: DispersionX_Audio.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Applies matrix transformations. peak absolute sample value in current audio buffer RMS energy of current audio buffer exponentially smoothed amplitude for gradual transitions

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.14159265359;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.2);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.2;
}
```

```

}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p2 = (tc - m) * ar;
    float ax = time_f * 0.5 + (amp_smooth * 1.5);
    mat3 R = rotX(ax);
    vec3 p3 = vec3(p2, 1.0);
    vec3 r = R * p3;
    float k = 0.6;
    float zf = 1.0 / (1.0 + r.z * k);
    vec2 projectedUV = r.xy * zf;
    float elect = fbm(projectedUV * (4.0 + amp_mid * 2.0) - time_f * 1.5 - (amp_rms * 4.0));
    float dist = length(projectedUV);
    float ripple = sin(dist * 20.0 - time_f * 3.0 - (amp_low * 15.0));
    float scale = 1.0 + 0.15 * ripple * elect - (amp_low * 0.25);
    projectedUV *= scale;
    vec2 tiledUV = 1.0 - abs(1.0 - 2.0 * (projectedUV + 0.5));
    tiledUV = tiledUV - floor(tiledUV);
    float dispersion = 0.02 * (0.5 + elect) + (amp_high * 0.08);
    vec2 dispOffset = normalize(projectedUV) * dispersion;
    vec2 uvR = tiledUV - dispOffset;
    vec2 uvG = tiledUV;
    vec2 uvB = tiledUV + dispOffset;
    float rChannel = texture(samp, clamp(uvR, 0.0, 1.0)).r;
    float gChannel = texture(samp, clamp(uvG, 0.0, 1.0)).g;
    float bChannel = texture(samp, clamp(uvB, 0.0, 1.0)).b;
    vec3 texColor = vec3(rChannel, gChannel, bChannel);
    vec3 neon = neonPalette(time_f + dist + (amp_smooth * 4.0));
    float glowMask = smoothstep(0.4 - (amp_mid * 0.2), 0.9, elect);
    vec3 finalColor = texColor + (neon * glowMask * (0.8 + amp_peak * 2.5));
    finalColor = finalColor / (1.0 + finalColor * 0.3);
    color = vec4(finalColor, 1.0);
}

```

Shader #223 - DispersionY

File: DispersionY.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Indexed color palette look-up table - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.14159265359;
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {

```

```

    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
               mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}

float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.2);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.2;
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p2 = (tc - m) * ar;
    float ay = time_f * 0.5;
    mat3 R = rotY(ay);
    vec3 p3 = vec3(p2, 1.0);
    vec3 r = R * p3;
    float k = 0.6;
    float zf = 1.0 / (1.0 + r.z * k);
    vec2 projectedUV = r.xy * zf;
    float elect = fbm(projectedUV * 4.0 - time_f * 1.5);
    float dist = length(projectedUV);
    float ripple = sin(dist * 20.0 - time_f * 3.0);
    float scale = 1.0 + 0.15 * ripple * elect;
    projectedUV *= scale;
    vec2 tiledUV = 1.0 - abs(1.0 - 2.0 * (projectedUV + 0.5));
    tiledUV = tiledUV - floor(tiledUV);
    float dispersion = 0.02 * (0.5 + elect);
    vec2 dispOffset = normalize(projectedUV) * dispersion;
    vec2 uvR = tiledUV - dispOffset;
    vec2 uvG = tiledUV;
    vec2 uvB = tiledUV + dispOffset;
    float rChannel = texture(samp, clamp(uvR, 0.0, 1.0)).r;
    float gChannel = texture(samp, clamp(uvG, 0.0, 1.0)).g;
    float bChannel = texture(samp, clamp(uvB, 0.0, 1.0)).b;
    vec3 texColor = vec3(rChannel, gChannel, bChannel);
    vec3 neon = neonPalette(time_f + dist);
    float glowMask = smoothstep(0.4, 0.9, elect);
    vec3 finalColor = texColor + (neon * glowMask * 0.8);
}

```

```

    finalColor = finalColor / (1.0 + finalColor * 0.3);
    color = vec4(finalColor, 1.0);
}

```

Shader #224 - DispersionZ

File: DispersionZ.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Indexed color palette look-up table - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.14159265359;
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.2);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.2;
}
void main(void) {

```

```

float aspect = iResolution.x / iResolution.y;
vec2 ar = vec2(aspect, 1.0);
vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
vec2 p2 = (tc - m) * ar;
float az = time_f * 0.5;
mat3 R = rotZ(az);
vec3 p3 = vec3(p2, 1.0);
vec3 r = R * p3;
float k = 0.6;
float zf = 1.0 / (1.0 + r.z * k);
vec2 projectedUV = r.xy * zf;
float elect = fbm(projectedUV * 4.0 - time_f * 1.5);
float dist = length(projectedUV);
float ripple = sin(dist * 20.0 - time_f * 3.0);
float scale = 1.0 + 0.15 * ripple * elect;
projectedUV *= scale;
vec2 tiledUV = 1.0 - abs(1.0 - 2.0 * (projectedUV + 0.5));
tiledUV = tiledUV - floor(tiledUV);
float dispersion = 0.02 * (0.5 + elect);
vec2 dispOffset = normalize(projectedUV) * dispersion;
vec2 uvR = tiledUV - dispOffset;
vec2 uvG = tiledUV;
vec2 uvB = tiledUV + dispOffset;
float rChannel = texture(samp, clamp(uvR, 0.0, 1.0)).r;
float gChannel = texture(samp, clamp(uvG, 0.0, 1.0)).g;
float bChannel = texture(samp, clamp(uvB, 0.0, 1.0)).b;
vec3 texColor = vec3(rChannel, gChannel, bChannel);
vec3 neon = neonPalette(time_f + dist);
float glowMask = smoothstep(0.4, 0.9, elect);
vec3 finalColor = texColor + (neon * glowMask * 0.8);
finalColor = finalColor / (1.0 + finalColor * 0.3);
color = vec4(finalColor, 1.0);
}

```

Shader #225 - distort disp

File: distort-disp.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main() {
    float displacement = pingPong(tc.x * iResolution.x + time_f * 50.0, 20.0);
    vec2 distortedTC = vec2(
        atan(tc.y - 0.5, tc.x - 0.5) / 3.14159 + 0.5,
        tc.y + sin(time_f * 2.0) * 0.02
    );
    distortedTC.x += displacement / iResolution.x;
    color = texture(samp, sin(distortedTC * time_f));
}

```

```
}
```

Shader #227 - distortf

File: distortf.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
    float fractalFactor = sin(length(offset) * 10.0 + time_f * 2.0) * 0.1;
    vec2 fractalUV = uv + fractalFactor * normalize(offset);
    float grid = abs(sin(fractalUV.x * 50.0) * sin(fractalUV.y * 50.0));
    grid = step(0.7, grid);
    float angle = atan(offset.y, offset.x) + fractalFactor * sin(time_f);
    float radius = length(offset);
    vec2 swirlUV = center + radius * vec2(cos(angle), sin(angle));
    vec2 combinedUV = mix(swirlUV, fractalUV, 0.5);
    vec4 texColor = texture(samp, combinedUV);
    vec3 rainbow = vec3(0.5 + 0.5 * sin(time_f + texColor.r),
                       0.5 + 0.5 * sin(time_f + texColor.g + 2.0),
                       0.5 + 0.5 * sin(time_f + texColor.b + 4.0));
    vec3 finalColor = mix(texColor.rgb * rainbow, vec3(grid), 0.3);
    color = vec4(finalColor, texColor.a);
}
```

Shader #228 - distortf2

File: distortf2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
```

```

float dist = length(offset);
float pulse = sin(dist * 30.0 - time_f * 10.0) * 0.1;
vec2 pulseUV = uv + pulse * normalize(offset);
float fractalFactor = cos(length(offset) * 15.0 + time_f * 5.0) * 0.2;
vec2 fractalUV = pulseUV + fractalFactor * normalize(offset);
vec2 explosionUV = fractalUV + dist * vec2(cos(time_f * 4.0), sin(time_f * 4.0)) * 0.2;
vec3 rainbow = vec3(0.5 + 0.5 * sin(time_f + explosionUV.x * 10.0),
                   0.5 + 0.5 * sin(time_f + explosionUV.y * 10.0),
                   0.5 + 0.5 * sin(time_f * 2.0));
float angle = atan(offset.y, offset.x) + sin(time_f * 3.0) * 2.0;
vec2 swirlUV = center + dist * vec2(cos(angle), sin(angle)) * 1.5;
vec2 combinedUV = mix(swirlUV, explosionUV, 0.7);
vec4 texColor = texture(samp, combinedUV);
vec3 intenseColor = texColor.rgb * rainbow * 2.0;
float glow = exp(-10.0 * dist) * 2.0;
vec3 finalColor = mix(intenseColor, vec3(glow), 0.4);
color = vec4(finalColor, texColor.a);
}

```

Shader #229 - distortf3

File: distortf3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc * 2.0 - 1.0;
    uv *= iResolution.y / iResolution.x;
    vec2 center = vec2(0.0, 0.0);
    vec2 offset = uv - center;
    float dist = length(offset);
    float pulse = sin(time_f * 2.0 + dist * 10.0) * 0.15;
    float expansion = cos(time_f * 3.0 - dist * 15.0) * 0.2;
    float spiral = sin(atan(offset.y, offset.x) * 3.0 + time_f * 2.0) * 0.1;
    vec2 morphUV = uv + normalize(offset) * (pulse + expansion);
    morphUV += vec2(cos(atan(offset.y, offset.x)), sin(atan(offset.y, offset.x))) * spiral;
    vec4 texColor = texture(samp, morphUV * 0.5 + 0.5);
    texColor.rgb *= 1.0 + 0.3 * sin(dist * 20.0 - time_f * 5.0);
    color = vec4(texColor.rgb, texColor.a);
}

```

Shader #230 - distortf4

File: distortf4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution


```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc * 2.0 - 1.0;
    uv *= iResolution.y / iResolution.x;
    vec2 center = vec2(0.0, 0.0);
    vec2 offset = uv - center;
    float dist = length(offset);
    float angle = atan(offset.y, offset.x);
    float ripple = sin(dist * 30.0 - time_f * 10.0) * 0.05;
    float spiral = sin(angle * 5.0 + time_f) * 0.05;
    float wave = sin(uv.x * 10.0 + time_f) * 0.03;
    vec2 warpedUV = uv + normalize(offset) * ripple;
    warpedUV += vec2(cos(angle), sin(angle)) * spiral;
    warpedUV.y += wave;
    vec4 texColor = texture(samp, warpedUV * 0.5 + 0.5);
    float shimmer = sin(dist * 10.0 + time_f * 5.0) * 0.1 + 0.9;
    vec3 finalColor = texColor.rgb * shimmer;
    color = vec4(finalColor, texColor.a);
}

```

Shader #234 - drain bend

File: drain_bend.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
 - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    float loopDuration = 25.0;
    float currentTime = mod(time_f, loopDuration);
    vec2 aspect = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 normCoord = (tc * 2.0 - 1.0) * aspect;
    normCoord.x = abs(normCoord.x);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralSpeed = 5.0;
    float inwardSpeed = currentTime / loopDuration;
    angle += (1.0 - smoothstep(0.0, 8.0, dist)) * currentTime * spiralSpeed;
    dist *= 1.0 - inwardSpeed;
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * tan(dist);
    spiralCoord = (spiralCoord / aspect + 1.0) * 0.5;
    vec2 uv = spiralCoord;
    vec2 c = uv - 0.5;
    float r = length(c);
    float t = time_f;
}

```

```

float k1 = 0.25 * sin(t * 0.6);
float k2 = 0.10 * cos(t * 0.4);
float rd = 1.0 + k1 * r*r + k2 * r*r*r*r;
c *= rd;
c.x += 0.015 * sin(8.0 * c.y + t * 1.3);
c.y += 0.015 * sin(8.0 * c.x - t * 1.1);
uv = clamp(c + 0.5, 0.0, 1.0);
color = texture(samp, uv);
}

```

Shader #244 - ds1

File: ds1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Center coordinates and scale Polar coordinates conversion Kaleidoscope parameters

Technique: Polar coordinate transformation (atan2 + length) - UV mirroring/folding for symmetry
- Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    const int slices = 6;
    float angleSlice = 2.0 * 3.14159 / float(slices);
    angle = mod(angle, angleSlice * 2.0);
    angle = abs(angle - angleSlice);
    angle += time_f * 0.5;
    radius *= 1.0 + 0.1 * sin(time_f * 2.0 + angle * 5.0);
    angle += sin(time_f * 1.5 + radius * 5.0) * 0.3;
    vec2 distortedUV = vec2(cos(angle), sin(angle)) * radius;
    distortedUV *= 0.5 + 0.3 * sin(time_f + radius * 3.0);
    distortedUV = abs(fract(distortedUV * 1.5) * 2.0 - 1.0);
    vec2 finalUV = (distortedUV + 1.0) * 0.5;
    finalUV = fract(finalUV + vec2(time_f * 0.1, 0.0));
    color = texture(samp, finalUV);
    color.rgb = mix(color.rgb, fract(color.rgb + time_f * 0.1), 0.2);
}

```

Shader #316 - effect

File: effect.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform vec2 iResolution;
uniform float time_f;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void) {
    vec2 uv = tc;
    vec2 warp = uv + vec2(
        sin(uv.y * 10.0 + time_f) * 0.1,
        sin(uv.x * 10.0 + time_f) * 0.1
    );
    vec3 colorShift = vec3(
        0.5 * sin(time_f * 0.5) + 0.5,
        0.5 * sin(time_f * 0.7 + 2.0) + 0.5,
        0.5 * sin(time_f * 0.3 + 4.0) + 0.5
    );
    float feedback = rand(uv + time_f);
    vec2 feedbackUv = warp;
    vec4 texColor = texture(samp, feedbackUv);
    vec3 finalColor = texColor.rgb + colorShift;
    color = sin(vec4(finalColor, texColor.a) * (time_f * 0.5));
    color.a = 1.0;
}

```

Shader #317 - elastic

File: elastic.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Compute vector from center Elastic stretching and bending effect Apply transformations

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 tcFromCenter = tc - center;
    float distance = length(tcFromCenter);
    float angle = atan(tcFromCenter.y, tcFromCenter.x);
    float spinSpeed = time_f * 2.0;
    angle += spinSpeed;
    float elasticity = 0.3 + sin(time_f * 2.0) * 0.2;
    float stretch = 1.0 + sin(time_f * 3.0) * 0.3;
    float wave = sin(distance * 10.0 + time_f * 5.0) * elasticity;
    float radius = distance * (stretch + wave);
    vec2 distortedTC = center + vec2(cos(angle), sin(angle)) * radius;
    distortedTC = clamp(distortedTC, 0.0, 1.0);
    color = texture(samp, distortedTC);
}

```

Shader #341 - fisheye

File: fisheye.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    float strength = 0.5 + sin(time_f) * 0.5 * 8.0;
    vec2 tcFromCenter = tc - center;
    float distance = length(tcFromCenter);
    float angle = atan(tcFromCenter.y, tcFromCenter.x);
    float radius = distance * (1.0 + strength * (distance * distance));
    vec2 distortedTC = center + vec2(cos(angle), sin(angle)) * radius;
    distortedTC = clamp(distortedTC, 0.0, 1.0);
    color = texture(samp, distortedTC);
}
```

Shader #343 - fisheye pi.mp4

File: fisheye_pi.mp4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Driven by: time, mouse, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p = (tc - m) * ar;
    float ax = 0.28 * sin(time_f * 0.7);
    float ay = 0.28 * cos(time_f * 0.6);
    float az = 0.5 * time_f;
    mat3 R = rotZ(az) * rotY(ay) * rotX(ax);
    vec3 q3 = R * vec3(p, 1.0);
    float persp = 0.7;
    float zf = 1.0 / (1.0 + q3.z * persp);
    vec2 q = q3.xy * zf;
```

```

float d = length(q);
float w = 1.0 - smoothstep(0.0, 0.55, length((tc - m) * ar));
float s = 0.5 + 4.0 * sin(pingPong(time_f * PI, 15.0));
float ang = atan(q.y, q.x);
float radius = d * (1.0 + s * w * (d * d));
vec2 r = vec2(cos(ang), sin(ang)) * sin(radius * time_f);
vec2 uv = r / ar + m;
uv = clamp(uv, 0.0, 1.0);
color = texture(samp, uv);
}

```

Shader #344 - fisheye warp

File: fisheye_warp.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec2 uv = tc - 0.5;
    float len = length(uv);
    float factor = sqrt(len) * 0.5;
    uv *= (1.0 + sin(factor * time_t));
    color = texture(samp, uv + 0.5);
}

```

Shader #352 - frac cache

File: frac_cache.glsl

Renders a fractal pattern using iterative computation (many iterations). Texture samplers
External uniforms Resolution of the viewport/window

Technique: Iterative fractal computation with complex-plane mapping - Multi-tap convolution blur (6 texture samples) - Polar coordinate transformation (atan2 + length) - Temporal feedback using cached frame buffers - 2D rotation matrix - Linear mix/lerp blending - Distance field computation - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform sampler2D samp1;
uniform sampler2D samp2;
uniform sampler2D samp3;
uniform sampler2D samp4;
uniform vec2 iResolution;

```

```

uniform float time_f;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec2 rotate2D(vec2 p, float angle) {
    mat2 r = mat2(cos(angle), -sin(angle),
                  sin(angle),  cos(angle));
    return r * p;
}
float fractalJulia(vec2 z, vec2 c, int iterations) {
    float m = 0.0;
    for(int i = 0; i < iterations; i++) {
        z = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + c;
        if(dot(z,z) > 4.0) {
            m = float(i) / float(iterations);
            break;
        }
    }
    return m;
}
vec2 swirl(vec2 uv, vec2 center, float radius, float strength) {
    float dist = distance(uv, center);
    if(dist < radius) {
        float percent = (radius - dist) / radius;
        float theta = percent * percent * strength;
        vec2 dir = uv - center;
        dir = rotate2D(dir, theta);
        uv = dir + center;
    }
    return uv;
}
vec2 kaleidoscope(vec2 p, float slices) {
    float angle = 2.0 * 3.1415926535 / slices;
    float r = length(p);
    float a = atan(p.y, p.x);
    a = mod(a, angle);
    return vec2(r*cos(a), r*sin(a));
}
void main(void)
{
    vec2 uv = 2.0 * (tc - 0.5);
    {
        vec2 swirlUV = swirl(uv, vec2(0.0), 1.0, sin(time_f) * 5.0);
        swirlUV = rotate2D(swirlUV, time_f * 0.2);
        swirlUV = swirlUV * 0.5 + 0.5;
        vec4 col1 = texture(samp1, swirlUV);
        col1 *= 1.2;
        vec4 colorSwirl = col1;
    }
    vec4 colorFractal2;
    {
        vec2 fractalUV = uv * 2.5;
        float fractVal = fractalJulia(fractalUV,
                                     vec2(sin(time_f*0.3)*0.5, cos(time_f*0.4)*0.5),
                                     16);
        vec2 newUV = (uv + fractVal * 0.3) * 0.5 + 0.5;
        colorFractal2 = texture(samp2, newUV);
        colorFractal2.rgb = mix(colorFractal2.rgb, vec3(fractVal, fractVal*0.7, fractVal*0.4), 0.5);
    }
}

```

```

}
vec4 colorKaleido3;
{
    vec2 kaleidoUV = kaleidoscope(uv, 6.0);
    kaleidoUV = rotate2D(kaleidoUV, time_f * 0.5);
    kaleidoUV *= (0.8 + sin(time_f * 0.7) * 0.2);
    kaleidoUV = kaleidoUV * 0.5 + 0.5;
    colorKaleido3 = texture(samp3, kaleidoUV);
    colorKaleido3.rgb = colorKaleido3.rgb * 1.3;
}
vec4 colorFractal4;
{
    vec2 swirl4UV = swirl(uv, vec2(0.0), 1.0, -sin(time_f*0.5)*3.0);
    swirl4UV = rotate2D(swirl4UV, time_f * 0.3);
    float fractVal2 = fractalJulia(swirl4UV*2.0, vec2(0.3, 0.5), 12);
    swirl4UV += fractVal2 * 0.1;
    swirl4UV = swirl4UV * 0.5 + 0.5;
    colorFractal4 = texture(samp4, swirl4UV);
    colorFractal4.rgb *= mix(0.5, 1.5, fractVal2);
}
vec4 baseColor = texture(samp, tc);
float hueShift = sin(time_f*0.3)*0.2;
mat3 hueRotation = mat3(
    1.0,      0.0,      0.0,
    0.0, cos(hueShift), -sin(hueShift),
    0.0, sin(hueShift),  cos(hueShift)
);
baseColor.rgb = hueRotation * baseColor.rgb;
vec4 colorSwirl = texture(samp1, swirl(uv, vec2(0.0), 1.0, sin(time_f) * 5.0) * 0.5 + 0.5);
vec4 colorFractal2F = colorFractal2;
vec4 colorKaleido3F = colorKaleido3;
vec4 colorFractal4F = colorFractal4;
vec4 combo =
    0.25 * colorSwirl +
    0.25 * colorFractal2F +
    0.25 * colorKaleido3F +
    0.25 * colorFractal4F;
vec4 finalColor = mix(baseColor, combo, 0.7);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalColor.rgb *= 1.0 + _ab * 0.6;
finalColor.rgb = mix(finalColor.rgb, finalColor.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15,
↪ 1.0 + clamp(amp_high, 0.0, 1.0) * 0.25), _ab);
color = finalColor;
}

```

Shader #372 - frac shader02 dmd6i air twist

File: frac_shader02_dmd6i_air_twist.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
 - Indexed color palette look-up table - 2D rotation matrix - UV mirroring/folding for symmetry
 - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

#version 330 core

```

in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
}

```



```

    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv, float tSlow, float tFast, float aspect) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 gdir = normalize(vec2(cos(tSlow * 0.27), sin(tSlow * 0.31)));
    float s = dot((uv - 0.5) * vec2(aspect, 1.0), gdir);
    float w = max(fwidth(s) * 4.0, 0.002);
    float band = smoothstep(-0.5 - w, -0.5 + w, sin(s * 2.2 + tFast * 0.9));
    vec3 neon = neonPalette(tFast);
    vec3 grad = mix(tex, mix(tex, neon, 0.6), 0.35 + 0.25 * band);
    grad = mix(grad, tex, 0.10);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 rot2(vec2 v, float a) {
    float c = cos(a), s = sin(a);
    return vec2(c * v.x - s * v.y, s * v.x + c * v.y);
}

vec2 h2(vec2 p) {
    return fract(sin(vec2(dot(p, vec2(127.1, 311.7)),
                        dot(p, vec2(269.5, 183.3))))) * 43758.5453;
}

vec3 limitHighlights(vec3 c) {
    float m = max(c.r, max(c.g, c.b));
    if (m > 0.9) c *= 0.9 / m;
    return c;
}

void main(void) {
    float a = clamp(amp, 0.0, 1.0);

```

```

float ua = clamp(uamp, 0.0, 1.0);
float aMix = clamp(amp * 0.7 + uamp * 1.3, 0.0, 4.0);
float a01 = clamp(aMix / 2.5, 0.0, 1.0);
float aspect = iResolution.x / iResolution.y;
vec2 ar = vec2(aspect, 1.0);
float tBase = time_f;
float tSlow = tBase * mix(0.2, 0.9, a01);
float tFast = tBase * mix(0.7, 3.5, a01);
vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution)
                        : fract(vec2(0.37 + 0.11 * sin(tSlow * 0.63 + seed),
                                0.42 + 0.13 * cos(tSlow * 0.57 + seed * 2.0)));

vec2 tc0 = tc;
vec2 center = vec2(0.5, 0.5);
float radius = length(tc0 - center);
float rippleSpeed = 5.0 * (1.0 + 2.0 * a01);
float rippleAmplitude = 0.03 * (0.5 + 1.5 * a01);
float rippleWavelength = 10.0;
float twistStrength = 1.0 + 4.0 * a01;
float ripple = sin(tc0.x * rippleWavelength + tFast * rippleSpeed) * rippleAmplitude;
ripple += sin(tc0.y * rippleWavelength + tFast * rippleSpeed) * rippleAmplitude;
vec2 rippleTC = tc0 + vec2(ripple, ripple);
float angleTwist = twistStrength * (radius - 1.0) + tSlow;
float cosA = cos(angleTwist);
float sinA = sin(angleTwist);
mat2 rotationMatrix = mat2(cosA, -sinA, sinA, cosA);
vec2 twistedTC = rotationMatrix * (tc0 - center) + center;
float mixRT = 0.5 + 0.35 * a01;
vec2 tcRT = mix(rippleTC, twistedTC, mixRT);
vec4 baseTex = texture(samp, tcRT);
vec2 uvForGlow = tcRT * 2.0 - 1.0;
uvForGlow.x *= aspect;
float rGlow = pingPong(sin(length(uvForGlow) * tSlow), 5.0);
float radiusMax = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radiusMax, radiusMax - 0.25, rGlow);
vec2 uv = tcRT;
float speedScale = 1.0 + 2.0 * a01 + 3.0 * ua;
float speedR = 5.0 * speedScale;
float speedG = 6.5 * speedScale;
float speedB = 4.0 * speedScale;
float ampR = 0.03 * (0.7 + 0.6 * a01);
float ampG = 0.025 * (0.7 + 0.6 * a01);
float ampB = 0.035 * (0.7 + 0.6 * a01);
float waveR = 10.0;
float waveG = 12.0;
float waveB = 8.0;
float rR = sin(uv.x * waveR + tFast * speedR) * ampR
        + sin(uv.y * waveR * 0.8 + tFast * speedR * 1.2) * ampR;
float rG = sin(uv.x * waveG * 1.5 + tFast * speedG) * ampG
        + sin(uv.y * waveG * 0.3 + tFast * speedG * 0.7) * ampG;
float rB = sin(uv.x * waveB * 0.5 + tFast * speedB) * ampB
        + sin(uv.y * waveB * 1.7 + tFast * speedB * 1.3) * ampB;
vec2 tcR = uv + vec2(rR, rR);
vec2 tcG = uv + vec2(rG, -0.5 * rG);
vec2 tcB = uv + vec2(0.3 * rB, rB);
vec3 pats[4] = vec3[](vec3(1,0,1), vec3(0,1,0), vec3(1,0,0), vec3(0,0,1));
float pspd = 4.0;
int pidx = int(mod(floor(tBase * pspd + seed * 4.0), 4.0));
vec3 mir = pats[pidx];
vec2 dR = tcR - m;
vec2 dG = tcG - m;
vec2 dB = tcB - m;
float fallR = smoothstep(0.55, 0.0, length(dR));

```

```

float fallG = smoothstep(0.55, 0.0, length(dG));
float fallB = smoothstep(0.55, 0.0, length(dB));
float sw = (0.12 + 0.38 * ua + 0.25 * a);
vec2 tangR = rot2(normalize(dR + 1e-4), 1.5707963);
vec2 tangG = rot2(normalize(dG + 1e-4), 1.5707963);
vec2 tangB = rot2(normalize(dB + 1e-4), 1.5707963);
vec2 airR = tangR * sw * fallR * (0.06 + 0.22 * a) * (0.6 + 0.4 * cos(uv.y * 40.0 + tFast * 3.0 +
↳ seed));
vec2 airG = tangG * sw * fallG * (0.06 + 0.22 * a) * (0.6 + 0.4 * cos(uv.y * 38.0 + tFast * 3.3 +
↳ seed * 1.7));
vec2 airB = tangB * sw * fallB * (0.06 + 0.22 * a) * (0.6 + 0.4 * cos(uv.y * 42.0 + tFast * 2.9 +
↳ seed * 0.9));
vec2 jit = (h2(uv * vec2(233.3, 341.9) + tFast + seed) - 0.5)
    * (0.0006 + 0.004 * ua);
tcR += airR + jit;
tcG += airG + jit;
tcB += airB + jit;
vec2 fR = vec2(mir.r > 0.5 ? 1.0 - tcR.x : tcR.x, tcR.y);
vec2 fG = vec2(mir.g > 0.5 ? 1.0 - tcG.x : tcG.x, tcG.y);
vec2 fB = vec2(mir.b > 0.5 ? 1.0 - tcB.x : tcB.x, tcB.y);
float seg = 4.0 + 2.0 * sin(tSlow * 0.33 + a01 * 2.0);
vec2 kUV = reflectUV(tcRT, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + 0.55 * sin(tSlow * 0.42 + a01 * 3.0);
kUV = fractalFold(kUV, foldZoom, tFast, m, aspect);
kUV = rotateUV(kUV, tSlow * 0.23 + a01 * 1.1, m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(tSlow * 0.2) * (PI * tSlow), 5.0);
float period = log(base) * pingPong(tSlow * PI, 5.0);
float tz = tSlow * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35
    + 0.35 * sin(rD * 18.0 + tFast * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dirK = normalize(pwrap + 1e-6);
vec2 off = dirK * (0.0015 + 0.001 * sin(tFast * 1.3)) * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tcRT - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec2 sR = mix(u0, fR, 0.6);
vec2 sG = mix(u1, fG, 0.6);
vec2 sB = mix(u2, fB, 0.6);
vec3 rC = preBlendColor(sR + off, tSlow, tFast, aspect);
vec3 gC = preBlendColor(sG, tSlow, tFast, aspect);
vec3 bC = preBlendColor(sB - off, tSlow, tFast, aspect);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + tFast * 1.2));
ring = ring * pingPong(tSlow * PI, 5.0);
float pulse = 0.5 + 0.5 * sin(tFast * 2.0 + rD * 28.0 + k * 12.0);
pulse *= mix(0.4, 1.4, a01);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.10
    + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.06;
float pulseAdd = 0.004 * (0.5 + 0.5 * sin(tFast * 3.7 + seed));

```

```

    outCol += bloom;
    outCol += pulseAdd * ua;
    outCol = clamp(outCol, vec3(0.0), vec3(1.0));
    outCol = limitHighlights(outCol);
    vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * (0.6 + 0.3 * a01));
    finalRGB = limitHighlights(finalRGB);
    finalRGB = clamp(finalRGB, 0.0, 1.0);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #392 - frac shader02 dmdXi3 warp

File: frac_shader02_dmdXi3_warp.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

```

```

}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec2 spinWarp(vec2 uv, float t){
    vec2 center = vec2(0.5);
    float angle = atan(uv.y - center.y, uv.x - center.x);
    float modulatedTime = pingPong(t, 5.0);
    angle += modulatedTime;
    vec2 rel = uv - center;
    float s = sin(angle), c = cos(angle);
    vec2 rot;
    rot.x = c * rel.x - s * rel.y;
    rot.y = s * rel.x + c * rel.y;
    vec2 rotated = rot + center;
    float warpFactor = sin(t) * 0.5 + 0.5;
    vec2 warped;
    warped.x = pingPong(rotated.x + t * 0.1, 1.0);
    warped.y = pingPong(rotated.y + t * 0.1, 1.0);
    return mix(rotated, warped, warpFactor);
}
void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
}

```

```

float t = time_f + iTime * (PI / 2.0);
float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
float osc = 0.5 + 0.5 * sin(t * 0.35 + 0.2 * sr);
float seg = 4.0 + 2.0 * sin(t * 0.25);
float zoom = mix(1.1, 2.1, osc);
float spinAngle = osc * 2.0 * PI;
vec2 baseUV = spinWarp(tc, t);
vec2 tcMix = mix(tc, baseUV, 0.75);
vec2 kUV = reflectUV(tcMix, seg, m, ar.x);
kUV = fractalFold(kUV, zoom, t, m, ar.x);
kUV = rotateUV(kUV, spinAngle, m, ar.x);
kUV = spinWarp(kUV, t * 0.7);
vec2 dir = (kUV - m) * ar;
float r = length(dir);
float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
ripple = sin(ripple * pingPong(time_f * PI, 3.0));
vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
vec2 uB = mix(baseUV, kUV, 0.88);
vec2 uC = mix(tcMix, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
float hueBase = fract(t * 0.06);
float sat = 0.7 + 0.25 * sin(t * 0.4 + sr);
float val = 0.6 + 0.35 * sin(t * 0.5 + dot(tc, vec2(2.3, 1.9)));
vec3 hsvTint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
vec3 neon = neonPalette(hueBase + r * 0.15);
vec3 tint = normalize(hsvTint * 0.7 + neon * 1.3);
float slowBeat = 0.5 + 0.5 * sin(t * 0.8 + aMix * 0.05);
float mix1 = 0.5 + 0.5 * sin(t * 0.6);
float mix2 = 0.5 + 0.5 * cos(t * 0.45);
vec3 warpCol = mix(t1.rgb, t2.rgb, mix1);
warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
warpCol *= tint * (0.92 + 0.08 * slowBeat);
vec3 base = textureGrad(samp, baseUV, dFdx(baseUV), dFdy(baseUV)).rgb;
vec3 combined = mix(base, warpCol * 3.0, 0.6);
vec3 bloom = combined * combined * 0.18 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.10;
combined += bloom;
combined = clamp(combined, vec3(0.0), vec3(1.0));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
combined *= 1.0 + _ab * 0.6;
combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(combined, 1.0);
}

```

Shader #454 - g5

File: g5.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections.
 Normalize coordinates to center and adjust for aspect ratio Create spiral effect Add underwater
 wobble distortion

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic
 aberration - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc - 0.5;
    uv.x *= iResolution.x / iResolution.y;
    float radius = length(uv) * 2.0;
    float angle = atan(uv.y, uv.x) + time_f * 0.5;
    float wobble = sin(time_f * 2.0 + radius * 5.0) * 0.01;
    vec2 spiralUV = vec2(
        cos(angle) * radius + sin(time_f + uv.y * 10.0) * 0.02 + wobble,
        sin(angle) * radius + cos(time_f + uv.x * 10.0) * 0.02 + wobble
    );
    float chromaOffset = 0.005;
    vec4 texColor;
    texColor.r = texture(samp, spiralUV + chromaOffset).r;
    texColor.g = texture(samp, spiralUV).g;
    texColor.b = texture(samp, spiralUV - chromaOffset).b;
    texColor.a = 1.0;
    vec3 waterColor = vec3(0.2, 0.4, 0.6);
    float depthEffect = clamp(radius * 0.8, 0.0, 1.0);
    vec3 tinted = mix(texColor.rgb, waterColor, 0.3 * depthEffect);
    float lightPattern = sin(uv.x * 20.0 + time_f * 1.5) *
        sin(uv.y * 15.0 + time_f) * 0.1 + 0.9;
    color = vec4(tinted * lightPattern, 1.0);
}

```

Shader #457 - g swirl

File: g_swirl.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright.

Technique: Distance field computation - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float ripple(vec2 pos, float time, float speed, float frequency) {
    float aspectRatio = iResolution.x / iResolution.y;
    pos.x *= aspectRatio;
    float d = distance(pos, vec2(0.5 * aspectRatio, 0.5));
    return sin(d * frequency - time * speed) * exp(-d * 3.0);
}
vec3 psychedelicColors(float value) {
    return 0.5 + 0.5 * cos(6.2831 * vec3(0.0, 0.33, 0.66) + value);
}
void main(void) {
    vec2 pos = tc - 0.5;
    pos.y *= iResolution.y / iResolution.x;
    float rippleDist = ripple(pos, time_f, 0.4, 10.0);
    vec2 distortedPos = pos + 0.05 * vec2(cos(rippleDist * 3.0), sin(rippleDist * 2.0));
    distortedPos += 0.1 * vec2(sin(time_f * 0.8), cos(time_f * 0.8));
    vec4 texColor = texture(samp, distortedPos + 0.5);
}

```

```

    vec3 colorCycle = psychedelicColors(time_f + rippleDist);
    color = vec4(texColor.rgb * colorCycle, texColor.a);
}

```

Shader #463 - gboil

File: gboil.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float rand(vec2 co) {
    return fract(sin(dot(co, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void) {
    vec2 uv = tc;
    float frequency = 30.0;
    float amplitude = 0.05;
    float speed = 5.0;
    float t = time_f * speed;
    float noiseX = rand(uv * frequency + t);
    float noiseY = rand(uv * frequency - t);
    float distortionX = amplitude * pingPong(t + uv.y * frequency + noiseX, 1.0);
    float distortionY = amplitude * pingPong(t + uv.x * frequency + noiseY, 1.0);
    vec2 tc_mod = uv + vec2(distortionX, distortionY);
    color = texture(samp, tc_mod);
}

```

Shader #464 - gem af

File: gem-af.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations. Scale space: This creates the "infinite zoom" feel Mix the original texture with the mathematical fractal glow Add a subtle "bloom" effect based on the fractal edges

Technique: UV mirroring/folding for symmetry - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec2 mirror(vec2 uv) {

```



```

    return abs(fract(uv * 0.5 + 0.5) * 2.0 - 1.0);
}
void main(void) {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float iter = 0.0;
    float max_iter = 6.0;
    float time_pulse = sin(time_f * 0.2) * 0.5 + 0.5;
    for (float i = 0.0; i < max_iter; i++) {
        uv = abs(uv) - 0.5;
        float angle = time_f * 0.1 + i * 0.5;
        float s = sin(angle), c = cos(angle);
        uv *= mat2(c, -s, s, c);
        uv *= 1.1 + 0.1 * time_pulse;
        iter = i;
    }
    vec2 fractal_tc = mirror(uv);
    vec4 texColor = texture(samp, fractal_tc);
    vec3 fractalCol = 0.5 + 0.5 * cos(vec3(0.0, 2.0, 4.0) + length(uv) + time_f);
    vec3 finalRGB = mix(texColor.rgb, fractalCol, 0.4);
    float bloom = 0.02 / abs(sin(length(uv) - time_f));
    finalRGB += bloom * fractalCol;
    color = vec4(finalRGB, 1.0);
}

```

Shader #468 - gem deep

File: gem-deep.glsl

Renders a fractal pattern using iterative computation (64 iterations). 'p' is our moving point, 'c' is the constant based on mouse or time Smoothly zoom in and out over time Space folding (Abs creates symmetry, like your old x%2 check but infinite)

Technique: Iterative fractal computation with complex-plane mapping - Bitwise XOR on integer color channels - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    vec2 p = uv;
    float iters = 0.0;
    const float max_iters = 64.0;
    float zoom = pow(0.5, mod(time_f * 0.5, 10.0));
    p *= zoom;
    for(float i = 0.0; i < max_iters; i++) {
        p = abs(p) / dot(p, p) - vec2(0.8, 0.5 + 0.1 * sin(time_f * 0.3));
        if (length(p) > 20.0) break;
        iters++;
    }
    float normIters = iters / max_iters;
    vec4 texColor = texture(samp, tc + p * 0.02);
    vec3 fractalColor;
    fractalColor.r = normIters * 2.0;
    fractalColor.g = sin(iters * 0.5 + time_f);
}

```

```

    fractalColor.b = length(p) * 0.1;
    float temp = fractalColor.r;
    fractalColor.r = fractalColor.b;
    fractalColor.b = temp;
    vec3 finalColor = (sin(time_f) > 0.0) ? vec3(1.0) - fractalColor : fractalColor;
    vec3 composite = mix(texColor.rgb, finalColor, 0.7);
    color = vec4(sin(composite * time_f), 1.0);
}

```

Shader #476 - gem spiral full

File: gem-spiral-full.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Discards pure black pixels for transparency. Instead of a fixed sphereRadius, we use a 'lens strength' This creates a bulge that covers the whole screen. Hyperbolic-style depth

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
 - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = (tc * 2.0 - 1.0);
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float d = length(uv);
    float lensStrength = 1.5;
    float z = sqrt(lensStrength * lensStrength + d * d);
    vec3 normal = normalize(vec3(uv, 1.0 / lensStrength));
    float fisheyeRadius = atan(d, 1.0);
    vec2 distortedUV = normalize(uv + 1e-6) * fisheyeRadius;
    float t = time_f * 0.8;
    float pTime = pingPong(time_f * 0.5, 2.0);
    float r_dist = length(distortedUV);
    float angle = atan(distortedUV.y, distortedUV.x);
    float spiral = angle + (log(r_dist + 0.1) * (2.0 + pTime)) - t * 1.5;
    float r = sin(spiral * 3.0 + t);
    float g = sin(spiral * 3.0 + t + 2.094);
    float b = sin(spiral * 3.0 + t + 4.188);
    vec3 spiralCol = vec3(r, g, b) * 0.5 + 0.5;
    vec3 lightDir = normalize(vec3(sin(time_f), cos(time_f), 1.0));
    float diff = max(dot(normal, lightDir), 0.0);
    float spec = pow(max(dot(reflect(-lightDir, normal), vec3(0,0,1)), 0.0), 16.0);
    vec3 texColor = texture(samp, tc).rgb;
    float mixFactor = smoothstep(0.0, 1.5, d);
    vec3 finalCol = mix(texColor, spiralCol * (diff + 0.5) + spec, 0.7);
    finalCol *= smoothstep(2.0, 0.5, d);
    color = vec4(finalCol, alpha);
}

```

Shader #477 - gem spoke

File: gem-spoke.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations. Maintain aspect ratio Folding space: mirroring the UVs creates symmetrical complexity Rotation over time to create a "Kaleidoscope" effect

Technique: UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Radial vignette falloff - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = (tc * 2.0 - 1.0);
    uv.x *= iResolution.x / iResolution.y;
    float zoom = sin(time_f * 0.2) * 0.5 + 1.5;
    vec3 fractalCol = vec3(0.0);
    float scale = 1.0;
    for (int i = 0; i < 6; i++) {
        uv = abs(uv) - 0.5;
        float a = time_f * 0.3;
        float s = sin(a), c = cos(a);
        uv *= mat2(c, -s, s, c);
        uv *= 1.2;
        scale *= 1.2;
        float d = length(uv);
        float glow = 0.01 / abs(sin(d * 8.0 - time_f) / 8.0);
        fractalCol += glow * vec3(0.2, 0.5, 1.0) / scale;
    }
    vec4 texColor = texture(samp, fract(uv + time_f * 0.1));
    vec3 finalRGB = texColor.rgb * fractalCol;
    finalRGB *= smoothstep(1.5, 0.5, length(tc * 2.0 - 1.0));
    color = vec4(finalRGB, 1.0);
}
```

Shader #479 - gem frac spir

File: gem_frac_spir.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Your original XOR logic Optimization: Using textureLod or a smaller kernel for performance if needed, but keeping your specific blur logic here.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
```

```

vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernelVals[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

mat2 rotate2d(float a) {
    return mat2(cos(a), -sin(a), sin(a), cos(a));
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float t_spiral = pingPong(time_f * 0.5, 5.0);
    float t_zoom = 1.0 + 0.5 * sin(time_f * 0.3);
    for(int i = 0; i < 4; i++) {
        uv = abs(uv) - 0.5;
        uv *= rotate2d(t_spiral * 0.2 + float(i));
        uv *= t_zoom;
    }
    float r = length(uv);
    float angle = atan(uv.y, uv.x);
    float spiral = angle + log(r + 0.1) * t_spiral;
    vec2 fractalTC = fract(uv * 0.5 + 0.5);
    vec4 tcolor = blur(samp, fractalTC, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec3 rainbow_color = 0.5 + 0.5 * cos(6.28318 * (spiral + time_f * 0.5 + vec3(0,2,4)/3.0));
    vec3 blended_color = mix(tcolor.rgb, rainbow_color, 0.6);
    vec4 xorColor = xor_RGB(
        vec4(sin(blended_color * time_t), 1.0),

```

```

        vec4(cos(blended_color * time_t), 1.0)
    );
    float vign = smoothstep(1.5, 0.2, length(tc * 2.0 - 1.0));
    color = vec4(xorColor.rgb * vign, tcolor.a);
}

```

Shader #485 - gem polar

File: gem_polar.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. 'dist' is the radius from the center 'angle' is the rotation around the center Using log(dist) makes the "tunnel" perspective smooth and infinite

Technique: Polar coordinate transformation (atan2 + length) - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float dist = length(uv);
    float angle = atan(uv.x, uv.y) / 3.14159;
    vec2 warpedTC;
    warpedTC.x = angle + (time_f * 0.05);
    warpedTC.y = (1.0 / (dist + 0.01)) + (time_f * 0.5);
    vec2 finalTC = abs(fract(warpedTC * 0.5) * 2.0 - 1.0);
    vec4 texColor = texture(samp, finalTC);
    vec3 rainbow = 0.5 + 0.5 * cos(6.28318 * (dist - time_f * 0.4 + vec3(0.0, 0.33, 0.67)));
    float vignette = smoothstep(0.0, 0.1, dist) * smoothstep(1.5, 0.5, dist);
    color = texColor * vec4(rainbow, 1.0) * vignette;
}

```

Shader #488 - gem txt

File: gem_txt.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Function to generate "fake" scrolling code/text patterns Vertical scroll speed Fish-eye math: pushes coordinates outward from the center

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
float text(vec2 p) {
    p.y += time_f * 0.2;
}

```

```

    vec2 grid = floor(p * vec2(20.0, 40.0));
    float h = fract(sin(dot(grid, vec2(12.9898, 78.233))) * 43758.5453);
    return step(0.5, h) * step(0.3, fract(p.x * 20.0)) * step(0.3, fract(p.y * 40.0));
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float r = length(uv);
    vec2 bulgeUV = tc + (uv * r * 0.2);
    float chromaticPower = 0.015 * r;
    float r_ch = texture(samp, bulgeUV + vec2(chromaticPower, 0.0)).r;
    float g_ch = texture(samp, bulgeUV).g;
    float b_ch = texture(samp, bulgeUV - vec2(chromaticPower, 0.0)).b;
    vec3 baseTex = vec3(r_ch, g_ch, b_ch);
    float characters = text(bulgeUV * 1.5);
    vec3 textColor = vec3(0.4, 1.0, 0.6) * characters * 0.4;
    float hueShift = sin(r * 3.0 - time_f) * 0.5 + 0.5;
    vec3 psychoTone = mix(vec3(0.8, 0.1, 0.5), vec3(0.1, 0.8, 0.9), hueShift);
    float flare = exp(-r * 3.5) * 2.0;
    vec3 flareColor = vec3(1.0, 1.0, 0.9) * flare;
    vec3 result = mix(baseTex, psychoTone, 0.3);
    result += textColor;
    result += flareColor;
    result *= smoothstep(1.5, 0.2, r);
    color = vec4(result, 1.0);
}

```

Shader #524 - gltichtest

File: gltichtest.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Flip texture coordinates upside down Center coordinates and calculate polar form Tornado spiral effect with time

Technique: Polar coordinate transformation (atan2 + length) - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = vec2(tc.x, 1.0 - tc.y);
    vec2 centeredUV = uv - 0.5;
    float radius = length(centeredUV);
    float angle = atan(centeredUV.y, centeredUV.x);
    float spiralStrength = 10.0;
    float spiralFactor = radius * spiralStrength + time_f * 2.0;
    angle += spiralFactor;
    float verticalStretch = radius * 0.5;
    radius = pow(radius, 0.9);
    vec2 distorted = vec2(cos(angle), sin(angle)) * radius;
    distorted.y -= verticalStretch;
    float tearShape = pow(uv.y, 2.0);
    distorted.y -= tearShape * 0.3;
    float iceRefract = sin(distorted.x * 50.0 + time_f * 4.0) * 0.01;
    iceRefract += cos(distorted.y * 40.0 + time_f * 3.0) * 0.01;
}

```

```

    distorted += iceRefract * tearShape;
    distorted += 0.5;
    float colorSplit = 0.02;
    vec4 r = texture(samp, distorted + vec2(-colorSplit, 0.0));
    vec4 g = texture(samp, distorted);
    vec4 b = texture(samp, distorted + vec2(colorSplit, 0.0));
    float edgeFade = 1.0 - smoothstep(0.4, 0.5, radius);
    color = vec4(r.r, g.g, b.b, 1.0);
}

```

Shader #525 - gmir

File: gmir.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Adapted sampleWarp to use 'samp' and hardcoded amp/freq Hardcoded values since we don't have uniforms for them anymore Using 'samp' here instead of 'texTexture'

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - Linear mix/lerp blending -
Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 5; i++) {
        v += a * noise(p);
        p *= 2.1;
        a *= 0.55;
    }
    return v;
}
vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}

```

```

vec3 sampleWarp(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    float ampControl = 1.0;
    float freqControl = 1.0;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    float f1 = fbm(p * 1.8 + t * 0.3);
    float f2 = fbm(p.yx * 2.3 - t * 0.25);
    float f3 = fbm(p * 3.1 + vec2(1.3, -0.7) * t * 0.2);
    vec2 swirl = p;
    float r = length(swirl);
    float a = atan(swirl.y, swirl.x);
    a += (f1 * 4.0 + f2 * 2.0) * strength * 0.6;
    swirl = vec2(cos(a), sin(a)) * r;
    float sliceBase = 8.0;
    float sliceRange = 8.0;
    float slices = sliceBase + sliceRange * (0.3 + 0.7 * ampControl) + 4.0 * sin(t * 0.17);
    vec2 k = kaleido(swirl + vec2(f2, f3) * 0.4 * strength, slices);
    vec2 flow = k;
    flow.x += f1 * 0.8 * strength;
    flow.y += (f2 - f3) * 0.8 * strength;
    vec2 base = flow / vec2(aspect, 1.0) + center;
    base = fract(base);
    float chromaBoost = 0.5 + 0.5 * ampControl;
    vec2 chromaShift = 0.0035 * strength * chromaBoost *
        vec2(sin(t + f1 * 6.0), cos(t * 1.3 + f2 * 6.0));
    float rC = texture(samp, base + chromaShift).r;
    float gC = texture(samp, base).g;
    float bC = texture(samp, base - chromaShift).b;
    vec3 col = vec3(rC, gC, bC);
    float bright = 0.7 + 0.6 * f3 + 0.4 * sin(t * 0.6 + f1 * 3.0);
    bright *= (0.6 + 0.8 * freqControl);
    col *= bright;
    float sat = 1.3 + 0.7 * sin(t * 0.43 + f2 * 5.0);
    sat *= (0.6 + 0.8 * freqControl);
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    col = mix(vec3(gray), col, sat);
    return col;
}

void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    float ampControl = 1.0;
    float freqControl = 1.0;
    float tSpeed = 0.3 + 1.7 * (freqControl * 0.5);
    float t = time_f * tSpeed;
    float strength = 0.6 + 1.6 * (ampControl * 0.5);
    vec2 center = vec2(0.5);
    vec3 colA = sampleWarp(uv, t, strength, center, iResolution);
    vec3 colB = sampleWarp(uv + vec2(0.01, -0.007),
        t + 3.14,
        strength * 0.9,
        center,
        iResolution);
    float blend = 0.5 + 0.5 * sin(t * 0.25);
    vec3 finalCol = mix(colA, colB, blend);
    color = vec4(finalCol, 1.0);
}

```


Shader #527 - goo

File: goo.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv *= iResolution.x / iResolution.y;
    float r = length(uv);
    float theta = atan(uv.y, uv.x);
    float spiralEffect = time_f * 0.2;
    r -= mod(spiralEffect, 4.0);
    theta += spiralEffect;
    vec2 distorted_uv = vec2(cos(theta), sin(theta)) * r;
    distorted_uv = (distorted_uv / (iResolution.x / iResolution.y)) * 0.5 + 0.5;
    vec4 texColor = texture(samp, distorted_uv);
    color = vec4(texColor.rgb, 1.0);
}
```

Shader #534 - gptswirl

File: gptswirl.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = uv - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float swirl = dist * 5.0;
    angle += swirl * sin(time_f) * pingPong(dist + time_f, 1.0);
    uv = center + vec2(cos(angle), sin(angle)) * dist;
    vec4 texColor = texture(samp, uv);
    float colorMod = 0.5 + 0.5 * sin(time_f + dist * 10.0);
    color = texColor * colorMod;
}
```

Shader #535 - gptswirl2

File: gptswirl2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec3 hue2rgb(float h) {
    h = mod(h, 1.0);
    float r = abs(h * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(h * 6.0 - 2.0);
    float b = 2.0 - abs(h * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = uv - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float swirl = dist * 5.0;
    angle += swirl * sin(time_f) * pingPong(dist + time_f, 1.0);
    uv = center + vec2(cos(angle), sin(angle)) * dist;
    vec4 texColor = texture(samp, uv);
    float hue = mod(angle / (2.0 * 3.14159265) + time_f * 0.1, 1.0);
    vec3 rainbow = hue2rgb(hue);
    color = vec4(texColor.rgb * rainbow, 2.0);
}
```

Shader #536 - grad

File: grad.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Convert to integer RGB values (0-255) Perform XOR operation and ensure values stay within 0-255 Convert back to normalized float RGB

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;

float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
```

```

}
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color = ivec3(icolor.rgb * 255.0);
    ivec3 isource = ivec3(source.rgb * 255.0);
    for (int i = 0; i < 3; ++i) {
        int_color[i] = (int_color[i] ^ isource[i]) % 256;
    }
    vec3 result_color = vec3(int_color) / 255.0;
    result_color = max(result_color, vec3(0.1));
    return vec4(result_color, 1.0);
}

void main(void) {
    if(time_f == 0.0) {
        color = vec4(1, 1, 1, 1);
        return;
    }
    vec2 uv = tc;
    vec2 warp = uv + vec2(
        sin(uv.y * 10.0 + time_f) * 0.1,
        sin(uv.x * 10.0 + time_f) * 0.1
    );
    vec3 colorShift = vec3(
        0.5 * sin(time_f * 0.5) + 0.5,
        0.5 * sin(time_f * 0.7 + 2.0) + 0.5,
        0.5 * sin(time_f * 0.3 + 4.0) + 0.5
    );
    float feedback = rand(uv + time_f);
    vec2 feedbackUv = tc;
    float time_t = mod(time_f, 50);
    vec4 texColor = texture(samp, feedbackUv);
    vec3 finalColor = texColor.rgb + colorShift;
    color = vec4(finalColor, texColor.a);
    color = color * (0.5 + 0.5 * sin(texColor * time_t));
    color = xor_RGB(color, texColor);
    color.rgb = clamp(color.rgb, vec3(0.2), vec3(1.0));
    color.a = 1.0;
}

```

Shader #537 - grad2

File: grad2.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}

vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
    }
}

```

```

        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
void main(void) {
    vec2 uv = tc;
    vec2 warp = uv + vec2(
        sin(uv.y * 10.0 + time_f) * 0.1,
        sin(uv.x * 10.0 + time_f) * 0.1
    );
    vec3 colorShift = vec3(
        0.5 * sin(time_f * 0.5) + 0.5,
        0.5 * sin(time_f * 0.7 + 2.0) + 0.5,
        0.5 * sin(time_f * 0.3 + 4.0) + 0.5
    );
    float feedback = rand(uv + time_f);
    vec2 feedbackUv = tc;
    float time_t = mod(time_f, 50);
    vec4 texColor = texture(samp, feedbackUv);
    vec3 finalColor = texColor.rgb + colorShift;
    color = vec4(finalColor, texColor.a);
    color = color * sin(texColor * log(time_t));
    color.a = 1.0;
}

```

Shader #538 - grad3

File: grad3.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
void main(void) {

```

```

    vec2 uv = tc;
    vec2 warp = uv + vec2(
        sin(uv.y * 10.0 + time_f) * 0.1,
        sin(uv.x * 20.0 + time_f) * 0.1
    );
    vec3 colorShift = vec3(
        0.2 * sin(time_f * 0.2) + 0.8,
        0.4 * sin(time_f * 0.4 + 6.0) + 0.5,
        0.8 * sin(time_f * 0.6 + 4.0) + 0.5
    );
    float time_t = mod(time_f, 50);
    float feedback = rand(uv + time_f);
    vec2 feedbackUv = warp;
    vec4 texColor = texture(samp, feedbackUv);
    vec3 finalColor = texColor.rgb + colorShift;
    color = sin(vec4(finalColor, texColor.a) * (time_f * 0.5));
    color.a = 1.0;
}

```

Shader #559 - gsupercool

File: gsupercool.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float rand(vec2 co) {
    return fract(sin(dot(co, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void) {
    vec2 uv = tc;
    float frequency = 30.0;
    float amplitude = 0.05;
    float speed = 5.0;
    float t = time_f * speed;
    float noiseX = rand(uv * frequency + t);
    float noiseY = rand(uv * frequency - t);
    float distortionX = amplitude * pingPong(t + uv.y * frequency + noiseX, 1.0);
    float distortionY = amplitude * pingPong(t + uv.x * frequency + noiseY, 1.0);
    vec2 uv_distorted = uv + vec2(distortionX, distortionY);
    vec2 uv_kaleidoscope = uv_distorted * 2.0 - 1.0;
    uv_kaleidoscope.x *= iResolution.x / iResolution.y;
    float angle = atan(uv_kaleidoscope.y, uv_kaleidoscope.x);
    float radius = length(uv_kaleidoscope) * 1.4142;
    float segments = 8.0;
    angle = mod(angle, 6.28318 / segments);
    angle = abs(angle - 3.14159 / segments);
    uv_kaleidoscope = vec2(cos(angle), sin(angle)) * radius;
}

```

```

uv_kaleidoscope = uv_kaleidoscope * 0.5 + 0.5;
uv_kaleidoscope = clamp(uv_kaleidoscope, 0.0, 1.0);
float time_t = pingPong(time_f * 0.5, 7.0) + 2.0;
float pattern = cos(radius * 10.0 - time_t * 5.0);
vec3 colorShift = vec3(
    0.5 + 0.5 * cos(pattern + time_t + 0.0),
    0.5 + 0.5 * cos(pattern + time_t + 2.094),
    0.5 + 0.5 * cos(pattern + time_t + 4.188)
);
vec4 texColorKaleido = texture(samp, uv_kaleidoscope);
vec4 texColorDistorted = texture(samp, uv_distorted);
vec3 finalColor = texColorKaleido.rgb * colorShift;
vec4 blendedColor = mix(vec4(finalColor, texColorKaleido.a), texColorDistorted, 0.5);
color = sin(blendedColor * time_t);
color.a = 1.0;
}

```

Shader #560 - gsupercool2

File: gsupercool2.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - 2D rotation matrix -
 Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float rand(vec2 co) {
    return fract(sin(dot(co, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void) {
    vec2 uv = tc;
    float angle = atan(uv.y - 0.5, uv.x - 0.5);
    float modulatedTime = pingPong(time_f, 5.0);
    angle += modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle) * (uv.x - 0.5) - sin(angle) * (uv.y - 0.5) + 0.5;
    rotatedTC.y = sin(angle) * (uv.x - 0.5) + cos(angle) * (uv.y - 0.5) + 0.5;
    vec2 flow;
    flow.x = sin((rotatedTC.x + time_f) * 10.0) * 0.02;
    flow.y = cos((rotatedTC.y + time_f) * 10.0) * 0.02;
    vec2 uv_flow = rotatedTC + flow;
    float frequency = 30.0;
    float amplitude = 0.05;
    float speed = 5.0;
    float t = time_f * speed;
    float noiseX = rand(uv_flow * frequency + t);
    float noiseY = rand(uv_flow * frequency - t);
    float distortionX = amplitude * pingPong(t + uv_flow.y * frequency + noiseX, 1.0);
    float distortionY = amplitude * pingPong(t + uv_flow.x * frequency + noiseY, 1.0);
    vec2 uv_distorted = uv_flow + vec2(distortionX, distortionY);
}

```

```

float zoom = sin(time_f * 0.5) * 0.2 + 1.0;
vec2 uv_zoom = (uv_distorted - vec2(0.5)) * zoom + vec2(0.5);
vec4 color_sample = texture(samp, uv_zoom);
vec3 color_effect;
color_effect.r = sin(color_sample.r * 10.0 + time_f);
color_effect.g = sin(color_sample.g * 10.0 + time_f + 2.0);
color_effect.b = sin(color_sample.b * 10.0 + time_f + 4.0);
color = vec4(color_effect, color_sample.a);
color.a = 1.0;
}

```

Shader #578 - huri

File: huri.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec2 vortexDistortion(vec2 uv, float time) {
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
    float distance = length(offset);
    float angle = atan(offset.y, offset.x) + sin(time + distance * 10.0) * 0.5;
    return center + vec2(cos(angle), sin(angle)) * distance;
}
void main(void) {
    vec2 uv = tc;
    uv = vortexDistortion(uv, time_f);
    vec4 texColor = texture(samp, uv);
    color = texColor;
}

```

Shader #579 - huri1

File: huri1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec2 vortexDistortion(vec2 uv, float time) {
    vec2 center = vec2(0.5, 0.5);

```

```

    vec2 offset = uv - center;
    float distance = length(offset);
    float angle = atan(offset.y, offset.x) + sin(time + distance * 10.0) *
0.5;
    return center + vec2(cos(angle), sin(angle)) * distance;
}
vec2 rotateX(vec2 uv, float angle) {
    float cosA = cos(angle);
    float sinA = sin(angle);
    uv -= 0.5;
    uv = vec2(uv.x, cosA * uv.y - sinA * uv.x);
    uv += 0.5;
    return uv;
}
void main(void) {
    vec2 uv = tc;
    uv = rotateX(uv, time_f);
    uv = vortexDistortion(uv, time_f);
    vec4 texColor = texture(samp, uv);
    color = texColor;
}

```

Shader #580 - huri2

File: huri2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
mat3 rotationMatrixX(float angle) {
    return mat3(
        1.0, 0.0, 0.0,
        0.0, cos(angle), -sin(angle),
        0.0, sin(angle), cos(angle)
    );
}
vec2 vortexDistortion(vec2 uv, float time) {
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
    float distance = length(offset);
    float angle = atan(offset.y, offset.x) + sin(time + distance * 10.0) *
0.5;
    return center + vec2(cos(angle), sin(angle)) * distance;
}
void main(void) {
    vec2 uv = tc;
    float angle = time_f * 0.5;
    mat3 rotation = rotationMatrixX(angle);
    uv = (rotation * vec3(uv - 0.5, 1.0)).xy + 0.5;
    uv = vortexDistortion(uv, time_f);
    vec4 texColor = texture(samp, uv);
    color = texColor;
}

```



```
}
```

Shader #581 - huri3

File: huri3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
mat3 rotationMatrixY(float angle) {
    return mat3(
        cos(angle), 0.0, sin(angle),
        0.0,        1.0, 0.0,
        -sin(angle), 0.0, cos(angle)
    );
}
vec2 vortexDistortion(vec2 uv, float time) {
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
    float distance = length(offset);
    float angle = atan(offset.y, offset.x) + sin(time + distance * 10.0) * 0.5;
    return center + vec2(cos(angle), sin(angle)) * distance;
}
void main(void) {
    vec2 uv = tc;
    float angle = time_f * 0.5;
    mat3 rotation = rotationMatrixY(angle);
    uv = (rotation * vec3(uv - 0.5, 1.0)).xy + 0.5;
    uv = vortexDistortion(uv, time_f);
    vec4 texColor = texture(samp, uv);
    color = texColor;
}
```

Shader #584 - hurixyz

File: hurixyz.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
mat3 rotationMatrixX(float angle) {
    return mat3(
```

```

        1.0, 0.0, 0.0,
        0.0, cos(angle), -sin(angle),
        0.0, sin(angle), cos(angle)
    );
}
mat3 rotationMatrixY(float angle) {
    return mat3(
        cos(angle), 0.0, sin(angle),
        0.0,        1.0, 0.0,
        -sin(angle), 0.0, cos(angle)
    );
}
mat2 rotationMatrixZ(float angle) {
    return mat2(
        cos(angle), -sin(angle),
        sin(angle),  cos(angle)
    );
}
vec2 vortexDistortion(vec2 uv, float time) {
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
    float distance = length(offset);
    float angle = atan2(offset.y, offset.x) + sin(time + distance * 10.0) * 0.5;
    return center + vec2(cos(angle), sin(angle)) * distance;
}
void main(void) {
    vec2 uv = tc;
    float phase = mod(time_f, 3.0);
    if (phase < 1.0) {
        float angle = time_f * 2.0;
        mat3 rotation = rotationMatrixX(angle);
        uv = (rotation * vec3(uv - 0.5, 1.0)).xy + 0.5;
    } else if (phase < 2.0) {
        float angle = time_f * 2.0;
        mat3 rotation = rotationMatrixY(angle);
        uv = (rotation * vec3(uv - 0.5, 1.0)).xy + 0.5;
    } else {
        float angle = time_f * 2.0;
        mat2 rotation = rotationMatrixZ(angle);
        uv = (rotation * (uv - 0.5)) + 0.5;
    }
    uv.x = mod(uv.x + sin(time_f) * 0.2, 1.0);
    uv.y = mod(uv.y + cos(time_f) * 0.2, 1.0);
    uv = vortexDistortion(uv, time_f);
    vec4 texColor = texture(samp, uv);
    color = texColor;
}

```

Shader #585 - huriz

File: huriz.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
mat2 rotationMatrixZ(float angle) {
    return mat2(
        cos(angle), -sin(angle),
        sin(angle),  cos(angle)
    );
}
vec2 vortexDistortion(vec2 uv, float time) {
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
    float distance = length(offset);
    float angle = atan(offset.y, offset.x) + sin(time + distance * 10.0) * 0.5;
    return center + vec2(cos(angle), sin(angle)) * distance;
}
void main(void) {
    vec2 uv = tc;
    float angle = time_f * 0.5;
    mat2 rotation = rotationMatrixZ(angle);
    uv = (rotation * (uv - 0.5)) + 0.5;
    uv = vortexDistortion(uv, time_f);
    vec4 texColor = texture(samp, uv);
    color = texColor;
}

```

Shader #589 - ice

File: ice.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
 Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float noise(vec2 p) {
    return fract(sin(dot(p, vec2(12.9898, 78.233))) * 43758.5453);
}
float smoothNoise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    float a = noise(i);
    float b = noise(i + vec2(1.0, 0.0));
    float c = noise(i + vec2(0.0, 1.0));
    float d = noise(i + vec2(1.0, 1.0));
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(a, b, u.x) + (c - a) * u.y * (1.0 - u.x) + (d - b - c + a) * u.x * u.y;
}
float fbm(vec2 p) {
    float total = 0.0;
    float amplitude = 1.0;
    float frequency = 1.0;
    for(int i = 0; i < 5; i++) {
        total += smoothNoise(p * frequency) * amplitude;
    }
}

```

```

        amplitude *= 0.5;
        frequency *= 2.0;
    }
    return total;
}

void main(void) {
    float distortionStrength = 0.02;
    vec2 distortion = vec2(
        fbm(tc * 10.0 + time_f) - 0.5,
        fbm(tc * 10.0 - time_f) - 0.5
    );
    vec2 distortedTC = tc + distortion * distortionStrength;
    color = texture(samp, distortedTC);
}

```

Shader #600 - kscopic

File: kscopic.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float N = 8.0;
    vec2 center = vec2(0.5, 0.5);
    vec2 pos = tc - center;
    float angle = atan(pos.y, pos.x);
    float radius = length(pos);
    float speed = time_f * 0.05;
    angle += speed;
    angle = mod(angle, 6.2831853 / N);
    angle = abs(angle - (3.14159265 / N));
    vec2 kaleidoscopicTC;
    kaleidoscopicTC.x = radius * cos(angle);
    kaleidoscopicTC.y = radius * sin(angle);
    kaleidoscopicTC += center;
    vec2 warpedCoords;
    warpedCoords.x = fract(kaleidoscopicTC.x + time_f * 1.0);
    warpedCoords.y = fract(kaleidoscopicTC.y + time_f * 1.0);
    color = texture(samp, warpedCoords);
}

```

Shader #622 - lucy

File: lucy.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void)
{
    vec2 uv = tc * 2.0 - 1.0;
    float t = time_f * 0.5;
    float r = length(uv);
    float a = atan(uv.y, uv.x);
    float radius = sin(t + r * 5.0) * 0.5 + 0.5;
    float angle = a + t + sin(r * 20.0 + t) * 0.1;
    vec2 distorted_uv = vec2(cos(angle), sin(angle)) * radius + 0.5;
    vec3 col = texture(samp, distorted_uv).rgb;
    color = vec4(col, 1.0);
}

```

Shader #731 - o3 01

File: o3-01.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Convert an HSV color to RGB. Start with the texture coordinate. Center the coordinate system around (0,0)

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
 - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
vec3 hsv2rgb(vec3 c) {
    vec3 rgb = clamp(abs(mod(c.x * 6.0 + vec3(0.0, 4.0, 2.0),
        6.0) - 3.0) - 1.0,
        0.0, 1.0);
    return c.z * mix(vec3(1.0), rgb, c.y);
}
void main(void) {
    vec2 uv = tc;
    uv = uv * 2.0 - 1.0;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    float swirl = sin(radius * 10.0 - time_f * 3.0) * 0.5;
    float mouseFactor = (iMouse.x > 0.0) ? (iMouse.x / iResolution.x) : 1.0;
    swirl *= mouseFactor;
    angle += swirl;
    uv = vec2(cos(angle), sin(angle)) * radius;
    uv = (uv + 1.0) * 0.5;
    vec4 texColor = texture(samp, uv);
    float hue = fract(time_f * 0.1 + radius);
    vec3 hsv = vec3(hue, 1.0, 1.0);
    vec3 psychedelicColor = hsv2rgb(hsv);
    vec3 finalColor = mix(texColor.rgb, psychedelicColor, 0.7);
    color = vec4(finalColor, texColor.a);
}

```

```
}
```

Shader #732 - obj stretch

File: obj_stretch.glsl

Applies a time-animated visual transformation to the input texture. Responds to mouse position and audio input (amplitude/volume) for interactive control. Convert mouse positions to texture space (0-1) Calculate distance from current pixel to click position Calculate falloff using smoothstep for better control

Technique: Hermite smoothstep interpolation - Distance field computation - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
void main(void) {
    vec2 mouseUV = iMouse.xy / iResolution.xy;
    vec2 clickUV = iMouse.zw / iResolution.xy;
    vec2 drag = mouseUV - clickUV;
    float dist = distance(tc, clickUV);
    float radius = uamp;
    float falloff = 1.0 - smoothstep(0.0, radius, dist);
    vec2 displacedTC = tc + drag * falloff * time_f;
    color = texture(samp, displacedTC);
}
```

Shader #733 - obj stretch2

File: obj_stretch2.glsl

Processes the input texture to produce the "obj stretch2" visual effect. Responds to mouse position for interactive control. Convert coordinates to [0,1] range Current mouse position Initial click position

Technique: Hermite smoothstep interpolation - Distance field computation - Driven by: mouse, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {
    vec2 currentPos = iMouse.xy / iResolution;
    vec2 clickPos = iMouse.zw / iResolution;
    if (iMouse.z > 0.0) {
        vec2 drag = currentPos - clickPos;
        float dist = distance(tc, clickPos);
        float radius = 0.3;
    }
}
```

```

        float strength = 0.5;
        float falloff = 1.0 - smoothstep(0.0, radius, dist);
        vec2 deformedUV = tc - (drag * falloff * strength);
        color = texture(samp, deformedUV);
    } else {
        color = texture(samp, tc);
    }
}

```

Shader #736 - ol

File: ol.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control. Time value for animation Screen resolution Parameters to control the glitch effect

Technique: Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float frequency = 0.5;
uniform float strength = 1.0;
void main() {
    vec2 warpedTexCoord = tc;
    float noise = fract(sin(warpedTexCoord.x * frequency + time_f) *
4.0);
    noise += fract(sin(warpedTexCoord.y * frequency * 2.0 + time_f) *
2.0);
    noise *= strength;
    vec2 mousePos = iMouse.xy;
    mousePos.x = sin(mousePos.x * 16.0 + time_f);
    mousePos.y = sin(mousePos.y * 16.0 + time_f);
    warpedTexCoord += noise * mousePos * strength;
    float layer1 = fract(sin(tc.x * 4.0 + time_f) * 2.0);
    float layer2 = fract(sin(tc.y * 4.0 * 2.0 + time_f) * 2.0);
    float layer3 = fract(sin((tc.x + tc.y) * 8.0 + time_f) * 1.0);
    vec2 finalTexCoord = tc + (layer1 + layer2 * mousePos.x + layer3 *
mousePos.y) * strength;
    color = mix(texture(samp, sin(finalTexCoord * time_f)), texture(samp, tc), 0.5);
}

```

Shader #779 - prism glass size

File: prism_glass_size.glsl

Applies an iterative, time-animated effect that evolves the image over time. Spectrum helper for realistic light splitting Equilateral triangle SDF scaled for screen space Normalize coordinates to -1.0 to 1.0, accounting for aspect ratio

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 spectrum(float w) {
    vec3 c;
    if (w < 0.2) c = mix(vec3(1,0,1), vec3(0,0,1), w * 5.0);
    else if (w < 0.4) c = mix(vec3(0,0,1), vec3(0,1,0), (w-0.2) * 5.0);
    else if (w < 0.6) c = mix(vec3(0,1,0), vec3(1,1,0), (w-0.4) * 5.0);
    else if (w < 0.8) c = mix(vec3(1,1,0), vec3(1,0,0), (w-0.6) * 5.0);
    else c = vec3(1,0,0);
    return c;
}
float sdTriangle(vec2 p, float r) {
    const float k = sqrt(3.0);
    p.x = abs(p.x) - r;
    p.y = p.y + r/k;
    if (p.x + k*p.y > 0.0) p = vec2(p.x - k*p.y, -k*p.x - p.y) / 2.0;
    p.x -= clamp(p.x, -2.0*r, 0.0);
    return -length(p) * sign(p.y);
}
void main(void) {
    vec2 uv = tc;
    vec2 p = (tc - 0.5) * 2.0;
    p.x *= iResolution.x / iResolution.y;
    float prismSize = 0.9;
    float dispersion = 0.12;
    int samples = 16;
    float d = sdTriangle(p, prismSize);
    if (d < 0.0) {
        vec3 accumulatedColor = vec3(0.0);
        float totalWeight = 0.0;
        vec2 refractDir = normalize(vec2(p.x, p.y + 0.5));
        for (int i = 0; i < samples; i++) {
            float w = float(i) / float(samples - 1);
            float shift = (w - 0.5) * dispersion;
            vec2 sampleUV = uv + (refractDir * shift * (1.0 - abs(d)));
            vec3 spec = spectrum(w);
            accumulatedColor += texture(samp, clamp(sampleUV, 0.0, 1.0)).rgb * spec;
            totalWeight += 1.0;
        }
        vec3 finalRGB = accumulatedColor / (float(samples) * 0.45);
        finalRGB += vec3(0.02, 0.03, 0.05);
        color = vec4(finalRGB, 1.0);
    } else {
        color = texture(samp, uv);
        float edge = smoothstep(0.02, 0.0, abs(d));
        color.rgb += edge * 0.3;
    }
}

```

Shader #799 - putty

File: putty.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float timeVar = time_f * 0.5;
    vec2 noise = vec2(
        pingPong(tc.x + timeVar, 1.0),
        pingPong(tc.y + timeVar, 1.0)
    );
    float stretchFactorX = 1.0 + 0.3 * sin(time_f + tc.y * 10.0);
    float stretchFactorY = 1.0 + 0.3 * cos(time_f + tc.x * 10.0);
    vec2 distortedUV = vec2(
        tc.x * stretchFactorX + noise.x * 0.1,
        tc.y * stretchFactorY + noise.y * 0.1
    );
    color = texture(samp, distortedUV);
}
```

Shader #801 - radial distortion

File: radial_distortion.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc - 0.5;
    float len = length(uv);
    float time_t = pingPong(time_f, 2.0) + 0.05;
    float radialDistortion = sin(len * 10.0 - time_f) * time_t;
    uv += uv * radialDistortion;
    float warpFactor = sin(time_f) * 0.5 + 0.5;
    vec2 warpedCoords;
    warpedCoords.x = pingPong(uv.x + time_f * 0.1, 1.0);
    warpedCoords.y = pingPong(uv.y + time_f * 0.1, 1.0);
    float timeVar = time_f * 0.5;
    vec2 noise = vec2(
        pingPong(warpedCoords.x + timeVar, 1.0),
        pingPong(warpedCoords.y + timeVar, 1.0)
    );
}
```

```

float stretchFactorX = 1.0 + 0.3 * sin(time_f + warpedCoords.y * 10.0);
float stretchFactorY = 1.0 + 0.3 * cos(time_f + warpedCoords.x * 10.0);
vec2 finalUV = vec2(
    warpedCoords.x * stretchFactorX + noise.x * 0.1,
    warpedCoords.y * stretchFactorY + noise.y * 0.1
);
color = texture(samp, finalUV + 0.5);
}

```

Shader #802 - radial distortion2

File: radial_distortion2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc - 0.5;
    float len = length(uv);
    float time_t = pingPong(time_f, 2.0) + 0.05;
    float radialDistortion = sin(len * 10.0 - time_f) * time_t;
    uv += uv * radialDistortion;
    float warpFactor = sin(time_f) * 0.5 + 0.5;
    vec2 warpedCoords;
    warpedCoords.x = pingPong(uv.x + time_f * 0.1, 1.0);
    warpedCoords.y = pingPong(uv.y + time_f * 0.1, 1.0);
    color = texture(samp, warpedCoords + 0.5);
}

```

Shader #803 - radwarp

File: radwarp.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {

```

```

    vec2 uv = tc - 0.5;
    float len = length(uv);
    float time_t = pingPong(time_f, 2.0) + 0.05;
    float distortion = sin(len * 10.0 - time_f) * time_t;
    uv += uv * distortion;
    color = texture(samp, uv + 0.5);
}

```

Shader #850 - react

File: react.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 uv = tc;
    float amplitude = sin(time_f) * 0.5 + 0.5;
    vec2 distortedUV = uv + vec2(sin(uv.y * 10.0 + time_f) * 0.1, cos(uv.x * 10.0 + time_f) * 0.1) *
    ↪ amplitude;
    color = texture(samp, distortedUV);
}

```

Shader #851 - react10

File: react10.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float amplitude = sin(time_f * 5.0) * 2.0;
    float distFromCenter = abs(tc.y - 0.5);
    vec2 distorted_tc = tc;
    distorted_tc.y += amplitude * (0.5 - distFromCenter) * distFromCenter;
    distorted_tc = clamp(distorted_tc, vec2(0.0), vec2(1.0));
    color = texture(samp, distorted_tc);
}

```

Shader #852 - react11

File: react11.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float amplitude = sin(time_f * 5.0) * 2.0;
    vec2 center = vec2(0.5, 0.5);
    vec2 direction = tc - center;
    float distance = length(direction);
    float angle = atan(direction.y, direction.x);
    float spin = amplitude * distance;
    float newAngle1 = angle + spin;
    float newAngle2 = angle - spin;
    vec2 distorted_tc1 = center + vec2(cos(newAngle1), sin(newAngle1)) * distance;
    vec2 distorted_tc2 = center + vec2(cos(newAngle2), sin(newAngle2)) * distance;
    distorted_tc1 = clamp(distorted_tc1, vec2(0.0), vec2(1.0));
    distorted_tc2 = clamp(distorted_tc2, vec2(0.0), vec2(1.0));
    vec4 color1 = texture(samp, distorted_tc1);
    vec4 color2 = texture(samp, distorted_tc2);
    color = mix(color1, color2, 0.5);
}
```

Shader #921 - set swirl pos mouse

File: set_swirl_pos_mouse.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 ar = vec2(1.0, iResolution.y / iResolution.x);
    vec2 iar = vec2(1.0, iResolution.x / iResolution.y);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 p = (tc - m) * ar;
    float r = length(p);
    float r0 = 0.35;
```

```

float k = 2.5 + 1.5 * sin(time_f * 0.5);
float t = k * (r0 - r);
float c = cos(t), s = sin(t);
vec2 pr = mat2(c, -s, s, c) * p;
vec2 tcoord = m + pr * iar;
vec2 uv = tcoord * 2.0 - 1.0;
uv.y *= iResolution.y / iResolution.x;
float wave = sin(uv.x * 10.0 + time_f * 2.0) * 0.1;
float angle = atan(uv.y + wave, uv.x) + time_f * 2.0;
vec3 rc = rainbow(fract(angle / (2.0 * 3.1415926535)));
vec4 tex = texture(samp, tcoord);
float w = smoothstep(r0, 0.0, r);
vec3 col = mix(tex.rgb, rc, 0.5 * w);
color = vec4(col, tex.a);
}

```

Shader #941 - sine swirl

File: sine_swirl.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    float hue = mod(t, 1.0) * 6.0;
    float c = 1.0;
    float x = 1.0 - abs(mod(hue, 2.0) - 1.0);
    vec3 rgb = (hue < 1.0) ? vec3(c, x, 0.0) :
                (hue < 2.0) ? vec3(x, c, 0.0) :
                (hue < 3.0) ? vec3(0.0, c, x) :
                (hue < 4.0) ? vec3(0.0, x, c) :
                (hue < 5.0) ? vec3(x, 0.0, c) :
                vec3(c, 0.0, x);
    return rgb;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float frequency = 1.0;
    float amplitude = 3.0;
    float spiralTime = sin(time_f * frequency) * amplitude;
    float scale = 1.0;
    float gradient_pos = mod(dist * scale + (angle + spiralTime) / (2.0 * 3.14159), 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec4 ctx = texture(samp, tc);
    float time_t = mod(time_f, 10.0) + 2.0;
    color = mix(sin(ctx * time_t), vec4(color_gradient, 1.0), 0.5);
    color.a = 1.0;
}

```

Shader #949 - smoke

File: smoke.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -

Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float noise(vec2 st) {
    return fract(sin(dot(st.xy, vec2(12.9898,78.233))) * 43758.5453123);
}
void main(void) {
    vec2 uv = tc;
    float time = time_f * 0.1;
    vec2 noiseOffset = vec2(noise(uv + time), noise(uv - time));
    float distortAmount = 0.2 + amp_low * 0.4;
    noiseOffset = (noiseOffset - 0.5) * distortAmount;
    vec2 nuv = uv + noiseOffset;
    vec4 texColor = texture(samp, nuv);
    vec4 smokeColor = mix(texColor, vec4(0.6, 0.6, 0.6, 1.0), 0.2);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    smokeColor.rgb *= 1.0 + _ab * 0.6;
    smokeColor.rgb = mix(smokeColor.rgb, smokeColor.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15,
↵ 1.0 + clamp(amp_high, 0.0, 1.0) * 0.25), _ab);
    color = smokeColor;
}
```

Shader #958 - spaz

File: spaz.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0);
    float distortionX = sin(normCoord.y * 10.0 + time_f * 3.0) * 0.05;
```

```

float distortionY = cos(normCoord.x * 15.0 + time_f * 2.0) * 0.05;
distortionX += sin(time_f * 5.0) * 0.02;
distortionY += cos(time_f * 7.0) * 0.02;
vec2 distortedCoord = normCoord + vec2(distortionX, distortionY);
distortedCoord = (distortedCoord + 1.0) / 2.0;
distortedCoord = clamp(distortedCoord, 0.0, 1.0);
color = texture(samp, distortedCoord);
}

```

Shader #960 - spiral center

File: spiral-center.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
 - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 spiralCenter = vec2(0.0, 0.0);
    vec2 localUV = uv - spiralCenter;
    float d = length(uv);
    float lensStrength = 1.5;
    vec3 normal = normalize(vec3(uv, 1.0 / lensStrength));
    float localD = length(localUV);
    float fisheyeRadius = atan(localD, 1.0);
    vec2 distortedUV = normalize(localUV + 1e-6) * fisheyeRadius;
    float t = time_f * 0.8;
    float pTime = pingPong(time_f * 0.5, 2.0);
    float r_dist = length(distortedUV);
    float angle = atan(distortedUV.y, distortedUV.x);
    float spiral = angle + (log(r_dist + 0.1) * (2.0 + pTime)) - t * 1.5;
    float r = sin(spiral * 3.0 + t);
    float g = sin(spiral * 3.0 + t + 2.094);
    float b = sin(spiral * 3.0 + t + 4.188);
    vec3 spiralCol = vec3(r, g, b) * 0.5 + 0.5;
    vec3 lightDir = normalize(vec3(sin(time_f), cos(time_f), 1.0));
    float diff = max(dot(normal, lightDir), 0.0);
    float spec = pow(max(dot(reflect(-lightDir, normal), vec3(0.0, 0.0, 1.0)), 0.0), 16.0);
    vec3 texColor = texture(samp, tc).rgb;
    float spiralMask = 1.0 - smoothstep(0.12, 0.85, localD);
    vec3 spiralLit = spiralCol * (diff + 0.5) + spec;
    vec3 finalCol = mix(texColor, spiralLit, 0.7 * spiralMask);
    finalCol *= smoothstep(2.0, 0.5, d);
    color = vec4(finalCol, alpha);
}

```

```
}
```

Shader #962 - spiral by time

File: spiral_by_time.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float h1(float n){ return fract(sin(n)*43758.5453123); }
vec2 h2(float n){ return fract(sin(vec2(n,n+1.0))*vec2(43758.5453,22578.1459)); }
void main(void) {
    float rate = 0.7;
    float t = time_f*rate;
    float t0 = floor(t);
    float a = fract(t);
    float w = a*a*(3.0-2.0*a);
    vec2 p0 = vec2(0.15) + h2(t0)*0.7;
    vec2 p1 = vec2(0.15) + h2(t0+1.0)*0.7;
    vec2 center = mix(p0, p1, w);
    vec2 off = tc - center;
    float maxR = length(vec2(0.5));
    float r = length(off);
    float nr = r / maxR + 1e-6;
    float ang = atan(off.y, off.x);
    float distortion = 0.5;
    float dr = nr + distortion*nr*nr;
    dr = clamp(dr, 0.0, 1.0)*maxR;
    float spiral = 3.0;
    ang += spiral*(1.0 - dr/maxR) + pingPong(time_f, 5.0);
    vec2 d = center + dr*vec2(cos(ang), sin(ang));
    vec2 rotc = vec2(0.5);
    vec2 q = d - rotc;
    float rot = 0.35*sin(time_f*0.6);
    float cs = cos(rot), sn = sin(rot);
    vec2 ruv = vec2(cs*q.x - sn*q.y, sn*q.x + cs*q.y) + rotc;
    vec2 warped;
    warped.x = pingPong(ruv.x + time_f*0.1, 1.0);
    warped.y = pingPong(ruv.y + time_f*0.1, 1.0);
    color = texture(samp, warped);
}
```


Shader #992 - sweb

File: sweb.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float radialPattern(vec2 uv, float rings, float spokes) {
    vec2 centered_uv = uv - 0.5;
    float r = length(centered_uv);
    float circle = smoothstep(0.02, 0.03, sin(rings * r * 6.28 + time_f * 0.5));
    float angle = atan(centered_uv.y, centered_uv.x);
    float spoke = smoothstep(0.02, 0.03, sin(spokes * angle * 0.5 + time_f * 0.2));
    return circle + spoke;
}
vec2 warpTextureCoords(vec2 uv, float webPattern) {
    float pingPongOffset = pingPong(time_f * 0.2, 0.5);
    vec2 warpFactor = vec2(1.0 + webPattern * 0.1, 1.0 - webPattern * 0.1);
    vec2 warpedUV = uv * warpFactor;
    warpedUV += 0.02 * vec2(sin(uv.y * 10.0 + pingPongOffset), cos(uv.x * 10.0 + pingPongOffset));
    return warpedUV;
}
void main(void) {
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    float web = radialPattern(uv, 10.0, 12.0);
    vec2 warpedTC = warpTextureCoords(tc, web);
    vec4 textureColor = texture(samp, warpedTC);
    color = textureColor * vec4(1.0, 1.0, 1.0, 1.0) * (0.5 + web * 0.5);
}
```

Shader #997 - swirl cache

File: swirl_cache.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Computes luminance for brightness-based effects. Applies matrix transformations. The default texture coordinate passed from your vertex shader Our five input textures Screen resolution and time

Technique: Multi-tap convolution blur (5 texture samples) - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - RGB <-> HSL color space conversion - Temporal feedback using cached frame buffers - Per-channel UV offset for chromatic aberration -
Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform sampler2D samp1;
```

```

uniform sampler2D samp2;
uniform sampler2D samp3;
uniform sampler2D samp4;
uniform vec2 iResolution;
uniform float time_f;
vec3 hueShift(vec3 color, float shift)
{
    float angle = 6.28318530718 * shift;
    float s = sin(angle);
    float c = cos(angle);
    mat3 toYCbCr = mat3(
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114,
        0.299, 0.587, 0.114
    );
    mat3 adjust = mat3(
        0.701, -0.587, -0.114,
        -0.299, 0.413, -0.114,
        -0.300, -0.588, 0.886
    );
    mat3 rot = mat3(
        c, -s, 0.0,
        s,  c, 0.0,
        0.0, 0.0, 1.0
    );
    return (toYCbCr + adjust * rot) * color;
}
vec2 swirl(vec2 uv, vec2 center, float radius, float angle)
{
    vec2 pos = uv - center;
    float r = length(pos);
    if (r < radius) {
        float percent = (radius - r) / radius;
        float theta = atan(pos.y, pos.x) + angle * percent;
        pos = r * vec2(cos(theta), sin(theta));
    }
    return pos + center;
}
void main(void)
{
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    float radius = 0.5;
    float swirlStrength = 0.75;
    float swirlAngle = swirlStrength * sin(time_f * 0.8);
    float waveX = sin(uv.y * 6.0 + time_f * 2.0) * 0.03;
    float waveY = cos(uv.x * 6.0 + time_f * 2.0) * 0.03;
    vec2 uv0 = swirl(uv + vec2(waveX, waveY), center, radius, swirlAngle);
    vec2 uv1 = swirl(uv + vec2(waveY, waveX), center, radius, swirlAngle * 0.5);
    vec2 uv2 = swirl(uv - vec2(waveX, waveY), center, radius, -swirlAngle);
    vec2 uv3 = swirl(uv + vec2(waveY * 2.0, -waveX * 2.0), center, radius, swirlAngle * 1.5);
    vec2 uv4 = swirl(uv, center, radius, swirlAngle * 2.0);
    vec4 base  = texture(samp, uv0);
    vec4 one   = texture(samp1, uv1);
    vec4 two   = texture(samp2, uv2);
    vec4 three = texture(samp3, uv3);
    vec4 four  = texture(samp4, uv4);
    vec3 color0 = hueShift(base.rgb, 0.00);
    vec3 color1 = hueShift(one.rgb, 0.20);
    vec3 color2 = hueShift(two.rgb, 0.40);
    vec3 color3 = hueShift(three.rgb, 0.60);
    vec3 color4 = hueShift(four.rgb, 0.80);
}

```

```

    vec3 combined = color0 + color1 + color2 + color3 + color4;
    combined /= 5.0;
    color = vec4(combined, 1.0);
}

```

Shader #1011 - thick glass

File: thick_glass.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = gl_FragCoord.xy / iResolution.xy;
    float distortionStrength = 0.2;
    float noiseFactor = sin(normCoord.x * 10.0 + time_f) * cos(normCoord.y * 10.0 + time_f);
    vec2 distortedCoord = normCoord + distortionStrength * vec2(noiseFactor, noiseFactor);
    distortedCoord = clamp(distortedCoord, 0.0, 1.0);
    color = texture(samp, distortedCoord);
}

```

Shader #1021 - tridim

File: tridim.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float t = mod(time_f, 10.0);
    float scale = pow(1.1, t);
    vec2 centeredTC = (tc - 0.5) * scale + 0.5;
    float xDistort = cos(centeredTC.y * 10.0 + time_f) * 0.1 / scale;
    float yDistort = sin(centeredTC.x * 10.0 + time_f) * 0.1 / scale;
    float tanDistortX = tan(centeredTC.x * 5.0 + time_f) * 0.05 / scale;
    float tanDistortY = tan(centeredTC.y * 5.0 + time_f) * 0.05 / scale;
    vec2 distortedTC = centeredTC + vec2(xDistort + tanDistortX, yDistort + tanDistortY);
    distortedTC = fract(distortedTC);
    color = texture(samp, distortedTC);
}

```

Shader #1024 - triwavedistort

File: triwavedistort.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float xDistort = cos(tc.y * 10.0 + time_f) * 0.1;
    float yDistort = sin(tc.x * 10.0 + time_f) * 0.1;
    float tanDistortX = tan(tc.x * 5.0 + time_f) * 0.05;
    float tanDistortY = tan(tc.y * 5.0 + time_f) * 0.05;
    vec2 distortedTC = tc + vec2(xDistort + tanDistortX, yDistort + tanDistortY);
    distortedTC = fract(distortedTC);
    color = texture(samp, distortedTC);
}
```

Shader #1026 - twarp

File: twarp.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float warpX = tan(uv.y * 10.0 + time_f) * 0.1;
    float warpY = tan(uv.x * 10.0 + time_f) * 0.1;
    uv.x += warpX;
    uv.y += warpY;
    color = texture(samp, uv);
}
```

Shader #1027 - twarp2

File: twarp2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
```

```

uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float warpX = tan(uv.y * 10.0 + time_f) * 0.1;
    float warpY = tan(uv.x * 10.0 + time_f) * 0.1;
    uv.x += warpX;
    uv.y += warpY;
    color = texture(samp, sin(uv * time_f));
}

```

Shader #1029 - twist

File: twist.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Linear mix/lerp blending - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float rippleSpeed = 5.0;
    float rippleAmplitude = 0.03;
    float rippleWavelength = 10.0;
    float twistStrength = 1.0;
    float radius = length(tc - vec2(0.5, 0.5));
    float ripple = sin(tc.x * rippleWavelength + time_f * rippleSpeed) * rippleAmplitude;
    ripple += sin(tc.y * rippleWavelength + time_f * rippleSpeed) * rippleAmplitude;
    vec2 rippleTC = tc + vec2(ripple, ripple);
    float angle = twistStrength * (radius - 1.0) + time_f;
    float cosA = cos(angle);
    float sinA = sin(angle);
    mat2 rotationMatrix = mat2(cosA, -sinA, sinA, cosA);
    vec2 twistedTC = (rotationMatrix * (tc - vec2(0.5, 0.5))) + vec2(0.5, 0.5);
    vec4 originalColor = texture(samp, tc);
    vec4 twistedRippleColor = texture(samp, mix(rippleTC, twistedTC, 0.5));
    color = mix(originalColor, twistedRippleColor, 0.5);
}

```

Shader #1030 - twist directions

File: twist_directions.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;

```

```

vec2 twist(vec2 coord, float twistAmount, float time, float direction) {
    float r = length(coord);
    float theta = atan(coord.y, coord.x);
    theta += (1.0 - r) * twistAmount * sin(time * direction);
    return vec2(cos(theta), sin(theta)) * r;
}

void main(void) {
    vec2 normCoord = gl_FragCoord.xy / iResolution.xy;
    vec2 centeredCoord = normCoord - vec2(0.5, 0.5);
    centeredCoord.x *= iResolution.x / iResolution.y;
    vec2 twistClockwise = twist(centeredCoord, 15.0, time_f, 1.0);
    vec2 twistCounterClockwise = twist(centeredCoord, 15.0, time_f, -1.0);
    twistClockwise.x *= iResolution.y / iResolution.x;
    twistCounterClockwise.x *= iResolution.y / iResolution.x;
    twistClockwise += vec2(0.5, 0.5);
    twistCounterClockwise += vec2(0.5, 0.5);
    vec4 texColorClockwise = texture(samp, twistClockwise);
    vec4 texColorCounterClockwise = texture(samp, twistCounterClockwise);
    color = mix(texColorClockwise, texColorCounterClockwise, 0.5);
}

```

Shader #1031 - twist full

File: twist_full.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. color = mix(originalColor, twistedRippleColor, 0.5);

Technique: 2D rotation matrix - Linear mix/lerp blending - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float rippleSpeed = 5.0;
    float rippleAmplitude = 0.03;
    float rippleWavelength = 10.0;
    float twistStrength = 1.0;
    float radius = length(tc - vec2(0.5, 0.5));
    float ripple = sin(tc.x * rippleWavelength + time_f * rippleSpeed) * rippleAmplitude;
    ripple += sin(tc.y * rippleWavelength + time_f * rippleSpeed) * rippleAmplitude;
    vec2 rippleTC = tc + vec2(ripple, ripple);
    float angle = twistStrength * (radius - 1.0) + time_f;
    float cosA = cos(angle);
    float sinA = sin(angle);
    mat2 rotationMatrix = mat2(cosA, -sinA, sinA, cosA);
    vec2 twistedTC = (rotationMatrix * (tc - vec2(0.5, 0.5))) + vec2(0.5, 0.5);
    vec4 originalColor = texture(samp, tc);
    vec4 twistedRippleColor = texture(samp, mix(rippleTC, twistedTC, 0.5));
    color = twistedRippleColor;
}

```

Shader #1032 - twist gpt

File: twist_gpt.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position and audio input (amplitude/volume) for interactive control. Applies matrix transformations.

Technique: Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float pp(float x, float l){float m=mod(x,l*2.0);return abs(m-l);}
float s2(float x){return x*x*(3.0-2.0*x);}
void main(void){
    vec2 uv = tc;
    float a = clamp(amp,0.0,1.0);
    float ua = clamp(uamp,0.0,1.0);
    vec2 c;
    if(iMouse.z>0.5) c = iMouse.xy / iResolution;
    else{
        float t = time_f*0.3;
        c = vec2(0.5) + 0.25*vec2(cos(t*0.9), sin(t*1.1))*0.6;
    }
    float r = length(uv - c);
    float rippleSpeed = mix(4.0,10.0,ua);
    float rippleAmp = mix(0.01,0.06,a);
    float rippleWave = mix(8.0,18.0,ua);
    float ripple = sin(uv.x*rippleWave + time_f*rippleSpeed) * rippleAmp;
    ripple += sin(uv.y*rippleWave*0.9 + time_f*rippleSpeed*1.07) * rippleAmp;
    vec2 uv_rip = uv + vec2(ripple);
    float twist = mix(0.8,2.2,a) * (r - 0.8) + time_f*(0.3+ua*0.7);
    float ca = cos(twist), sa = sin(twist);
    mat2 rot = mat2(ca,-sa,sa,ca);
    vec2 uv_twist = (rot*(uv - c)) + c;
    float w = s2(clamp(1.0 - r*2.0, 0.0, 1.0));
    float k = mix(0.35,0.75,ua);
    vec2 uv_mix = mix(uv_rip, uv_twist, k*w + (1.0-k)*0.5);
    float zoom = 1.0 + 0.02*sin(time_f*0.8 + ua*2.0);
    vec2 zc = (uv_mix - c)/zoom + c;
    vec4 col0 = texture(samp, uv);
    vec4 col1 = texture(samp, zc);
    color = mix(col0, col1, 0.7);
}
```

Shader #1033 - twist gpt full

File: twist_gpt_full.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position and audio input (amplitude/volume) for interactive control. Applies matrix transformations.

Technique: Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
```

```

out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float pp(float x, float l){float m=mod(x,l*2.0);return abs(m-l);}
float s2(float x){return x*x*(3.0-2.0*x);}
void main(void){
    vec2 uv = tc;
    float a = clamp(amp,0.0,1.0);
    float ua = clamp(uamp,0.0,1.0);
    vec2 c;
    if(iMouse.z>0.5) c = iMouse.xy / iResolution;
    else{
        float t = time_f*0.3;
        c = vec2(0.5) + 0.25*vec2(cos(t*0.9), sin(t*1.1))*0.6;
    }
    float r = length(uv - c);
    float rippleSpeed = mix(4.0,10.0,ua);
    float rippleAmp = mix(0.01,0.06,a);
    float rippleWave = mix(8.0,18.0,ua);
    float ripple = sin(uv.x*rippleWave + time_f*rippleSpeed) * rippleAmp;
    ripple += sin(uv.y*rippleWave*0.9 + time_f*rippleSpeed*1.07) * rippleAmp;
    vec2 uv_rip = uv + vec2(ripple);
    float twist = mix(0.8,2.2,a) * (r - 0.8) + time_f*(0.3+ua*0.7);
    float ca = cos(twist), sa = sin(twist);
    mat2 rot = mat2(ca,-sa,sa,ca);
    vec2 uv_twist = (rot*(uv - c)) + c;
    float w = s2(clamp(1.0 - r*2.0, 0.0, 1.0));
    float k = mix(0.35,0.75,ua);
    vec2 uv_mix = mix(uv_rip, uv_twist, k*w + (1.0-k)*0.5);
    float zoom = 1.0 + 0.02*sin(time_f*0.8 + ua*2.0);
    vec2 zc = (uv_mix - c)/zoom + c;
    vec4 col0 = texture(samp, uv);
    vec4 col1 = texture(samp, zc);
    color = col1;
}

```

Shader #1034 - twistex

File: twistex.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. color = mix(originalColor, twistedRippleColor, 0.5);

Technique: 2D rotation matrix - Linear mix/lerp blending - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void) {
    float rippleSpeed = 5.0;

```



```

float rippleAmplitude = 0.03;
float rippleWavelength = 10.0;
float twistStrength = 1.0;
float radius = length(tc - vec2(0.5, 0.5));
float ripple = sin(tc.x * rippleWavelength + time_f * rippleSpeed) * rippleAmplitude;
ripple += sin(tc.y * rippleWavelength + time_f * rippleSpeed) * rippleAmplitude;
vec2 rippleTC = tc + vec2(ripple, ripple);
float angle = twistStrength * (radius - 1.0) + time_f;
float cosA = cos(angle);
float sinA = sin(angle);
mat2 rotationMatrix = mat2(cosA, -sinA, sinA, cosA);
vec2 twistedTC = (rotationMatrix * (tc - vec2(0.5, 0.5))) + vec2(0.5, 0.5);
vec4 originalColor = texture(samp, tc);
vec4 twistedRippleColor = texture(samp, mix(sin(rippleTC * pingPong(time_f, 10.0)), twistedTC, 0.5))
↪ ;
color = twistedRippleColor;
}

```

Shader #1041 - warp atan

File: warp-atan.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float angle = atan(tc.y - 0.5, tc.x - 0.5);
    float modulatedTime = pingPong(time_f, 5.0);
    angle += modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle) * (tc.x - 0.5) - sin(angle) * (tc.y - 0.5) + 0.5;
    rotatedTC.y = sin(angle) * (tc.x - 0.5) + cos(angle) * (tc.y - 0.5) + 0.5;
    float warpFactor = sin(time_f) * 0.5 + 0.5;
    vec2 warpedCoords;
    warpedCoords.x = pingPong(rotatedTC.x + time_f * 0.1, 1.0);
    warpedCoords.y = pingPong(rotatedTC.y + time_f * 0.1, 1.0);
    color = texture(samp, warpedCoords);
}

```

Shader #1042 - warp ppong time

File: warp-ppong-time.glsl

Renders a fractal pattern using iterative computation (many iterations).

Technique: Iterative fractal computation with complex-plane mapping - Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float len){
    float m = mod(x, len*2.0);
    return m <= len ? m : len*2.0 - m;
}
void main(void){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (tc - 0.5) * ar;
    float base = 1.65;
    float period = log(base);
    float t = time_f * 0.45;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float k = fract((log(r) - t) / period);
    float rw = exp(k * period);
    a += t * 0.4;
    vec2 z = vec2(cos(a), sin(a)) * rw;
    float swirl = sin(time_f * 0.6 + r * 5.0) * 0.1;
    z += swirl * normalize(p);
    vec2 uv = z / ar + 0.5;
    uv = fract(uv);
    vec4 tex = texture(samp, uv);
    color = tex;
}

```

Shader #1043 - warp ppong

File: warp_ppong.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float warpFactor = sin(time_f) * 0.5 + 0.5;
    vec2 warpedCoords;
    warpedCoords.x = pingPong(tc.x + time_f * 0.1, 1.0);
    warpedCoords.y = pingPong(tc.y + time_f * 0.1, 1.0);
    color = texture(samp, warpedCoords);
}

```

Shader #1046 - warp tunnel

File: warp_tunnel.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright.

Technique: Polar coordinate transformation (atan2 + length) - Radial vignette falloff - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {
    ivec4 isource = ivec4(isourcex * 255);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
void main() {
    vec2 uv = (tc - 0.5) * (iResolution.x / iResolution.y, 1.0);
    float radius = length(uv);
    float warp = radius * 10.0 - time_f * 5.0;
    float angle = atan(uv.y, uv.x) + time_f;
    vec3 colorMix = 0.5 + 0.5 * cos(warp + vec3(0, 2, 4));
    float vignette = 0.3 / radius;
    color = vec4(colorMix * vignette, 1.0);
    color = xor_RGB(color, texture(samp, tc));
}
```

Shader #1065 - wspiral

File: wspiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float speed = 1.5;
    float amplitude = 0.02;
    float frequency = 6.0;
    float twist = 8.0;
    vec2 vortexCenter = vec2(-0.5, 1.5);
    vec2 adjustedTC = tc - vortexCenter;
```

```

float angle = atan(adjustedTC.y, adjustedTC.x);
float radius = length(adjustedTC);
float vortex = sin(twist * angle + radius * frequency - time_f * speed) * amplitude * radius;
vec2 vortexDistortedTC = tc + normalize(adjustedTC) * vortex;
float whirlSpeed = 2.0;
float whirlFrequency = 8.0;
float whirlAmplitude = 0.03;
float whirlScale = 0.5 + 0.5 * sin(time_f * whirlSpeed);
vec2 whirlCenter = vec2(0.5, 0.5);
vec2 whirlAdjustedTC = (tc - whirlCenter) * whirlScale;
float whirlAngle = atan(whirlAdjustedTC.y, whirlAdjustedTC.x);
float whirlRadius = length(whirlAdjustedTC);
float whirlDistortion = sin(whirlFrequency * whirlAngle + whirlRadius * whirlFrequency - time_f *
    ↪ whirlSpeed) * whirlAmplitude;
vec2 whirlDistortedTC = tc + normalize(whirlAdjustedTC) * whirlDistortion;
vec4 originalColor = texture(samp, tc);
vec4 vortexColor = texture(samp, vortexDistortedTC);
vec4 whirlColor = texture(samp, whirlDistortedTC);
float lightEffect = sin(whirlRadius * 10.0 - time_f * 5.0) * 0.5 + 0.5;
vec4 lightColor = vec4(1.0, 1.0, 1.0, 1.0) * lightEffect;
color = mix(originalColor, mix(vortexColor, whirlColor, 0.5), 0.6 + 0.4 * sin(time_f));
color += lightColor * (1.0 - lightEffect) * 0.3;
}

```

Mirror & Symmetry (93 shaders)

Shader #231 - dmd gpt

File: dmd_gpt.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Driven by: resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
float diamond(vec2 p){
    p = abs(p);
    float d = p.x + p.y;
    return 1.0 - smoothstep(0.48, 0.5, d);
}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 uv = tc;
    vec2 uva = uv * ar;
    float tiles = 8.0;
    vec2 gv = uva * tiles;
    vec2 f = fract(gv) - 0.5;
    float mask = diamond(f);
    vec2 mirrored = abs(fract(uv * ar) - 0.5) + 0.5;
    vec2 tuv = mirrored / ar;
    vec4 c1 = texture(samp, tuv);
    vec4 c2 = texture(samp, tuv * 0.5);
    vec4 c3 = texture(samp, tuv * 0.25);
    vec4 c4 = texture(samp, tuv * 0.125);
}

```

```

    vec4 mixcol = c1 * 0.4 + c2 * 0.3 + c3 * 0.2 + c4 * 0.1;
    mixcol.rgb *= mask;
    color = vec4(mixcol.rgb, 1.0);
}

```

Shader #236 - drain mirror

File: drain_mirror.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    float loopDuration = 25.0;
    float currentTime = mod(time_f, loopDuration);
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    normCoord.x = abs(normCoord.x);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralSpeed = 5.0;
    float inwardSpeed = currentTime / loopDuration;
    angle += (1.0 - smoothstep(0.0, 8.0, dist)) * currentTime * spiralSpeed;
    dist *= 1.0 - inwardSpeed;
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * tan(dist);
    spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
    color = texture(samp, spiralCoord);
}

```

Shader #237 - drain mirror amp

File: drain_mirror_amp.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to audio input (amplitude/volume) for interactive control. Normalize texture coordinates to [-1, 1] and adjust for aspect ratio Make the spiral symmetric horizontally Calculate distance and angle for polar coordinates

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Driven by: time, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float uamp;
uniform float amp;
void main(void) {

```

```

float loopDuration = 100.0;
float currentTime = mod(amp, loopDuration);
vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
normCoord.x = abs(normCoord.x);
float dist = length(normCoord);
float angle = atan(normCoord.y, normCoord.x);
float spiralSpeed = 5.0 + uamp * 10.0;
float inwardSpeed = currentTime / loopDuration;
angle += (1.0 - smoothstep(0.0, 8.0, dist)) * currentTime * spiralSpeed;
dist *= 1.0 - inwardSpeed * (1.0 + uamp * 0.5);
vec2 spiralCoord = vec2(cos(angle), sin(angle)) * tan(dist);
spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
color = texture(samp, spiralCoord);
}

```

Shader #238 - drain mirror top

File: drain_mirror_top.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
 - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    float loopDuration = 25.0;
    float currentTime = mod(time_f, loopDuration);
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    normCoord.y = abs(normCoord.y);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralSpeed = 5.0;
    float inwardSpeed = currentTime / loopDuration;
    angle += (1.0 - smoothstep(0.0, 8.0, dist)) * currentTime * spiralSpeed;
    dist *= 1.0 - inwardSpeed;
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * tan(dist);
    spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
    color = texture(samp, spiralCoord);
}

```

Shader #279 - echo mirror 1

File: echo_mirror_1.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```

#version 330
in vec2 tc;

```

```

out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec2 tc1 = tc;
    vec2 tc2 = tc;
    tc1[0] = 1.0-tc1[0];
    tc2[1] = 1.0-tc2[1];
    vec4 color2 = texture(samp, tc/2);
    vec4 color3 = texture(samp, tc1);
    vec4 color4 = texture(samp, tc2);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```

Shader #280 - echo mirror 2

File: echo_mirror_2.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec2 tc1 = tc;
    vec2 tc2 = tc;
    tc1[0] = 1.0-tc1[0];

```

```

    tc1[1] = 1.0-tc1[1];
    tc2[1] = 1.0-tc2[1];
    vec4 color2 = texture(samp, tc/2);
    vec4 color3 = texture(samp, tc1);
    vec4 color4 = texture(samp, tc2);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```

Shader #281 - echo mirror 3

File: echo_mirror_3.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec2 tc1 = tc;
    vec2 tc2 = tc;
    tc1[0] = 1.0-tc1[0];
    tc2[0] = 1.0-tc2[0];
    tc2[1] = 1.0-tc2[1];
    vec4 color2 = texture(samp, tc/2);
    vec4 color3 = texture(samp, tc1);
    vec4 color4 = texture(samp, tc2);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```

Shader #320 - Electric Fold

File: Electric_Fold.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Cosine Palette for the "Electric" glow Combined FBM: 'ridges' true = Electricity, 'ridges' false = Soft Clouds Initial Rotation from controls

Technique: Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;

const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;
const float iSaturation = 1.0;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;

vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}

vec3 rotateHue(vec3 col, float angle) {
    float U = cos(angle); float W = sin(angle);
    mat3 R = mat3(
        0.299+0.701*U+0.168*W, 0.587-0.587*U+0.330*W, 0.114-0.114*U-0.497*W,
        0.299-0.299*U-0.328*W, 0.587+0.413*U+0.035*W, 0.114-0.114*U+0.292*W,
        0.299-0.300*U+1.250*W, 0.587-0.588*U-1.050*W, 0.114+0.886*U-0.203*W
    );
    return clamp(R * col, 0.0, 1.0);
}

vec3 applyColorAdjustments(vec3 col) {
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    col = rotateHue(col, iHueShift);
    return clamp(col, 0.0, 1.0);
}

vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d + iHueShift));
}

vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}

vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}

vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
```

```

    return textureLod(tex, sampleUV, lod);
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}
float fbm(vec2 p, bool ridges) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        if (ridges) n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.1;
        a *= 0.55;
    }
    return v;
}
vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}
vec3 sampleMerged(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.001);
    p /= zoom;
    float f1 = fbm(p * 2.0 + t * 0.2, false);
    float f2 = fbm(p * 2.5 - t * 0.3, false);
    float rLen = length(p);
    float ang = atan(p.y, p.x);
    ang += (f1 * 3.0 + f2) * strength * 0.5;
    p = vec2(cos(ang), sin(ang)) * rLen;
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + t * 0.1, false);
    q.y = fbm(p + vec2(5.2, 1.3) - t * 0.1, false);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0 * q + vec2(t * 0.2, 9.2), true);
    r.y = fbm(p + 4.0 * q + vec2(8.3, 2.8), false);
    p += r * (0.15 * strength);
    float slices = 8.0 + 4.0 * sin(t * 0.2);
    p = kaleido(p, slices);
    int iterations = 4 + int(iAmplitude * 1.5);
    float scale = 1.2 + (iFrequency * 0.2);
    float shift = 0.1 * strength;
    float rotAng = t * 0.2 + (f1 * 0.5);
    for(int i = 0; i < iterations; i++) {

```

```

        p = abs(p);
        p -= shift;
        p *= scale;
        p = rotate2D(p, rotAng + float(i)*0.4);
    }
    vec2 finalUV = p * 0.5 + center;
    vec2 chroma = vec2(0.005 * strength * f2, 0.0);
    float red = mxTexture(samp, finalUV + chroma).r;
    float grn = mxTexture(samp, finalUV).g;
    float blu = mxTexture(samp, finalUV - chroma).b;
    vec3 texCol = vec3(red, grn, blu);
    float energy = (length(r) * 0.5 + f1 * 0.5);
    float glow = pow(energy, 3.0) * iContrast;
    vec3 pal = palette(length(p) + t * 0.5);
    vec3 finalCol = mix(texCol, pal, 0.3 * iSaturation);
    finalCol += pal * glow * strength;
    return finalCol;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float freqControl = clamp(iFrequency, 0.0, 2.0);
    float t = time_f * (0.2 + freqControl * 0.3);
    float strength = 0.5 + (ampControl * 1.0);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec3 colA = sampleMerged(uv, t, strength, center, iResolution);
    color.rgb = applyColorAdjustments(colA);
    color.a = 1.0;
}

```

Shader #321 - Electric Fold Texture

File: Electric_Fold_Texture.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Cosine Palette for the "Electric" glow Initial Rotation Chromatic Aberration on Texture

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;
const float iSaturation = 1.0;
const float iHueShift = 0.0;
const float iZoom = 1.0;

```

```

const float iRotation    = 0.0;
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 rotateHue(vec3 col, float angle) {
    float U = cos(angle); float W = sin(angle);
    mat3 R = mat3(
        0.299+0.701*U+0.168*W, 0.587-0.587*U+0.330*W, 0.114-0.114*U-0.497*W,
        0.299-0.299*U-0.328*W, 0.587+0.413*U+0.035*W, 0.114-0.114*U+0.292*W,
        0.299-0.300*U+1.250*W, 0.587-0.588*U-1.050*W, 0.114+0.886*U-0.203*W
    );
    return clamp(R * col, 0.0, 1.0);
}
vec3 applyColorAdjustments(vec3 col) {
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    col = rotateHue(col, iHueShift);
    return clamp(col, 0.0, 1.0);
}
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d + iHueShift));
}
vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
    return textureLod(tex, sampleUV, lod);
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}
float fbm(vec2 p, bool ridges) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {

```

```

        float n = noise(p);
        if (ridges) n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.1;
        a *= 0.55;
    }
    return v;
}

vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}

vec3 sampleMerged(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.001);
    p /= zoom;
    float f1 = fbm(p * 2.0 + t * 0.2, false);
    float f2 = fbm(p * 2.5 - t * 0.3, false);
    float rLen = length(p);
    float ang = atan(p.y, p.x);
    ang += (f1 * 3.0 + f2) * strength * 0.5;
    p = vec2(cos(ang), sin(ang)) * rLen;
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + t * 0.1, false);
    q.y = fbm(p + vec2(5.2, 1.3) - t * 0.1, false);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0 * q + vec2(t * 0.2, 9.2), true);
    r.y = fbm(p + 4.0 * q + vec2(8.3, 2.8), false);
    p += r * (0.15 * strength);
    float slices = 8.0 + 4.0 * sin(t * 0.2);
    p = kaleido(p, slices);
    int iterations = 4 + int(iAmplitude * 1.5);
    float scale = 1.2 + (iFrequency * 0.2);
    float shift = 0.1 * strength;
    float rotAng = t * 0.2 + (f1 * 0.5);
    for(int i = 0; i < iterations; i++) {
        p = abs(p);
        p -= shift;
        p *= scale;
        p = rotate2D(p, rotAng + float(i)*0.4);
    }
    vec2 finalUV = p * 0.5 + center;
    vec2 chroma = vec2(0.005 * strength * f2, 0.0);
    float red = mxTexture(samp, finalUV + chroma).r;
    float grn = mxTexture(samp, finalUV).g;
    float blu = mxTexture(samp, finalUV - chroma).b;
    vec3 texCol = vec3(red, grn, blu);
    float energy = (length(r) * 0.5 + f1 * 0.5);
    float glow = pow(energy, 4.0) * iContrast;
    vec3 pal = palette(length(p) + t * 0.5);
    vec3 finalCol = texCol + (pal * glow * strength);
    return finalCol;
}

void main() {
    vec2 uv = tc;

```

```

    uv = wrapUV(uv);
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float freqControl = clamp(iFrequency, 0.0, 2.0);
    float t = time_f * (0.2 + freqControl * 0.3);
    float strength = 0.5 + (ampControl * 1.0);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec3 col = sampleMerged(uv, t, strength, center, iResolution);
    color.rgb = applyColorAdjustments(col);
    color.a = 1.0;
}

```

Shader #322 - ElectricMirror

File: ElectricMirror.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position and audio input (amplitude/volume) for interactive control. Generates "waves of electricity" Create jagged noise based on angle and time The "Wave" structure

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
float hash12(vec2 p) {
    vec3 p3 = fract(vec3(p.xy * .1031);
    p3 += dot(p3, p3.yzx + 33.33);
    return fract((p3.x + p3.y) * p3.z);
}

```

```

float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    f = f * f * (3.0 - 2.0 * f);
    float res = mix(
        mix(hash12(i), hash12(i + vec2(1.0, 0.0)), f.x),
        mix(hash12(i + vec2(0.0, 1.0)), hash12(i + vec2(1.0, 1.0)), f.x), f.y);
    return res;
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * 0.35 + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 electricLayer(vec2 uv, float t, float intensity) {
    vec2 p = uv - 0.5;
    p.x *= iResolution.x / iResolution.y;
    float r = length(p);
    float a = atan(p.y, p.x);
    float n = noise(vec2(a * 8.0 - t * 2.0, r * 3.0 - t * 4.0));
    n += 0.5 * noise(vec2(a * 20.0 + t * 5.0, t * 8.0));
    float beam = sin(a * 12.0 + n * 3.0 - t * 3.0);
    float bolt = 0.02 / (abs(beam) + 0.05);
    float ring = 0.01 / (abs(sin(r * 30.0 - t * 10.0 + n)) + 0.05);
    float mask = smoothstep(0.0, 0.2, r) * (1.0 - smoothstep(0.5, 0.8, r));
    float finalElec = (bolt + ring * 0.5) * mask * intensity;
}

```

```

    vec3 eColor = vec3(0.4, 0.9, 1.0) * finalElec;
    eColor += vec3(0.8, 0.2, 1.0) * finalElec * 0.5;
    return eColor;
}

vec3 warpTexture(vec2 baseUV, float seg, float zoom, float t, vec2 m, float aspect, float aMix, float sr
    ↪ , float spinAngle){
    vec2 ar = vec2(aspect, 1.0);
    vec2 kUV = reflectUV(baseUV, seg, m, aspect);
    kUV = fractalFold(kUV, zoom, t, m, aspect);
    kUV = rotateUV(kUV, spinAngle, m, aspect);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    ripple = sin(ripple * pingPong(time_f * PI, 3.0));
    vec2 nDir = normalize(dir + 1e-5);
    vec2 uA = kUV + nDir * ripple;
    vec2 uB = mix(baseUV, kUV, 0.88);
    vec2 uC = mix(baseUV, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
    vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
    vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
    vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
    float ampNorm = clamp(aMix * 0.12, 0.0, 1.5);
    float hueBase = fract(t * 0.06 + r * 0.22 + ampNorm * 0.3);
    float sat = 0.75 + 0.20 * sin(t * 0.4 + sr * 0.7 + ampNorm * 0.4);
    float val = 0.70 + 0.35 * sin(t * 0.55 + r * 1.9 + ampNorm * 0.5);
    vec3 tint = hsv2rgb(vec3(hueBase, sat, val));
    vec3 neon = neonPalette(t + r * 2.0 + ampNorm);
    float mix1 = 0.5 + 0.5 * sin(t * 0.6 + ampNorm * 0.3);
    float mix2 = 0.5 + 0.5 * cos(t * 0.45 + ampNorm * 0.5);
    vec3 warpCol = mix(t1.rgb, t2.rgb, mix1);
    warpCol = mix(warpCol, t3.rgb, mix2 * 0.7);
    warpCol *= tint * neon;
    warpCol *= 1.1 + 0.7 * clamp(aMix * 0.1, 0.0, 1.0);
    vec3 baseTex = textureGrad(samp, baseUV, dFdx(baseUV), dFdy(baseUV)).rgb;
    float warpMix = 0.55 + 0.35 * clamp(aMix * 0.08, 0.0, 1.0);
    vec3 combined = mix(baseTex, warpCol * 1.9, warpMix);
    vec3 bloom = combined * combined * 0.12 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.08;
    combined += bloom;
    return combined;
}

vec3 applyBitwiseBlend(vec3 currentCol, vec3 avgCol) {
    ivec3 curI = ivec3(currentCol * 255.0);
    ivec3 avgI = ivec3(avgCol * 255.0);
    ivec3 sumI = avgI;
    ivec3 xorI = curI ^ (ivec3(1) + sumI);
    vec3 xorF = vec3(xorI) / 255.0;
    vec3 avgF = vec3(avgI) / 255.0;
    vec3 newCol = mix(xorF, avgF, 0.5);
    float boost = 1.8;
    vec3 midpoint = vec3(0.5);
    newCol = midpoint + (newCol - midpoint) * boost;
    return clamp(newCol, 0.0, 1.0);
}

void main(void){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime * (PI / 2.0);
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float ampNorm = clamp(aMix * 0.12, 0.0, 1.5);

```



```

float osc = 0.5 + 0.5 * sin(t * 0.35 + 0.2 * sr + ampNorm * 0.2);
float seg = 4.0 + 2.0 * sin(t * 0.25) + 2.5 * clamp(ampNorm, 0.0, 2.0);
float zoom = mix(1.1, 2.3 + ampNorm * 0.4, osc);
float spinAngle = osc * 2.0 * PI * (0.6 + 0.4 * clamp(ampNorm, 0.0, 1.5));
vec2 centerPx = iResolution * 0.5;
vec2 texCoordPx = tc * iResolution;
vec2 deltaPx = texCoordPx - centerPx;
float dist = length(deltaPx);
float radius = 0.45 + 0.3 * clamp(ampNorm * 0.7, 0.0, 1.0);
float maxRadius = min(iResolution.x, iResolution.y) * radius;
float scaleFactor = 1.0 - pow(clamp(dist / maxRadius, 0.0, 1.0), 2.0);
scaleFactor *= 1.0 + 0.9 * clamp(ampNorm, 0.0, 1.5);
scaleFactor = clamp(scaleFactor, 0.0, 1.9);
vec2 dirN = dist > 0.0 ? deltaPx / dist : vec2(0.0);
float offsetBase = mix(0.008, 0.03, clamp(ampNorm * 0.6, 0.0, 1.0));
float offsetR = offsetBase;
float offsetG = 0.0;
float offsetB = -offsetBase;
vec3 colBase = warpTexture(tc, seg, zoom, t, m, aspect, aMix, sr, spinAngle);
vec3 colBubble = colBase;
if(dist < maxRadius){
    vec2 texCoordR = centerPx + deltaPx * scaleFactor + dirN * offsetR * maxRadius;
    vec2 texCoordG = centerPx + deltaPx * scaleFactor + dirN * offsetG * maxRadius;
    vec2 texCoordB = centerPx + deltaPx * scaleFactor + dirN * offsetB * maxRadius;
    vec2 uvR = clamp(texCoordR / iResolution, 0.0, 1.0);
    vec2 uvG = clamp(texCoordG / iResolution, 0.0, 1.0);
    vec2 uvB = clamp(texCoordB / iResolution, 0.0, 1.0);
    vec3 colR = warpTexture(uvR, seg, zoom, t, m, aspect, aMix, sr, spinAngle);
    vec3 colG = warpTexture(uvG, seg, zoom, t, m, aspect, aMix, sr, spinAngle);
    vec3 colB = warpTexture(uvB, seg, zoom, t, m, aspect, aMix, sr, spinAngle);
    vec3 rgbSplit = vec3(colR.r, colG.g, colB.b);
    float mask = smoothstep(maxRadius, maxRadius * 0.65, dist);
    colBubble = mix(colBase, rgbSplit, mask);
}
vec3 baseVideo = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
float globalMix = 0.50 + 0.35 * clamp(ampNorm * 0.5, 0.0, 1.0);
vec3 combined = mix(baseVideo, colBubble, globalMix);
float electricIntensity = 1.0 + aMix * 2.0;
vec3 electric = electricLayer(tc, t, electricIntensity);
combined += electric;
combined = combined / (1.0 + combined);
combined = clamp(combined, vec3(0.0), vec3(1.0));
vec3 finalColor = applyBitwiseBlend(combined, baseVideo);
color = vec4(finalColor, 1.0);
}

```

Shader #346 - fold fov

File: fold-fov.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

```

```

void main(void) {
    vec2 uv = tc;
    float foldAmount = sin(time_f * 3.14159);
    if (foldAmount > 0.0) {
        if (uv.x < 0.5) {
            uv.x = mix(uv.x, 0.5, foldAmount);
        } else {
            uv.x = mix(uv.x, 0.5, foldAmount);
        }
    } else {
        uv.x = mix(0.5 + abs(uv.x - 0.5), uv.x, abs(foldAmount));
    }
    vec2 translate = vec2(sin(time_f * 2.0), cos(time_f * 2.0)) * 0.25;
    uv += translate * foldAmount;
    uv = (uv - 0.5) * 1.5 + 0.5;
    uv = clamp(uv, 0.0, 1.0);
    color = texture(samp, uv);
}

```

Shader #347 - fold mirror

File: fold-mirror.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float mirrorAmount = sin(time_f * 3.14159);
    vec2 mirroredTC = vec2(abs(mirrorAmount * (tc.x - 0.5)) + 0.5 * sign(mirrorAmount), tc.y);
    color = texture(samp, mirroredTC);
}

```

Shader #348 - fold spin

File: fold-spin.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float angle = atan(uv.y, uv.x);
}

```

```

float radius = length(uv);
float spiralTime = mod(time_f, 4.0);
float maxRadius = sqrt(2.0);
if (spiralTime < 2.0) {
    radius = mix(radius, maxRadius, spiralTime / 2.0);
} else {
    radius = mix(maxRadius, radius, (spiralTime - 2.0) / 2.0);
}
angle += spiralTime * 3.14159;
uv = vec2(cos(angle), sin(angle)) * radius;
uv = (uv + 1.0) / 2.0;
color = texture(samp, uv);
}

```

Shader #350 - fold

File: fold.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float foldAmount = sin(time_f * 3.14159);
    vec2 foldedTC = vec2(tc.x, abs(foldAmount * (tc.y - 0.5)) + 0.5 * sign(foldAmount));
    color = texture(samp, foldedTC);
}

```

Shader #379 - frac shader02 dmd6iFold

File: frac_shader02_dmd6iFold.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Initial fractal folding Neon color palette Transparency manipulation

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;

```

```

uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 8; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        if (i % 2 == 0) p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
void main() {
    vec2 uv = tc;
    float aspect = iResolution.x / iResolution.y;

```

```

    vec2 m = iMouse.xy / iResolution.xy;
        if(iMouse.z == 0.0)
            m = uv / aspect;
    uv = fractalFold(uv, 1.45 + 0.55 * sin(time_f * 0.42), time_f, m, aspect);
    vec3 col = neonPalette(sin(time_f * aspect));
    float alpha = 0.8 - 0.6 * abs(sin(time_f + length(uv - m) * 10.0));
        vec3 fcol = texture(samp, tc).rgb;
    col = sin(col * (1.0+pingPong(time_f, 10.0))) * fcol;
    col *= alpha;
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    col *= 1.0 + _ab * 0.6;
    col = mix(col, col * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high, 0.0, 1.0) *
    ↪ 0.25), _ab);
    color = vec4(col, 1.0);
}

```

Shader #452 - funny mirror

File: funny_mirror.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float loop_dur = 15.0;
    float current_t = mod(time_f, loop_dur);
    vec2 uv = (tc - 0.5) * (iResolution.x / iResolution.y, 1.0);
    float radius = length(uv) * current_t;
    vec2 coord = sin(tc * radius/2);
    color = texture(samp, coord);
}

```

Shader #453 - funnymirror2

File: funnymirror2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float loop_dur = 15.0;
    float current_t = mod(time_f, loop_dur);
    vec2 uv = (tc - 0.5) * (iResolution.x / iResolution.y, 1.0);

```

```

float radius = length(uv) * current_t;
vec2 coord = sin(tc*time_f * radius/2);
color = texture(samp, coord);
}

```

Shader #638 - mirdmdrmatch

File: mirdmdrmatch.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = uv * 2.0 - 1.0;
    float speedFactor = sin(time_f * 0.5) * 0.5 + 0.5;
    float angle1 = time_f * (0.2 + speedFactor * 0.8);
    float angle2 = time_f * (0.2 + (1.0 - speedFactor) * 0.8);
    float angle = centeredUV.x >= 0.0 ? angle1 : angle2;
    mat2 rotation = mat2(cos(angle), -sin(angle), sin(angle), cos(angle));
    centeredUV = rotation * centeredUV;
    centeredUV = abs(centeredUV);
    vec2 mirroredUV = mod(centeredUV, 1.0);
    vec4 texColor = texture(samp, mirroredUV);
    float pulse = 0.5 + 0.5 * sin(time_f * 3.0);
    vec3 pulsatingColor = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 10.0),
        0.5 + 0.5 * cos(time_f + uv.y * 10.0),
        0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 10.0)
    ) * pulse;
    color = vec4(texColor.rgb * 0.7 + pulsatingColor * 0.3, texColor.a);
}

```

Shader #639 - mirdmdrspeed

File: mirdmdrspeed.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = uv * 2.0 - 1.0;
    float speedFactor = sin(time_f * 0.5) * 0.5 + 0.5;
    float angle1 = time_f * (0.2 + speedFactor * 0.8);
    float angle2 = time_f * (0.2 + (1.0 - speedFactor) * 0.8);

```

```

    if (centeredUV.x >= 0.0) {
        mat2 rotation1 = mat2(cos(angle1), -sin(angle1), sin(angle1), cos(angle1));
        centeredUV = rotation1 * centeredUV;
    } else {
        mat2 rotation2 = mat2(cos(angle2), -sin(angle2), sin(angle2), cos(angle2));
        centeredUV = rotation2 * centeredUV;
    }
    centeredUV = abs(centeredUV);
    vec2 mirroredUV = mod(centeredUV, 1.0);
    vec4 texColor = texture(samp, mirroredUV);
    float pulse = 0.5 + 0.5 * sin(time_f * 3.0);
    vec3 pulsatingColor = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 10.0),
        0.5 + 0.5 * cos(time_f + uv.y * 10.0),
        0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 10.0)
    ) * pulse;
    color = vec4(texColor.rgb * 0.7 + pulsatingColor * 0.3, texColor.a);
    color = vec4(sin(color.rgb * time_f), texColor.a);
}

```

Shader #641 - mirror air bowl

File: mirror-air-bowl.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control.

Technique: Multi-tap convolution blur (4 texture samples) - Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
float h1(float n){return fract(sin(n*91.345+37.12)*43758.5453123);}
vec2 h2(vec2 p){return fract(sin(vec2(dot(p,vec2(127.1,311.7)),dot(p,vec2(269.5,183.3))))*43758.5453);}
vec2 rot(vec2 v,float a){float c=cos(a),s=sin(a);return vec2(c*v.x-s*v.y,s*v.x+c*v.y);}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void){
    float a = clamp(amp,0.0,1.0);
    float ua = clamp(uamp,0.0,1.0);
    float t = time_f;
    vec2 center = vec2(0.5, 0.5);
    vec2 cv = 1.0 - abs(1.0 - 2.0 * tc);
    cv = cv - floor(cv);
    vec2 baseUV = cv;
    vec2 offset = baseUV - center;
    float maxRadius = length(vec2(0.5, 0.5));
    float radius = length(offset);

```

```

float normalizedRadius = radius / maxRadius;
float distortion = (0.25 + 0.45*ua + 0.3*a);
float distortedRadius = normalizedRadius + distortion * normalizedRadius * normalizedRadius;
distortedRadius = clamp(distortedRadius, 0.0, 1.0);
distortedRadius *= maxRadius;
vec2 distortedCoords = center + distortedRadius * (radius > 0.0 ? offset / radius : vec2(0.0));
float spinSpeed = 0.6 + 1.8*(0.3 + 0.7*a);
float modulatedTime = pingPong(t * spinSpeed, 5.0);
float angle = atan(distortedCoords.y - center.y, distortedCoords.x - center.x) + modulatedTime;
vec2 rotatedTC;
rotatedTC.x = cos(angle) * (distortedCoords.x - center.x) - sin(angle) * (distortedCoords.y - center
↪ .y) + center.x;
rotatedTC.y = sin(angle) * (distortedCoords.x - center.x) + cos(angle) * (distortedCoords.y - center
↪ .y) + center.y;
float warpAmp = 0.02 + 0.06*ua + 0.04*a;
vec2 warpedCoords;
warpedCoords.x = pingPong(rotatedTC.x + t * 0.12 * (1.0 + warpAmp*5.0), 1.0);
warpedCoords.y = pingPong(rotatedTC.y + t * 0.12 * (1.0 + warpAmp*5.0), 1.0);
vec2 uv = cv * warpedCoords;
float speedR=5.0, ampR=0.03, waveR=10.0;
float speedG=6.5, ampG=0.025, waveG=12.0;
float speedB=4.0, ampB=0.035, waveB=8.0;
float rR=sin(uv.x*waveR+t*speedR)*ampR + sin(uv.y*waveR*0.8+t*speedR*1.2)*ampR;
float rG=sin(uv.x*waveG*1.5+t*speedG)*ampG + sin(uv.y*waveG*0.3+t*speedG*0.7)*ampG;
float rB=sin(uv.x*waveB*0.5+t*speedB)*ampB + sin(uv.y*waveB*1.7+t*speedB*1.3)*ampB;
vec2 tcR=uv+vec2(rR,rR);
vec2 tcG=uv+vec2(rG,-0.5*rG);
vec2 tcB=uv+vec2(0.3*rB,rB);
vec3 pats[4]=vec3[(vec3(1,0,1),vec3(0,1,0),vec3(1,0,0),vec3(0,0,1))];
float pspd=4.0;
int pidx=int(mod(floor(t*pspd+seed*4.0),4.0));
vec3 mir=pats[pidx];
vec2 m = iMouse.z>0.5 ? (iMouse.xy/iResolution) : fract(vec2(0.37+0.11*sin(t*0.63+seed),0.42+0.13*
↪ cos(t*0.57+seed*2.0)));
vec2 dR=tcR-m, dG=tcG-m, dB=tcB-m;
float fallR=smoothstep(0.55,0.0,length(dR));
float fallG=smoothstep(0.55,0.0,length(dG));
float fallB=smoothstep(0.55,0.0,length(dB));
float sw=(0.12+0.38*ua+0.25*a);
vec2 tangR=rot(normalize(dR+1e-4),1.5707963);
vec2 tangG=rot(normalize(dG+1e-4),1.5707963);
vec2 tangB=rot(normalize(dB+1e-4),1.5707963);
vec2 airR=tangR*sw*fallR*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*40.0+t*3.0+seed));
vec2 airG=tangG*sw*fallG*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*38.0+t*3.3+seed*1.7));
vec2 airB=tangB*sw*fallB*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*42.0+t*2.9+seed*0.9));
vec2 jit = (h2(uv*vec2(233.3,341.9)+t+seed)-0.5)*(0.0006+0.004*ua);
tcR += airR + jit;
tcG += airG + jit;
tcB += airB + jit;
vec2 fR=vec2(mir.r>0.5?1.0-tcR.x:tcR.x, tcR.y);
vec2 fG=vec2(mir.g>0.5?1.0-tcG.x:tcG.x, tcG.y);
vec2 fB=vec2(mir.b>0.5?1.0-tcB.x:tcB.x, tcB.y);
float ca=0.0015+0.004*a;
vec4 C=texture(samp,uv);
C.r=texture(samp,fR+vec2(ca,0)).r;
C.g=texture(samp,fG).g;
C.b=texture(samp,fB+vec2(-ca,0)).b;
float pulse=0.004*(0.5+0.5*sin(t*3.7+seed));
C.rgb+=pulse*ua;
color=vec4(C.rgb,1.0);
}

```


Shader #642 - mirror airshader1

File: mirror-airshader1.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control.

Technique: Color quantization via floor rounding - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
float h1(float n){return fract(sin(n*91.345+37.12)*43758.5453123);}
vec2 h2(vec2 p){return fract(sin(vec2(dot(p,vec2(127.1,311.7)),dot(p,vec2(269.5,183.3))))*43758.5453);}
vec2 rot(vec2 v,float a){float c=cos(a),s=sin(a);return vec2(c*v.x-s*v.y,s*v.x+c*v.y);}
void main(void){
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    float a=clamp(amp,0.0,1.0);
    float ua=clamp(uamp,0.0,1.0);
    float t=time_f;
    float speedR=5.0, ampR=0.03, waveR=10.0;
    float speedG=6.5, ampG=0.025, waveG=12.0;
    float speedB=4.0, ampB=0.035, waveB=8.0;
    float rR=sin(uv.x*waveR+t*speedR)*ampR + sin(uv.y*waveR*0.8+t*speedR*1.2)*ampR;
    float rG=sin(uv.x*waveG*1.5+t*speedG)*ampG + sin(uv.y*waveG*0.3+t*speedG*0.7)*ampG;
    float rB=sin(uv.x*waveB*0.5+t*speedB)*ampB + sin(uv.y*waveB*1.7+t*speedB*1.3)*ampB;
    vec2 tcR=uv+vec2(rR,rR);
    vec2 tcG=uv+vec2(rG,-0.5*rG);
    vec2 tcB=uv+vec2(0.3*rB,rB);
    vec3 pats[4]=vec3[(vec3(1,0,1),vec3(0,1,0),vec3(1,0,0),vec3(0,0,1))];
    float pspd=4.0;
    int pidx=int(mod(floor(t*pspd+seed*4.0),4.0));
    vec3 mir=pats[pidx];
    vec2 m = iMouse.z>0.5 ? (iMouse.xy/iResolution) : fract(vec2(0.37+0.11*sin(t*0.63+seed),0.42+0.13*
    ↪ cos(t*0.57+seed*2.0)));
    vec2 dR=tcR-m, dG=tcG-m, dB=tcB-m;
    float fallR=smoothstep(0.55,0.0,length(dR));
    float fallG=smoothstep(0.55,0.0,length(dG));
    float fallB=smoothstep(0.55,0.0,length(dB));
    float sw=(0.12+0.38*ua+0.25*a);
    vec2 tangR=rot(normalize(dR+1e-4),1.5707963);
    vec2 tangG=rot(normalize(dG+1e-4),1.5707963);
    vec2 tangB=rot(normalize(dB+1e-4),1.5707963);
    vec2 airR=tangR*sw*fallR*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*40.0+t*3.0+seed));
    vec2 airG=tangG*sw*fallG*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*38.0+t*3.3+seed*1.7));
    vec2 airB=tangB*sw*fallB*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*42.0+t*2.9+seed*0.9));
    vec2 jit = (h2(uv*vec2(233.3,341.9)+t+seed)-0.5)*(0.0006+0.004*ua);
    tcR += airR + jit;
    tcG += airG + jit;
    tcB += airB + jit;
    vec2 fR=vec2(mir.r>0.5?1.0-tcR.x:tcR.x, tcR.y);
    vec2 fG=vec2(mir.g>0.5?1.0-tcG.x:tcG.x, tcG.y);
```

```

    vec2 fB=vec2(mir.b>0.5?1.0-tcB.x:tcB.x, tcB.y);
    float ca=0.0015+0.004*a;
    vec4 C=texture(samp,uv);
    C.r=texture(samp,fR+vec2( ca,0)).r;
    C.g=texture(samp,fG          ).g;
    C.b=texture(samp,fB+vec2(-ca,0)).b;
    float pulse=0.004*(0.5+0.5*sin(t*3.7+seed));
    C.rgb+=pulse*ua;
    color=vec4(C.rgb,1.0);
}

```

Shader #643 - mirror atan

File: mirror-atan.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
    float maxRadius = length(vec2(0.5, 0.5));
    float radius = length(offset);
    float normalizedRadius = radius / maxRadius;
    float angle = atan(offset.y, offset.x);
    float distortion = 0.5;
    float distortedRadius = normalizedRadius + distortion * pow(normalizedRadius, 2.0);
    distortedRadius = clamp(distortedRadius, 0.0, 1.0);
    distortedRadius *= maxRadius;
    vec2 distortedCoords = center + distortedRadius * vec2(cos(angle), sin(angle));
    float modulatedTime = pingPong(time_f, 5.0);
    angle += modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle) * (distortedCoords.x - center.x) - sin(angle) * (distortedCoords.y - center
↵ .y) + center.x;
    rotatedTC.y = sin(angle) * (distortedCoords.x - center.x) + cos(angle) * (distortedCoords.y - center
↵ .y) + center.y;
    vec2 warpedCoords;
    warpedCoords.x = pingPong(rotatedTC.x + time_f * 0.1, 1.0);
    warpedCoords.y = pingPong(rotatedTC.y + time_f * 0.1, 1.0);
    color = texture(samp, warpedCoords);
}

```

Shader #644 - mirror bowl by time

File: mirror-bowl-by-time.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float h1(float n){ return fract(sin(n)*43758.5453123); }
vec2 h2(float n){ return fract(sin(vec2(n, n+1.0))*vec2(43758.5453,22578.1459)); }
void main(void){
    float rate = 0.6;
    float t = time_f * rate;
    float t0 = floor(t);
    float a = fract(t);
    float w = a*a*(3.0-2.0*a);
    vec2 p0 = vec2(0.15) + h2(t0)*0.7;
    vec2 p1 = vec2(0.15) + h2(t0+1.0)*0.7;
    vec2 center = mix(p0, p1, w);
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec2 p = uv - center;
    float r = length(p);
    float ang = atan(p.y, p.x);
    float swirl = 2.2 + 0.8*sin(time_f*0.35);
    float spin = 0.6*sin(time_f*0.2);
    ang += swirl * r + spin;
    float bend = 0.35;
    float rp = r + bend * r * r;
    uv = center + vec2(cos(ang), sin(ang)) * rp;
    uv += 0.02 * vec2(sin((tc.y+time_f)*4.0), cos((tc.x-time_f)*3.5));
    uv = vec2(pingPong(uv.x, 1.0), pingPong(uv.y, 1.0));
    color = texture(samp, uv);
}
```

Shader #645 - mirror bowl

File: mirror-bowl.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
```

```

uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float h1(float n){ return fract(sin(n)*43758.5453123); }
vec2 h2(float n){ return fract(sin(vec2(n, n+1.0))*vec2(43758.5453,22578.1459)); }
void main(void){
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    float rate = 0.6;
    float t = time_f * rate;
    float t0 = floor(t);
    float a = fract(t);
    float w = a*a*(3.0-2.0*a);
    vec2 p0 = vec2(0.15) + h2(t0)*0.7;
    vec2 p1 = vec2(0.15) + h2(t0+1.0)*0.7;
    vec2 center = mix(p0, p1, w);
    vec2 p = uv - center;
    float r = length(p);
    float ang = atan(p.y, p.x);
    float swirl = 2.2 + 0.8*sin(time_f*0.35);
    float spin = 0.6*sin(time_f*0.2);
    ang += swirl * r + spin;
    float bend = 0.35;
    float rp = r + bend * r * r;
    vec2 uvx = center + vec2(cos(ang), sin(ang)) * rp;
    uvx += 0.02 * vec2(sin((tc.y+time_f)*4.0), cos((tc.x-time_f)*3.5));
    uv = vec2(pingPong(uvx.x, 1.0), pingPong(uvx.y, 1.0));
    color = texture(samp, uvx);
}

```

Shader #648 - mirror c

File: mirror-c.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform float time_f;
uniform sampler2D samp;
float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main() {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    color = texture(samp, sin(uv * pingPong(time_f * PI, 3.0)));
}

```

Shader #649 - mirror center

File: mirror-center.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    float dist = length(uv);
    float angle = time_f * 2.0 + dist * 5.0;
    float s = sin(angle);
    float c = cos(angle);
    uv = vec2(
        uv.x * c - uv.y * s,
        uv.x * s + uv.y * c
    );
    uv += center;
    color = texture(samp, uv);
}
```

Shader #651 - mirror color o1

File: mirror-color-o1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 m = (iMouse.z > 0.5 ? iMouse.xy : 0.5 * iResolution) / iResolution;
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    uv = (uv - m) * vec2(iResolution.x / iResolution.y, 1.0);
    float t = time_f * 0.5;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x) + t;
    float radMod = pingPong(radius + t * 0.5, 0.5);
    float wave = sin(radius * 10.0 - t * 5.0) * 0.5 + 0.5;
```

```

float r = sin(angle * 3.0 + radMod * 10.0 + wave * 6.2831);
float g = sin(angle * 4.0 - radMod * 8.0 + wave * 4.1230);
float b = sin(angle * 5.0 + radMod * 12.0 - wave * 3.4560);
vec3 col = vec3(r, g, b) * 0.5 + 0.5;
vec3 texColor = texture(samp, tc).rgb;
col = mix(col, texColor, 0.3);
color = vec4(col, alpha);
}

```

Shader #652 - mirror color swirl

File: mirror-color-swirl.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
uniform vec4 iMouse;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 m = (iMouse.z > 0.5 ? iMouse.xy : 0.5 * iResolution) / iResolution;
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    uv = (uv - m) * vec2(iResolution.x / iResolution.y, 1.0);
    float t = time_f * 0.5;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    angle += t;
    float radMod = pingPong(radius + t * 0.5, 0.5);
    float wave = sin(radius * 10.0 - t * 5.0) * 0.5 + 0.5;
    float r = sin(angle * 3.0 + radMod * 10.0 + wave * 6.2831);
    float g = sin(angle * 4.0 - radMod * 8.0 + wave * 4.1230);
    float b = sin(angle * 5.0 + radMod * 12.0 - wave * 3.4560);
    vec3 col = vec3(r, g, b) * 0.5 + 0.5;
    vec3 texColor = texture(samp, tc).rgb;
    col = mix(col, texColor, 0.3);
    color = vec4(sin(col * pingPong(time_f, 8.0) + 2.0), alpha);
}

```

Shader #656 - mirror geometric

File: mirror-geometric.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float pingPong(float x,float length){
    float m=mod(x,length*2.0);
    return m<=length?m:length*2.0-m;
}
void main(void){
    float A=clamp(amp,0.0,1.0);
    float U=clamp(uamp,0.0,1.0);
    float time_z=pingPong(time_f,4.0)+0.5;
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec2 center=(iMouse.z>0.5||iMouse.w>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 normPos=(uv-center)*vec2(iResolution.x/iResolution.y,1.0);
    float dist=length(normPos);
    float phase=sin(dist*10.0-time_f*4.0);
    float phaseGain=mix(0.6,1.4,A);
    vec2 tcAdjusted=uv+(normPos*0.305*phase*phaseGain);
    float dispersionScale=0.02*(0.5+U);
    vec2 dispersionOffset=normPos*dist*dispersionScale;
    vec2 tcAdjustedR=tcAdjusted-dispersionOffset;
    vec2 tcAdjustedG=tcAdjusted;
    vec2 tcAdjustedB=tcAdjusted+dispersionOffset;
    float r=texture(samp,tcAdjustedR).r;
    float g=texture(samp,tcAdjustedG).g;
    float b=texture(samp,tcAdjustedB).b;
    color=vec4(r,g,b,1.0);
}
```

Shader #657 - mirror goofy

File: mirror-goofy.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - 2D rotation matrix - Hermite smoothstep interpolation - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void) {
    vec2 center = vec2(0.5);
```

```

    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    uv = uv - center;
    float r = length(uv);
    float t = time_f;
    float s = pingPong(t, 10.0) * 0.1;
    float bendR = 0.15 + 0.1*sin(t*0.5);
    float swirl = (0.35 + 0.25*sin(t*0.33)) * (1.0 - smoothstep(0.0, 0.707, r));
    float ang = atan(uv.y, uv.x) + swirl;
    float rb = r * (1.0 + bendR * sin(r*12.0 + t*1.7));
    vec2 n1 = vec2(cos(t*0.37), sin(t*0.37));
    vec2 n2 = vec2(cos(t*0.53+1.7), sin(t*0.53+1.7));
    float w1 = sin(dot(uv, n1)*18.0 + t*1.3);
    float w2 = sin(dot(uv, n2)*14.0 - t*1.1);
    vec2 dirBend = normalize(n1)*w1 + normalize(n2)*w2;
    vec2 uvb = vec2(cos(ang), sin(ang)) * rb;
    uvb += dirBend * (0.025 + 0.02*sin(t*0.21)) * (0.5 + 0.5*sin(r*10.0 + t));
    float rot = sin(t*3.14159265*0.2) * 0.6;
    mat2 R = mat2(cos(rot), -sin(rot), sin(rot), cos(rot));
    uvb = R * uvb;
    uv = uvb + center;
    uv -= sin(uv*6.28318 + t) * (0.01 + 0.01*s);
    color = texture(samp, uv);
}

```

Shader #658 - mirror gpt

File: mirror-gpt.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Adjust the blur radius for a stronger effect

Technique: Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color = ivec3(icolor.rgb * 255.0);
    ivec3 isource = ivec3(source.rgb * 255.0);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int_color[i] ^ isource[i];
        int_color[i] = int_color[i] % 256;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 enhancedBlur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float totalWeight = 0.0;

```



```

    int radius = 5;
    for (int x = -radius; x <= radius; ++x) {
        for (int y = -radius; y <= radius; ++y) {
            float distance = length(vec2(x, y));
            float weight = exp(-distance * distance / (2.0 * radius * radius));
            vec2 offset = vec2(x, y) * texelSize;
            result += texture(image, uv + offset) * weight;
            totalWeight += weight;
        }
    }
    return result / totalWeight;
}

void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    uv.x += sin(uv.y * 20.0 + time_f * 2.0) * 0.005;
    uv.y += cos(uv.x * 20.0 + time_f * 2.0) * 0.005;
    vec4 tcolor = enhancedBlur(samp, uv, iResolution);
    float time_t = pingPong(time_f * 0.3, 5.0) + 1.0;
    vec4 modColor = vec4(
        tcolor.r * (0.5 + 0.5 * sin(time_f + uv.x * 10.0)),
        tcolor.g * (0.5 + 0.5 * sin(time_f + uv.y * 10.0)),
        tcolor.b * (0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 5.0)),
        tcolor.a
    );
    color = xor_RGB(sin(tcolor * time_f), modColor);
    color.rgb = pow(color.rgb, vec3(1.2));
}

```

Shader #659 - mirror grad

File: mirror-grad.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}

vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}

void main(void) {

```

```

        vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec2 warp = uv + vec2(
        sin(uv.y * 10.0 + time_f) * 0.1,
        sin(uv.x * 20.0 + time_f) * 0.1
    );
    vec3 colorShift = vec3(
        0.2 * sin(time_f * 0.2) + 0.8,
        0.4 * sin(time_f * 0.4 + 6.0) + 0.5,
        0.8 * sin(time_f * 0.6 + 4.0) + 0.5
    );
    float time_t = mod(time_f, 50);
    float feedback = rand(uv + time_f);
    vec2 feedbackUv = warp;
    vec4 texColor = texture(samp, feedbackUv);
    vec3 finalColor = texColor.rgb + colorShift;
    color = sin(vec4(finalColor, texColor.a) * (time_f * 0.5));
    color.a = 1.0;
}

```

Shader #660 - mirror in

File: mirror-in.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float angle = sin(time_f) * 3.14159265;
    float s = sin(angle);
    float c = cos(angle);
    uv = mat2(c, -s, s, c) * uv;
    uv = abs(uv);
    uv = uv * 0.5 + 0.5;
    color = texture(samp, uv);
}

```

Shader #661 - mirror l

File: mirror-l.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: UV mirroring/folding for symmetry - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 pos = tc;
    float time_t = mod(time_f, 3.0);
    pos.x = 1.0-tc.x;
    color = mix(texture(samp, tan(tc * time_t)), texture(samp, tan(pos * time_t)), 0.5);
}

```

Shader #663 - mirror pebble

File: mirror-pebble.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec2 normCoord = (uv * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float dist = length(normCoord);
    float maxRippleRadius = 25.0;
    float rippleSpeed = 2.0 * pingPong(time_f, 10.0);
    float phase = mod(time_f * rippleSpeed, maxRippleRadius);
    float ripple = sin((dist - phase) * 10.0) * exp(-dist * 3.0);
    vec2 displacedCoord = vec2(tc.x, tc.y + ripple * sin(time_f));
    color = texture(samp, displacedCoord);
}

```

Shader #664 - mirror pong

File: mirror-pong.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

```

```

}
void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    float angle1 = atan(uv.y - 0.5, uv.x - 0.5);
    float modulatedTime1 = pingPong(time_f, 3.0);
    angle1 += modulatedTime1;
    float angle2 = atan(uv.x - 0.5, uv.y - 0.5);
    float modulatedTime2 = pingPong(time_f * 0.5, 2.5);
    angle2 += modulatedTime2;
    float angle3 = atan(uv.y - 0.5 + modulatedTime2, uv.x - 0.5 + modulatedTime1);
    float modulatedTime3 = pingPong(time_f * 1.5, 4.0);
    angle3 += modulatedTime3;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle3) * (uv.x - 0.5) - sin(angle3) * (uv.y - 0.5) + 0.5;
    rotatedTC.y = sin(angle3) * (uv.x - 0.5) + cos(angle3) * (uv.y - 0.5) + 0.5;
    rotatedTC = sin(rotatedTC * (modulatedTime1 * modulatedTime2 * modulatedTime3));
    color = texture(samp, rotatedTC);
}

```

Shader #665 - mirror pong2

File: mirror-pong2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float h1(float n){ return fract(sin(n)*43758.5453123); }
vec2 h2(float n){ return fract(sin(vec2(n,n+1.0))*vec2(43758.5453,22578.1459)); }
void main(void) {
    float rate = 0.35;
    float t = time_f*rate;
    float k = floor(t);
    float a = fract(t);
    a = a*a*(3.0-2.0*a);
    vec2 p0 = h2(k)*2.0-1.0;
    vec2 p1 = h2(k+1.0)*2.0-1.0;
    vec2 shift = mix(p0,p1,a)*0.12;
    float s0 = 0.9 + 0.2*h1(k+10.0);
    float s1 = 0.9 + 0.2*h1(k+11.0);
    float scale = mix(s0,s1,a);
    float dir = sign(h1(k+20.0)-0.5);
    float modulatedTime = pingPong(time_f + h1(k+30.0)*2.0, 5.0);
    float rot = dir * modulatedTime;
    vec2 center = vec2(0.5) + shift;
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
}

```

```

    vec2 d = uv - center;
    float ang = atan(d.y, d.x) + rot;
    float r = length(d)/scale;
    vec2 rotatedTC = center + r*vec2(cos(ang), sin(ang));
    rotatedTC = fract(rotatedTC);
    color = texture(samp, rotatedTC);
}

```

Shader #666 - mirror prism

File: mirror-prism.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (5 texture samples) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    color = texture(samp, uv);
    float radius = 1.0;
    vec2 center = iResolution * 0.5;
    vec2 texCoord = uv * iResolution;
    vec2 delta = texCoord - center;
    float dist = length(delta);
    float maxRadius = min(iResolution.x, iResolution.y) * radius;
    if (dist < maxRadius) {
        float scaleFactor = 1.0 - pow(dist / maxRadius, 2.0);
        vec2 direction = normalize(delta);
        float offsetR = 0.015;
        float offsetG = 0.0;
        float offsetB = -0.015;
        vec2 texCoordR = center + delta * scaleFactor + direction * offsetR * maxRadius;
        vec2 texCoordG = center + delta * scaleFactor + direction * offsetG * maxRadius;
        vec2 texCoordB = center + delta * scaleFactor + direction * offsetB * maxRadius;
        texCoordR /= iResolution;
        texCoordG /= iResolution;
        texCoordB /= iResolution;
        texCoordR = clamp(texCoordR, 0.0, 1.0);
        texCoordG = clamp(texCoordG, 0.0, 1.0);
        texCoordB = clamp(texCoordB, 0.0, 1.0);
        float r = texture(samp, texCoordR).r;
        float g = texture(samp, texCoordG).g;
        float b = texture(samp, texCoordB).b;
        color = vec4(r, g, b, 1.0);
    } else {
        vec2 newTexCoord = clamp(tc, 0.0, 1.0);
        color = texture(samp, newTexCoord);
    }
}

```

Shader #667 - mirror psyce wave all

File: mirror-psyce-wave-all.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void)
{
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec2 normCoord = ((tc.xy / iResolution.xy) * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float distanceFromCenter = length(normCoord);
    float wave = sin(distanceFromCenter * 12.0 - time_f * 4.0);
    vec2 tcAdjusted = uv + (normCoord * 0.301 * wave);
    vec4 textureColor = texture(samp, tcAdjusted);
    color = textureColor;
}
```

Shader #668 - mirror putty

File: mirror-putty.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    float timeVar = time_f * 0.5;
    vec2 noise = vec2(
        pingPong(uv.x + timeVar, 1.0),
        pingPong(uv.y + timeVar, 1.0)
    );
    float stretchFactorX = 1.0 + 0.3 * sin(time_f + uv.y * 10.0);
    float stretchFactorY = 1.0 + 0.3 * cos(time_f + uv.x * 10.0);
    vec2 distortedUV = vec2(
        uv.x * stretchFactorX + noise.x * 0.1,
        uv.y * stretchFactorY + noise.y * 0.1
    );
    color = texture(samp, distortedUV);
}
```

Shader #669 - mirror self

File: mirror-self.glsl

Processes the input texture to produce the "mirror self" visual effect.

Technique: Direct texture color manipulation with arithmetic operations

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
void main() {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    color = texture(samp, uv);
}
```

Shader #670 - mirror sin osc

File: mirror-sin-osc.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main() {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec2 centeredUV = uv * 2.0 - 1.0;
    float angle = atan(centeredUV.y, centeredUV.x);
    float radius = length(centeredUV);
    float spin = time_f * 0.5;
    angle += floor(mod(angle + 3.14159, 3.14159 / 4.0)) * spin;
    angle = angle * pingPong(sin(time_f), 30.0);
    vec2 rotatedUV = vec2(cos(angle), sin(angle)) * radius;
    rotatedUV = abs(mod(rotatedUV, 2.0) - 1.0);
    vec4 texColor = texture(samp, rotatedUV * 0.5 + 0.5);
    vec3 gradientEffect = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 15.0),
        0.5 + 0.5 * cos(time_f + uv.y * 15.0),
        0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 15.0)
    );
    vec3 finalColor = mix(texColor.rgb, gradientEffect, 0.3);
    color = vec4(finalColor, texColor.a);
}
```

Shader #671 - mirror spiral aura

File: mirror-spiral-aura.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float spiral = angle + radius * 5.0 - time_f;
    float reflection = mod(spiral, 3.14159 * 2.0);
    vec2 spiralUV = vec2(cos(spiral), sin(spiral)) * radius + 0.5;
    vec2 reflectUV = vec2(cos(reflection), sin(reflection)) * radius + 0.5;
    vec4 gradientColor = vec4(0.5 + 0.5 * cos(time_f + radius * 10.0), 0.5 + 0.5 * sin(time_f + radius *
    ↪ 10.0), 0.5 + 0.5 * cos(time_f - radius * 10.0), 1.0);
    vec4 textureColor = texture(samp, reflectUV);
    color = mix(gradientColor, textureColor, 0.5);
}
```

Shader #672 - mirror spiral f

File: mirror-spiral-f.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * iResolution.xy;
    vec2 center = iResolution * 0.5;
    vec2 offset = uv - center;
    float radius = length(offset);
    float angle = atan(offset.y, offset.x);
    float spiralFactor = sin(time_f + radius * 10.0) * 0.5 + 0.5;
    radius *= spiralFactor;
    float time_t = pingPong(time_f, 25.0) + 2.0;
```



```

    vec2 newUV = vec2(
        center.x + radius * cos(angle * time_t),
        center.y + radius * sin(angle * time_t)
    ) / iResolution;
    newUV = mod(newUV, 1.0);
    newUV = abs(newUV);
    color = texture(samp, newUV);
}

```

Shader #673 - mirror spiral

File: mirror-spiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * iResolution.xy;
    vec2 center = iResolution * 0.5;
    vec2 offset = uv - center;
    float radius = length(offset);
    float angle = atan(offset.y, offset.x);
    float spiralFactor = sin(time_f + radius * 10.0) * 0.5 + 0.5;
    radius *= spiralFactor;
    vec2 newUV = vec2(
        center.x + radius * cos(angle),
        center.y + radius * sin(angle)
    ) / iResolution;
    newUV = mod(newUV, 1.0);
    newUV = abs(newUV);
    color = texture(samp, newUV);
}

```

Shader #674 - mirror swirly

File: mirror-swirly.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look. Responds to mouse position for interactive control.

Technique: Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
void main(void) {

```

```

    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 d = tc - m;
    float r = length(d);
    vec2 dir = d / max(r, 1e-6);
    float waveLength = 0.05;
    float amplitude = 0.02;
    float speed = 2.0;
    float ripple = sin((r / waveLength - time_f * speed) * 6.2831853);
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    uv = uv + dir * ripple * amplitude;
    color = texture(samp, uv);
}

```

Shader #675 - mirror twist gpt

File: mirror-twist-gpt.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position and audio input (amplitude/volume) for interactive control. Applies matrix transformations.

Technique: Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float pp(float x, float l){float m=mod(x,l*2.0);return abs(m-l);}
float s2(float x){return x*x*(3.0-2.0*x);}
void main(void){
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    float a = clamp(amp,0.0,1.0);
    float ua = clamp(uamp,0.0,1.0);
    vec2 c;
    if(iMouse.z>0.5) c = iMouse.xy / iResolution;
    else{
        float t = time_f*0.3;
        c = vec2(0.5) + 0.25*vec2(cos(t*0.9), sin(t*1.1))*0.6;
    }
    float r = length(uv - c);
    float rippleSpeed = mix(4.0,10.0,ua);
    float rippleAmp = mix(0.01,0.06,a);
    float rippleWave = mix(8.0,18.0,ua);
    float ripple = sin(uv.x*rippleWave + time_f*rippleSpeed) * rippleAmp;
    ripple += sin(uv.y*rippleWave*0.9 + time_f*rippleSpeed*1.07) * rippleAmp;
    vec2 uv_rip = uv + vec2(ripple);
    float twist = mix(0.8,2.2,a) * (r - 0.8) + time_f*(0.3+ua*0.7);
    float ca = cos(twist), sa = sin(twist);
    mat2 rot = mat2(ca,-sa,sa,ca);
    vec2 uv_twist = (rot*(uv - c)) + c;
    float w = s2(clamp(1.0 - r*2.0, 0.0, 1.0));
    float k = mix(0.35,0.75,ua);
    vec2 uv_mix = mix(uv_rip, uv_twist, k*w + (1.0-k)*0.5);
}

```

```

float zoom = 1.0 + 0.02*sin(time_f*0.8 + ua*2.0);
vec2 zc = (uv_mix - c)/zoom + c;
vec4 col0 = texture(samp, uv);
vec4 col1 = texture(samp, zc);
color = col1;
}

```

Shader #676 - mirror twist sin

File: mirror-twist-sin.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    float angle = time_f * 0.5;
    float radius = length(uv - 0.5);
    float twist = radius * 5.0;
    float s = sin(twist + angle);
    float c = cos(twist + angle);
    float time_t = pingPong(time_f, 25.0) + 1.0;
    uv -= 0.5;
    uv = mat2(c, -s, s, c) * uv;
    uv += 0.5;
    if (uv.x > 0.5) {
        uv.x = 1.0 - uv.x;
    } else {
        uv.x = uv.x;
    }
    color = texture(samp, sin(uv * time_t));
}

```

Shader #677 - mirror twist

File: mirror-twist.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float angle = time_f * 0.5;
    float radius = length(uv - 0.5);
    float twist = radius * 5.0;
    float s = sin(twist + angle);
    float c = cos(twist + angle);
    uv -= 0.5;
    uv = mat2(c, -s, s, c) * uv;
    uv += 0.5;
    if (uv.x > 0.5) {
        uv.x = 1.0 - uv.x;
    } else {
        uv.x = uv.x;
    }
    color = texture(samp, uv);
}

```

Shader #678 - mirror twisted

File: mirror-twisted.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    float angle = time_f * 0.5;
    float radius = length(uv - 0.5);
    float twist = radius * 5.0;
    float s = sin(twist + angle);
    float c = cos(twist + angle);
    float time_t = pingPong(time_f, 25.0) + 1.0;
    uv -= 0.5;
    uv = mat2(c, -s, s, c) * uv;
    uv += 0.5;
    if (uv.x > 0.5) {
        uv.x = 1.0 - uv.x;
    } else {
        uv.x = uv.x;
    }
    color = texture(samp, sin(uv * time_t));
}

```

Shader #679 - mirror ufo wrap

File: mirror-ufo-wrap.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec2 p2 = (uv - m) * ar;
    float ax = 0.25 * sin(time_f * 0.7);
    float ay = 0.25 * cos(time_f * 0.6);
    float az = time_f * 0.5;
    vec3 p3 = vec3(p2, 1.0);
    mat3 R = rotZ(az) * rotY(ay) * rotX(ax);
    vec3 r = R * p3;
    float k = 0.6;
    float zf = 1.0 / (1.0 + r.z * k);
    vec2 q = r.xy * zf;
    float dist = length(p2);
    float scale = 1.0 + 0.2 * sin(dist * 15.0 - time_f * 2.0);
    q *= scale;
    vec2 uvx = q / ar + m;
    uv = clamp(uvx, 0.0, 1.0);
    color = texture(samp, uv);
}
```

Shader #680 - mirror wrap dmd6i

File: mirror-wrap-dmd6i.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
```

```

uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * 0.35 + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;

```

```

    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = 0.3 + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - 0.25, r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + 2.0 * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * 0.23, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * 0.65;
    float rD = diamondRadius(p) + 1e-6;

```

```

float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * (0.0015 + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, 1.15, vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0);
float pulse = 0.5 + 0.5 * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #681 - mirror wrap edge

File: mirror-wrap-edge.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Hermite smoothstep interpolation - Driven by: time

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
void main() {
    vec2 p = tc - 0.5;
    float r = max(abs(p.x), abs(p.y)) * 2.0;
    float w = 1.0 - smoothstep(0.7, 1.0, r);
    w *= w;
    float a = time_f * 0.8 * w;
    float c = cos(a);
    float s = sin(a);
    vec2 q = vec2(p.x * c - p.y * s,
                  p.x * s + p.y * c) + 0.5;
    vec2 uv = 1.0 - abs(1.0 - 2.0 * q);
    uv = fract(uv);
    color = texture(samp, uv);
}

```


Shader #682 - mirror wrap rotate snap

File: mirror-wrap-rotate-snap.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
void main() {
    float raw = time_f * 0.5;
    float step = 1.57079632679;
    float a = round(raw / step) * step;
    float c = cos(a);
    float s = sin(a);
    vec2 p = tc - 0.5;
    vec2 uv = vec2(p.x * c - p.y * s,
                  p.x * s + p.y * c) + 0.5;
    uv = abs(fract(uv) * 2.0 - 1.0);
    color = texture(samp, uv);
}
```

Shader #683 - mirror wrap rotate

File: mirror-wrap-rotate.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
void main() {
    float a = time_f * 0.5;
    float c = cos(a);
    float s = sin(a);
    vec2 p = tc - 0.5;
    vec2 r = vec2(p.x * c - p.y * s,
                  p.x * s + p.y * c);
    vec2 uv = r + 0.5;
    uv = 1.0 - abs(1.0 - 2.0 * uv);
    uv = uv - floor(uv);
    color = texture(samp, uv);
}
```

Shader #684 - mirror wrap rotate90

File: mirror-wrap-rotate90.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
void main() {
    float a = time_f * 0.5;
    float c = cos(a);
    float s = sin(a);
    vec2 p = tc - 0.5;
    vec2 uv = vec2(p.x * c - p.y * s,
                  p.x * s + p.y * c) + 0.5;
    uv = abs(fract(uv) * 2.0 - 1.0);
    color = texture(samp, uv);
}

```

Shader #685 - mirror wrap spin

File: mirror-wrap-spin.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. or a separate angle uniform if you prefer rotate in UV space around the center of the texture mirror tiling so edges match when used as a texture

Technique: 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
void main() {
    float a = time_f * 0.5;
    float c = cos(a);
    float s = sin(a);
    vec2 p = tc - 0.5;
    vec2 tc_rot = vec2(
        p.x * c - p.y * s,
        p.x * s + p.y * c
    ) + 0.5;
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc_rot);
    uv = fract(uv);
    color = texture(samp, uv);
}

```

Shader #686 - mirror wrap

File: mirror-wrap.glsl

Processes the input texture to produce the "mirror wrap" visual effect.

Technique: Direct texture color manipulation with arithmetic operations

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

void main() {
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    color = texture(samp, uv);
}

```

Shader #687 - mirror zoom

File: mirror-zoom.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void){
    const float PI = 3.1415926535897932384626433832795;
    float aspect=iResolution.x/iResolution.y;
    vec2 ar=vec2(aspect,1.0);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 p=(tc-m)*ar;
    vec3 v=vec3(p,1.0);
    float ax=0.25*sin(time_f*0.7);
    float ay=0.25*cos(time_f*0.6);
    float az=0.4*time_f;
    mat3 R=rotZ(az)*rotY(ay)*rotX(ax);
    vec3 r=R*v;
    float persp=0.6;
    float zf=1.0/(1.0+r.z*persp);
    vec2 q=r.xy*zf;
    float eps=1e-6;
    float base=1.72;
    float period=log(base);
    float t=time_f*0.5;
    float rad=length(q)+eps;
    float ang=atan(q.y,q.x)+t*0.3;
    float k=fract((log(rad)-t)/period);
    float rw=exp(k*period);
    vec2 qwrap=vec2(cos(ang),sin(ang))*rw;
    float N=8.0;
    float stepA=6.28318530718/N;
    float a=atan(qwrap.y,qwrap.x)+time_f*0.05;
    a=mod(a,stepA);
    a=abs(a-stepA*0.5);
    vec2 kdir=vec2(cos(a),sin(a));
    vec2 kaleido=kdir*length(qwrap);

```

```

    kaleido = 1.0 - abs(1.0 - 2.0 * kaleido);
    kaleido = fract(kaleido);
    vec2 uv=kaleido/ar+m;
    uv=fract(uv);
    color=texture(samp,sin(uv * (PI * pingPong(fract(time_f), 1.0))));
}

```

Shader #688 - mirror1

File: mirror1.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec2 vpos = 1.0-tc;
    vec4 color2 = texture(samp, vpos);
    color = (0.5 * color) + (0.5 * color2);
}

```

Shader #689 - mirror1 op

File: mirror1_op.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;

```

```

uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
vec4 opposite(vec4 inputx) {
    ivec4 col = ivec4(inputx * 255);
    col[0] = 255-col[0];
    col[1] = 255-col[1];
    col[2] = 255-col[2];
    vec4 val;
    for(int i = 0; i < 4; ++i) {
        val[i] = float(col[i]) / 255;
    }
    return val;
}
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec2 vpos = 1.0-tc;
    vec4 color2 = opposite(texture(samp, vpos));
    color = (0.5 * color) + (0.5 * color2);
}

```

Shader #690 - mirror1 s

File: mirror1_s.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;

```

```

uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec2 vpos = 1.0-tc;
    vec4 color2 = texture(samp, vpos);
    color = (0.8 * color) + (0.8 * color2);
}

```

Shader #691 - mirror2

File: mirror2.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec2 vpos;
    vpos[0] = 1.0-tc[0];
    vpos[1] = tc[1];
    vec4 color2 = texture(samp, vpos);
    color = (0.5 * color) + (0.5 * color2);
}

```

Shader #692 - mirror2 op

File: mirror2_op.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution;

vec4 opposite(vec4 inputx) {
    ivec4 col = ivec4(inputx * 255);
    col[0] = 255-col[0];
    col[1] = 255-col[1];
    col[2] = 255-col[2];
    vec4 val;
    for(int i = 0; i < 4; ++i) {
        val[i] = float(col[i]) / 255;
    }
    return val;
}

void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution.xy;
    vec2 vpos;
    vpos[0] = 1.0-tc[0];
    vpos[1] = tc[1];
    vec4 color2 = opposite(texture(samp, vpos));
    color = (0.5 * color) + (0.5 * color2);
}
```

Shader #693 - mirror2 s

File: mirror2_s.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
```

```

uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec2 vpos;
    vpos[0] = 1.0-tc[0];
    vpos[1] = tc[1];
    vec4 color2 = texture(samp, vpos);
    color = (0.8 * color) + (0.8 * color2);
}

```

Shader #694 - mirror3

File: mirror3.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;

```



```

    vec2 vpos;
    vpos[1] = 1.0-tc[1];
    vpos[0] = tc[0];
    vec4 color2 = texture(samp, vpos);
    color = (0.5 * color) + (0.5 * color2);
}

```

Shader #695 - mirror3 op

File: mirror3_op.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
vec4 opposite(vec4 inputx) {
    ivec4 col = ivec4(inputx * 255);
    col[0] = 255-col[0];
    col[1] = 255-col[1];
    col[2] = 255-col[2];
    vec4 val;
    for(int i = 0; i < 4; ++i) {
        val[i] = float(col[i]) / 255;
    }
    return val;
}
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec2 vpos;
    vpos[1] = 1.0-tc[1];
    vpos[0] = tc[0];
    vec4 color2 = opposite(texture(samp, vpos));
    color = (0.5 * color) + (0.5 * color2);
}

```

Shader #696 - mirror3 s

File: mirror3_s.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution.xy;
    vec2 vpos;
    vpos[0] = 1.0-tc[0];
    vpos[1] = tc[1];
    vec4 color2 = texture(samp, vpos);
    color = (0.8 * color) + (0.8 * color2);
}
```

Shader #697 - mirror 8

File: mirror_8.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
mat2 rotate(float angle) {
    float c = cos(angle);
    float s = sin(angle);
    return mat2(c, -s, s, c);
}
vec2 mirror(vec2 uv) {
    return abs(mod(uv, 2.0) - 1.0);
}
```

```

}
void main() {
    vec2 uv = tc * iResolution.xy;
    vec2 sectionSize = iResolution / vec2(4.0, 2.0);
    vec2 sectionIndex = floor(uv / sectionSize);
    vec2 localUV = mod(uv, sectionSize) / sectionSize;
    float direction = mod(sectionIndex.x + sectionIndex.y, 2.0) * 2.0 - 1.0;
    float angle = direction * (time_f + length(sectionIndex) * 0.5);
    localUV = rotate(angle) * (localUV - 0.5) + 0.5;
    localUV = mirror(localUV);
    vec2 texCoord = (sectionIndex + localUV) * sectionSize / iResolution.xy;
    color = texture(samp, texCoord);
}

```

Shader #698 - mirror b

File: mirror_b.glsl

Processes the input texture to produce the "mirror b" visual effect.

Technique: Driven by: resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc;
    if (uv.x < 0.5) {
        uv.x = 1.0 - uv.x;
    }
    color = texture(samp, uv);
}

```

Shader #699 - mirror blur1

File: mirror_blur1.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;

```

```

uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec2 vpos = 1.0-tc;
    vec4 color2 = texture(samp, vpos);
    vec4 color3 = texture(samp, tc-0.01);
    vec4 color4 = texture(samp, tc+0.01);
    color = (0.3 * color) + (0.3 * color2) + (0.3 * color3) + (0.3 * color4);
}

```

Shader #700 - mirror blur2

File: mirror_blur2.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec2 vpos;
    vpos[0] = 1.0-tc[0];
    vpos[1] = tc[1];
    vec4 color2 = texture(samp, vpos);
    vec4 color3 = texture(samp, tc-0.01);
    vec4 color4 = texture(samp, tc+0.01);
    color = (0.3 * color) + (0.3 * color2) + (0.3 * color3) + (0.3 * color4);
}

```

Shader #701 - mirror blur3

File: mirror_blur3.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution.xy;
    vec2 vpos;
    vpos[1] = 1.0-tc[1];
    vpos[0] = tc[0];
    vec4 color2 = texture(samp, vpos);
    vec4 color3 = texture(samp, tc-0.01);
    vec4 color4 = texture(samp, tc+0.01);
    color = (0.3 * color) + (0.3 * color2) + (0.3 * color3) + (0.3 * color4);
}
```

Shader #702 - mirror c

File: mirror_c.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
mat2 rotate(float angle) {
    float c = cos(angle);
    float s = sin(angle);
```

```

    return mat2(c, -s, s, c);
}
vec2 mirror(vec2 uv) {
    return abs(mod(uv, 2.0) - 1.0);
}
void main() {
    vec2 uv = tc * iResolution.xy;
    vec2 sectionSize = vec2(100.0);
    vec2 sectionIndex = floor(uv / sectionSize);
    vec2 localUV = mod(uv, sectionSize) / sectionSize;
    float angle = time_f + length(sectionIndex) * 0.5;
    localUV = rotate(angle) * (localUV - 0.5) + 0.5;
    localUV = mirror(localUV);
    vec2 texCoord = (sectionIndex + localUV) * sectionSize / iResolution.xy;
    color = texture(samp, texCoord);
}

```

Shader #703 - mirror dir

File: mirror_dir.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: UV mirroring/folding for symmetry - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void)
{
    vec2 mirrored_tc = tc;
    int direction = int(mod(time_f, 4.0));
    if (direction == 0) {
        mirrored_tc = tc;
    } else if (direction == 1) {
        mirrored_tc.x = 1.0 - tc.x;
    } else if (direction == 2) {
        mirrored_tc.y = 1.0 - tc.y;
    } else if (direction == 3) {
        mirrored_tc = 1.0 - tc;
    }
    color = texture(samp, mirrored_tc);
}

```

Shader #704 - mirror half dir

File: mirror_half_dir.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;

```

```

uniform float time_f;
void main(void)
{
    vec2 uv = tc;
    float duration = 2.0;
    float totalDuration = 2.0 * duration;
    float currentTime = mod(time_f, totalDuration);
    if (currentTime < duration) {
        if (uv.x > 0.5) {
            uv.x = 1.0 - uv.x;
        }
    } else {
        if (uv.x < 0.5) {
            uv.x = 1.0 - uv.x;
        }
    }
    color = texture(samp, uv);
}

```

Shader #705 - mirror offset

File: mirror_offset.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    vec2 pos = gl_FragCoord.xy/iResolution_.xy;
    vec2 vpos = tc;
    vpos[0] = vpos[0]-alpha_r;
    vpos[1] = alpha_g+vpos[1];
    vec4 color2 = texture(samp, vpos);
    color = (0.5 * color) + (0.5 * color2);
}

```

Shader #706 - mirror offset rev

File: mirror_offset_rev.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution;
void main(void)
{
    color = texture(samp, tc);
    vec2 pos = gl_FragCoord.xy/iResolution.xy;
    vec2 vpos = tc;
    vpos[0] = vpos[0]+alpha_r;
    vpos[1] = alpha_g-vpos[1];
    vec4 color2 = texture(samp, vpos);
    color = (0.5 * color) + (0.5 * color2);
}
```

Shader #707 - mirror p

File: mirror_p.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float pingPong(float t, float l) {
    return abs(mod(t, 2.0 * l) - l);
}
void main() {
    vec2 uv = tc;
    float stretch = 1.0 + pingPong(time_f, 2.0) * 0.5;
    if (uv.x < 0.5) {
        uv.x = 0.5 - (0.5 - uv.x) * stretch;
        uv.x = 1.0 - uv.x;
    }
```



```

    } else {
        uv.x = 0.5 + (uv.x - 0.5) * stretch;
    }
    color = texture(samp, uv);
}

```

Shader #708 - mirror r4

File: mirror_r4.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look.

Technique: Color quantization via floor rounding - 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
mat2 rotate(float angle) {
    float c = cos(angle);
    float s = sin(angle);
    return mat2(c, -s, s, c);
}
vec2 mirror(vec2 uv) {
    return abs(mod(uv, 2.0) - 1.0);
}
void main() {
    vec2 uv = tc - 0.5;
    uv = rotate(time_f * 0.5) * uv + 0.5;
    vec2 sectionSize = iResolution / vec2(2.0, 2.0);
    vec2 center = (floor(uv * vec2(2.0, 2.0)) + 0.5) * sectionSize / iResolution;
    uv = (uv - center) * iResolution.xy / sectionSize;
    float direction = mod(floor(center.x * 2.0) + floor(center.y * 2.0), 2.0) * 2.0 - 1.0;
    uv = rotate(direction * (time_f + length(center))) * uv;
    uv = mirror(uv);
    vec2 texCoord = (uv + 0.5) * sectionSize / iResolution + center;
    color = texture(samp, texCoord);
}

```

Shader #709 - mirror r8

File: mirror_r8.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look.

Technique: Color quantization via floor rounding - 2D rotation matrix - UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
mat2 rotate(float angle) {

```

```

    float c = cos(angle);
    float s = sin(angle);
    return mat2(c, -s, s, c);
}
vec2 mirror(vec2 uv) {
    return abs(mod(uv, 2.0) - 1.0);
}
void main() {
    vec2 uv = tc - 0.5;
    uv = rotate(time_f * 0.5) * uv + 0.5;
    vec2 sectionSize = iResolution / vec2(4.0, 2.0);
    vec2 center = (floor(uv * vec2(4.0, 2.0)) + 0.5) * sectionSize / iResolution;
    uv = (uv - center) * iResolution.xy / sectionSize;
    float direction = mod(floor(center.x * 4.0) + floor(center.y * 2.0), 2.0) * 2.0 - 1.0;
    uv = rotate(direction * (time_f + length(center))) * uv;
    uv = mirror(uv);
    vec2 texCoord = (uv + 0.5) * sectionSize / iResolution + center;
    color = texture(samp, texCoord);
}

```

Shader #710 - mirror rad

File: mirror_rad.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float pingPong(float t, float l) {
    return abs(mod(t, 2.0 * l) - l);
}
void main() {
    vec2 uv = tc;
    float stretch = 1.0 + pingPong(time_f, 2.0) * 0.5;
    if (uv.x < 0.5) {
        uv.x = 0.5 - (0.5 - uv.x) * stretch;
        uv.x = 1.0 - uv.x;
    } else {
        uv.x = 0.5 + (uv.x - 0.5) * stretch;
    }
    float radius = 1.0;
    vec2 center = vec2(0.5, 0.5) * iResolution;
    vec2 texCoord = uv * iResolution;
    vec2 delta = texCoord - center;
    float dist = length(delta);
    float maxRadius = min(iResolution.x, iResolution.y) * radius;
    if (dist < maxRadius) {
        float scaleFactor = pingPong(time_f, 8.0) * (1.0 - pow(dist / maxRadius, 2.0));
        vec2 direction = normalize(delta);
        texCoord += direction * scaleFactor * 50.0;
    }
    uv = texCoord / iResolution;
    color = texture(samp, uv);
}

```

Shader #711 - mirror rev

File: mirror_rev.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)
{
    vec2 vpos;
    vpos[0] = 1.0-tc[0];
    vpos[1] = tc[1];
    vec2 pos = gl_FragCoord.xy / iResolution.xy;
    if(pos[0] < 0.5)
        color = texture(samp, tc);
    else
        color = texture(samp, vpos);
}
```

Shader #712 - mirror rev2

File: mirror_rev2.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)
{
    vec2 vpos;
    vpos[0] = 1.0-tc[0];
    vpos[1] = tc[1];
    vec2 pos = gl_FragCoord.xy / iResolution.xy;
    if(pos[0] > 0.5)
        color = texture(samp, tc);
    else
        color = texture(samp, vpos);
}
```

Shader #713 - mirror rev2y

File: mirror_rev2y.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
```

```

uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
void main(void)
{
    vec2 vpos;
    vpos[1] = 1.0-tc[1];
    vpos[0] = tc[0];
    vec2 pos = gl_FragCoord.xy / iResolution.xy;
    if(pos[1] > 0.5)
        color = texture(samp, tc);
    else
        color = texture(samp, vpos);
}

```

Shader #714 - mirror revy

File: mirror_revy.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;

```

```

uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)
{
    vec2 vpos;
    vpos[1] = 1.0-tc[1];
    vpos[0] = tc[0];
    vec2 pos = gl_FragCoord.xy / iResolution.xy;
    if(pos[1] < 0.5)
        color = texture(samp, tc);
    else
        color = texture(samp, vpos);
}

```

Shader #715 - mirrorg

File: mirrorg.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
 - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec2 centeredUV = (uv - 0.5) * 2.0;
    float radius = length(centeredUV);
    float mask = smoothstep(1.0, 0.9, radius);
    float angle = atan(centeredUV.y, centeredUV.x) + time_f * 0.3;
    float refractAmount = pow(1.0 - radius, 2.0) * 0.2;
    vec2 refractedUV = vec2(
        centeredUV.x + sin(angle) * refractAmount,
        centeredUV.y + cos(angle) * refractAmount
    );
    vec2 crystalUV = refractedUV * 0.5 + 0.5;
    vec4 texColorBall = texture(samp, crystalUV);
    vec4 texColorBackground = texture(samp, uv);
    color = mix(texColorBackground, texColorBall, mask);
}

```

Shader #964 - spiral mirror

File: spiral_mirror.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    normCoord.x = abs(normCoord.x);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralAmount = time_f;
    angle += spiralAmount;
    dist = mix(dist, 1.0 - dist, sin(spiralAmount * 0.5));
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * dist;
    float scale = 1.0 + sin(spiralAmount * 0.2) * 0.5;
    spiralCoord *= scale;
    spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
    color = texture(samp, spiralCoord);
}
```

Shader #965 - spiral mirror 2

File: spiral_mirror_2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float pingPong(float x, float l){float m=mod(x,l*2.0);return m<=l?m:l*2.0-m;}
mat2 rot(float a){float s=sin(a),c=cos(a);return mat2(c,-s,s,c);}
void main(void){
    vec2 toA=vec2(iResolution.x/iResolution.y,1.0);
    vec2 fromA=vec2(1.0/toA.x,1.0);
    vec2 p=(tc*2.0-1.0)*toA;
    p.x=abs(p.x);
    float r=length(p);
    float a=atan(p.y,p.x);
    float rotSpeed=0.9;
    float zoomPeriod=6.0;
    float zPhase=pingPong(time_f,zoomPeriod)/zoomPeriod;
```

```

float minZ=0.6;
float maxZ=2.4;
float zoom=mix(minZ,maxZ,zPhase);
a+=time_f*rotSpeed;
r/=zoom;
vec2 s=vec2(cos(a),sin(a))*r;
vec2 q=s;
q=rot(0.35)*q;
for(int i=0;i<5;i++){
    q=abs(q)/(dot(q,q)+0.001)-0.5;
    q=rot(0.25)*q;
}
float mixAmt=0.55+0.35*sin(time_f*0.3);
vec2 uv=mix(s,q,mixAmt);
uv=uv*fromA*0.5+0.5;
color=texture(samp,uv);
}

```

Shader #966 - spiral mirror rev

File: spiral_mirror_rev.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    normCoord.y = abs(normCoord.y);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralAmount = sin(time_f);
    angle += spiralAmount;
    dist = mix(dist, 1.0 - dist, sin(spiralAmount * 0.5));
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * dist;
    float scale = 1.0 + sin(spiralAmount * 0.2) * 0.5;
    spiralCoord *= scale;
    spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
    color = texture(samp, spiralCoord);
}

```

Shader #967 - spiral mirror top

File: spiral_mirror_top.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution


```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    if (normCoord.y < 0.0) {
        normCoord.y = -normCoord.y;
    }
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralAmount = time_f;
    angle += spiralAmount;
    dist = mix(dist, 1.0 - dist, sin(spiralAmount * 0.5));
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * dist;
    float scale = 1.0 + sin(spiralAmount * 0.2) * 0.5;
    spiralCoord *= scale;
    spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
    color = texture(samp, spiralCoord);
}

```

Shader #968 - spiral mirror wave

File: spiral_mirror_wave.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    normCoord.x = abs(normCoord.x);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralAmount = time_f;
    angle += spiralAmount;
    dist = mix(dist, 1.0 - dist, sin(spiralAmount * 0.5));
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * dist;
    float scale = 1.0 + sin(spiralAmount * 0.2) * 0.5;
    spiralCoord *= scale;
    spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
    float time_t = mod(time_f, 100);
    float waveScale = time_t;
    float speed = 16.0;
    float offset = sin(time_f * speed + spiralCoord.x * waveScale) * 0.05;
    vec2 tcOffset = vec2(spiralCoord.x, spiralCoord.y + offset);
    color = texture(samp, tcOffset);
}

```

Shader #1010 - tex fold

File: tex_fold.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float zoomFactor = 0.5 + 0.5 * sin(time_f);
    float angle = time_f;
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float cosAngle = cos(angle);
    float sinAngle = sin(angle);
    vec2 rotatedTc = vec2(
        cosAngle * dir.x - sinAngle * dir.y,
        sinAngle * dir.x + cosAngle * dir.y
    ) * zoomFactor + center;
    color = texture(samp, rotatedTc);
}
```

Audio Reactive (61 shaders)

Shader #6 - abc

File: abc.glsl

Uses distance fields with sinusoidal modulation to generate animated visual patterns. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float uamp;
uniform float amp;
uniform vec4 iMouse;
void main(void) {
    vec2 mousePos = iMouse.xy / iResolution.xy;
    float dist = distance(tc, mousePos);
    float glitchStrength = exp(-dist * 10.0) * (uamp) * 4.0;
    vec2 glitchOffset = vec2(
        sin(time_f * 10.0 + tc.y * 20.0) * glitchStrength,
        cos(time_f * 15.0 + tc.x * 25.0) * glitchStrength
    );
    vec2 distortedTc = tc + glitchOffset;
    color = texture(samp, distortedTc);
}
```

```

    vec4 originalColor = texture(samp, tc);
    float blendFactor = smoothstep(0.0, 0.1, glitchStrength);
    color = mix(originalColor, color, blendFactor);
    color = vec4(sin(color.rgb * time_f), 1.0);
}

```

Shader #7 - abc123

File: abc123.glsl

Uses distance fields with sinusoidal modulation to generate animated visual patterns. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation
 - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float uamp;
uniform float amp;
uniform vec4 iMouse;
void main(void) {
    vec2 mousePos = iMouse.xy / iResolution.xy;
    float dist = distance(tc, mousePos);
    float glitchStrength = exp(-dist * 10.0) * (uamp) * 4.0;
    vec2 glitchOffset = vec2(
        sin(time_f * 10.0 + tc.y * 20.0) * glitchStrength,
        cos(time_f * 15.0 + tc.x * 25.0) * glitchStrength
    );
    vec2 distortedTc = tc + glitchOffset;
    color = texture(samp, distortedTc);
    vec4 originalColor = texture(samp, tc);
    float blendFactor = smoothstep(0.0, 0.1, glitchStrength);
    color = mix(originalColor, color, blendFactor);
    color = vec4(sin(color.rgb * time_f), 1.0);
}

```

Shader #19 - air bowl

File: air-bowl.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control.

Technique: Multi-tap convolution blur (4 texture samples) - Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;

```

```

uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
float h1(float n){return fract(sin(n*91.345+37.12)*43758.5453123);}
vec2 h2(vec2 p){return fract(sin(vec2(dot(p,vec2(127.1,311.7)),dot(p,vec2(269.5,183.3))))*43758.5453);}
vec2 rot(vec2 v,float a){float c=cos(a),s=sin(a);return vec2(c*v.x-s*v.y,s*v.x+c*v.y);}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void){
    float a = clamp(amp,0.0,1.0);
    float ua = clamp(uamp,0.0,1.0);
    float t = time_f;
    vec2 center = vec2(0.5, 0.5);
    vec2 baseUV = tc;
    vec2 offset = baseUV - center;
    float maxRadius = length(vec2(0.5, 0.5));
    float radius = length(offset);
    float normalizedRadius = radius / maxRadius;
    float distortion = (0.25 + 0.45*ua + 0.3*a);
    float distortedRadius = normalizedRadius + distortion * normalizedRadius * normalizedRadius;
    distortedRadius = clamp(distortedRadius, 0.0, 1.0);
    distortedRadius *= maxRadius;
    vec2 distortedCoords = center + distortedRadius * (radius > 0.0 ? offset / radius : vec2(0.0));
    float spinSpeed = 0.6 + 1.8*(0.3 + 0.7*a);
    float modulatedTime = pingPong(t * spinSpeed, 5.0);
    float angle = atan(distortedCoords.y - center.y, distortedCoords.x - center.x) + modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle) * (distortedCoords.x - center.x) - sin(angle) * (distortedCoords.y - center
    ↪ .y) + center.x;
    rotatedTC.y = sin(angle) * (distortedCoords.x - center.x) + cos(angle) * (distortedCoords.y - center
    ↪ .y) + center.y;
    float warpAmp = 0.02 + 0.06*ua + 0.04*a;
    vec2 warpedCoords;
    warpedCoords.x = pingPong(rotatedTC.x + t * 0.12 * (1.0 + warpAmp*5.0), 1.0);
    warpedCoords.y = pingPong(rotatedTC.y + t * 0.12 * (1.0 + warpAmp*5.0), 1.0);
    vec2 uv = warpedCoords;
    float speedR=5.0, ampR=0.03, waveR=10.0;
    float speedG=6.5, ampG=0.025, waveG=12.0;
    float speedB=4.0, ampB=0.035, waveB=8.0;
    float rR=sin(uv.x*waveR+t*speedR)*ampR + sin(uv.y*waveR*0.8+t*speedR*1.2)*ampR;
    float rG=sin(uv.x*waveG*1.5+t*speedG)*ampG + sin(uv.y*waveG*0.3+t*speedG*0.7)*ampG;
    float rB=sin(uv.x*waveB*0.5+t*speedB)*ampB + sin(uv.y*waveB*1.7+t*speedB*1.3)*ampB;
    vec2 tcR=uv+vec2(rR,rR);
    vec2 tcG=uv+vec2(rG,-0.5*rG);
    vec2 tcB=uv+vec2(0.3*rB,rB);
    vec3 pats[4]=vec3[] (vec3(1,0,1),vec3(0,1,0),vec3(1,0,0),vec3(0,0,1));
    float pspd=4.0;
    int pidx=int(mod(floor(t*pspd+seed*4.0),4.0));
    vec3 mir=pats[pidx];
    vec2 m = iMouse.z>0.5 ? (iMouse.xy/iResolution) : fract(vec2(0.37+0.11*sin(t*0.63+seed),0.42+0.13*
    ↪ cos(t*0.57+seed*2.0)));
    vec2 dR=tcR-m, dG=tcG-m, dB=tcB-m;
    float fallR=smoothstep(0.55,0.0,length(dR));
    float fallG=smoothstep(0.55,0.0,length(dG));
    float fallB=smoothstep(0.55,0.0,length(dB));
    float sw=(0.12+0.38*ua+0.25*a);
    vec2 tangR=rot(normalize(dR+1e-4),1.5707963);
    vec2 tangG=rot(normalize(dG+1e-4),1.5707963);

```

```

vec2 tangB=rot(normalize(dB+1e-4),1.5707963);
vec2 airR=tangR*sw*fallR*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*40.0+t*3.0+seed));
vec2 airG=tangG*sw*fallG*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*38.0+t*3.3+seed*1.7));
vec2 airB=tangB*sw*fallB*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*42.0+t*2.9+seed*0.9));
vec2 jit = (h2(uv*vec2(233.3,341.9)+t+seed)-0.5)*(0.0006+0.004*ua);
tcR += airR + jit;
tcG += airG + jit;
tcB += airB + jit;
vec2 fR=vec2(mir.r>0.5?1.0-tcR.x:tcR.x, tcR.y);
vec2 fG=vec2(mir.g>0.5?1.0-tcG.x:tcG.x, tcG.y);
vec2 fB=vec2(mir.b>0.5?1.0-tcB.x:tcB.x, tcB.y);
float ca=0.0015+0.004*a;
vec4 C=texture(samp,uv);
C.r=texture(samp,fR+vec2(ca,0)).r;
C.g=texture(samp,fG+vec2(0,ca)).g;
C.b=texture(samp,fB+vec2(-ca,0)).b;
float pulse=0.004*(0.5+0.5*sin(t*3.7+seed));
C.rgb+=pulse*ua;
color=vec4(C.rgb,1.0);
}

```

Shader #24 - airshader1

File: airshader1.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control.

Technique: Color quantization via floor rounding - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
float h1(float n){return fract(sin(n*91.345+37.12)*43758.5453123);}
vec2 h2(vec2 p){return fract(sin(vec2(dot(p,vec2(127.1,311.7)),dot(p,vec2(269.5,183.3))))*43758.5453);}
vec2 rot(vec2 v,float a){float c=cos(a),s=sin(a);return vec2(c*v.x-s*v.y,s*v.x+c*v.y);}
void main(void){
    vec2 uv=tc;
    float a=clamp(amp,0.0,1.0);
    float ua=clamp(uamp,0.0,1.0);
    float t=time_f;
    float speedR=5.0, ampR=0.03, waveR=10.0;
    float speedG=6.5, ampG=0.025, waveG=12.0;
    float speedB=4.0, ampB=0.035, waveB=8.0;
    float rR=sin(uv.x*waveR+t*speedR)*ampR + sin(uv.y*waveR*0.8+t*speedR*1.2)*ampR;
    float rG=sin(uv.x*waveG*1.5+t*speedG)*ampG + sin(uv.y*waveG*0.3+t*speedG*0.7)*ampG;
    float rB=sin(uv.x*waveB*0.5+t*speedB)*ampB + sin(uv.y*waveB*1.7+t*speedB*1.3)*ampB;
    vec2 tcR=uv+vec2(rR,rR);
    vec2 tcG=uv+vec2(rG,-0.5*rG);
    vec2 tcB=uv+vec2(0.3*rB,rB);
    vec3 pats[4]=vec3[](vec3(1,0,1),vec3(0,1,0),vec3(1,0,0),vec3(0,0,1));
    float pspd=4.0;
}

```

```

int pidx=int(mod(floor(t*pspd+seed*4.0),4.0));
vec3 mir=pats[pidx];
vec2 m = iMouse.z>0.5 ? (iMouse.xy/iResolution) : fract(vec2(0.37+0.11*sin(t*0.63+seed),0.42+0.13*
↳ cos(t*0.57+seed*2.0)));
vec2 dR=tcR-m, dG=tcG-m, dB=tcB-m;
float fallR=smoothstep(0.55,0.0,length(dR));
float fallG=smoothstep(0.55,0.0,length(dG));
float fallB=smoothstep(0.55,0.0,length(dB));
float sw=(0.12+0.38*ua+0.25*a);
vec2 tangR=rot(normalize(dR+1e-4),1.5707963);
vec2 tangG=rot(normalize(dG+1e-4),1.5707963);
vec2 tangB=rot(normalize(dB+1e-4),1.5707963);
vec2 airR=tangR*sw*fallR*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*40.0+t*3.0+seed));
vec2 airG=tangG*sw*fallG*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*38.0+t*3.3+seed*1.7));
vec2 airB=tangB*sw*fallB*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*42.0+t*2.9+seed*0.9));
vec2 jit = (h2(uv*vec2(233.3,341.9)+t+seed)-0.5)*(0.0006+0.004*ua);
tcR += airR + jit;
tcG += airG + jit;
tcB += airB + jit;
vec2 fR=vec2(mir.r>0.5?1.0-tcR.x:tcR.x, tcR.y);
vec2 fG=vec2(mir.g>0.5?1.0-tcG.x:tcG.x, tcG.y);
vec2 fB=vec2(mir.b>0.5?1.0-tcB.x:tcB.x, tcB.y);
float ca=0.0015+0.004*a;
vec4 C=texture(samp,uv);
C.r=texture(samp,fR+vec2(ca,0)).r;
C.g=texture(samp,fG).g;
C.b=texture(samp,fB+vec2(-ca,0)).b;
float pulse=0.004*(0.5+0.5*sin(t*3.7+seed));
C.rgb+=pulse*ua;
color=vec4(C.rgb,1.0);
}

```

Shader #31 - animation

File: animation.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position and audio input (amplitude/volume) for interactive control. How fast the liquid moves Size of the swirls (lower = bigger swirls) Intensity of the melt effect

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Per-channel UV offset for chromatic aberration - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];

```

```

uniform float iSampleRate;
const float FLOW_SPEED = 0.5;
const float SWIRL_SCALE = 3.0;
const float DISTORTION = 0.2;
const float RGB_SPLIT = 0.02;
float random(vec2 st) {
    return fract(sin(dot(st.xy, vec2(12.9898, 78.233))) * 43758.5453123);
}
float noise(vec2 st) {
    vec2 i = floor(st);
    vec2 f = fract(st);
    f = f * f * (3.0 - 2.0 * f);
    float a = random(i);
    float b = random(i + vec2(1.0, 0.0));
    float c = random(i + vec2(0.0, 1.0));
    float d = random(i + vec2(1.0, 1.0));
    return mix(mix(a, b, f.x), mix(c, d, f.x), f.y);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
float fbm(vec2 st) {
    float value = 0.0;
    float amplitude = 0.5;
    for (int i = 0; i < 3; i++) {
        value += amplitude * noise(st);
        st *= 2.0;
        amplitude *= 0.5;
    }
    return value;
}
void main(void) {
    vec2 uv = tc;
    vec2 q = vec2(0.0);
    q.x = fbm(uv * SWIRL_SCALE + iTime * FLOW_SPEED * 0.1);
    q.y = fbm(uv * SWIRL_SCALE + vec2(1.0));
    vec2 r = vec2(0.0);
    r.x = fbm(uv * SWIRL_SCALE + 1.0 * q + vec2(1.7, 9.2) + 0.15 * iTime * FLOW_SPEED);
    r.y = fbm(uv * SWIRL_SCALE + 1.0 * q + vec2(8.3, 2.8) + 0.126 * iTime * FLOW_SPEED);
    vec2 distortedUV = uv + r * DISTORTION;
    float rChannel = texture(samp, distortedUV + vec2(0.0)).r;
    float gChannel = texture(samp, distortedUV).g;
    float bChannel = texture(samp, distortedUV - vec2(0.0)).b;
    vec3 finalColor = vec3(rChannel, gChannel, bChannel);
    float flowPattern = length(q);
    finalColor += vec3(0.1, 0.05, 0.2) * flowPattern * 0.5;
    color = vec4(finalColor, 1.0);
}

```

Shader #36 - audio1

File: audio1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Texcoords passed from vertex shader Time (in seconds, typically) The texture we're sampling

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
vec2 swirl(vec2 uv, float centerRadius, float swirlAmount) {
    float r = length(uv);
    float a = atan(uv.y, uv.x);
    float f = smoothstep(centerRadius, 0.0, r);
    a += swirlAmount * f;
    return r * vec2(cos(a), sin(a));
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void)
{
    float A = uamp * amp / time_f;
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    uv = swirl(uv, 0.8, 2.0 * A * sin(time_f * 0.5));
    uv.x /= aspect;
    uv = uv * 0.5 + 0.5;
    vec4 baseColor = texture(samp, uv);
    float pulseR = 0.3 * sin(time_f * 3.0 + A * 4.0);
    float pulseG = 0.3 * sin(time_f * 5.0 + A * 6.0);
    float pulseB = 0.3 * sin(time_f * 7.0 + A * 2.0);
    baseColor.r += pulseR;
    baseColor.g += pulseG;
    baseColor.b += pulseB;
    uv = swirl(uv - 0.5, 1.0, A * 10.0) + 0.5;
    baseColor *= texture(samp, uv);
    float t_amp = pingPong(amp, 8.0);
    color = sin(baseColor * (t_amp + uamp));
    color.a = 1.0;
}

```

Shader #37 - audio2

File: audio2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position and audio input (amplitude/volume) for interactive control. Passed-in texture coordinates Current time (seconds) Resolution (width, height)

Technique: 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;

```



```

uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 uv = tc - center;
    float swirlAngle = (time_f/amp) * length(uv) + time_f * uamp;
    float amp_t = pingPong(amp, 8.0) + 1.0;
    swirlAngle = sin(swirlAngle * amp_t);
    float s = sin(swirlAngle);
    float c = cos(swirlAngle);
    mat2 rotation = mat2(c, -s,
                        s,  c);
    uv = rotation * uv;
    uv += center;
    vec4 baseColor = texture(samp, uv);
    vec3 rainbow = 0.5 + 0.5 * cos( (uv.xy + time_f) * 10.0
                                   + vec3(0.0, 2.0, 4.0) );
    vec3 finalColor = mix(baseColor.rgb, rainbow, 0.3);
    color = vec4(finalColor, 1.0);
}

```

Shader #44 - aura5

File: aura5.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Multi-tap convolution blur (4 texture samples) - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){

```

```

    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
float hash21(vec2 p){
    p = fract(p*vec2(123.34, 456.21));
    p += dot(p, p*78.233);
    return fract(p.x*p.y);
}
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*0.35+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t*0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a*k1 + b*k2 + c*k3) * 1.05;
}
vec3 softTone(vec3 c){
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299,0.587,0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res){
    vec2 ts = 1.0 / res;

```

```

    vec3 s00 = textureGrad(img, uv + ts*vec2(-1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts*vec2( 0.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts*vec2( 1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts*vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts*vec2( 1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts*vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts*vec2( 0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts*vec2( 1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0*s10 + s20 + 2.0*s01 + 4.0*s11 + 2.0*s21 + s02 + 2.0*s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv, float seed, float drive){
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 gdir = normalize(vec2(cos(seed*0.27+drive), sin(seed*0.31-drive)));
    float s = dot((uv-0.5)*vec2(iResolution.x/iResolution.y,1.0), gdir);
    float w = max(fwidth(s)*(3.0+drive*0.5), 0.002);
    float band = smoothstep(-0.5-w, -0.5+w, sin(s*(2.2+drive*0.6) + seed*0.9));
    vec3 neon = neonPalette(seed + drive*0.25);
    vec3 grad = mix(tex, mix(tex, neon, 0.6), 0.35 + 0.25*band);
    grad = mix(grad, tex, 0.10);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec3 aces(vec3 x){
    const float a=2.51;
    const float b=0.03;
    const float c=2.43;
    const float d=0.59;
    const float e=0.14;
    x = (x*(a*x+b))/(x*(c*x+d)+e);
    return clamp(x,0.0,1.0);
}

void main(void){
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    float t = time_f + iTime*0.75;
    float daySeed = iDate.x*37.0 + iDate.y*17.0 + iDate.z*7.0 + iDate.w;
    float seed = t + daySeed*0.013;
    float fr = max(iFrameRate, 1.0);
    float dt = max(iTimeDelta, 1.0/fr);
    float sr = clamp(iSampleRate/48000.0, 0.25, 4.0);
    float aMix = clamp(amp*0.7 + uamp*0.3, 0.0, 20.0);
    float drive = clamp(aMix*0.08, 0.0, 2.5);
    float segBase = 4.0 + 2.0*sin(seed*0.33 + 0.15*sr);
    segBase += 2.0*(step(0.5, fract(iMouseClicked.x*0.5 + iMouseClicked.y*0.5)))*sin(seed*0.5);
    float seg = clamp(segBase + 2.0*sin(dot(iChannelResolution[0].xy, vec2(0.003,0.004))) + iChannelTime
    ↪ [0]*0.23), 3.0, 12.0);
    float zoom = 1.45 + 0.55 * sin(seed * (0.42+0.15*sr)) + 0.25*sin(iChannelTime[1]*0.31 + length(
    ↪ iChannelResolution[1].xy)*0.0007);

```

```

vec2 uv = tc*2.0-1.0;
uv.x *= aspect;
float beat = 0.5 + 0.5*sin(t*3.1 + drive*6.0);
float ringR = mix(0.9, 1.4, beat);
ringR *= 1.0 + 0.3*sin(iChannelTime[2]*0.7);
float r = length(uv);
float glow = smoothstep(ringR, ringR - 0.25, r);
vec3 baseCol = preBlendColor(tc, seed, drive);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
kUV = fractalFold(kUV, zoom, seed, m, aspect);
kUV = rotateUV(kUV, seed * (0.23 + 0.12*drive), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p); if(q.y > q.x) q = q.yx;
float base = 1.82 + 0.18*sin(seed*0.2);
float period = log(base);
float tz = seed * (0.65 + 0.2*sin(iChannelTime[3]*0.21));
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + seed*0.6 + drive*3.0);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap*(1.03+0.02*drive)) / ar + m);
vec2 u2 = fract((pwrap*(0.97-0.02*drive)) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
float chroma = (0.0012 + 0.0010*drive) * (1.0 + 0.3*sin(seed*1.7));
vec2 off = dir * chroma * vec2(1.0, 1.0/aspect);
float vign = 1.0 - smoothstep(0.72, 1.12, length((tc - m)*ar));
vign = mix(0.88, 1.18, vign);
vec3 rC = preBlendColor(u0 + off, seed+0.1, drive);
vec3 gC = preBlendColor(u1, seed+0.2, drive);
vec3 bC = preBlendColor(u2 - off, seed+0.3, drive);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD+1e-3)*(9.5+2.0*drive) + seed*1.2));
float pulse = 0.5 + 0.5*sin(seed*2.0 + rD*28.0 + k*12.0 + drive*8.0);
float shutter = smoothstep(0.0, 1.0, sin(PI*clamp(dt*fr*0.5,0.0,1.0)));
vec2 shake = (hash21(vec2(seed, rD))*2.0-1.0) * 0.0015 * shutter * vec2(1.0, 1.0/aspect);
vec3 outCol = kaleidoRGB;
outCol *= (0.70 + 0.30*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = outCol*outCol*(0.14+0.10*drive) + pow(max(outCol-0.6,0.0), vec3(2.0))*(0.10+0.08*drive)
↳ ;
outCol += bloom;
outCol = mix(outCol, baseCol, 0.16);
float dayGate = step(0.5, fract(daySeed*0.001 + sin(iDate.w+seed)*0.5+0.5));
outCol = mix(outCol, hsv2rgb(vec3(fract(seed*0.07 + rD*0.15), 0.9, 1.0)), 0.10*dayGate);
outCol = clamp(outCol, vec3(0.04), vec3(1.2));
vec3 src0 = texture(samp, tc + shake).rgb;
vec3 srcR = texture(samp, tc + off + shake).rgb;
vec3 srcB = texture(samp, tc - off + shake).rgb;
vec3 srcCA = vec3(srcR.r, src0.g, srcB.b);
float mixBase = pingPong(glow*PI, 5.0) * (0.75 + 0.25*drive);
float focus = smoothstep(0.0, 1.0, 1.0 - length((tc-m)*ar)*1.1);
vec3 combined = mix(srcCA, outCol, clamp(mixBase*focus, 0.0, 1.0));
combined = aces(combined);
color = vec4(combined, baseTex.a);
}

```

Shader #45 - aura6

File: aura6.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*0.35+float(i)))) - 0.5 + c;
```

```

        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t){
    vec3 pink   = vec3(1.0, 0.35, 0.85);
    vec3 blue    = vec3(0.20, 0.60, 1.0);
    vec3 green   = vec3(0.25, 1.00, 0.55);
    float ph = fract(t*0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a*k1 + b*k2 + c*k3);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res){
    vec2 ts = 1.0 / res;
    vec3 s00 = texture(img, uv + ts*vec2(-1.0,-1.0)).rgb;
    vec3 s10 = texture(img, uv + ts*vec2( 0.0,-1.0)).rgb;
    vec3 s20 = texture(img, uv + ts*vec2( 1.0,-1.0)).rgb;
    vec3 s01 = texture(img, uv + ts*vec2(-1.0, 0.0)).rgb;
    vec3 s11 = texture(img, uv).rgb;
    vec3 s21 = texture(img, uv + ts*vec2( 1.0, 0.0)).rgb;
    vec3 s02 = texture(img, uv + ts*vec2(-1.0, 1.0)).rgb;
    vec3 s12 = texture(img, uv + ts*vec2( 0.0, 1.0)).rgb;
    vec3 s22 = texture(img, uv + ts*vec2( 1.0, 1.0)).rgb;
    return (s00 + 2.0*s10 + s20 + 2.0*s01 + 4.0*s11 + 2.0*s21 + s02 + 2.0*s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv){
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec3 neon = neonPalette(time_f);
    return mix(tex, neon, 0.4);
}

float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec3 aces(vec3 x){
    const float a=2.51;
    const float b=0.03;
    const float c=2.43;
    const float d=0.59;
    const float e=0.14;
    x = (x*(a*x+b))/(x*(c*x+d)+e);
    return clamp(x,0.0,1.0);
}

void main(void){
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    float t = time_f + iTime*0.5;
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 3.0);

```

```

float pulse = 0.5 + 0.5*sin(t*2.0 + aMix*2.0);
float seg = 4.0 + 2.0*sin(t*0.33);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + 0.55 * sin(t * 0.42);
kUV = fractalFold(kUV, foldZoom, t, m, aspect);
kUV = rotateUV(kUV, t * 0.23, m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec3 baseCol = preBlendColor(tc);
vec3 glowCol = preBlendColor(kUV);
glowCol *= (0.85 + 0.15*pulse);
float auraDist = length((tc - m)*ar);
float auraStrength = smoothstep(0.6, 0.1, auraDist);
vec3 auraColor = vec3(1.0, 0.6, 0.85);
auraColor *= (0.6 + 0.4*sin(t*1.2 + amp));
auraColor *= 0.4;
vec3 outCol = mix(baseCol, glowCol, 0.7);
outCol += auraColor * auraStrength;
outCol = aces(outCol);
color = vec4(outCol, baseTex.a);
}

```

Shader #46 - aura7

File: aura7.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
float hash21(vec2 p){
    p = fract(p*vec2(123.34, 456.21));

```

```

    p += dot(p, p*78.233);
    return fract(p.x*p.y);
}
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*0.35+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.00, 0.20, 0.80);
    vec3 blue = vec3(0.10, 0.65, 1.00);
    vec3 green = vec3(0.15, 1.00, 0.55);
    float ph = fract(t*0.12);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a*k1 + b*k2 + c*k3);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res){
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts*vec2(-1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts*vec2( 0.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts*vec2( 1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts*vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts*vec2( 1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts*vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts*vec2( 0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts*vec2( 1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0*s10 + s20 + 2.0*s01 + 4.0*s11 + 2.0*s21 + s02 + 2.0*s12 + s22) / 16.0;
}

```



```

vec3 sharpen3(sampler2D img, vec2 uv, vec2 res){
    vec3 blur = tentBlur3(img, uv, res);
    vec3 base = texture(img, uv).rgb;
    return clamp(base*1.6 - blur*0.6, 0.0, 1.0);
}

vec3 preBlendColor(vec2 uv, float seed, float drive){
    vec3 tex = sharpen3(samp, uv, iResolution);
    vec3 neon = neonPalette(seed + drive*0.2);
    vec3 grad = mix(tex, mix(tex, neon, 0.75), 0.45);
    float l = dot(grad, vec3(0.299,0.587,0.114));
    float satAmt = 1.25 + drive*0.35;
    grad = mix(vec3(l), grad, satAmt);
    return clamp(grad, 0.0, 1.0);
}

float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec3 aces(vec3 x){
    const float a=2.51;
    const float b=0.03;
    const float c=2.43;
    const float d=0.59;
    const float e=0.14;
    x = (x*(a*x+b))/(x*(c*x+d)+e);
    return clamp(x,0.0,1.0);
}

vec3 contrastCurve(vec3 c, float k){
    c = clamp(c, 0.0, 1.0);
    return clamp((c-0.5)*(1.0+k)+0.5, 0.0, 1.0);
}

void main(void){
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 mouseN = iMouse.xy / iResolution;
    vec2 clickN = iMouseClick / iResolution;
    bool hasClick = max(iMouseClick.x, iMouseClick.y) > 0.0;
    vec2 mCore = (iMouse.z > 0.5) ? mouseN : (hasClick ? clickN : vec2(0.5));
    float t = time_f + iTime*0.75 + float(iFrame)*0.0005;
    float daySeed = iDate.x*37.0 + iDate.y*17.0 + iDate.z*7.0 + iDate.w;
    float seed = t + daySeed*0.013 + float(iFrame)*0.001;
    float fr = max(iFrameRate, 1.0);
    float dt = max(iTimeDelta, 1.0/fr);
    float sr = clamp(iSampleRate/48000.0, 0.25, 4.0);
    float aMix = clamp(amp*0.7 + uamp*0.3, 0.0, 20.0);
    float drive = clamp(aMix*0.10, 0.0, 3.0);
    vec2 uvCenter = (tc - mCore) * 2.0;
    uvCenter.x *= aspect;
    float beat = 0.5 + 0.5*sin(t*3.1 + drive*5.0);
    float ringR = mix(0.85, 1.35, beat*(0.85+0.15*sr));
    float r = length(uvCenter);
    float glow = smoothstep(ringR, ringR - 0.22, r);
    float seg = clamp(4.0 + 3.0*sin(seed*0.33 + 0.1*sr), 3.0, 12.0);
    float zoom = 1.55 + 0.65*sin(seed * (0.42+0.10*sr));

```

```

vec3 baseCol = preBlendColor(tc, seed, drive);
vec2 kUV = reflectUV(tc, seg, mCore, aspect);
kUV = diamondFold(kUV, mCore, aspect);
kUV = fractalFold(kUV, zoom, seed, mCore, aspect);
kUV = rotateUV(kUV, seed * (0.26 + 0.10*drive), mCore, aspect);
kUV = diamondFold(kUV, mCore, aspect);
vec2 p = (kUV - mCore) * ar;
vec2 q = abs(p); if(q.y > q.x) q = q.yx;
float base = 1.82 + 0.18*sin(seed*0.2);
float period = log(base);
float tz = seed * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + seed*0.6 + drive*3.0);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + mCore);
vec2 u1 = fract((pwrap*(1.03+0.02*drive)) / ar + mCore);
vec2 u2 = fract((pwrap*(0.97-0.02*drive)) / ar + mCore);
vec2 dir = normalize(pwrap + 1e-6);
float chroma = (0.0016 + 0.0012*drive) * (1.0 + 0.25*sin(seed*1.4));
vec2 off = dir * chroma * vec2(1.0, 1.0/aspect);
float vign = 1.0 - smoothstep(0.78, 1.18, length((tc - mCore)*ar));
vign = mix(0.92, 1.22, vign);
vec3 rC = preBlendColor(u0 + off, seed+0.1, drive);
vec3 gC = preBlendColor(u1, seed+0.2, drive);
vec3 bC = preBlendColor(u2 - off, seed+0.3, drive);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD+1e-3)*(9.5+2.0*drive) + seed*1.2));
float pulse = 0.5 + 0.5*sin(seed*2.2 + rD*28.0 + k*12.0 + drive*8.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.72 + 0.28*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = outCol*outCol*(0.18+0.10*drive) + pow(max(outCol-0.55,0.0), vec3(2.0))*(0.14+0.10*drive
↳ );
outCol += bloom;
float exposure = 1.25 + 0.35*drive;
outCol *= exposure;
outCol = contrastCurve(outCol, 0.45 + 0.25*drive);
float l = dot(outCol, vec3(0.299,0.587,0.114));
outCol = mix(vec3(l), outCol, 1.35 + 0.25*drive);
outCol = clamp(outCol, 0.0, 3.0);
outCol = aces(outCol);
color = vec4(mix(baseTex.rgb, outCol, 0.85), baseTex.a);
}

```

Shader #47 - aura8

File: aura8.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Spiral twist field driven by texture luminance and audio Multitap echo that samples along the spiral at expanding radii

Technique: Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;

```

```

in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
float luma(vec3 c){ return dot(c, vec3(0.299, 0.587, 0.114)); }
vec3 neonPalette(float t){
    vec3 a = vec3(1.00, 0.20, 0.80);
    vec3 b = vec3(0.10, 0.65, 1.00);
    vec3 c = vec3(0.15, 1.00, 0.55);
    float ph = fract(t*0.12);
    vec3 k1 = mix(a, b, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(b, c, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(c, a, smoothstep(0.66, 1.00, ph));
    float s1 = step(ph, 0.33);
    float s2 = step(0.33, ph) * step(ph, 0.66);
    float s3 = step(0.66, ph);
    vec3 outc = s1*k1 + s2*k2 + s3*k3;
    return normalize(outc + 1e-6);
}
vec3 tentBlur3RGB(sampler2D img, vec2 uv, vec2 res){
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts*vec2(-1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts*vec2( 0.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts*vec2( 1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts*vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;

```

```

    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts*vec2( 1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts*vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts*vec2( 0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts*vec2( 1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0*s10 + s20 + 2.0*s01 + 4.0*s11 + 2.0*s21 + s02 + 2.0*s12 + s22) / 16.0;
}

vec3 aces(vec3 x){
    const float a=2.51;
    const float b=0.03;
    const float c=2.43;
    const float d=0.59;
    const float e=0.14;
    x = (x*(a*x+b))/(x*(c*x+d)+e);
    return clamp(x,0.0,1.0);
}

vec3 contrastCurve(vec3 c, float k){
    c = clamp(c, 0.0, 1.0);
    return clamp((c-0.5)*(1.0+k)+0.5, 0.0, 1.0);
}

vec2 spiralTwist(vec2 uv, vec2 center, float aspect, float t, float drive){
    vec2 p = uv - center;
    p.x *= aspect;
    float r = length(p)+1e-6;
    float a = atan(p.y, p.x);
    vec3 tex = textureGrad(samp, uv, dFdx(uv), dFdy(uv)).rgb;
    float lum = luma(tex);
    float swirl = (0.65 + 0.85*drive) * (1.0 + 0.6*lum) / (1.0 + 8.0*r);
    float spin = (0.35 + 0.25*drive) * t;
    a += spin + swirl;
    float pull = (0.12 + 0.25*drive*lum) / (1.0 + 6.0*r);
    r = max(r - pull, 0.0);
    vec2 q = vec2(cos(a), sin(a)) * r;
    q.x /= aspect;
    return q + center;
}

vec3 spiralEcho(vec2 uv, vec2 center, float aspect, float t, float drive){
    const int TAPS = 7;
    vec3 acc = vec3(0.0);
    float wsum = 0.0;
    float baseHue = fract(t*0.05);
    vec3 neon = neonPalette(t*0.6);
    for(int i=0;i<TAPS;i++){
        float k = float(i);
        float radScale = 1.0 + 0.08*k + 0.05*drive*k;
        float timeLag = 0.08*k;
        vec2 u = spiralTwist(uv, center, aspect, t - timeLag, drive);
        u = rotateUV(u, (0.12+0.06*drive)*k, center, aspect);
        vec2 dir = (u-center);
        dir.x *= aspect;
        float r = length(dir)+1e-6;
        float chroma = (0.0015 + 0.0012*drive) * (0.8 + 0.6*exp(-r*3.0));
        vec2 off = normalize(dir) * chroma * vec2(1.0, 1.0/aspect);
        vec3 sR = textureGrad(samp, fract(mix(uv, u, 0.85)/radScale + off), dFdx(uv), dFdy(uv)).rgb;
        vec3 sG = textureGrad(samp, fract(mix(uv, u, 0.85)/radScale
        ↪      ), dFdx(uv), dFdy(uv)).rgb;
        vec3 sB = textureGrad(samp, fract(mix(uv, u, 0.85)/radScale - off), dFdx(uv), dFdy(uv)).rgb;
        vec3 s = vec3(sR.r, sG.g, sB.b);
        float lum = luma(s);
        vec3 colorize = mix(s, s*neon, 0.35 + 0.25*drive*lum);
        float w = 1.0 / (1.0 + 0.9*k);
        acc += colorize * w;
    }
}

```

```

        wsum += w;
    }
    return acc / max(wsum, 1e-6);
}

void main(void){
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 mouseN = iMouse.xy / iResolution;
    vec2 clickN = iMouseClick / iResolution;
    bool hasClick = max(iMouseClick.x, iMouseClick.y) > 0.0;
    vec2 center = (iMouse.z > 0.5) ? mouseN : (hasClick ? clickN : vec2(0.5));
    float t = time_f + iTime*0.75 + float(iFrame)*0.0005;
    float daySeed = iDate.x*37.0 + iDate.y*17.0 + iDate.z*7.0 + iDate.w;
    t += daySeed*0.013;
    float sr = clamp(iSampleRate/48000.0, 0.25, 4.0);
    float aMix = clamp(amp*0.7 + uamp*0.3, 0.0, 20.0);
    float drive = clamp(aMix*0.10, 0.0, 3.0);
    float seg = clamp(4.0 + 3.0*sin(t*0.33 + 0.1*sr), 3.0, 12.0);
    vec2 kUV = reflectUV(tc, seg, center, aspect);
    vec3 echoCol = spiralEcho(kUV, center, aspect, t, drive);
    vec3 local = tentBlur3RGB(samp, tc, iResolution);
    float mixDetail = 0.25 + 0.25*sin(t*0.7 + drive*0.8);
    vec3 mixCol = mix(local, echoCol, 0.65 + mixDetail);
    float r = length((tc - center)*ar);
    float ring = smoothstep(0.95, 0.45, r + 0.05*sin(8.0*r - t*2.0 - drive*3.0));
    vec3 glow = neonPalette(t) * (0.35 + 0.35*pingPong(t + drive, 2.5));
    vec3 outCol = mixCol + glow*ring;
    outCol = contrastCurve(outCol, 0.55 + 0.25*drive);
    float exposure = 1.25 + 0.35*drive;
    outCol *= exposure;
    outCol = clamp(outCol, 0.0, 3.0);
    outCol = aces(outCol);
    float focus = smoothstep(0.0, 1.0, 1.0 - r*1.05);
    vec3 finalRGB = mix(baseTex.rgb, outCol, 0.70 + 0.25*focus);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #48 - aura9

File: aura9.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Spiral twist driven by luminance + audio Multitap spiral echo Radial ripple (from your second shader), now centered at the movable sphere and content/audio aware

Technique: Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;

```

```

uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
float luma(vec3 c){ return dot(c, vec3(0.299, 0.587, 0.114)); }
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec3 tentBlur3RGB(sampler2D img, vec2 uv, vec2 res){
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts*vec2(-1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts*vec2( 0.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts*vec2( 1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts*vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts*vec2( 1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts*vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts*vec2( 0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts*vec2( 1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0*s10 + s20 + 2.0*s01 + 4.0*s11 + 2.0*s21 + s02 + 2.0*s12 + s22) / 16.0;
}
vec3 neonPalette(float t){
    vec3 a = vec3(1.00, 0.20, 0.80);
    vec3 b = vec3(0.10, 0.65, 1.00);
    vec3 c = vec3(0.15, 1.00, 0.55);
    float ph = fract(t*0.12);
    vec3 k1 = mix(a, b, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(b, c, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(c, a, smoothstep(0.66, 1.00, ph));
    float s1 = step(ph, 0.33);
    float s2 = step(0.33, ph) * step(ph, 0.66);
    float s3 = step(0.66, ph);
    vec3 outc = s1*k1 + s2*k2 + s3*k3;
    return normalize(outc + 1e-6);
}

```

```

}
vec3 aces(vec3 x){
    const float a=2.51;
    const float b=0.03;
    const float c=2.43;
    const float d=0.59;
    const float e=0.14;
    x = (x*(a*x+b))/(x*(c*x+d)+e);
    return clamp(x,0.0,1.0);
}
vec3 contrastCurve(vec3 c, float k){
    c = clamp(c, 0.0, 1.0);
    return clamp((c-0.5)*(1.0+k)+0.5, 0.0, 1.0);
}
vec2 spiralTwist(vec2 uv, vec2 center, float aspect, float t, float drive){
    vec2 p = uv - center;
    p.x *= aspect;
    float r = length(p)+1e-6;
    float a = atan(p.y, p.x);
    vec3 tex = textureGrad(samp, uv, dFdx(uv), dFdy(uv)).rgb;
    float lum = luma(tex);
    float swirl = (0.65 + 0.85*drive) * (1.0 + 0.6*lum) / (1.0 + 8.0*r);
    float spin = (0.35 + 0.25*drive) * t;
    a += spin + swirl;
    float pull = (0.12 + 0.25*drive*lum) / (1.0 + 6.0*r);
    r = max(r - pull, 0.0);
    vec2 q = vec2(cos(a), sin(a)) * r;
    q.x /= aspect;
    return q + center;
}
vec3 spiralEcho(vec2 uv, vec2 center, float aspect, float t, float drive){
    const int TAPS = 7;
    vec3 acc = vec3(0.0);
    float wsum = 0.0;
    vec3 neon = neonPalette(t*0.6);
    for(int i=0;i<TAPS;i++){
        float k = float(i);
        float radScale = 1.0 + 0.08*k + 0.05*drive*k;
        float timeLag = 0.08*k;
        vec2 u = spiralTwist(uv, center, aspect, t - timeLag, drive);
        u = rotateUV(u, (0.12+0.06*drive)*k, center, aspect);
        vec2 dir = (u-center);
        dir.x *= aspect;
        float r = length(dir)+1e-6;
        float chroma = (0.0015 + 0.0012*drive) * (0.8 + 0.6*exp(-r*3.0));
        vec2 off = normalize(dir) * chroma * vec2(1.0, 1.0/aspect);
        vec3 sR = textureGrad(samp, fract(mix(uv, u, 0.85)/radScale + off), dFdx(uv), dFdy(uv)).rgb;
        vec3 sG = textureGrad(samp, fract(mix(uv, u, 0.85)/radScale - off), dFdx(uv), dFdy(uv)).rgb;
        vec3 sB = textureGrad(samp, fract(mix(uv, u, 0.85)/radScale - off), dFdx(uv), dFdy(uv)).rgb;
        vec3 s = vec3(sR.r, sG.g, sB.b);
        float lum = luma(s);
        vec3 colorize = mix(s, s*neon, 0.35 + 0.25*drive*lum);
        float w = 1.0 / (1.0 + 0.9*k);
        acc += colorize * w;
        wsum += w;
    }
    return acc / max(wsum, 1e-6);
}
vec2 rippleDisplace(vec2 uv, vec2 center, float aspect, float t, float drive){
    vec2 d = uv - center;
    float dist = length(d);
    vec2 dir = d / max(dist, 1e-6);

```

```

float baseWave = 0.05;
float baseAmp  = 0.02;
float speed    = 2.0;
float texLum   = luma(texture(samp, uv).rgb);
float waveLength = baseWave * (0.8 + 0.6*texLum);
float amplitude = baseAmp * (0.7 + 0.8*drive);
float ripple = sin((dist / max(waveLength,1e-4) - t * speed) * 6.2831853);
float window = smoothstep(0.95, 0.15, dist);
ripple *= window;
return uv + dir * ripple * amplitude;
}

void main(void){
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 mouseN = iMouse.xy / iResolution;
    vec2 clickN = iMouseClicked / iResolution;
    bool hasClick = max(iMouseClicked.x, iMouseClicked.y) > 0.0;
    vec2 center = (iMouse.z > 0.5) ? mouseN : (hasClick ? clickN : vec2(0.5));
    float t = time_f + iTime*0.75 + float(iFrame)*0.0005;
    float daySeed = iDate.x*37.0 + iDate.y*17.0 + iDate.z*7.0 + iDate.w;
    t += daySeed*0.013;
    float sr = clamp(iSampleRate/48000.0, 0.25, 4.0);
    float aMix = clamp(amp*0.7 + uamp*0.3, 0.0, 20.0);
    float drive = clamp(aMix*0.10, 0.0, 3.0);
    vec2 uvRipple = rippleDisplace(tc, center, aspect, t, drive);
    float seg = clamp(4.0 + 3.0*sin(t*0.33 + 0.1*sr), 3.0, 12.0);
    vec2 kUV = reflectUV(uvRipple, seg, center, aspect);
    vec3 echoCol = spiralEcho(kUV, center, aspect, t, drive);
    vec3 local = tentBlur3RGB(samp, uvRipple, iResolution);
    float mixDetail = 0.25 + 0.25*sin(t*0.7 + drive*0.8);
    vec3 mixCol = mix(local, echoCol, 0.65 + mixDetail);
    float r = length((tc - center)*ar);
    float ring = smoothstep(0.95, 0.45, r + 0.05*sin(8.0*r - t*2.0 - drive*3.0));
    vec3 glow = neonPalette(t) * (0.35 + 0.35*pingPong(t + drive, 2.5));
    vec3 outCol = mixCol + glow*ring;
    outCol = contrastCurve(outCol, 0.55 + 0.25*drive);
    float exposure = 1.25 + 0.35*drive;
    outCol *= exposure;
    outCol = clamp(outCol, 0.0, 3.0);
    outCol = aces(outCol);
    float focus = smoothstep(0.0, 1.0, 1.0 - r*1.05);
    vec3 finalRGB = mix(baseTex.rgb, outCol, 0.70 + 0.25*focus);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #49 - auraXi1

File: auraXi1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```



```

uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform vec2 iMouseClicked;
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
float luma(vec3 c){ return dot(c, vec3(0.299, 0.587, 0.114)); }
vec3 hue(float t){
    return clamp(vec3(abs(t*6.0-3.0)-1.0, 2.0-abs(t*6.0-2.0), 2.0-abs(t*6.0-4.0)),0.0,1.0);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 rippleDisplace(vec2 uv, vec2 center, float aspect, float t, float drive){
    vec2 d = uv - center;
    float dist = length(d);
    vec2 dir = d / max(dist, 1e-6);
    float baseWave = 0.05;
    float baseAmp = 0.02;
    float speed = 2.0;
    float texLum = luma(texture(samp, uv).rgb);
    float waveLength = baseWave * (0.8 + 0.6*texLum);
    float amplitude = baseAmp * (0.7 + 0.8*drive);
    float ripple = sin((dist / max(waveLength,1e-4) - t * speed) * 6.2831853);
    ripple *= smoothstep(0.95, 0.15, dist);
    return uv + dir * ripple * amplitude;
}
vec2 spiralTwist(vec2 uv, vec2 center, float aspect, float t, float drive){
    vec2 p = uv - center;
    p.x *= aspect;
    float r = length(p)+1e-6;
    float a = atan(p.y, p.x);
    float swirl = (0.7 + 0.6*drive) / (1.0 + 8.0*r);
    float spin = (0.35 + 0.25*drive) * t;
    a += spin + swirl;
    float pull = 0.12 / (1.0 + 6.0*r);
    r = max(r - pull, 0.0);
    vec2 q = vec2(cos(a), sin(a)) * r;
    q.x /= aspect;
    return q + center;
}
vec3 spiralEcho(vec2 uv, vec2 center, float aspect, float t, float drive){
    const int TAPS = 6;
    vec3 acc = vec3(0.0);
    float wsum = 0.0;
    for(int i=0;i<TAPS;i++){
        float k = float(i);
        float radScale = 1.0 + 0.09*k;

```

```

        float timeLag = 0.1*k;
        vec2 u = spiralTwist(uv, center, aspect, t - timeLag, drive);
        u = rotateUV(u, (0.15+0.05*drive)*k, center, aspect);
        vec3 s = texture(samp, fract(u / radScale)).rgb;
        float w = 1.0 / (1.0 + 0.8*k);
        acc += s * w;
        wsum += w;
    }
    return acc / max(wsum, 1e-6);
}

void main(void){
    vec2 mouseN = iMouse.xy / iResolution;
    vec2 clickN = iMouseClicked / iResolution;
    bool hasClick = max(iMouseClicked.x, iMouseClicked.y) > 0.0;
    vec2 center = (iMouse.z > 0.5) ? mouseN : (hasClick ? clickN : vec2(0.5));
    float t = time_f + iTime*0.75 + float(iFrame)*0.0005;
    float aMix = clamp(amp*0.7 + uamp*0.3, 0.0, 20.0);
    float drive = clamp(aMix*0.10, 0.0, 3.0);
    float aspect = iResolution.x / iResolution.y;
    vec2 uvRipple = rippleDisplace(tc, center, aspect, t, drive);
    vec3 echoCol = spiralEcho(uvRipple, center, aspect, t, drive);
    vec2 m = (iMouse.z > 0.0) ? (iMouse.xy / iResolution.xy) : vec2(0.5);
    float a = time_f*0.6;
    vec2 dir = normalize(vec2(cos(a), sin(a)));
    float span = 0.25 + 0.15*sin(time_f*0.7);
    vec2 startP = m - dir*span;
    vec2 endP = m + dir*span;
    vec2 ab = endP - startP;
    float L = max(length(ab), 1e-4);
    vec2 u = ab / L;
    float gradT = clamp(dot(tc - startP, u) / L, 0.0, 1.0);
    vec3 c0 = hue(fract(time_f*0.1));
    vec3 c1 = hue(fract(time_f*0.1 + 0.5));
    vec3 grad = mix(c0, c1, gradT);
    vec3 tex = texture(samp, uvRipple).rgb;
    vec3 mixed = mix(tex, echoCol, 0.6);
    mixed = mix(mixed, mixed * grad, clamp(amp, 0.0, 1.0));
    mixed = pow(abs(sin(mixed * (1.5 + 0.5*uamp))), vec3(1.0));
    mixed = clamp(mixed * (1.3 + 0.3*drive), 0.0, 1.0);
    color = vec4(mixed, 1.0);
}

```

Shader #50 - auraXi2

File: auraXi2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;

```

```

uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform vec2 iMouseClicked;
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
float luma(vec3 c){ return dot(c, vec3(0.2126, 0.7152, 0.0722)); }
vec3 hue(float t){
    return clamp(vec3(abs(t*6.0-3.0)-1.0, 2.0-abs(t*6.0-2.0), 2.0-abs(t*6.0-4.0)),0.0,1.0);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 rippleDisplace(vec2 uv, vec2 center, float aspect, float t, float drive){
    vec2 d = uv - center;
    float dist = length(d);
    vec2 dir = d / max(dist, 1e-6);
    float baseWave = 0.055;
    float baseAmp = 0.018;
    float speed = 1.35;
    vec3 tx = textureGrad(samp, uv, dFdx(uv), dFdy(uv)).rgb;
    float waveLength = baseWave * (0.9 + 0.5*luma(tx));
    float amplitude = baseAmp * (0.75 + 0.5*drive);
    float ripple = sin((dist / max(waveLength,1e-4) - t * speed) * 6.2831853);
    ripple *= smoothstep(0.95, 0.2, dist);
    return uv + dir * ripple * amplitude;
}
vec2 spiralTwist(vec2 uv, vec2 center, float aspect, float t, float drive){
    vec2 p = uv - center;
    p.x *= aspect;
    float r = length(p)+1e-6;
    float a = atan(p.y, p.x);
    float swirl = (0.55 + 0.5*drive) / (1.0 + 8.0*r);
    float spin = (0.22 + 0.18*drive) * t;
    a += spin + swirl;
    float pull = 0.10 / (1.0 + 6.0*r);
    r = max(r - pull, 0.0);
    vec2 q = vec2(cos(a), sin(a)) * r;
    q.x /= aspect;
    return q + center;
}
vec3 spiralEcho(vec2 uv, vec2 center, float aspect, float t, float drive){
    const int TAPS = 6;
    vec3 acc = vec3(0.0);
    float wsum = 0.0;
    for(int i=0;i<TAPS;i++){
        float k = float(i);
        float radScale = 1.0 + 0.085*k;
        float timeLag = 0.11*k;
        vec2 u = spiralTwist(uv, center, aspect, t - timeLag, drive);
        u = rotateUV(u, (0.12+0.04*drive)*k, center, aspect);
        vec3 s = textureGrad(samp, fract(u / radScale), dFdx(uv), dFdy(uv)).rgb;

```

```

        float w = 1.0 / (1.0 + 0.85*k);
        acc += s * w;
        wsum += w;
    }
    return acc / max(wsum, 1e-6);
}
vec3 reinhard(vec3 x){ return x/(1.0+x); }
vec3 softLimit(vec3 c, float maxY){
    float Y = luma(c);
    if(Y > maxY) c *= maxY/(Y+1e-6);
    return c;
}
void main(void){
    vec2 mouseN = iMouse.xy / iResolution;
    vec2 clickN = iMouseClicked / iResolution;
    bool hasClick = max(iMouseClicked.x, iMouseClicked.y) > 0.0;
    vec2 center = (iMouse.z > 0.5) ? mouseN : (hasClick ? clickN : vec2(0.5));
    float t = (time_f + iTime*0.6) + float(iFrame)*0.0003;
    float aMix = clamp(amp*0.7 + uamp*0.3, 0.0, 12.0);
    float drive = tanh(aMix*0.07);
    float aspect = iResolution.x / iResolution.y;
    vec2 uvRipple = rippleDisplace(tc, center, aspect, t, drive);
    vec3 echoCol = spiralEcho(uvRipple, center, aspect, t, drive);
    vec2 m = (iMouse.z > 0.0) ? (iMouse.xy / iResolution.xy) : vec2(0.5);
    float a = time_f*0.45;
    vec2 dir = normalize(vec2(cos(a), sin(a)));
    float span = 0.22 + 0.10*sin(time_f*0.35);
    vec2 startP = m - dir*span;
    vec2 endP = m + dir*span;
    vec2 ab = endP - startP;
    float L = max(length(ab), 1e-4);
    vec2 u = ab / L;
    float gradT = clamp(dot(tc - startP, u) / L, 0.0, 1.0);
    vec3 c0 = hue(fract(time_f*0.08));
    vec3 c1 = hue(fract(time_f*0.08 + 0.5));
    vec3 grad = mix(c0, c1, gradT);
    vec3 tex = textureGrad(samp, uvRipple, dFdx(uvRipple), dFdy(uvRipple)).rgb;
    vec3 mixed = mix(tex, echoCol, 0.6);
    float colorMix = 0.55*clamp(amp, 0.0, 1.0);
    mixed = mix(mixed, mixed * grad, colorMix);
    float satAmt = 1.08 + 0.18*drive;
    float Y = luma(mixed);
    mixed = mix(vec3(Y), mixed, satAmt);
    mixed *= 1.08 + 0.15*drive;
    mixed = softLimit(mixed, 0.9);
    mixed = reinhard(mixed);
    mixed = pow(clamp(mixed,0.0,1.0), vec3(0.95));
    color = vec4(mixed, 1.0);
}

```

Shader #51 - auraXi3

File: auraXi3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform vec2 iMouseClicked;
uniform float iSampleRate;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
float luma(vec3 c){ return dot(c, vec3(0.2126, 0.7152, 0.0722)); }
vec3 hue(float t){
    return clamp(vec3(abs(t*6.0-3.0)-1.0, 2.0-abs(t*6.0-2.0), 2.0-abs(t*6.0-4.0)),0.0,1.0);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 rippleDisplace(vec2 uv, vec2 center, float aspect, float t, float drive){
    vec2 d = uv - center;
    float dist = length(d);
    vec2 dir = d / max(dist, 1e-6);
    float baseWave = 0.055;
    float baseAmp = 0.018;
    float speed = 1.2;
    vec3 tx = textureGrad(samp, uv, dFdx(uv), dFdy(uv)).rgb;
    float waveLength = baseWave * (0.9 + 0.5*luma(tx));
    float amplitude = baseAmp * (0.75 + 0.5*drive);
    float ripple = sin((dist / max(waveLength,1e-4) - t * speed) * 6.2831853);
    ripple *= smoothstep(0.95, 0.22, dist);
    return uv + dir * ripple * amplitude;
}
vec2 spiralTwist(vec2 uv, vec2 center, float aspect, float t, float drive){
    vec2 p = uv - center;
    p.x *= aspect;
    float r = length(p)+1e-6;
    float a = atan(p.y, p.x);
    float swirl = (0.5 + 0.45*drive) / (1.0 + 8.0*r);
    float spin = (0.18 + 0.16*drive) * t;
    a += spin + swirl;
    float pull = 0.09 / (1.0 + 6.0*r);
    r = max(r - pull, 0.0);
    vec2 q = vec2(cos(a), sin(a)) * r;
    q.x /= aspect;
    return q + center;
}
vec3 spiralEcho(vec2 uv, vec2 center, float aspect, float t, float drive){
    const int TAPS = 6;
    vec3 acc = vec3(0.0);
    float wsum = 0.0;

```

```

    for(int i=0;i<TAPS;i++){
        float k = float(i);
        float radScale = 1.0 + 0.085*k;
        float timeLag = 0.11*k;
        vec2 u = spiralTwist(uv, center, aspect, t - timeLag, drive);
        u = rotateUV(u, (0.10+0.04*drive)*k, center, aspect);
        vec3 s = textureGrad(samp, fract(u / radScale), dFdx(uv), dFdy(uv)).rgb;
        float w = 1.0 / (1.0 + 0.85*k);
        acc += s * w;
        wsum += w;
    }
    return acc / max(wsum, 1e-6);
}

vec3 reinhard(vec3 x){ return x/(1.0+x); }
vec3 softLimit(vec3 c, float maxY){
    float Y = luma(c);
    if(Y > maxY) c *= maxY/(Y+1e-6);
    return c;
}

vec3 satBoost(vec3 c, float s){
    float Y = luma(c);
    return mix(vec3(Y), c, 1.0 + s);
}

void main(void){
    vec2 mouseN = iMouse.xy / iResolution;
    vec2 clickN = iMouseClicked / iResolution;
    bool hasClick = max(iMouseClicked.x, iMouseClicked.y) > 0.0;
    vec2 center = (iMouse.z > 0.5) ? mouseN : (hasClick ? clickN : vec2(0.5));
    float t = (time_f + iTime*0.6) + float(iFrame)*0.00025;
    float aMix = clamp(amp*0.7 + uamp*0.3, 0.0, 12.0);
    float drive = tanh(aMix*0.07);
    float aspect = iResolution.x / iResolution.y;
    vec2 uvRipple = rippleDisplace(tc, center, aspect, t, drive);
    vec3 echoCol = spiralEcho(uvRipple, center, aspect, t, drive);
    vec3 texCol = textureGrad(samp, uvRipple, dFdx(uvRipple), dFdy(uvRipple)).rgb;
    vec3 mixed = mix(texCol, echoCol, 0.62);
    float a = time_f*0.12;
    vec2 dir = normalize(vec2(cos(a), sin(a)));
    float span = 0.30;
    vec2 startP = center - dir*span;
    vec2 endP = center + dir*span;
    vec2 ab = endP - startP;
    float L = max(length(ab), 1e-4);
    vec2 u = ab / L;
    float g = clamp(dot(tc - startP, u) / L, 0.0, 1.0);
    float slowHue = fract(0.08*time_f);
    float slowHue2 = fract(0.08*time_f + 0.33);
    vec3 g0 = hue(slowHue);
    vec3 g1 = hue(slowHue2);
    vec3 gradCol = mix(g0, g1, smoothstep(0.0,1.0,g));
    float ring = smoothstep(0.9, 0.2, length((tc-center)*vec2(aspect,1.0)));
    float colorAmt = 0.45 + 0.25*drive;
    vec3 colored = mix(mixed, mixed*(gradCol*(0.85+0.15*ring)), colorAmt);
    colored = satBoost(colored, 0.35 + 0.25*drive);
    colored *= 1.08 + 0.12*drive;
    colored = softLimit(colored, 0.88);
    colored = reinhard(colored);
    colored = pow(clamp(colored,0.0,1.0), vec3(0.96));
    color = vec4(colored, 1.0);
}

```

Shader #62 - black rings

File: black_rings.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position and audio input (amplitude/volume) for interactive control. Applies matrix transformations.

Technique: Multi-tap convolution blur (4 texture samples) - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI=3.1415926535897932384626433832795;
float pingPong(float x,float l){float m=mod(x,l*2.0);return m<=l?m:l*2.0-m;}
float hash(float n){return fract(sin(n)*43758.5453123);}
float n2(vec2 p){
    vec2 i=floor(p),f=fract(p);
    float a=hash(i.x+i.y*57.0);
    float b=hash(i.x+1.0+i.y*57.0);
    float c=hash(i.x+(i.y+1.0)*57.0);
    float d=hash(i.x+1.0+(i.y+1.0)*57.0);
    vec2 u=f*f*(3.0-2.0*f);
    return mix(mix(a,b,u.x),mix(c,d,u.x),u.y);
}
float fbm(vec2 p){
    float s=0.0,a=0.5;
    for(int i=0;i<5;i++){s+=a*n2(p);p*=2.02;a*=0.5;}
    return s;
}
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
vec2 rot(vec2 p,float a){float s=sin(a),c=cos(a);return mat2(c,-s,s,c)*p;}
vec2 kale(vec2 uv,float seg,vec2 c,float aspect){
    vec2 p=uv-c;p.x*=aspect;
    float a=atan(p.y,p.x),r=length(p);
    float stepA=2.0*PI/seg;
    a=mod(a,stepA);
    a=abs(a-stepA*0.5);
    vec2 q=vec2(cos(a),sin(a))*r;
```

```

    q.x/=aspect;
    return q+c;
}
vec3 tentBlur3(sampler2D img,vec2 uv,vec2 res){
    vec2 ts=1.0/res;
    vec3 s00=texture(img,uv+ts*vec2(-1.0,-1.0)).rgb;
    vec3 s10=texture(img,uv+ts*vec2(0.0,-1.0)).rgb;
    vec3 s20=texture(img,uv+ts*vec2(1.0,-1.0)).rgb;
    vec3 s01=texture(img,uv+ts*vec2(-1.0,0.0)).rgb;
    vec3 s11=texture(img,uv).rgb;
    vec3 s21=texture(img,uv+ts*vec2(1.0,0.0)).rgb;
    vec3 s02=texture(img,uv+ts*vec2(-1.0,1.0)).rgb;
    vec3 s12=texture(img,uv+ts*vec2(0.0,1.0)).rgb;
    vec3 s22=texture(img,uv+ts*vec2(1.0,1.0)).rgb;
    return (s00+2.0*s10+s20+2.0*s01+4.0*s11+2.0*s21+s02+2.0*s12+s22)/16.0;
}
void main(){
    vec2 R=iResolution;
    float aspect=R.x/max(R.y,1.0);
    vec2 uv=tc;
    vec2 cuv=(uv-0.5)*vec2(aspect,1.0);
    float t=time_f+iTime+float(iFrame)*max(iTimeDelta,0.0);
    float fr=max(iFrameRate,1.0);
    float sr=clamp(iSampleRate/48000.0,0.25,4.0);
    float dateSeed=dot(iDate,vec4(1.0,31.0,12.0,0.001));
    float chs=0.0;
    for(int i=0;i<4;i++){
        chs+=0.5+0.5*sin(iChannelTime[i]*0.73+float(i)*1.17);
        chs+=0.0003*length(iChannelResolution[i].xy)+0.000001*iChannelResolution[i].z;
    }
    float aMix=clamp(amp*0.7+uamp*0.3,0.0,20.0);
    vec2 m=(iMouse.w>0.5?iMouse.xy/iResolution:vec2(0.5));
    vec2 click=mix(vec2(0.5),iMouseClicked,step(0.0,length(iMouseClicked)));
    float clickBurst=exp(-10.0*length(uv-click))*step(0.0,length(iMouseClicked));
    float seg=3.0+mod(float(iFrame),9.0)+smoothstep(0.0,1.0,fract(t*0.11+chs*0.07))*3.0;
    float swirl=0.6+0.4*sin(t*0.3+chs*0.5);
    vec2 warp0=vec2(fbm(uv*6.0+vec2(t*0.3,chs)),fbm(uv*6.0+vec2(-t*0.4,chs*0.7)))-(0.5);
    vec2 warp1=vec2(n2(uv*12.0+rot(vec2(t*0.7,chs),t*0.2)),n2(uv*12.0-rot(vec2(t*0.6,chs),t*0.25)))
    ↪ -(0.5);
    vec2 warp=(warp0*0.035+warp1*0.02)*(1.0+0.6*sin(t*0.8))*sr*(0.75+0.25*sin(fr*0.02));
    vec2 kUV=kale(uv,seg,m,aspect);
    vec2 p=(kUV-m)*vec2(aspect,1.0);
    p=rot(p,swirl*(0.5+0.5*sin(t*0.21+chs)));
    float r=length(p);
    float rip=sin(22.0*r - t*9.0 - chs*3.0 + aMix*0.15)*0.015/(1.0+20.0*r);
    vec2 dir=normalize(p+1e-5);
    vec2 uA=uv+warp+dir*rip;
    vec2 uB=mix(uv,fract((p*0.18+warp*2.0)/vec2(aspect,1.0)+m),0.85);
    vec2 uC=uv+rot(warp,0.5+t*0.1);
    float zoom=0.9+0.25*sin(pingPong(t*0.6+chs,2.0)*PI+clickBurst*2.0);
    uA=(uA-m)*zoom+m;
    uB=(uB-m)*(1.0/zoom)+m;
    uC=(uC-m)*(1.0+0.15*sin(t*0.9+fr*0.03))+m;
    vec4 t1=texture(samp,fract(uA));
    vec4 t2=texture(samp,fract(uB));
    vec4 t3=texture(samp,fract(uC));
    vec3 blur=tentBlur3(samp,fract(uv+warp*2.0),iResolution);
    float hueBase=fract(t*0.07+dateSeed*0.13+chs*0.11+float(iFrame)*0.0007);
    float sat=0.65+0.35*sin(chs+t*0.6+sr);
    float val=0.55+0.45*sin(t*0.7+dot(uv,vec2(3.1,2.7)));
    vec3 tint=HSV2rgb(vec3(hueBase+length(p)*0.3+0.07*sin(iTimeDelta*120.0+fr*0.02),sat,val));
    vec4 mixA=mix(t1,t2,0.5+0.5*sin(t+chs+aMix*0.05));

```



```

    vec4 mixB=mix(t2,t3,0.5+0.5*cos(t*1.3+sr));
    vec4 mixC=mix(mixA,mixB,0.5+0.5*sin(t*0.7+dateSeed));
    float ring=smoothstep(0.0,0.8,sin(log(r+1e-3)*9.5+t*1.2+fr*0.01));
    float pulse=0.5+0.5*sin(t*2.0+r*28.0+chs*2.0+aMix*0.1)+clickBurst;
    float vign=1.0-smoothstep(0.75,1.15,length((uv-m)*vec2(aspect,1.0)));
    vec3 col=mixC.rgb;
    col=col*(0.75+0.25*ring)*(0.85+0.15*pulse)*vign;
    col=mix(col,blur,0.12*(0.5+0.5*sin(t+chs)));
    col*=tint*(0.9+0.1*sin(float(iFrame)*0.02+chs));
    vec3 bloom=col*col*0.18+pow(max(col-0.6,0.0),vec3(2.0))*0.12;
    col+=bloom;
    vec3 base=texture(samp,tc).rgb;
    float mixTex=pingPong(pulse*PI,5.0)*0.22+0.1*abs(sin(aMix*0.05));
    col=mix(col,base,mixTex);
    col=clamp(col,vec3(0.02),vec3(0.98));
    color=vec4(col,1.0);
}

```

Shader #68 - blue 2

File: blue_2.glsl

Uses distance fields with sinusoidal modulation to generate animated visual patterns. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Linear mix/lerp blending - Distance field computation - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float sat(float x){return clamp(x,0.0,1.0);}
void main(void){
    vec2 uv = tc;
    vec2 m = (iMouse.z>0.0||iMouse.w>0.0)? iMouse.xy/iResolution : vec2(0.5);
    float v = 0.4 + amp*0.2;
    float s = 0.22;
    float width = 0.10;
    float base = mod(time_f*v, s);
    vec2 dir = normalize(uv - m + 1e-6);
    float r = distance(uv, m);
    float waveSum = 0.0;
    for(int k=0;k<4;k++){
        float rc = base + float(k)*s;
        float a = (r - rc)/width;
        waveSum += exp(-a*a);
    }
    vec2 duv = dir * sin(pingPong(waveSum * PI, 5.0)) * (0.015 + 0.015*uamp);
    vec4 tex = texture(samp, uv + duv);
}

```

```

float blend = sat(waveSum * (0.45 + 0.55*uamp));
vec3 blue = mix(tex.rgb, vec3(0.06, 0.28, 1.0), blend);
color = vec4(blue, 1.0);
}

```

Shader #69 - blue 3

File: blue_3.glsl

Uses distance fields with sinusoidal modulation to generate animated visual patterns. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Linear mix/lerp blending - Distance field computation - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float sat(float x){return clamp(x,0.0,1.0);}
void main(void){
    vec2 uv = tc;
    vec2 m = (iMouse.z>0.0||iMouse.w>0.0)? iMouse.xy/iResolution : vec2(0.5);
    float v = 0.4 + amp*0.2;
    float s = 0.22;
    float width = 0.10;
    float base = pingPong(sin(mod(time_f*v, s) * PI), 5.0);
    vec2 dir = normalize(uv - m + 1e-6);
    float r = distance(uv, m);
    float waveSum = 0.0;
    for(int k=0;k<4;k++){
        float rc = base + float(k)*s;
        float a = (r - rc)/width;
        waveSum += exp(-a*a);
    }
    vec2 duv = dir * sin(pingPong(waveSum * PI, 5.0)) * (0.015 + 0.015*uamp);
    vec4 tex = texture(samp, uv + duv);
    float blend = sat(waveSum * (0.45 + 0.55*uamp));
    vec3 blue = mix(tex.rgb, vec3(0.06, 0.28, 1.0), blend);
    color = vec4(blue, 1.0);
}

```

Shader #70 - blue 4

File: blue_4.glsl

Uses distance fields with sinusoidal modulation to generate animated visual patterns. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Linear mix/lerp blending - Distance field computation - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
float sat(float x){return clamp(x,0.0,1.0);}
void main(void){
    vec2 uv = tc;
    vec2 m = (iMouse.z>0.0||iMouse.w>0.0)? iMouse.xy/iResolution : vec2(0.5);
    float v = 0.4 + amp*0.2;
    float s = 0.22;
    float width = 0.10;
    float base = mod(time_f*v, s);
    vec2 dir = normalize(uv - m + 1e-6);
    float r = distance(uv, m);
    float waveSum = 0.0;
    for(int k=0;k<4;k++){
        float rc = base + float(k)*s;
        float a = (r - rc)/width;
        waveSum += exp(-a*a);
    }
    vec2 duv = dir * sin(pingPong(waveSum * PI, 5.0)) * (0.015 + 0.015*uamp);
    vec4 tex = texture(samp, uv + duv);
    float blend = pingPong(sat(waveSum * (0.45 + 0.55*uamp)) * (time_f * PI), 5.0);
    vec3 blue = mix(tex.rgb, vec3(0.06, 0.28, 1.0), blend);
    color = vec4(blue, 1.0);
}
```

Shader #169 - crystalprism

File: crystalprism.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - UV mirroring/folding for symmetry - Linear mix/lerp blending - Driven by: time, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp;
uniform float uamp;
```

```

uniform float blendAmt;
float PI=3.141592653589793;
float mirror1(float x,float m){
    float p=mod(x,m*2.0);
    return p>m?(2.0*m-p):p;
}
vec2 kaleido(vec2 p,float seg){
    vec2 c=vec2(0.5);
    vec2 d=p-c;
    float a=atan(d.y,d.x);
    float r=length(d);
    float m=PI/max(2.0,seg);
    a=mirror1(a,m);
    return c+vec2(cos(a),sin(a))*max(r,1e-6);
}
vec3 softLight(vec3 base,vec3 blend){
    vec3 d=step(vec3(0.5),blend);
    vec3 a=2.0*blend*base+base*base*(1.0-2.0*blend);
    vec3 b=2.0*base*(1.0-blend)+sqrt(base)*(2.0*blend-1.0);
    return mix(a,b,d);
}
void main(){
    vec2 uv=tc;
    vec4 otex=texture(samp,uv);
    vec3 orig=otex.rgb;
    float luma=dot(orig,vec3(0.2126,0.7152,0.0722));
    float drive=clamp(amp+uamp,0.0,2.0);
    float seg=floor(mix(6.0,12.0,clamp(drive*0.5,0.0,1.0)));
    vec2 k=kaleido(uv,seg);
    float r=length(k-0.5);
    vec2 dir=normalize(k-0.5+vec2(1e-6,0.0));
    float texWarp=mix(0.4,1.0,luma);
    float bend=(0.04+0.12*uamp)*texWarp/(r+0.05);
    vec2 wob=0.010*(0.3+amp)*texWarp*vec2(sin(30.0*r-1.6*time_f),cos(26.0*r-1.2*time_f));
    vec2 base=k+dir*bend+wob;
    float angle=atan(dir.y,dir.x);
    float disp=(0.002+0.010*amp)*texWarp;
    vec2 oR=base+disp*vec2(cos(angle+0.03),sin(angle+0.03));
    vec2 oG=base;
    vec2 oB=base+disp*vec2(cos(angle-0.03),sin(angle-0.03));
    vec3 warped;
    warped.r=texture(samp,oR).r;
    warped.g=texture(samp,oG).g;
    warped.b=texture(samp,oB).b;
    float wedge=PI/max(2.0,seg);
    float edge=abs(mod(angle+wedge,2.0*wedge)-wedge)/wedge;
    float edgeGlow=pow(1.0-edge,6.0)*mix(0.08,0.30,amp);
    warped+=edgeGlow;
    float t=clamp(mix(0.35,0.85,blendAmt)*(0.6+0.4*clamp(drive*0.5,0.0,1.0)),0.0,1.0);
    vec3 fused=mix(orig,softLight(orig,warped),t);
    color=vec4(fused,otex.a);
}

```

Shader #195 - datam

File: datam.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Using time_f as requested Displacement distance Glitch probability/frequency

Technique: Color quantization via floor rounding - Per-channel UV offset for chromatic aberration
- Driven by: time, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp;
uniform float uamp;
float hash(vec2 p) {
    p = fract(p * vec2(123.34, 456.21));
    p += dot(p, p + 45.32);
    return fract(p.x * p.y);
}
vec3 getDatamosh(vec2 uv) {
    float blockSize = 16.0;
    vec2 blockUV = floor(uv * iResolution / blockSize) * blockSize / iResolution;
    vec3 blockColor = texture(samp, blockUV).rgb;
    float luma = dot(blockColor, vec3(0.299, 0.587, 0.114));
    vec2 motion;
    motion.x = sin(luma * 20.0 + time_f) * 0.05;
    motion.y = cos(luma * 15.0 + time_f) * 0.05;
    float moshTrigger = hash(blockUV + floor(time_f * 8.0));
    vec2 finalUV = uv;
    if (moshTrigger < uamp * 0.5) {
        finalUV -= motion * amp;
    }
    vec3 col;
    col.r = texture(samp, finalUV).r;
    col.g = texture(samp, finalUV + (motion * amp * 0.2)).g;
    col.b = texture(samp, finalUV + (motion * amp * 0.4)).b;
    return col;
}
void main(void) {
    vec2 uv = tc;
    vec3 finalColor = getDatamosh(uv);
    finalColor = floor(finalColor * 16.0) / 16.0;
    color = vec4(finalColor, texture(samp, uv).a);
}
```

Shader #196 - datam2

File: datam2.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Assuming uniform sampler2D for texture sampling Time in seconds, smoothly varying over frames Amplitude of the glitch effect

Technique: Multi-tap convolution blur (5 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Driven by: time, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
```

```

uniform float amp;
uniform float uamp;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(12.9898, 78.233))) * 43758.5453);
}
vec3 getDatamosh(vec2 uv) {
    float blockSize = 16.0;
    vec2 blockUV = floor(uv * iResolution / blockSize) * (blockSize / iResolution);
    vec3 blockColor = texture(samp, blockUV).rgb;
    float luma = dot(blockColor, vec3(0.299, 0.587, 0.114));
    vec2 motion = vec2(sin(luma * 20.0 + time_f) * amp, cos(luma * 20.0 + time_f) * amp);
    float moshTrigger = hash(blockUV + floor(time_f * 8.0));
    vec2 finalUV = uv;
    if (moshTrigger < uamp) {
        finalUV += motion * sin(time_f);
    }
    vec3 col;
    col.r = texture(samp, finalUV).r;
    col.g = texture(samp, finalUV + (motion * amp * 0.2)).g;
    col.b = texture(samp, finalUV + (motion * amp * 0.4)).b;
    return col;
}
void main(void) {
    vec2 uv = tc;
    vec3 finalColor = getDatamosh(uv);
    finalColor = floor(finalColor * 16.0) / 16.0;
    color = vec4(finalColor, texture(samp, uv).a);
}

```

Shader #197 - datam3

File: datam3.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Assuming uniform sampler2D for texture sampling Time in seconds, smoothly varying over frames Amplitude of the glitch effect

Technique: Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Driven by: time, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp;
uniform float uamp;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(12.9898, 78.233))) * 43758.5453);
}
vec4 getDatamosh(vec2 uv) {
    float blockSize = 16.0;
    vec2 blockUV = floor(uv * iResolution / blockSize) * (blockSize / iResolution);
    vec4 blockColor = texture(samp, blockUV);
    float luma = dot(blockColor.rgb, vec3(0.299, 0.587, 0.114));
    vec2 motion = vec2(sin(luma * 20.0 + time_f) * amp, cos(luma * 20.0 + time_f) * amp);
    float moshTrigger = hash(blockUV + floor(time_f * 8.0));
    vec2 finalUV = uv;

```

```

    if (moshTrigger < uamp) {
        finalUV += motion * sin(time_f);
    }
    vec3 col;
    col.r = texture(samp, finalUV).r;
    col.g = texture(samp, finalUV + (motion * amp * 0.2)).g;
    col.b = texture(samp, finalUV + (motion * amp * 0.4)).b;
    return vec4(col, 1.0);
}

void main(void) {
    vec2 uv = tc;
    vec4 finalColor = getDatamosh(uv);
    finalColor.rgb = floor(finalColor.rgb * 16.0) / 16.0;
    color = finalColor;
}

```

Shader #198 - datam4

File: datam4.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Assuming uniform sampler2D for texture sampling Time in seconds, smoothly varying over frames Amplitude of the glitch effect

Technique: Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Driven by: time, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp;
uniform float uamp;
uniform float alpha_r;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(12.9898, 78.233))) * 43758.5453);
}

vec4 getDatamosh(vec2 uv) {
    float blockSize = (alpha_r * 2.0) * 16.0;
    vec2 blockUV = floor(uv * iResolution / blockSize) * (blockSize / iResolution);
    vec4 blockColor = texture(samp, blockUV);
    float luma = dot(blockColor.rgb, vec3(0.299, 0.587, 0.114));
    vec2 motion = vec2(sin(luma * 20.0 + time_f) * amp, cos(luma * 20.0 + time_f) * amp);
    float moshTrigger = hash(blockUV + floor(time_f * 8.0));
    vec2 finalUV = uv;
    if (moshTrigger < uamp) {
        finalUV += motion * sin(time_f);
    }
    vec3 col;
    col.r = texture(samp, finalUV).r;
    col.g = texture(samp, finalUV + (motion * amp * 0.2)).g;
    col.b = texture(samp, finalUV + (motion * amp * 0.4)).b;
    return vec4(col, 1.0);
}

void main(void) {
    vec2 uv = tc;
    vec4 finalColor = getDatamosh(uv);
}

```

```

    finalColor.rgb = floor(finalColor.rgb * 16.0) / 16.0;
    color = finalColor;
}

```

Shader #199 - datam5

File: datam5.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Assuming uniform sampler2D for texture sampling Time in seconds, smoothly varying over frames Amplitude of the glitch effect

Technique: Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Driven by: time, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp;
uniform float uamp;
uniform float alpha_r;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(12.9898, 78.233))) * 43758.5453);
}
vec4 getDatamosh(vec2 uv) {
    float blockSize = sin(alpha_r * time_f) * 16.0;
    vec2 blockUV = floor(uv * iResolution / blockSize) * (blockSize / iResolution);
    vec4 blockColor = texture(samp, blockUV);
    float luma = dot(blockColor.rgb, vec3(0.299, 0.587, 0.114));
    vec2 motion = vec2(sin(luma * 20.0 + time_f) * amp, cos(luma * 20.0 + time_f) * amp);
    float moshTrigger = hash(blockUV + floor(time_f * 8.0));
    vec2 finalUV = uv;
    if (moshTrigger < uamp) {
        finalUV += motion * sin(time_f);
    }
    vec3 col;
    col.r = texture(samp, finalUV).r;
    col.g = texture(samp, finalUV + (motion * amp * 0.2)).g;
    col.b = texture(samp, finalUV + (motion * amp * 0.4)).b;
    return vec4(col, 1.0);
}
void main(void) {
    vec2 uv = tc;
    vec4 finalColor = getDatamosh(uv);
    finalColor.rgb = floor(finalColor.rgb * 16.0) / 16.0;
    color = finalColor;
}

```

Shader #200 - datam a

File: datam_a.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds

to audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Assuming uniform sampler2D for texture sampling Time in seconds, smoothly varying over frames Amplitude of the glitch effect

Technique: Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Driven by: time, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp;
uniform float uamp;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(12.9898, 78.233))) * 43758.5453);
}
vec4 getDatamosh(vec2 uv) {
    float blockSize = sin(alpha_r * time_f) * 16.0;
    vec2 blockUV = floor(uv * iResolution / blockSize) * (blockSize / iResolution);
    vec4 blockColor = texture(samp, blockUV);
    float luma = dot(blockColor.rgb, vec3(0.299, 0.587, 0.114));
    vec2 motion = vec2(sin(luma * 20.0 + time_f) * alpha_r, cos(luma * 20.0 + time_f) * alpha_g);
    float moshTrigger = hash(blockUV + floor(time_f * 8.0));
    vec2 finalUV = uv;
    if (moshTrigger < alpha_b * 0.5) {
        finalUV += motion * sin(time_f);
    }
    vec3 col;
    col.r = texture(samp, finalUV).r;
    col.g = texture(samp, finalUV + (motion * amp * 0.2)).g;
    col.b = texture(samp, finalUV + (motion * amp * 0.4)).b;
    return vec4(col, 1.0);
}
void main(void) {
    vec2 uv = tc;
    vec4 finalColor = getDatamosh(uv);
    finalColor.rgb = floor(finalColor.rgb * 16.0) / 16.0;
    color = finalColor;
}
```

Shader #323 - empty

File: empty.glsl

Applies a time-animated visual transformation to the input texture. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
```

```

uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
void main(void) {
    color = texture(samp, tc);
}

```

Shader #358 - frac shade02 dmdXi3 rot

File: frac_shade02_dmdXi3_rot.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

```

```

}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime * (PI / 2.0);
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float osc = 0.5 + 0.5 * sin(t * 0.35 + 0.2 * sr);
    float seg = 4.0 + 2.0 * sin(t * 0.25);
    float zoom = mix(1.1, 2.1, osc);
    float spinAngle = osc * 2.0 * PI;
    vec2 kUV = reflectUV(tc, seg, m, ar.x);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    kUV = rotateUV(kUV, spinAngle, m, ar.x);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    ripple = sin(ripple * pingPong(time_f * PI, 3.0));
    vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
    vec2 uB = mix(tc, kUV, 0.88);
    vec2 uC = mix(tc, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
    vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
    vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
    vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
    float hueBase = fract(t * 0.06);
    float sat = 0.7 + 0.25 * sin(t * 0.4 + sr);
    float val = 0.6 + 0.35 * sin(t * 0.5 + dot(tc, vec2(2.3, 1.9)));
    vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
}

```

```

float slowBeat = 0.5 + 0.5 * sin(t * 0.8 + aMix * 0.05);
float mix1 = 0.5 + 0.5 * sin(t * 0.6);
float mix2 = 0.5 + 0.5 * cos(t * 0.45);
vec3 warpCol = mix(t1.rgb, t2.rgb, mix1);
warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
warpCol *= tint * (0.92 + 0.08 * slowBeat);
vec3 base = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
vec3 combined = mix(base, warpCol * 3.0, 0.6);
vec3 bloom = combined * combined * 0.18 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.10;
combined += bloom;
combined = clamp(combined, vec3(0.0), vec3(1.0));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
combined *= 1.0 + _ab * 0.6;
combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(combined, 1.0);
}

```

Shader #365 - frac shader02 dmd2 amp

File: frac_shader02_dmd2_amp.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;

```

```

    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv, float t) {
    float aspect = iResolution.x / iResolution.y;
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 gdir = normalize(vec2(cos(t * 0.27), sin(t * 0.31)));
    float s = dot((uv - 0.5) * vec2(aspect, 1.0), gdir);

```

```

float w = max(fwidth(s) * 4.0, 0.002);
float band = smoothstep(-0.5 - w, -0.5 + w, sin(s * 2.2 + t * 0.9));
vec3 neon = neonPalette(t);
vec3 grad = mix(tex, mix(tex, neon, 0.6), 0.35 + 0.25 * band);
grad = mix(grad, tex, 0.10);
grad = softTone(grad);
return grad;
}
float diamondRadius(vec2 p) {
    p = abs(p);
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
vec3 limitHighlights(vec3 c) {
    float m = max(c.r, max(c.g, c.b));
    if (m > 0.95) c *= 0.95 / m;
    return c;
}
void main(void) {
    float aAcc = clamp(amp, 0.0, 20.0);
    float aInst = clamp(uamp, 0.0, 20.0);
    float aMix = clamp(aAcc * 0.5 + aInst * 1.0, 0.0, 20.0);
    float a01 = clamp(aMix / 6.0, 0.0, 1.0);
    float aspect = iResolution.x / iResolution.y;
    vec2 center = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5, 0.5);
    float tSpin = time_f * mix(0.3, 1.8, a01);
    float tSlow = time_f * mix(0.2, 0.9, a01);
    float tFast = time_f * mix(0.8, 3.2, a01);
    vec2 offset = tc - center;
    float maxRadius = length(vec2(0.5, 0.5));
    float radius = length(offset);
    float normalizedRadius = radius / maxRadius;
    float angle = atan(offset.y, offset.x);
    float distortion = 0.25 + 0.75 * a01;
    float distortedRadius = normalizedRadius + distortion * normalizedRadius * normalizedRadius;
    distortedRadius = clamp(distortedRadius, 0.0, 1.0);
    distortedRadius *= maxRadius;
    vec2 distortedCoords = center + distortedRadius * vec2(cos(angle), sin(angle));
    float modulatedTime = pingPong(tSpin, 5.0);
    angle += modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle) * (distortedCoords.x - center.x)
        - sin(angle) * (distortedCoords.y - center.y) + center.x;
    rotatedTC.y = sin(angle) * (distortedCoords.x - center.x)
        + cos(angle) * (distortedCoords.y - center.y) + center.y;
    float warpSpeed = 0.05 + 0.25 * a01 + 0.2 * (aInst / (aInst + 1.0));
    vec2 warpedCoords;
    warpedCoords.x = pingPong(rotatedTC.x + tSpin * warpSpeed, 1.0);
    warpedCoords.y = pingPong(rotatedTC.y + tSpin * warpSpeed, 1.0);
    vec2 uvGlow = warpedCoords * 2.0 - 1.0;
    uvGlow.x *= aspect;
    float rGlow = length(uvGlow);
    float radiusMax = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radiusMax, radiusMax - 0.25, rGlow);
    vec2 m = center;
    vec2 ar = vec2(aspect, 1.0);

```

```

vec3 baseColBlur = preBlendColor(warpedCoords, tSlow);
float seg = 4.0 + 2.0 * sin(tSlow * 0.33 + a01 * 2.0);
vec2 kUV = reflectUV(warpedCoords, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + 0.55 * sin(tSlow * 0.42 + a01 * 3.0);
kUV = fractalFold(kUV, foldZoom, tFast, m, aspect);
kUV = rotateUV(kUV, tSlow * 0.23 + a01 * 1.1, m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * sin(tSlow * 0.2);
float period = log(base);
float tz = tSlow * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + tFast * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * (0.0015 + 0.001 * sin(tFast * 1.3)) * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((warpedCoords - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off, tFast);
vec3 gC = preBlendColor(u1, tFast);
vec3 bC = preBlendColor(u2 - off, tFast);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + tFast * 1.2));
float pulse = 0.5 + 0.5 * sin(tFast * 2.0 + rD * 28.0 + k * 12.0);
pulse *= mix(0.4, 1.4, a01);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.10
    + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.06;
outCol += bloom;
outCol = mix(outCol, baseColBlur, 0.18 + 0.3 * a01);
outCol = limitHighlights(outCol);
outCol = clamp(outCol, 0.0, 1.0);
vec4 baseTex = texture(samp, warpedCoords);
float mixAmt = pingPong(glow * PI, 5.0) * mix(0.4, 0.9, a01);
vec3 finalRGB = mix(baseTex.rgb, outCol, mixAmt);
finalRGB = limitHighlights(finalRGB);
finalRGB = clamp(finalRGB, 0.0, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #371 - frac shader02 dmd6i air

File: frac_shader02_dmd6i_air.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control.

Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
- Indexed color palette look-up table - 2D rotation matrix - UV mirroring/folding for symmetry
- Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
```



```

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv, float tSlow, float tFast, float aspect) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 gdir = normalize(vec2(cos(tSlow * 0.27), sin(tSlow * 0.31)));
    float s = dot((uv - 0.5) * vec2(aspect, 1.0), gdir);
    float w = max(fwidth(s) * 4.0, 0.002);
    float band = smoothstep(-0.5 - w, -0.5 + w, sin(s * 2.2 + tFast * 0.9));
    vec3 neon = neonPalette(tFast);
    vec3 grad = mix(tex, mix(tex, neon, 0.6), 0.35 + 0.25 * band);
    grad = mix(grad, tex, 0.10);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 rot2(vec2 v, float a){
    float c = cos(a), s = sin(a);
    return vec2(c*v.x - s*v.y, s*v.x + c*v.y);
}

vec2 h2(vec2 p){
    return fract(sin(vec2(dot(p,vec2(127.1,311.7)),
                        dot(p,vec2(269.5,183.3))))) * 43758.5453);
}

```

```

}
vec3 limitHighlights(vec3 c){
    float m = max(c.r, max(c.g, c.b));
    if(m > 0.9) c *= 0.9 / m;
    return c;
}
void main(void) {
    float a = clamp(amp, 0.0, 1.0);
    float ua = clamp(uamp, 0.0, 1.0);
    float aMix = clamp(amp * 0.7 + uamp * 1.3, 0.0, 4.0);
    float a01 = clamp(aMix / 2.5, 0.0, 1.0);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    float tBase = time_f;
    float tSlow = tBase * mix(0.2, 0.9, a01);
    float tFast = tBase * mix(0.7, 3.5, a01);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution)
        : fract(vec2(0.37 + 0.11 * sin(tSlow*0.63 + seed),
            0.42 + 0.13 * cos(tSlow*0.57 + seed*2.0)));

    vec4 baseTex = texture(samp, tc);
    vec2 uvForGlow = tc * 2.0 - 1.0;
    uvForGlow.x *= aspect;
    float rGlow = pingPong(sin(length(uvForGlow) * tSlow), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - 0.25, rGlow);
    vec2 uv = tc;
    float speedScale = 1.0 + 2.0 * a01 + 3.0 * ua;
    float speedR = 5.0 * speedScale;
    float speedG = 6.5 * speedScale;
    float speedB = 4.0 * speedScale;
    float ampR = 0.03 * (0.7 + 0.6 * a01);
    float ampG = 0.025 * (0.7 + 0.6 * a01);
    float ampB = 0.035 * (0.7 + 0.6 * a01);
    float waveR = 10.0;
    float waveG = 12.0;
    float waveB = 8.0;
    float rR = sin(uv.x*waveR + tFast*speedR)*ampR
        + sin(uv.y*waveR*0.8 + tFast*speedR*1.2)*ampR;
    float rG = sin(uv.x*waveG*1.5 + tFast*speedG)*ampG
        + sin(uv.y*waveG*0.3 + tFast*speedG*0.7)*ampG;
    float rB = sin(uv.x*waveB*0.5 + tFast*speedB)*ampB
        + sin(uv.y*waveB*1.7 + tFast*speedB*1.3)*ampB;
    vec2 tcR = uv + vec2(rR, rR);
    vec2 tcG = uv + vec2(rG, -0.5*rG);
    vec2 tcB = uv + vec2(0.3*rB, rB);
    vec3 pats[4] = vec3[](vec3(1,0,1), vec3(0,1,0), vec3(1,0,0), vec3(0,0,1));
    float pspd = 4.0;
    int pidx = int(mod(floor(tBase*pspd + seed*4.0), 4.0));
    vec3 mir = pats[pidx];
    vec2 dR = tcR - m;
    vec2 dG = tcG - m;
    vec2 dB = tcB - m;
    float fallR = smoothstep(0.55, 0.0, length(dR));
    float fallG = smoothstep(0.55, 0.0, length(dG));
    float fallB = smoothstep(0.55, 0.0, length(dB));
    float sw = (0.12 + 0.38*ua + 0.25*a);
    vec2 tangR = rot2(normalize(dR + 1e-4), 1.5707963);
    vec2 tangG = rot2(normalize(dG + 1e-4), 1.5707963);
    vec2 tangB = rot2(normalize(dB + 1e-4), 1.5707963);
    vec2 airR = tangR * sw * fallR * (0.06+0.22*a) * (0.6+0.4*cos(uv.y*40.0 + tFast*3.0 + seed));
    vec2 airG = tangG * sw * fallG * (0.06+0.22*a) * (0.6+0.4*cos(uv.y*38.0 + tFast*3.3 + seed*1.7));
    vec2 airB = tangB * sw * fallB * (0.06+0.22*a) * (0.6+0.4*cos(uv.y*42.0 + tFast*2.9 + seed*0.9));
}

```

```

vec2 jit = (h2(uv*vec2(233.3,341.9) + tFast + seed) - 0.5)
           * (0.0006 + 0.004*ua);
tcR += airR + jit;
tcG += airG + jit;
tcB += airB + jit;
vec2 fR = vec2(mir.r>0.5 ? 1.0 - tcR.x : tcR.x, tcR.y);
vec2 fG = vec2(mir.g>0.5 ? 1.0 - tcG.x : tcG.x, tcG.y);
vec2 fB = vec2(mir.b>0.5 ? 1.0 - tcB.x : tcB.x, tcB.y);
float seg = 4.0 + 2.0 * sin(tSlow * 0.33 + a01 * 2.0);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + 0.55 * sin(tSlow * 0.42 + a01 * 3.0);
kUV = fractalFold(kUV, foldZoom, tFast, m, aspect);
kUV = rotateUV(kUV, tSlow * 0.23 + a01 * 1.1, m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(tSlow * 0.2) * (PI * tSlow), 5.0);
float period = log(base) * pingPong(tSlow * PI, 5.0);
float tz = tSlow * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35
           + 0.35 * sin(rD * 18.0 + tFast * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dirK = normalize(pwrap + 1e-6);
vec2 off = dirK * (0.0015 + 0.001 * sin(tFast * 1.3)) * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec2 sR = mix(u0, fR, 0.6);
vec2 sG = mix(u1, fG, 0.6);
vec2 sB = mix(u2, fB, 0.6);
vec3 rC = preBlendColor(sR + off, tSlow, tFast, aspect);
vec3 gC = preBlendColor(sG, tSlow, tFast, aspect);
vec3 bC = preBlendColor(sB - off, tSlow, tFast, aspect);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + tFast * 1.2));
ring = ring * pingPong(tSlow * PI, 5.0);
float pulse = 0.5 + 0.5 * sin(tFast * 2.0 + rD * 28.0 + k * 12.0);
pulse *= mix(0.4, 1.4, a01);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.10
           + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.06;
float pulseAdd = 0.004 * (0.5 + 0.5 * sin(tFast * 3.7 + seed));
outCol += bloom;
outCol += pulseAdd * ua;
outCol = clamp(outCol, vec3(0.0), vec3(1.0));
outCol = limitHighlights(outCol);
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * (0.6 + 0.3 * a01));
finalRGB = limitHighlights(finalRGB);
finalRGB = clamp(finalRGB, 0.0, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);

```

```
    color = vec4(finalRGB, baseTex.a);  
}
```

Shader #373 - frac shader02 dmd6i amp

File: frac_shader02_dmd6i_amp.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core  
out vec4 color;  
in vec2 tc;  
uniform sampler2D samp;  
uniform vec2 iResolution;  
uniform float time_f;  
uniform vec4 iMouse;  
uniform float amp;  
uniform float uamp;  
uniform float amp_peak;  
uniform float amp_rms;  
uniform float amp_smooth;  
uniform float amp_low;  
uniform float amp_mid;  
uniform float amp_high;  
uniform float iamp;  
const float PI = 3.1415926535897932384626433832795;  
float gAmp01;  
float gSlow;  
float gFast;  
float gDetail;  
float pingPong(float x, float length) {  
    float m = mod(x, length * 2.0);  
    return m <= length ? m : length * 2.0 - m;  
}  
vec3 hsv2rgb(vec3 c) {  
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);  
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);  
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);  
}  
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {  
    float s = sin(angle), cc = cos(angle);  
    vec2 p = uv - c;  
    p.x *= aspect;  
    p = mat2(cc, -s, s, cc) * p;  
    p.x /= aspect;  
    return p + c;  
}  
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {  
    vec2 p = uv - c;  
    p.x *= aspect;  
    float ang = atan(p.y, p.x);  
    float rad = length(p);  
    float stepA = 6.28318530718 / segments;  
    ang = mod(ang, stepA);
```

```

    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    float f = mix(0.4, 2.0, gAmp01);
    p = sin(abs(p * f));
}

```

```

    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    float aAcc = clamp(amp, 0.0, 4.0);
    float aInst = clamp(uamp, 0.0, 4.0);
    float ampMix = clamp(aAcc * 0.6 + aInst * 1.4, 0.0, 4.0);
    gAmp01 = clamp(ampMix / 2.5, 0.0, 1.0);
    gSlow = time_f * mix(0.15, 0.7, gAmp01);
    gFast = time_f * mix(0.6, 3.5, gAmp01);
    gDetail = time_f * mix(0.3, 2.0, gAmp01);
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * gSlow), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + 2.0 * sin(gSlow * 0.33 + gAmp01 * 2.0);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(pingPong(gSlow * PI, 25.0) * 0.42 + gAmp01 * 3.0);
    kUV = fractalFold(kUV, foldZoom, gDetail, m, aspect);
    kUV = rotateUV(kUV, gSlow * 0.23 + gAmp01 * 1.2, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(gSlow * 0.2) * (PI * gSlow), 5.0);
    float period = log(base) * pingPong(gSlow * PI, 5.0);
    float tz = gSlow * 0.65;
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + gFast * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * (0.0015 + 0.001 * sin(gFast * 1.3)) * vec2(1.0, 1.0 / aspect);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + gFast * 1.2));
    ring = ring * pingPong(gSlow * PI, 5.0);
    float pulse = 0.5 + 0.5 * sin(gFast * 2.0 + rD * 28.0 + k * 12.0);
    pulse *= mix(0.4, 1.4, gAmp01);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
}

```

```

    vec3 bloom = outCol * outCol * 0.10 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.08;
    outCol += bloom;
    outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    outCol *= mix(0.7, 1.15, gAmp01);
    vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
    finalRGB = clamp(finalRGB, 0.0, 1.0);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #374 - frac shader02 dmd6i audio

File: frac_shader02_dmd6i_audio.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. AUDIO: Smooth energy accelerates the neon color cycling AUDIO: Bass increases the number of kaleido reflection segments AUDIO: Mids push the fractal folding zoom

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

```

```

}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * 0.35 + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3 + (amp_smooth * 4.0));
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = 0.3 + 0.4 * (1.0 - neonAmt);
}

```



```

    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - 0.25, r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + 2.0 * sin(time_f * 0.33) + (amp_low * 6.0);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(time_f * 0.42) + (amp_mid * 0.6);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * 0.23, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * 0.65 + (amp_smooth * 2.5);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * (0.0015 + 0.001 * sin(time_f * 1.3) + (amp_high * 0.02)) * vec2(1.0, 1.0 / aspect);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, 1.15, vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * PI), 5.0);
    float pulse = 0.5 + 0.5 * sin(time_f * 2.0 + rD * 28.0 + k * 12.0) + (amp_peak * 0.8);
    pulse *= iamp;
    vec3 outCol = kaleidoRGB;
    outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
}

```

```

    vec3 bloom = outCol * outCol * (0.18 + amp_rms * 0.6) + pow(max(outCol - 0.6, 0.0), vec3(2.0)) *
    ↪ (0.12 + amp_peak * 0.4);
    outCol += bloom;
    outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #375 - frac shader02 dmd6i bowl

File: frac_shader02_dmd6i_bowl.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control. Computes luminance for brightness-based effects.

Technique: Multi-tap convolution blur (5 texture samples) - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float h1(float n){return fract(sin(n*91.345+37.12)*43758.5453123);}
vec2 h2(vec2 p){return fract(sin(vec2(dot(p,vec2(127.1,311.7)),dot(p,vec2(269.5,183.3))))*43758.5453);}
vec2 rot(vec2 v,float a){float c=cos(a),s=sin(a);return vec2(c*v.x-s*v.y,s*v.x+c*v.y);}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 cent, float aspect) {
    float s = sin(angle), cc = cos(angle);

```

```

    vec2 p = uv - cent;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + cent;
}

vec2 reflectUV(vec2 uv, float segments, vec2 cent, float aspect) {
    vec2 p = uv - cent;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + cent;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 cent, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - cent) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + cent;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, cent, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 col) {
    col = pow(max(col, 0.0), vec3(0.95));
    float l = dot(col, vec3(0.299, 0.587, 0.114));
    col = mix(vec3(l), col, 0.9);
    return clamp(col, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);

```

```

float r = length(p);
float t = time_f;
vec3 neon = neonPalette(t + r * 1.3);
float neonAmt = smoothstep(0.1, 0.8, r);
neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
vec3 grad = mix(tex, neon, neonAmt);
grad = mix(grad, tex, 0.2);
grad = softTone(grad);
return grad;
}
float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 cent, float aspect) {
    vec2 p = (uv - cent) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + cent;
}
void main(void){
    float a = clamp(amp,0.0,1.0);
    float ua = clamp(uamp,0.0,1.0);
    float t = time_f;
    vec2 center = vec2(0.5, 0.5);
    vec2 baseUV = tc;
    vec2 offset = baseUV - center;
    float maxRadius = length(vec2(0.5, 0.5));
    float radius = length(offset);
    float normalizedRadius = radius / maxRadius;
    float distortion = 0.25 + 0.45*ua + 0.3*a;
    float distortedRadius = normalizedRadius + distortion * normalizedRadius * normalizedRadius;
    distortedRadius = clamp(distortedRadius, 0.0, 1.0);
    distortedRadius *= maxRadius;
    vec2 normDir = radius > 0.0 ? offset / radius : vec2(0.0);
    vec2 distortedCoords = center + distortedRadius * normDir;
    float spinSpeed = 0.6 + 1.8*(0.3 + 0.7*a);
    float modulatedTime = pingPong(t * spinSpeed, 5.0);
    float angSpin = atan(distortedCoords.y - center.y, distortedCoords.x - center.x) + modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angSpin) * (distortedCoords.x - center.x) - sin(angSpin) * (distortedCoords.y -
    ↪ center.y) + center.x;
    rotatedTC.y = sin(angSpin) * (distortedCoords.x - center.x) + cos(angSpin) * (distortedCoords.y -
    ↪ center.y) + center.y;
    float warpAmp = 0.02 + 0.06*ua + 0.04*a;
    vec2 uvWarp;
    uvWarp.x = pingPong(rotatedTC.x + t * 0.12 * (1.0 + warpAmp*5.0), 1.0);
    uvWarp.y = pingPong(rotatedTC.y + t * 0.12 * (1.0 + warpAmp*5.0), 1.0);
    vec2 uv = uvWarp;
    float speedR=5.0, ampR=0.03, waveR=10.0;
    float speedG=6.5, ampG=0.025, waveG=12.0;
    float speedB=4.0, ampB=0.035, waveB=8.0;
    float rR=sin(uv.x*waveR+t*speedR)*ampR + sin(uv.y*waveR*0.8+t*speedR*1.2)*ampR;
    float rG=sin(uv.x*waveG*1.5+t*speedG)*ampG + sin(uv.y*waveG*0.3+t*speedG*0.7)*ampG;
    float rB=sin(uv.x*waveB*0.5+t*speedB)*ampB + sin(uv.y*waveB*1.7+t*speedB*1.3)*ampB;
    vec2 tcR=uv+vec2(rR,rR);
    vec2 tcG=uv+vec2(rG,-0.5*rG);
    vec2 tcB=uv+vec2(0.3*rB,rB);
    vec3 pats[4]=vec3[] (vec3(1,0,1),vec3(0,1,0),vec3(1,0,0),vec3(0,0,1));
    float pspd=4.0;
    int pidx=int(mod(floor(t*pspd+seed*4.0),4.0));

```

```

vec3 mir=pats[pidx];
vec2 m = iMouse.z>0.5 ? (iMouse.xy/iResolution) : fract(vec2(0.37+0.11*sin(t*0.63+seed),0.42+0.13*
↳ cos(t*0.57+seed*2.0)));
vec2 dR=tcR-m, dG=tcG-m, dB=tcB-m;
float fallR=smoothstep(0.55,0.0,length(dR));
float fallG=smoothstep(0.55,0.0,length(dG));
float fallB=smoothstep(0.55,0.0,length(dB));
float sw=(0.12+0.38*ua+0.25*a);
vec2 tangR=rot(normalize(dR+1e-4),1.5707963);
vec2 tangG=rot(normalize(dG+1e-4),1.5707963);
vec2 tangB=rot(normalize(dB+1e-4),1.5707963);
vec2 airR=tangR*sw*fallR*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*40.0+t*3.0+seed));
vec2 airG=tangG*sw*fallG*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*38.0+t*3.3+seed*1.7));
vec2 airB=tangB*sw*fallB*(0.06+0.22*a)*(0.6+0.4*cos(uv.y*42.0+t*2.9+seed*0.9));
vec2 jit = (h2(uv*vec2(233.3,341.9)+t+seed)-0.5)*(0.0006+0.004*ua);
tcR += airR + jit;
tcG += airG + jit;
tcB += airB + jit;
vec2 fR=vec2(mir.r>0.5?1.0-tcR.x:tcR.x, tcR.y);
vec2 fG=vec2(mir.g>0.5?1.0-tcG.x:tcG.x, tcG.y);
vec2 fB=vec2(mir.b>0.5?1.0-tcB.x:tcB.x, tcB.y);
float ca=0.0015+0.004*a;
vec4 C=texture(samp,uv);
C.r=texture(samp,fR+vec2(ca,0)).r;
C.g=texture(samp,fG).g;
C.b=texture(samp,fB+vec2(-ca,0)).b;
float pulseChrom=0.004*(0.5+0.5*sin(t*3.7+seed));
C.rgb+=pulseChrom*ua;
vec3 chromRGB = C.rgb;
vec4 baseTex = texture(samp, uv);
vec2 uvN = tc * 2.0 - 1.0;
float aspect = iResolution.x / iResolution.y;
uvN.x *= aspect;
float r = pingPong(sin(length(uvN) * time_f), 5.0);
float radiusN = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radiusN, radiusN - 0.25, r);
vec2 ar = vec2(aspect, 1.0);
vec3 baseCol = preBlendColor(uv);
float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
vec2 kUV = reflectUV(uv, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float baseZ = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(baseZ) * pingPong(time_f * PI, 5.0);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;

```

```

float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
float mixNeonBase = pingPong(pulse * PI, 5.0) * 0.18;
vec3 neonBaseMix = mix(baseCol, outCol, 0.65 + 0.35*ua);
vec3 neonOut = mix(baseTex.rgb, neonBaseMix, mixNeonBase + 0.45);
neonOut = clamp(neonOut, vec3(0.05), vec3(0.97));
float glowMix = pingPong(glow * PI, 5.0) * 0.8;
vec3 neonFinal = mix(baseTex.rgb, neonOut, glowMix);
float layerMix = clamp(0.35 + 0.4*a + 0.25*ua, 0.0, 1.0);
vec3 finalRGB = mix(neonFinal, chromRGB, layerMix);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, 1.0);
}

```

Shader #387 - frac shader02 dmdi radial

File: frac_shader02_dmdi_radial.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);

```

```

    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;

```

```

    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 frag = gl_FragCoord.xy / iResolution;
    vec2 normCoord = (frag * 2.0 - 1.0) * vec2(aspect, 1.0);
    float distanceFromCenter = length(normCoord);
    float wave = sin(distanceFromCenter * 12.0 - time_f * 4.0);
    float aNorm = clamp(amp, 0.0, 1.5);
    float uNorm = clamp(uamp, 0.0, 1.5);
    float waveStrength = mix(0.06, 0.30, clamp(uNorm * 0.7, 0.0, 1.0));
    waveStrength *= 0.8 + 0.4 * aNorm;
    vec2 waveOffset = normCoord * waveStrength * wave;
    vec2 tcWave = tc + waveOffset;
    vec4 baseTex = texture(samp, tcWave);
    vec2 uv = tcWave * 2.0 - 1.0;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tcWave);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33 + uNorm * 0.5);
    vec2 kUV = reflectUV(tcWave, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.35 + (0.65 + amp_low * 0.4) * sin(time_f * 0.42 + uNorm * 0.8);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.18 + 0.10 * aNorm), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.75 + 0.25 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);

```



```

float period = log(base) * pingPong(time_f * PI, 5.0);
float tz = time_f * (0.55 + 0.25 * uNorm);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * (1.03 + 0.02 * uNorm)) / ar + m);
vec2 u2 = fract((pwrap * (0.97 - 0.02 * uNorm)) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * (0.0010 + 0.0015 * uNorm) * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tcWave - m) * ar));
vign = mix(0.9, 1.18, vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring *= pingPong(time_f * PI + uNorm * 2.5, 5.0);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0 + aNorm * 1.5);
vec3 outCol = kaleidoRGB;
float gain = 0.7 + 0.3 * uNorm;
outCol *= (0.75 + 0.25 * ring * gain) * (0.85 + 0.15 * pulse * gain) * vign;
vec3 bloom = outCol * outCol * (0.04 + 0.12 * uNorm) +
    pow(max(outCol - 0.6, 0.0), vec3(2.0)) * (0.04 + 0.08 * uNorm);
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * (0.15 + 0.15 * (1.0 - uNorm * 0.5)));
outCol = clamp(outCol, vec3(0.04), vec3(0.96));
float mixK = pingPong(glow * PI, 5.0) * (0.45 + 0.35 * uNorm);
vec3 finalRGB = mix(baseTex.rgb, outCol, mixK);
finalRGB = clamp(finalRGB, vec3(0.03), vec3(0.97));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #388 - frac shader02 dmdXi

File: frac_shader02_dmdXi.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;

```

```

uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
}

```

```

float a = step(ph, 0.33);
float b = step(0.33, ph) * step(ph, 0.66);
float c = step(0.66, ph);
return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime;
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float seg = 4.0 + 2.0 * sin(t * 0.25);
    float zoom = 1.45 + 0.45 * sin(t * 0.35 + 0.2 * sr);
    vec2 kUV = reflectUV(tc, seg, m, ar.x);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    kUV = rotateUV(kUV, t * 0.22, m, ar.x);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
    vec2 uB = mix(tc, kUV, 0.88);
    vec2 uC = mix(tc, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
    vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
    vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
    vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
    float hueBase = fract(t * 0.06);
    float sat = 0.7 + 0.25 * sin(t * 0.4 + sr);
    float val = 0.6 + 0.35 * sin(t * 0.5 + dot(tc, vec2(2.3, 1.9)));
    vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
    float slowBeat = 0.5 + 0.5 * sin(t * 0.8 + aMix * 0.05);
    float mix1 = 0.5 + 0.5 * sin(t * 0.6);
    float mix2 = 0.5 + 0.5 * cos(t * 0.45);
    vec3 warpCol = mix(t1.rgb, t2.rgb, mix1);
    warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
    warpCol *= tint * (0.92 + 0.08 * slowBeat);
    vec3 base = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
    vec3 combined = mix(base, warpCol * 3.0, 0.6);
    vec3 bloom = combined * combined * 0.18 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.10;
    combined += bloom;
    combined = clamp(combined, vec3(0.0), vec3(1.0));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    combined *= 1.0 + _ab * 0.6;
    combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(combined, 1.0);
}

```

Shader #389 - frac shader02 dmdXi2

File: frac_shader02_dmdXi2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

```

```

}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime;
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float seg = 4.0 + 2.0 * sin(t * 0.25);
    float zoom = 1.45 + 0.45 * sin(t * 0.35 + 0.2 * sr);
    vec2 kUV = reflectUV(tc, seg, m, ar.x);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    kUV = rotateUV(kUV, t * 0.22, m, ar.x);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
    vec2 uB = mix(tc, kUV, 0.88);
    vec2 uC = mix(tc, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
    vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
    vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
    vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
    float hueBase = fract(t * 0.06);
    float sat = 0.7 + 0.25 * sin(t * 0.4 + sr);
    float val = 0.6 + 0.35 * sin(t * 0.5 + dot(tc, vec2(2.3, 1.9)));
    vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
    float slowBeat = 0.5 + 0.5 * sin(t * 0.8 + aMix * 0.05);
    float mix1 = 0.5 + 0.5 * sin(t * 0.6);
    float mix2 = 0.5 + 0.5 * cos(t * 0.45);
    vec3 warpCol = mix(sin(t1.rgb * pingPong(time_f * PI, 10.0)), sin(t2.rgb * pingPong(time_f * PI
    ↪ /2,8.0)), mix1);
    warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
    warpCol *= tint * (0.92 + 0.08 * slowBeat);
    vec3 base = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
    vec3 combined = mix(base, warpCol * 3.0, 0.6);
    vec3 bloom = combined * combined * 0.18 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.10;
    combined += bloom;
    combined = clamp(combined, vec3(0.0), vec3(1.0));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    combined *= 1.0 + _ab * 0.6;
    combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(combined, 1.0);
}

```

Shader #390 - frac shader02 dmdXi3

File: frac_shader02_dmdXi3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
```

```

    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime * (PI/2);
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float seg = 4.0 + 2.0 * sin(t * 0.25);
    float zoom = 1.45 + 0.45 * sin(t * 0.35 + 0.2 * sr);
    vec2 kUV = reflectUV(tc, seg, m, ar.x);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    kUV = rotateUV(kUV, t * 0.22, m, ar.x);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    ripple = sin(ripple * pingPong(time_f * PI, 3.0));
    vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
    vec2 uB = mix(tc, kUV, 0.88);
    vec2 uC = mix(tc, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
    vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
    vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
    vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
    float hueBase = fract(t * 0.06);
    float sat = 0.7 + 0.25 * sin(t * 0.4 + sr);
    float val = 0.6 + 0.35 * sin(t * 0.5 + dot(tc, vec2(2.3, 1.9)));
    vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
    float slowBeat = 0.5 + 0.5 * sin(t * 0.8 + aMix * 0.05);
    float mix1 = 0.5 + 0.5 * sin(t * 0.6);
    float mix2 = 0.5 + 0.5 * cos(t * 0.45);
    vec3 warpCol = mix(t1.rgb, t2.rgb, mix1);
    warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
    warpCol *= tint * (0.92 + 0.08 * slowBeat);
    vec3 base = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
    vec3 combined = mix(base, warpCol * 3.0, 0.6);
    vec3 bloom = combined * combined * 0.18 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.10;
    combined += bloom;
    combined = clamp(combined, vec3(0.0), vec3(1.0));
    float _ab = clamp(amp_peak, 0.0, 1.0);
}

```

```

float _abass = clamp(amp_low, 0.0, 1.0);
combined *= 1.0 + _ab * 0.6;
combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(combined, 1.0);
}

```

Shader #391 - frac shader02 dmdXi3 drain

File: frac_shader02_dmdXi3_drain.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

```



```

}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        float z = zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i));
        p = abs((p - c) * z) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    m += (iMouseClicked / iResolution) * 0.1;
    float t = time_f + iTime * (PI / 2.0);
    float rate = max(iFrameRate, 1.0);
    float sr = clamp((iSampleRate / 48000.0) * (rate / 60.0), 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float chanAspect = iChannelResolution[0].y > 0.0
        ? iChannelResolution[0].x / iChannelResolution[0].y
        : 1.0;
    float chanBeat = 0.5 + 0.5 * sin(iChannelTime[0] * 0.5 + iDate.x * 0.1);
    float loopDuration = 25.0;
    float currentTime = mod(time_f + float(iFrame) * 0.01, loopDuration);
    float zoomPing = pingPong((time_f + iTimeDelta * 60.0) * 0.25 + chanBeat, 1.0);
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    normCoord.x = abs(normCoord.x);
    float dist = length(normCoord);
    dist = clamp(dist, 0.0, 1.4);
    float angle = atan(normCoord.y, normCoord.x);
    float spiralSpeed = 5.0;
    float inwardBase = currentTime / loopDuration;
    float inwardSpeed = mix(inwardBase, zoomPing, 0.5);
    float edgeFade = 1.0 - smoothstep(0.0, 8.0, dist);
    angle += edgeFade * currentTime * spiralSpeed * (0.6 + 0.4 * chanBeat);
    dist *= 1.0 - inwardSpeed;
    float distWarp = tan(dist * (0.4 + 0.3 * zoomPing));
    vec2 spiralCoord = vec2(cos(angle), sin(angle)) * distWarp;
    spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) * 0.5;
    vec2 baseUV = mix(tc, spiralCoord, 0.75);
    baseUV = fract(baseUV);
    float osc = 0.5 + 0.5 * sin(t * 0.35 + 0.2 * sr + iDate.y * 0.01);
    float seg = 4.0 + 2.0 * sin(t * 0.25 + chanAspect * 0.2);
    float zoomBase = mix(1.1, 2.1, osc);
    float zoom = zoomBase * (0.7 + 1.8 * zoomPing);
    float spinAngle = osc * 2.0 * PI * (0.8 + 0.4 * chanBeat);
    vec2 kUV = reflectUV(baseUV, seg, m, ar.x);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    kUV = rotateUV(kUV, spinAngle, m, ar.x);
    vec2 dir = (kUV - m) * ar;
}

```

```

float r = length(dir);
float ripple = sin(20.0 * r - t * 9.0 - iDate.z * 0.02) * 0.012 / (1.0 + 18.0 * r);
ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
ripple = sin(ripple * pingPong(time_f * PI + chanBeat, 3.0 + 1.5 * zoomPing));
ripple *= 1.0 + 0.6 * sin(aMix * 0.1 + iTimeDelta * 10.0);
vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
vec2 uB = mix(baseUV, kUV, 0.88);
vec2 uC = mix(baseUV, kUV, 0.94)
          + vec2(0.002 * sin(t + iDate.w * 0.05), 0.002 * cos(t + iTimeDelta * 5.0));
vec2 fA = fract(uA);
vec2 fB = fract(uB);
vec2 fC = fract(uC);
vec4 t1 = textureGrad(samp, fA, dFdx(fA), dFdy(fA));
vec4 t2 = textureGrad(samp, fB, dFdx(fB), dFdy(fB));
vec4 t3 = textureGrad(samp, fC, dFdx(fC), dFdy(fC));
float hueBase = fract(t * 0.06 + iDate.x * 0.01);
float sat = 0.7 + 0.25 * sin(t * 0.4 + sr + chanBeat);
float val = 0.6 + 0.35 * sin(t * 0.5 + dot(baseUV, vec2(2.3, 1.9)) + iFrame * 0.002);
vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
float slowBeat = 0.5 + 0.5 * sin(t * 0.8 + aMix * 0.05 + chanBeat);
float mix1 = 0.5 + 0.5 * sin(t * 0.6 + zoomPing * PI);
float mix2 = 0.5 + 0.5 * cos(t * 0.45 + inwardBase * PI);
vec3 warpCol = mix(t1.rgb, t2.rgb, mix1);
warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
warpCol *= tint * (0.92 + 0.08 * slowBeat);
vec3 baseTex = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
vec3 spiralTex = textureGrad(samp, spiralCoord, dFdx(spiralCoord), dFdy(spiralCoord)).rgb;
vec3 spiralMix = mix(baseTex, spiralTex, 0.7 + 0.3 * zoomPing);
vec3 combined = mix(spiralMix, warpCol * 3.0, 0.6);
vec3 bloom = combined * combined * 0.18
              + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.10;
combined += bloom;
combined = clamp(combined, vec3(0.0), vec3(1.0));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
combined *= 1.0 + _ab * 0.6;
combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(combined, 1.0);
}

```

Shader #393 - frac shader02 dmdXi4

File: frac_shader02_dmdXi4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;

```

```

uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec2 inwardSpiral(vec2 uv, vec2 c, float aspect, float t){
    vec2 p = uv - c;
    p.x *= aspect;
    float r = length(p) + 1e-6;
    float ang = atan(p.y, p.x);
    float spin = -0.65 * t;
    float swirl = -1.35 / (1.0 + 10.0 * r);

```

```

    ang += spin + swirl;
    float pull = 0.18 / (1.0 + 8.0 * r);
    r = max(r - pull, 0.0);
    vec2 q = vec2(cos(ang), sin(ang)) * r;
    q.x /= aspect;
    return q + c;
}
vec3 satBoost(vec3 c, float s){
    float l = dot(c, vec3(0.2126,0.7152,0.0722));
    return mix(vec3(l), c, 1.0 + s);
}
vec3 vibrance(vec3 c, float v){
    float mx = max(c.r, max(c.g, c.b));
    float mn = min(c.r, min(c.g, c.b));
    float sat = mx - mn;
    float amt = 1.0 + v * (1.0 - sat);
    float l = dot(c, vec3(0.2126,0.7152,0.0722));
    return mix(vec3(l), c, amt);
}
vec3 screen(vec3 a, vec3 b){
    return 1.0 - (1.0 - a)*(1.0 - b);
}
vec3 aces(vec3 x){
    const float a=2.51;
    const float b=0.03;
    const float c=2.43;
    const float d=0.59;
    const float e=0.14;
    x = (x*(a*x+b))/(x*(c*x+d)+e);
    return clamp(x,0.0,1.0);
}
void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = (time_f + iTime);
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float seg = 6.0 + 4.0 * sin(t * 0.25);
    float zoom = 1.65 + 0.55 * sin(t * 0.35 + 0.2 * sr);
    vec2 kUV = reflectUV(tc, seg, m, ar.x);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    kUV = rotateUV(kUV, t * 0.28, m, ar.x);
    kUV = inwardSpiral(kUV, m, ar.x, t);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(22.0 * r - t * 9.5) * 0.013 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
    vec2 uB = mix(tc, kUV, 0.94);
    vec2 uC = mix(tc, kUV, 0.985) + vec2(0.0018 * sin(t), 0.0018 * cos(t));
    vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
    vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
    vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
    float hueSweep = fract(t*0.07 + tc.x*0.35 + tc.y*0.18 + r*0.25);
    float hueSweep2 = fract(t*0.05 - tc.y*0.28 + tc.x*0.12 - r*0.18);
    float sat = 0.85 + 0.12*sin(t*0.6 + sr);
    vec3 gradA = hsv2rgb(vec3(hueSweep, sat, 1.0));
    vec3 gradB = hsv2rgb(vec3(hueSweep2, 0.95, 1.0));
    vec3 gradient = mix(gradA, gradB, 0.5 + 0.5*sin(t*0.4 + dot(tc, vec2(1.7,1.3))));
    float slowBeat = 0.5 + 0.5 * sin(t * 0.8 + aMix * 0.05);
    float mix1 = 0.55 + 0.45 * sin(t * 0.6);
    float mix2 = 0.55 + 0.45 * cos(t * 0.45);

```

```

    vec3 warpCol = mix(t1.rgb, t2.rgb, mix1);
    warpCol = mix(warpCol, t3.rgb, mix2 * 0.65);
    vec3 neon = screen(warpCol, gradient);
    neon = satBoost(neon, 0.65);
    neon = vibrance(neon, 0.9);
    neon = pow(max(neon, 0.0), vec3(0.9))*(1.15 + 0.1*slowBeat);
    vec3 base = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
    float patMix = clamp(0.82 + 0.16*slowBeat + 0.12*clamp(aMix*0.08,0.0,1.0), 0.0, 0.98);
    vec3 combined = mix(base, neon, patMix);
    vec3 bloom = combined*combined*0.16 + pow(max(combined-0.7,0.0), vec3(2.0))*0.10;
    combined += bloom;
    combined = aces(combined);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    combined *= 1.0 + _ab * 0.6;
    combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(combined, 1.0);
    color = mix(color, texture(samp, tc), 0.8);
}

```

Shader #394 - frac shader02 dmdXi5

File: frac_shader02_dmdXi5.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Multi-tap convolution blur (9 texture samples) - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;

```

```

float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}

vec2 mirror(vec2 uv){
    vec2 w = abs(fract(uv*0.5+0.5)*2.0-1.0);
    return clamp(w, 0.0, 1.0);
}

vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        float z = zoom + 0.10 * sin(t * 0.20 + float(i)*0.7);
        p = abs((p - c) * z) - 0.5 + c;
        p = rotateUV(p, t * 0.10 + float(i) * 0.06, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.04);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.02;
}

vec3 softTone(vec3 c){
    c = pow(max(c, 0.0), vec3(0.96));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res){

```

```

    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv, float t, float sr){
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 gdir = normalize(vec2(cos(t * 0.18 + 0.3*sr), sin(t * 0.20 - 0.3*sr)));
    float s = dot((uv - 0.5) * vec2(iResolution.x / iResolution.y, 1.0), gdir);
    float w = max(fwidth(s) * 6.0, 0.003);
    float band = smoothstep(-0.5 - w, -0.5 + w, sin(s * 1.6 + t * 0.6));
    vec3 neon = neonPalette(t);
    vec3 grad = mix(tex, mix(tex, neon, 0.55), 0.30 + 0.20 * band);
    grad = mix(grad, tex, 0.12);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p){
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 wormholeUV(vec2 uv, vec2 c, float aspect, float t){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float swirl = 0.18 / r;
    a += t * 0.25 + swirl;
    float z = 0.55 + 0.30 * sin(t * 0.45) + 0.10 * sin(r * 9.0 - t * 2.0);
    float rr = 1.0 / (r * 3.2 + 0.08) + 0.06 * z;
    vec2 q = vec2(cos(a), sin(a)) * rr;
    q /= ar;
    return mirror(q + c);
}

float sceneLuma(){
    vec3 s0 = texture(samp, vec2(0.25,0.25)).rgb;
    vec3 s1 = texture(samp, vec2(0.50,0.25)).rgb;
    vec3 s2 = texture(samp, vec2(0.75,0.25)).rgb;
    vec3 s3 = texture(samp, vec2(0.25,0.50)).rgb;
    vec3 s4 = texture(samp, vec2(0.50,0.50)).rgb;
    vec3 s5 = texture(samp, vec2(0.75,0.50)).rgb;
    vec3 s6 = texture(samp, vec2(0.25,0.75)).rgb;
    vec3 s7 = texture(samp, vec2(0.50,0.75)).rgb;
    vec3 s8 = texture(samp, vec2(0.75,0.75)).rgb;
    float L = 0.0;
    L += dot(s0, vec3(0.299,0.587,0.114));
    L += dot(s1, vec3(0.299,0.587,0.114));
    L += dot(s2, vec3(0.299,0.587,0.114));

```

```

    L += dot(s3, vec3(0.299,0.587,0.114));
    L += dot(s4, vec3(0.299,0.587,0.114));
    L += dot(s5, vec3(0.299,0.587,0.114));
    L += dot(s6, vec3(0.299,0.587,0.114));
    L += dot(s7, vec3(0.299,0.587,0.114));
    L += dot(s8, vec3(0.299,0.587,0.114));
    return L / 9.0;
}

vec3 toneDownWhite(vec3 c){
    float lum = dot(c, vec3(0.299, 0.587, 0.114));
    float factor = smoothstep(0.6, 1.0, lum);
    c = mix(c, c * 0.7, factor);
    c = pow(c, vec3(0.95 + 0.05 * (1.0 - factor)));
    return clamp(c, 0.0, 1.0);
}

void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime;
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float seg = 4.0 + 2.0 * sin(t * 0.20 + 0.25 * aMix);
    float zoom = 1.45 + 0.40 * sin(t * 0.28 + 0.15 * sr);
    vec2 kUV = reflectUV(tc, seg, m, ar.x);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    kUV = rotateUV(kUV, t * 0.18, m, ar.x);
    kUV = diamondFold(kUV, m, ar.x);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(14.0 * r - t * 5.0) * 0.008 / (1.0 + 12.0 * r);
    ripple *= 1.0 / (1.0 + 10.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
    vec2 uB = mix(tc, kUV, 0.82);
    vec2 uC = mix(tc, kUV, 0.90) + vec2(0.0015 * sin(t*0.8), 0.0015 * cos(t*0.8));
    vec2 sA = mirror(uA);
    vec2 sB = mirror(uB);
    vec2 sC = mirror(uC);
    vec4 t1 = textureGrad(samp, sA, dFdx(sA), dFdy(sA));
    vec4 t2 = textureGrad(samp, sB, dFdx(sB), dFdy(sB));
    vec4 t3 = textureGrad(samp, sC, dFdx(sC), dFdy(sC));
    vec3 warpCol = mix(t1.rgb, t2.rgb, 0.5 + 0.5 * sin(t * 0.45));
    warpCol = mix(warpCol, t3.rgb, 0.25 + 0.25 * cos(t * 0.33));
    float sat = 0.72 + 0.22 * sin(t * 0.32 + sr);
    float val = 0.62 + 0.30 * sin(t * 0.36 + dot(tc, vec2(2.1, 1.7)));
    vec3 tint = hsv2rgb(vec3(fract(t * 0.045) + r * 0.20, sat, val));
    warpCol *= tint * (0.94 + 0.06 * (0.5 + 0.5 * sin(t * 0.6 + aMix * 0.04)));
    vec3 baseTex = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
    vec3 preK = preBlendColor(mirror(kUV), t, sr);
    vec2 wh0 = wormholeUV(tc, m, ar.x, t);
    vec2 wh1 = wormholeUV(tc + vec2(0.0009, 0.0), m, ar.x, t + 0.02);
    vec2 wh2 = wormholeUV(tc - vec2(0.0009, 0.0), m, ar.x, t - 0.02);
    vec2 off = normalize(dir + 1e-5) * (0.0012 + 0.0008 * sin(t * 0.9)) * vec2(1.0, 1.0 / ar.x);
    vec3 whR = preBlendColor(mirror(wh0 + off), t, sr);
    vec3 whG = preBlendColor(mirror(wh1), t, sr);
    vec3 whB = preBlendColor(mirror(wh2 - off), t, sr);
    vec3 wormRGB = vec3(sin(whR.r * pingPong(t * PI, 1.1)),
                       sin(whG.g * pingPong(t * PI, 1.2)),
                       sin(whB.b * pingPong(t * PI, 1.3)));
    float rCenter = length((tc - m) * ar);
    float throatBase = smoothstep(0.45, 0.10, rCenter);
    float throat = mix(throatBase, throatBase*throatBase, 0.5) * (0.55 + 0.45 * sin(t * 0.35));
    float swirlGate = smoothstep(0.8, 1.6, t * 0.18 + 0.25 * sin(t * 0.5));

```



```

float gateRaw = throat * (0.55 + 0.35 * pingPong(t * PI, 3.0)) + swirlGate * 0.12;
float wGate = max(fwidth(gateRaw)*2.0, 0.002);
float gate = smoothstep(0.0, 1.0, clamp(gateRaw, 0.0, 1.0));
gate = smoothstep(0.0, 1.0, mix(gate, gate, 1.0 - wGate));
vec3 mixA = mix(preK, warpCol, 0.52);
vec3 mixB = mix(mixA, wormRGB, gate * 0.72);
vec3 bloom = mixB * mixB * 0.14 + pow(max(mixB - 0.65, 0.0), vec3(2.0)) * 0.08;
vec3 combined = mixB + bloom;
float vign = 1.0 - smoothstep(0.78, 1.10, length((tc - m) * ar));
combined *= mix(0.96, 1.06, vign);
float Lscene = sceneLuma();
float baseFactor = mix(1.0, 0.75, smoothstep(0.55, 0.9, Lscene));
float finalFactor = mix(1.0, 0.72, smoothstep(0.55, 0.9, Lscene));
baseTex = toneDownWhite(baseTex) * baseFactor;
vec3 master = mix(baseTex, combined * 2.6, 0.58) * finalFactor;
master = clamp(master, vec3(0.03), vec3(0.97));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
master *= 1.0 + _ab * 0.6;
master = mix(master, master * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
↳ 0.0, 1.0) * 0.25), _ab);
color = vec4(master, 1.0);
}

```

Shader #395 - frac shader02 dmdXi amp

File: frac_shader02_dmdXi_amp.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;

```

```

const float PI = 3.1415926535897932384626433832795;
float gAmp01;
float gInst01;
float gSlow;
float gFast;
float gDetail;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 limitHighlights(vec3 c){
    float m = max(c.r, max(c.g, c.b));
    if(m > 0.9) c *= 0.9 / m;
    return c;
}
void main(void){

```

```

float aAcc = clamp(amp, 0.0, 20.0);
float aInst = clamp(uamp, 0.0, 20.0);
float aMix = clamp(aAcc * 0.7 + aInst * 0.3, 0.0, 20.0);
gAmp01 = clamp(aMix / 8.0, 0.0, 1.0);
gInst01 = clamp(aInst / 8.0, 0.0, 1.0);
float tGlobal = time_f + iTime;
float fpsNorm = clamp(iFrameRate / 60.0, 0.4, 2.5);
float t = tGlobal * fpsNorm;
float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
float chT0 = iChannelTime[0];
vec2 chRes0 = iChannelResolution[0].xy;
float chAspect = (chRes0.y > 0.0) ? chRes0.x / chRes0.y : (iResolution.x / iResolution.y);
float framePhase = float(iFrame) * 0.009 + iTimeDelta * 3.0;
float frameJitter = sin(framePhase) * 0.004;
float datePhase = iDate.y * 0.13 + iDate.z * 0.03 + iDate.w * 0.001;
gSlow = t * mix(0.18, 0.7, gAmp01);
gFast = t * mix(0.7, 4.0, gAmp01);
gDetail = t * mix(0.35, 2.3, gAmp01);
vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
vec2 baseCenter = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
vec2 clickCenter = (iMouseClicked.x > 0.0 || iMouseClicked.y > 0.0)
    ? (iMouseClicked / iResolution)
    : baseCenter;
float clickMix = 0.25 + 0.35 * smoothstep(0.2, 1.0, gAmp01);
vec2 m = mix(baseCenter, clickCenter, clickMix);
float aspect = ar.x;
vec2 uv0 = tc * 2.0 - 1.0;
uv0.x *= aspect;
float baseR = length(uv0);
float segBase = 4.0 + 2.0 * sin(gSlow * 0.25 + datePhase);
float segAmp = 3.0 * gAmp01 + 4.0 * gInst01;
float seg = segBase + segAmp * sin(gFast * 0.21 + chT0 * 0.37);
float zoomBase = 1.35 + 0.35 * sin(gSlow * 0.35 + 0.2 * sr);
float zoomAmp = 0.5 + 0.9 * gAmp01;
float zoom = zoomBase + zoomAmp * sin(gFast * 0.27 + baseR * 2.0 + framePhase);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = fractalFold(kUV, zoom, gDetail, m, aspect);
kUV = rotateUV(kUV, gSlow * 0.22 + gInst01 * 1.3, m, aspect + 0.2 * (chAspect - aspect));
vec2 dir = (kUV - m) * ar;
float r = length(dir);
float rippleAmp = mix(0.006, 0.02, gAmp01) * (0.7 + 0.6 * gInst01);
float rippleFreq = 18.0 + 6.0 * gAmp01 + 8.0 * gInst01;
float rippleTime = gFast * (0.8 + 0.4 * gInst01 * sr);
float ripple = sin(rippleFreq * r - rippleTime) * rippleAmp / (1.0 + 18.0 * r);
ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
ripple += frameJitter * 0.4;
vec2 dirN = normalize(dir + 1e-5);
vec2 uA = kUV + dirN * ripple;
vec2 uB = mix(tc, kUV, 0.84 + 0.08 * gAmp01);
vec2 jitter = vec2(0.002 * sin(t + framePhase), 0.002 * cos(t * 0.91 + framePhase * 1.3));
jitter *= (0.6 + 0.8 * gInst01);
vec2 uC = mix(tc, kUV, 0.93) + jitter;
vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
float hueBase = fract(gSlow * 0.04 + chT0 * 0.05 + datePhase * 0.1);
float sat = 0.65 + 0.25 * sin(gSlow * 0.4 + sr + gAmp01 * 2.0);
float val = 0.6 + 0.3 * sin(gFast * 0.35 + dot(tc, vec2(2.3, 1.9)) + gInst01 * 3.0);
vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
float slowBeat = 0.5 + 0.5 * sin(gSlow * 0.8 + aMix * 0.05);
float mix1 = 0.5 + 0.5 * sin(gFast * 0.6 + chT0 * 0.3);
float mix2 = 0.5 + 0.5 * cos(gSlow * 0.45 + framePhase);

```

```

    vec3 warpCol = mix(t1.rgb, t2.rgb, mix1);
    warpCol = mix(warpCol, t3.rgb, mix2 * (0.4 + 0.4 * gAmp01));
    float gain = mix(0.9, 1.5, gAmp01) * (0.9 + 0.3 * slowBeat);
    warpCol *= tint * gain;
    vec3 base = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
    float warpMix = 0.45 + 0.35 * gAmp01;
    vec3 combined = mix(base, warpCol, warpMix);
    vec3 bloom = combined * combined * 0.08
        + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.05;
    combined += bloom;
    combined = limitHighlights(combined);
    combined = clamp(combined, vec3(0.0), vec3(1.0));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    combined *= 1.0 + _ab * 0.6;
    combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(combined, 1.0);
}

```

Shader #397 - frac shader02 octo amp

File: frac_shader02_octo_amp.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);

```

```

    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(1), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv, float audio) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);

```

```

float r = length(p);
float t = time_f;
vec3 neon = neonPalette(t + r * (1.3 + audio * 0.4));
float neonAmt = smoothstep(0.1, 0.8, r);
neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
neonAmt += audio * 0.25;
neonAmt = clamp(neonAmt, 0.0, 0.95);
vec3 grad = mix(tex, neon, neonAmt);
grad = mix(grad, tex, 0.2);
grad = softTone(grad);
return grad;
}

float octaRadius(vec2 p) {
    p = abs(p);
    return p.x + p.y;
}

vec2 octaFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    float audio = clamp(amp * 1.2 + uamp * 0.8, 0.0, 3.0);
    float audioNorm = clamp(audio * 0.35, 0.0, 1.0);
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc, audioNorm);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33 + audioNorm * 0.8);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = octaFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42 + audioNorm * 0.6);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + audioNorm * 0.12), m, aspect);
    kUV = octaFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + audioNorm * 0.25);
    float rD = octaRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6 + audioNorm * 2.0);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * (1.045 + audioNorm * 0.05)) / ar + m);
    vec2 u2 = fract((pwrap * (0.955 - audioNorm * 0.05)) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    float offScale = 0.0015 + 0.001 * sin(time_f * 1.3 + audioNorm * 2.5);
    offScale *= 1.0 + audioNorm * 0.8;
    vec2 off = dir * offScale * vec2(1.0, 1.0 / aspect);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));

```

```

    vign = mix(0.9, 1.15 + audioNorm * 0.1, vign);
    vec3 rC = preBlendColor(u0 + off, audioNorm);
    vec3 gC = preBlendColor(u1, audioNorm);
    vec3 bC = preBlendColor(u2 - off, audioNorm);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2 + audioNorm * 3.0));
    ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
    float pulse = 0.5 + 0.4 * sin(time_f * 2.0 + rD * 28.0 + k * 12.0) + audioNorm * 0.5;
    pulse = clamp(pulse, 0.0, 1.4);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
    float level = fract((log(rD) - tz) / period);
    float distToEdge = min(level, 1.0 - level);
    float edgeWidth = mix(0.08, 0.03, audioNorm);
    float edge = 1.0 - smoothstep(0.0, edgeWidth, distToEdge);
    vec3 darkCol = outCol * (0.25 - 0.15 * audioNorm);
    outCol = mix(outCol, darkCol, edge);
    float bloomAmt = 0.18 + 0.10 * audioNorm;
    float hotAmt = 0.12 + 0.10 * audioNorm;
    vec3 bloom = outCol * outCol * bloomAmt + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * hotAmt;
    outCol += bloom;
    outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * (0.18 + 0.25 * audioNorm));
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    float mixAmt = pingPong(glow * PI, 5.0) * (0.5 + 0.3 * audioNorm);
    mixAmt = clamp(mixAmt, 0.0, 0.95);
    vec3 finalRGB = mix(baseTex.rgb, outCol, mixAmt);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #398 - frac shader02 octo strobe

File: frac_shader02_octo_strobe.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;

```

```

uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
}

```



```

    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv, float audio) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * (1.3 + audio * 0.4));
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    neonAmt += audio * 0.25;
    neonAmt = clamp(neonAmt, 0.0, 0.95);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float octaRadius(vec2 p) {
    p = abs(p);
    return p.x + p.y;
}

vec2 octaFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    float audio = clamp(amp * 1.2 + uamp * 0.8, 0.0, 3.0);
    float audioNorm = clamp(audio * 0.35, 0.0, 1.0);
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc, audioNorm);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33 + audioNorm * 0.8);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = octaFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42 + audioNorm * 0.6);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + audioNorm * 0.12), m, aspect);
    kUV = octaFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);

```

```

float tz = time_f * (0.65 + audioNorm * 0.25);
float rD = octaRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6 + audioNorm * 2.0);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * (1.045 + audioNorm * 0.05)) / ar + m);
vec2 u2 = fract((pwrap * (0.955 - audioNorm * 0.05)) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
float offScale = 0.0015 + 0.001 * sin(time_f * 1.3 + audioNorm * 2.5);
offScale *= 1.0 + audioNorm * 0.8;
vec2 off = dir * offScale * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, 1.15 + audioNorm * 0.1, vign);
vec3 rC = preBlendColor(u0 + off, audioNorm);
vec3 gC = preBlendColor(u1, audioNorm);
vec3 bC = preBlendColor(u2 - off, audioNorm);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2 + audioNorm * 3.0));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + 0.4 * sin(time_f * 2.0 + rD * 28.0 + k * 12.0) + audioNorm * 0.5;
pulse = clamp(pulse, 0.0, 1.4);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
float level = fract((log(rD) - tz) / period);
float distToEdge = min(level, 1.0 - level);
float edgeWidth = mix(0.08, 0.03, audioNorm);
float edge = 1.0 - smoothstep(0.0, edgeWidth, distToEdge);
float edgePhase = time_f * (8.0 + audioNorm * 14.0) + rD * 24.0;
float strobe = 0.5 + 0.5 * sin(edgePhase);
strobe = pow(strobe, 3.0);
vec3 edgeDark = outCol * (0.30 - 0.18 * audioNorm);
vec3 edgeBright = outCol * (1.15 + 0.45 * audioNorm);
vec3 edgeCol = mix(edgeDark, edgeBright, strobe);
outCol = mix(outCol, edgeCol, edge);
float bloomAmt = 0.18 + 0.10 * audioNorm;
float hotAmt = 0.12 + 0.10 * audioNorm;
vec3 bloom = outCol * outCol * bloomAmt + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * hotAmt;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * (0.18 + 0.25 * audioNorm));
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
float mixAmt = pingPong(glow * PI, 5.0) * (0.5 + 0.3 * audioNorm);
mixAmt = clamp(mixAmt, 0.0, 0.95);
vec3 finalRGB = mix(baseTex.rgb, outCol, mixAmt);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #399 - frac shader02 octo strobe2

File: frac_shader02_octo_strobe2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
 - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation -
 Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
```

```

    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv, float audio) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * (1.3 + audio * 0.4));
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    neonAmt += audio * 0.25;
    neonAmt = clamp(neonAmt, 0.0, 0.95);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float octaRadius(vec2 p) {
    p = abs(p);
    return p.x + p.y;
}

vec2 octaFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    float audio = clamp(amp * 1.2 + uamp * 0.8, 0.0, 3.0);
    float audioNorm = clamp(audio * 0.35, 0.0, 1.0);
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;

```

```

float r = pingPong(sin(length(uv) * time_f), 5.0);
float radius = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
vec2 ar = vec2(aspect, 1.0);
vec3 baseCol = preBlendColor(tc, audioNorm);
float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33 + audioNorm * 0.8);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = octaFold(kUV, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42 + audioNorm * 0.6);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + audioNorm * 0.12), m, aspect);
kUV = octaFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0);
float tz = time_f * (0.65 + audioNorm * 0.25);
float rD = octaRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6 + audioNorm * 2.0);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * (1.045 + audioNorm * 0.05)) / ar + m);
vec2 u2 = fract((pwrap * (0.955 - audioNorm * 0.05)) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
float offScale = 0.0015 + 0.001 * sin(pingPong(sin(time_f * PI), 5.0) * 1.3 + audioNorm * 2.5);
offScale *= 1.0 + audioNorm * 0.8;
vec2 off = dir * offScale * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, 1.15 + audioNorm * 0.1, vign);
vec3 rC = preBlendColor(u0 + off, audioNorm);
vec3 gC = preBlendColor(u1, audioNorm);
vec3 bC = preBlendColor(u2 - off, audioNorm);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2 + audioNorm * 3.0));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + 0.4 * sin(time_f * 2.0 + rD * 28.0 + k * 12.0) + audioNorm * 0.5;
pulse = clamp(pulse, 0.0, 1.4);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
float level = fract((log(rD) - tz) / period);
float distToEdge = min(level, 1.0 - level);
float edgeWidth = mix(0.08, 0.03, audioNorm);
float edge = 1.0 - smoothstep(0.0, edgeWidth, distToEdge);
float edgePhase = sin(time_f * PI) * (8.0 + audioNorm * 14.0) + rD * 24.0;
float strobe = 0.5 + 0.13 * sin(edgePhase * PI);
strobe = pow(strobe, 3.0);
strobe = atan(strobe * PI);
vec3 edgeDark = outCol * (0.30 - 0.18 * audioNorm);
vec3 edgeBright = outCol * (1.15 + 0.45 * audioNorm);
vec3 edgeCol = mix(edgeDark, edgeBright, strobe);
outCol = mix(outCol, edgeCol, edge);
float bloomAmt = 0.18 + 0.10 * audioNorm;
float hotAmt = 0.12 + 0.10 * audioNorm;
vec3 bloom = outCol * outCol * bloomAmt + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * hotAmt;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * (0.18 + 0.25 * audioNorm));
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
float mixAmt = pingPong(glow * PI, 5.0) * (0.5 + 0.3 * audioNorm);

```

```

    mixAmt = clamp(mixAmt, 0.0, 0.95);
    vec3 finalRGB = mix(baseTex.rgb, outCol, mixAmt);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #400 - frac shader02 prisim

File: frac_shader02_prisim.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;

```

```

    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 warpTexture(vec2 baseUV, float seg, float zoom, float t, vec2 m, float aspect, float aMix, float sr
    ↪ , float spinAngle){
    vec2 ar = vec2(aspect, 1.0);
    vec2 kUV = reflectUV(baseUV, seg, m, aspect);
    kUV = fractalFold(kUV, zoom, t, m, aspect);
    kUV = rotateUV(kUV, spinAngle, m, aspect);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    ripple = sin(ripple * pingPong(time_f * PI, 3.0));
    vec2 nDir = normalize(dir + 1e-5);
    vec2 uA = kUV + nDir * ripple;
    vec2 uB = mix(baseUV, kUV, 0.88);
    vec2 uC = mix(baseUV, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
    vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
    vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
    vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
    float ampNorm = clamp(aMix * 0.12, 0.0, 1.5);
    float hueBase = fract(t * 0.06 + r * 0.22 + ampNorm * 0.3);
    float sat = 0.75 + 0.20 * sin(t * 0.4 + sr * 0.7 + ampNorm * 0.4);
    float val = 0.70 + 0.35 * sin(t * 0.55 + r * 1.9 + ampNorm * 0.5);
    vec3 tint = hsv2rgb(vec3(hueBase, sat, val));
    vec3 neon = neonPalette(t + r * 2.0 + ampNorm);
    float mix1 = 0.5 + 0.5 * sin(t * 0.6 + ampNorm * 0.3);

```

```

float mix2 = 0.5 + 0.5 * cos(t * 0.45 + ampNorm * 0.5);
vec3 warpCol = mix(t1.rgb, t2.rgb, mix1);
warpCol = mix(warpCol, t3.rgb, mix2 * 0.7);
warpCol *= tint * neon;
warpCol *= 1.1 + 0.7 * clamp(aMix * 0.1, 0.0, 1.0);
vec3 baseTex = textureGrad(samp, baseUV, dFdx(baseUV), dFdy(baseUV)).rgb;
float warpMix = 0.55 + 0.35 * clamp(aMix * 0.08, 0.0, 1.0);
vec3 combined = mix(baseTex, warpCol * 1.9, warpMix);
vec3 bloom = combined * combined * 0.12 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.08;
combined += bloom;
return combined;
}

void main(void){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime * (PI / 2.0);
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float ampNorm = clamp(aMix * 0.12, 0.0, 1.5);
    float osc = 0.5 + 0.5 * sin(t * 0.35 + 0.2 * sr + ampNorm * 0.2);
    float seg = 4.0 + 2.0 * sin(t * 0.25) + 2.5 * clamp(ampNorm, 0.0, 2.0);
    float zoom = mix(1.1, 2.3 + ampNorm * 0.4, osc);
    float spinAngle = osc * 2.0 * PI * (0.6 + 0.4 * clamp(ampNorm, 0.0, 1.5));
    vec2 centerPx = iResolution * 0.5;
    vec2 texCoordPx = tc * iResolution;
    vec2 deltaPx = texCoordPx - centerPx;
    float dist = length(deltaPx);
    float radius = 0.45 + 0.3 * clamp(ampNorm * 0.7, 0.0, 1.0);
    float maxRadius = min(iResolution.x, iResolution.y) * radius;
    float scaleFactor = 1.0 - pow(clamp(dist / maxRadius, 0.0, 1.0), 2.0);
    scaleFactor *= 1.0 + 0.9 * clamp(ampNorm, 0.0, 1.5);
    scaleFactor = clamp(scaleFactor, 0.0, 1.9);
    vec2 dirN = dist > 0.0 ? deltaPx / dist : vec2(0.0);
    float offsetBase = mix(0.008, 0.03, clamp(ampNorm * 0.6, 0.0, 1.0));
    float offsetR = offsetBase;
    float offsetG = 0.0;
    float offsetB = -offsetBase;
    vec3 colBase = warpTexture(tc, seg, zoom, t, m, aspect, aMix, sr, spinAngle);
    vec3 colBubble = colBase;
    if(dist < maxRadius){
        vec2 texCoordR = centerPx + deltaPx * scaleFactor + dirN * offsetR * maxRadius;
        vec2 texCoordG = centerPx + deltaPx * scaleFactor + dirN * offsetG * maxRadius;
        vec2 texCoordB = centerPx + deltaPx * scaleFactor + dirN * offsetB * maxRadius;
        vec2 uvR = clamp(texCoordR / iResolution, 0.0, 1.0);
        vec2 uvG = clamp(texCoordG / iResolution, 0.0, 1.0);
        vec2 uvB = clamp(texCoordB / iResolution, 0.0, 1.0);
        vec3 colR = warpTexture(uvR, seg, zoom, t, m, aspect, aMix, sr, spinAngle);
        vec3 colG = warpTexture(uvG, seg, zoom, t, m, aspect, aMix, sr, spinAngle);
        vec3 colB = warpTexture(uvB, seg, zoom, t, m, aspect, aMix, sr, spinAngle);
        vec3 rgbSplit = vec3(colR.r, colG.g, colB.b);
        float mask = smoothstep(maxRadius, maxRadius * 0.65, dist);
        colBubble = mix(colBase, rgbSplit, mask);
    }
    vec3 baseVideo = textureGrad(samp, tc, dFdx(tc), dFdy(tc)).rgb;
    float globalMix = 0.50 + 0.35 * clamp(ampNorm * 0.5, 0.0, 1.0);
    vec3 combined = mix(baseVideo, colBubble, globalMix);
    combined = combined / (1.0 + combined);
    combined = clamp(combined, vec3(0.0), vec3(1.0));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    combined *= 1.0 + _ab * 0.6;
}

```



```

    combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(combined, 1.0);
}

```

Shader #402 - frac shader03 worm4 amp

File: frac_shader03_worm4_amp.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float gAmp01;
float gInstAmp;
float gSlow;
float gFast;
float gDetail;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);

```

```

    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

```

```

float diamondRadius(vec2 p) {
    float f = mix(0.6, 2.0, gAmp01);
    p = sin(abs(p * f));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 wormholeUV(vec2 uv, vec2 c, float aspect, float t) {
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float swirl = 0.22 / r;
    float spinBase = 0.15 + 0.35 * gAmp01;
    float spinHit = 0.2 + 2.0 * gInstAmp;
    a += t * (spinBase + spinHit) + swirl;
    float z = 0.65 + 0.35 * sin(t * 0.6) + 0.15 * sin(r * 12.0 - t * (2.5 + 1.5 * gAmp01 + 1.5 *
        ↪ gInstAmp));
    float rr = 1.0 / (r * 3.5 + 0.06) + 0.08 * z;
    vec2 q = vec2(cos(a), sin(a)) * rr;
    q /= ar;
    return fract(q + c);
}

vec3 limitHighlights(vec3 c) {
    float m = max(c.r, max(c.g, c.b));
    if (m > 0.9) c *= 0.9 / m;
    return c;
}

void main(void) {
    float aAcc = clamp(amp, 0.0, 4.0);
    float aInst = clamp(uamp, 0.0, 4.0);
    float ampMix = clamp(aAcc * 0.6 + aInst * 1.4, 0.0, 4.0);
    gAmp01 = clamp(ampMix / 2.5, 0.0, 1.0);
    gInstAmp = clamp(aInst / 2.5, 0.0, 1.0);
    gSlow = time_f * mix(0.15, 0.7, gAmp01);
    gFast = time_f * mix(0.6, 3.5, gAmp01);
    gDetail = time_f * mix(0.3, 2.0, gAmp01);
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= aspect;
    float rLen = length(uv);
    float r = pingPong(sin(rLen * gSlow), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + 2.0 * sin(gSlow * 0.33 + gAmp01 * 2.0);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(gSlow * 0.42 + gAmp01 * 3.0);
    kUV = fractalFold(kUV, foldZoom, gDetail, m, aspect);
    kUV = rotateUV(kUV, gSlow * 0.23 + gAmp01 * 1.1, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);

```

```

if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(gSlow * 0.2) * (PI * gSlow), 5.0);
float period = log(base) * pingPong(gSlow * PI, 5.0);
float tz = gSlow * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + gFast * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * (0.0015 + 0.001 * sin(gFast * 1.3)) * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
float oscK = pingPong(gSlow * PI, 3.0 + gAmp01 * 2.0);
vec3 kaleidoRGB = vec3(
    sin(rC.r * oscK),
    sin(gC.g * oscK * 1.05),
    sin(bC.b * oscK * 1.1)
);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + gFast * 1.2));
ring = ring * pingPong(gSlow * PI, 5.0);
float pulse = 0.5 + 0.5 * sin(gFast * 2.0 + rD * 28.0 + k * 12.0);
pulse *= mix(0.4, 1.4, gAmp01);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.10 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.07;
vec2 wh0 = wormholeUV(tc, m, aspect, gSlow);
vec2 wh1 = wormholeUV(tc + vec2(0.0009, 0.0), m, aspect, gSlow + 0.03);
vec2 wh2 = wormholeUV(tc - vec2(0.0009, 0.0), m, aspect, gSlow - 0.03);
vec3 whR = preBlendColor(wh0 + off);
vec3 whG = preBlendColor(wh1);
vec3 whB = preBlendColor(wh2 - off);
float oscW = pingPong(gFast * PI, 1.3 + gAmp01 * 1.2);
vec3 wormRGB = vec3(
    sin(whR.r * oscW),
    sin(whG.g * oscW * 1.05),
    sin(whB.b * oscW * 1.1)
);
float rCenter = length((tc - m) * ar);
float throat = sin(smoothstep(0.38, 0.06, rCenter) * pingPong(gSlow * PI, 5.0));
float swirlGate = smoothstep(0.9, 1.6, gSlow * 0.25 + 0.35 * sin(gSlow * 0.7))
    * pingPong(gSlow * PI, 4.0);
float gateBase = clamp(throat * (0.65 + 0.35 * pingPong(gSlow * PI, 5.0))
    + pingPong(swirlGate * PI, 8.0) * 0.15, 0.0, 1.0);
float hitBoost = clamp(aInst * 0.6, 0.0, 1.2);
float gate = clamp(gateBase * (0.6 + 0.9 * gAmp01) + hitBoost, 0.0, 1.0);
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = mix(outCol, wormRGB, gate * 0.85);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
outCol *= mix(0.85, 1.20, gAmp01);
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
finalRGB = limitHighlights(finalRGB);
finalRGB = clamp(finalRGB, 0.0, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);

```

```

    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #407 - frac shader03 wormhole amp

File: frac_shader03_wormhole_amp.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float gAmp01;
float gSlow;
float gFast;
float gDetail;
float gInstAmp;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;

```

```

    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

```

```

}
float diamondRadius(vec2 p) {
    float f = mix(0.5, 2.0, gAmp01);
    p = sin(abs(p * f));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
vec2 wormholeUV(vec2 uv, vec2 c, float aspect, float t) {
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float swirl = 0.22 / r;
    float spinBase = 0.15 + 0.35 * gAmp01;
    float spinHit = 0.2 + 2.2 * gInstAmp;
    a += t * (spinBase + spinHit) + swirl;
    float z = 0.65 + 0.35 * sin(t * 0.6)
        + 0.15 * sin(r * 12.0 - t * (2.5 + 1.5 * gAmp01 + 2.0 * gInstAmp));
    float rr = 1.0 / (r * 3.5 + 0.06) + 0.08 * z;
    vec2 q = vec2(cos(a), sin(a)) * rr;
    q /= ar;
    return fract(q + c);
}
vec3 limitHighlights(vec3 c) {
    float m = max(c.r, max(c.g, c.b));
    if (m > 0.9) c *= 0.9 / m;
    return c;
}
void main(void) {
    float aAcc = clamp(amp, 0.0, 4.0);
    float aInst = clamp(uamp, 0.0, 4.0);
    float ampMix = clamp(aAcc * 0.6 + aInst * 1.4, 0.0, 4.0);
    gAmp01 = clamp(ampMix / 2.5, 0.0, 1.0);
    gInstAmp = clamp(aInst / 2.5, 0.0, 1.0);
    gSlow = time_f * mix(0.15, 0.7, gAmp01);
    gFast = time_f * mix(0.6, 3.5, gAmp01);
    gDetail = time_f * mix(0.3, 2.0, gAmp01);
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * gSlow), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + 2.0 * sin(gSlow * 0.33 + gAmp01 * 2.0);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(gSlow * 0.42 + gAmp01 * 3.0);
    kUV = fractalFold(kUV, foldZoom, gDetail, m, aspect);
    kUV = rotateUV(kUV, gSlow * 0.23 + gAmp01 * 1.1, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);

```

```

if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(gSlow * 0.2) * (PI * gSlow), 5.0);
float period = log(base) * pingPong(gSlow * PI, 5.0);
float tz = gSlow * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35
            + 0.35 * sin(rD * 18.0 + gFast * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * (0.0015 + 0.001 * sin(gFast * 1.3))
            * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + gFast * 1.2));
ring = ring * pingPong(gSlow * PI, 5.0);
float pulse = 0.5 + 0.5 * sin(gFast * 2.0 + rD * 28.0 + k * 12.0);
pulse *= mix(0.4, 1.4, gAmp01);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.10
            + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.07;
vec2 wh0 = wormholeUV(tc, m, aspect, gSlow);
vec2 wh1 = wormholeUV(tc + vec2(0.0009, 0.0), m, aspect, gSlow + 0.03);
vec2 wh2 = wormholeUV(tc - vec2(0.0009, 0.0), m, aspect, gSlow - 0.03);
vec3 whR = preBlendColor(wh0 + off);
vec3 whG = preBlendColor(wh1);
vec3 whB = preBlendColor(wh2 - off);
vec3 wormRGB = vec3(whR.r, whG.g, whB.b);
float rCenter = length((tc - m) * ar);
float throat = smoothstep(0.38, 0.06, rCenter);
float swirlGate = smoothstep(0.9, 1.6,
                             gSlow * 0.25 + 0.35 * sin(gSlow * 0.7));
float gateBase = clamp(throat * (0.55 + 0.45 * pingPong(gSlow * PI, 5.0))
                      + swirlGate * 0.15, 0.0, 1.0);
float hitBoost = clamp(aInst * 0.6, 0.0, 1.2);
float gate = clamp(gateBase * (0.6 + 0.9 * gAmp01) + hitBoost, 0.0, 1.0);
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = mix(outCol, wormRGB, gate * 0.85);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
outCol *= mix(0.85, 1.20, gAmp01);
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
finalRGB = limitHighlights(finalRGB);
finalRGB = clamp(finalRGB, 0.0, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #469 - gem fish

File: gem-fish.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to audio input (amplitude/volume) for interactive control. 'swirl' increases rotation based on distance and time 'zoom' pulls the texture coordinates inward over time Offset R, G, and B slightly for chromatic aberration

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Driven by: time, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp;
uniform float uamp;
void main(void) {
    vec2 uv = tc - 0.5;
    float dist = length(uv);
    float angle = atan(uv.y, uv.x);
    float swirl = angle + (dist * 5.0) + (time_f * 0.5 * amp);
    float zoom = dist * (0.5 + sin(time_f * 0.2) * 0.2);
    vec2 warpedUV;
    warpedUV.x = cos(swirl) * zoom;
    warpedUV.y = sin(swirl) * zoom;
    vec3 finalCol = vec3(0.0);
    int samples = 12;
    float blurSize = 0.1 * (amp + uamp);
    for(int i = 0; i < samples; i++) {
        float slice = float(i) / float(samples);
        float scale = 1.0 - (slice * blurSize);
        vec2 rUV = (warpedUV * scale * 0.95) + 0.5;
        vec2 gUV = (warpedUV * scale * 1.00) + 0.5;
        vec2 bUV = (warpedUV * scale * 1.05) + 0.5;
        finalCol.r += texture(samp, rUV).r;
        finalCol.g += texture(samp, gUV).g;
        finalCol.b += texture(samp, bUV).b;
    }
    finalCol /= float(samples);
    float centerGlow = smoothstep(0.4, 0.0, dist) * (1.0 + uamp);
    finalCol += vec3(0.6, 0.8, 1.0) * centerGlow * 0.5;
    color = vec4(finalCol, 1.0);
}
```

Shader #471 - gem image

File: gem-image.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Responds to mouse position and audio input (amplitude/volume) for interactive control. Assuming general amplitude/scale Assuming intensity/speed multiplier 'uamp' and 'time_f' drive the warping

Technique: 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Radial vignette falloff - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
void main(void) {
    vec2 uv = tc - 0.5;
    uv = abs(uv);
    float t = time_f * (0.5 + uamp);
    for(int i = 0; i < 3; i++) {
        float s = sin(t * 0.2);
        float c = cos(t * 0.2);
        uv *= mat2(c, -s, s, c);
        uv = abs(uv * 1.1) - 0.2 * amp;
    }
    vec2 warpedTC = uv + 0.5;
    warpedTC = abs(mod(warpedTC - 1.0, 2.0) - 1.0);
    vec4 texColor = texture(samp, warpedTC);
    float vignette = smoothstep(0.8, 0.2, length(tc - 0.5));
    texColor.rgb *= vec3(0.8, 0.95, 1.2);
    texColor.rgb *= (vignette + 0.2);
    color = texColor;
}

```

Shader #480 - gem glass

File: gem_glass.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to audio input (amplitude/volume) for interactive control. Calculate the Normal (the direction the "glass" surface faces) The spiral is now "distorted" by the surface normals Refraction in glass usually splits light into its component colors

Technique: Multi-tap convolution blur (6 texture samples) - Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Radial vignette falloff - Driven by: time, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp;
uniform float uamp;
void main(void) {
    vec2 uv = tc;
    float pulse = sin(time_f * 0.5) * 0.1 + 0.5;
    float delta = 0.01;
    float h = dot(texture(samp, uv).rgb, vec3(0.33));
    float h1 = dot(texture(samp, uv + vec2(delta, 0.0)).rgb, vec3(0.33));
    float h2 = dot(texture(samp, uv + vec2(0.0, delta)).rgb, vec3(0.33));
    vec2 normal = vec2(h1 - h, h2 - h);
    vec2 centeredUV = uv - 0.5;
    float dist = length(centeredUV);

```

```

float angle = atan(centeredUV.y, centeredUV.x);
float spiral = angle + (dist * 8.0) + (time_f * 0.2) + (normal.x * 2.0 * amp);
float zoom = dist * (0.8 + uamp * 0.2);
vec2 glassUV;
glassUV.x = cos(spiral) * zoom + 0.5 + (normal.x * 0.05);
glassUV.y = sin(spiral) * zoom + 0.5 + (normal.y * 0.05);
vec3 finalCol;
float refractor = 0.02 * (amp + uamp + 0.1);
finalCol.r = texture(samp, glassUV + (normal * refractor)).r;
finalCol.g = texture(samp, glassUV).g;
finalCol.b = texture(samp, glassUV - (normal * refractor)).b;
float specular = pow(max(0.0, 1.0 - length(normal * 20.0)), 10.0);
finalCol += vec3(1.0) * specular * 0.4;
finalCol *= smoothstep(1.2, 0.3, dist);
color = vec4(finalCol, 1.0);
}

```

Shader #483 - gem p

File: gem_p.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to audio input (amplitude/volume) for interactive control. Create 3 layers of interference patterns Domain Warping: Use the spiral math to bend the texture coordinates This makes the "samp" texture melt and swirl

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp;
uniform float uamp;
vec3 getRainbow(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 0.7, 0.4);
    vec3 d = vec3(0.0, 0.15, 0.20);
    return a + b * cos(6.28318 * (c * t + d + time_f * 0.5));
}
vec2 rotate(vec2 p, float a) {
    float s = sin(a);
    float c = cos(a);
    return vec2(p.x * c - p.y * s, p.x * s + p.y * c);
}
void main(void) {
    vec2 uv = tc;
    vec2 centeredUV = (uv * 2.0 - 1.0);
    centeredUV.x *= iResolution.x / iResolution.y;
    float r = length(centeredUV);
    float angle = atan(centeredUV.y, centeredUV.x);
    float noise = sin(angle * 5.0 + time_f) * 0.1;
    noise += sin(r * 20.0 - time_f * 2.0) * 0.05;
    float spiral = sin(15.0 * r - time_f * 4.0 + (angle * 4.0));
    float twist = cos(10.0 * r + time_f + spiral);
}

```

```

    vec2 distortedTC = tc;
    distortedTC.x += (spiral * 0.03) * cos(angle);
    distortedTC.y += (twist * 0.03) * sin(angle);
    vec4 tex = texture(samp, distortedTC);
    float colorPattern = sin(r * 12.0 - time_f * 3.0 + angle * 2.0);
    colorPattern += cos(uv.x * 10.0 + time_f) * 0.5;
    colorPattern += sin(uv.y * 10.0 - time_f) * 0.5;
    vec3 psychedelicOverlay = getRainbow(colorPattern * 0.5 + r);
    vec3 finalRGB = mix(tex.rgb, psychedelicOverlay, 0.6);
    finalRGB += psychedelicOverlay * (spiral * 0.2);
    color = vec4(finalRGB, tex.a);
}

```

Shader #586 - HyperFcousAmp

File: HyperFcousAmp.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position and audio input (amplitude/volume) for interactive control. Applies matrix transformations. Bass / Volume (0.0 to 1.0) Treble / High Freq (0.0 to 1.0) Mouse Interaction: If mouse is pressed, move the vortex center

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
const float PI = 3.14159265359;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 3; i++) {
        v += a * noise(p);
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}

```

```

vec3 electricPalette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.00, 0.33, 0.67);
    return a + b * cos(6.28318 * (c * t + d));
}

vec3 renderLayer(vec2 uv, float t, float bassStr, float trebleStr) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.5) {
        center = iMouse.xy / iResolution;
    }
    vec2 p = (uv - center) * ar;
    float len = length(p);
    float twistBase = 3.0 + (bassStr * 4.0);
    float angle = atan(p.y, p.x) + twistBase / (len + 0.1) + t * (0.5 + bassStr);
    vec2 twistedP = vec2(cos(angle), sin(angle)) * len;
    float rippleWavelength = 8.0;
    float rippleSpeed = 4.0 + (trebleStr * 10.0);
    float noiseVal = fbm(twistedP * 5.0 + t);
    float rippleX = sin(twistedP.x * rippleWavelength + t * rippleSpeed + noiseVal * 2.0);
    float rippleY = cos(twistedP.y * rippleWavelength + t * rippleSpeed - noiseVal * 2.0);
    float rippleAmp = 0.05 + 0.05 * noiseVal * (1.0 + trebleStr * 2.0);
    vec2 rippledP = twistedP + vec2(rippleX, rippleY) * rippleAmp;
    vec3 p3 = vec3(rippledP, 1.0);
    float shake = bassStr * 0.1 * sin(t * 20.0);
    mat3 R = rotX(t * 0.2 + shake) * rotY(t * 0.15) * rotZ(t * 0.1);
    vec3 r = R * p3;
    float zScale = 1.0 / (1.0 + r.z * 0.5);
    vec2 projUV = r.xy * zScale;
    vec2 tiledUV = 1.0 - abs(1.0 - 2.0 * (projUV + 0.5));
    tiledUV = tiledUV - floor(tiledUV);
    float aberrationStr = 0.01 + (0.04 + bassStr * 0.05) * (1.0 / (len + 0.5));
    vec2 offset = vec2(aberrationStr, 0.0);
    vec3 texR = texture(samp, clamp(tiledUV - offset, 0.0, 1.0)).rgb;
    vec3 texG = texture(samp, clamp(tiledUV, 0.0, 1.0)).rgb;
    vec3 texB = texture(samp, clamp(tiledUV + offset, 0.0, 1.0)).rgb;
    vec3 layerColor = vec3(texR.r, texG.g, texB.b);
    float glowMask = smoothstep(0.6, 0.95, abs(rippleX * rippleY));
    vec3 palette = electricPalette(len - t);
    return mix(layerColor, layerColor + palette, glowMask * (0.8 + bassStr * 0.5));
}

void main(void) {
    float simBass = (amp <= 0.001) ? abs(sin(time_f * 4.0)) * 0.5 : amp;
    float simTreble = (uamp <= 0.001) ? abs(cos(time_f * 8.0)) * 0.5 : uamp;
    vec3 col0 = renderLayer(tc, time_f, simBass, simTreble);
    vec3 col1 = renderLayer(tc, time_f - 0.1, simBass * 0.8, simTreble * 0.8);
    vec3 col2 = renderLayer(tc, time_f - 0.2, simBass * 0.6, simTreble * 0.6);
    vec3 finalColor = col0 + (col1 * 0.5) + (col2 * 0.25);
    finalColor = finalColor / (1.0 + finalColor * 0.4);
    color = vec4(finalColor, 1.0);
}

```

Shader #755 - pencil draw

File: pencil_draw.glsl

Performs edge detection on the input image, highlighting boundaries and contours. Responds to

mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Minimum edge strength required to show color (filters out noise in black areas) How fast the neon colors cycle How thick the edge check is (1.0 is standard pixel neighbor)

Technique: Sobel/Laplacian convolution kernel for edge detection - Multi-tap convolution blur (8 texture samples) - Indexed color palette look-up table - Hermite smoothstep interpolation -
Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
const float THICKNESS = 1.0;
const float THRESHOLD = 0.1;
const float SPEED = 2.0;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
#define THICKNESS 1.0
#define THRESHOLD 0.1
#define SPEED 2.0
float getGray(vec4 c) {
    return dot(c.rgb, vec3(0.299, 0.587, 0.114));
}
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d));
}
void main(void) {
    vec2 texel = vec2(THICKNESS) / iResolution.xy;
    float t00 = getGray(texture(samp, tc + vec2(-1, -1) * texel));
    float t10 = getGray(texture(samp, tc + vec2( 0, -1) * texel));
    float t20 = getGray(texture(samp, tc + vec2( 1, -1) * texel));
    float t01 = getGray(texture(samp, tc + vec2(-1,  0) * texel));
    float t21 = getGray(texture(samp, tc + vec2( 1,  0) * texel));
    float t02 = getGray(texture(samp, tc + vec2(-1,  1) * texel));
    float t12 = getGray(texture(samp, tc + vec2( 0,  1) * texel));
    float t22 = getGray(texture(samp, tc + vec2( 1,  1) * texel));
    float Gx = t00 + 2.0 * t10 + t20 - t02 - 2.0 * t12 - t22;
    float Gy = t00 + 2.0 * t01 + t02 - t20 - 2.0 * t21 - t22;
    float edge = sqrt(Gx * Gx + Gy * Gy);
    edge = smoothstep(THRESHOLD, THRESHOLD + 0.1, edge);
    vec3 neon = palette(length(tc) + iTime * SPEED * 0.2);
    color = vec4(neon * edge, 1.0);
}
```

Shader #756 - pencil draw lines

File: pencil_draw_lines.glsl

Performs edge detection on the input image, highlighting boundaries and contours. Responds to mouse position and audio input (amplitude/volume) for interactive control. Discards pure black pixels for transparency. Computes luminance for brightness-based effects. Noise gate (removes background) Speed of color cycling How strong the distortion is (try 0.01 to 0.05)

Technique: Sobel/Laplacian convolution kernel for edge detection - Multi-tap convolution blur (8 texture samples) - Indexed color palette look-up table - Hermite smoothstep interpolation -
Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float THICKNESS = 1.0;
const float THRESHOLD = 0.15;
const float NEON_SPEED = 3.0;
const float WOBBLE_AMP = 0.02;
const float WOBBLE_FREQ = 10.0;
const float WOBBLE_SPEED = 2.0;
float getGray(vec4 c) {
    return dot(c.rgb, vec3(0.299, 0.587, 0.114));
}
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d));
}
void main(void) {
    vec2 wobbleOffset;
    wobbleOffset.x = sin(tc.y * WOBBLE_FREQ + iTime * WOBBLE_SPEED) * WOBBLE_AMP;
    wobbleOffset.y = cos(tc.x * WOBBLE_FREQ + iTime * WOBBLE_SPEED) * WOBBLE_AMP;
    vec2 distortedTC = tc + wobbleOffset;
    vec2 texel = vec2(THICKNESS) / iResolution.xy;
    float t00 = getGray(texture(samp, distortedTC + vec2(-1, -1) * texel));
    float t10 = getGray(texture(samp, distortedTC + vec2( 0, -1) * texel));
    float t20 = getGray(texture(samp, distortedTC + vec2( 1, -1) * texel));
    float t01 = getGray(texture(samp, distortedTC + vec2(-1,  0) * texel));
    float t21 = getGray(texture(samp, distortedTC + vec2( 1,  0) * texel));
    float t02 = getGray(texture(samp, distortedTC + vec2(-1,  1) * texel));
    float t12 = getGray(texture(samp, distortedTC + vec2( 0,  1) * texel));
```

```

float t22 = getGray(texture(samp, distortedTC + vec2( 1, 1) * texel));
float Gx = t00 + 2.0 * t10 + t20 - t02 - 2.0 * t12 - t22;
float Gy = t00 + 2.0 * t01 + t02 - t20 - 2.0 * t21 - t22;
float edge = sqrt(Gx * Gx + Gy * Gy);
if (edge < THRESHOLD) {
    color = vec4(0.0, 0.0, 0.0, 1.0);
    return;
}
edge = smoothstep(THRESHOLD, THRESHOLD + 0.1, edge);
vec3 neonColor = palette(length(distortedTC) + iTime * NEON_SPEED * 0.1);
color = vec4(neonColor * edge, 1.0);
}

```

Shader #922 - shader1

File: shader1.glsl

Uses distance fields with sinusoidal modulation to generate animated visual patterns. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation
 - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float uamp;
uniform float amp;
uniform vec4 iMouse;
void main(void) {
    vec2 mousePos = iMouse.xy / iResolution.xy;
    float dist = distance(tc, mousePos);
    float glitchStrength = exp(-dist * 10.0) * (uamp) * 4.0;
    vec2 glitchOffset = vec2(
        sin(time_f * 10.0 + tc.y * 20.0) * glitchStrength,
        cos(time_f * 15.0 + tc.x * 25.0) * glitchStrength
    );
    vec2 distortedTc = tc + glitchOffset;
    color = texture(samp, distortedTc);
    vec4 originalColor = texture(samp, tc);
    float blendFactor = smoothstep(0.0, 0.1, glitchStrength);
    color = mix(originalColor, color, blendFactor);
    color = vec4(sin(color.rgb * time_f), 1.0);
}

```

Shader #1000 - tdye

File: tdye.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to audio input (amplitude/volume) for interactive control. A simple swirl-based approach: 1. Shift coords so the center is at (0, 0). 2. Compute radius r and angle theta from the center. 3. Apply a time-varying distortion to theta based on r (and/or noise). 4.

Transform back, sample the texture, and output it. Shift the texture coords so (0.5, 0.5) is the center Compute polar coordinates

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float uamp;
uniform float amp;
void main(void)
{
    vec2 uv = tc - 0.5;
    float r = length(uv);
    float angle = atan(uv.y, uv.x);
    float swirlFreq = 12.0;
    float swirlAmp = 25.0 * uamp;
    float timeMult = 1.5;
    float swirl = sin(r * swirlFreq - time_f * timeMult) * swirlAmp * r;
    angle += swirl;
    uv = r * vec2(cos(angle), sin(angle));
    uv += 0.5;
    vec4 texColor = texture(samp, uv);
    color = texColor;
}
```

Shader #1001 - tdye2

File: tdye2.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float uamp;
uniform float amp;
float hash(vec2 p){return fract(sin(dot(p,vec2(127.1,311.7))))*43758.5453123);}
float noise(vec2 p){
    vec2 i=floor(p),f=fract(p);
    float a=hash(i),b=hash(i+vec2(1,0)),c=hash(i+vec2(0,1)),d=hash(i+vec2(1,1));
    vec2 u=f*f*(3.0-2.0*f);
    return mix(mix(a,b,u.x),mix(c,d,u.x),u.y);
}
float fbm(vec2 p){
    float v=0.0,a=0.5;
    mat2 m=mat2(1.6,1.2,-1.2,1.6);
    for(int i=0;i<5;i++){v+=a*noise(p);p=m*p+1.7;a*=0.5;}
}
```

```

    return v;
}
float pingPong(float x,float L){float m=mod(x,2.0*L);return m<=L?m:(2.0*L-m);}
vec3 grad4(vec3 c0, vec3 c1, vec3 c2, vec3 c3, float x){
    x=clamp(x,0.0,1.0);
    if(x<0.25) return mix(c0,c1,x/0.25);
    else if(x<0.50) return mix(c1,c2,(x-0.25)/0.25);
    else if(x<0.75) return mix(c2,c3,(x-0.50)/0.25);
    else return mix(c3,c0,(x-0.75)/0.25);
}
vec3 rgb2yc(vec3 c){
    float y=dot(c,vec3(0.299,0.587,0.114));
    float cb=(c.b - y)*0.565;
    float cr=(c.r - y)*0.713;
    return vec3(y,cb,cr);
}
vec3 yc2rgb(vec3 ycc){
    float y=ycc.x, cb=ycc.y, cr=ycc.z;
    return vec3(y + 1.403*cr, y - 0.344*cb - 0.714*cr, y + 1.770*cb);
}
void main(){
    vec2 uv=tc;
    vec2 res=iResolution;
    float t=time_f;
    vec2 p=uv-0.5;
    p.x*=res.x/max(res.y,1.0);
    float speed=0.25+0.45*uamp+0.25*amp;
    float zPhase=pingPong(t*speed,1.0);
    float minPow=-1.5;
    float maxPow=12.0;
    float scale=exp2(mix(minPow,maxPow,zPhase));
    float rot=0.25*t+0.12*sin(t*0.29)+0.08*sin(t*0.47);
    float cs=cos(rot), sn=sin(rot);
    mat2 R=mat2(cs,-sn,sn,cs);
    vec2 warp=(vec2(fbm(p*2.3+vec2(0.0,t*0.18)), fbm(p*2.0+vec2(t*0.21,0.0)))-0.5)*0.55;
    vec2 zoomPos=(R*p)*(3.0*scale)+warp;
    float ring=length(zoomPos);
    float angle=atan(zoomPos.y,zoomPos.x);
    float drift=0.23*sin(t*0.23)+0.18*sin(t*0.41)+0.12*sin(t*0.59);
    float bands=ring*(8.0+12.0*amp) + 0.9*sin(angle*3.0 + t*0.7) + fbm(zoomPos*1.7 + t*0.2)*1.8;
    float gCoord=fract(bands*0.25 + drift);
    vec3 G0=vec3(1.00,0.20,0.30);
    vec3 G1=vec3(1.00,0.90,0.20);
    vec3 G2=vec3(0.20,0.95,0.35);
    vec3 G3=vec3(0.25,0.45,1.00);
    vec3 dyeRGB=grad4(G0,G1,G2,G3,gCoord);
    dyeRGB=pow(dyeRGB,vec3(0.9))*1.25;
    vec2 reTex=(R*p)*(0.9*scale)+warp*0.15;
    reTex.x/= (res.x/max(res.y,1.0));
    vec2 sampleUV=fract(reTex+0.5);
    vec3 base=texture(samp,sampleUV).rgb;
    float Y=dot(base,vec3(0.299,0.587,0.114));
    vec3 dyeYC=rgb2yc(dyeRGB);
    float yBlend=mix(0.15,0.45,clamp(uamp,0.0,1.0));
    float Ymix=mix(Y, dot(dyeRGB,vec3(0.299,0.587,0.114)), yBlend);
    vec3 outYC=vec3(Ymix,dyeYC.yz);
    vec3 dyed=yc2rgb(outYC);
    float caAmt=0.0016+0.003*uamp;
    vec3 chroma;
    chroma.r=texture(samp,fract(sampleUV+vec2(caAmt,0.0))).r;
    chroma.g=dyed.g;
    chroma.b=texture(samp,fract(sampleUV-vec2(caAmt,0.0))).b;
}

```

```

    dyed=mix(dyed,chroma,0.18);
    float vign=smoothstep(1.15,0.35,length(p));
    dyed*=mix(1.0,0.93,vign);
    color=vec4(clamp(dyed,0.0,1.0),1.0);
}

```

Shader #1008 - test

File: test.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
float hash(float n){return fract(sin(n)*43758.5453123);}
float n2(vec2 p){
    vec2 i=floor(p), f=fract(p);
    float a=hash(i.x+i.y*57.0);
    float b=hash(i.x+1.0+(i.y)*57.0);
    float c=hash(i.x+(i.y+1.0)*57.0);
    float d=hash(i.x+1.0+(i.y+1.0)*57.0);
    vec2 u=f*f*(3.0-2.0*f);
    return mix(mix(a,b,u.x),mix(c,d,u.x),u.y);
}
vec3 hsv2rgb(vec3 c){
    vec4 K=vec4(1.0,2.0/3.0,1.0/3.0,3.0);
    vec3 p=abs(fract(c.xxx+K.xyz)*6.0-K.www);
    return c.z*mix(K.xxx,clamp(p-K.xxx,0.0,1.0),c.y);
}
float pingPong(float x,float l){float m=mod(x,l*2.0);return m<=l?m:l*2.0-m;}
void main(){
    vec2 R=iResolution;
    vec2 uv=tc;
    vec2 cn=R.xy/max(R.x,R.y);
    vec2 m=vec2(iMouse.x/iResolution.x,iMouse.y/iResolution.y);
    m=clamp(m,0.0,1.0);
    float t=time_f+iTime+float(iFrame)*iTimeDelta*0.25;
    float fr= max(iFrameRate,1.0);
    float dateSeed=dot(iDate,vec4(1.0,31.0,12.0,0.001));

```

```

float chs=0.0;
for(int i=0;i<4;i++){
    chs+=sin(iChannelTime[i]*0.73+float(i)*1.1)+
        0.001*length(iChannelResolution[i].xy)+
        0.000001*iChannelResolution[i].z;
}
float sr=clamp(iSampleRate/48000.0,0.25,2.0);
float aMix=clamp(amp*0.5+uamp*0.5,0.0,10.0);
vec2 c=m*0.75+0.125;
float clickT=length(iMouseClicked-0.5)*2.0;
float speed=0.2+0.4*sr+0.15*sin(dateSeed);
float kSides=3.0+mod(float(iFrame),9.0);
vec2 p=(uv-c);
p.x*=R.x/R.y;
float r=length(p);
float ang=atan(p.y,p.x);
ang=abs(fract(ang/3.14159265*kSides)*2.0-1.0);
float kale=sin(ang*6.28318+chs*0.5+t*2.0);
float ripple=sin(20.0*r- t*12.0 - clickT*10.0)*0.02*(1.0/(1.0+20.0*r));
float swirl=(kale*0.25+0.25)*r;
float wave=sin((uv.x+uv.y*1.7+chs*0.1+t*speed*fr*0.01)*40.0*sr)*0.005;
vec2 nWarp=vec2(n2(uv*8.0+vec2(t)),n2(uv*8.0+vec2(-t)));
vec2 warp=(nWarp-0.5)*0.08*aMix;
vec2 uv1=uv+warp+vec2(wave,-wave)+normalize(p+0.0001)*ripple;
vec2 uv2=uv+vec2(swirl,-swirl)*0.15*aMix;
vec2 uv3=uv+vec2(sin(t+uv.y*10.0),cos(t+uv.x*10.0))*0.01*(1.0+0.5*sin(dateSeed+chs));
float zoom=0.9+0.2*sin(pingPong(t*0.5+chs,2.0)*3.14159);
uv1=(uv1-c)*zoom+c;
uv2=(uv2-c)*zoom+c;
uv3=(uv3-c)*zoom+c;
vec4 s1=texture(samp,fract(uv1));
vec4 s2=texture(samp,fract(uv2));
vec4 s3=texture(samp,fract(uv3));
float hueBase=fract(t*0.05+dateSeed*0.1+chs*0.05);
float sat=0.75+0.25*sin(chs+t+sr);
float val=0.6+0.4*sin(t*0.7+dot(uv,vec2(3.1,2.7)));
vec3 tint=HSV2rgb(vec3(hueBase+length(p)*0.25+0.1*sin(iTimeDelta*60.0),sat,val));
vec4 mixA=mix(s1,s2,0.5+0.5*sin(t+chs));
vec4 mixB=mix(s2,s3,0.5+0.5*cos(t*1.3));
vec4 mixC=mix(mixA,mixB,0.5+0.5*sin(t*0.7+dateSeed));
float vign=1.0-smoothstep(0.6,1.1,r);
vec3 col=mixC.rgb*tint*(0.8+0.2*sin(float(iFrame)*0.01+chs))+0.25*tint;
col*=vign;
color=vec4(col,1.0);
}

```

Shader #1009 - test cache

File: test_cache.glsl

Blends the current frame with cached previous frames to create motion trails and temporal echoes. Responds to mouse position and audio input (amplitude/volume) for interactive control. Dynamic UV distortion Distortion based on sine waves Sample textures with modulation

Technique: Temporal feedback using cached frame buffers - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;

```

```

uniform float time_f;
uniform sampler2D samp;
uniform sampler2D samp1;
uniform sampler2D samp2;
uniform sampler2D samp3;
uniform sampler2D samp4;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
vec3 rainbowGradient(float t) {
    float r = 0.5 + 0.5 * cos(6.2831 * (t + 0.0) + 0.0);
    float g = 0.5 + 0.5 * cos(6.2831 * (t + 0.3) + 2.0);
    float b = 0.5 + 0.5 * cos(6.2831 * (t + 0.6) + 4.0);
    return vec3(r, g, b);
}
void main(void) {
    vec2 p = tc - 0.5;
    p.x *= iResolution.x / iResolution.y;
    float len = length(p);
    vec2 distortedTC = tc + vec2(
        sin(time_f * 0.5 + tc.y * 8.0) * 0.03 * amp,
        cos(time_f * 0.4 + tc.x * 10.0) * 0.03 * uamp
    );
    vec4 tex1 = texture(samp1, distortedTC);
    vec4 tex2 = texture(samp2, distortedTC * 1.2);
    vec4 tex3 = texture(samp3, tc * 0.8 + tex1.xy * 0.05);
    vec4 tex4 = texture(samp4, tc * 1.1 - tex2.xy * 0.03);
    vec4 baseTex = texture(samp, distortedTC);
    float energyWave = sin(len * 8.0 - time_f * 3.0) * 0.5 + 0.5;
    vec3 energyColor = rainbowGradient(tc.x + tc.y + time_f * 0.2);
    energyColor = mix(energyColor, vec3(1.0, 0.2, 0.8), energyWave);
    vec3 finalColor = mix(tex1.rgb, tex2.rgb, 0.5) * 0.8
        + tex3.rgb * 0.4
        + tex4.rgb * 0.3
        + energyColor * 0.5;
    float mouseDist = length(tc - iMouse.xy / iResolution);
    finalColor += smoothstep(0.2, 0.0, mouseDist) * vec3(0.8, 0.4, 1.0);
    color = vec4(finalColor * baseTex.rgb, 1.0);
}

```

Shader #1015 - today

File: today.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Control warp intensity with audio values Wrap texture coordinates

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;

```

```

uniform float uamp;
void main(void) {
    vec2 uv = tc - 0.5;
    float radius = length(uv);
    float angle = atan(uv.y, uv.x);
    float audioEffect = uamp * 10.5;
    float wave = sin(radius * 10.0 + time_f * 2.0) * audioEffect;
    angle += wave;
    vec2 tunnelUV = vec2(cos(angle), sin(angle)) * radius + 0.5;
    tunnelUV = fract(tunnelUV);
    color = texture(samp, tunnelUV);
}

```

Shader #1063 - wormhole audio

File: wormhole_audio.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. AUDIO: Smooth energy accelerates color cycle AUDIO: Mids increase the swirl distortion AUDIO: Bass aggressively pushes the depth/zoom

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - RGB $\leftrightarrow$ HSV color space conversion - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {

```

```

    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * 0.35 + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3 + (amp_smooth * 4.0));
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = 0.3 + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
}

```

```

    grad = softTone(grad);
    return grad;
}
float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
vec2 wormholeUV(vec2 uv, vec2 c, float aspect, float t) {
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    float r = length(p) + 1e-6;
    float a = atan(p.y, p.x);
    float swirl = (0.22 + amp_mid * 0.4) / r;
    a += t * 0.35 + swirl;
    float z = 0.65 + 0.35 * sin(t * 0.6) + 0.15 * sin(r * 12.0 - t * 3.0) + (amp_low * 1.2);
    float rr = 1.0 / (r * 3.5 + 0.06) + 0.08 * z;
    vec2 q = vec2(cos(a), sin(a)) * rr;
    q /= ar;
    return fract(q + c);
}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - 0.25, r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + 2.0 * sin(time_f * 0.33) + (amp_low * 6.0);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + 0.55 * sin(time_f * 0.42) + (amp_mid * 0.6);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * 0.23, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * 0.65 + (amp_smooth * 2.5);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * (0.0015 + 0.001 * sin(time_f * 1.3) + (amp_high * 0.02)) * vec2(1.0, 1.0 / aspect);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));

```



```

vign = mix(0.9, 1.15, vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0);
float pulse = 0.5 + 0.5 * sin(time_f * 2.0 + rD * 28.0 + k * 12.0) + (amp_peak * 0.8);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * (0.18 + amp_rms * 0.6) + pow(max(outCol - 0.6, 0.0), vec3(2.0)) *
↳ (0.12 + amp_peak * 0.4);
vec2 wh0 = wormholeUV(tc, m, aspect, pingPong((time_f * PI), 5.0));
vec2 wh1 = wormholeUV(tc + vec2(0.0009, 0.0), m, aspect, time_f + 0.03);
vec2 wh2 = wormholeUV(tc - vec2(0.0009, 0.0), m, aspect, time_f - 0.03);
vec3 whR = preBlendColor(wh0 + off);
vec3 whG = preBlendColor(wh1);
vec3 whB = preBlendColor(wh2 - off);
vec3 wormRGB = vec3(whR.r, whG.g, whB.b);
float rCenter = length((tc - m) * ar);
float d00 = length(ar * (vec2(0.0, 0.0) - m));
float d10 = length(ar * (vec2(1.0, 0.0) - m));
float d01 = length(ar * (vec2(0.0, 1.0) - m));
float d11 = length(ar * (vec2(1.0, 1.0) - m));
float cornerR = max(max(d00, d10), max(d01, d11));
float phase = pingPong(time_f * 0.6, 1.0);
float holeR = mix(max(1.5 / max(iResolution.x, iResolution.y), 0.0008), cornerR, phase) + (amp_low *
↳ 0.25);
float rimW = mix(0.02, 0.12, smoothstep(0.0, 1.0, phase)) * cornerR + (amp_high * 0.08);
float gate = smoothstep(holeR, holeR - rimW, rCenter);
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = mix(outCol, wormRGB, gate * 0.95);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
color = vec4(finalRGB, baseTex.a);
}

```

Liquid & Fluid (56 shaders)

Shader #92 - c ripple

File: c_ripple.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0);
    float dist = sqrt(tc.x * tc.x + tc.y * tc.y);

```

```

    dist = sin(dist * time_t);
    float ripple = sin(dist * 10.0 - time_f * 5.0);
    vec2 offset = tc + ripple * 0.02;
    vec4 texColor = texture(samp, offset);
    color = texColor;
}

```

Shader #159 - cripple

File: cripple.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    vec2 center = iResolution * 0.5;
    float distance = length(uv * iResolution - center);
    float ripple = 0.05 * cos(distance * 10.0 - time_f * 5.0);
    uv += ripple;
    color = texture(samp, uv);
}

```

Shader #160 - cripple2

File: cripple2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    vec2 center = iResolution * 0.5;
    float distance = length(uv * iResolution - center);
    distance = sin(distance * time_f);
    float ripple = 0.05 * cos(distance * 10.0 - time_f * 5.0);
    uv += ripple;
    color = texture(samp, uv);
}

```

Shader #175 - cxripple

File: cxripple.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc - 0.5;
    float dist = length(uv);
    float ripple = sin(dist * 20.0 - time_f * 5.0);
    float angle = ripple + time_f * 2.0;
    float s = sin(angle);
    float c = cos(angle);
    vec2 offset = vec2(c, s) * ripple * 0.05;
    vec2 pattern = sin(fract(uv * 10.0 + offset) * time_f);
    float brightness = fract(pattern.x * pattern.y * 10.0);
    vec3 colorValue = vec3(0.5 + 0.5 * sin(10.0 * dist + time_f), 0.5 + 0.5 * cos(10.0 * dist + time_f),
        ↪ brightness);
    vec4 texColor = texture(samp, uv + 0.5 + offset);
    color = vec4(colorValue, 1.0) * texColor;
}
```

Shader #330 - eye2

File: eye2.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
float s3 = 1.7320508;
float hash12(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453);
}
vec2 hash22(vec2 p) {
    float n = sin(dot(p, vec2(269.5, 183.3)));
    return fract(vec2(262144.0, 32768.0) * n);
}
vec4 sampleRipple(vec2 uv) {
    float speedR = 5.0, amplitudeR = 0.03, wavelengthR = 10.0;
    float speedG = 6.5, amplitudeG = 0.025, wavelengthG = 12.0;
    float speedB = 4.0, amplitudeB = 0.035, wavelengthB = 8.0;
    float rR = sin(uv.x * wavelengthR + time_f * speedR) * amplitudeR;
    rR += sin(uv.y * wavelengthR * 0.8 + time_f * speedR * 1.2) * amplitudeR;
    vec2 uvR = uv + vec2(rR, rR);
    float rG = sin(uv.x * wavelengthG * 1.5 + time_f * speedG) * amplitudeG;
```

```

    rG += sin(uv.y * wavelengthG * 0.3 + time_f * speedG * 0.7) * amplitudeG;
    vec2 uvG = uv + vec2(rG, -rG * 0.5);
    float rB = sin(uv.x * wavelengthB * 0.5 + time_f * speedB) * amplitudeB;
    rB += sin(uv.y * wavelengthB * 1.7 + time_f * speedB * 1.3) * amplitudeB;
    vec2 uvB = uv + vec2(rB * 0.3, rB);
    vec4 c = texture(samp, uv);
    c.r = texture(samp, uvR).r;
    c.g = texture(samp, uvG).g;
    c.b = texture(samp, uvB).b;
    return c;
}

float sdHex(vec2 p, float r) {
    p = abs(p);
    return max(dot(p, vec2(0.8660254, 0.5)), p.y) - r;
}

vec4 compoundSample(vec2 uv) {
    float density = 22.0;
    vec2 g = vec2(density, density * 0.8660254);
    vec2 puv = uv * g;
    float row = floor(puv.y);
    float shift = mod(row, 2.0) * 0.5;
    float x = puv.x - shift;
    float col = floor(x);
    vec2 cell = vec2(col, row);
    vec2 bestC = cell;
    float bestD = 1e9;
    for (int j = -1; j <= 1; ++j) {
        for (int i = -1; i <= 1; ++i) {
            vec2 nb = cell + vec2(i, j);
            float sh = mod(nb.y, 2.0) * 0.5;
            vec2 center = vec2(nb.x + sh + 0.5, nb.y + 0.5);
            vec2 lp = vec2(puv.x, puv.y) - center;
            float d = dot(lp, lp);
            if (d < bestD) { bestD = d; bestC = nb; }
        }
    }

    float shBest = mod(bestC.y, 2.0) * 0.5;
    vec2 cCenter = vec2(bestC.x + shBest + 0.5, bestC.y + 0.5);
    vec2 lp = vec2(puv.x, puv.y) - cCenter;
    vec2 lpN = lp * 2.0;
    float edge = sdHex(lpN, 0.98);
    float seam = smoothstep(0.02, 0.0, 0.98 - abs(edge));
    float mask = 1.0 - smoothstep(0.0, 0.02, edge);
    vec2 cellId = bestC;
    vec2 jitter = (hash22(cellId) - 0.5) * 0.7;
    float tw = time_f * (1.0 + hash12(cellId) * 0.5);
    vec2 micro = normalize(vec2(cos(tw + jitter.x*6.2831), sin(tw + jitter.y*6.2831)));
    float r = clamp(1.0 - length(lp) * 2.0, 0.0, 1.0);
    float angleAmt = 0.02 + 0.02 * hash12(cellId + 3.7);
    vec2 baseOffset = micro * angleAmt * (0.15 + 0.85 * r * r);
    float disp = 0.003 + 0.004 * (1.0 - r);
    vec2 offR = baseOffset + vec2(disp, 0.0);
    vec2 offG = baseOffset;
    vec2 offB = baseOffset + vec2(-disp, 0.0);
    vec2 back = (cCenter + lp) / g;
    vec2 uvR = clamp(back + offR, 0.0, 1.0);
    vec2 uvG = clamp(back + offG, 0.0, 1.0);
    vec2 uvB = clamp(back + offB, 0.0, 1.0);
    vec3 c;
    vec4 sr = sampleRipple(uvR);
    vec4 sg = sampleRipple(uvG);
    vec4 sb = sampleRipple(uvB);

```

```

    c = vec3(sr.r, sg.g, sb.b);
    float theta = atan(lp.y, lp.x);
    float pol = 0.08 * (0.5 + 0.5 * cos(2.0 * theta + time_f * 1.7 + hash12(cellId) * 6.2831));
    c *= (1.0 + pol * r);
    float rim = smoothstep(0.0, 0.02, edge + 0.02);
    c *= mix(1.0, 0.15, rim);
    float grain = hash12(cellId + floor(time_f * 10.0)) * 0.06;
    c += (grain - 0.03) * r;
    return vec4(c, mask * (1.0 - rim) * seam);
}

void main() {
    vec4 cs = compoundSample(tc);
    vec3 base = sampleRipple(tc).rgb;
    vec3 outc = mix(base * 0.2, cs.rgb, cs.a);
    float v = smoothstep(1.0, 0.7, length(tc - 0.5));
    outc *= v;
    color = vec4(outc, 1.0);
}

```

Shader #331 - eyes

File: eyes.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Driven by: time

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
vec4 sampleRipple(vec2 uv) {
    float speedR = 5.0, amplitudeR = 0.03, wavelengthR = 10.0;
    float speedG = 6.5, amplitudeG = 0.025, wavelengthG = 12.0;
    float speedB = 4.0, amplitudeB = 0.035, wavelengthB = 8.0;
    float rR = sin(uv.x * wavelengthR + time_f * speedR) * amplitudeR;
    rR += sin(uv.y * wavelengthR * 0.8 + time_f * speedR * 1.2) * amplitudeR;
    vec2 uvR = uv + vec2(rR, rR);
    float rG = sin(uv.x * wavelengthG * 1.5 + time_f * speedG) * amplitudeG;
    rG += sin(uv.y * wavelengthG * 0.3 + time_f * speedG * 0.7) * amplitudeG;
    vec2 uvG = uv + vec2(rG, -rG * 0.5);
    float rB = sin(uv.x * wavelengthB * 0.5 + time_f * speedB) * amplitudeB;
    rB += sin(uv.y * wavelengthB * 1.7 + time_f * speedB * 1.3) * amplitudeB;
    vec2 uvB = uv + vec2(rB * 0.3, rB);
    vec4 c = texture(samp, uv);
    c.r = texture(samp, uvR).r;
    c.g = texture(samp, uvG).g;
    c.b = texture(samp, uvB).b;
    return c;
}

vec4 lens(vec2 uv, vec2 center, float radius, float k) {
    vec2 d = (uv - center) / radius;
    float r = length(d);
    if (r > 1.0) return vec4(0.0);
    float r2 = r * r;
    float s = 1.0 - k * r2;
    vec2 suv = center + d * s * radius;
    float v = 1.0 - smoothstep(0.82, 1.0, r);
    vec4 c = sampleRipple(suv);
}

```

```

    c.rgb *= v;
    c.a = v;
    return c;
}
void main() {
    const vec2 centers[8] = vec2[](
        vec2(0.35, 0.45),
        vec2(0.65, 0.45),
        vec2(0.25, 0.35),
        vec2(0.50, 0.30),
        vec2(0.75, 0.35),
        vec2(0.25, 0.65),
        vec2(0.50, 0.70),
        vec2(0.75, 0.65)
    );
    const float radii[8] = float[](
        0.18, 0.18, 0.12, 0.12, 0.12, 0.12, 0.12, 0.12
    );
    const float k = 0.35;
    vec3 acc = vec3(0.0);
    float w = 0.0;
    for (int i = 0; i < 8; ++i) {
        vec4 c = lens(tc, centers[i], radii[i], k);
        acc += c.rgb;
        w += c.a;
    }
    if (w > 0.0) acc /= w;
    else acc = sampleRipple(tc).rgb;
    color = vec4(acc, 1.0);
}

```

Shader #349 - fold water

File: fold-water.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 getRainbowColor(float position) {
    float r = sin(position + 0.0) * 0.5 + 0.5;
    float g = sin(position + 2.0) * 0.5 + 0.5;
    float b = sin(position + 4.0) * 0.5 + 0.5;
    return vec3(r, g, b);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0) + 2.0;
    float wave = sin(tc.y * 10.0 + time_f) * 0.05;
    vec2 new_tc = vec2(tc.x + wave, tc.y);
    vec4 texColor = texture(samp, new_tc);
}

```

```

float spiralPosX = tc.x * cos(time_f) - tc.y * sin(time_f);
float spiralPosY = tc.x * sin(time_f) + tc.y * cos(time_f);
float rainbowPos = sqrt(spiralPosX * spiralPosX + spiralPosY * spiralPosY) * 10.0 + time_f * 5.0;
vec3 rainbowColor = getRainbowColor(sin(rainbowPos * time_t));
color = sin(vec4(mix(texColor.rgb, rainbowColor, 0.5), texColor.a) * time_t);
color.a = 1.0;
}

```

Shader #455 - g drum

File: g_drum.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 tc_centered = tc - center;
    float dist = length(tc_centered);
    vec2 dir = tc_centered / max(dist, 1e-6);
    float waveLength = 0.05;
    float amplitude = 0.02;
    float speed = 2.0;
    float ripple = sin((dist / waveLength - time_f * speed) * 6.2831853);
    vec2 tc_displaced = tc + dir * ripple * amplitude;
    vec4 texColor = texture(samp, tc_displaced);
    color = texColor;
}

```

Shader #474 - gem ripple

File: gem-ripple.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections.
Better random vector generation Smooth transition Dynamic Swirl that breathes

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec2 mirror(vec2 uv) {

```

```

    vec2 m = mod(uv, 2.0);
    return mix(m, 2.0 - m, step(1.0, m));
}
vec2 hash2(float n) {
    return fract(sin(vec2(n, n + 1.0)) * vec4(43758.5453, 12345.6789, 22578.1459, 98765.4321).xy);
}
void main(void) {
    float t = time_f * 0.5;
    float t_floor = floor(t);
    float t_fract = smoothstep(0.0, 1.0, fract(t));
    vec2 p0 = hash2(t_floor);
    vec2 p1 = hash2(t_floor + 1.0);
    vec2 center = mix(p0, p1, t_fract);
    vec2 uv = tc - center;
    float r = length(uv);
    float angle = atan(uv.y, uv.x);
    float swirl = sin(time_f * 0.4) * 3.0;
    angle += swirl * exp(-r * 2.0);
    vec2 warpedUV = center + vec2(cos(angle), sin(angle)) * r;
    warpedUV += 0.015 * vec2(sin(tc.y * 10.0 + time_f), cos(tc.x * 10.0 + time_f));
    float shift = 0.005 * r;
    float r_chan = texture(samp, mirror(warpedUV + shift)).r;
    float g_chan = texture(samp, mirror(warpedUV)).g;
    float b_chan = texture(samp, mirror(warpedUV - shift)).b;
    vec3 finalCol = vec3(r_chan, g_chan, b_chan);
    float vignette = smoothstep(1.2, 0.2, r);
    finalCol *= vignette;
    color = vec4(finalCol, 1.0);
}

```

Shader #482 - gem metal

File: gem_metal.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Precise spectral palette to match the pinks and cyans in your image This creates the repeating concentric circles seen in your image Map the rings to the spectrum

Technique: Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.3, 0.2, 0.2);
    return a + b * cos(6.28318 * (c * t + d));
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float r = length(uv);
    float angle = atan(uv.y, uv.x);
    float ripples = sin(angle * 12.0 + time_f) * 0.05;

```



```

float dist = r + ripples;
float ringPattern = sin(dist * 25.0 - time_f * 3.0);
vec2 distortedUV = tc + (uv / r) * ringPattern * 0.02;
vec4 tex = texture(samp, distortedUV);
vec3 ringColor = palette(dist * 2.0 - time_f * 0.5);
float centerGlow = exp(-r * 4.0) * 1.2;
vec3 finalRGB = mix(tex.rgb, ringColor, 0.4);
finalRGB += centerGlow * vec3(1.0, 0.95, 0.8);
finalRGB *= (0.8 + 0.2 * ringPattern);
color = vec4(finalRGB, 1.0);
}

```

Shader #484 - gem plas

File: gem_plas.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to audio input (amplitude/volume) for interactive control. Simple hash for noise Gradient noise for smooth fluid motion This creates the "liquid" flow effect

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
void main(void) {
    vec2 uv = tc;
    float distortX = noise(uv * 4.0 + time_f * 0.5);
    float distortY = noise(uv * 4.0 - time_f * 0.5);
    vec2 warpedUV = uv + vec2(distortX, distortY) * 0.05;
    vec4 texColor = texture(samp, warpedUV);
    vec3 neon = pow(texColor.rgb, vec3(1.5));
    neon *= vec3(1.2, 1.5, 2.0);
    float glow = smoothstep(0.4, 0.7, distortX) * 0.2;
    neon += glow * vec3(0.0, 0.8, 1.0);
    color = vec4(neon * (amp + 0.8), 1.0);
}

```

Shader #569 - halluc liquid

File: halluc_liquid.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. IQ's Cosine Palette - standard for psychedelic shaders Trippy Blue/Purple/Cyan offset Triangle wave wrap function

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;
const float iSaturation = 1.0;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d + iHueShift));
}
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
    return textureLod(tex, sampleUV, lod);
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}
```

```

float fbm(vec2 p, bool ridges) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        if (ridges) n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.05;
        a *= 0.5;
    }
    return v;
}

vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}

mat2 rot(float a) {
    float c = cos(a), s = sin(a);
    return mat2(c, -s, s, c);
}

vec3 renderEnergy(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p *= rot(iRotation + t * 0.1);
    float zoomLvl = max(iZoom, 0.1);
    p /= zoomLvl;
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + t * 0.2, false);
    q.y = fbm(p + vec2(5.2, 1.3) - t * 0.15, false);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0 * q + vec2(t * 0.5, 9.2), true);
    r.y = fbm(p + 4.0 * q + vec2(8.3, 2.8), false);
    vec2 warpedUV = p + strength * r;
    float slices = 6.0 + 10.0 * sin(t * 0.1) * iAmplitude;
    vec2 k = kaleido(warpedUV, max(3.0, slices));
    vec2 texUV = k / vec2(aspect, 1.0) + center;
    texUV += r * 0.1;
    vec3 texCol = mxTexture(samp, texUV).rgb;
    float warpLen = length(r);
    float electric = pow(warpLen, 3.0) * iContrast;
    vec3 pal = palette(length(q) + t * 0.4);
    vec3 finalCol = mix(texCol, pal, 0.5 * iSaturation);
    finalCol += pal * electric * strength;
    finalCol += vec3(smoothstep(0.8, 1.0, electric)) * 2.0;
    return finalCol;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float t = time_f * (0.2 + iFrequency * 0.5);
    float strength = iAmplitude * 1.5;
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec3 col;
    vec2 offset = vec2(0.005 * strength, 0.0);

```

```

    col.r = renderEnergy(uv + offset, t, strength, center, iResolution).r;
    col.g = renderEnergy(uv, t, strength, center, iResolution).g;
    col.b = renderEnergy(uv - offset, t, strength, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.25);
    col *= vig;
    color = vec4(col, 1.0);
}

```

Shader #588 - HyperVortex

File: HyperVortex.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Applies matrix transformations. Reduced iterations for performance. Normalize coordinates centered at 0,0. Twist increases closer to center, creating a vortex feel.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.14159265359;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 3; i++) {
        v += a * noise(p);
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}
vec3 electricPalette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
}

```

```

    vec3 d = vec3(0.00, 0.33, 0.67);
    return a + b * cos(6.28318 * (c * t + d));
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 center = vec2(0.5);
    vec2 p = (tc - center) * ar;
    float len = length(p);
    float twistStrength = 3.0 * sin(time_f * 0.3);
    float angle = atan(p.y, p.x) + twistStrength / (len + 0.1) + time_f * 0.5;
    vec2 twistedP = vec2(cos(angle), sin(angle)) * len;
    float rippleWavelength = 8.0;
    float rippleSpeed = 4.0;
    float noiseVal = fbm(twistedP * 5.0 + time_f);
    float rippleX = sin(twistedP.x * rippleWavelength + time_f * rippleSpeed + noiseVal * 2.0);
    float rippleY = cos(twistedP.y * rippleWavelength + time_f * rippleSpeed - noiseVal * 2.0);
    float rippleAmp = 0.05 + 0.05 * noiseVal;
    vec2 rippledP = twistedP + vec2(rippleX, rippleY) * rippleAmp;
    vec3 p3 = vec3(rippledP, 1.0);
    mat3 R = rotX(time_f * 0.2) * rotY(time_f * 0.15) * rotZ(time_f * 0.1);
    vec3 r = R * p3;
    float zScale = 1.0 / (1.0 + r.z * 0.5);
    vec2 projUV = r.xy * zScale;
    vec2 tiledUV = 1.0 - abs(1.0 - 2.0 * (projUV + 0.5));
    tiledUV = tiledUV - floor(tiledUV);
    float aberrationStr = 0.01 + 0.04 * (1.0 / (len + 0.5));
    vec2 offset = vec2(aberrationStr, 0.0);
    vec4 texR = texture(samp, clamp(tiledUV - offset, 0.0, 1.0));
    vec4 texG = texture(samp, clamp(tiledUV, 0.0, 1.0));
    vec4 texB = texture(samp, clamp(tiledUV + offset, 0.0, 1.0));
    vec3 finalColor = vec3(texR.r, texG.g, texB.b);
    float glowMask = smoothstep(0.6, 0.95, abs(rippleX * rippleY));
    vec3 palette = electricPalette(len - time_f);
    finalColor = mix(finalColor, finalColor + palette, glowMask * 0.8);
    color = vec4(finalColor, 1.0);
}

```

Shader #591 - inflate ripple

File: inflate-ripple.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float cycle_time = mod(time_f, 5.0);
    float inflate = min(cycle_time * 0.4, 1.5);
    vec2 center = vec2(0.5, 0.5);
    float distance = length(tc - center);
    float ripple = sin((distance - cycle_time) * 10.0) * 0.02;
    vec2 adjusted_tc = (tc - center) * inflate + center + normalize(tc - center) * ripple;
    color = texture(samp, adjusted_tc);
}

```

```
}
```

Shader #610 - Liquid Censorship

File: Liquid_Censorship.glsl

Applies geometric transformations including rotation and scaling to distort the image. This prevents the spiral from looking like an oval Shifts the y-lookup based on x-position and time atan(y, x) gives the angle. We add time to rotate it.

Technique: 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
const float PI = 3.14159;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    if (iResolution.x != 0.0) {
        uv.y *= iResolution.y / iResolution.x;
    }
    float wave = sin(uv.x * 10.0 + time_f * 2.0) * 0.1;
    float angle = atan(uv.y + wave, uv.x) + time_f * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * PI));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    float pulse = 0.75 + 0.25 * sin(time_f * 2.0);
    color = vec4(blended_color * pulse, original_color.a);
}
```

Shader #611 - Liquid Crystal

File: Liquid_Crystal.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. IQ Cosine Palette (from Shader 2) Combined FBM: Can do smooth liquid or sharp ridges Ridges = True for electric feel

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
```

```

uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;
const float iSaturation = 1.0;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 rotateHue(vec3 col, float angle) {
    float U = cos(angle); float W = sin(angle);
    mat3 R = mat3(
        0.299+0.701*U+0.168*W, 0.587-0.587*U+0.330*W, 0.114-0.114*U-0.497*W,
        0.299-0.299*U-0.328*W, 0.587+0.413*U+0.035*W, 0.114-0.114*U+0.292*W,
        0.299-0.300*U+1.250*W, 0.587-0.588*U-1.050*W, 0.114+0.886*U-0.203*W
    );
    return clamp(R * col, 0.0, 1.0);
}
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d + iHueShift));
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
    return textureLod(tex, sampleUV, lod);
}
vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}
float fbm(vec2 p, bool ridges) {

```

```

float v = 0.0;
float a = 0.5;
for (int i = 0; i < 4; i++) {
    float n = noise(p);
    if (ridges) n = 1.0 - abs(n * 2.0 - 1.0);
    v += a * n;
    p *= 2.05;
    a *= 0.5;
}
return v;
}

vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}

vec3 sampleHybrid(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.01);
    p /= zoom;
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + t * 0.1, false);
    q.y = fbm(p + vec2(5.2, 1.3) - t * 0.1, false);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0 * q + vec2(t * 0.2, 9.2), true);
    r.y = fbm(p + 4.0 * q + vec2(8.3, 2.8), false);
    p += r * (0.1 * strength);
    float slices = 6.0 + floor(iAmplitude * 4.0);
    p = kaleido(p, slices);
    int iterations = 4 + int(iAmplitude * 1.5);
    float scale = 1.1 + (iFrequency * 0.3);
    float shift = 0.1 * strength;
    float angle = t * 0.2;
    for(int i = 0; i < iterations; i++) {
        p = abs(p);
        p -= shift;
        p *= scale;
        p = rotate2D(p, angle + float(i)*0.5 + (r.x * 0.2));
    }
    vec2 finalUV = p * 0.5 + center;
    vec3 texCol = mxTexture(samp, finalUV).rgb;
    float electric = pow(length(r), 3.0) * iContrast;
    vec3 pal = palette(length(p) + length(q) + t);
    vec3 finalCol = mix(texCol, pal, 0.4 * iSaturation);
    finalCol += pal * electric * strength;
    return finalCol;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float t = time_f * (0.2 + iFrequency * 0.2);
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float strength = 0.5 + (ampControl * 1.0);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
}

```



```

    }
    vec2 offset = vec2(0.003 * strength, 0.0);
    vec3 col;
    col.r = sampleHybrid(uv + offset, t, strength, center, iResolution).r;
    col.g = sampleHybrid(uv, t, strength, center, iResolution).g;
    col.b = sampleHybrid(uv - offset, t, strength, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.15);
    col *= vig;
    color = vec4(col, 1.0);
}

```

Shader #612 - Liquid Crystal 2

File: Liquid_Crystal_2.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. IQ Cosine Palette Calculates the distortion and returns the modified point + the intensity of distortion. Frequency increases with distance.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.2;
const float iSaturation = 1.2;
const float iHueShift = 0.0;
const float iZoom = 0.8;
const float iRotation = 0.0;
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d + iHueShift));
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}

```

```

}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    return textureLod(tex, sampleUV, 0.0);
}
vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        v += a * noise(p);
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}
vec3 rippleBendExpand(vec2 p, float t, float strength) {
    float d = length(p);
    float expansion = 1.0 + (sin(t * 2.0) * 0.1 * strength);
    p /= expansion;
    float wavePhase = d * (10.0 * iFrequency) - (t * 5.0);
    float ripple = sin(wavePhase);
    p += (p / (d+0.001)) * ripple * 0.05 * strength;
    float angle = atan(p.y, p.x);
    float twist = sin(d * 4.0 - t) * 0.5 * strength;
    angle += twist;
    p = vec2(cos(angle), sin(angle)) * length(p);
    return vec3(p, abs(ripple) + abs(twist));
}
vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}
vec3 sampleHybrid(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.01);
    p /= zoom;
    vec3 distort = rippleBendExpand(p, t, strength);
    p = distort.xy;
}

```

```

float extractMask = distort.z;
vec2 q = vec2(fbm(p + t * 0.1), fbm(p + vec2(5.2, 1.3)));
vec2 r = vec2(fbm(p + 4.0 * q + t * 0.2), fbm(p + q));
p += r * (0.15 * strength);
float slices = 8.0;
p = kaleido(p, slices);
int iterations = 4;
float scale = 1.3;
float shift = 0.2 * strength;
for(int i = 0; i < iterations; i++) {
    p = abs(p);
    p -= shift;
    p *= scale;
    p = rotate2D(p, t * 0.1 + float(i));
}
vec2 finalUV = p * 0.5 + center;
vec3 texCol = mxTexture(samp, finalUV).rgb;
vec3 energyCol = palette(length(p) * 0.2 + t * 0.5 + extractMask);
vec3 finalCol = mix(texCol, energyCol, smoothstep(0.2, 1.5, extractMask));
finalCol += energyCol * extractMask * 0.5 * strength;
return finalCol;
}

void main() {
    vec2 uv = tc;
    float t = time_f * (0.2 + iFrequency * 0.2);
    float ampControl = clamp(iAmplitude, 0.0, 3.0);
    float strength = 0.8 + (ampControl * 0.5);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec2 offset = vec2(0.005 * strength, 0.0);
    vec3 col;
    col.r = sampleHybrid(uv + offset, t, strength, center, iResolution).r;
    col.g = sampleHybrid(uv, t, strength, center, iResolution).g;
    col.b = sampleHybrid(uv - offset, t, strength, center, iResolution).b;
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    col = adjustBrightness(col, iBrightness);
    float vig = 1.0 - length(uv - 0.5);
    col *= smoothstep(0.0, 1.2, vig);
    color = vec4(col, 1.0);
}

```

Shader #613 - Liquid Crystal Rainbow1

File: Liquid_Crystal_Rainbow1.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. slightly lowered default so texture pops A. Sample the texture using the warped 'p' coordinate This applies the liquid shape to the texture image

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
const float PI = 3.14159265359;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
const float iAmplitude = 0.5;
const float iFrequency = 1.5;
const float iBrightness = 0.8;
const float iContrast = 1.1;
const float iSaturation = 0.5;
const float iZoom = 1.0;
float iRotation = pingPong(time_f, 10.0);
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    return texture(tex, sampleUV);
}
vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}
float fbm(vec2 p, bool ridges) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        if (ridges) n = 1.0 - abs(n * 2.0 - 1.0);
    }
}

```

```

        v += a * n;
        p *= 2.05;
        a *= 0.5;
    }
    return v;
}

vec2 kaleido(vec2 p, float slices) {
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = PI * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}

vec3 sampleLiquidRainbow(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.01);
    p /= zoom;
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + t * 0.1, false);
    q.y = fbm(p + vec2(5.2, 1.3) - t * 0.1, false);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0 * q + vec2(t * 0.2, 9.2), true);
    r.y = fbm(p + 4.0 * q + vec2(8.3, 2.8), false);
    p += r * (0.15 * strength);
    float slices = 6.0 + floor(iAmplitude * 4.0);
    p = kaleido(p, slices);
    int iterations = 4 + int(iAmplitude * 1.5);
    float scale = 1.1 + (iFrequency * 0.3);
    float shift = 0.1 * strength;
    float angle = t * 0.2;
    for(int i = 0; i < iterations; i++) {
        p = abs(p);
        p -= shift;
        p *= scale;
        p = rotate2D(p, angle + float(i)*0.5 + (r.x * 0.2));
    }
    float spiralAngle = atan(p.y, p.x);
    float colorIndex = (spiralAngle / (2.0 * PI)) + length(p) * 0.5 + t * 0.5;
    vec3 rainbowCol = rainbow(colorIndex);
    vec2 finalUV = p * 0.5 + center;
    vec3 texCol = mxTexture(samp, finalUV).rgb;
    float electric = pow(length(r), 3.0) * iContrast;
    float texLum = dot(texCol, vec3(0.299, 0.587, 0.114));
    vec3 colorizedTex = rainbowCol * texLum * 2.5;
    vec3 finalCol = mix(texCol, colorizedTex, iSaturation);
    finalCol += rainbowCol * electric * strength;
    return finalCol;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float t = time_f * (0.2 + iFrequency * 0.1);
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float strength = 0.5 + (ampControl * 1.0);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec2 offset = vec2(0.003 * strength, 0.0);

```

```

    vec3 col;
    col.r = sampleLiquidRainbow(uv + offset, t, strength, center, iResolution).r;
    col.g = sampleLiquidRainbow(uv, t, strength, center, iResolution).g;
    col.b = sampleLiquidRainbow(uv - offset, t, strength, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.15);
    col *= vig;
    float pulse = 0.85 + 0.15 * sin(time_f * 2.0);
    color = vec4(col * pulse, 1.0);
}

```

Shader #615 - Liquid Heat

File: Liquid_Heat.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Controls Liquid Intensity Controls Fractal/Kaleidoscope Complexity Note: Hue shift handled in post-process now

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.1;
const float iSaturation = 1.3;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 rotateHue(vec3 col, float angle) {
    float U = cos(angle); float W = sin(angle);
    mat3 R = mat3(
        0.299+0.701*U+0.168*W, 0.587-0.587*U+0.330*W, 0.114-0.114*U-0.497*W,
        0.299-0.299*U-0.328*W, 0.587+0.413*U+0.035*W, 0.114-0.114*U+0.292*W,
        0.299-0.300*U+1.250*W, 0.587-0.588*U-1.050*W, 0.114+0.886*U-0.203*W
    );
}

```

```

    return clamp(R * col, 0.0, 1.0);
}
vec3 getNeonPalette(float t) {
    float x = fract(t);
    vec3 colBlue   = vec3(0.05, 0.2, 1.0);
    vec3 colPurple = vec3(0.7, 0.0, 1.0);
    vec3 colPink   = vec3(1.0, 0.05, 0.5);
    if (x < 0.33) return mix(colBlue, colPurple, x * 3.0);
    else if (x < 0.66) return mix(colPurple, colPink, (x - 0.33) * 3.0);
    else return mix(colPink, colBlue, (x - 0.66) * 3.0);
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
    vec2 deriv = fwidth(tc);
    if (deriv.x > 0.0 || deriv.y > 0.0) {
        lod = log2(max(max(deriv.x, deriv.y) * max(ts.x, ts.y), 1.0));
    }
    return textureLod(tex, sampleUV, lod);
}
vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}
vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i), hash(i + vec2(1.0, 0.0)), u.x),
               mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    mat2 rot = mat2(cos(0.5), sin(0.5), -sin(0.5), cos(0.5));
    for (int i = 0; i < 4; i++) {
        v += a * noise(p);
        p = rot * p * 2.0 + vec2(100.0);
        a *= 0.5;
    }
    return v;
}
vec3 sampleSuperShader(vec2 uv, float t, vec2 center, vec2 res) {
    float aspect = res.x / res.y;

```

```

    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.01);
    p /= zoom;
    float time_t = pingPong(time_f, 10.0) + 1.0;
    float time_s = pingPong(time_f, 5.0);
    float waveFreq = 2.0 + iFrequency * 2.0;
    float waveAmp = 0.1 * sin(0.2 * time_t) * iAmplitude;
    float waveSpd = sin(0.3 * time_s);
    p.y += sin(p.x * waveFreq + time_f * waveSpd) * waveAmp;
    if (iFrequency > 0.2) {
        float slices = 4.0 + floor(iFrequency * 6.0);
        p = kaleido(p, slices);
        int iterations = 2 + int(iFrequency * 2.0);
        float scale = 1.1;
        float shift = 0.1;
        for(int i = 0; i < iterations; i++) {
            p = abs(p);
            p -= shift;
            p *= scale;
            p = rotate2D(p, t * 0.05 + float(i));
        }
    }
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + 0.05*t);
    q.y = fbm(p + vec2(5.2, 1.3) + 0.05*t);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0*q + vec2(t * 0.1));
    r.y = fbm(p + 4.0*q + vec2(t * 0.05, 2.8));
    float liquidStrength = 0.2 + (iAmplitude * 0.5);
    vec2 distortion = r * liquidStrength;
    vec2 finalUV = uv + distortion;
    finalUV.y += sin(finalUV.x * waveFreq + time_f * waveSpd) * waveAmp;
    vec3 texCol = mxTexture(samp, finalUV).rgb;
    float pattern = length(q) + length(r) + t * 0.1;
    vec3 neon = getNeonPalette(pattern);
    vec3 col = texCol * neon * 2.0;
    float turbulence = length(r);
    col = mix(col, texCol, max(0.0, 1.0 - turbulence * 2.0));
    return col;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float t = time_f * 0.2;
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    float caStrength = 0.003 * (1.0 + iAmplitude);
    vec3 col;
    col.r = sampleSuperShader(uv + vec2(caStrength, 0.0), t, center, iResolution).r;
    col.g = sampleSuperShader(uv, t, center, iResolution).g;
    col.b = sampleSuperShader(uv - vec2(caStrength, 0.0), t, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    col = rotateHue(col, iHueShift);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.2);
    col *= vig;
}

```



```

    color = vec4(col, 1.0);
}

```

Shader #616 - Liquid Heat blend

File: Liquid_Heat_blend.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Boost to see more rainbow, lower to see more original texture Only activates if iFrequency is > 0.2 Apply wave logic to lookup to keep it synced

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;
const float iSaturation = 1.0;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 rotateHue(vec3 col, float angle) {
    float U = cos(angle); float W = sin(angle);
    mat3 R = mat3(
        0.299+0.701*U+0.168*W, 0.587-0.587*U+0.330*W, 0.114-0.114*U-0.497*W,
        0.299-0.299*U-0.328*W, 0.587+0.413*U+0.035*W, 0.114-0.114*U+0.292*W,
        0.299-0.300*U+1.250*W, 0.587-0.588*U-1.050*W, 0.114+0.886*U-0.203*W
    );
    return clamp(R * col, 0.0, 1.0);
}
vec3 getNeonPalette(float t) {
    float x = fract(t);
    vec3 colBlue = vec3(0.05, 0.2, 1.0);
    vec3 colPurple = vec3(0.7, 0.0, 1.0);
    vec3 colPink = vec3(1.0, 0.05, 0.5);
    if (x < 0.33) return mix(colBlue, colPurple, x * 3.0);
    else if (x < 0.66) return mix(colPurple, colPink, (x - 0.33) * 3.0);
    else return mix(colPink, colBlue, (x - 0.66) * 3.0);
}

```

```

vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}

vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
    vec2 deriv = fwidth(tc);
    if (deriv.x > 0.0 || deriv.y > 0.0) {
        lod = log2(max(max(deriv.x, deriv.y) * max(ts.x, ts.y), 1.0));
    }
    return textureLod(tex, sampleUV, lod);
}

vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}

vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}

float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}

float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}

float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    mat2 rot = mat2(cos(0.5), sin(0.5), -sin(0.5), cos(0.5));
    for (int i = 0; i < 4; i++) {
        v += a * noise(p);
        p = rot * p * 2.0 + vec2(100.0);
        a *= 0.5;
    }
    return v;
}

vec3 sampleSuperShader(vec2 uv, float t, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.01);
    p /= zoom;
    float time_t = pingPong(time_f, 10.0) + 1.0;
    float time_s = pingPong(time_f, 5.0);
    float waveFreq = 2.0 + iFrequency * 2.0;
    float waveAmp = 0.1 * sin(0.2 * time_t) * iAmplitude;
    float waveSpd = sin(0.3 * time_s);
    p.y += sin(p.x * waveFreq + time_f * waveSpd) * waveAmp;
    if (iFrequency > 0.2) {

```

```

        float slices = 4.0 + floor(iFrequency * 6.0);
        p = kaleido(p, slices);
        int iterations = 2 + int(iFrequency * 2.0);
        float scale = 1.1;
        float shift = 0.1;
        for(int i = 0; i < iterations; i++) {
            p = abs(p);
            p -= shift;
            p *= scale;
            p = rotate2D(p, t * 0.05 + float(i));
        }
    }
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + 0.05*t);
    q.y = fbm(p + vec2(5.2, 1.3) + 0.05*t);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0*q + vec2(t * 0.1));
    r.y = fbm(p + 4.0*q + vec2(t * 0.05, 2.8));
    float liquidStrength = 0.2 + (iAmplitude * 0.5);
    vec2 distortion = r * liquidStrength;
    vec2 finalUV = uv + distortion;
    finalUV.y += sin(finalUV.x * waveFreq + time_f * waveSpd) * waveAmp;
    vec3 texCol = mxTexture(samp, finalUV).rgb;
    float pattern = length(q) + length(r) + t * 0.1;
    vec3 neon = getNeonPalette(pattern);
    vec3 col = texCol;
    float turbulence = length(r);
    vec3 rainbowOverlay = texCol * neon * 1.5;
    col = mix(col, rainbowOverlay, turbulence * iSaturation * 0.8);
    col += neon * pow(turbulence, 3.0) * 0.5 * iAmplitude;
    return col;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float t = time_f * 0.2;
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    float caStrength = 0.003 * (1.0 + iAmplitude);
    vec3 col;
    col.r = sampleSuperShader(uv + vec2(caStrength, 0.0), t, center, iResolution).r;
    col.g = sampleSuperShader(uv, t, center, iResolution).g;
    col.b = sampleSuperShader(uv - vec2(caStrength, 0.0), t, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    col = rotateHue(col, iHueShift);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.2);
    col *= vig;
    color = vec4(col, 1.0);
}

```

Shader #617 - Liquid Light Rainbow Blend

File: Liquid_Light_Rainbow_Blend.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Cycles strictly Pink -> Purple -> Blue Deep Neon Purple Calculate final distortion amount

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;

const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.2;
const float iContrast = 1.1;
const float iSaturation = 1.4;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;

vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}

vec3 getNeonPalette(float t) {
    float x = fract(t + iHueShift);
    vec3 colBlue = vec3(0.05, 0.2, 1.0);
    vec3 colPurple = vec3(0.7, 0.0, 1.0);
    vec3 colPink = vec3(1.0, 0.05, 0.5);
    if (x < 0.33) {
        return mix(colBlue, colPurple, x * 3.0);
    } else if (x < 0.66) {
        return mix(colPurple, colPink, (x - 0.33) * 3.0);
    } else {
        return mix(colPink, colBlue, (x - 0.66) * 3.0);
    }
}

vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}

vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    return texture(tex, sampleUV);
}

vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}

vec2 sineWave(vec2 p, float t) {
    float waveAmp = 0.05 * iAmplitude;
    float waveFreq = 3.0 * iFrequency;
    p.y += sin(p.x * waveFreq + t) * waveAmp;
    p.x += cos(p.y * waveFreq * 0.5 + t) * (waveAmp * 0.5);
    return p;
}
```

```

}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i), hash(i + vec2(1.0, 0.0)), u.x),
               mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    mat2 rot = mat2(cos(0.5), sin(0.5), -sin(0.5), cos(0.5));
    for (int i = 0; i < 5; i++) {
        v += a * noise(p);
        p = rot * p * 2.0 + vec2(2.0);
        a *= 0.5;
    }
    return v;
}
vec3 sampleMergedLiquid(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.1);
    p /= zoom;
    p = sineWave(p, t * 1.5);
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + 0.05*t);
    q.y = fbm(p + vec2(5.2, 1.3) + 0.05*t);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0*q + vec2(t * 0.2));
    r.y = fbm(p + 4.0*q + vec2(t * 0.1, 2.8));
    vec2 distortion = r * 0.3 * strength * iAmplitude;
    vec2 fluidUV = uv + distortion;
    fluidUV = sineWave(fluidUV, t * 1.5);
    vec3 texCol = mxTexture(samp, fluidUV).rgb;
    float pattern = length(q) + length(r) + t * 0.1;
    vec3 neon = getNeonPalette(pattern);
    float lum = dot(texCol, vec3(0.299, 0.587, 0.114));
    vec3 finalCol = texCol * neon * 2.5;
    finalCol = mix(finalCol, texCol * vec3(0.5, 0.3, 0.8), 0.2);
    return finalCol;
}
void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float t = time_f * (0.2 + iFrequency * 0.1);
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float strength = 1.0 + (ampControl * 0.5);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec2 offset = vec2(0.004 * strength, 0.0);
    vec3 col;
    col.r = sampleMergedLiquid(uv + offset, t, strength, center, iResolution).r;
    col.g = sampleMergedLiquid(uv, t, strength, center, iResolution).g;
    col.b = sampleMergedLiquid(uv - offset, t, strength, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
}

```

```

    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.2);
    col *= vig;
    col = mix(vec3(0.05, 0.0, 0.1), col, vig);
    color = vec4(col, 1.0);
}

```

Shader #618 - Liquid Light Time

File: Liquid_Light_Time.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Replaced 'time_f' with 't' (accelerated time) Sync wave logic to the lookup Blend logic: Only add neon where the liquid is active

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Bitwise XOR on integer color channels - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.0;
const float iSaturation = 1.0;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 rotateHue(vec3 col, float angle) {
    float U = cos(angle); float W = sin(angle);
    mat3 R = mat3(
        0.299+0.701*U+0.168*W, 0.587-0.587*U+0.330*W, 0.114-0.114*U-0.497*W,
        0.299-0.299*U-0.328*W, 0.587+0.413*U+0.035*W, 0.114-0.114*U+0.292*W,
        0.299-0.300*U+1.250*W, 0.587-0.588*U-1.050*W, 0.114+0.886*U-0.203*W
    );
    return clamp(R * col, 0.0, 1.0);
}
vec3 getNeonPalette(float t) {
    float x = fract(t);

```

```

    vec3 colBlue   = vec3(0.05, 0.2, 1.0);
    vec3 colPurple = vec3(0.7, 0.0, 1.0);
    vec3 colPink   = vec3(1.0, 0.05, 0.5);
    if (x < 0.33) return mix(colBlue, colPurple, x * 3.0);
    else if (x < 0.66) return mix(colPurple, colPink, (x - 0.33) * 3.0);
    else return mix(colPink, colBlue, (x - 0.66) * 3.0);
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    float lod = 0.0;
    vec2 deriv = fwidth(tc);
    if (deriv.x > 0.0 || deriv.y > 0.0) {
        lod = log2(max(max(deriv.x, deriv.y) * max(ts.x, ts.y), 1.0));
    }
    return textureLod(tex, sampleUV, lod);
}
vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}
vec2 kaleido(vec2 p, float slices) {
    float pi = 3.14159265359;
    float r = length(p);
    float a = atan(p.y, p.x);
    float sector = pi * 2.0 / slices;
    a = mod(a, sector);
    a = abs(a - sector * 0.5);
    return vec2(cos(a), sin(a)) * r;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i), hash(i + vec2(1.0, 0.0)), u.x),
               mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    mat2 rot = mat2(cos(0.5), sin(0.5), -sin(0.5), cos(0.5));
    for (int i = 0; i < 4; i++) {
        v += a * noise(p);
        p = rot * p * 2.0 + vec2(100.0);
        a *= 0.5;
    }
    return v;
}
vec3 sampleSuperShader(vec2 uv, float t, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.01);
    p /= zoom;

```

```

float time_t = pingPong(t, 10.0) + 1.0;
float time_s = pingPong(t, 5.0);
float waveFreq = 2.0 + iFrequency * 2.0;
float waveAmp = 0.1 * sin(0.2 * time_t) * iAmplitude;
float waveSpd = sin(0.3 * time_s);
p.y += sin(p.x * waveFreq + t * waveSpd) * waveAmp;
if (iFrequency > 0.2) {
    float slices = 4.0 + floor(iFrequency * 6.0);
    p = kaleido(p, slices);
    int iterations = 2 + int(iFrequency * 2.0);
    float scale = 1.1;
    float shift = 0.1;
    for(int i = 0; i < iterations; i++) {
        p = abs(p);
        p -= shift;
        p *= scale;
        p = rotate2D(p, t * 0.05 + float(i));
    }
}
vec2 q = vec2(0.0);
q.x = fbm(p + vec2(0.0, 0.0) + 0.05*t);
q.y = fbm(p + vec2(5.2, 1.3) + 0.05*t);
vec2 r = vec2(0.0);
r.x = fbm(p + 4.0*q + vec2(t * 0.1));
r.y = fbm(p + 4.0*q + vec2(t * 0.05, 2.8));
float liquidStrength = 0.2 + (iAmplitude * 0.5);
vec2 distortion = r * liquidStrength;
vec2 finalUV = uv + distortion;
finalUV.y += sin(finalUV.x * waveFreq + t * waveSpd) * waveAmp;
vec3 texCol = mxTexture(samp, finalUV).rgb;
float pattern = length(q) + length(r) + t * 0.1;
vec3 neon = getNeonPalette(pattern);
vec3 col = texCol;
float turbulence = length(r);
vec3 rainbowOverlay = texCol * neon * 1.5;
col = mix(col, rainbowOverlay, turbulence * iSaturation * 0.8);
col += neon * pow(turbulence, 3.0) * 0.5 * iAmplitude;
return col;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float accelTime = (time_f * 0.2) + (time_f * time_f * 0.05);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    float caStrength = 0.003 * (1.0 + iAmplitude);
    vec3 col;
    col.r = sampleSuperShader(uv + vec2(caStrength, 0.0), accelTime, center, iResolution).r;
    col.g = sampleSuperShader(uv, accelTime, center, iResolution).g;
    col.b = sampleSuperShader(uv - vec2(caStrength, 0.0), accelTime, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    col = rotateHue(col, iHueShift);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.2);
    col *= vig;
    color = vec4(col, 1.0);
}

```

Shader #619 - Liquid Rainbow

File: Liquid_Rainbow.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Slightly boosted for light effect IQ Cosine Palette - Tweaked for Full Rainbow This offset vector creates the rainbow spectrum

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.2;
const float iContrast = 1.1;
const float iSaturation = 1.2;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.00, 0.33, 0.67);
    return a + b * cos(6.28318 * (c * t + d + iHueShift));
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    return texture(tex, sampleUV);
}
vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
```

```

    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}

float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    mat2 rot = mat2(cos(0.5), sin(0.5), -sin(0.5), cos(0.50));
    for (int i = 0; i < 5; i++) {
        v += a * noise(p);
        p = rot * p * 2.0 + vec2(2.0);
        a *= 0.5;
    }
    return v;
}

vec3 sampleLiquid(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.1);
    p /= zoom;
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + 0.05*t);
    q.y = fbm(p + vec2(5.2, 1.3) + 0.05*t);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0*q + vec2(t * 0.15));
    r.y = fbm(p + 4.0*q + vec2(t * 0.05, 2.8));
    float f = fbm(p + 4.0*r);
    vec2 fluidUV = uv + (r * 0.2 * strength * iAmplitude);
    vec3 texCol = mxTexture(samp, fluidUV).rgb;
    vec3 rainbow = palette(length(q) + f + t * 0.2);
    vec3 col = mix(texCol, texCol * rainbow * 1.5, 0.6 * iSaturation);
    float shine = f * f * f * 1.5 * iContrast;
    col += shine * rainbow * strength;
    col *= 0.8 + 0.5 * f;
    return col;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float t = time_f * (0.1 + iFrequency * 0.1);
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float strength = 1.0 + (ampControl * 0.5);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec2 offset = vec2(0.005 * strength, 0.0);
    vec3 col;
    col.r = sampleLiquid(uv + offset, t, strength, center, iResolution).r;
    col.g = sampleLiquid(uv, t, strength, center, iResolution).g;
    col.b = sampleLiquid(uv - offset, t, strength, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.15);

```

```

    col *= vig;
    color = vec4(col, 1.0);
}

```

Shader #620 - Liquid Rainbow Wave

File: Liquid_Rainbow_Wave.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Higher contrast for neon look Boost saturation for deep colors Phase shift t to loop

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
out vec4 color;
in vec2 tc;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.2;
const float iSaturation = 1.3;
const float iHueShift = 0.0;
const float iZoom = 1.0;
const float iRotation = 0.0;
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec3 getNeonPalette(float t) {
    float x = fract(t + iHueShift);
    vec3 colBlue = vec3(0.05, 0.1, 1.0);
    vec3 colPurple = vec3(0.6, 0.0, 0.9);
    vec3 colPink = vec3(1.0, 0.05, 0.6);
    if (x < 0.33) {
        return mix(colBlue, colPurple, x * 3.0);
    } else if (x < 0.66) {
        return mix(colPurple, colPink, (x - 0.33) * 3.0);
    } else {
        return mix(colPink, colBlue, (x - 0.66) * 3.0);
    }
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    return texture(tex, sampleUV);
}

```

```

vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}

vec2 sineWave(vec2 p, float t) {
    float waveAmp = 0.1 * iAmplitude;
    float waveFreq = 4.0 * iFrequency;
    p.y += sin(p.x * waveFreq + t) * waveAmp;
    p.x += cos(p.y * waveFreq * 0.5 + t) * (waveAmp * 0.5);
    return p;
}

float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}

float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    float a = hash(i + vec2(0.0, 0.0));
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    return mix(mix(a, b, u.x), mix(c, d, u.x), u.y);
}

float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    mat2 rot = mat2(cos(0.5), sin(0.5), -sin(0.5), cos(0.5));
    for (int i = 0; i < 5; i++) {
        v += a * noise(p);
        p = rot * p * 2.0 + vec2(2.0);
        a *= 0.5;
    }
    return v;
}

vec3 sampleLiquid(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.1);
    p /= zoom;
    p = sineWave(p, t * 2.0);
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + 0.05*t);
    q.y = fbm(p + vec2(5.2, 1.3) + 0.05*t);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0*q + vec2(t * 0.15));
    r.y = fbm(p + 4.0*q + vec2(t * 0.05, 2.8));
    float f = fbm(p + 4.0*r);
    vec2 fluidUV = uv + (r * 0.15 * strength * iAmplitude);
    fluidUV = sineWave(fluidUV, t * 2.0);
    vec3 texCol = mxTexture(samp, fluidUV).rgb;
    vec3 neon = getNeonPalette(length(q) + f + t * 0.1);
    vec3 base = texCol * 0.5;
    vec3 col = mix(base, neon, f * iSaturation);
    float ridges = 1.0 - abs(f * 2.0 - 1.0);
    ridges = pow(ridges, 3.0);
    col += neon * ridges * strength * 0.8;
    col = mix(col, col * vec3(0.8, 0.5, 1.0), 0.3);
    return col;
}

void main() {

```

```

    vec2 uv = tc;
    uv = wrapUV(uv);
    float t = (time_f * 3.0) * (0.2 + iFrequency * 0.1);
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float strength = 1.0 + (ampControl * 0.5);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec2 offset = vec2(0.005 * strength, 0.0);
    vec3 col;
    col.r = sampleLiquid(uv + offset, t, strength, center, iResolution).r;
    col.g = sampleLiquid(uv, t, strength, center, iResolution).g;
    col.b = sampleLiquid(uv - offset, t, strength, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.15);
    col *= vig;
    col = mix(vec3(0.1, 0.0, 0.2), col, vig);
    color = vec4(col, 1.0);
}

```

Shader #734 - ocean

File: ocean.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = tc;
    float waveAmplitude = 0.08;
    float waveFrequency = 8.0;
    float waveSpeed = 2.0;
    float phase = sin(time_f * 0.5) * 3.14;
    float yWave = sin((normCoord.x + phase) * waveFrequency - (time_f * waveSpeed)) * waveAmplitude;
    vec2 wavedCoord = vec2(normCoord.x, normCoord.y + yWave);
    color = texture(samp, wavedCoord);
}

```

Shader #735 - ocean fast

File: ocean_fast.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = tc;
    float waveAmplitude = 0.15;
    float waveFrequency = 15.0;
    float waveSpeed = 8.0;
    float phase = sin(time_f * 0.5) * 3.14;
    float yWave = sin((normCoord.x + phase) * waveFrequency - (time_f * waveSpeed)) * waveAmplitude;
    vec2 wavedCoord = vec2(normCoord.x, normCoord.y + yWave);
    color = texture(samp, wavedCoord);
}

```

Shader #737 - old film water

File: old-film-water.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
 Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void) {
    vec2 normCoord = tc;
    float timeAdjustedFrequency = 10.0 + sin(time_f) * 5.0;
    float timeAdjustedAmplitude = 0.01 + (sin(time_f * 0.5) * 0.5 + 0.5) * 0.05;
    float zigzagFactor = abs(fract(normCoord.y * timeAdjustedFrequency) - 0.5) * 2.0;
    float xDisplacement = zigzagFactor * timeAdjustedAmplitude;
    vec2 displacedCoord = vec2(normCoord.x + xDisplacement, normCoord.y);
    displacedCoord = clamp(displacedCoord, 0.0, 1.0);
    vec4 texColor = texture(samp, displacedCoord);
    float noise = rand(displacedCoord * time_f) * 0.1;
    float line = step(0.995, rand(vec2(displacedCoord.y * 100.0, time_f * 0.1)));
    texColor.rgb += noise;
    texColor.rgb = mix(texColor.rgb, vec3(0.0), line * 0.5);
    color = texColor;
}

```

Shader #788 - psych ripple

File: psych_ripple.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {
    ivec4 isource = ivec4(isourcex * 255);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)), dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 normPos = (gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0;
    float dist = length(normPos);
    float phase = sin(dist * 10.0 - time_f * 4.0);
    vec2 tcAdjusted = tc + (normPos * 0.305 * phase);
    vec4 texColor = texture(samp, tcAdjusted);
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float time = time_f * 0.5;
    float sine1 = sin(uv.x * 10.0 + time) * cos(uv.y * 10.0 + time);
    float sine2 = sin(uv.y * 15.0 - time) * cos(uv.x * 15.0 - time);
    float sine3 = sin(sqrt(uv.x * uv.x + uv.y * uv.y) * 20.0 + time);
    float pattern = sine1 + sine2 + sine3;
    float colorIntensity = pattern * 0.5 + 0.5;
    vec3 psychedelicColor = vec3(
        sin(colorIntensity * 6.28318 + 0.0) * 0.5 + 0.5,
        sin(colorIntensity * 6.28318 + 2.09439) * 0.5 + 0.5,
        sin(colorIntensity * 6.28318 + 4.18879) * 0.5 + 0.5
    );
    color = vec4(psychedelicColor, 1.0);
}
```

```

);
vec4 finalColor = vec4(psychedelicColor, 1.0);
vec4 xorColor = xor_RGB(finalColor, texColor);
vec2 uv2 = tc * 2.0 - 1.0;
uv2.y *= iResolution.y / iResolution.x;
float wave = sin(uv2.x * 10.0 + time_f * 2.0) * 0.1;
vec2 random_direction = smoothRandom2(time_f) * 0.5;
float expand = 0.5 + 0.5 * sin(time_f * 2.0);
vec2 spiral_uv = uv2 * expand + random_direction;
float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_f * 2.0;
vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
vec4 original_color = texture(samp, tc);
vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
float time_t = mod(time_f, 30.0);
vec4 finalBlendedColor = vec4(sin(blended_color * time_t), original_color.a);
color = mix(xorColor, finalBlendedColor, 0.5);
}

```

Shader #791 - psyche ripple

File: psyche_ripple.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normPos = (gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0;
    float dist = length(normPos);
    float phase = sin(dist * 10.0 - time_f * 4.0);
    vec2 tcAdjusted = tc + (normPos * 0.305 * phase);
    vec4 color1 = texture(samp, tcAdjusted);
    vec2 centeredCoord = (tc * 2.0) - vec2(1.0, 1.0);
    float stretchFactor = 1.0 + (1.0 - abs(centeredCoord.y)) * 0.5;
    centeredCoord.x *= sin(stretchFactor * time_f);
    vec2 stretchedCoord = (centeredCoord + vec2(1.0, 1.0)) / 2.0;
    vec4 color2 = texture(samp, stretchedCoord);
    color = mix(color1, color2, 0.5);
}

```

Shader #797 - purple haze

File: purple_haze.glsl

Blends multiple texture samples together using interpolation for a composite effect. Common parameters Strong purple color Wave 1: Diagonal Red Wave

Technique: 2D rotation matrix - Linear mix/lerp blending - Driven by: time

```

#version 330 core
in vec2 tc;
out vec4 color;

```



```

uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec2 center = vec2(0.5);
    vec3 purpleTint = vec3(0.7, 0.0, 0.7);
    float ripple1 = sin(tc.x * 12.0 + time_f * 5.0) * 0.03;
    ripple1 += sin(tc.y * 9.6 + time_f * 6.0) * 0.03;
    vec2 tc1 = tc + vec2(ripple1);
    tc1.y += sin(time_f * 2.5) * 0.02;
    vec2 pos1 = tc1 - center;
    float angle1 = length(pos1) * 8.0 + time_f * 3.0;
    mat2 rot1 = mat2(cos(angle1), -sin(angle1), sin(angle1), cos(angle1));
    tc1 = rot1 * pos1 + center;
    float ripple2 = sin(tc.x * 15.0 + time_f * 6.5) * 0.025;
    ripple2 += sin(tc.y * 4.5 + time_f * 4.5) * 0.025;
    vec2 tc2 = tc + vec2(ripple2 * 1.5, -ripple2 * 0.7);
    tc2.y += sin(time_f * 3.0) * 0.015;
    vec2 pos2 = tc2 - center;
    float angle2 = -length(pos2) * 6.0 + time_f * 2.5;
    mat2 rot2 = mat2(cos(angle2), -sin(angle2), sin(angle2), cos(angle2));
    tc2 = rot2 * pos2 + center;
    float ripple3 = sin(tc.x * 6.0 + time_f * 4.0) * 0.035;
    ripple3 += sin(tc.y * 14.0 + time_f * 5.2) * 0.035;
    vec2 tc3 = tc + vec2(ripple3 * 0.4, ripple3);
    tc3.y += sin(time_f * 4.5) * 0.025;
    vec2 pos3 = tc3 - center;
    float angle3 = length(pos3) * 10.0 + time_f * 4.0;
    mat2 rot3 = mat2(cos(angle3), -sin(angle3), sin(angle3), cos(angle3));
    tc3 = rot3 * pos3 + center;
    vec3 c = vec3(0.0);
    c.r += texture(samp, tc1).r * 1.2;
    c.b += texture(samp, tc2).b * 1.2;
    c.rb += texture(samp, tc3).rb * 0.8;
    color = vec4(c * purpleTint * 1.5, 1.0);
    color = mix(color, texture(samp, tc), 0.5);
    color = vec4(color.rgb, texture(samp, tc).a);
}

```

Shader #891 - ripple amp

File: ripple-amp.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Random phase per scanline Faster wave motion Slightly higher amplitude, with randomized modulation

Technique: Color quantization via floor rounding - Scanline overlay using modular y-coordinate
- Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
void main(void) {
    float yPos = gl_FragCoord.y / iResolution.y;
    float xPos = gl_FragCoord.x / iResolution.x;

```

```

float rndY = hash(floor(yPos * 200.0));
float rndX = hash(floor(xPos * 150.0));
float waveY = sin(yPos * (10.0 + rndY * 5.0) + time_f * (4.0 + rndX * 2.0));
float waveX = cos(xPos * (8.0 + rndX * 4.0) + time_f * (3.0 + rndY * 2.0));
float amplitude = 0.025 + 0.015 * hash(floor(time_f * 0.5));
vec2 offset = vec2(waveX, waveY) * amplitude;
vec2 coord = clamp(tc + offset, 0.001, 0.999);
color = texture(samp, coord);
}

```

Shader #892 - ripple

File: ripple.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright.

Technique: Linear mix/lerp blending - Distance field computation - Radial vignette falloff -
Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float ripple(vec2 pos, float time, float speed, float frequency) {
    float aspectRatio = iResolution.x / iResolution.y;
    pos.x *= aspectRatio;
    float d = distance(pos, vec2(0.5 * aspectRatio, 0.5));
    return sin(d * frequency - time * speed) * exp(-d * 3.0);
}
void main(void) {
    vec2 pos = tc;
    float time_t = mod(time_f, 10.0);
    float audioFreq = 12.0 + amp_mid * 20.0;
    float x = ripple(pos, time_f, 0.5, audioFreq);
    pos *= sin(x * time_t);
    color = texture(samp, pos);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #893 - ripple a

File: ripple_a.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
- Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
const float PI = 3.14159265359;
const vec3 NEON_COLOR = vec3(0.8, 0.2, 1.0);
float pingPong(float value, float length) {
    float modValue = mod(value, length * 2.0);
    return length - abs(modValue - length);
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec3 vortexEffect(vec2 uv, float time) {
    uv = uv * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float angle = atan(uv.y, uv.x) + time * 0.5;
    float radius = length(uv);
    uv = vec2(cos(angle), sin(angle)) * radius;
    float ripple = sin(radius * 10.0 - time * 5.0) * 0.05;
    uv += ripple * vec2(cos(angle), sin(angle));
    vec3 texColor = texture(samp, uv * 0.5 + 0.5).rgb;
    texColor += NEON_COLOR * exp(-10.0 * radius);
    return texColor;
}
void main(void) {
    vec2 uv = tc;
    vec3 colorEffect = vortexEffect(uv, time_f);
    float offset = 0.005 * sin(time_f);
    vec3 finalColor = vec3(
        vortexEffect(uv + vec2(offset, 0.0), time_f).r,
        colorEffect.g,
        vortexEffect(uv - vec2(offset, 0.0), time_f).b
    );
    color = vec4(finalColor, 1.0);
}
```

Shader #894 - ripple cache

File: ripple_cache.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. A quick pseudo-random noise function "uv" coordinates (same as tc) 1) Create a circular ripple effect centered in the middle (0.5,0.5)

Technique: Multi-tap convolution blur (5 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Temporal feedback using cached frame buffers - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
```

```

uniform sampler2D samp1;
uniform sampler2D samp2;
uniform sampler2D samp3;
uniform sampler2D samp4;
uniform vec2 iResolution;
uniform float time_f;
float rand2D(in vec2 n) {
    return fract(sin(dot(n, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void)
{
    vec2 uv = tc;
    float dist = length(uv - 0.5);
    float ripple = sin(dist * 30.0 - time_f * 5.0);
    vec2 rippleUV = uv + 0.02 * ripple * normalize(uv - 0.5);
    float glitch = rand2D(uv + time_f * 0.1) * 2.0 - 1.0;
    vec2 glitchOff = 0.01 * vec2(glitch, glitch);
    vec4 f1 = texture(samp1, rippleUV + glitchOff * 0.5);
    vec4 f2 = texture(samp2, rippleUV + glitchOff * 1.0);
    vec4 f3 = texture(samp3, rippleUV - glitchOff * 0.5);
    vec4 f4 = texture(samp4, rippleUV - glitchOff * 1.0);
    vec4 combined = (f1 + f2 + f3 + f4) * 0.25;
    vec4 baseTex = texture(samp, uv);
    float glitchFactor = sin(time_f * 2.0 + uv.y * 20.0) * 0.5 + 0.5;
    color = mix(baseTex, combined, glitchFactor);
    color.a = 1.0;
}

```

Shader #895 - ripple cycle

File: ripple_cycle.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 normPos = gl_FragCoord.xy / iResolution.xy;
    float cycle = sin(time_f * 0.5);
    float movingPhase = normPos.x + cycle;
    float phase = sin(movingPhase * 10.0 - time_f * 2.0);
    vec2 tcAdjusted = tc + (vec2(phase, 0) * 0.302);
    color = texture(samp, tcAdjusted);
}

```

Shader #896 - ripple cycle2

File: ripple_cycle2.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: UV mirroring/folding for symmetry - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 normPos = gl_FragCoord.xy / iResolution.xy;
    vec2 mirroredPos = vec2(
        abs(fract(normPos.x * 2.0) - 0.5) * 2.0,
        abs(fract(normPos.y * 2.0) - 0.5) * 2.0
    );
    float time_t = pingPong(time_f, 6.0) + 2.0;
    float cycle = sin(time_t * 0.5);
    float movingPhase = mirroredPos.x + mirroredPos.y + cycle;
    float phase = sin(movingPhase * 10.0 - time_f * 2.0);
    vec2 tcAdjusted = tc + vec2(phase * 0.302, phase * 0.302);
    color = texture(samp, tcAdjusted);
}

```

Shader #897 - ripple cycle circle

File: ripple_cycle_circle.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - UV mirroring/folding for symmetry
- Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 normPos = gl_FragCoord.xy / iResolution.xy;
    vec2 mirroredPos = vec2(
        abs(fract(normPos.x * 2.0) - 0.5) * 2.0,
        abs(fract(normPos.y * 2.0) - 0.5) * 2.0
    );
    float time_t = pingPong(time_f, 6.0) + 2.0;
    float cycle = sin(time_t * 0.5);
    float movingPhase = mirroredPos.x + mirroredPos.y + cycle;
    float phase = sin(movingPhase * 10.0 - time_f * 2.0);
    vec2 centeredPos = mirroredPos - 0.5;
    float radius = length(centeredPos);
    float angle = atan(centeredPos.y, centeredPos.x) + radius * 5.0 - time_f * 1.5;
    vec2 spiralOffset = radius * vec2(cos(angle), sin(angle));
    float wave = sin(mirroredPos.y * 10.0 + time_f * 2.0) * 0.02;
}

```

```

    vec2 waveOffset = vec2(sin(wave * time_t), 0);
    vec2 tcAdjusted = tc + vec2(phase * 0.302, phase * 0.302) + spiralOffset * 0.1 + waveOffset;
    color = texture(samp, tcAdjusted);
}

```

Shader #898 - ripple cycle prism

File: ripple_cycle_prism.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    vec2 reflectedUV = vec2(1.0 - uv.x, uv.y);
    float noise = sin(reflectedUV.y * 50.0 + time_f * 5.0) * 0.01;
    float distortionX = noise;
    float distortionY = cos(reflectedUV.x * 50.0 + time_f * 5.0) * 0.01;
    vec2 distortion = vec2(distortionX, distortionY);
    float r = texture(samp, reflectedUV + distortion * 1.2).r;
    float g = texture(samp, reflectedUV + distortion * 0.8).g;
    float b = texture(samp, reflectedUV + distortion * 0.4).b;
    color = vec4(r, g, b, 1.0);
}

```

Shader #899 - ripple left

File: ripple_left.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 normTC = gl_FragCoord.xy / iResolution.xy;
    float diagonalDist = length(vec2(normTC.x + normTC.y - 1.0, normTC.x - normTC.y));
    float phase = sin(diagonalDist * 20.0 - time_f * 5.0);
    vec2 tcAdjusted = tc + (normTC * 0.301 * phase);
    color = texture(samp, tcAdjusted);
}

```

Shader #901 - ripple out

File: ripple_out.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 normPos = (gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0;
    float dist = length(normPos);
    float phase = sin(dist * 10.0 - time_f * 4.0);
    vec2 tcAdjusted = tc + (normPos * 0.305 * phase);
    color = texture(samp, tcAdjusted);
}
```

Shader #903 - ripple ppong

File: ripple_ppong.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 center = iResolution / 2.0;
    vec2 uv = tc * iResolution;
    float distanceFromCenter = length(uv - center);
    float rippleSpeed = 2.0;
    float rippleFrequency = 20.0;
    float time = time_f * rippleSpeed;
    float wave = sin(pingPong(distanceFromCenter + time, rippleFrequency));
    float warpStrength = 0.05;
    float warpFrequency = 5.0;
    vec2 warp = vec2(sin(uv.y * warpFrequency + time), sin(uv.x * warpFrequency + time)) * warpStrength;
    vec2 rippleUV = tc + normalize(uv - center) * wave * 0.01 + warp;
    color = texture(samp, rippleUV);
}
```

Shader #904 - ripple prisim

File: ripple_prisim.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.
Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float pingPong(float x,float length){
    float m=mod(x,length*2.0);
    return m<=length?m:length*2.0-m;
}
void main(void){
    float A=clamp(amp,0.0,1.0);
    float U=clamp(uamp,0.0,1.0);
    float time_z=pingPong(time_f,4.0)+0.5;
    vec2 uv=tc;
    vec2 center=(iMouse.z>0.5||iMouse.w>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 normPos=(uv-center)*vec2(iResolution.x/iResolution.y,1.0);
    float dist=length(normPos);
    float phase=sin(dist*10.0-time_f*4.0);
    float phaseGain=mix(0.6,1.4,A);
    vec2 tcAdjusted=uv+(normPos*0.305*phase*phaseGain);
    float dispersionScale=0.02*(0.5+U);
    vec2 dispersionOffset=normPos*dist*dispersionScale;
    vec2 tcAdjustedR=tcAdjusted-dispersionOffset;
    vec2 tcAdjustedG=tcAdjusted;
    vec2 tcAdjustedB=tcAdjusted+dispersionOffset;
    float r=texture(samp,tcAdjustedR).r;
    float g=texture(samp,tcAdjustedG).g;
    float b=texture(samp,tcAdjustedB).b;
    color=vec4(r,g,b,1.0);
}

```

Shader #905 - ripple prism

File: ripple_prism.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 normPos = (gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0;
    float dist = length(normPos);
    float phase = sin(dist * 10.0 - time_f * 4.0);
    vec2 tcAdjusted = tc + (normPos * 0.305 * phase);
    float dispersionScale = 0.02;
    vec2 dispersionOffset = normPos * dist * dispersionScale;
    vec2 tcAdjustedR = tcAdjusted + dispersionOffset * (-1.0);
    vec2 tcAdjustedG = tcAdjusted;
    vec2 tcAdjustedB = tcAdjusted + dispersionOffset * 1.0;
}

```



```

float r = texture(samp, tcAdjustedR).r;
float g = texture(samp, tcAdjustedG).g;
float b = texture(samp, tcAdjustedB).b;
color = vec4(r, g, b, 1.0);
}

```

Shader #906 - ripple rainbow

File: ripple_rainbow.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float wave = sin(uv.x * 10.0 + time_f * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_f) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_f * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_f * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    float time_t = mod(time_f, 30.0);
    vec2 normPos = (gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0;
    float dist = length(normPos);
    float phase = sin(dist * 10.0 - time_f * 4.0);
}

```

```

    vec2 tcAdjusted = tc + (normPos * 0.305 * phase);
    vec4 adjusted_color = texture(samp, tcAdjusted);
    vec3 final_color = mix(blended_color, adjusted_color.rgb, 0.5);
    color = vec4(sin(final_color * time_t), original_color.a);
}

```

Shader #940 - sine ripple

File: sine_ripple.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec3 rainbow(float t) {
    float hue = mod(t, 1.0) * 6.0;
    float c = 1.0;
    float x = 1.0 - abs(mod(hue, 2.0) - 1.0);
    vec3 rgb = hue < 1.0 ? vec3(c, x, 0.0) :
                hue < 2.0 ? vec3(x, c, 0.0) :
                hue < 3.0 ? vec3(0.0, c, x) :
                hue < 4.0 ? vec3(0.0, x, c) :
                hue < 5.0 ? vec3(x, 0.0, c) :
                vec3(c, 0.0, x);

    return rgb;
}

void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float rippleFrequency = 10.0;
    float rippleAmplitude = 0.02;
    float ripple = sin(dist * rippleFrequency - time_f * 5.0) * rippleAmplitude;
    vec2 ripple_tc = tc + normalize(dir) * ripple;
    ripple_tc = clamp(ripple_tc, vec2(0.0), vec2(1.0));
    float gradient_pos = mod(dist * 3.0 + angle / (2.0 * 3.14159) + time_f * 0.5, 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec4 ctx = texture(samp, ripple_tc);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = mix(sin(ctx * time_t), vec4(color_gradient, 1.0), 0.5);
    color.a = 1.0;
}

```

Shader #1022 - tripple

File: tripple.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    float distance = length(uv - center);
    uv.x += tan(distance * 20.0 - time_f * 2.0) * 0.02;
    uv.y += tan(distance * 20.0 - time_f * 2.0) * 0.02;
    color = texture(samp, uv);
}
```

Shader #1023 - tripple2

File: tripple2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    float distance = length(uv - center);
    float time_t = pingPong(time_f, 15.0) + 2.0;
    distance = sin(distance * time_t);
    uv.x += tan(distance * 20.0 - time_f * 2.0) * 0.02;
    uv.y += tan(distance * 20.0 - time_f * 2.0) * 0.02;
    color = texture(samp, uv);
}
```

Shader #1035 - underwater

File: underwater.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main() {
    float t = time_f;
    vec2 uv = tc;
    vec2 p = uv * 2.0 - 1.0;
    float r2 = dot(p, p);
    p += p * r2 * 0.035 * sin(t * 0.4);
    uv = p * 0.5 + 0.5;
    vec2 d1 = vec2(sin(uv.y * 12.0 - t * 2.0), cos(uv.x * 12.0 + t * 1.6)) * (0.015 + amp_low * 0.03);
    vec2 d2 = vec2(sin((uv.x + uv.y) * 24.0 + t * 1.2), -cos((uv.x - uv.y) * 24.0 - t * 1.8)) * (0.009 +
    ↪ amp_mid * 0.02);
    vec2 d3 = vec2(cos(uv.y * 40.0 + t * 3.5), sin(uv.x * 40.0 - t * 3.0)) * (0.003 + amp_high * 0.01);
    uv += d1 + d2 + d3;
    uv = clamp(uv, 0.0, 1.0);
    color = texture(samp, uv);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #1036 - underwaterenchanced

File: underwaterenchanced.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
float h1(float n){return fract(sin(n)*43758.5453123);}
vec2 h2(float n){return fract(sin(vec2(n,n+1.0))*vec2(43758.5453,22578.1459));}
void main() {
    float t = time_f;
    vec2 uv = tc;
    float rate = 0.35;
    float k = floor(t*rate);
    float a = fract(t*rate);
    a = a*a*(3.0-2.0*a);
    vec2 p0 = h2(k);
    vec2 p1 = h2(k+1.0);
}

```

```

    vec2 shift = (mix(p0,p1,a)-0.5)*0.08;
    vec2 c = vec2(0.5)+shift;
    vec2 p = uv - c;
    float r2 = dot(p, p);
    p += p * r2 * (0.035 + 0.01*h1(k+7.0)) * sin(t * (0.4 + 0.05*h1(k+13.0)));
    uv = p + c;
    vec2 d1 = vec2(sin(uv.y * 12.0 - t * 2.0), cos(uv.x * 12.0 + t * 1.6)) * 0.015;
    vec2 d2 = vec2(sin((uv.x + uv.y) * 24.0 + t * 1.2), -cos((uv.x - uv.y) * 24.0 - t * 1.8)) * 0.009;
    vec2 d3 = vec2(cos(uv.y * 40.0 + t * 3.5), sin(uv.x * 40.0 - t * 3.0)) * 0.003;
    uv += d1 + d2 + d3;
    uv = clamp(uv, 0.0, 1.0);
    color = texture(samp, uv);
}

```

Shader #1049 - water fold ex.

File: water-fold-ex.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
vec3 getRainbowColor(float position) {
    float r = sin(position + 0.0) * 0.5 + 0.5;
    float g = sin(position + 2.0) * 0.5 + 0.5;
    float b = sin(position + 4.0) * 0.5 + 0.5;
    return vec3(r, g, b);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float a = clamp(amp, 0.0, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    float waveAmp = mix(0.02, 0.12, a);
    float wave = sin((tc.y - m.y) * 10.0 + time_f * (1.0 + a * 3.0)) * waveAmp;
    vec2 new_tc = vec2(tc.x + wave, tc.y);
    new_tc = clamp(new_tc, vec2(0.0), vec2(1.0));
    vec4 texColor = texture(samp, new_tc);
    vec2 p = tc - m;
    float c = cos(time_f);
    float s = sin(time_f);
    vec2 pr = vec2(p.x * c - p.y * s, p.x * s + p.y * c);
    float rainbowPos = length(pr) * 10.0 + time_f * 5.0;
    vec3 rainbowColor = getRainbowColor(sin(rainbowPos * time_t));
    float mixAmt = 0.25 + 0.5 * a;
    vec3 mixed = mix(texColor.rgb, rainbowColor, mixAmt);
    vec4 outc = vec4(mixed, texColor.a);
    color = sin(outc * time_t);
}

```

```
    color.a = 1.0;
}
```

Shader #1050 - water

File: water.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float speed = 5.0;
    float amplitude = 0.03;
    float wavelength = 10.0;
    float ripple = sin(tc.x * wavelength + time_f * speed) * amplitude;
    ripple += sin(tc.y * wavelength + time_f * speed) * amplitude;
    vec2 rippleTC = tc + vec2(ripple, ripple);
    vec4 originalColor = texture(samp, tc);
    vec4 rippleColor = texture(samp, rippleTC);
    color = mix(originalColor, rippleColor, 0.5);
}
```

Shader #1051 - water full

File: water_full.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

color = mix(originalColor, rippleColor, 0.5);

Technique: Linear mix/lerp blending - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float speed = 5.0;
    float amplitude = 0.03;
    float wavelength = 10.0;
    float ripple = sin(tc.x * wavelength + time_f * speed) * amplitude;
    ripple += sin(tc.y * wavelength + time_f * speed) * amplitude;
    vec2 rippleTC = tc + vec2(ripple, ripple);
    vec4 originalColor = texture(samp, tc);
    vec4 rippleColor = texture(samp, rippleTC);
    color = rippleColor;
}
```

Shader #1052 - water prism

File: water_prism.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec3 hsv2rgb(vec3 c)
{
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    float time_t = pingPong(time_f, 7.0);
    vec2 uv = tc * 2.0 - 1.0;
    float frequency = 10.0;
    float amplitude = 0.02;
    float waveX = sin(uv.y * frequency + time_t) * amplitude;
    waveX += sin(uv.y * frequency * 0.5 + time_t * 1.5) * amplitude * 0.5;
    float waveY = sin(uv.x * frequency + time_t * 0.8) * amplitude;
    waveY += sin(uv.x * frequency * 0.5 + time_t * 1.2) * amplitude * 0.5;
    uv += vec2(waveX, waveY);
    float hue = mod((uv.y + 1.0) * 0.5 + time_t * 0.1, 1.0);
    float saturation = 1.0;
    float value = 1.0;
    vec3 neonColor = hsv2rgb(vec3(hue, saturation, value));
    vec4 texColor = texture(samp, uv * 0.5 + 0.5);
    float depth = (uv.y + 1.0) * 0.5;
    float attenuation = mix(1.0, 0.6, depth);
    vec3 finalColor = mix(neonColor, sin(texColor.rgb * time_t), 0.5) * attenuation;
    color = vec4(finalColor, 1.0);
}
```

Shader #1053 - water r

File: water_r.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Channel-specific parameters Red channel wave (diagonal) Green channel wave (horizontal emphasis)

Technique: Driven by: time

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
```

```

float speedR = 5.0;
float amplitudeR = 0.03;
float wavelengthR = 10.0;
float speedG = 6.5;
float amplitudeG = 0.025;
float wavelengthG = 12.0;
float speedB = 4.0;
float amplitudeB = 0.035;
float wavelengthB = 8.0;
float rippleR = sin(tc.x * wavelengthR + time_f * speedR) * amplitudeR;
rippleR += sin(tc.y * wavelengthR * 0.8 + time_f * speedR * 1.2) * amplitudeR;
vec2 rippleTC_R = tc + vec2(rippleR, rippleR);
float rippleG = sin(tc.x * wavelengthG * 1.5 + time_f * speedG) * amplitudeG;
rippleG += sin(tc.y * wavelengthG * 0.3 + time_f * speedG * 0.7) * amplitudeG;
vec2 rippleTC_G = tc + vec2(rippleG, -rippleG * 0.5);
float rippleB = sin(tc.x * wavelengthB * 0.5 + time_f * speedB) * amplitudeB;
rippleB += sin(tc.y * wavelengthB * 1.7 + time_f * speedB * 1.3) * amplitudeB;
vec2 rippleTC_B = tc + vec2(rippleB * 0.3, rippleB);
vec4 originalColor = texture(samp, tc);
originalColor.r = texture(samp, rippleTC_R).r;
originalColor.g = texture(samp, rippleTC_G).g;
originalColor.b = texture(samp, rippleTC_B).b;
color = originalColor;
}

```

Shader #1054 - water rgb

File: water_rgb.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.
Create wave displacements Pattern configuration R and B mirrored

Technique: Driven by: time

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float speedR = 5.0;
    float amplitudeR = 0.03;
    float wavelengthR = 10.0;
    float speedG = 6.5;
    float amplitudeG = 0.025;
    float wavelengthG = 12.0;
    float speedB = 4.0;
    float amplitudeB = 0.035;
    float wavelengthB = 8.0;
    float rippleR = sin(tc.x * wavelengthR + time_f * speedR) * amplitudeR;
    rippleR += sin(tc.y * wavelengthR * 0.8 + time_f * speedR * 1.2) * amplitudeR;
    vec2 rippleTC_R = tc + vec2(rippleR, rippleR);
    float rippleG = sin(tc.x * wavelengthG * 1.5 + time_f * speedG) * amplitudeG;
    rippleG += sin(tc.y * wavelengthG * 0.3 + time_f * speedG * 0.7) * amplitudeG;
    vec2 rippleTC_G = tc + vec2(rippleG, -rippleG * 0.5);
    float rippleB = sin(tc.x * wavelengthB * 0.5 + time_f * speedB) * amplitudeB;
    rippleB += sin(tc.y * wavelengthB * 1.7 + time_f * speedB * 1.3) * amplitudeB;
    vec2 rippleTC_B = tc + vec2(rippleB * 0.3, rippleB);
    const vec3 patterns[4] = vec3[(
        vec3(1.0, 0.0, 1.0),

```



```

        vec3(0.0, 1.0, 0.0),
        vec3(1.0, 0.0, 0.0),
        vec3(0.0, 0.0, 1.0)
    );
    float patternSpeed = 4.0;
    int patternIndex = int(mod(time_f * patternSpeed, 4.0));
    vec3 mirrorFlags = patterns[patternIndex];
    vec2 finalTC_R = vec2(mirrorFlags.r > 0.5 ? 1.0 - rippleTC_R.x : rippleTC_R.x, rippleTC_R.y);
    vec2 finalTC_G = vec2(mirrorFlags.g > 0.5 ? 1.0 - rippleTC_G.x : rippleTC_G.x, rippleTC_G.y);
    vec2 finalTC_B = vec2(mirrorFlags.b > 0.5 ? 1.0 - rippleTC_B.x : rippleTC_B.x, rippleTC_B.y);
    vec4 originalColor = texture(samp, tc);
    originalColor.r = texture(samp, finalTC_R).r;
    originalColor.g = texture(samp, finalTC_G).g;
    originalColor.b = texture(samp, finalTC_B).b;
    color = originalColor;
}

```

Shader #1055 - waterbend

File: waterbend.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float PI = 3.141592653589793;
float mirror1(float x, float m) {
    float p = mod(x, m * 2.0);
    return p > m ? (2.0 * m - p) : p;
}
vec2 kaleido(vec2 p, vec2 c, float seg) {
    vec2 d = p - c;
    float a = atan(d.y, d.x);
    float r = length(d);
    float m = PI / max(2.0, seg);
    a = mirror1(a, m);
    return c + vec2(cos(a), sin(a)) * max(r, 1e-6);
}
void main(void) {
    vec2 uv = tc;
    vec2 mouse = iMouse.xy / iResolution.xy;
    mouse = clamp(mouse, 0.0, 1.0);
    float seg = floor(4.0 + (amp + uamp) * 8.0);
    vec2 k = kaleido(uv, mouse, seg);
    float r = length(k - mouse);
    vec2 dir = normalize(k - mouse + vec2(1e-6, 0.0));
    float bendBase = 0.15 + 0.35 * uamp;
    float bend = bendBase / (r + 0.02);

```

```

    vec2 wob = 0.02 * (0.5 + amp) * vec2(sin(40.0 * r - 2.0 * time_f), cos(40.0 * r - 1.5 * time_f));
    vec2 base = k + dir * bend + wob;
    float disp = (0.002 + 0.01 * amp) / (r + 0.02);
    vec2 tR = base + dir * disp;
    vec2 tG = base;
    vec2 tB = base - dir * disp;
    vec3 c;
    c.r = texture(samp, tR).r;
    c.g = texture(samp, tG).g;
    c.b = texture(samp, tB).b;
    vec2 dk = k - mouse;
    float a = atan(dk.y, dk.x);
    float wedge = PI / max(2.0, seg);
    float edge = abs(mod(a + wedge, 2.0 * wedge) - wedge) / wedge;
    float edgeGlow = pow(1.0 - edge, 6.0) * (0.4 + 0.6 * amp);
    c += edgeGlow;
    float influence = smoothstep(0.35, 0.0, length(uv - mouse));
    vec3 orig = texture(samp, uv).rgb;
    vec3 outc = mix(orig, c, influence);
    color = vec4(outc, 1.0);
}

```

XOR & Math (55 shaders)

Shader #28 - alpha xor

File: alpha_xor.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = color[0] * alpha_r;
    color[1] = color[1] * alpha_g;
    color[2] = color[2] * alpha_b;
}

```

```

    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #113 - color effect xor

File: color_effect_xor.glsl

Generates procedural noise patterns combined with sinusoidal distortion. + feedback * 0.01 warp
 /* feedback 0.01*/;

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void) {
    vec2 uv = tc;
    vec2 warp = uv + vec2(
        sin(uv.y * 10.0 + time_f) * 0.1,
        sin(uv.x * 10.0 + time_f) * 0.1
    );
    vec3 colorShift = vec3(
        0.5 * sin(time_f * 0.5) + 0.5,
        0.5 * sin(time_f * 0.7 + 2.0) + 0.5,
        0.5 * sin(time_f * 0.3 + 4.0) + 0.5
    );
    float feedback = rand(uv + time_f);
    vec2 feedbackUv = tc;
    float time_t = mod(time_f, 10);
    vec4 texColor = texture(samp, feedbackUv);
    vec3 finalColor = texColor.rgb + colorShift;
    color = vec4(finalColor, texColor.a);
    color = xor_RGB(texture(samp, tc), color);
    color.a = 1.0;
}

```

```
}
```

Shader #134 - color xor

File: color_xor.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = color[0] * fract(alpha_r*timeval);
    color[1] = color[1] * fract(alpha_g*timeval);
    color[2] = color[2] * fract(alpha_b*timeval);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #182 - dark addup xor

File: dark_addup_xor.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    float val = (color[0]+color[1]+color[2])*alpha;
    for(int q = 0; q < 3; ++q)
        color[q] = color[q] * (sin(val * color[q]/0.3));
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #213 - diamond

File: diamond.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 hash3(vec2 p) {
    vec3 q = vec3(dot(p, vec2(127.1, 311.7)),
                  dot(p, vec2(269.5, 183.3)),
                  dot(p, vec2(419.2, 371.9)));
    return fract(sin(q) * 43758.5453);
}
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;

```

```

    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}

void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    vec2 skewedUv = uv;
    skewedUv.x = (uv.x + uv.y) * 0.5;
    skewedUv.y = (uv.x - uv.y) * 0.5;
    float scale = abs(sin(time_f)) * 20.0 + 1.0;
    skewedUv *= scale;
    vec2 c = floor(skewedUv);
    vec2 checker = mod(c, 2.0);
    bool isEvenDiamond = (checker.x == checker.y);
    vec3 diamondColor = hash3(c);
    vec3 finalColor = isEvenDiamond ? diamondColor * 0.8 : diamondColor * 0.5;
    color = xor_RGB(vec4(finalColor, 1.0), texture(samp, tc));
}

```

Shader #218 - disk2

File: disk2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Hermite smoothstep interpolation
 - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float spinningSpeed = mix(5.0, 1.0, radius);

```

```

float spinningAngle = angle + time_f * spinningSpeed;
vec2 spinningUV = vec2(cos(spinningAngle), sin(spinningAngle)) * radius * 0.5 + 0.5;
vec4 texColor = texture(samp, spinningUV);
float centerGlow = exp(-radius * 10.0);
float outerGlow = smoothstep(0.4, 0.5, radius) * 0.5 + 0.5;
float glow = max(centerGlow, outerGlow);
color = texColor * vec4(glow, glow, glow, 1.0);
color = xor_RGB(color, texture(samp, tc));
}

```

Shader #263 - echo alpha two xor

File: echo_alpha_two_xor.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(color * 255);
    vec4 color2 = texture(samp, tc / (0.9 * alpha));
    vec4 color3 = texture(samp, tc / (1.5 * alpha));
    vec4 color4 = texture(samp, tc / (2.0 * alpha));
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    color = xor_RGB(color, source);
}

```

Shader #264 - echo and or xor

File: echo_and_or_xor.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / (0.9 + alpha));
    vec4 color3 = texture(samp, tc / (1.5 + alpha));
    vec4 color4 = texture(samp, tc / (2.0 + alpha));
    ivec4 source = ivec4(color * 255);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4);
    ivec4 blend = ivec4(color * 255);
    blend[0] = source[0] & blend[0];
    blend[1] = source[1] | blend[1];
    blend[2] = source[2] ^ blend[2];
    color = xor_RGB(blend, source);
}
```

Shader #286 - echo or and xor

File: echo_or_and_xor.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
```



```

in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / (0.9 + alpha));
    vec4 color3 = texture(samp, tc / (1.5 + alpha));
    vec4 color4 = texture(samp, tc / (2.0 + alpha));
    ivec4 source = ivec4(color * 255);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    ivec4 blend = ivec4(color * 255);
    blend[0] = source[0] | blend[0];
    blend[1] = source[1] & blend[1];
    blend[2] = source[2] ^ blend[2];
    color = xor_RGB(blend, source);
}

```

Shader #308 - echo xor

File: echo_xor.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;

```

```

uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc /0.9);
    vec4 color3 = texture(samp, tc/ 1.5);
    vec4 color4 = texture(samp, tc/ 2.0);
    ivec4 icolor1 = ivec4(color2 * 255);
    ivec4 icolor2 = ivec4(color3 * 255);
    ivec4 icolor3 = ivec4(color4 * 255);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    ivec4 icolor = ivec4(color * 255);
    icolor[0] = icolor[0] ^ icolor1[2];
    icolor[1] = icolor[1] ^ icolor2[1];
    icolor[2] = icolor[2] ^ icolor3[0];
    color = xor_RGB(color, icolor);
}

```

Shader #309 - echo xor color blend

File: echo_xor_color_blend.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;

```

```

uniform float time_f;
uniform float restore_black;
vec4 color_blend(vec4 color) {
    vec4 color2 = color;
    ivec4 color_source = ivec4(color * 255);
    color = color*alpha;
    ivec4 color_i = ivec4(color * 255);
    for(int i = 0; i < 3; ++i) {
        if(color_i[i] >= 255)
            color_i[i] = color_i[i]%255;
        if(color_source[i] >= 255)
            color_source[i] = color_source[i]%255;
        color_i[i] = color_i[i] ^ color_source[i];
        color[i] = float(color_i[i])/255;
    }
    for(int i = 0; i < 3; ++i)
        if(color[i] < 0.2) color[i] = color2[i];
    return color;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    color = color_blend(color);
}

```

Shader #310 - echo xor or and

File: echo_xor_or_and.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)

```

```

        int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / (0.9 + alpha));
    vec4 color3 = texture(samp, tc / (1.5 + alpha));
    vec4 color4 = texture(samp, tc / (2.0 + alpha));
    ivec4 source = ivec4(color * 255);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    ivec4 blend = ivec4(color * 255);
    blend[0] = source[0] ^ blend[0];
    blend[1] = source[1] | blend[1];
    blend[2] = source[2] & blend[2];
    color = xor_RGB(blend, source);
}

```

Shader #332 - faster

File: faster.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Per-channel UV offset for chromatic aberration - Linear mix/lerp blending -
Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
float h1(float n){return fract(sin(n)*43758.5453123);}
float h2(vec2 p){return fract(sin(dot(p,vec2(12.9898,78.233)))*43758.5453);}
vec4 xor_RGB(vec4 icolor, vec4 source){
    ivec3 ic; ivec4 isrc = ivec4(source*255.0);
    for(int i=0;i<3;++i){
        ic[i]=int(255.0*icolor[i]);
        ic[i]=ic[i]^isrc[i];
        if(ic[i]>255) ic[i]=ic[i]%255;
        icolor[i]=float(ic[i])/255.0;
    }
    icolor.a=1.0;
    return icolor;
}
vec3 fastBlurRot(sampler2D tex, vec2 uv, vec2 res, float t){
    float ang = t*2.3 + h1(floor(t*0.37))*6.2831853;
    vec2 dir = vec2(cos(ang), sin(ang));
    vec2 px = dir / res;
    vec3 a = texture(tex, uv - 4.0*px).rgb;
    vec3 b = texture(tex, uv - 2.5*px).rgb;
    vec3 c = texture(tex, uv - 1.5*px).rgb;
}

```

```

    vec3 d = texture(tex, uv - 0.5*px).rgb;
    vec3 e = texture(tex, uv + 0.5*px).rgb;
    vec3 f = texture(tex, uv + 1.5*px).rgb;
    vec3 g = texture(tex, uv + 2.5*px).rgb;
    vec3 h = texture(tex, uv + 4.0*px).rgb;
    vec3 m = texture(tex, uv).rgb;
    return (a*0.05+b*0.08+c*0.12+d*0.15+m*0.2+e*0.15+f*0.12+g*0.08+h*0.05);
}

float pingPong(float x,float l){
    float m=mod(x,l*2.0);
    return m<=l?m:l*2.0-m;
}

void main(void){
    float t=time_f;
    float speed = 2.0 + h1(floor(t*0.31))*2.5;
    float amp = 0.015 + 0.01*h1(floor(t*0.53));
    vec2 jitter = vec2(h2(tc*123.0+t), h2(tc*321.0-t))*amp*0.75;
    vec2 uv = tc + jitter;
    vec3 base = fastBlurRot(samp, uv, iResolution, t*speed);
    float cyc = mod(t*(1.0+0.4*h1(floor(t*0.19))), 6.0);
    vec3 gate = vec3(1.0);
    if(cyc<1.0){ gate.r = mix(0.0,1.0,cyc); gate.g=0.0; gate.b=0.0; }
    else if(cyc<2.0){ gate.r=1.0; gate.g=mix(0.0,1.0,cyc-1.0); gate.b=0.0; }
    else if(cyc<3.0){ gate.r=1.0; gate.g=1.0; gate.b=mix(0.0,1.0,cyc-2.0); }
    else if(cyc<4.0){ gate=vec3(1.0); gate.b=mix(1.0,alpha_b,cyc-3.0); }
    else if(cyc<5.0){ gate.r=1.0; gate.b=alpha_b; gate.g=mix(1.0,alpha_g,cyc-4.0); }
    else { gate.g=alpha_g; gate.b=alpha_b; gate.r=mix(1.0,alpha_r,cyc-5.0); }
    vec2 chrom = (vec2(cos(t*0.7), sin(t*0.9))*0.75 + vec2(0.3,-0.2))* (0.35+0.35*h1(floor(t*0.27)));
    chrom /= iResolution;
    vec3 chromSample;
    chromSample.r = texture(samp, uv + chrom*1.0).r;
    chromSample.g = texture(samp, uv).g;
    chromSample.b = texture(samp, uv - chrom*1.0).b;
    vec3 mixin = mix(base, chromSample, 0.35 + 0.25*h1(floor(t*0.41)));
    vec4 gateColor = vec4(gate,1.0);
    float tt = pingPong(t, 20.0) + 2.0;
    vec4 xored = xor_RGB(vec4(mixin,1.0), gateColor);
    vec4 s = sin(xored * tt);
    s.a = 1.0;
    color = s;
}

```

Shader #339 - fill white

File: fill_white.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;

```

```

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    if(color[0] > 0.9 && color[1] > 0.9 && color[2] > 0.9) {
        vec4 color2 = texture(samp, tc / (0.9 + alpha));
        vec4 color3 = texture(samp, tc / (1.5 + alpha));
        vec4 color4 = texture(samp, tc / (2.0 + alpha));
        ivec4 source = ivec4(color * 255);
        color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
        ivec4 blend = ivec4(color * 255);
        blend[0] = source[0] ^ blend[0];
        blend[1] = source[1] | blend[1];
        blend[2] = source[2] & blend[2];
        color = xor_RGB(blend, source);
    }
}

```

Shader #385 - frac shader02 dmidi6i zoom xor

File: frac_shader02_dmidi6i_zoom_xor.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;

```

```

uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
vec4 xor_RGB(vec4 icolor, vec4 source){
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255.0);
    for(int i=0;i<3;++i){
        int_color[i] = int(255.0*icolor[i]);
        int_color[i] = int_color[i] ^ isource[i];
        if(int_color[i]>255) int_color[i] = int_color[i] % 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}

float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}

vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);

```

```

    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5 || iMouse.w > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float zoomPhase = time_f * 0.12;
    float zCycle = floor(zoomPhase);
    float zLocal = fract(zoomPhase);
    float tri = 1.0 - abs(zLocal * 2.0 - 1.0);
    float sgn = mix(-1.0, 1.0, step(0.5, mod(zCycle, 2.0)));
    float depth = 1.0 + zCycle * 0.35;
    float zoomExp = sgn * tri * depth;
    float zoom = pow(1.45, zoomExp);
    vec2 z = tc - m;
    z.x *= aspect;
    z /= zoom;
}

```



```

z.x /= aspect;
vec2 zoomTC = fract(z + m);
vec4 baseTex = texture(samp, zoomTC);
vec2 uv = zoomTC * 2.0 - 1.0;
uv.x *= aspect;
float r = pingPong(sin(length(uv) * time_f), 5.0);
float radius = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
vec3 baseCol = preBlendColor(zoomTC);
float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
vec2 kUV = reflectUV(zoomTC, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
↵;
float vign = 1.0 - smoothstep(0.75, 1.2, length((zoomTC - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalCore = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
vec2 center = m;
vec2 pSwirl = (zoomTC - center) * vec2(1.0, iResolution.y / iResolution.x);
float intensity = pingPong(time_f, 10.0);
float angleS = atan(pSwirl.y, pSwirl.x);
float radiusS = length(pSwirl);
float swirl = sin(time_f * 0.5) * 0.5 + 0.5;
angleS += intensity * swirl * sin(radiusS * 10.0 + time_f);
vec2 qSwirl = vec2(cos(angleS), sin(angleS)) * radiusS;
vec2 uvSwirl = qSwirl * vec2(1.0, iResolution.x / iResolution.y) + center;
vec4 texSwirl = texture(samp, uvSwirl);
float fluctuation = sin(time_f * 2.0) * 0.5 + 0.5;
vec3 xorPalette = mix(vec3(1.0, 0.0, 0.0), vec3(0.0, 0.0, 1.0), fluctuation);
xorPalette = mix(xorPalette, neonPalette(time_f), 0.5);

```

```

    vec4 fluctuatedColor = vec4(xorPalette, 1.0);
    vec4 xorResult = xor_RGB(texSwirl, fluctuatedColor);
    vec3 xorMix = mix(texSwirl.rgb, xorResult.rgb, 0.6);
    float xorStrength = (0.25 + 0.35 * pulse) * vign;
    xorStrength = clamp(xorStrength, 0.0, 1.0);
    vec3 finalRGB = mix(finalCore, xorMix, xorStrength);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #386 - frac shader02 dmdi6i zoom xor amp

File: frac_shader02_dmdi6i_zoom_xor_amp.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float gAmp01;
float gSlow;
float gFast;
float gDetail;
vec4 xor_RGB(vec4 icolor, vec4 source){
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255.0);
    for(int i=0;i<3;++i){
        int_color[i] = int(255.0*icolor[i]);
        int_color[i] = int_color[i] ^ isource[i];
        if(int_color[i]>255) int_color[i] = int_color[i] % 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {

```

```

    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;

```

```

    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    float aAcc = clamp(amp, 0.0, 4.0);
    float aInst = clamp(uamp, 0.0, 4.0);
    float ampMix = clamp(aAcc * 0.6 + aInst * 1.4, 0.0, 4.0);
    gAmp01 = clamp(ampMix / 2.5, 0.0, 1.0);
    gSlow = time_f * mix(0.15, 0.7, gAmp01);
    gFast = time_f * mix(0.6, 3.5, gAmp01);
    gDetail = time_f * mix(0.3, 2.0, gAmp01);
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5 || iMouse.w > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float zoomPhase = gSlow * 0.12;
    float zCycle = floor(zoomPhase);
    float zLocal = fract(zoomPhase);
    float tri = 1.0 - abs(zLocal * 2.0 - 1.0);
    float sgn = mix(-1.0, 1.0, step(0.5, mod(zCycle, 2.0)));
    float depth = 1.0 + zCycle * 0.35 + gAmp01 * 1.2;
    float zoomExp = sgn * tri * depth;
    float zoom = pow(1.45, zoomExp);
    vec2 z = tc - m;
    z.x *= aspect;
    z /= zoom;
    z.x /= aspect;
    vec2 zoomTC = fract(z + m);
    vec4 baseTex = texture(samp, zoomTC);
    vec2 uv = zoomTC * 2.0 - 1.0;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * gSlow), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec3 baseCol = preBlendColor(zoomTC);

```

```

float seg = 4.0 + 2.0 * sin(gSlow * 0.33 + gAmp01 * 2.0);
vec2 kUV = reflectUV(zoomTC, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + 0.55 * sin(gSlow * 0.42 + gAmp01 * 3.0);
kUV = fractalFold(kUV, foldZoom, gDetail, m, aspect);
kUV = rotateUV(kUV, gSlow * 0.23 + gAmp01 * 1.2, m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(gSlow * 0.2) * (PI * gSlow), 5.0);
float period = log(base) * pingPong(gSlow * PI, 5.0);
float tz = gSlow * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + gFast * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * (0.0015 + 0.001 * sin(gFast * 1.3)) * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((zoomTC - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + gFast * 1.2));
ring = ring * pingPong(gSlow * PI, 5.0);
float pulse = 0.5 + 0.5 * sin(gFast * 2.0 + rD * 28.0 + k * 12.0);
pulse *= mix(0.4, 1.6, gAmp01);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
outCol *= mix(0.7, 1.5, gAmp01);
vec3 finalCore = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
vec2 center = m;
vec2 pSwirl = (zoomTC - center) * vec2(1.0, iResolution.y / iResolution.x);
float intensity = pingPong(gSlow, 10.0) * (0.3 + 1.7 * gAmp01);
float angleS = atan(pSwirl.y, pSwirl.x);
float radiusS = length(pSwirl);
float swirl = sin(gSlow * 0.5) * 0.5 + 0.5;
angleS += intensity * swirl * sin(radiusS * 10.0 + gFast);
vec2 qSwirl = vec2(cos(angleS), sin(angleS)) * radiusS;
vec2 uvSwirl = qSwirl * vec2(1.0, iResolution.x / iResolution.y) + center;
vec4 texSwirl = texture(samp, uvSwirl);
float fluctuation = sin(gFast * 2.0) * 0.5 + 0.5;
vec3 xorPalette = mix(vec3(1.0, 0.0, 0.0), vec3(0.0, 0.0, 1.0), fluctuation);
xorPalette = mix(xorPalette, neonPalette(gFast), 0.5);
vec4 fluctuatedColor = vec4(xorPalette, 1.0);
vec4 xorResult = xor_RGB(texSwirl, fluctuatedColor);
vec3 xorMix = mix(texSwirl.rgb, xorResult.rgb, 0.6);
float xorStrength = (0.25 + 0.35 * pulse) * vign * mix(0.4, 1.4, gAmp01);
xorStrength = clamp(xorStrength, 0.0, 1.0);
vec3 finalRGB = mix(finalCore, xorMix, xorStrength);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);

```

```

    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #418 - frac shader xor

File: frac_shader_xor.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
 - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation -
 Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
}

```

```

    }
    return p;
}

vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res){
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts*vec2(-1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts*vec2( 0.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts*vec2( 1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts*vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts*vec2( 1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts*vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts*vec2( 0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts*vec2( 1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0*s10 + s20 + 2.0*s01 + 4.0*s11 + 2.0*s21 + s02 + 2.0*s12 + s22) / 16.0;
}

vec3 softTone(vec3 c){
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299,0.587,0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 neonPalette(float t){
    vec3 pink  = vec3(1.0, 0.15, 0.75);
    vec3 blue  = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t*0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a*k1 + b*k2 + c*k3) * 1.05;
}

vec3 xorRGB_neon(vec3 baseCol, vec3 neonCol, float t){
    ivec3 b = ivec3(round(clamp(baseCol,0.0,1.0)*255.0));
    ivec3 n = ivec3(round(clamp(neonCol,0.0,1.0)*255.0));
    int mask = int(mod(floor(t*64.0), 256.0));
    for(int i=0;i<3;i++){
        int v = b[i] ^ (n[i] ^ mask);
        v = v & 255;
        b[i] = v;
    }
    vec3 xr = vec3(b)/255.0;
    return mix(baseCol, xr, 0.6);
}

```

```

}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}
vec3 samplePreColor(vec2 uv){
    return preBlendColor(uv);
}
void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec4 originalTex = texture(samp, tc);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p); if(q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18*sin(time_f*0.2);
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + time_f*0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap*1.045) / ar + m);
    vec2 u2 = fract((pwrap*0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = samplePreColor(u0 + off);
    vec3 gC = samplePreColor(u1);
    vec3 bC = samplePreColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD+1e-3)*9.5 + time_f*1.2));
    float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + rD*28.0 + k*12.0);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
    vec3 bloom = outCol*outCol*0.18 + pow(max(outCol-0.6,0.0), vec3(2.0))*0.12;
    outCol += bloom;
    float retain = 0.18;
    outCol = mix(outCol, baseCol, retain);
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    color = (0.5 * vec4(outCol,1.0)) + (0.5 * texture(samp, tc));
}

```



```

float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
color.rgb *= 1.0 + _ab * 0.6;
color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #419 - frac shader xor2

File: frac_shader_xor2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;

float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){

```

```

        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}

vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res){
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts*vec2(-1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts*vec2( 0.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts*vec2( 1.0,-1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts*vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts*vec2( 1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts*vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts*vec2( 0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts*vec2( 1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0*s10 + s20 + 2.0*s01 + 4.0*s11 + 2.0*s21 + s02 + 2.0*s12 + s22) / 16.0;
}

vec3 softTone(vec3 c){
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299,0.587,0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

mat2 rot(float a){ float s=sin(a), c=cos(a); return mat2(c,-s,s,c); }

void triPoints(out vec2 a, out vec2 b, out vec2 c){
    vec2 cen = vec2(0.5);
    float ang = time_f*0.25;
    float r = 0.42 + 0.03*sin(time_f*0.7);
    a = cen + (rot(ang) * vec2(0.0, r));
    b = cen + (rot(ang+2.0944) * vec2(0.0, r));
    c = cen + (rot(ang+4.1888) * vec2(0.0, r));
}

vec3 triColors(){
    vec3 c0 = vec3(1.0, 0.15, 0.75);
    vec3 c1 = vec3(0.10, 0.55, 1.0);
    vec3 c2 = vec3(0.10, 1.00, 0.45);
    float t = fract(time_f*0.08);
    vec3 s0 = mix(c0, c1, smoothstep(0.00,0.33,t));
    vec3 s1 = mix(c1, c2, smoothstep(0.33,0.66,t));
    vec3 s2 = mix(c2, c0, smoothstep(0.66,1.00,t));
    float a = step(t,0.33);
    float b = step(0.33,t)*step(t,0.66);
    float c = step(0.66,t);
    return normalize(a*s0 + b*s1 + c*s2) * 1.05;
}

```

```

vec3 triGradient(vec2 uv){
    vec2 A,B,C; triPoints(A,B,C);
    vec2 v0 = B - A, v1 = C - A, v2 = uv - A;
    float d00 = dot(v0,v0);
    float d01 = dot(v0,v1);
    float d11 = dot(v1,v1);
    float d20 = dot(v2,v0);
    float d21 = dot(v2,v1);
    float denom = max(d00*d11 - d01*d01, 1e-6);
    float w1 = (d11*d20 - d01*d21) / denom;
    float w2 = (d00*d21 - d01*d20) / denom;
    float w0 = 1.0 - w1 - w2;
    w0 = max(w0, 0.0); w1 = max(w1, 0.0); w2 = max(w2, 0.0);
    float sum = max(w0+w1+w2, 1e-6);
    w0 /= sum; w1 /= sum; w2 /= sum;
    vec3 c0 = vec3(1.0, 0.15, 0.75);
    vec3 c1 = vec3(0.10, 0.55, 1.0);
    vec3 c2 = vec3(0.10, 1.00, 0.45);
    float s = fract(time_f*0.12);
    c0 = mix(c0, c1, smoothstep(0.0,1.0,s));
    c1 = mix(c1, c2, smoothstep(0.0,1.0,s));
    c2 = mix(c2, c0, smoothstep(0.0,1.0,s));
    vec3 col = w0*c0 + w1*c1 + w2*c2;
    vec2 g = vec2(length(fwidth(uv)));
    vec3 col2 = w0*c0 + w1*c1 + w2*c2;
    return mix(col, col2, 0.5);
}

vec3 preBlendColor(vec2 uv){
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec3 gcol = triGradient(uv);
    vec3 grad = mix(tex, tex*gcol, 0.55);
    grad = softTone(grad);
    return grad;
}

vec3 gradientModulateTexture(vec2 uv){
    vec3 tex = texture(samp, uv).rgb;
    vec3 gcol = triGradient(uv);
    vec3 mixed = mix(tex, tex*gcol, 0.6);
    vec2 ts = vec2(max(1.0/iResolution.x, 1.0/iResolution.y));
    vec3 nb = 0.25*(textureGrad(samp, uv+ts, dFdx(uv), dFdy(uv)).rgb +
                    textureGrad(samp, uv-ts, dFdx(uv), dFdy(uv)).rgb +
                    textureGrad(samp, uv+ts*vec2(1,-1), dFdx(uv), dFdy(uv)).rgb +
                    textureGrad(samp, uv+ts*vec2(-1,1), dFdx(uv), dFdy(uv)).rgb);
    return mix(mixed, nb, 0.08);
}

void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p); if(q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18*sin(time_f*0.2);
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);

```

```

float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + time_f*0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap*1.045) / ar + m);
vec2 u2 = fract((pwrap*0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD+1e-3)*9.5 + time_f*1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + rD*28.0 + k*12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = outCol*outCol*0.18 + pow(max(outCol-0.6,0.0), vec3(2.0))*0.12;
outCol += bloom;
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 texCol = gradientModulateTexture(tc);
vec3 gradWithTex = mix(texCol, baseCol, 0.5);
vec3 finalCol = mix(outCol, gradWithTex, 0.5);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalCol *= 1.0 + _ab * 0.6;
finalCol = mix(finalCol, finalCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalCol, 1.0);
}

```

Shader #523 - glow

File: glow.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {

```

```

        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}

void main(void) {
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    uv = uv * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float dist = sin(length(uv) * time_f);
    float glowFrequency = 15.0 + amp_high * 20.0;
    float glowSpeed = 10.0;
    float glow = sin(dist * glowFrequency - time_f * glowSpeed) * 0.5 + 0.5;
    float expansion = abs(sin(time_f * (0.5 + amp_low * 2.0)));
    float beam = smoothstep(0.0, 1.0, expansion - dist);
    vec3 beamColor = vec3(0.0, 0.8, 1.0);
    vec3 glowColor = vec3(1.0, 1.0, 1.0);
    vec3 finalColor = mix(beamColor, glowColor, glow) * beam;
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalColor *= 1.0 + _ab * 0.6;
    finalColor = mix(finalColor, finalColor * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalColor, 1.0);
    color = xor_RGB(color, texture(samp, tc));
}

```

Shader #565 - gxor

File: gxor.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec4 texColor = texture(samp, uv);
    float xor_effect_r = mod((texColor.r * 255.0) + sin(uv.x * 20.0 + time_f) * 255.0, 256.0) / 255.0;
    float xor_effect_g = mod((texColor.g * 255.0) + cos(uv.y * 20.0 - time_f) * 255.0, 256.0) / 255.0;
    float xor_effect_b = mod((texColor.b * 255.0) + sin((uv.x + uv.y) * 20.0 + time_f) * 255.0, 256.0) /
        ↪ 255.0;
    vec3 finalColor = mix(texColor.rgb, vec3(xor_effect_r, xor_effect_g, xor_effect_b), 0.5);
    color = vec4(finalColor, texColor.a);
}

```

Shader #566 - gxor2

File: gxor2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec4 texColor = texture(samp, uv);
    float distortion = sin(uv.x * 50.0 + time_f * 2.0) * cos(uv.y * 50.0 - time_f * 2.0);
    float xor_effect_r = mod((texColor.r * 255.0) + distortion * 255.0, 256.0) / 255.0;
    float xor_effect_g = mod((texColor.g * 255.0) + distortion * 255.0, 256.0) / 255.0;
    float xor_effect_b = mod((texColor.b * 255.0) + distortion * 255.0, 256.0) / 255.0;
    vec3 rainbow = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 10.0),
        0.5 + 0.5 * sin(time_f + uv.y * 10.0 + 2.0),
        0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 10.0 + 4.0)
    );
    vec3 finalColor = mix(vec3(xor_effect_r, xor_effect_g, xor_effect_b), rainbow, 0.5);
    color = vec4(finalColor, texColor.a);
}
```

Shader #628 - matrix mul

File: matrix_mul.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
}
```

```

return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    mat4 matrix = mat4(alpha, 0.1, 0.2, 0.3,
                        0.4, -alpha, 0.5, 0.6,
                        0.7, 0.8, alpha, 0.9,
                        0, 0, 0, -alpha);
    color = color * (0.5 * matrix);
    color = xor_RGB(color, source);
    vec4 color2 = texture(samp, tc+0.01);
    vec4 color3 = texture(samp, tc-0.01);
    color = (0.5 * color) + (0.3 * color2) + (0.3 * color3);
}

```

Shader #629 - matrix mul2

File: matrix_mul2.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    mat4 matrix = mat4(alpha * tc[0], 0.1, 0.2, 0.3,

```

```

        0.4, -alpha, 0.5, 0.6,
        0.7, 0.8, alpha * tc[1], 0.9,
        0, 0, 0, -alpha);
color = color * (0.5 * matrix);
color = xor_RGB(color, source);
vec4 color2 = texture(samp, tc+0.01);
vec4 color3 = texture(samp, tc-0.01);
color = (0.5 * color) + (0.3 * color2) + (0.3 * color3);
}

```

Shader #630 - matrix mul3

File: matrix_mul3.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 tcolor = color;
    ivec4 source = ivec4(color * 255) / 2;
    mat4 matrix = mat4(
        tc[0], tc[1], tc[1], tc[0],
        tc[1], tc[0], tc[0], tc[1],
        tc[0], tc[0], tc[1], tc[0],
        1,1,1,1);
    color = color * matrix;
    color = xor_RGB(color, source);
}

```



```
    color = (0.5 * tcolor) + (0.5 * color);  
}
```

Shader #631 - matrix mul4

File: matrix_mul4.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```
#version 330  
in vec2 tc;  
out vec4 color;  
uniform float alpha_r;  
uniform float alpha_g;  
uniform float alpha_b;  
uniform float alpha;  
uniform vec4 optx;  
uniform vec4 random_var;  
uniform float alpha_value;  
uniform mat4 mv_matrix;  
uniform mat4 proj_matrix;  
uniform sampler2D samp;  
uniform float value_alpha_r, value_alpha_g, value_alpha_b;  
uniform float index_value;  
uniform float time_f;  
uniform float restore_black;  
vec4 xor_RGB(vec4 icolor, ivec4 isource) {  
    ivec3 int_color;  
    for(int i = 0; i < 3; ++i) {  
        int_color[i] = int(255 * icolor[i]);  
        int_color[i] = int_color[i]^isource[i];  
        if(int_color[i] > 255)  
            int_color[i] = int_color[i]%255;  
        icolor[i] = float(int_color[i])/255;  
    }  
    icolor.a = 1.0;  
    return icolor;  
}  
void main(void)  
{  
    color = texture(samp, tc);  
    mat4 matrix = mat4(1.0, 0.0, 0.0, color[2],  
                      0.0, 1.0, 0.0, color[1],  
                      0.0, 0.0, 1.0, color[0],  
                      0.0, 0.0, 0.0, 1.0);  
    color = color * matrix;  
}
```

Shader #726 - new xor scale strobe

File: new_xor_scale_strobe.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float alpha;
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {
    ivec4 isource = ivec4(isourcex * 255);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
void main() {
    vec4 col2 = texture(samp, tc);
    float t = mod(time_f, 10) * mod(alpha, 1.0);
    col2 *= t;
    vec4 col = xor_RGB(texture(samp, tc), col2);
    color = col;
}

```

Shader #727 - nl

File: nl.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
 Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform vec2 iResolution;
uniform float time_f;
uniform sampler2D samp;
float rand(vec2 co){
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
vec3 smoothBaseColor(float t) {

```

```

    vec3 colors[3] = vec3[](vec3(0.1, 0.1, 0.2), vec3(0.05, 0.05, 0.15), vec3(0.15, 0.15, 0.25));
    float index = fract(t * 0.1);
    float transition = sin(index * 3.14159265 * 2.0) * 0.5 + 0.5;
    int i = int(fract(t * 0.05) * 3.0);
    int next = (i + 1) % 3;
    return mix(colors[i], colors[next], transition);
}

void main(void) {
    float aspectRatio = iResolution.x / iResolution.y;
    vec2 adjustedTC = tc;
    adjustedTC.y /= aspectRatio;
    vec2 center = vec2(0.5, 0.5);
    vec2 toCenter = adjustedTC - center;
    float dist = length(toCenter) * aspectRatio;
    vec3 baseColor = smoothBaseColor(time_f);
    float wave1 = sin(tc.x * 3.0 + time_f * 1.5) * 0.5 + 0.5;
    float wave2 = sin(tc.y * 2.0 + time_f * 2.0 + cos(tc.x * 5.0)) * 0.5 + 0.5;
    float auroraFactor = wave1 * wave2;
    vec3 auroraColor = mix(vec3(0.0, 1.0, 0.0), vec3(0.0, 0.5, 1.0), wave2);
    vec3 finalColor = mix(baseColor, auroraColor, auroraFactor * 0.6);
    vec4 ocolor = texture(samp, tc);
    color = vec4(finalColor, 1.0);
    color *= time_f;
    color = mix(color, texture(samp, tc), 0.5);
    color = xor_RGB(color, ivec4(ocolor * 255));
}

```

Shader #767 - plasma xor

File: plasma_xor.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
float PI = 3.1415926535897932384626433832795;
uniform sampler2D samp;
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {
    ivec4 isource = ivec4(isourcex * 255);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}

void main() {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float plasma = 0.0;
    plasma += sin((uv.x + time_f) * 5.0);
}

```

```

    plasma += sin((uv.y + time_f) * 5.0);
    plasma += sin((uv.x + uv.y + time_f) * 5.0);
    plasma += cos(length(uv + time_f) * 10.0);
    plasma *= 0.25;
    vec3 col;
    col.r = cos(plasma * PI + time_f * 0.2) * 0.5 + 0.5;
    col.g = sin(plasma * PI + time_f * 0.2) * 0.5 + 0.5;
    col.b = sin(plasma * PI + time_f * 0.4) * 0.5 + 0.5;
    color = vec4(col, 1.0);
    color = xor_RGB(color, texture(samp, tc));
}

```

Shader #787 - psych pattern

File: psych_pattern.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {
    ivec4 isource = ivec4(isourcex * 255);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
float kaleido(vec2 uv) {
    uv = uv * 2.0 - 1.0;
    float angle = atan(uv.y, uv.x) + time_f;
    float radius = length(uv);
    angle = mod(angle, 3.14159 / 4.0) * 8.0;
    return abs(cos(angle * radius * 10.0 - time_f * 4.0));
}
vec3 colorCycle(float t) {
    return vec3(
        sin(t * 2.0 + 0.0) * 0.5 + 0.5,
        sin(t * 2.0 + 2.09439) * 0.5 + 0.5,
        sin(t * 2.0 + 4.18879) * 0.5 + 0.5
    );
}
void main() {
    vec2 uv = tc - 0.5;
    uv.x *= iResolution.x / iResolution.y;
    float pattern = kaleido(uv);
    vec3 col = colorCycle(pattern + time_f);
    color = vec4(col, 1.0);
}

```

```
    color = xor_RGB(color, texture(samp, tc));  
}
```

Shader #790 - psych xor

File: psych_xor.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core  
in vec2 tc;  
out vec4 color;  
uniform float time_f;  
uniform vec2 iResolution;  
uniform sampler2D samp;  
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {  
    ivec4 isource = ivec4(isourcex * 255);  
    ivec3 int_color;  
    for(int i = 0; i < 3; ++i) {  
        int_color[i] = int(255 * icolor[i]);  
        int_color[i] ^= isource[i];  
        if(int_color[i] > 255)  
            int_color[i] %= 255;  
        icolor[i] = float(int_color[i]) / 255.0;  
    }  
    icolor.a = 1.0;  
    return icolor;  
}  
void main() {  
    vec2 uv = (tc - 0.5) * 2.0;  
    uv.x *= iResolution.x / iResolution.y;  
    float time = time_f * 0.5;  
    float sine1 = sin(uv.x * 10.0 + time) * cos(uv.y * 10.0 + time);  
    float sine2 = sin(uv.y * 15.0 - time) * cos(uv.x * 15.0 - time);  
    float sine3 = sin(sqrt(uv.x * uv.x + uv.y * uv.y) * 20.0 + time);  
    float pattern = sine1 + sine2 + sine3;  
    float colorIntensity = pattern * 0.5 + 0.5;  
    vec3 psychedelicColor = vec3(  
        sin(colorIntensity * 6.28318 + 0.0) * 0.5 + 0.5,  
        sin(colorIntensity * 6.28318 + 2.09439) * 0.5 + 0.5,  
        sin(colorIntensity * 6.28318 + 4.18879) * 0.5 + 0.5  
    );  
    color = vec4(psychedelicColor, 1.0);  
    color = xor_RGB(color, texture(samp, tc));  
}
```

Shader #818 - rainbow expand noise xor

File: rainbow_expand_noise_xor.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
```

```

in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec3 source) {
    ivec3 int_color;
    ivec3 isource = ivec3(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
        dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0) * 0.5 + 0.5;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float rotation_period = 3.0;
    float rotation_angle = mod(time_f, rotation_period * 2.0) < rotation_period ? time_t : -time_t;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + rotation_angle * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec4 blended_color = xor_RGB(original_color, rainbow_color);
    color = sin(time_t * blended_color);
    color.a = 1.0;
}

```

Shader #860 - react19

File: react19.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);
    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
void main() {
    vec2 uv = tc / iResolution;
    float amplitude = sin(time_f) * 0.5 + 0.5;
    vec3 gradientColor = mix(vec3(0.0, 0.0, 1.0), vec3(1.0, 0.0, 0.0), amplitude);
```

```

    vec4 texColor = blur(samp, tc, iResolution);
    vec3 finalColor = mix(texColor.rgb, gradientColor, 0.5 * sin(time_f * 2.0 + uv.y * 3.14159) + 0.5);
    color = vec4(finalColor, texColor.a);
    float time_t = pingPong(time_f, 10.0);
    color = xor_RGB(color, blur(samp, tc, iResolution) * time_t);
    if(color.rgb == vec3(color.r)) {
        color.rgb += vec3(0.05, 0.05, 0.05) * sin(time_f * 3.0);
    }
}

```

Shader #879 - reverse xor

File: reverse_xor.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Bitwise XOR on integer color channels - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}

void main(void)
{
    color = texture(samp, tc);
    ivec4 source =ivec4(color * 255);
    vec4 c = vec4(0.2, 0.5, 0.8, 1);
    vec4 cval;
    color += 0.03;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[3-i-1];
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #884 - rgb time xor

File: rgb_time_xor.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = (color[0]*(0.3214*timeval*alpha_b))*0.8;
    color[1] = (color[1]*(0.5321*timeval*alpha_g))*0.8;
    color[2] = (color[2]*(0.7342*timeval*alpha_r))*0.8;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int(tc[0])^int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #909 - sac geo xor

File: sac-geo-xor.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
```

```

out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {
    ivec4 isource = ivec4(isourcex * 255.0);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255.0 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
return icolor;
}
vec3 sacredGeometry(vec2 uv, float time) {
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
    float frequency = 5.0;
    float amplitude = 0.5;
    float sacredPattern = sin(radius * frequency + time) * amplitude + 0.5;
    float hue = fract(sacredPattern + time * 0.1);
    float saturation = 1.0;
    float value = 1.0;
    return hsv2rgb(vec3(hue, saturation, value));
}
void main() {
    vec2 uv = tc * 2.0 - 1.0;
    uv *= vec2(iResolution.x / iResolution.y, 1.0);
    float time = time_f * 0.1;
    vec3 sacredColor = sacredGeometry(uv, time);
    vec2 complexCoord = uv * vec2(cos(time), sin(time));
    vec3 complexColor = sacredGeometry(complexCoord, time);
    vec4 textureColor = texture(samp, tc);
    vec3 combinedColor = mix(sacredColor, complexColor, 0.5);
    color = vec4(mix(textureColor.rgb, combinedColor, 0.5), 1.0);
    color = xor_RGB(color, textureColor);
}

```

Shader #919 - self alphablend scale xor

File: self_alphablend_scale_xor.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;

```

```

uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = color;
    ivec4 color_source = ivec4(color * 255);
    color = color*alpha;
    ivec4 color_i = ivec4(color * 255);
    for(int i = 0; i < 3; ++i) {
        if(color_i[i] >= 255)
            color_i[i] = color_i[i]%255;
        if(color_source[i] >= 255)
            color_source[i] = color_source[i]%255;
        color_i[i] = color_i[i] ^ color_source[i];
        color[i] = float(color_i[i])/255;
    }
    for(int i = 0; i < 3; ++i)
        if(color[i] < 0.2) color[i] = color2[i];
}

```

Shader #1013 - time frac xor

File: time_frac_xor.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
uniform vec4 inc_value;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;

```

```

    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(color * 255);
    color = (0.5 * color) + (0.5 * fract(inc_value/255 * time_f));
    color = xor_RGB(color, source);
}

```

Shader #1014 - time xor

File: time_xor.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = timeval*0.1*(color[0]*alpha_r);
    color[1] = timeval*0.2*(color[1]*alpha_r);
    color[2] = timeval*0.3*(color[2]*alpha_r);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)

```

```

        int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #1067 - xor increase

File: xor_increase.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Bitwise XOR on integer color channels - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
in vec2 iResolution_;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}
void main(void)
{
    color = texture(samp, tc);
    vec2 pos = gl_FragCoord.xy / iResolution_.xy * timeval;
    float r = random(pos);
    ivec4 source = ivec4(color * 255);
    vec4 c = vec4(0.2, 0.5, 0.8, 1);
    vec4 cval;
    vec4 a = inc_valuex/255 * 2.0;
    color = color * a * 6.0;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

```
}
```

Shader #1068 - xor positional offset

File: xor_positional_offset.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Bitwise XOR on integer color channels - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
in vec2 iResolution_;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}
void main(void)
{
    color = texture(samp, tc);
    vec2 pos = gl_FragCoord.xy / iResolution_.xy * inc_valuex.xy/6;
    float r = random(pos);
    ivec4 source =ivec4(color * 255);
    vec4 a = vec4(0.2, 0.5, 0.8, 1.0);
    color = color + a;
    color = color * vec4(pos+alpha, 1, 1);
    color = color * 0.08;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #1069 - xor rgb

File: xor_rgb.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec4 tcolor = texture(samp, tc);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = xor_RGB(tcolor, tcolor * time_t);
}
```

Shader #1070 - xor rgb dark

File: xor_rgb_dark.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
}
```

```

    icolor.a = 1.0;
return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = xor_RGB(sin(tcolor * time_t), tcolor * time_t);
}

```

Shader #1071 - xor rgb dark 2

File: xor_rgb_dark_2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)

```



```

        int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = xor_RGB(sin(tcolor * time_t), cos(tcolor * time_t));
}

```

Shader #1072 - xor rgb dark 3

File: xor_rgb_dark_3.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {

```

```

        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = xor_RGB(sin(tcolor * time_t), texture(samp, tc));
}

```

Shader #1073 - xor rgb dark 4

File: xor_rgb_dark_4.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {

```

```

    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    float time_m = pingPong(time_f, 5.0) + 2.0;
    color = xor_RGB(sin(tcolor * time_t), tcolor * time_m);
}

```

Shader #1074 - xor rgb dark rainbow

File: xor_rgb_dark_rainbow.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;

```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float t = pingPong(time_f, 20) + 1.0;
    float angle = atan(uv.y, uv.x) + t * 20.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec3 blended_color = mix(tcolor.rgb, rainbow_color, 0.5);
}

```

```

    vec4 xorColor = xor_RGB(sin(vec4(blended_color,1) * time_t), cos(vec4(blended_color,1) * time_t));
    color = vec4(xorColor.rgb, tcolor.a);
}

```

Shader #1075 - xor rgb new

File: xor_rgb_new.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void) {
    vec4 tex = texture(samp, tc);
    float lum = dot(tex.rgb, vec3(0.299, 0.587, 0.114));
    float tHue = time_f * 0.15;
    float hueBase = fract(lum * 0.8 + tHue);
    vec3 neon1 = hsv2rgb(vec3(hueBase, 1.0, 1.0));
    vec3 neon2 = hsv2rgb(vec3(fract(hueBase + 0.33), 1.0, 1.0));
    float wave = pingPong(time_f * 0.25, 1.0);
    vec3 neon = mix(neon1, neon2, wave);
    float strength = 0.3 + 0.7 * wave;
    vec3 mixed = mix(tex.rgb, neon, strength);
    mixed = pow(mixed, vec3(0.8));
    mixed = clamp(mixed, 0.0, 1.0);
    color = vec4(mixed, tex.a);
}

```

Shader #1076 - xor rgb rainbow 1

File: xor_rgb_rainbow_1.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);
    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
```

```

vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}

vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}

vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    float time_m = pingPong(time_f, 5.0) + 2.0;
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float time_wave = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_wave * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_wave) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_wave * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_wave * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = tcolor;
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = xor_RGB(sin(vec4(blended_color, 1) * time_t), original_color * time_m);
}

```

Shader #1078 - xor rgb rainbow swirl limit

File: xor_rgb_rainbow_swirl_limit.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
    }
}

```

```

        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float t = pingPong(time_f, 15) + 1.0;
    float wave = sin(uv.x * 10.0 + t * 2.0) * 0.1;
    float expand = 0.5 + 0.5 * sin(t * 2.0);
    vec2 spiral_uv = uv * expand + vec2(cos(t), sin(t)) * 0.2;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec3 blended_color = mix(tcolor.rgb, rainbow_color, 0.5);
    color = xor_RGB(sin(vec4(blended_color, 1) * time_t), tcolor * time_t);
}

```


Shader #1082 - xor sine swirl

File: xor_sine_swirl.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rainbow(float t) {
    float hue = mod(t, 1.0) * 6.0;
    float c = 1.0;
    float x = 1.0 - abs(mod(hue, 2.0) - 1.0);
    vec3 rgb = (hue < 1.0) ? vec3(c, x, 0.0) :
                (hue < 2.0) ? vec3(x, c, 0.0) :
                (hue < 3.0) ? vec3(0.0, c, x) :
                (hue < 4.0) ? vec3(0.0, x, c) :
                (hue < 5.0) ? vec3(x, 0.0, c) :
                vec3(c, 0.0, x);
    return rgb;
}
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 dir = tc - center;
    float dist = length(dir);
    float angle = atan(dir.y, dir.x);
    float frequency = 1.0;
    float amplitude = 3.0;
    float spiralTime = sin(time_f * frequency) * amplitude;
    float scale = 1.0;
    float gradient_pos = mod(dist * scale + (angle + spiralTime) / (2.0 * 3.14159), 1.0);
    vec3 color_gradient = rainbow(gradient_pos);
    vec4 ctx = texture(samp, tc);
    float time_t = mod(time_f, 10.0) + 2.0;
    color = xor_RGB(ctx, vec4(color_gradient, 1.0));
}
```

Shader #1083 - XorKern

File: XorKern.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns. 7x7

Gaussian Kernel Applies the Contrast Boost logic: $\text{midpoint} + (\text{val} - \text{midpoint}) * \text{boost}$ 128 in 8-bit is 0.5 in float

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[49];
    kernel[0] = 0.5; kernel[1] = 1.0; kernel[2] = 2.0; kernel[3] = 2.5; kernel[4] = 2.0; kernel[5] =
    ↪ 1.0; kernel[6] = 0.5;
    kernel[7] = 1.0; kernel[8] = 2.0; kernel[9] = 3.0; kernel[10] = 3.5; kernel[11] = 3.0; kernel[12] =
    ↪ 2.0; kernel[13] = 1.0;
    kernel[14] = 2.0; kernel[15] = 3.0; kernel[16] = 4.0; kernel[17] = 4.5; kernel[18] = 4.0; kernel[19]
    ↪ = 3.0; kernel[20] = 2.0;
    kernel[21] = 2.5; kernel[22] = 3.5; kernel[23] = 4.5; kernel[24] = 5.0; kernel[25] = 4.5; kernel[26]
    ↪ = 3.5; kernel[27] = 2.5;
    kernel[28] = 2.0; kernel[29] = 3.0; kernel[30] = 4.0; kernel[31] = 4.5; kernel[32] = 4.0; kernel[33]
    ↪ = 3.0; kernel[34] = 2.0;
    kernel[35] = 1.0; kernel[36] = 2.0; kernel[37] = 3.0; kernel[38] = 3.5; kernel[39] = 3.0; kernel[40]
    ↪ = 2.0; kernel[41] = 1.0;
    kernel[42] = 0.5; kernel[43] = 1.0; kernel[44] = 2.0; kernel[45] = 2.5; kernel[46] = 2.0; kernel[47]
    ↪ = 1.0; kernel[48] = 0.5;
    float kernelSum = 272.0;
    for (int x = -3; x <= 3; ++x) {
        for (int y = -3; y <= 3; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 3) * 7 + (x + 3)];
        }
    }
    return result / kernelSum;
}
vec3 applyContrast(vec3 col) {
    float boost = 1.8;
    float midpoint = 0.5;
    vec3 result;
    result.r = midpoint + (col.r - midpoint) * boost;
    result.g = midpoint + (col.g - midpoint) * boost;
    result.b = midpoint + (col.b - midpoint) * boost;
    return clamp(result, 0.0, 1.0);
}
vec3 cudaXorLogic(vec3 current, vec3 averaged, float sum_simulator) {
    ivec3 iCur = ivec3(current * 255.0);
    ivec3 iSum = ivec3(averaged * 255.0 * sum_simulator);
    ivec3 iXor;
    iXor.r = iCur.r ^ int((1 + iSum.r) % 255);
    iXor.g = iCur.g ^ int((1 + iSum.g) % 255);
    iXor.b = iCur.b ^ int((1 + iSum.b) % 255);
    return vec3(iXor % 255) / 255.0;
}
void main(void) {
    vec4 sharpColor = texture(samp, tc);
```

```

    vec4 blurredColor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec3 xorResult = cudaXorLogic(sharpColor.rgb, blurredColor.rgb, time_t);
    vec3 blendResult = (xorResult * 0.5) + (blurredColor.rgb * 0.5);
    color = vec4(applyContrast(blendResult), 1.0);
}

```

Shader #1084 - XorKernBrighter

File: XorKernBrighter.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns. 7x7 Gaussian Kernel Applies contrast boost and an overall brightness multiplier Adjust this value. 1.0 = original, > 1.0 = brighter.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[49];
    kernel[0] = 0.5; kernel[1] = 1.0; kernel[2] = 2.0; kernel[3] = 2.5; kernel[4] = 2.0; kernel[5] =
    ↪ 1.0; kernel[6] = 0.5;
    kernel[7] = 1.0; kernel[8] = 2.0; kernel[9] = 3.0; kernel[10] = 3.5; kernel[11] = 3.0; kernel[12] =
    ↪ 2.0; kernel[13] = 1.0;
    kernel[14] = 2.0; kernel[15] = 3.0; kernel[16] = 4.0; kernel[17] = 4.5; kernel[18] = 4.0; kernel[19]
    ↪ = 3.0; kernel[20] = 2.0;
    kernel[21] = 2.5; kernel[22] = 3.5; kernel[23] = 4.5; kernel[24] = 5.0; kernel[25] = 4.5; kernel[26]
    ↪ = 3.5; kernel[27] = 2.5;
    kernel[28] = 2.0; kernel[29] = 3.0; kernel[30] = 4.0; kernel[31] = 4.5; kernel[32] = 4.0; kernel[33]
    ↪ = 3.0; kernel[34] = 2.0;
    kernel[35] = 1.0; kernel[36] = 2.0; kernel[37] = 3.0; kernel[38] = 3.5; kernel[39] = 3.0; kernel[40]
    ↪ = 2.0; kernel[41] = 1.0;
    kernel[42] = 0.5; kernel[43] = 1.0; kernel[44] = 2.0; kernel[45] = 2.5; kernel[46] = 2.0; kernel[47]
    ↪ = 1.0; kernel[48] = 0.5;
    float kernelSum = 272.0;
    for (int x = -3; x <= 3; ++x) {
        for (int y = -3; y <= 3; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 3) * 7 + (x + 3)];
        }
    }
    return result / kernelSum;
}
vec3 applyContrastAndBrightness(vec3 col) {
    float contrastBoost = 1.8;
    float midpoint = 0.5;
    float brightnessScale = 1.5;
    vec3 result;
    result.r = midpoint + (col.r - midpoint) * contrastBoost;

```

```

    result.g = midpoint + (col.g - midpoint) * contrastBoost;
    result.b = midpoint + (col.b - midpoint) * contrastBoost;
    result *= brightnessScale;
    return clamp(result, 0.0, 1.0);
}

vec3 cudaXorLogic(vec3 current, vec3 averaged, float sum_simulator) {
    ivec3 iCur = ivec3(current * 255.0);
    ivec3 iSum = ivec3(averaged * 255.0 * sum_simulator);
    ivec3 iXor;
    iXor.r = iCur.r ^ int((1 + iSum.r) % 255);
    iXor.g = iCur.g ^ int((1 + iSum.g) % 255);
    iXor.b = iCur.b ^ int((1 + iSum.b) % 255);
    return vec3(iXor % 255) / 255.0;
}

void main(void) {
    vec4 sharpColor = texture(samp, tc);
    vec4 blurredColor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec3 xorResult = cudaXorLogic(sharpColor.rgb, blurredColor.rgb, time_t);
    vec3 blendResult = (xorResult * 0.5) + (blurredColor.rgb * 0.5);
    color = vec4(applyContrastAndBrightness(blendResult), 1.0);
}

```

Shader #1086 - xorsheet

File: xorsheet.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 color_blend(vec4 color) {
    vec4 color2 = color;
    ivec4 color_source = ivec4(color * 255);
    color = color*alpha;
    ivec4 colori = ivec4(color * 255);
    for(int i = 0; i < 3; ++i) {
        if(colori[i] >= 255)
            colori[i] = colori[i]%255;
        if(color_source[i] >= 255)
            color_source[i] = color_source[i]%255;
        colori[i] = colori[i] ^ color_source[i];
    }
}

```

```

        color[i] = float(colori[i])/255;
    }
    for(int i = 0; i < 3; ++i)
        if(color[i] < 0.2) color[i] = color2[i];
    return color;
}
void main(void)
{
    color = texture(samp, tc);
    vec2 cord1 = vec2(tc[0]/2, tc[0]/2);
    vec2 cord2 = vec2(tc[0]/4, tc[0]/4);
    vec2 cord3 = vec2(tc[0]/8, tc[0]/8);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec2 p = gl_FragCoord.xy / 32;
    vec2 d = fract(p);
    color[1] = (color[1]+col1[1])*(d[0]+d[1]);
    color[2] = (color[2]+col2[2])*(d[0]+d[1]);
    color = color_blend(color);
}

```

Shader #1087 - xorsheet 2

File: xorsheet_2.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main(void)

```

```

{
    color = texture(samp, tc);
    ivec4 source = ivec4(color * 255);
    vec2 start = gl_FragCoord.xy / 64;
    vec2 val = fract(start);
    for(int i = 0; i < 3; ++i) {
        color[i] += color[i]*(val[0]+val[1]);
    }
    color = xor_RGB(color, source);
}

```

Shader #1088 - xorstroke

File: xorstroke.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec3 colorval;
    colorval[0] = (source[0] * alpha_r)/255;
    colorval[1] = (source[1] * alpha_g)/255;
    colorval[2] = (source[2] * alpha_b)/255;
    color[0] = colorval[0] * alpha_r;
    color[1] = colorval[1] * alpha_g;
    color[2] = colorval[2] * alpha_b;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int(tc[0])^int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Trail & Motion (51 shaders)

Shader #247 - echo pingpong

File: echo-pingpong.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void)
{
    float time_x = pingPong(time_f, 2.0) + 1.0;
    float time_y = pingPong(time_f, 4.0) + 1.0;
    float time_z = pingPong(time_f, 8.0) + 1.0;
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / time_x);
    vec4 color3 = texture(samp, tc / time_y);
    vec4 color4 = texture(samp, tc / time_z);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #248 - echo pingpong c

File: echo-pingpong_c.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
```

```

uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void)
{
    float time_x = pingPong(time_f, 2.0) + 1.0;
    float time_y = pingPong(time_f, 4.0) + 1.0;
    float time_z = pingPong(time_f, 8.0) + 1.0;
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / time_x);
    vec4 color3 = texture(samp, tc / time_y);
    vec4 color4 = texture(samp, tc / time_z);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    float time_q = pingPong(time_f, 25.0);
    color = cos(sin(length(time_q) * color));
}

```

Shader #249 - echo

File: echo.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc / 4);
    vec4 color4 = texture(samp, tc / 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```

Shader #250 - echo1

File: echo1.glsl

Samples the texture at multiple coordinates and composites the results.

Technique: Direct texture color manipulation with arithmetic operations

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
void main(void) {
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #251 - echo2

File: echo2.glsl

Samples the texture at multiple coordinates and composites the results.

Technique: Direct texture color manipulation with arithmetic operations

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
void main(void) {
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 4);
    vec4 color3 = texture(samp, tc/ 8);
    vec4 color4 = texture(samp, tc/ 12);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #252 - echo3

File: echo3.glsl

Samples the texture at multiple coordinates and composites the results.

Technique: Direct texture color manipulation with arithmetic operations

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
void main(void) {
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc/3);
    vec4 color3 = texture(samp, tc/5);
    vec4 color4 = texture(samp, tc/9);
```

```
    vec4 color5 = texture(samp, tc/15);  
    color = (color * 0.3) + (color2 * 0.3) + (color3 * 0.3) + (color4 * 0.3) + (color5 * 0.3);  
}
```

Shader #253 - echo 2

File: echo_2.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330  
in vec2 tc;  
out vec4 color;  
uniform float alpha_r;  
uniform float alpha_g;  
uniform float alpha_b;  
uniform float alpha;  
uniform vec4 optx;  
uniform vec4 random_var;  
uniform float alpha_value;  
uniform mat4 mv_matrix;  
uniform mat4 proj_matrix;  
uniform sampler2D samp;  
uniform float value_alpha_r, value_alpha_g, value_alpha_b;  
uniform float index_value;  
uniform float time_f;  
uniform float restore_black;  
void main(void)  
{  
    color = texture(samp, tc);  
    vec4 color2 = texture(samp, tc / 4);  
    vec4 color3 = texture(samp, tc/ 8);  
    vec4 color4 = texture(samp, tc/ 12);  
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;  
}
```

Shader #254 - echo 3

File: echo_3.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330  
in vec2 tc;  
out vec4 color;  
uniform float alpha_r;  
uniform float alpha_g;  
uniform float alpha_b;  
uniform float alpha;  
uniform vec4 optx;  
uniform vec4 random_var;  
uniform float alpha_value;  
uniform mat4 mv_matrix;  
uniform mat4 proj_matrix;
```

```

uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc/3);
    vec4 color3 = texture(samp, tc/5);
    vec4 color4 = texture(samp, tc/9);
    vec4 color5 = texture(samp, tc/15);
    color = (color * 0.3) + (color2 * 0.3) + (color3 * 0.3) + (color4 * 0.3) + (color5 * 0.3);
}

```

Shader #255 - echo 4

File: echo_4.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc/4);
    vec4 color3 = texture(samp, tc/7);
    vec4 color4 = texture(samp, tc/11);
    vec4 color5 = texture(samp, tc/18);
    color = (color * 0.3) + (color2 * 0.3) + (color3 * 0.3) + (color4 * 0.3) + (color5 * 0.3);
}

```

Shader #256 - echo 4 mix

File: echo_4_mix.glsl

Samples the texture at multiple coordinates and composites the results.

Technique: Direct texture color manipulation with arithmetic operations

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
void main(void) {
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color[0] = (0.4 * color[2]) + (0.4 * color2[1]) + (0.4 * color3[1]) + (0.4 * color4[0]);
    color[1] = (0.4 * color[1]) + (0.4 * color2[1]) + (0.4 * color3[2]) + (0.4 * color4[0]);
    color[2] = (0.4 * color[0]) + (0.4 * color2[2]) + (0.4 * color3[2]) + (0.4 * color4[1]);
}

```

Shader #257 - echo 4 rev

File: echo_4_rev.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 reverse_vec(vec4 inputv) {
    vec4 rev;
    rev[0] = inputv[2];
    rev[1] = inputv[1];
    rev[2] = inputv[0];
    return rev;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc/4);
    vec4 color3 = texture(samp, tc/7);
    vec4 color4 = texture(samp, tc/11);
    vec4 color5 = texture(samp, tc/18);
    color = (color * 0.3) + (reverse_vec(color2) * 0.3) + (color3 * 0.3) + (reverse_vec(color4) * 0.3) +
    ↪ (color5 * 0.3);
}

```

Shader #258 - echo 5 mix rev

File: echo_5_mix_rev.glsl

Samples the texture at multiple coordinates and composites the results.

Technique: Direct texture color manipulation with arithmetic operations

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
void main(void) {
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color[2] = (0.4 * color[2]) + (0.4 * color2[1]) + (0.4 * color3[1]) + (0.4 * color4[0]);
    color[1] = (0.4 * color[1]) + (0.4 * color2[1]) + (0.4 * color3[2]) + (0.4 * color4[0]);
    color[0] = (0.4 * color[0]) + (0.4 * color2[2]) + (0.4 * color3[2]) + (0.4 * color4[1]);
}
```

Shader #259 - echo alpha

File: echo_alpha.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / (6.0/alpha));
    vec4 color3 = texture(samp, tc/ (4.0/alpha));
    vec4 color4 = texture(samp, tc/ (8.0/alpha));
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #260 - echo alpha three

File: echo_alpha_three.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / (0.9 + alpha));
    vec4 color3 = texture(samp, tc / (1.5 + alpha));
    vec4 color4 = texture(samp, tc / (2.0 + alpha));
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #262 - echo alpha two

File: echo_alpha_two.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / (0.9 * alpha));
    vec4 color3 = texture(samp, tc / (1.5 * alpha));
    vec4 color4 = texture(samp, tc / (2.0 * alpha));
```

```
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;  
}
```

Shader #265 - echo bgr

File: echo_bgr.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.
color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;

Technique: Driven by: time

```
#version 330  
in vec2 tc;  
out vec4 color;  
uniform float alpha_r;  
uniform float alpha_g;  
uniform float alpha_b;  
uniform float alpha;  
uniform vec4 optx;  
uniform vec4 random_var;  
uniform float alpha_value;  
uniform mat4 mv_matrix;  
uniform mat4 proj_matrix;  
uniform sampler2D samp;  
uniform float value_alpha_r, value_alpha_g, value_alpha_b;  
uniform float index_value;  
uniform float time_f;  
uniform float restore_black;  
void main(void)  
{  
    color = texture(samp, tc);  
    vec4 color2 = texture(samp, tc / 2);  
    vec4 color3 = texture(samp, tc / 4);  
    vec4 color4 = texture(samp, tc / 8);  
    color[0] = (color[2] * 0.5) + (color2[0] * 0.5);  
    color[1] = (color[1] * 0.5) + (color3[1] * 0.5);  
    color[2] = (color[0] * 0.5) + (color4[2] * 0.5);  
}
```

Shader #266 - echo cf

File: echo_cf.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330  
in vec2 tc;  
out vec4 color;  
uniform float alpha_r;  
uniform float alpha_g;  
uniform float alpha_b;  
uniform float alpha;  
uniform vec4 optx;  
uniform vec4 random_var;  
uniform float alpha_value;
```

```

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    vec2 cord1 = vec2(tc[0]/2, tc[1]/2);
    vec2 cord2 = vec2(tc[0]/4, tc[1]/4);
    vec2 cord3 = vec2(tc[0]/8, tc[1]/8);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    color[0] = color[0] * col1[0];
    color[1] = color[1] * col2[1];
    color[2] = color[2] * col3[2];
}

```

Shader #273 - echo div2

File: echo_div2.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc/2);
    vec2 tc1 = tc;
    vec2 tc2 = tc;
    tc1[0] = 1.0-tc1[0];
    tc2[1] = 1.0-tc2[1];
    vec4 color3 = texture(samp, tc1/2);

```



```
    vec4 color4 = texture(samp, tc2/2);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #274 - echo div4

File: echo_div4.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc/2);
    vec2 tc1 = tc;
    vec2 tc2 = tc;
    tc1[0] = 1.0-tc1[0];
    tc2[1] = 1.0-tc2[1];
    vec4 color3 = texture(samp, tc1/4);
    vec4 color4 = texture(samp, tc2/4);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #276 - echo g

File: echo_g.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
```

```

    return modVal <= length ? modVal : length * 2.0 - modVal;
}

void main(void) {
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec2 offset = vec2(0.01, 0.01) * time_t;
    vec4 color1 = texture(samp, tc);
    vec4 color2 = texture(samp, tc - offset);
    vec4 color3 = texture(samp, tc - offset * 2.0);
    vec4 color4 = texture(samp, tc - offset * 3.0);
    float weight1 = 0.5 * (1.0 - time_t);
    float weight2 = 0.3 * time_t;
    float weight3 = 0.15 * time_t;
    float weight4 = 0.05 * time_t;
    float totalWeight = weight1 + weight2 + weight3 + weight4;
    color = (color1 * weight1 + color2 * weight2 + color3 * weight3 + color4 * weight4) / sin(
        ↪ totalWeight * time_f);
    color.a = 1.0;
}

```

Shader #277 - echo loop

File: echo_loop.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 echo_loop(vec4 color, int count) {
    vec4 col = color;
    for(int i = 2; i < count; i += 2) {
        vec4 pos = texture(samp, tc/ (i * fract(time_f)));
        col = (0.6 * col) + (0.3 * pos);
    }
    return col;
}

void main(void)
{
    color = texture(samp, tc);
    color = echo_loop(color, 14);
}

```

Shader #278 - echo loop2

File: echo_loop2.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 echo_loop(vec4 color, int count) {
    vec4 col = color;
    for(int i = 2; i < count; i += 2) {
        vec4 pos = texture(samp, (1.0-tc)/i);
        col = (0.6 * col) + (0.6 * pos);
    }
    return col;
}
void main(void)
{
    color = texture(samp, tc);
    color = echo_loop(color, 8);
}
```

Shader #282 - echo mix

File: echo_mix.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
```

```

uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2;
    color2 = texture(samp, tc);
    vec4 color3;
    color3 = texture(samp, tc/2);
    vec4 color4;
    color4 = texture(samp, tc/4);
    color[0] = (0.5 * color[0]) + (0.5 * color2[0]);
    color[1] = (0.5 * color[1]) + (0.5 * color3[1]);
    color[2] = (0.5 * color[2]) + (0.5 * color4[2]);
}

```

Shader #285 - echo or

File: echo_or.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 or_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i] | isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i] % 255;
        icolor[i] = float(int_color[i]) / 255;
    }
}

```

```

    icolor.a = 1.0;
return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc /0.9);
    vec4 color3 = texture(samp, tc/ 1.5);
    vec4 color4 = texture(samp, tc/ 2.0);
    ivec4 icolor1 = ivec4(color2 * 255);
    ivec4 icolor2 = ivec4(color3 * 255);
    ivec4 icolor3 = ivec4(color4 * 255);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    ivec4 icolor = ivec4(color * 255);
    icolor[0] = icolor[0] ^ icolor1[2];
    icolor[1] = icolor[1] ^ icolor2[1];
    icolor[2] = icolor[2] ^ icolor3[0];
    color = or_RGB(color, icolor);
}

```

Shader #291 - echo rev x1

File: echo_rev_x1.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec2 tc1 = tc;
    tc1[0] = 1.0-tc[0];
    vec4 color2 = texture(samp, tc1/2);
    vec4 color3 = texture(samp, tc1/4);
    vec4 color4 = texture(samp, tc1/8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```

Shader #292 - echo rev x2

File: echo_rev_x2.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec2 tc1 = tc;
    tc1[0] = 1.0-tc[0];
    vec4 color2 = texture(samp, tc1/2);
    vec4 color3 = texture(samp, tc/4);
    vec4 color4 = texture(samp, tc1/8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #296 - echo s

File: echo_s.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void)
{
    vec2 center = vec2(0.5, 0.5);
    float angle = time_f;
    vec2 tc1 = tc;
    vec2 tc2 = tc - center;
    tc2 *= 0.8;
    tc2 = vec2(
        tc2.x * cos(angle + 1.0) - tc2.y * sin(angle + 1.0),
        tc2.x * sin(angle + 1.0) + tc2.y * cos(angle + 1.0)
    );
}
```

```

    tc2 += center;
    tc2 = fract(tc2);
    vec2 tc3 = tc - center;
    tc3 *= 0.6;
    tc3 = vec2(
        tc3.x * cos(angle + 2.0) - tc3.y * sin(angle + 2.0),
        tc3.x * sin(angle + 2.0) + tc3.y * cos(angle + 2.0)
    );
    tc3 += center;
    tc3 = fract(tc3);
    vec2 tc4 = tc - center;
    tc4 *= 0.4;
    tc4 = vec2(
        tc4.x * cos(angle + 3.0) - tc4.y * sin(angle + 3.0),
        tc4.x * sin(angle + 3.0) + tc4.y * cos(angle + 3.0)
    );
    tc4 += center;
    tc4 = fract(tc4);
    vec4 color1 = texture(samp, tc1);
    vec4 color2 = texture(samp, tc2);
    vec4 color3 = texture(samp, tc3);
    vec4 color4 = texture(samp, tc4);
    color = (color1 + color2 + color3 + color4) * 0.25;
}

```

Shader #298 - echo shift

File: echo_shift.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
vec4 shift_vec(vec4 inputv, int dir) {
    vec4 rev;
    if(dir == 1) {
        rev[0] = inputv[2];
        rev[1] = inputv[0];
        rev[2] = inputv[1];
    } else {
        rev[0] = inputv[1];
        rev[1] = inputv[2];
        rev[2] = inputv[0];
    }
}

```

```

    }
    return rev;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc/4);
    vec4 color3 = texture(samp, tc/7);
    vec4 color4 = texture(samp, tc/11);
    vec4 color5 = texture(samp, tc/18);
    color = (color * 0.3) + (shift_vec(color2, 1) * 0.3) + (color3 * 0.3) + (shift_vec(color4, 0) * 0.3)
    ↪ + (color5 * 0.3);
}

```

Shader #299 - echo sin

File: echo_sin.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc / 4);
    vec4 color4 = texture(samp, tc / 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = sin(color * time_t);
    color.a = 1.0;
}

```

Shader #300 - echo sin particle

File: echo_sin_particle.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -

Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

```



```

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec2 hash(vec2 p) {
    p = vec2(dot(p, vec2(127.1, 311.7)), dot(p, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(p) * 43758.5453123);
}

float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(
        mix(dot(hash(i + vec2(0.0, 0.0)), f - vec2(0.0, 0.0)), dot(hash(i + vec2(1.0, 0.0)), f - vec2
        ↪ (1.0, 0.0)), u.x),
        mix(dot(hash(i + vec2(0.0, 1.0)), f - vec2(0.0, 1.0)), dot(hash(i + vec2(1.0, 1.0)), f - vec2
        ↪ (1.0, 1.0)), u.x),
        u.y);
}

void main() {
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    vec3 col = vec3(0.0);
    float numParticles = 50.0;
    for (float i = 0.0; i < numParticles; i += 1.0) {
        float t = time_f * 0.1 + i * 6.28 / numParticles;
        vec2 pos = vec2(sin(t + noise(vec2(i, t)) * 10.0), cos(t + noise(vec2(i, t)) * 10.0)) * 0.3 +
        ↪ 0.5;
        float d = length(uv - pos);
        col += vec3(0.1, 0.5, 0.8) * (1.0 - d) * exp(-d * 10.0);
    }
    color = vec4(col, 1.0);
    color = mix(texture(samp, tc), color, 0.5);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc / 4);
    vec4 color4 = texture(samp, tc / 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = sin(color * time_t);
    color.a = 1.0;
}

```

Shader #301 - echo strobe x1

File: echo_strobe_x1.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;

```

```

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
    float value = fract(timeval);
    if(value >= 0 && value <= 0.3) {
        color[0] *= value;
    } else if (value >= 0.3 && value <= 0.6) {
        color[1] *= value;
    } else if(value >= 0.6 && value <= 1.0) {
        color[2] *= value;
    }
}

```

Shader #302 - echo two

File: echo_two.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc /0.9);
    vec4 color3 = texture(samp, tc/ 1.5);
    vec4 color4 = texture(samp, tc/ 2.0);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```

Shader #303 - echo x2

File: echo_x2.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec4 inc_value;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color = color * inc_value/255 * alpha;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #304 - echo x3

File: echo_x3.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
```

```

uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec4 inc_value;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 vpos;
    vpos[0] = 1.0-tc[0];
    vpos[1] = tc[1];
    vec4 color2 = texture(samp, vpos);
    color = (0.5 * color) + (0.5 * color2);
    vec4 color2x = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color = (color * 0.3) + (color2x * 0.3) + (color3 * 0.3) + (color4 * 0.3) ;
}

```

Shader #305 - echo x4

File: echo_x4.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates. Applies matrix transformations.

Technique: UV mirroring/folding for symmetry - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
in vec3 vpos;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec4 inc_value;
uniform float restore_black;
void main(void)

```

```

{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 vpos;
    vpos[0] = 1.0-tc[0];
    vpos[1] = tc[1];
    vec4 color2 = texture(samp, vpos);
    color = (0.5 * color) + (0.5 * color2);
    vec4 color2x = texture(samp, tc / 2);
    color2x[0] = 1.0;
    vec4 color3 = texture(samp, tc/ 4);
    color3[1] = 1.0;
    vec4 color4 = texture(samp, tc/ 8);
    color4[2] = 1.0;
    color = (color * 0.3) + (color2x * 0.3) + (color3 * 0.3) + (color4 * 0.3) ;
}

```

Shader #306 - echo x5

File: echo_x5.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc/ 4);
    vec4 color4 = texture(samp, tc/ 8);
    color2[0] *= (alpha/8);
    color3[1] *= (alpha/12);
    color4[2] *= (alpha/14);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```

Shader #307 - echo x6

File: echo_x6.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc/2);
    vec4 color3 = texture(samp, tc/alpha);
    vec4 color4 = texture(samp, tc/4);
    color2[2] *= (alpha_r/2);
    color3[1] *= (alpha_g/4);
    color4[0] *= (alpha_b/8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}
```

Shader #408 - frac shader04 echo

File: frac_shader04_echo.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - RGB $\leftrightarrow$ HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
```

```

uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);

```

```

vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
float t = time_f + iTime;
float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
float seg = 4.0 + 2.0 * sin(t * 0.25);
float zoom = 1.45 + 0.45 * sin(t * 0.35 + 0.2 * sr);
vec2 kUV = reflectUV(tc, seg, m, ar.x);
kUV = fractalFold(kUV, zoom, t, m, ar.x);
kUV = rotateUV(kUV, t * 0.22, m, ar.x);
vec2 dir = (kUV - m) * ar;
float r = length(dir);
float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
vec2 uB = mix(tc, kUV, 0.88);
vec2 uC = mix(tc, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
float hueBase = fract(t * 0.06);
float sat = 0.7 + 0.25 * sin(t * 0.4 + sr);
float val = 0.6 + 0.35 * sin(t * 0.5 + dot(tc, vec2(2.3, 1.9)));
vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
float slowBeat = 0.5 + 0.5 * sin(t * 0.8 + aMix * 0.05);
float mix1 = 0.5 + 0.5 * sin(t * 0.6);
float mix2 = 0.5 + 0.5 * cos(t * 0.45);
vec3 warpCol = mix(sin(t1.rgb * pingPong(time_f * PI, 10.0)), sin(t2.rgb * pingPong(time_f * PI /
↳ 2.0, 8.0)), mix1);
warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
warpCol *= tint * (0.92 + 0.08 * slowBeat);
vec2 dtx = dFdx(tc);
vec2 dty = dFdy(tc);
vec3 base = textureGrad(samp, tc, dtx, dty).rgb;
vec4 s1 = textureGrad(samp, tc, dtx, dty);
vec4 s2 = textureGrad(samp, tc * 0.5, dtx * 0.5, dty * 0.5);
vec4 s3 = textureGrad(samp, tc * 0.25, dtx * 0.25, dty * 0.25);
vec4 s4 = textureGrad(samp, tc * 0.125, dtx * 0.125, dty * 0.125);
vec3 multi = (s1.rgb + s2.rgb + s3.rgb + s4.rgb) * 0.25;
vec3 combined = mix(base, warpCol * 3.0, 0.6);
float multiMix = 0.45 + 0.35 * sin(t * 0.25 + aMix * 0.02);
vec3 multiBlend = mix(base, multi, 0.7);
combined = mix(combined, multiBlend, multiMix);
vec3 bloom = combined * combined * 0.18 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.10;
combined += bloom;
combined = clamp(combined, vec3(0.0), vec3(1.0));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
combined *= 1.0 + _ab * 0.6;
combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(combined, 1.0);
}

```

Shader #409 - frac shader04 echo2

File: frac_shader04_echo2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
 - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation -
 Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect, float spin){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x) + spin;
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
```

```

        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime;
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float seg = 4.0 + 2.0 * sin(t * 0.25);
    float zoom = 1.45 + 0.45 * sin(t * 0.35 + 0.2 * sr);
    float kaleidoSpin = t * 0.5 + aMix * 0.05;
    vec2 kUV = reflectUV(tc, seg, m, ar.x, kaleidoSpin);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    float radial = length((kUV - m) * ar);
    float globalSpin = t * 0.6;
    kUV = rotateUV(kUV, globalSpin + radial * 4.0, m, ar.x);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
    vec2 uB = mix(tc, kUV, 0.88);
    vec2 uC = mix(tc, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
    vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
    vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
    vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
    float hueBase = fract(t * 0.06);
    float sat = 0.7 + 0.25 * sin(t * 0.4 + sr);
    float val = 0.6 + 0.35 * sin(t * 0.5 + dot(tc, vec2(2.3, 1.9)));
    vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
    float slowBeat = 0.5 + 0.5 * sin(t * 0.8 + aMix * 0.05);
    float mix1 = 0.5 + 0.5 * sin(t * 0.6);
    float mix2 = 0.5 + 0.5 * cos(t * 0.45);
    vec3 warpCol = mix(sin(t1.rgb * pingPong(time_f * PI, 10.0)), sin(t2.rgb * pingPong(time_f * PI /
    ↪ 2.0, 8.0)), mix1);
    warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
    warpCol *= tint * (0.92 + 0.08 * slowBeat);
    vec2 dtx = dFdx(tc);
    vec2 dty = dFdy(tc);
    vec3 base = textureGrad(samp, tc, dtx, dty).rgb;
    vec4 s1 = textureGrad(samp, tc, dtx, dty);
    vec4 s2 = textureGrad(samp, tc * 0.5, dtx * 0.5, dty * 0.5);
    vec4 s3 = textureGrad(samp, tc * 0.25, dtx * 0.25, dty * 0.25);
    vec4 s4 = textureGrad(samp, tc * 0.125, dtx * 0.125, dty * 0.125);
    vec3 multi = (s1.rgb + s2.rgb + s3.rgb + s4.rgb) * 0.25;
    vec3 combined = mix(base, warpCol * 3.0, 0.6);
}

```

```

float multiMix = 0.45 + 0.35 * sin(t * 0.25 + aMix * 0.02);
vec3 multiBlend = mix(base, multi, 0.7);
combined = mix(combined, multiBlend, multiMix);
vec3 bloom = combined * combined * 0.18 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.10;
combined += bloom;
combined = clamp(combined, vec3(0.0), vec3(1.0));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
combined *= 1.0 + _ab * 0.6;
combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(combined, 1.0);
}

```

Shader #410 - frac shader04 echo3 spin

File: frac_shader04_echo3_spin.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

```

```

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect, float spin){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x) + spin;
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime;
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float seg = 4.0 + 2.0 * sin(t * 0.25);
    float zoom = 1.45 + 0.45 * sin(t * 0.35 + 0.2 * sr);
    float kaleidoSpin = t * 0.5 + aMix * 0.05;
    vec2 kUV = reflectUV(tc, seg, m, ar.x, kaleidoSpin);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    float radial = length((kUV - m) * ar);
    float globalSpin = t * 0.6;
    kUV = rotateUV(kUV, globalSpin + radial * 4.0, m, ar.x);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
    vec2 uB = mix(tc, kUV, 0.88);
    vec2 uC = mix(tc, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
}

```

```

vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
float hueBase = fract(t * 0.06);
float sat = 0.7 + 0.25 * sin(t * 0.4 + sr);
float val = 0.6 + 0.35 * sin(t * 0.5 + dot(tc, vec2(2.3, 1.9)));
vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
float mix1 = 0.5 + 0.5 * sin(t * 0.6);
float mix2 = 0.5 + 0.5 * cos(t * 0.45);
vec3 warpCol = mix(sin(t1.rgb * pingPong(time_f * PI, 10.0)), sin(t2.rgb * pingPong(time_f * PI /
↳ 2.0, 8.0)), mix1);
warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
warpCol *= tint;
vec2 dtx = dFdx(tc);
vec2 dty = dFdy(tc);
vec3 base = textureGrad(samp, tc, dtx, dty).rgb;
vec4 s1 = textureGrad(samp, tc, dtx, dty);
vec4 s2 = textureGrad(samp, tc * 0.5, dtx * 0.5, dty * 0.5);
vec4 s3 = textureGrad(samp, tc * 0.25, dtx * 0.25, dty * 0.25);
vec4 s4 = textureGrad(samp, tc * 0.125, dtx * 0.125, dty * 0.125);
vec3 multi = (s1.rgb + s2.rgb + s3.rgb + s4.rgb) * 0.25;
vec3 multiBlend = mix(base, multi, 0.7);
vec3 pattern = mix(warpCol, multiBlend, 0.4);
float PATTERN_ALPHA = 0.85;
float BASE_ALPHA = 0.15;
vec3 combined = base * BASE_ALPHA + pattern * PATTERN_ALPHA;
vec3 bloom = combined * combined * 0.18 + pow(max(combined - 0.6, 0.0), vec3(2.0)) * 0.10;
combined += bloom;
combined = clamp(combined, vec3(0.0), vec3(1.0));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
combined *= 1.0 + _ab * 0.6;
combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(combined, 1.0);
}

```

Shader #416 - frac shader echo4 spin frac

File: frac_shader_echo4_spin_frac.glsl

Renders a fractal pattern using iterative computation (many iterations). Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Iterative fractal computation with complex-plane mapping - Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;

```

```

uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float rand(vec2 co){
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 screenToComplex(vec2 screenPos, vec2 center, float zoom){
    vec2 scale = vec2(zoom * (iResolution.x / iResolution.y), zoom);
    return center + (screenPos - iResolution * 0.5) * scale;
}
const int maxIterations = 300;
float mandelbrotSmooth(vec2 c, out float itCount){
    vec2 z = vec2(0.0);
    float maxI = float(maxIterations);
    float iFloat = 0.0;
    for(int j = 0; j < maxIterations; j++){
        iFloat = float(j);
        z = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + c;
        if(dot(z, z) > 4.0){
            float log_zn = log(dot(z, z)) * 0.5;
            float nu = iFloat + 1.0 - log(log_zn / log(2.0)) / log(2.0);
            itCount = iFloat;
            return clamp(nu / maxI, 0.0, 1.0);
        }
    }
    itCount = maxI;
    return 1.0;
}
vec2 interestingCenters(int index){
    vec2 centers[5];
    centers[0] = vec2(-0.743643887037151, 0.13182590420533);
    centers[1] = vec2(-0.745428, 0.113009);
    centers[2] = vec2(-0.7453, 0.1127);
    centers[3] = vec2(-0.7435, 0.1318);
    centers[4] = vec2(-0.743644786, 0.1318252536);
    return centers[index % 5];
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;

```

```

    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect, float spin){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x) + spin;
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

void main(void){
    vec2 res = iResolution;
    vec2 uv = tc;
    vec2 ar = vec2(res.x / res.y, 1.0);
    float baseTime = time_f + iTime;
    float t = baseTime + float(iFrame) * 0.002;
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float deltaFactor = clamp(iTimeDelta * 60.0, 0.2, 3.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / res) : vec2(0.5));
    vec2 clickUV = (iMouseClicked.x > 0.0 || iMouseClicked.y > 0.0) ? (iMouseClicked / res) : m;
    float datePhase = dot(iDate.xy, vec2(0.01, 0.003));
    float frPhase = clamp(iFrameRate / 60.0, 0.4, 2.0);
    float chBeat = 0.0;
    for(int i = 0; i < 4; i++){
        chBeat += sin(iChannelTime[i] * 0.35 + float(i));
    }
    chBeat *= 0.25;
    float zoomCycleSpeed = 0.035 * deltaFactor * frPhase;
    float zoomIntensity = 14.0 + aMix * 0.25;
    float zoomPing = pingPong(baseTime * zoomCycleSpeed * (0.6 + 0.4 * sr), 1.0);
    float zoom = exp(-zoomPing * zoomIntensity);
    int cycleIndex = int(floor(baseTime * 0.08 + chBeat * 2.0));
    vec2 center = interestingCenters(cycleIndex);

```

```

float seg = 5.0 + 3.0 * sin(t * 0.25 + chBeat);
float kaleidoSpin = t * 0.45 + aMix * 0.03;
vec2 kUV = reflectUV(uv, seg, m, ar.x, kaleidoSpin);
float zoomFold = 1.3 + 0.4 * sin(t * 0.3 + sr) + 0.2 * chBeat;
kUV = fractalFold(kUV, zoomFold, t, m, ar.x);
float radial = length((kUV - m) * ar);
float globalSpin = t * 0.6 + 0.3 * chBeat;
kUV = rotateUV(kUV, globalSpin + radial * 3.5, m, ar.x);
vec2 frag = vec2(uv.x * res.x, uv.y * res.y);
vec2 fragWarp = frag
    + (kUV - uv) * res.y * (0.35 + 0.25 * aMix / 20.0)
    + (clickUV - m) * res * 0.6 * exp(-radial * 4.0);
vec2 cplx = screenToComplex(fragWarp, center, zoom);
float spin = t * 0.18 + 0.25 * chBeat;
float cs = cos(spin), sn = sin(spin);
vec2 cRel = cplx - center;
cRel = vec2(cRel.x * cs - cRel.y * sn, cRel.x * sn + cRel.y * cs);
cplx = cRel + center;
float iterCount;
float nu = mandelbrotSmooth(cplx, iterCount);
vec3 neon = neonPalette(t + nu * 6.0 + datePhase);
float glow = smoothstep(0.0, 1.0, pow(nu, 0.9));
vec3 fractCol = mix(neon * 0.4, neon * 1.8, glow);
float ring = smoothstep(0.2, 0.0, abs(nu - 0.5));
fractCol *= 0.8 + 0.6 * ring;
fractCol *= 0.7 + 0.6 * aMix / 20.0;
vec2 dtx = dFdx(uv);
vec2 dty = dFdy(uv);
vec4 baseTex = textureGrad(samp, uv, dtx, dty);
vec3 base = baseTex.rgb;
vec4 s2 = textureGrad(samp, uv * 0.5, dtx * 0.5, dty * 0.5);
vec4 s3 = textureGrad(samp, uv * 0.25 + 0.03 * vec2(sin(t), cos(t)), dtx * 0.25, dty * 0.25);
vec4 s4 = textureGrad(samp, uv * 0.125 + 0.09 * (kUV - m) * ar, dtx * 0.125, dty * 0.125);
vec3 multi = (base + s2.rgb + s3.rgb + s4.rgb) * 0.25;
float multiMix = 0.25 + 0.35 * (1.0 - nu);
vec3 fractEnhanced = fractCol;
fractEnhanced *= 0.9 + 0.4 * sin(t * 0.4 + nu * 12.0 + chBeat);
vec3 pattern = mix(fractEnhanced, multi, multiMix);
float PATTERN_ALPHA = 0.88;
float BASE_ALPHA = 0.20;
vec3 combined = base * BASE_ALPHA + pattern * PATTERN_ALPHA;
float vignette = 1.0 - 0.45 * radial * radial;
combined *= clamp(vignette + 0.2, 0.0, 1.3);
vec3 bloom = combined * combined * 0.35 + pow(max(combined - 0.55, 0.0), vec3(3.0)) * 0.25;
combined += bloom;
combined = pow(max(combined, 0.0), vec3(0.9));
combined = clamp(combined, vec3(0.0), vec3(1.0));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
combined *= 1.0 + _ab * 0.6;
combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(combined, 1.0);
}

```

Shader #417 - frac shader echo4 spin full

File: frac_shader_echo4_spin_full.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling

effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
- Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation -
Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect, float spin){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x) + spin;
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
```

```

    vec2 p = uv;
    for(int i = 0; i < 6; i++){
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t){
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.05);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

void main(void){
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 m = (iMouse.z > 0.5 ? (iMouse.xy / iResolution) : vec2(0.5));
    float t = time_f + iTime;
    float sr = clamp(iSampleRate / 48000.0, 0.25, 4.0);
    float aMix = clamp(amp * 0.7 + uamp * 0.3, 0.0, 20.0);
    float seg = 4.0 + 2.0 * sin(t * 0.25);
    float zoom = 1.45 + 0.45 * sin(t * 0.35 + 0.2 * sr);
    float kaleidoSpin = t * 0.5 + aMix * 0.05;
    vec2 kUV = reflectUV(tc, seg, m, ar.x, kaleidoSpin);
    kUV = fractalFold(kUV, zoom, t, m, ar.x);
    float radial = length((kUV - m) * ar);
    float globalSpin = t * 0.6;
    kUV = rotateUV(kUV, globalSpin + radial * 4.0, m, ar.x);
    vec2 dir = (kUV - m) * ar;
    float r = length(dir);
    float ripple = sin(20.0 * r - t * 9.0) * 0.012 / (1.0 + 18.0 * r);
    ripple *= 1.0 / (1.0 + 12.0 * (abs(dFdx(r)) + abs(dFdy(r))));
    vec2 uA = kUV + normalize(dir + 1e-5) * ripple;
    vec2 uB = mix(tc, kUV, 0.88);
    vec2 uC = mix(tc, kUV, 0.94) + vec2(0.002 * sin(t), 0.002 * cos(t));
    vec4 t1 = textureGrad(samp, fract(uA), dFdx(uA), dFdy(uA));
    vec4 t2 = textureGrad(samp, fract(uB), dFdx(uB), dFdy(uB));
    vec4 t3 = textureGrad(samp, fract(uC), dFdx(uC), dFdy(uC));
    float hueBase = fract(t * 0.06);
    float sat = 0.8;
    float val = 0.75 + 0.25 * sin(t * 0.5 + dot(tc, vec2(2.3, 1.9)));
    vec3 tint = hsv2rgb(vec3(hueBase + r * 0.25, sat, val));
    float mix1 = 0.5 + 0.5 * sin(t * 0.6);
    float mix2 = 0.5 + 0.5 * cos(t * 0.45);
    vec3 warpCol = mix(sin(t1.rgb * 5.0), sin(t2.rgb * 4.0), mix1);
    warpCol = mix(warpCol, t3.rgb, mix2 * 0.5);
    warpCol *= tint;
    warpCol = pow(max(warpCol, 0.0), vec3(0.85)) * 1.8;
    vec2 dtx = dFdx(tc);
    vec2 dty = dFdy(tc);
    vec3 base = textureGrad(samp, tc, dtx, dty).rgb;
    vec4 s1 = textureGrad(samp, tc, dtx, dty);
    vec4 s2 = textureGrad(samp, tc * 0.5, dtx * 0.5, dty * 0.5);
    vec4 s3 = textureGrad(samp, tc * 0.25, dtx * 0.25, dty * 0.25);
    vec4 s4 = textureGrad(samp, tc * 0.125, dtx * 0.125, dty * 0.125);
    vec3 multi = (s1.rgb + s2.rgb + s3.rgb + s4.rgb) * 0.25;

```

```

    vec3 multiBlend = mix(base, multi, 0.7);
    vec3 pattern = mix(warpCol, multiBlend, 0.35);
    pattern *= 1.7;
    float PATTERN_ALPHA = 0.8;
    float BASE_ALPHA = 0.2;
    vec3 combined = base * BASE_ALPHA + pattern * PATTERN_ALPHA;
    vec3 bloom = combined * combined * 0.25 + pow(max(combined - 0.5, 0.0), vec3(3.0)) * 0.22;
    combined += bloom;
    combined = clamp(combined, vec3(0.0), vec3(1.0));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    combined *= 1.0 + _ab * 0.6;
    combined = mix(combined, combined * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(combined, 1.0);
}

```

Shader #497 - gghost

File: gghost.glsl

Blends multiple texture samples together using interpolation for a composite effect.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec4 texColor = texture(samp, uv);
    float trail = sin(time_f * 2.0 + length(uv - 0.5)) * 0.5 + 0.5;
    vec3 ghostColor = texColor.rgb * vec3(0.8, 0.9, 1.0) * trail;
    vec3 shiftedColor = vec3(
        texture(samp, uv + vec2(0.01, 0.02) * sin(time_f)).r,
        texture(samp, uv + vec2(-0.02, 0.01) * cos(time_f)).g,
        texture(samp, uv + vec2(0.02, -0.01) * sin(time_f)).b
    );
    vec3 finalColor = mix(ghostColor, shiftedColor, 0.5);
    color = vec4(finalColor, texColor.a);
}

```

Shader #500 - gkale echo

File: gkale_echo.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;

```

```

uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 firstEffect(vec2 tc) {
    vec2 center = vec2(0.5, 0.5);
    float angle = time_f;
    vec2 tc1 = tc;
    vec2 tc2 = tc - center;
    tc2 *= 0.8;
    tc2 = vec2(
        tc2.x * cos(angle + 1.0) - tc2.y * sin(angle + 1.0),
        tc2.x * sin(angle + 1.0) + tc2.y * cos(angle + 1.0)
    );
    tc2 += center;
    tc2 = fract(tc2);
    vec2 tc3 = tc - center;
    tc3 *= 0.6;
    tc3 = vec2(
        tc3.x * cos(angle + 2.0) - tc3.y * sin(angle + 2.0),
        tc3.x * sin(angle + 2.0) + tc3.y * cos(angle + 2.0)
    );
    tc3 += center;
    tc3 = fract(tc3);
    vec2 tc4 = tc - center;
    tc4 *= 0.4;
    tc4 = vec2(
        tc4.x * cos(angle + 3.0) - tc4.y * sin(angle + 3.0),
        tc4.x * sin(angle + 3.0) + tc4.y * cos(angle + 3.0)
    );
    tc4 += center;
    tc4 = fract(tc4);
    vec4 color1 = texture(samp, tc1);
    vec4 color2 = texture(samp, tc2);
    vec4 color3 = texture(samp, tc3);
    vec4 color4 = texture(samp, tc4);
    return (color1 + color2 + color3 + color4) * 0.25;
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv) * 1.4142;
    float segments = 8.0;
    angle = mod(angle, 6.28318 / segments);
    angle = abs(angle - 3.14159 / segments);
    uv = vec2(cos(angle), sin(angle)) * radius;
    uv = uv * 0.5 + 0.5;
    uv = clamp(uv, 0.0, 1.0);
    float time_t = pingPong(time_f * 0.5, 7.0) + 2.0;
    vec4 texColor = firstEffect(uv);
    float pattern = cos(radius * 10.0 - time_t * 5.0);
    vec3 colorShift = vec3(
        0.5 + 0.5 * cos(pattern + time_t + 0.0),
        0.5 + 0.5 * cos(pattern + time_t + 2.094),
        0.5 + 0.5 * cos(pattern + time_t + 4.188)
    );
    vec3 finalColor = texColor.rgb * colorShift;
    color = vec4(finalColor, texColor.a);
    color = mix(color, firstEffect(tc), 0.5);
}

```

```

    color = sin(color * time_t);
    color.a = 1.0;
}

```

Shader #501 - gkale echo2

File: gkale_echo2.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 firstEffect(vec2 tc) {
    vec2 center = vec2(0.5, 0.5);
    float angle = time_f;
    vec2 tc1 = tc;
    vec2 tc2 = (tc - center) * 0.9;
    tc2 = vec2(
        tc2.x * cos(angle + 1.5) - tc2.y * sin(angle + 1.5),
        tc2.x * sin(angle + 1.5) + tc2.y * cos(angle + 1.5)
    ) + center;
    tc2 = fract(tc2);
    vec2 tc3 = (tc - center) * 0.7;
    tc3 = vec2(
        tc3.x * cos(angle + 3.0) - tc3.y * sin(angle + 3.0),
        tc3.x * sin(angle + 3.0) + tc3.y * cos(angle + 3.0)
    ) + center;
    tc3 = fract(tc3);
    vec2 tc4 = (tc - center) * 0.5;
    tc4 = vec2(
        tc4.x * cos(angle + 4.5) - tc4.y * sin(angle + 4.5),
        tc4.x * sin(angle + 4.5) + tc4.y * cos(angle + 4.5)
    ) + center;
    tc4 = fract(tc4);
    vec4 color1 = texture(samp, tc1);
    vec4 color2 = texture(samp, tc2);
    vec4 color3 = texture(samp, tc3);
    vec4 color4 = texture(samp, tc4);
    return (color1 + color2 + color3 + color4) * 0.4;
}

vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255.0);
    for (int i = 0; i < 3; ++i) {
        int_color[i] = int(255.0 * icolor[i]);
        int_color[i] = int_color[i] ^ isource[i];
        if (int_color[i] > 255)
            int_color[i] = int_color[i] % 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
}

```

```

    }
    icolor.a = 1.0;
return icolor;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](
        0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
        1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
        1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
        2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
        2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
        2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
        2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
        1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
        1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
        0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5
    );
    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv) * 1.4142;
    float segments = 12.0;
    angle = mod(angle, 6.28318 / segments);
    angle = abs(angle - 3.14159 / segments);
    uv = vec2(cos(angle), sin(angle)) * radius;
    uv = uv * 0.5 + 0.5;
    uv = clamp(uv, 0.0, 1.0);
    float time_t1 = pingPong(time_f * 0.7, 7.0) + 2.0;
    vec4 texColor = firstEffect(uv);
    float pattern = sin(radius * 15.0 - time_t1 * 7.0);
    vec3 colorShift = vec3(
        0.6 + 0.4 * cos(pattern + time_t1 * 1.5 + 0.0),
        0.6 + 0.4 * cos(pattern + time_t1 * 1.5 + 2.094),
        0.6 + 0.4 * cos(pattern + time_t1 * 1.5 + 4.188)
    );
    vec3 vibrantColor = texColor.rgb * colorShift * 1.5;
    vibrantColor = clamp(vibrantColor, 0.0, 1.0);
    float time_z = pingPong(time_f, 8.0) + 2.0;
    vec4 color1 = vec4(vibrantColor, texColor.a);
    color1 = mix(sin(color1 * time_z), firstEffect(tc), 0.6);
    vec4 tcolor = blur(samp, uv, iResolution);
    float time_t2 = pingPong(time_f, 10.0) + 2.0;
    color = xor_RGB(color1, tcolor * time_t2);
    color.a = 1.0;
}

```

Shader #528 - gpt echo

File: gpt_echo.glsl

Creates mirror/symmetry effects by reflecting or folding the texture coordinates.

Technique: UV mirroring/folding for symmetry

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
void main(void) {
    vec2 mirrored_tc = abs(fract(tc) - 0.5) + 0.5;
    vec4 color1 = texture(samp, mirrored_tc);
    vec4 color2 = texture(samp, mirrored_tc * 0.5);
    vec4 color3 = texture(samp, mirrored_tc * 0.25);
    vec4 color4 = texture(samp, mirrored_tc * 0.125);
    color = (color1 * 0.4) + (color2 * 0.3) + (color3 * 0.2) + (color4 * 0.1);
}
```

Shader #547 - green echo

File: green_echo.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec4 color3 = texture(samp, tc / 4);
    vec4 color_value = vec4(0.1, 0.6, 0.2, 1.0);
    color = ((color*color_value) * 0.4) + (color2 * 0.4) + (color3 * 0.4) * color_value * alpha;
}
```

Shader #562 - gtrail

File: gtrail.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec4 texColor = texture(samp, uv);
    vec2 trailOffset1 = vec2(0.02, 0.03) * sin(time_f + length(uv - 0.5) * 10.0);
    vec2 trailOffset2 = vec2(-0.03, 0.02) * cos(time_f * 1.5 + length(uv - 0.5) * 15.0);
    vec2 trailOffset3 = vec2(0.01, -0.01) * sin(time_f * 3.0);
    vec3 trail1 = texture(samp, uv + trailOffset1).rgb * 0.7;
    vec3 trail2 = texture(samp, uv + trailOffset2).rgb * 0.5;
    vec3 trail3 = texture(samp, uv + trailOffset3).rgb * 0.3;
    vec3 intenseTrail = trail1 + trail2 + trail3;
    vec3 vibrantShift = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 20.0),
        0.5 + 0.5 * cos(time_f + uv.y * 20.0),
        0.5 + 0.5 * sin(time_f + uv.x * uv.y * 20.0)
    );
    vec3 finalColor = mix(intenseTrail, vibrantShift, 0.6);
    color = vec4(finalColor, texColor.a);
}
```

Shader #563 - gtrail2

File: gtrail2.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
out vec4 color;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main() {
    vec2 uv = gl_FragCoord.xy / iResolution;
    vec4 texColor = texture(samp, uv);
    vec2 trailOffset1 = vec2(0.02, 0.03) * sin(time_f + length(uv - 0.5) * 10.0);
    vec2 trailOffset2 = vec2(-0.03, 0.02) * cos(time_f * 1.5 + length(uv - 0.5) * 15.0);
    vec2 trailOffset3 = vec2(0.01, -0.01) * sin(time_f * 3.0);
    vec3 trail1 = texture(samp, uv + trailOffset1).rgb * 0.7;
    vec3 trail2 = texture(samp, uv + trailOffset2).rgb * 0.5;
    vec3 trail3 = texture(samp, uv + trailOffset3).rgb * 0.3;
    vec3 intenseTrail = trail1 + trail2 + trail3;
```



```

    vec3 vibrantShift = vec3(
        0.5 + 0.5 * sin(time_f + uv.x * 20.0),
        0.5 + 0.5 * cos(time_f + uv.y * 20.0),
        0.5 + 0.5 * sin(time_f + uv.x * uv.y * 20.0)
    );
    vec3 finalColor = mix(intenseTrail, vibrantShift, 0.6);
    color = vec4(sin(finalColor * (pingPong(time_f, 8.0) + 2.0)), texColor.a);
}

```

Shader #587 - HyperFocusTrails

File: HyperFocusTrails.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Applies matrix transformations. Normalize coordinates PASS 1: Current Frame (Brightest) PASS 2: Past Frame 1 (Dimmer, delayed)

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
const float PI = 3.14159265359;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 3; i++) {
        v += a * noise(p);
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}
vec3 electricPalette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.00, 0.33, 0.67);
    return a + b * cos(6.28318 * (c * t + d));
}

```

```

vec3 renderLayer(vec2 uv, float t) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 center = vec2(0.5);
    vec2 p = (uv - center) * ar;
    float len = length(p);
    float twistStrength = 3.0 * sin(t * 0.3);
    float angle = atan(p.y, p.x) + twistStrength / (len + 0.1) + t * 0.5;
    vec2 twistedP = vec2(cos(angle), sin(angle)) * len;
    float rippleWavelength = 8.0;
    float rippleSpeed = 4.0;
    float noiseVal = fbm(twistedP * 5.0 + t);
    float rippleX = sin(twistedP.x * rippleWavelength + t * rippleSpeed + noiseVal * 2.0);
    float rippleY = cos(twistedP.y * rippleWavelength + t * rippleSpeed - noiseVal * 2.0);
    float rippleAmp = 0.05 + 0.05 * noiseVal;
    vec2 rippledP = twistedP + vec2(rippleX, rippleY) * rippleAmp;
    vec3 p3 = vec3(rippedP, 1.0);
    mat3 R = rotX(t * 0.2) * rotY(t * 0.15) * rotZ(t * 0.1);
    vec3 r = R * p3;
    float zScale = 1.0 / (1.0 + r.z * 0.5);
    vec2 projUV = r.xy * zScale;
    vec2 tiledUV = 1.0 - abs(1.0 - 2.0 * (projUV + 0.5));
    tiledUV = tiledUV - floor(tiledUV);
    float aberrationStr = 0.01 + 0.04 * (1.0 / (len + 0.5));
    vec2 offset = vec2(aberrationStr, 0.0);
    vec3 texR = texture(samp, clamp(tiledUV - offset, 0.0, 1.0)).rgb;
    vec3 texG = texture(samp, clamp(tiledUV, 0.0, 1.0)).rgb;
    vec3 texB = texture(samp, clamp(tiledUV + offset, 0.0, 1.0)).rgb;
    vec3 layerColor = vec3(texR.r, texG.g, texB.b);
    float glowMask = smoothstep(0.6, 0.95, abs(rippleX * rippleY));
    vec3 palette = electricPalette(len - t);
    return mix(layerColor, layerColor + palette, glowMask * 0.8);
}

void main(void) {
    vec3 col0 = renderLayer(tc, time_f);
    vec3 col1 = renderLayer(tc, time_f - 0.1);
    vec3 col2 = renderLayer(tc, time_f - 0.2);
    vec3 finalColor = col0 + (col1 * 0.5) + (col2 * 0.25);
    finalColor = finalColor / (1.0 + finalColor * 0.4);
    color = vec4(finalColor, 1.0);
}

```

Shader #1007 - Temporal

File: Temporal.glsl

Applies a time-animated visual transformation to the input texture. Controls the height of the horizontal bands Simulate the buffer size Simulate start_index moving over time

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    int square_size = 64;
    int numFrames = 30;

```

```

    int start_index = int(time_f * 20.0);
    int start_dir = 1;
    int y = int(tc.y * iResolution.y);
    int block_row = y / square_size;
    int period = 2 * (numFrames - 1);
    if (period <= 0) {
        period = 1;
    }
    int start_pos = (start_dir == 1) ? start_index : (2 * (numFrames - 1)) - start_index;
    int pos = (start_pos + block_row) % period;
    if (pos < 0) pos += period;
    int frame_index = (pos < numFrames) ? pos : period - pos;
    if (frame_index < 0) frame_index = 0;
    if (frame_index >= numFrames) frame_index = numFrames - 1;
    vec4 currentFrame = texture(samp, tc);
    float timeDisplacement = float(frame_index) * 0.005;
    vec2 historyUV = tc;
    historyUV.x += timeDisplacement;
    if(historyUV.x > 1.0) historyUV.x -= 1.0;
    vec4 historyFrame = texture(samp, historyUV);
    color = (currentFrame * 0.5) + (historyFrame * 0.5);
    color.a = 1.0;
}

```

Glitch & Digital (43 shaders)

Shader #3 - 8bit norm

File: 8bit-norm.glsl

Applies a time-animated visual transformation to the input texture. Normalizes the color palette by reducing the depth per channel Scales 0.0-1.0 to 0.0-levels, rounds it, then scales back This creates discrete color steps without shifting the base hue

Technique: Indexed color palette look-up table - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 NormalizePalette(vec3 col, float levels) {
    return floor(col * levels + 0.5) / levels;
}
void main(void) {
    vec2 pixelSize = vec2(256.0, 240.0);
    vec2 coord = floor(tc * pixelSize) / pixelSize;
    vec4 texColor = texture(samp, coord);
    vec3 quantizedColor = NormalizePalette(texColor.rgb, 6.0);
    color = vec4(quantizedColor, texColor.a);
}

```

Shader #4 - 8bit

File: 8bit.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic.

Technique: Indexed color palette look-up table - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 NESPalette(vec3 col) {
    float[16] rVals = float[16](0.0, 0.26, 0.5, 0.74, 1.0, 0.88, 0.62, 0.38, 0.14, 0.08, 0.22, 0.54,
    ↪ 0.78, 0.92, 0.96, 0.1);
    float[16] gVals = float[16](0.0, 0.13, 0.25, 0.37, 0.5, 0.44, 0.31, 0.19, 0.08, 0.13, 0.37, 0.44,
    ↪ 0.56, 0.69, 0.82, 0.96);
    float[16] bVals = float[16](0.0, 0.06, 0.13, 0.25, 0.38, 0.44, 0.56, 0.69, 0.82, 0.96, 0.74, 0.5,
    ↪ 0.25, 0.13, 0.06, 0.0);
    int idxR = int(floor(col.r * 15.0 + 0.5));
    int idxG = int(floor(col.g * 15.0 + 0.5));
    int idxB = int(floor(col.b * 15.0 + 0.5));
    return vec3(rVals[idxR], gVals[idxG], bVals[idxB]);
}
void main(void) {
    vec2 pixelSize = vec2(256.0, 240.0);
    vec2 coord = floor(tc * pixelSize) / pixelSize;
    vec4 texColor = texture(samp, coord);
    vec3 quantizedColor = NESPalette(texColor.rgb);
    color = vec4(quantizedColor, texColor.a);
}
```

Shader #29 - analog

File: analog.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Scanline overlay using modular y-coordinate - Hermite smoothstep interpolation - Radial vignette falloff - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 analogEffect(vec2 uv) {
    vec3 col = texture(samp, uv).rgb;
    float offset = 0.002;
    col.r = texture(samp, uv + vec2(offset, 0.0)).r;
    col.b = texture(samp, uv - vec2(offset, 0.0)).b;
    float noise = fract(sin(dot(uv.xy + time_f, vec2(12.9898, 78.233)))) * 43758.5453;
    col += noise * 0.05;
    float scanline = sin(uv.y * iResolution.y * 3.0 + time_f * 20.0) * 0.05;
    col -= vec3(scanline);
    float vignette = smoothstep(1.0, 0.8, length(uv - 0.5) * 1.5);
    col *= vignette;
    return col;
}
void main(void) {
    vec2 uv = tc;
    vec3 col = analogEffect(uv);
}
```

```
    color = vec4(col, 1.0);
}
```

Shader #35 - atan glitch

File: atan-glitch.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position for interactive control. Convert mouse position to UV space Base rotation effect RGB channel separation

Technique: Multi-tap convolution blur (5 texture samples) - Color quantization via floor rounding - Scanline overlay using modular y-coordinate - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Distance field computation - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

float hash(float n) {
    return fract(sin(n) * 43758.5453);
}

void main(void) {
    vec2 mouseUV = iMouse.xy / iResolution.xy;
    float mouseInfluence = smoothstep(0.2, 0.1, distance(tc, mouseUV));
    float angle = atan(tc.y - 0.5, tc.x - 0.5);
    float modulatedTime = pingPong(time_f, 5.0);
    angle += modulatedTime;
    vec2 rotatedTC;
    rotatedTC.x = cos(angle) * (tc.x - 0.5) - sin(angle) * (tc.y - 0.5) + 0.5;
    rotatedTC.y = sin(angle) * (tc.x - 0.5) + cos(angle) * (tc.y - 0.5) + 0.5;
    vec2 glitchOffset = vec2(
        sin(time_f * 10.0 + tc.y * 50.0) * 0.02 * mouseInfluence,
        cos(time_f * 8.0 + tc.x * 40.0) * 0.02 * mouseInfluence
    );
    float rgbShift = 0.02 * mouseInfluence * sin(time_f * 15.0);
    vec4 shiftedColor = vec4(
        texture(samp, rotatedTC + glitchOffset + vec2(rgbShift, 0.0)).r,
        texture(samp, rotatedTC + glitchOffset).g,
        texture(samp, rotatedTC + glitchOffset - vec2(rgbShift, 0.0)).b,
        1.0
    );
    float scanLine = mod(gl_FragCoord.y * 0.5, 1.0) < 0.5 ? 0.8 : 1.0;
    scanLine *= 1.0 - (0.3 * mouseInfluence * sin(time_f * 20.0));
    color = shiftedColor * scanLine;
    float noise = hash(tc.x + tc.y + time_f) * 0.3 * mouseInfluence;
    color.rgb += noise;
    vec2 warpedCoords = mix(rotatedTC, rotatedTC + glitchOffset, mouseInfluence);
    color = mix(color, texture(samp, warpedCoords), 0.7);
    float blockSize = 50.0;
    vec2 blockCoord = floor(tc * blockSize) / blockSize;
    float blockGlitch = hash(blockCoord.x + blockCoord.y + floor(time_f * 5.0)) * 0.1 * mouseInfluence;
    color.rgb = texture(samp, warpedCoords + vec2(blockGlitch)).rgb;
}
```

```
}
```

Shader #52 - average pixels

File: average_pixels.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc+0.01);
    vec4 color3 = texture(samp, tc-0.01);
    vec2 pos = tc;
    pos[0] -= 0.01;
    vec2 pos2 = tc;
    pos2[1] += 0.01;
    vec4 color4 = texture(samp, pos);
    vec4 color5 = texture(samp, pos2);
    color = (color + color2 + color3 + color4 + color5) / 5;
}
```

Shader #65 - block pixels

File: block_pixels.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main() {
    float size = mod(time_f, 30);
    float pixelSize = 1.5 * size;
```

```

    vec2 pixelCount = iResolution / pixelSize;
    vec2 pixelatedTC = floor(tc * pixelCount) / pixelCount;
    color = texture(samp, pixelatedTC);
}

```

Shader #117 - color glitch dark

File: color_glitch_dark.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 rgb2hsb(vec3 c) {
    vec4 K = vec4(0.0, -1.0/3.0, 2.0/3.0, -1.0);
    vec4 p = mix(vec4(c.bg, K.wz), vec4(c.gb, K.xy), step(c.b, c.g));
    vec4 q = mix(vec4(p.xyw, c.r), vec4(c.r, p.yzx), step(p.x, c.r));
    float d = q.x - min(q.w, q.y);
    float e = 1.0e-10;
    return vec3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e), q.x);
}
vec3 hsb2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
void main() {
    vec2 uv = tc;
    vec4 texColor = texture(samp, uv);
    vec2 texelSize = 1.0 / iResolution;
    vec3 result = vec3(0.0);
    float weight[5];
    weight[0] = 0.227027;
    weight[1] = 0.1945946;
    weight[2] = 0.1216216;
    weight[3] = 0.054054;
    weight[4] = 0.016216;
    for (int i = -4; i <= 4; i++) {
        result += texture(samp, uv + vec2(float(i) * texelSize.x, 0.0)).rgb * weight[abs(i)];
    }
}

```

```

    for (int j = -4; j <= 4; j++) {
        result += texture(samp, uv + vec2(0.0, float(j) * texelSize.y)).rgb * weight[abs(j)];
    }
    vec3 hsb = rgb2hsb(result / 2.0);
    hsb.x += time_f * 0.1;
    hsb.y = 0.5;
    hsb.z = 0.9;
    vec3 rgb = hsb2rgb(hsb);
    color = vec4(rgb * 0.8 + 0.2, texColor.a);
    color = xor_RGB(sin(color * time_f) + 0.5, texture(samp, tc));
}

```

Shader #142 - comp inc

File: comp_inc.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Scanline overlay using modular y-coordinate - Hermite smoothstep interpolation - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 compositeEffect(vec2 uv) {
    float offset = 0.01;
    vec3 col;
    col.r = texture(samp, uv + vec2(offset, 0.0)).r;
    col.g = texture(samp, uv).g;
    col.b = texture(samp, uv - vec2(offset, 0.0)).b;
    float noise = fract(sin(dot(uv.xy, vec2(12.9898, 78.233))) * 43758.5453);
    col += noise * 0.05;
    float scanline = sin(uv.y * iResolution.y * 1.5) * 0.1;
    col -= scanline;
    float bleed = sin(uv.y * iResolution.y * 0.2 + time_f * 5.0) * 0.005;
    col.r += bleed * 0.002;
    col.b -= bleed * 0.002;
    return col;
}
vec3 aces(vec3 x){
    const mat3 a=mat3(0.59719,0.35458,0.04823,0.07600,0.90834,0.01566,0.02840,0.13383,0.83777);
    const mat3 b=mat3(1.60475,-0.53108,-0.07367,-0.10208,1.10813,-0.00605,-0.00327,-0.07276,1.07602);
    vec3 v=a*x;
    v=(v*(v+0.0245786)-0.000090537)*b;
    return clamp(v,0.0,1.0);
}
float hash12(vec2 p){return fract(sin(dot(p,vec2(127.1,311.7))))*43758.5453123);}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect,1.0);
    vec2 c = vec2(0.5);
    vec2 p = (tc - c) * ar;
    float r2 = dot(p,p);
    float k1 = 0.15;
    float k2 = 0.05;
    vec2 pd = p*(1.0 + k1*r2 + k2*r2*r2);
    vec2 uvd = pd/ar + c;

```



```

    vec2 dir = normalize(p + 1e-6);
    float ab = 0.0015 + 0.0015*sin(time_f*0.5);
    vec2 uvR = uvd + dir*ab;
    vec2 uvG = uvd;
    vec2 uvB = uvd - dir*ab;
    uvR = clamp(uvR,0.0,1.0);
    uvG = clamp(uvG,0.0,1.0);
    uvB = clamp(uvB,0.0,1.0);
    vec3 cR = compositeEffect(uvR);
    vec3 cG = compositeEffect(uvG);
    vec3 cB = compositeEffect(uvB);
    vec3 col = vec3(cR.r, cG.g, cB.b);
    vec2 ts = 1.0 / iResolution;
    float shAmt = 0.4;
    vec3 bsum = vec3(0.0);
    bsum += compositeEffect(uvd + vec2(1,1)*ts);
    bsum += compositeEffect(uvd + vec2(-1,1)*ts);
    bsum += compositeEffect(uvd + vec2(1,-1)*ts);
    bsum += compositeEffect(uvd + vec2(-1,-1)*ts);
    bsum *= 0.25;
    col += (col - bsum) * shAmt;
    float v = smoothstep(0.95, 0.3, length(p));
    col *= v;
    float gAmt = 0.03;
    float g = hash12(gl_FragCoord.xy + time_f*123.45) - 0.5;
    col += g * gAmt;
    col = aces(col);
    col = pow(col, vec3(1.0/2.2));
    color = vec4(clamp(col,0.0,1.0),1.0);
}

```

Shader #146 - compo

File: compo.glsl

Generates procedural noise patterns combined with sinusoidal distortion. A simple pseudo-random noise function based on texture coordinates Determine pixel size in texture coordinates Simulate chroma separation: offset the red and blue channels slightly.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Per-channel UV offset for chromatic aberration - Scanline overlay using modular y-coordinate - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co, vec2(12.9898,78.233))) * 43758.5453);
}
void main(void) {
    vec2 pixelSize = vec2(1.0) / iResolution;
    float displacement = sin(time_f + tc.y * 50.0) * 0.5 * pixelSize.x;
    vec2 offset = vec2(displacement, 0.0);
    float r = texture(samp, tc + offset).r;
    float g = texture(samp, tc).g;
    float b = texture(samp, tc - offset).b;
    vec3 col = vec3(r, g, b);
}

```

```

float scanline = 0.85 + 0.15 * sin(tc.y * iResolution.y * 3.14159);
col *= scanline;
float noise = (rand(tc * iResolution + time_f) - 0.5) * 0.1;
col += noise;
color = vec4(col, 1.0);
}

```

Shader #152 - composite vhs

File: composite_vhs.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Scanline overlay using modular y-coordinate - Linear mix/lerp blending - Distance field computation - Radial vignette falloff - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 p) {
    return fract(sin(dot(p, vec2(12.9898, 78.233))) * 43758.5453);
}
vec3 compositeEffect(vec2 uv) {
    vec2 cuv = uv * 2.0 - 1.0;
    float r2 = dot(cuv, cuv);
    cuv *= 1.0 + 0.25 * r2;
    uv = cuv * 0.5 + 0.5;
    float vJitter = (rand(vec2(time_f * 3.17, 0.0)) - 0.5) * 0.01;
    uv.y = clamp(uv.y + vJitter, 0.0, 1.0);
    uv.x += sin(uv.y * 180.0 + time_f * 4.0) * 0.002;
    float lineDrift = sin(time_f * 0.7) * 0.01;
    uv.y = clamp(uv.y + lineDrift, 0.0, 1.0);
    vec2 uvR = uv + vec2(0.0025, 0.0);
    vec2 uvB = uv - vec2(0.0025, 0.0);
    vec3 srcR = texture(samp, uvR).rgb;
    vec3 srcG = texture(samp, uv).rgb;
    vec3 srcB = texture(samp, uvB).rgb;
    vec3 col;
    col.r = srcR.r;
    col.g = srcG.g;
    col.b = srcB.b;
    vec3 ghost = texture(samp, uv + vec2(0.006, 0.0)).rgb;
    col = mix(col, ghost, 0.15);
    float scanline = 0.7 + 0.3 * sin(uv.y * iResolution.y * 3.14159);
    col *= scanline;
    float fineNoise = rand(uv * iResolution.xy + time_f * 37.0);
    float coarseNoise = rand(vec2(0.0, floor(uv.y * iResolution.y * 0.5) + time_f * 5.0));
    float noise = fineNoise * 0.04 + coarseNoise * 0.06;
    col += noise;
    float band = step(0.97, fract(uv.y * 80.0 + time_f * 0.5));
    col *= 1.0 - band * 0.5;
    float dist = distance(uv, vec2(0.5));
    float vignette = clamp(1.0 - dist * 1.4, 0.0, 1.0);
    col *= vignette;
    col = pow(col, vec3(0.85));
    return col;
}

```

```

}
void main(void) {
    vec2 uv = tc;
    vec3 col = compositeEffect(uv);
    color = vec4(col, 1.0);
}

```

Shader #153 - composite vhs flat

File: composite_vhs_flat.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Scanline overlay using modular y-coordinate - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 p) {
    return fract(sin(dot(p, vec2(12.9898, 78.233))) * 43758.5453);
}
vec3 compositeEffect(vec2 uv) {
    float vJitter = (rand(vec2(time_f * 3.17, 0.0)) - 0.5) * 0.01;
    uv.y = clamp(uv.y + vJitter, 0.0, 1.0);
    uv.x += sin(uv.y * 180.0 + time_f * 4.0) * 0.002;
    float lineDrift = sin(time_f * 0.7) * 0.01;
    uv.y = clamp(uv.y + lineDrift, 0.0, 1.0);
    vec2 uvR = uv + vec2(0.0025, 0.0);
    vec2 uvB = uv - vec2(0.0025, 0.0);
    vec3 srcR = texture(samp, uvR).rgb;
    vec3 srcG = texture(samp, uv).rgb;
    vec3 srcB = texture(samp, uvB).rgb;
    vec3 col;
    col.r = srcR.r;
    col.g = srcG.g;
    col.b = srcB.b;
    vec3 ghost = texture(samp, uv + vec2(0.006, 0.0)).rgb;
    col = mix(col, ghost, 0.15);
    float scanline = 0.7 + 0.3 * sin(uv.y * iResolution.y * 3.14159);
    col *= scanline;
    float fineNoise = rand(uv * iResolution.xy + time_f * 37.0);
    float coarseNoise = rand(vec2(0.0, floor(uv.y * iResolution.y * 0.5) + time_f * 5.0));
    float noise = fineNoise * 0.04 + coarseNoise * 0.06;
    col += noise;
    float band = step(0.97, fract(uv.y * 80.0 + time_f * 0.5));
    col *= 1.0 - band * 0.5;
    col = pow(col, vec3(0.85));
    return col;
}
void main(void) {
    vec2 uv = tc;
    vec3 col = compositeEffect(uv);
    color = vec4(col, 1.0);
}

```

Shader #215 - DigitalLight

File: DigitalLight.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control. Calculate overall intensity based on audio levels $\text{amp} = \text{smoothed volume}$, $\text{uamp} = \text{raw/instant volume}$ Calculate the distortion amount.

Technique: Multi-tap convolution blur (4 texture samples) - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
void main(void) {
    float intensity = clamp(amp * 0.5 + uamp * 1.0, 0.0, 3.0);
    float distAmt = 0.05 * intensity;
    vec2 uv = tc;
    float rR = sin(uv.y * 10.0 + time_f * 5.0 + seed) * distAmt;
    float rG = sin(uv.x * 12.0 + time_f * 4.0 + seed * 1.3) * distAmt;
    float rB = sin(uv.y * 8.0 + time_f * 6.0 + seed * 0.7) * distAmt;
    vec2 tcR = uv + vec2(rR, 0.0);
    vec2 tcG = uv + vec2(0.0, rG);
    vec2 tcB = uv + vec2(rB, rB * 0.5);
    vec4 texR = texture(samp, tcR);
    vec4 texG = texture(samp, tcG);
    vec4 texB = texture(samp, tcB);
    vec3 finalCol = vec3(texR.r, texG.g, texB.b);
    float alpha = texture(samp, tc).a;
    color = vec4(finalCol, alpha);
}
```

Shader #216 - DigitalLightStorm

File: DigitalLightStorm.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Responds to mouse position and audio input (amplitude/volume) for interactive control. Combined electrical fold with audio-reactive zoom 4 iterations to keep it sharp and efficient Use seed to vary the fractal folding structure slightly

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Indexed color palette look-up table - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
```

```

uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float seed = 1.0;
const float PI = 3.1415926535897932384626433832795;
vec4 xor_RGB(vec4 icolor, vec4 source){
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255.0);
    for(int i=0;i<3;++i){
        int_color[i] = int(255.0*icolor[i]);
        int_color[i] = int_color[i] ^ isource[i];
        if(int_color[i]>255) int_color[i] = int_color[i] % 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 electricalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 4; i++) {
        float iSeed = seed * 0.1 * float(i);
        p = abs((p - c) * (zoom + 0.1 * sin(t * 1.5 + float(i) + iSeed))) - 0.5 + c;
        p = rotateUV(p, t * 0.2 + float(i) * 0.1, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 electric = vec3(0.10, 0.90, 1.0);
    vec3 purple = vec3(0.60, 0.10, 1.0);
    float ph = fract(t * 0.4);
    vec3 k1 = mix(pink, electric, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(electric, purple, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(purple, pink, smoothstep(0.66, 1.00, ph));
    return normalize(k1 + k2 + k3) * 1.3;
}
vec3 limitHighlights(vec3 c){

```

```

    float m = max(c.r, max(c.g, c.b));
    if(m > 0.9) c *= 0.9 / m;
    return c;
}

void main(void) {
    seed = pingPong(time_f * PI, 15.0);
    float a = clamp(amp, 0.0, 1.0);
    float ua = clamp(uamp, 0.0, 1.0);
    float intensity = clamp(amp * 0.6 + uamp * 1.2, 0.0, 3.0);
    float tFast = time_f + intensity;
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    float distAmt = 0.02 * intensity;
    float rR = sin(uv.y * 10.0 + time_f * 5.0 + seed) * distAmt;
    float rG = sin(uv.x * 12.0 + time_f * 4.0 + seed * 1.3) * distAmt;
    float rB = sin(uv.y * 8.0 + time_f * 6.0 + seed * 0.7) * distAmt;
    vec2 tcR = uv + vec2(rR, 0.0);
    vec2 tcG = uv + vec2(0.0, rG);
    vec2 tcB = uv + vec2(rB, rB * 0.5);
    vec4 texR = texture(samp, tcR);
    vec4 texG = texture(samp, tcG);
    vec4 texB = texture(samp, tcB);
    vec4 baseTex = vec4(texR.r, texG.g, texB.b, 1.0);
    vec2 kUV = tc;
    float segs = 4.0 + floor(intensity * 2.0) * 2.0;
    kUV = reflectUV(kUV, segs, m, aspect);
    float zoom = 1.1 + 0.3 * sin(time_f) + 0.4 * intensity;
    vec2 electricUV = electricalFold(kUV, zoom, tFast * 0.3, m, aspect);
    float dist = length(electricUV - m);
    float bolt = 0.015 / (dist + 0.001);
    bolt = pow(bolt, 1.2);
    float pulse = pingPong(tFast * 4.0, 1.0);
    bolt *= (0.5 + 0.5 * pulse);
    bolt *= (0.8 + 0.5 * intensity);
    vec3 boltColor = neonPalette(time_f + dist * 2.0);
    vec4 interference = vec4(boltColor, 1.0) * bolt;
    vec4 xorLayer = xor_RGB(baseTex, interference * 2.5);
    vec3 finalCol = baseTex.rgb;
    finalCol = mix(finalCol, xorLayer.rgb, clamp(bolt * 0.6, 0.0, 0.8));
    finalCol += boltColor * bolt * 0.8;
    if (uamp > 0.8) {
        finalCol += vec3(0.1, 0.1, 0.2) * uamp;
    }
    finalCol = limitHighlights(finalCol);
    color = vec4(finalCol, 1.0);
}

```

Shader #507 - glitch no noise

File: glitch-no-noise.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0) + 2.0;
    vec2 uv = tc;
    float glitchStrength = 0.001;
    uv.x += (rand(uv + time_f) - 0.5) * glitchStrength;
    uv.y += (rand(uv + time_f * 1.5) - 0.5) * glitchStrength;
    vec4 texColor = blur(samp, uv, iResolution);
    vec4 colorShift = vec4(texColor.r, texColor.g * 0.5 + 0.5, texColor.b * 0.5 + 0.5, texColor.a);
    colorShift = sin(colorShift * time_t);
    float glitchNoise = rand(uv + time_f);
    vec4 glitchColor = vec4(vec3(glitchNoise), 1.0) * glitchStrength;
    color = mix(colorShift, glitchColor, glitchStrength * glitchNoise);
    color = sin(color * time_t);
    color.a = 1.0;
}

```

Shader #508 - glitch noise

File: glitch-noise.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
float noise(vec2 pos) {
    vec2 i = floor(pos);
    vec2 f = fract(pos);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(rand(i + vec2(0.0, 0.0)), rand(i + vec2(1.0, 0.0)), u.x),
        mix(rand(i + vec2(0.0, 1.0)), rand(i + vec2(1.0, 1.0)), u.x), u.y);
}
void main(void) {
    vec2 uv = tc;
    float glitchStrength = 0.01;
    float timeNoise = noise(uv * 10.0 + time_f * 0.5);
    uv.x += (rand(uv + time_f) - 0.5) * glitchStrength;
    uv.y += (rand(uv + time_f * 1.5) - 0.5) * glitchStrength;
    vec4 texColor = texture(samp, uv);
    vec4 colorShift = vec4(texColor.r, texColor.g * 0.5 + 0.5 * timeNoise, texColor.b * 0.5 + 0.5 * (1.0
    ↪ - timeNoise), texColor.a);
    float glitchNoise = rand(uv + time_f);
    vec4 glitchColor = vec4(vec3(sin(glitchNoise * time_f)), 1.0) * glitchStrength;
    color = mix(colorShift, glitchColor, glitchStrength * glitchNoise);
    color = sin(color * time_f);
    color.a = 1.0;
}
```

Shader #509 - glitch react color

File: glitch-react-color.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec2 glitchEffect(vec2 uv, float timeAmp) {
    float time = floor(timeAmp);
    float amp = fract(timeAmp);
    uv.y += amp * 0.1 * sin(time * 10.0 + uv.x * 100.0);
}
```



```

    uv.x += amp * 0.05 * sin(time * 5.0 + uv.y * 200.0);
    uv.x += rand(vec2(uv.y * time, uv.x)) * 0.1 * amp;
    uv.y += rand(vec2(uv.x * time, uv.y)) * 0.1 * amp;
    return uv;
}

void main(void) {
    vec2 uv = tc;
    uv = glitchEffect(uv, time_f);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = sin(texture(samp, uv) * time_t);
    color.a = 1.0;
}

```

Shader #510 - glitch react

File: glitch-react.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}

vec2 glitchEffect(vec2 uv, float timeAmp) {
    float time = floor(timeAmp);
    float amp = fract(timeAmp);
    uv.y += amp * 0.1 * sin(time * 10.0 + uv.x * 100.0);
    uv.x += amp * 0.05 * sin(time * 5.0 + uv.y * 200.0);
    uv.x += rand(vec2(uv.y * time, uv.x)) * 0.1 * amp;
    uv.y += rand(vec2(uv.x * time, uv.y)) * 0.1 * amp;
    return uv;
}

void main(void) {
    vec2 uv = tc;
    uv = glitchEffect(uv, time_f);
    color = texture(samp, uv);
}

```

Shader #511 - glitch zoom

File: glitch-zoom.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co){
    return fract(sin(dot(co.xy, vec2(12.9898,78.233))) * 43758.5453);
}
float pingPong(float x, float length){
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution){
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](
        0.5,1.0,1.5,2.0,2.5,2.5,2.0,1.5,1.0,0.5,
        1.0,2.0,2.5,3.0,3.5,3.5,3.0,2.5,2.0,1.0,
        1.5,2.5,3.0,3.5,4.0,4.0,3.5,3.0,2.5,1.5,
        2.0,3.0,3.5,4.0,4.5,4.5,4.0,3.5,3.0,2.0,
        2.5,3.5,4.0,4.5,5.0,5.0,4.5,4.0,3.5,2.5,
        2.5,3.5,4.0,4.5,5.0,5.0,4.5,4.0,3.5,2.5,
        2.0,3.0,3.5,4.0,4.5,4.5,4.0,3.5,3.0,2.0,
        1.5,2.5,3.0,3.5,4.0,4.0,3.5,3.0,2.5,1.5,
        1.0,2.0,2.5,3.0,3.5,3.5,3.0,2.5,2.0,1.0,
        0.5,1.0,1.5,2.0,2.5,2.5,2.0,1.5,1.0,0.5
    );
    for(int i=0;i<100;i++) kernel[i]=kernelVals[i];
    float kernelSum = 0.0;
    for(int i=0;i<100;i++) kernelSum += kernel[i];
    for(int x=-5;x<=4;++x){
        for(int y=-5;y<=4;++y){
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y+5)*10 + (x+5)];
        }
    }
    return result / kernelSum;
}
void main(void){
    float time_t = pingPong(time_f, 10.0) + 2.0;
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = vec2(0.5);
    vec2 p = (tc - m) * ar;
    float eps = 1e-6;
    float base = 1.65;
    float period = log(base);
    float t = time_f * 0.5;
    float r = length(p) + eps;
    float theta = atan(p.y, p.x) + t * 0.25 * 6.28318530718;
    float k = fract((log(r) - t) / period);
    float rw = exp(k * period);
    vec2 wrapped = vec2(cos(theta), sin(theta)) * rw;
    vec2 uv = wrapped / ar + m;
    uv = fract(uv);
    float glitchStrength = 0.001;
    vec2 uvGlitch = uv;
    uvGlitch.x += (rand(uv + time_f) - 0.5) * glitchStrength;
    uvGlitch.y += (rand(uv + time_f * 1.5) - 0.5) * glitchStrength;
    vec4 texColor = blur(samp, uvGlitch, iResolution);
    vec4 colorShift = vec4(texColor.r,
        texColor.g * 0.5 + 0.5,
        texColor.b * 0.5 + 0.5,
        texColor.a);
}

```

```

    colorShift = sin(colorShift * time_t);
    float glitchNoise = rand(uv + time_f);
    vec4 glitchColor = vec4(vec3(glitchNoise), 1.0) * glitchStrength;
    color = mix(colorShift, glitchColor, glitchStrength * glitchNoise);
    color = sin(color * time_t);
    color.a = 1.0;
}

```

Shader #512 - glitch1

File: glitch1.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to mouse position for interactive control. Horizontal distortion Vertical distortion Random seed based on uv and time

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Driven by: time, mouse, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
float random(vec2 st) {
    return fract(sin(dot(st.xy, vec2(12.9898, 78.233))) * 43758.5453123);
}
vec2 glitchOffset(vec2 uv, vec2 seed) {
    float strength = random(seed) * 0.1 - 0.05;
    if (random(seed * 2.0 + time_f) > 0.5) {
        uv.y += strength;
    } else {
        uv.x += strength;
    }
    return uv;
}
void main(void) {
    vec2 uv = tc;
    vec2 seed = floor(uv * 10.0) + vec2(time_f);
    vec2 glitch_uv = glitchOffset(uv, seed);
    vec4 texColor = texture(samp, glitch_uv);
    float glitch_strength = random(seed) * 0.3;
    if (random(seed + 1.0) > 0.7) {
        texColor.r += glitch_strength;
        texColor.g -= glitch_strength;
    }
    color = texColor;
}

```

Shader #513 - glitch boil

File: glitch_boil.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float random(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(random(i + vec2(0.0, 0.0)), random(i + vec2(1.0, 0.0)), u.x),
        mix(random(i + vec2(0.0, 1.0)), random(i + vec2(1.1, 1.0)), u.x), u.y);
}
float bubble(vec2 uv, vec2 position, float size, float time) {
    float dist = length(uv - position);
    float pop = smoothstep(size, size - 0.05, dist) * (1.0 - smoothstep(size - 0.05, size - 0.1, dist));
    return pop * (1.0 - smoothstep(0.0, 1.0, time));
}
void main(void) {
    vec2 uv = tc;
    float time = mod(time_f, 10.0);
    float boil = noise(uv * 10.0 + time * 2.0) * 0.1;
    boil += noise(uv * 15.0 + time * 3.0) * 0.05;
    boil += noise(uv * 20.0 + time * 5.0) * 0.02;
    vec2 distorted_uv = uv + boil * vec2(noise(uv + time), noise(uv + time + vec2(10.0, 10.0)));
    vec3 col = texture(samp, distorted_uv).rgb;
    float bubble1 = bubble(uv, vec2(0.5, 0.5), 0.1, time);
    float bubble2 = bubble(uv, vec2(0.3, 0.6), 0.08, time + 2.0);
    float bubble3 = bubble(uv, vec2(0.7, 0.4), 0.12, time + 4.0);
    col += vec3(bubble1 + bubble2 + bubble3) * 0.5;
    color = vec4(col, 1.0);
}
```

Shader #514 - glitch boil2

File: glitch_boil2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    float boil = sin(uv.y * 20.0 + time_f * 6.0) * 0.2 * time_f;
```

```

    boil += cos(uv.x * 30.0 + time_f * 4.0) * 0.2 * time_f;
    boil += sin((uv.x + uv.y) * 25.0 + time_f * 5.0) * 0.1 * time_f;
    uv.y += boil;
    uv.x += boil;
    float time_t = pingPong(time_f, 10.0);
    vec4 distorted = texture(samp, sin(uv * time_t));
    vec4 baseTexture = texture(samp, tc);
    color = mix(baseTexture, distorted, 0.5);
}

```

Shader #515 - glitch effect

File: glitch_effect.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
void main(void)
{
    float time = time_f * 0.2;
    vec2 glitchOffset = vec2(sin(time * 2.0) * 0.005, cos(time * 3.0) * 0.005);
    glitchOffset += vec2(random_value.x * 0.005, random_value.y * 0.005);
    vec2 tcOffset = tc + glitchOffset;
    vec4 baseColor = texture(samp, tcOffset);
    vec4 glitchColor = texture(samp, tc + glitchOffset * 2.0);
    color = mix(baseColor, glitchColor, 0.5);
    color.rgb *= vec3(1.0 + sin(time_f * 5.0) * 0.1, 1.0 + cos(time_f * 5.0) * 0.1, 1.0 + sin(time_f *
↵ 7.0) * 0.1);
}

```

Shader #516 - glitch jump

File: glitch_jump.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
void main(void) {
    vec2 pos = tc;
    float time_t = mod(time_f, 10.0);
    pos = pos + sin(value_alpha_r * time_t);
    color = texture(samp, pos);
}
```

Shader #517 - glitch light

File: glitch_light.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 random_var;
vec4 blendColors(float t) {
    vec4 color1 = vec4(0.6, 0.2, 0.8, 1.0);
    vec4 color2 = vec4(0.2, 0.8, 0.6, 1.0);
    vec4 color3 = vec4(0.8, 0.6, 0.2, 1.0);
    vec4 color4 = vec4(0.3, 0.4, 0.6, 1.0);
    float phase = mod(t / 20.0, 2.0 * 3.14159);
    float blend = (sin(phase) + 1.0) / 2.0;
    if (t < 25.0)
        return mix(color1, color2, blend);
    else if (t < 50.0)
        return mix(color2, color3, blend);
    else
        return mix(color3, color4, blend);
}
void main(void) {
    color = texture(samp, tc);
    float time_t = mod(time_f, 75.0);
    vec4 light = blendColors(time_t);
    color = tan(0.25 + sin((color * light) * time_t));
    color.a = 1.0;
}
```

Shader #518 - glitch wave

File: glitch_wave.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.
Texture coordinates Time variable for animation Screen resolution

Technique: Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 uv = tc;
    vec2 reflectedUV = vec2(uv.x, uv.y);
    float waveAmplitude = 0.5;
    float waveFrequency = 3.0;
    float waveSpeed = 2.0;
    reflectedUV.y += sin(reflectedUV.x * waveFrequency + time_f * waveSpeed) * waveAmplitude;
    reflectedUV = clamp(reflectedUV, 0.0, 1.0);
    vec4 reflectedColor = texture(samp, reflectedUV);
    color = reflectedColor;
}
```

Shader #519 - glitchf

File: glitchf.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Pseudo-random noise via fract(sin(dot()))
hash function - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float random(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
void main() {
    vec2 uv = tc;
    float glitchStrength = 0.05 + 0.03 * sin(time_f * 3.0);
    float glitchLine = step(0.5 + 0.1 * sin(time_f * 10.0), fract(uv.y * 10.0 + time_f * 5.0));
    float displacement = glitchStrength * (random(uv + time_f) - 0.5);
    vec2 distortedUV = uv;
    distortedUV.x += displacement * glitchLine;
    vec4 texColor = texture(samp, distortedUV);
    vec2 offset = vec2(0.01 * sin(time_f * 15.0), 0.01 * cos(time_f * 10.0));
    vec3 glitchRGB = vec3(
        texture(samp, distortedUV + vec2(offset.x, 0.0)).r,
        texture(samp, distortedUV + vec2(-offset.x, offset.y)).g,
        texture(samp, distortedUV + vec2(0.0, -offset.y)).b
    );
    float strobe = step(0.5, fract(time_f * 10.0)) * 0.2;
    color = vec4(mix(texColor.rgb, glitchRGB, glitchLine + strobe), texColor.a);
}
```

Shader #520 - glitchy rainbow

File: glitchy-rainbow.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
- Linear mix/lerp blending - Driven by: time

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec4 getColor(void) {
    vec4 ctx = texture(samp, tc);
    float adjusted_time = time_f * 0.1;
    float hue = mod(tc.x + adjusted_time, 1.0);
    vec3 hsvColor = vec3(hue, 1.0, 1.0);
    vec3 smoothColor = hsv2rgb(hsvColor);
    ctx.rgb = mix(ctx.rgb, smoothColor, 0.5);
    return ctx;
}
void main(void)
{
    vec4 color1 = getColor();
    vec2 uv = tc * 2.0 - 1.0;
    float t = time_f * 0.5;
    float glitchX = sin(uv.y * 10.0 + t) * 0.1;
    float glitchY = cos(uv.x * 10.0 + t) * 0.1;
    vec2 glitchUV = vec2(uv.x + glitchX, uv.y + glitchY);
    float r = length(glitchUV);
    float a = atan(glitchUV.y, glitchUV.x);
    float radius = sin(t + r * 5.0) * 0.5 + 0.5;
    float angle = a + t + sin(r * 20.0 + t) * 0.1;
    vec2 distorted_uv = vec2(cos(angle), sin(angle)) * radius + 0.5;
    vec3 col = texture(samp, distorted_uv).rgb;
    col = mix(col, color1.rgb, 0.8);
    col = mix(col, vec3(1.0, 0.0, 0.0), 0.5 * sin(t + length(uv) * 5.0));
    color = vec4(col, 1.0);
}
```

Shader #521 - glitchy squish

File: glitchy-squish.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time

```
#version 330 core
in vec2 tc;
out vec4 color;
```



```

uniform sampler2D samp;
uniform float time_f;
void main(void)
{
    vec2 uv = tc * 2.0 - 1.0;
    float t = time_f * 0.5;
    float glitchX = sin(uv.y * 10.0 + t) * 0.1;
    float glitchY = cos(uv.x * 10.0 + t) * 0.1;
    vec2 glitchUV = vec2(uv.x + glitchX, uv.y + glitchY);
    float r = length(glitchUV);
    float a = atan(glitchUV.y, glitchUV.x);
    float radius = sin(t + r * 5.0) * 0.5 + 0.5;
    float angle = a + t + sin(r * 20.0 + t) * 0.1;
    vec2 distorted_uv = vec2(cos(angle), sin(angle)) * radius + 0.5;
    vec3 col = texture(samp, distorted_uv).rgb;
    col = mix(col, vec3(1.0, 0.0, 0.0), 0.5 * sin(t + length(uv) * 5.0));
    color = vec4(col, 1.0);
}

```

Shader #522 - glitchy

File: glitchy.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float glitchFactorX = sin(time_f * 10.0 + uv.y * 20.0) * 0.1;
    float glitchFactorY = cos(time_f * 7.0 + uv.x * 15.0) * 0.1;
    uv.x += glitchFactorX;
    uv.y += glitchFactorY;
    float glitchLine = fract(sin(dot(uv * time_f, vec2(12.9898, 78.233))) * 43758.5453);
    if (glitchLine < 0.1) {
        uv.x += sin(time_f * 50.0) * 0.02;
    } else if (glitchLine < 0.2) {
        uv.y += cos(time_f * 50.0) * 0.02;
    }
    float glitchBlock = fract(sin(dot(uv * time_f, vec2(39.3468, 11.135))) * 23454.2345);
    if (glitchBlock < 0.15) {
        uv.x = mod(uv.x * 5.0, 1.0);
        uv.y = mod(uv.y * 5.0, 1.0);
    }
    vec2 uvOffset = vec2(sin(time_f * 20.0 + uv.y * 30.0), cos(time_f * 20.0 + uv.x * 30.0)) * 0.03;
    uv += uvOffset;
    color = texture(samp, uv);
}

```

Shader #725 - new glitch

File: new_glitch.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
void main(void) {
    vec2 pos = tc;
    vec2 randomOffset = vec2(rand(pos + time_f), rand(pos - time_f));
    pos.x += (value_alpha_r * randomOffset.x - 0.5) * 0.05;
    pos.y += (value_alpha_g * randomOffset.y - 0.5) * 0.05;
    vec4 texColor = texture(samp, pos);
    vec4 shiftedColor1 = texture(samp, pos + vec2(value_alpha_b * 0.01, 0.0));
    vec4 shiftedColor2 = texture(samp, pos + vec2(0.0, value_alpha_b * 0.01));
    color = mix(texColor, (shiftedColor1 + shiftedColor2) * 0.5, 0.5);
}
```

Shader #761 - pixels

File: pixels.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    vec2 pixelSize = vec2(0.01, 0.01);
    uv = floor(uv / pixelSize) * pixelSize;
    color = texture(samp, uv);
}
```

Shader #841 - random pixels static

File: random_pixels_static.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
```

```

uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
uniform float restore_black;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec2 pos1 = tc;
    vec2 pos2 = tc;
    vec4 iv1 = inc_valuex/255;
    vec4 iv2 = inc_value/255;
    pos1[0] = tc[0]-iv1[0];
    pos1[1] = tc[1]-iv1[1];
    pos2[0] = tc[0]-iv2[0];
    pos2[1] = tc[1]-iv2[1];
    vec2 pos3 = tc;
    pos3[0] = tc[0]-iv1[2];
    pos3[1] = tc[1]-iv2[2];
    float rand_x = random(pos1);
    float rand_y = random(pos2);
    float rand_z = random(pos3);
    color = (0.5 * color) + (0.5 * vec4(rand_x, rand_y, rand_z, 1));
}

```

Shader #857 - react16

File: react16.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normPos = (gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0;
    float dist = length(normPos);
    float phase = sin(dist * 10.0 - time_f * 4.0);
    vec2 tcAdjusted = tc + (normPos * 0.305 * phase);
}

```

```

    vec4 color1 = texture(samp, tcAdjusted);
    vec2 centeredCoord = (tc * 2.0) - vec2(1.0, 1.0);
    float stretchFactor = 1.0 + (1.0 - abs(centeredCoord.y)) * 0.5;
    centeredCoord.x *= sin(stretchFactor * time_f);
    vec2 stretchedCoord = (centeredCoord + vec2(1.0, 1.0)) / 2.0;
    vec4 color2 = texture(samp, stretchedCoord);
    float glitchFactor = sin(time_f * 50.0);
    vec4 glitchColor = texture(samp, tc + vec2(glitchFactor * 0.1, glitchFactor * 0.1));
    color = mix(mix(color1, color2, 0.5), glitchColor, 0.5);
}

```

Shader #858 - react17

File: react17.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 normPos = gl_FragCoord.xy / iResolution.xy;
    float cycle = sin(time_f * 0.5);
    float movingPhase = normPos.x + cycle;
    float phase = sin(movingPhase * 10.0 - time_f * 2.0);
    vec2 tcAdjusted = tc + (vec2(phase, 0) * 0.302);
    float glitchFactor = sin(time_f);
    vec2 glitchOffset = vec2(glitchFactor * 0.1, glitchFactor * 0.1);
    vec4 glitchColor = texture(samp, tc + glitchOffset);
    vec4 baseColor = texture(samp, tcAdjusted);
    color = mix(baseColor, glitchColor, 0.5);
}

```

Shader #859 - react18

File: react18.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float glitchStrength = 0.05;

```

```

    vec2 glitch = vec2(
        pingPong(time_f * 10.0 + tc.y * 20.0, 1.0) * glitchStrength,
        pingPong(time_f * 10.0 + tc.x * 20.0, 1.0) * glitchStrength
    );
    vec2 displacedTc = tc + glitch;
    color = texture(samp, displacedTc);
}

```

Shader #864 - react5

File: react5.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Linear mix/lerp blending - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec4 baseColor = texture(samp, tc);
    float glitchOffsetX = sin(time_f * 10.0 + tc.y * 20.0) * 0.05;
    float glitchOffsetY = cos(time_f * 15.0 + tc.x * 25.0) * 0.05;
    vec2 glitchTc = tc + vec2(glitchOffsetX, glitchOffsetY);
    vec4 glitchColor = texture(samp, glitchTc);
    float glitchStrength = 0.5 + 0.5 * sin(time_f * 3.0);
    color = mix(baseColor, glitchColor, glitchStrength);
}

```

Shader #865 - react6

File: react6.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec4 baseColor = texture(samp, tc);
    float glitchOffsetX = fract(sin(dot(tc, vec2(12.9898, 78.233))) * 43758.5453) * 0.1 * sin(time_f *
↵ 20.0);
    float glitchOffsetY = fract(cos(dot(tc, vec2(4.898, 7.23))) * 23421.6312) * 0.1 * cos(time_f * 20.0)
↵ ;
    vec2 glitchTc = tc + vec2(glitchOffsetX, glitchOffsetY);
    vec4 glitchColor = texture(samp, glitchTc);
    float glitchStrength = 0.5 + 0.5 * cos(time_f * 5.0);
    color = mix(baseColor, glitchColor, glitchStrength);
}

```

Shader #867 - react8

File: react8.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc;
    float glitchStrength = sin(time_f * 10.0) * 0.1;
    float glitchOffsetX = fract(sin(dot(uv, vec2(12.9898, 78.233))) * 43758.5453) * glitchStrength;
    float glitchOffsetY = fract(cos(dot(uv, vec2(4.898, 7.23))) * 23421.6312) * glitchStrength;
    uv.x += glitchOffsetX;
    uv.y += glitchOffsetY;
    vec4 colorA = texture(samp, uv);
    vec4 colorB = texture(samp, uv + vec2(0.01 * sin(time_f * 50.0), 0.01 * cos(time_f * 50.0)));
    float noise = fract(sin(dot(uv, vec2(12.9898, 78.233))) * 43758.5453);
    color = mix(colorA, colorB, noise);
}
```

Shader #951 - smooth pixel

File: smooth_pixel.glsl

Blends multiple texture samples together using interpolation for a composite effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
void main(void) {
    vec4 c0 = texture(samp, tc);
    float lum = dot(c0.rgb, vec3(0.2126, 0.7152, 0.0722));
    float threshold = 0.7;
    float spread = 1.5;
    float k = smoothstep(threshold - 0.02, threshold + 0.02, lum);
    float strength = clamp(amp + uamp, 0.0, 1.0) * k;
    vec2 texel = 1.0 / iResolution;
    vec2 o = texel * spread;
    vec4 cN = texture(samp, tc + vec2(0.0, o.y));
    vec4 cS = texture(samp, tc + vec2(0.0, -o.y));
    vec4 cE = texture(samp, tc + vec2(o.x, 0.0));
    vec4 cW = texture(samp, tc + vec2(-o.x, 0.0));
}
```

```

    vec4 cNE = texture(samp, tc + vec2( o.x,  o.y));
    vec4 cNW = texture(samp, tc + vec2(-o.x,  o.y));
    vec4 cSE = texture(samp, tc + vec2( o.x, -o.y));
    vec4 cSW = texture(samp, tc + vec2(-o.x, -o.y));
    vec4 tent = (c0 * 4.0 +
                 (cN + cS + cE + cW) * 2.0 +
                 (cNE + cNW + cSE + cSW)) / 16.0;
    color = mix(c0, tent, strength);
}

```

Shader #1037 - vhs palette

File: vhs-palette.glsl

Processes the input texture to produce the "vhs palette" visual effect.

Technique: Direct texture color manipulation with arithmetic operations

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
void main(void) {
    vec4 texColor = texture(samp, tc);
    vec4 vhsColor = texColor;
    vhsColor.rgb = vhsColor.rgb * vec3(0.9, 0.85, 0.8) + vec3(0.1, 0.1, 0.15);
    vhsColor.rgb = clamp(vhsColor.rgb, 0.0, 1.0);
    color = vhsColor;
}

```

Shader #1038 - vhs

File: vhs.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (4 texture samples) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float random(float x) {
    return fract(sin(x) * 43758.5453123);
}
void main(void) {
    vec2 uv = tc;
    float y = uv.y * iResolution.y;
    float offset = sin(y * 10.0 + time_f * 5.0) * 0.005;
    if (random(y + time_f * 10.0) < 0.1) {
        offset += (random(y) - 0.5) * 0.02;
    }
    uv.x += offset;
    vec4 texColor = texture(samp, uv);
    float r = texture(samp, uv + vec2(0.002, 0.0)).r;
}

```

```

float g = texture(samp, uv).g;
float b = texture(samp, uv - vec2(0.002, 0.0)).b;
vec4 compositeColor = vec4(r, g, b, texColor.a);
vec4 vhsColor = mix(texColor, compositeColor, 0.5);
vhsColor.rgb = vhsColor.rgb * vec3(0.9, 0.85, 0.8) + vec3(0.1, 0.1, 0.15);
vhsColor.rgb = clamp(vhsColor.rgb, 0.0, 1.0);
color = vhsColor;
}

```

Shader #1039 - vhs2

File: vhs2.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Computes luminance for brightness-based effects.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Per-channel UV offset for chromatic aberration - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 p){return fract(sin(dot(p,vec2(12.9898,78.233)))*43758.5453);}
float nrnd(vec2 p){return rand(p)*2.0-1.0;}
vec3 lumaChroma(vec3 c){
    float y = dot(c, vec3(0.299,0.587,0.114));
    vec2 cbcr = vec2((c.b - y)*0.565, (c.r - y)*0.713);
    return vec3(y, cbcr);
}
vec3 yc2rgb(vec3 yc){
    float y=yc.x, cb=yc.y, cr=yc.z;
    float r = y + 1.403*cr;
    float g = y - 0.344*cb - 0.714*cr;
    float b = y + 1.770*cb;
    return vec3(r,g,b);
}
vec3 chromaLowpass(sampler2D s, vec2 uv, vec2 px, float k){
    vec2 o = vec2(px.x*1.5, 0.0);
    vec3 c0 = texture(s, uv - o).rgb;
    vec3 c1 = texture(s, uv).rgb;
    vec3 c2 = texture(s, uv + o).rgb;
    vec3 m = (c0 + c1 + c2)/3.0;
    vec3 yc = lumaChroma(m);
    yc.yz *= k;
    return yc2rgb(yc);
}
void main(){
    vec2 uv = tc;
    float t = time_f;
    vec2 res = iResolution;
    vec2 px = 1.0 / res;
    float weave = 0.002*(sin(t*0.97)+sin(t*1.31+uv.y*3.3));
    float lineWobble = 0.0008*sin(uv.y*120.0 + t*40.0);
    uv.x += weave + lineWobble;
    float holdTrig = step(0.995, rand(vec2(floor(t*1.7), 0.123)));
    uv.y = fract(uv.y + holdTrig * rand(vec2(t,0.37)) * 0.18);
}

```



```

float tearY = 0.84 + 0.04*rand(vec2(floor(t*0.6), 4.2));
float tearBand = smoothstep(tearY-0.015, tearY, uv.y) * (1.0 - smoothstep(tearY, tearY+0.05, uv.y));
float tear = tearBand * (0.08*nrand(vec2(uv.y*900.0, t*150.0)) + 0.03*sin(t*200.0));
uv.x += tear;
vec2 jitter = vec2(0.0015*nrand(vec2(t*3.0, uv.y*7.0)), 0.0);
uv += jitter;
float scanRate = 800.0;
float line = floor(uv.y * res.y);
float field = mod(line + floor(t*60.0), 2.0);
float scanDark = mix(0.92, 0.78, field);
float fineScan = 0.06*sin(uv.y*scanRate + t*12.0);
float coarse = 0.03*sin(uv.y*6.28318*3.0 + t*2.0);
float cx = (0.003 + 0.002*rand(vec2(line, t)));
vec3 ca;
ca.r = texture(samp, uv + vec2(cx,0.0)).r;
ca.g = texture(samp, uv).g;
ca.b = texture(samp, uv - vec2(cx,0.0)).b;
vec3 bleed = chromaLowpass(samp, uv, px, 0.65);
vec3 base = mix(ca, bleed, 0.45);
vec3 hl = texture(samp, uv + vec2(px.x*0.75,0.0)).rgb;
vec3 hr = texture(samp, uv - vec2(px.x*0.75,0.0)).rgb;
base = mix(base, (base+hl+hr)/3.0, 0.35);
float dropout = step(0.988, rand(vec2(line*0.73, floor(t*60.0)))) * smoothstep(0.0, 0.4, rand(vec2(
↳ line, t*0.23)));
vec3 dropCol = vec3(0.15 + 0.85*rand(vec2(line*3.1, t)));
base = mix(base, dropCol, dropout);
float snow = 0.12*nrand(vec2(uv*res*1.3 + t*80.0)) + 0.08*nrand(vec2(uv*res*3.7 - t*50.0));
base += snow;
float track = step(0.9975, rand(vec2(line*0.11 + floor(t*25.0), 9.9)));
base += track * (0.25 + 0.25*rand(vec2(line, t)));
float hueDrift = 0.02*sin(t*0.7) + 0.02*sin(t*1.3 + uv.y*5.0);
base.gb += vec2(hueDrift, -hueDrift);
float flicker = 0.94 + 0.06*rand(vec2(floor(t*12.0)));
base *= flicker;
float vign = smoothstep(0.0, 0.25, uv.y) * smoothstep(1.0, 0.75, uv.y);
float edgeX = smoothstep(0.0, 0.05, uv.x) * smoothstep(1.0, 0.95, uv.x);
float edge = mix(vign, vign*edgeX, 0.6);
base *= mix(0.92, 1.0, edge);
base *= scanDark;
base -= fineScan + coarse;
base = clamp(base, 0.0, 1.0);
color = vec4(pow(base, vec3(1.05)), 1.0);
}

```

Shader #1040 - vhs damage

File: vhs_damage.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Random function for noise effects Chromatic aberration effect Tape stretch/wobble

Technique: Multi-tap convolution blur (5 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Per-channel UV offset for chromatic aberration - Scanline overlay using modular y-coordinate - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898,78.233))) * 43758.5453);
}
vec3 chromaticAberration(vec2 uv, float intensity) {
    float r = texture(samp, uv + vec2(intensity, 0.0)).r;
    float g = texture(samp, uv).g;
    float b = texture(samp, uv - vec2(intensity, 0.0)).b;
    return vec3(r, g, b);
}
void main(void) {
    vec2 uv = tc;
    uv.y += sin(time_f * 2.0 + uv.x * 20.0) * 0.005;
    float vjitter = (rand(vec2(time_f)) - 0.5) * 0.02;
    uv.y = mod(uv.y + vjitter, 1.0);
    uv.x += sin(time_f * 10.0) * 0.005;
    float scanLine = fract(time_f * 0.5);
    if(abs(uv.y - scanLine) < 0.005 + rand(vec2(time_f)) * 0.01) {
        uv.x += (rand(vec2(uv.y + time_f)) - 0.5) * 0.1;
    }
    vec3 col = chromaticAberration(uv, 0.003 + rand(uv) * 0.003);
    float noiseIntensity = 0.3;
    vec3 noise = vec3(rand(uv * time_f)) * noiseIntensity;
    col += noise;
    float scanLines = sin(uv.y * 800.0 + time_f * 10.0) * 0.1;
    col -= scanLines * 0.1;
    vec2 bleedOffset = vec2(0.002 + sin(time_f) * 0.001, 0.0);
    col.r = texture(samp, uv + bleedOffset).r;
    col.b = texture(samp, uv - bleedOffset).b;
    if(rand(vec2(time_f * 0.1, uv.y)) > 0.99) {
        col *= 0.1 + rand(uv) * 0.5;
    }
    col = mix(col, col.grb, sin(time_f) * 0.1);
    col *= 0.9 + 0.1 * rand(vec2(time_f * 0.5));
    float trackLines = step(0.995, rand(vec2(uv.y * 100.0, time_f)));
    col += trackLines * 0.3;
    color = vec4(col, 1.0);
    color *= smoothstep(0.0, 0.1, uv.y) * smoothstep(1.0, 0.9, uv.y);
}

```

Fractal & Mandelbrot (33 shaders)

Shader #235 - drain mandella

File: drain_mandella.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {

```

```

float loopDuration = 25.0;
float t = mod(time_f, loopDuration);
vec2 aspect = vec2(iResolution.x / iResolution.y, 1.0);
vec2 nc = (tc * 2.0 - 1.0) * aspect;
nc.x = abs(nc.x);
float d = length(nc);
float a = atan(nc.y, nc.x);
float spiralSpeed = 5.0;
float inward = t / loopDuration;
a += (1.0 - smoothstep(0.0, 8.0, d)) * t * spiralSpeed;
d *= 1.0 - inward;
vec2 spiral = vec2(cos(a), sin(a)) * tan(d);
vec2 uv0 = (spiral / aspect + 1.0) * 0.5;
vec2 p = (uv0 * 2.0 - 1.0) * aspect;
float r = length(p);
float ang = atan(p.y, p.x);
float N = 10.0;
float tau = 6.28318530718;
float sector = tau / N;
ang = mod(ang + 0.5 * sector, sector);
ang = abs(ang - 0.5 * sector);
float ringFreq = 6.0;
float ring = fract(r * ringFreq + 0.15 * sin(time_f * 0.5));
float ringMirror = abs(ring - 0.5) * 2.0;
float swirl = 0.25 * sin(time_f * 0.3);
ang += swirl * r;
float zoom = 0.85 + 0.1 * sin(time_f * 0.27);
vec2 m = vec2(cos(ang), sin(ang)) * (r * zoom * (0.85 + 0.15 * ringMirror));
vec2 uv = (m / aspect + 1.0) * 0.5;
vec2 px = 1.0 / iResolution;
vec3 c = texture(samp, uv).rgb;
c += texture(samp, uv + vec2(px.x, 0)).rgb;
c += texture(samp, uv + vec2(-px.x, 0)).rgb;
c += texture(samp, uv + vec2(0, px.y)).rgb;
c += texture(samp, uv + vec2(0, -px.y)).rgb;
c *= 0.2;
float glow = smoothstep(0.95, 0.2, r) * (0.6 + 0.4 * ringMirror);
vec3 base = c * (0.7 + 0.3 * glow);
float hue = fract(ang / tau + 0.5 + 0.03 * time_f);
float rC = clamp(abs(hue * 6.0 - 3.0) - 1.0, 0.0, 1.0);
float gC = clamp(2.0 - abs(hue * 6.0 - 2.0), 0.0, 1.0);
float bC = clamp(2.0 - abs(hue * 6.0 - 4.0), 0.0, 1.0);
vec3 rainbow = vec3(rC, gC, bC);
float rbAmt = 0.35 * smoothstep(0.15, 0.85, ringMirror);
vec3 outCol = mix(base, base * rainbow, rbAmt);
float vign = smoothstep(1.2, 0.4, length((tc - 0.5) * aspect));
outCol *= vign;
color = vec4(outCol, 1.0);
}

```

Shader #383 - frac shader02 dmd mandella

File: frac_shader02_dmd_mandella.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Responds to mouse position and audio input (amplitude/volume) and random seed value for interactive control. Computes luminance for brightness-based effects. Spiral / sector warp from the extra shader, amplitude-aware. Base coordinates from original screen Ripple + twist (driven by amplitude)

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion

- Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float seed;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
```

```

    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv, float tSlow, float tFast, float aspect) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 gdir = normalize(vec2(cos(tSlow * 0.27), sin(tSlow * 0.31)));
    float s = dot((uv - 0.5) * vec2(aspect, 1.0), gdir);
    float w = max(fwidth(s) * 4.0, 0.002);
    float band = smoothstep(-0.5 - w, -0.5 + w, sin(s * 2.2 + tFast * 0.9));
    vec3 neon = neonPalette(tFast);
    vec3 grad = mix(tex, mix(tex, neon, 0.6), 0.35 + 0.25 * band);
    grad = mix(grad, tex, 0.10);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 rot2(vec2 v, float a) {
    float c = cos(a), s = sin(a);
    return vec2(c * v.x - s * v.y, s * v.x + c * v.y);
}

vec2 h2(vec2 p) {
    return fract(sin(vec2(dot(p, vec2(127.1, 311.7)),
                        dot(p, vec2(269.5, 183.3))))) * 43758.5453);
}

vec3 limitHighlights(vec3 c) {
    float m = max(c.r, max(c.g, c.b));

```

```

    if (m > 0.9) c *= 0.9 / m;
    return c;
}
vec2 spiralWarp(vec2 tcIn, float tBase, float a01, out float ringMirrorOut) {
    float loopDuration = 25.0;
    float t = mod(tBase * (0.6 + 1.8 * a01), loopDuration);
    vec2 aspect2 = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 nc = (tcIn * 2.0 - 1.0) * aspect2;
    nc.x = abs(nc.x);
    float d = length(nc);
    float a = atan(nc.y, nc.x);
    float spiralSpeed = 5.0 * (0.7 + 1.5 * a01);
    float inward = (t / loopDuration) * (0.5 + 0.5 * a01);
    a += (1.0 - smoothstep(0.0, 8.0, d)) * t * spiralSpeed;
    d *= 1.0 - inward;
    vec2 spiral = vec2(cos(a), sin(a)) * tan(d);
    vec2 uv0 = (spiral / aspect2 + 1.0) * 0.5;
    vec2 p = (uv0 * 2.0 - 1.0) * aspect2;
    float r = length(p);
    float ang = atan(p.y, p.x);
    float N = mix(8.0, 16.0, a01);
    float tau = 6.28318530718;
    float sector = tau / N;
    ang = mod(ang + 0.5 * sector, sector);
    ang = abs(ang - 0.5 * sector);
    float ringFreq = 6.0 + 4.0 * a01;
    float ring = fract(r * ringFreq + 0.15 * sin(tBase * 0.5));
    float ringMirror = abs(ring - 0.5) * 2.0;
    float swirl = 0.25 * sin(tBase * 0.3) * (0.5 + 1.0 * a01);
    ang += swirl * r;
    float zoom = 0.85 + 0.1 * sin(tBase * 0.27) + 0.08 * a01;
    vec2 m = vec2(cos(ang), sin(ang)) * (r * zoom * (0.85 + 0.15 * ringMirror));
    vec2 uv = (m / aspect2 + 1.0) * 0.5;
    ringMirrorOut = ringMirror;
    return uv;
}
void main(void) {
    float a = clamp(amp, 0.0, 1.0);
    float ua = clamp(uamp, 0.0, 1.0);
    float aMix = clamp(amp * 0.7 + uamp * 1.3, 0.0, 4.0);
    float a01 = clamp(aMix / 2.5, 0.0, 1.0);
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    float tBase = time_f;
    float tSlow = tBase * mix(0.2, 0.9, a01);
    float tFast = tBase * mix(0.7, 3.5, a01);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution)
        : fract(vec2(0.37 + 0.11 * sin(tSlow * 0.63 + seed),
            0.42 + 0.13 * cos(tSlow * 0.57 + seed * 2.0)));
    vec2 tc0 = tc;
    vec2 center = vec2(0.5, 0.5);
    float radius = length(tc0 - center);
    float rippleSpeed = 5.0 * (1.0 + 2.0 * a01);
    float rippleAmplitude = 0.03 * (0.5 + 1.5 * a01);
    float rippleWavelength = 10.0;
    float twistStrength = 1.0 + 4.0 * a01;
    float ripple = sin(tc0.x * rippleWavelength + tFast * rippleSpeed) * rippleAmplitude;
    ripple += sin(tc0.y * rippleWavelength + tFast * rippleSpeed) * rippleAmplitude;
    vec2 rippleTC = tc0 + vec2(ripple, ripple);
    float angleTwist = twistStrength * (radius - 1.0) + tSlow;
    float cosA = cos(angleTwist);
    float sinA = sin(angleTwist);

```

```

mat2 rotationMatrix = mat2(cosA, -sinA, sinA, cosA);
vec2 twistedTC = rotationMatrix * (tc0 - center) + center;
float mixRT = 0.5 + 0.35 * a01;
vec2 tcRT = mix(rippleTC, twistedTC, mixRT);
float ringMirrorSpiral;
vec2 spiralTC = spiralWarp(tcRT, tBase, a01, ringMirrorSpiral);
vec4 baseTex = texture(samp, spiralTC);
vec2 uvForGlow = spiralTC * 2.0 - 1.0;
uvForGlow.x *= aspect;
float rGlow = pingPong(sin(length(uvForGlow) * tSlow), 5.0);
float radiusMax = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radiusMax, radiusMax - 0.25, rGlow);
glow *= (0.8 + 0.4 * ringMirrorSpiral);
vec2 uv = spiralTC;
float speedScale = 1.0 + 2.0 * a01 + 3.0 * ua;
float speedR = 5.0 * speedScale;
float speedG = 6.5 * speedScale;
float speedB = 4.0 * speedScale;
float ampR = 0.03 * (0.7 + 0.6 * a01);
float ampG = 0.025 * (0.7 + 0.6 * a01);
float ampB = 0.035 * (0.7 + 0.6 * a01);
float waveR = 10.0;
float waveG = 12.0;
float waveB = 8.0;
float rR = sin(uv.x * waveR + tFast * speedR) * ampR
+ sin(uv.y * waveR * 0.8 + tFast * speedR * 1.2) * ampR;
float rG = sin(uv.x * waveG * 1.5 + tFast * speedG) * ampG
+ sin(uv.y * waveG * 0.3 + tFast * speedG * 0.7) * ampG;
float rB = sin(uv.x * waveB * 0.5 + tFast * speedB) * ampB
+ sin(uv.y * waveB * 1.7 + tFast * speedB * 1.3) * ampB;
vec2 tcR = uv + vec2(rR, rR);
vec2 tcG = uv + vec2(rG, -0.5 * rG);
vec2 tcB = uv + vec2(0.3 * rB, rB);
vec3 pats[4] = vec3[](vec3(1,0,1), vec3(0,1,0), vec3(1,0,0), vec3(0,0,1));
float pspd = 4.0;
int pidx = int(mod(floor(tBase * pspd + seed * 4.0), 4.0));
vec3 mir = pats[pidx];
vec2 dR = tcR - m;
vec2 dG = tcG - m;
vec2 dB = tcB - m;
float fallR = smoothstep(0.55, 0.0, length(dR));
float fallG = smoothstep(0.55, 0.0, length(dG));
float fallB = smoothstep(0.55, 0.0, length(dB));
float sw = (0.12 + 0.38 * ua + 0.25 * a);
vec2 tangR = rot2(normalize(dR + 1e-4), 1.5707963);
vec2 tangG = rot2(normalize(dG + 1e-4), 1.5707963);
vec2 tangB = rot2(normalize(dB + 1e-4), 1.5707963);
vec2 airR = tangR * sw * fallR * (0.06 + 0.22 * a) * (0.6 + 0.4 * cos(uv.y * 40.0 + tFast * 3.0 +
↪ seed));
vec2 airG = tangG * sw * fallG * (0.06 + 0.22 * a) * (0.6 + 0.4 * cos(uv.y * 38.0 + tFast * 3.3 +
↪ seed * 1.7));
vec2 airB = tangB * sw * fallB * (0.06 + 0.22 * a) * (0.6 + 0.4 * cos(uv.y * 42.0 + tFast * 2.9 +
↪ seed * 0.9));
vec2 jit = (h2(uv * vec2(233.3, 341.9) + tFast + seed) - 0.5)
* (0.0006 + 0.004 * ua);
tcR += airR + jit;
tcG += airG + jit;
tcB += airB + jit;
vec2 fR = vec2(mir.r > 0.5 ? 1.0 - tcR.x : tcR.x, tcR.y);
vec2 fG = vec2(mir.g > 0.5 ? 1.0 - tcG.x : tcG.x, tcG.y);
vec2 fB = vec2(mir.b > 0.5 ? 1.0 - tcB.x : tcB.x, tcB.y);
float seg = 4.0 + 2.0 * sin(tSlow * 0.33 + a01 * 2.0);

```

```

vec2 kUV = reflectUV(spiralTC, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + 0.55 * sin(tSlow * 0.42 + a01 * 3.0);
kUV = fractalFold(kUV, foldZoom, tFast, m, aspect);
kUV = rotateUV(kUV, tSlow * 0.23 + a01 * 1.1, m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(tSlow * 0.2) * (PI * tSlow), 5.0);
float period = log(base) * pingPong(tSlow * PI, 5.0);
float tz = tSlow * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35
            + 0.35 * sin(rD * 18.0 + tFast * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dirK = normalize(pwrap + 1e-6);
vec2 off = dirK * (0.0015 + 0.001 * sin(tFast * 1.3)) * vec2(1.0, 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((spiralTC - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign) * (0.8 + 0.3 * ringMirrorSpiral);
vec2 sR = mix(u0, fR, 0.6);
vec2 sG = mix(u1, fG, 0.6);
vec2 sB = mix(u2, fB, 0.6);
vec3 rC = preBlendColor(sR + off, tSlow, tFast, aspect);
vec3 gC = preBlendColor(sG, tSlow, tFast, aspect);
vec3 bC = preBlendColor(sB - off, tSlow, tFast, aspect);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + tFast * 1.2));
ring = ring * pingPong(tSlow * PI, 5.0) * (0.7 + 0.5 * ringMirrorSpiral);
float pulse = 0.5 + 0.5 * sin(tFast * 2.0 + rD * 28.0 + k * 12.0);
pulse *= mix(0.4, 1.4, a01);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.10
            + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.06;
float pulseAdd = 0.004 * (0.5 + 0.5 * sin(tFast * 3.7 + seed));
outCol += bloom;
outCol += pulseAdd * ua;
outCol = clamp(outCol, vec3(0.0), vec3(1.0));
outCol = limitHighlights(outCol);
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * (0.6 + 0.3 * a01));
finalRGB = limitHighlights(finalRGB);
finalRGB = clamp(finalRGB, 0.0, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #412 - frac shader04 julia

File: frac_shader04_julia.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB. Responds to mouse position for interactive control.

Technique: RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    vec2 d = normalize(vec2(cos(t*0.27), sin(t*0.31)));
    float s = dot(p, d);
    float band = 0.5 + 0.5*sin(s*6.28318530718*0.35 + t*0.9);
    float h = fract(s*0.22 + t*0.07 + 0.15*sin(t*0.33));
    float S = 0.75 + 0.25*sin(t*0.21 + s*2.0);
    float V = 0.75 + 0.25*band;
    vec3 base = hsv2rgb(vec3(h, S, V));
    float edge = smoothstep(0.2, 0.8, band);
    return mix(base*0.6, base, edge);
}
mat2 rot2(float a){ float s=sin(a), c=cos(a); return mat2(c,-s,s,c); }
void juliaEval(vec2 z0, vec2 cJ, out float iterCount, out float smoothIter, out float orbitTrapVal){
    vec2 z = z0;
    float i = 0.0;
    orbitTrapVal = 1e9;
    const int MAX_IT = 120;
    for(int k=0;k<MAX_IT;k++){
        orbitTrapVal = min(orbitTrapVal, length(z));
        vec2 z2 = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + cJ;
        z = z2;
        i += 1.0;
        if(dot(z,z) > 256.0) break;
    }
}
```

```

    }
    float r1 = length(z);
    smoothIter = i - log2(max(0.000001, log(max(r1, 1e-6))));
    iterCount = i;
}

float juliaScore(vec2 z0, vec2 cJ){
    vec2 z = z0;
    const int MAX_S = 28;
    float i = 0.0;
    for(int k=0;k<MAX_S;k++){
        vec2 z2 = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + cJ;
        z = z2;
        i += 1.0;
        if(dot(z,z) > 256.0) break;
    }
    float r1 = length(z);
    float sI = i - log2(max(0.000001, log(max(r1, 1e-6))));
    float tgt = 18.0;
    float score = 1.0/(1.0 + abs(sI - tgt));
    return score;
}

vec2 findAnchor(vec2 base, vec2 cJ, float rad){
    float best = -1.0;
    vec2 bestOff = vec2(0.0);
    for(int k=0;k<8;k++){
        float a = (6.28318530718/8.0)*float(k) + time_f*0.11;
        vec2 off = vec2(cos(a), sin(a)) * rad;
        float s = juliaScore(base + off, cJ);
        if(s > best){ best = s; bestOff = off; }
    }
    return bestOff;
}

void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    float t = time_f;
    float zoomSpeed = 0.28;
    float spin = 0.23;
    float scale = exp(-t * zoomSpeed);
    vec2 p = (uv - m) * ar;
    p = rot2(t*spin) * p;
    vec2 cJ = (iMouse.z > 0.5)
        ? ((iMouse.xy/iResolution)*2.0-1.0) * vec2(aspect,1.0) * 0.6
        : vec2(0.285 + 0.15*sin(t*0.17), 0.01 + 0.15*cos(t*0.21));
    float probeRad = 0.75 * scale;
    vec2 anchor = findAnchor(vec2(0.0), cJ, probeRad);
    vec2 z0 = (p + anchor) * scale * 2.0;
    float iterCount, smoothIter, orbitTrapVal;
    juliaEval(z0, cJ, iterCount, smoothIter, orbitTrapVal);
    float rD = length(p);
    float hue = fract(0.12*smoothIter + 0.07*t + 0.15*sin(orbitTrapVal*3.0));
    float sat = 0.65 + 0.25*sin(0.7*t + orbitTrapVal*5.0);
    float val = 0.80 + 0.20*sin(0.9*t + smoothIter*0.35);
    vec3 tint = hsv2rgb(vec3(hue, sat, val));
    float ring = smoothstep(0.0, 0.7, sin(log(rD*scale+1e-3)*9.5 + t*1.2));
    float pulse = 0.5 + 0.5*sin(t*2.0 + rD*28.0 + smoothIter*0.6);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
    vign = mix(0.85, 1.2, vign);
    vec2 pr = z0;

```

```

vec2 dir = normalize(pr + 1e-6);
vec2 off = dir * (0.0015 + 0.001 * sin(t*1.3)) * vec2(1.0, 1.0/aspect);
vec2 u0 = fract(pr/ar + m);
vec2 u1 = fract((pr*1.045)/ar + m);
vec2 u2 = fract((pr*0.955)/ar + m);
float rC = texture(samp, u0 + off).r;
float gC = texture(samp, u1).g;
float bC = texture(samp, u2 - off).b;
vec3 texRGB = vec3(rC, gC, bC);
float shade = smoothstep(0.0, 1.0, 1.0 - clamp(orbitTrapVal*0.5, 0.0, 1.0));
vec3 fractRGB = mix(texRGB, texRGB*tint, 0.65) * mix(0.7, 1.25, shade);
vec4 fractalColor = vec4(fractRGB * tint, 1.0);
float blendFactor = 0.58;
vec4 merged = mix(fractalColor, originalTexture, blendFactor);
merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
vec3 outCol = merged.rgb + bloom;
float wob = 0.9 + 0.1*sin(t + rD*14.0 + smoothIter*0.5);
outCol *= wob;
vec3 grad = movingGradient(tc, m, t, aspect);
float gradAmt = 0.35 + 0.25*sin(t*0.5 + rD*7.0 + smoothIter*0.35);
vec3 screenBlend = 1.0 - (1.0 - outCol) * (1.0 - grad);
outCol = mix(outCol, screenBlend, gradAmt);
vec4 tcol = texture(samp, tc);
outCol = mix(outCol, outCol*tcol.rgb, 0.8);
outCol = sin(outCol * (0.5 + 0.5*pingPong(t, 12.0)));
outCol = clamp(outCol, vec3(0.08), vec3(0.96));
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
outCol *= 1.0 + _ab * 0.6;
outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
↪ 0.0, 1.0) * 0.25), _ab);
color = vec4(outCol, 1.0);
}

```

Shader #431 - fractal fold neon

File: fractal-fold-neon.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Increase the number of folds for more complex patterns Alternate between rotation and scaling for more chaos Adding some scaling variation

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float time_f;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;

```

```

uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 4; i++) {
        p = abs((p - c) * (zoom + 0.10 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        if (i % 2 == 0) {
            p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
        } else {
            float scaleFactor = 1.0 + sin(t * 0.15 + float(i)) * 0.3;
            p *= scaleFactor;
        }
    }
    return p;
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 uv = tc;
    float A = clamp(amp, 0.0, 1.0);
    float U = clamp(uamp, 0.0, 1.0);
    vec2 m = (iMouse.z > 0.5 || iMouse.w > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 normPos = (uv - m) * vec2(aspect, 1.0);
    float dist = length(normPos);
    float phase = sin(dist * 8.0 - time_f * 2.0);
    float rippleStrength = 0.02 + ((0.05 + amp_rms * 0.04) * A);
    vec2 rippledUV = uv + (normPos * phase * rippleStrength);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.1);
    vec2 kUV = reflectUV(rippledUV, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.05 + (0.1 + amp_low * 0.4) * sin(time_f * 0.2);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    vec2 mapUV = (kUV - m) * vec2(aspect, 1.0);
    mapUV *= 0.8;
    float rot = time_f * 0.1;

```

```

float s = sin(rot);
float c = cos(rot);
mapUV = mat2(c, -s, s, c) * mapUV;
float dispersion = (0.01 + amp_high * 0.02) + (U * 0.05);
vec2 dispOffset = normalize(mapUV) * dispersion * length(mapUV);
vec2 centerBase = mapUV + m;
vec2 uvR = centerBase - dispOffset;
vec2 uvG = centerBase;
vec2 uvB = centerBase + dispOffset;
vec2 texR = abs(mod(uvR - 1.0, 2.0) - 1.0);
vec2 texG = abs(mod(uvG - 1.0, 2.0) - 1.0);
vec2 texB = abs(mod(uvB - 1.0, 2.0) - 1.0);
float r = texture(samp, texR).r;
float g = texture(samp, texG).g;
float b = texture(samp, texB).b;
vec3 neonColor = vec3(sin(time_f * (2.0 + amp_mid * 3.0)), sin(time_f * 1.5 + 2.0), sin(time_f +
↳ 1.0)) * 0.5 + 0.5;
neonColor *= 1.5;
vec3 finalCol = vec3(r, g, b) * neonColor;
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalCol *= 1.0 + _ab * 0.6;
finalCol = mix(finalCol, finalCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalCol, 1.0);
}

```

Shader #432 - fractal fold sketch

File: fractal-fold-sketch.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Increase the number of folds for more complex patterns Alternate between rotation and scaling for more chaos Adding some scaling variation

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float time_f;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
}

```

```

    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 4; i++) {
        p = abs((p - c) * (zoom + 0.10 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        if (i % 2 == 0) {
            p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
        } else {
            float scaleFactor = 1.0 + sin(t * 0.15 + float(i)) * 0.3;
            p *= scaleFactor;
        }
    }
    return p;
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 uv = tc;
    float A = clamp(amp, 0.0, 1.0);
    float U = clamp(uamp, 0.0, 1.0);
    vec2 m = (iMouse.z > 0.5 || iMouse.w > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 normPos = (uv - m) * vec2(aspect, 1.0);
    float dist = length(normPos);
    float phase = sin(dist * 8.0 - time_f * 2.0);
    float rippleStrength = 0.02 + ((0.05 + amp_rms * 0.04) * A);
    vec2 rippledUV = uv + (normPos * phase * rippleStrength);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.1);
    vec2 kUV = reflectUV(rippledUV, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.05 + (0.1 + amp_low * 0.4) * sin(time_f * 0.2);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    vec2 mapUV = (kUV - m) * vec2(aspect, 1.0);
    mapUV *= 0.8;
    float rot = time_f * 0.1;
    float s = sin(rot);
    float c = cos(rot);
    mapUV = mat2(c, -s, s, c) * mapUV;
    float dispersion = (0.01 + amp_high * 0.02) + (U * 0.05);
    vec2 dispOffset = normalize(mapUV) * dispersion * length(mapUV);
    vec2 centerBase = mapUV + m;
    vec2 uvR = centerBase - dispOffset;
    vec2 uvG = centerBase;
}

```

```

    vec2 uvB = centerBase + dispOffset;
    vec2 texR = abs(mod(uvR - 1.0, 2.0) - 1.0);
    vec2 texG = abs(mod(uvG - 1.0, 2.0) - 1.0);
    vec2 texB = abs(mod(uvB - 1.0, 2.0) - 1.0);
    float r = texture(samp, texR).r;
    float g = texture(samp, texG).g;
    float b = texture(samp, texB).b;
    float lightness = (r + g + b) / 3.0;
    vec3 pencilSketchColor = vec3(lightness);
    vec3 finalCol = pencilSketchColor * 0.8;
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalCol *= 1.0 + _ab * 0.6;
    finalCol = mix(finalCol, finalCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalCol, 1.0);
}

```

Shader #433 - fractal foldzx

File: fractal-foldzx.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Increase the number of folds for more complex patterns Alternate between rotation and scaling for more chaos Adding some scaling variation

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float time_f;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;

```

```

    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 4; i++) {
        p = abs((p - c) * (zoom + 0.10 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        if (i % 2 == 0) {
            p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
        } else {
            float scaleFactor = 1.0 + sin(t * 0.15 + float(i)) * 0.3;
            p *= scaleFactor;
        }
    }
    return p;
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 uv = tc;
    float A = clamp(amp, 0.0, 1.0);
    float U = clamp(uamp, 0.0, 1.0);
    vec2 m = (iMouse.z > 0.5 || iMouse.w > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 normPos = (uv - m) * vec2(aspect, 1.0);
    float dist = length(normPos);
    float phase = sin(dist * 8.0 - time_f * 2.0);
    float rippleStrength = 0.02 + ((0.05 + amp_rms * 0.04) * A);
    vec2 rippledUV = uv + (normPos * phase * rippleStrength);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.1);
    vec2 kUV = reflectUV(rippledUV, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.05 + (0.1 + amp_low * 0.4) * sin(time_f * 0.2);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    vec2 mapUV = (kUV - m) * vec2(aspect, 1.0);
    mapUV *= 0.8;
    float rot = time_f * 0.1;
    float s = sin(rot);
    float c = cos(rot);
    mapUV = mat2(c, -s, s, c) * mapUV;
    float dispersion = (0.01 + amp_high * 0.02) + (U * 0.05);
    vec2 dispOffset = normalize(mapUV) * dispersion * length(mapUV);
    vec2 centerBase = mapUV + m;
    vec2 uvR = centerBase - dispOffset;
    vec2 uvG = centerBase;
    vec2 uvB = centerBase + dispOffset;
    vec2 texR = abs(mod(uvR - 1.0, 2.0) - 1.0);
    vec2 texG = abs(mod(uvG - 1.0, 2.0) - 1.0);
    vec2 texB = abs(mod(uvB - 1.0, 2.0) - 1.0);
    float r = texture(samp, texR).r;
    float g = texture(samp, texG).g;
    float b = texture(samp, texB).b;
    float light = 1.0 + 0.5 * sin(dist * 20.0 - time_f * 5.0);
    vec3 finalCol = vec3(r, g, b) * light;
}

```



```

float vign = 1.0 - smoothstep(0.5, 1.5, dist);
finalCol *= vign;
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalCol *= 1.0 + _ab * 0.6;
finalCol = mix(finalCol, finalCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalCol, 1.0);
}

```

Shader #434 - fractal

File: fractal.glsl

Renders a fractal pattern using iterative computation (many iterations).

Technique: Iterative fractal computation with complex-plane mapping - Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 screenToComplex(vec2 screenPos, vec2 center, float zoom) {
    vec2 scale = vec2(zoom * (iResolution.x / iResolution.y), zoom);
    return center + (screenPos - iResolution * 0.5) * scale;
}
const int maxIterations = 300;
int mandelbrot(vec2 c) {
    vec2 z = vec2(0.0);
    int n = 0;
    for (int i = 0; i < maxIterations; i++) {
        z = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + c;
        if (dot(z, z) > 4.0) break;
        n++;
    }
    return n;
}
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {
    ivec4 isource = ivec4(isourcex * 255);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
}

```

```

    icolor.a = 1.0;
return icolor;
}
void main() {
    float zoomCycleSpeed = (0.3 + amp_low * 0.2);
    float zoomIntensity = (20.0 + amp_low * 5.0);
    float zoom = exp(-abs(sin(time_f * zoomCycleSpeed)) * zoomIntensity);
    vec2 center = vec2(-0.743643887037151, 0.13182590420533);
    vec2 c = screenToComplex(gl_FragCoord.xy, center, zoom * iResolution.y / iResolution.x);
    int n = mandelbrot(c);
    float colorIndex = rand(vec2(n, time_f));
    vec3 col = 0.5 + 0.5 * cos(6.28318 * (vec3(colorIndex) + vec3(0.0, 0.33, 0.67)));
    color = n == maxIterations ? vec4(0.0, 0.0, 0.0, 1.0) : vec4(col, 1.0);
    color = xor_RGB(color, texture(samp, tc));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↵ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #435 - Fractal1

File: Fractal1.glsl

Renders a fractal pattern using iterative computation (400 iterations). Convert normalized coordinates [0,1] to [-1,1] space Dynamic zooming to ensure the fractal remains visible Slower exponential zoom

Technique: Iterative fractal computation with complex-plane mapping - Linear mix/lerp blending
- Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void)
{
    vec2 uv = (tc * 2.0) - 1.0;
    float zoom = pow(0.8, time_f * 0.5);
    uv *= zoom;
    vec2 center = vec2(-0.75, 0.0);
    center += vec2(-0.3, -0.4) * (1.0 - zoom);
    vec2 z = vec2(0.0);
    vec2 c = uv + center;
    const int MAX_ITER = 400;
    int i;
    for(i = 0; i < MAX_ITER; i++)
    {
        z = vec2(z.x * z.x - z.y * z.y + c.x,
            2.0 * z.x * z.y + c.y);
        if(dot(z, z) > 4.0)
            break;
    }
    float normalized = float(i) / float(MAX_ITER);
    vec3 rainbow = 0.5 + 0.5 * cos(6.28318 * (vec3(0.5, 0.3, 0.1) + normalized * 3.0));
    vec3 texColor = texture(samp, tc).rgb;
    color = vec4(mix(texColor, rainbow, 0.8), 1.0);
}

```

```
}
```

Shader #436 - fractal2

File: fractal2.glsl

Renders a fractal pattern using iterative computation (many iterations).

Technique: Iterative fractal computation with complex-plane mapping - Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 screenToComplex(vec2 screenPos, vec2 center, float zoom) {
    vec2 scale = vec2(zoom * (iResolution.x / iResolution.y), zoom);
    return center + (screenPos - iResolution * 0.5) * scale;
}
const int maxIterations = 300;
int mandelbrot(vec2 c) {
    vec2 z = vec2(0.0);
    int n = 0;
    for (int i = 0; i < maxIterations; i++) {
        z = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + c;
        if (dot(z, z) > 4.0) break;
        n++;
    }
    return n;
}
vec4 xor_RGB(vec4 icolor, vec4 isource) {
    ivec4 isource = ivec4(isource * 255.0);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255.0 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
void main() {
    float zoomCycleSpeed = (0.3 + amp_low * 0.2);
    float zoomIntensity = (20.0 + amp_low * 5.0);
    float zoom = exp(-abs(sin(time_f * zoomCycleSpeed)) * zoomIntensity);
    vec2 center = vec2(-0.743643887037151, 0.13182590420533);
```

```

    vec2 c = screenToComplex(gl_FragCoord.xy, center, zoom * iResolution.y / iResolution.x);
    int n = mandelbrot(c);
    float colorIndex = rand(vec2(n, time_f));
    vec3 col = 0.5 + 0.5 * cos(6.28318 * (vec3(colorIndex) + vec3(0.0, 0.33, 0.67)));
    vec4 mandelbrotColor = n == maxIterations ? vec4(0.0, 0.0, 0.0, 1.0) : vec4(col, 1.0);
    vec4 textureColor = texture(samp, tc);
    color = xor_RGB(mandelbrotColor, textureColor);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #437 - fractal3

File: fractal3.glsl

Renders a fractal pattern using iterative computation (many iterations).

Technique: Iterative fractal computation with complex-plane mapping - Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 screenToComplex(vec2 screenPos, vec2 center, float zoom) {
    vec2 scale = vec2(zoom * (iResolution.x / iResolution.y), zoom);
    return center + (screenPos - iResolution * 0.5) * scale;
}
const int maxIterations = 300;
int mandelbrot(vec2 c) {
    vec2 z = vec2(0.0);
    int n = 0;
    for (int i = 0; i < maxIterations; i++) {
        z = vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + c;
        if (dot(z, z) > 4.0) break;
        n++;
    }
    return n;
}
vec4 xor_RGB(vec4 icolor, vec4 isourcex) {
    ivec4 isource = ivec4(isourcex * 255.0);
    ivec3 int_color;
    for (int i = 0; i < 3; ++i) {
        int_color[i] = int(255.0 * icolor[i]);
        int_color[i] ^= isource[i];
    }
}

```

```

        if (int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
return icolor;
}
void main() {
    float zoomCycleSpeed = (0.1 + amp_low * 0.2);
    float zoomIntensity = (20.0 + amp_low * 5.0);
    float zoom = exp(-abs(sin(time_f * zoomCycleSpeed)) * zoomIntensity);
    vec2 center = vec2(-0.745, 0.113);
    vec2 c = screenToComplex(gl_FragCoord.xy, center, zoom * iResolution.y / iResolution.x);
    int n = mandelbrot(c);
    float colorIndex = rand(vec2(n, time_f));
    vec3 col = 0.5 + 0.5 * cos(6.28318 * (vec3(colorIndex) + vec3(0.0, 0.33, 0.67)));
    vec4 mandelbrotColor = n == maxIterations ? vec4(0.0, 0.0, 0.0, 1.0) : vec4(col, 1.0);
    vec4 textureColor = texture(samp, tc);
    color = xor_RGB(mandelbrotColor, textureColor);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #438 - fractal4

File: fractal4.glsl

Renders a fractal pattern using iterative computation (many iterations).

Technique: Iterative fractal computation with complex-plane mapping - Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 screenToComplex(vec2 screenPos, vec2 center, float zoom) {
    vec2 scale = vec2(zoom * (iResolution.x / iResolution.y), zoom);
    return center + (screenPos - iResolution * 0.5) * scale;
}
const int maxIterations = 300;
int mandelbrot(vec2 c) {
    vec2 z = vec2(0.0);
    int n = 0;
    for (int i = 0; i < maxIterations; i++) {

```

```

        z = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + c;
        if (dot(z, z) > 4.0) break;
        n++;
    }
    return n;
}
vec4 xor_RGB(vec4 icolor, vec4 isource) {
    ivec4 isource = ivec4(isource * 255.0);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255.0 * icolor[i]);
        int_color[i] ^= isource[i];
        if(int_color[i] > 255)
            int_color[i] %= 255;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
vec2 interestingCenters(int index) {
    vec2 centers[5];
    centers[0] = vec2(-0.743643887037151, 0.13182590420533);
    centers[1] = vec2(-0.745428, 0.113009);
    centers[2] = vec2(-0.7453, 0.1127);
    centers[3] = vec2(-0.7435, 0.1318);
    centers[4] = vec2(-0.743644786, 0.1318252536);
    return centers[index % 5];
}
void main() {
    float zoomCycleSpeed = (0.045 + amp_low * 0.2);
    float zoomIntensity = (20.0 + amp_low * 5.0);
    float zoom = exp(-abs(sin(time_f * zoomCycleSpeed)) * zoomIntensity);
    int cycleIndex = int(floor(time_f / (3.14159 / zoomCycleSpeed)));
    vec2 center = interestingCenters(cycleIndex);
    vec2 c = screenToComplex(gl_FragCoord.xy, center, zoom * iResolution.y / iResolution.x);
    int n = mandelbrot(c);
    float colorIndex = rand(vec2(n, time_f));
    vec3 col = 0.5 + 0.5 * cos(6.28318 * (vec3(colorIndex) + vec3(0.0, 0.33, 0.67)));
    vec4 mandelbrotColor = n == maxIterations ? vec4(0.0, 0.0, 0.0, 1.0) : vec4(col, 1.0);
    vec4 textureColor = texture(samp, tc);
    color = xor_RGB(mandelbrotColor, textureColor);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #439 - fractal color wave

File: fractal_color_wave.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Low iterations (3) for large shapes Vignette & Pulse Mix the original background with the fractal overlay.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

```

```

vec3 getRainbowTexture(vec2 uv) {
    vec2 centered = uv * 2.0 - 1.0;
    float wave = sin(centered.x * 10.0 + time_f * 2.0) * 0.1;
    float angle = atan(centered.y + wave, centered.x) + time_f * 2.0;
    vec3 rain = rainbow(angle / (2.0 * PI));
    vec3 tex = texture(samp, uv).rgb;
    return mix(tex, rain, 0.5);
}

void main(void) {
    vec4 background = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.15 + (0.15 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float base = 1.82;
    float period = log(base);
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw * 0.5;
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * 0.002 * vec2(1.0, 1.0 / aspect);
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec3 rC = getRainbowTexture(u0 + off);
    vec3 gC = getRainbowTexture(u1);
    vec3 bC = getRainbowTexture(u2 - off);
    vec3 fractalCol = vec3(rC.r, gC.g, bC.b);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    float pulse = 0.75 + 0.25 * sin(time_f * 2.0);
    fractalCol *= vign * pulse;
    fractalCol = clamp(fractalCol, 0.0, 1.0);
    vec3 finalRGB = mix(background.rgb, fractalCol, 0.6);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, background.a);
}

```

Shader #440 - fractal diamond rainbow

File: fractal_diamond_rainbow.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position for interactive control. The Rainbow Generator This ensures the "small fractals of the image" remain visible rather than turning into pixel noise.

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending -
Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
```

```

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 centered = uv * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    float wave = sin(centered.x * 10.0 + time_f * 2.0) * 0.1;
    float angle = atan(centered.y + wave, centered.x) + time_f * 2.0;
    vec3 rain = rainbow(angle / (2.0 * PI));
    return mix(tex, rain, 0.5);
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.3 + (0.3 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82;
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵ ;
    vec2 u0 = fract(pwrap / ar + m);
}

```

```

    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.8 + 0.2 * pulse) * vign;
    outCol = clamp(outCol, 0.0, 1.0);
    vec3 finalRGB = mix(baseTex.rgb, outCol, 0.6);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #441 - fractal diamond rainbow blend

File: fractal_diamond_rainbow_blend.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Keeping iterations low (3) so the texture remains visible and doesn't turn into noise. Logarithmic Spiral Projection

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
}

```

```

    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 centered = uv * 2.0 - 1.0;
    float wave = sin(centered.x * 10.0 + time_f * 2.0) * 0.1;
    float angle = atan(centered.y + wave, centered.x) + time_f * 2.0;
    vec3 rain = rainbow(angle / (2.0 * PI));
    return mix(tex, rain, 0.5);
}
float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);

```

```

vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.3 + (0.3 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82;
float period = log(base);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 fractalRGB = vec3(rC.r, gC.g, bC.b);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
fractalRGB *= (0.8 + 0.2 * pulse) * vign;
fractalRGB = clamp(fractalRGB, 0.0, 1.0);
float wave = sin(time_f * 0.4 + PI * 1.5);
float morph = wave * 0.5 + 0.5;
morph = smoothstep(0.0, 1.0, morph);
vec3 finalRGB = mix(baseTex.rgb, fractalRGB, morph);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #442 - fractal diamond rainbow no tex

File: fractal_diamond_rainbow_no_tex.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position for interactive control. Iterations = 3. This keeps the image chunks large and visible rather than turning into noise. The texture IS the fractal now, colored by the rainbow.

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;

```

```

uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);

```

```

    vec2 centered = uv * 2.0 - 1.0;
    float wave = sin(centered.x * 10.0 + time_f * 2.0) * 0.1;
    float angle = atan(centered.y + wave, centered.x) + time_f * 2.0;
    vec3 rain = rainbow(angle / (2.0 * PI));
    return mix(tex, rain, 0.5);
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.3 + (0.3 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82;
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵ ;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 outCol = vec3(rC.r, gC.g, bC.b);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    outCol *= (0.8 + 0.2 * pulse) * vign;
    outCol = clamp(outCol, 0.0, 1.0);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    outCol *= 1.0 + _ab * 0.6;
    outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
    ↵ 0.0, 1.0) * 0.25), _ab);
    color = vec4(outCol, 1.0);
}

```

```
}
```

Shader #443 - fractal diamond rainbow overflow

File: fractal_diamond_rainbow_overflow.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Same smooth wave as before Calculate the pure XOR result between original and fractal Smoothly fade between Original and the XOR'd Fractal

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
```



```

        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 centered = uv * 2.0 - 1.0;
    float wave = sin(centered.x * 10.0 + time_f * 2.0) * 0.1;
    float angle = atan(centered.y + wave, centered.x) + time_f * 2.0;
    vec3 rain = rainbow(angle / (2.0 * PI));
    return mix(tex, rain, 0.5);
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec3 xorBlend(vec3 colorA, vec3 colorB, float s) {
    ivec3 iA = ivec3(colorA * s * 255.0);
    ivec3 iB = ivec3(colorB * s * 255.0);
    ivec3 iXor = iA ^ iB;
    return vec3(iXor) / 255.0;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.3 + (0.3 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82;
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);

```

```

float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 fractalRGB = vec3(rC.r, gC.g, bC.b);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
fractalRGB *= (0.8 + 0.2 * pulse) * vign;
fractalRGB = clamp(fractalRGB, 0.0, 1.0);
float wave = sin(time_f * 0.4 + PI * 1.5);
float morph = smoothstep(0.0, 1.0, wave * 0.5 + 0.5);
vec3 xorResult = xorBlend(baseTex.rgb, fractalRGB, morph);
vec3 finalRGB = mix(baseTex.rgb, xorResult, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #444 - fractal diamond rainbow overflow const

File: fractal_diamond_rainbow_overflow_const.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Directly blending original + fractal using XOR logic

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
vec3 rainbow(float t) {
    t = fract(t);

```

```

    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    vec2 centered = uv * 2.0 - 1.0;
    float wave = sin(centered.x * 10.0 + time_f * 2.0) * 0.1;
    float angle = atan(centered.y + wave, centered.x) + time_f * 2.0;
    vec3 rain = rainbow(angle / (2.0 * PI));
    return mix(tex, rain, 0.5);
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
}

```

```

    p.x /= aspect;
    return p + c;
}
vec3 xorBlend(vec3 colorA, vec3 colorB) {
    ivec3 iA = ivec3(colorA * 255.0);
    ivec3 iB = ivec3(colorB * 255.0);
    ivec3 iXor = iA ^ iB;
    return vec3(iXor) / 255.0;
}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.3 + (0.3 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82;
    float period = log(base);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
    ↪ ;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 fractalRGB = vec3(rC.r, gC.g, bC.b);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    fractalRGB *= (0.8 + 0.2 * pulse) * vign;
    fractalRGB = clamp(fractalRGB, 0.0, 1.0);
    vec3 finalRGB = xorBlend(baseTex.rgb, fractalRGB);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #445 - fractal liquid no zero

File: fractal_liquid_no_zero.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Smooth FBM for Liquid Effect Generate flow vectors based on screen position and time Apply flow to the initial coordinates BEFORE they enter the fractal loop

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending -
Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        v += a * noise(p);
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
```

```

    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.1), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.1, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s = texture(img, uv).rgb * 4.0;
    s += texture(img, uv + ts * vec2(1.0, 1.0)).rgb;
    s += texture(img, uv + ts * vec2(-1.0, -1.0)).rgb;
    s += texture(img, uv + ts * vec2(-1.0, 1.0)).rgb;
    s += texture(img, uv + ts * vec2(1.0, -1.0)).rgb;
    return s / 8.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    vec3 neon = neonPalette(time_f + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = max(grad, neon * 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));

```

```

    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float flow1 = fbm(uv * 1.5 + time_f * 0.2);
    float flow2 = fbm(uv * 2.0 - time_f * 0.15);
    vec2 flow = vec2(cos(flow1 * 5.0), sin(flow2 * 5.0)) * 0.05;
    vec2 warpedUV = tc + flow;
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(warpedUV, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42 + flow1);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6 + flow2 * 3.0);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    float spread = 1.045 + (0.02 * flow1);
    vec2 u1 = fract((pwrap * spread) / ar + m);
    vec2 u2 = fract((pwrap * (1.0/spread)) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵ ;
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = 0.3 + 0.7 * ring;
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    pulse = 0.4 + 0.6 * pulse;
    vec3 effectColor = kaleidoRGB;
    effectColor *= (ring * pulse);
    vec3 fillPalette = neonPalette(length(p) + time_f) * 0.15;
    effectColor = max(effectColor, fillPalette);
    vec3 bloom = effectColor * effectColor * 0.18 + pow(max(effectColor - 0.6, 0.0), vec3(2.0)) * 0.12;
    effectColor += bloom * 1.2;
    float intensity = 0.6 + 0.2 * flow1 + 0.4 * pingPong(time_f, 2.0);
    vec3 finalRGB = baseTex.rgb + (effectColor * intensity);
    float _ab = clamp(amp_peak, 0.0, 1.0);

```

```

float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(clamp(finalRGB, 0.0, 1.0), baseTex.a);
}

```

Shader #446 - fractal no zero

File: fractal_no_zero.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Prevent crushing blacks FORCE BRIGHTNESS: If result is dark, boost it with the palette Log Polar Tunnel

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);
        n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}

```



```

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.1), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.1, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s = texture(img, uv).rgb * 4.0;
    s += texture(img, uv + ts * vec2(1.0, 1.0)).rgb;
    s += texture(img, uv + ts * vec2(-1.0, -1.0)).rgb;
    s += texture(img, uv + ts * vec2(-1.0, 1.0)).rgb;
    s += texture(img, uv + ts * vec2(1.0, -1.0)).rgb;
    return s / 8.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    vec3 neon = neonPalette(time_f + r * 1.3);
}

```

```

    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = max(grad, neon * 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float electricity = fbm(p * 5.0 - time_f);
    float electricPulse = pow(electricity, 3.0);
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    ang += electricPulse * 0.2;
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    float spread = 1.045 + ((0.05 + amp_high * 0.03) * electricPulse);
    vec2 u1 = fract((pwrap * spread) / ar + m);
    vec2 u2 = fract((pwrap * (1.0/spread)) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵ ;
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = 0.3 + 0.7 * ring;
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    pulse = 0.4 + 0.6 * pulse;
}

```

```

    vec3 effectColor = kaleidoRGB;
    effectColor *= (ring * pulse);
    vec3 fillPalette = neonPalette(length(p) + time_f) * 0.15;
    effectColor = max(effectColor, fillPalette);
    vec3 bloom = effectColor * effectColor * 0.18 + pow(max(effectColor - 0.6, 0.0), vec3(2.0)) * 0.12;
    effectColor += bloom * (1.0 + electricPulse + amp_low * 0.5);
    float intensity = 0.6 + (0.4 + amp_smooth * 0.3) * pingPong(time_f, 2.0);
    vec3 finalRGB = baseTex.rgb + (effectColor * intensity);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(clamp(finalRGB, 0.0, 1.0), baseTex.a);
}

```

Shader #447 - fractal noise

File: fractal_noise.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Ridged FBM for Electricity Keeping your Tent Blur for the fractal generation (so the light is soft) ... Simplified kernel for performance while maintaining look ...

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Indexed color palette look-up table - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i + vec2(0.0, 0.0)), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 4; i++) {
        float n = noise(p);

```

```

        n = 1.0 - abs(n * 2.0 - 1.0);
        v += a * n;
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}

float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s = texture(img, uv).rgb * 4.0;
    s += texture(img, uv + ts * vec2(1.0, 1.0)).rgb;

```

```

    s += texture(img, uv + ts * vec2(-1.0, -1.0)).rgb;
    s += texture(img, uv + ts * vec2(-1.0, 1.0)).rgb;
    s += texture(img, uv + ts * vec2(1.0, -1.0)).rgb;
    return s / 8.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    vec3 neon = neonPalette(time_f + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float electricity = fbm(p * 5.0 - time_f);
    float electricPulse = pow(electricity, 3.0);
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    ang += electricPulse * 0.2;
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    float spread = 1.045 + ((0.05 + amp_high * 0.03) * electricPulse);
    vec2 u1 = fract((pwrap * spread) / ar + m);
    vec2 u2 = fract((pwrap * (1.0/spread)) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);

```

```

    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
    ↪ ;
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    vec3 effectColor = kaleidoRGB;
    effectColor *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse);
    vec3 bloom = effectColor * effectColor * 0.18 + pow(max(effectColor - 0.6, 0.0), vec3(2.0)) * 0.12;
    effectColor += bloom * (1.0 + electricPulse + amp_low * 0.5);
    float intensity = 0.6 + (0.4 + amp_smooth * 0.3) * pingPong(time_f, 2.0);
    vec3 finalRGB = baseTex.rgb + (effectColor * intensity);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(clamp(finalRGB, 0.0, 1.0), baseTex.a);
}

```

Shader #448 - fractal texture large

File: fractal_texture_large.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. CHANGED: Reduced iterations from 6 to 3. This keeps the diamonds large and prevents the geometry from becoming "noisy" or pixelated. Darkened texture return

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
}

```

```

    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    return tex * 0.7;
}
float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    vec2 xuv = 1.0 - abs(1.0 - 2.0 * tc);
    xuv = xuv - floor(xuv);
    vec4 baseTex = texture(samp, xuv);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.5 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
}

```

```

    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.15 + (0.15 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw * 0.5;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
    ↪ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.6 + 0.4 * pulse);
    outCol *= vign;
    outCol = clamp(outCol, vec3(0.0), vec3(1.0));
    vec3 finalRGB = mix(baseTex.rgb * 0.8, outCol, pingPong(glow * PI, 5.0) * 0.6);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #449 - fractal texture large audio

File: fractal_texture_large_audio.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position for interactive control. new audio uniforms peak absolute sample value in current audio buffer RMS energy of current audio buffer

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
```



```

out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * 0.35 + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    return tex * 0.7;
}

```

```

}
float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    vec2 xuv = 1.0 - abs(1.0 - 2.0 * tc);
    xuv = xuv - floor(xuv);
    vec4 baseTex = texture(samp, xuv);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - 0.5, r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + 2.0 * sin(time_f * 0.33) + (amp_low * 6.0);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.15 + 0.15 * sin(time_f * 0.42) + (amp_mid * 0.4);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * 0.23, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * 0.65 + (amp_smooth * 1.5);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw * 0.5;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * (0.0015 + 0.001 * sin(time_f * 1.3) + (amp_high * 0.02)) * vec2(1.0, 1.0 / aspect);
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, 1.15, vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * PI), 5.0);
    float pulse = 0.5 + 0.5 * sin(time_f * 2.0 + rD * 28.0 + k * 12.0) + (amp_peak * 0.6);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.6 + 0.4 * pulse);
    outCol *= vign;
    outCol = clamp(outCol, vec3(0.0), vec3(1.0));
    vec3 finalRGB = mix(baseTex.rgb * 0.8, outCol, pingPong(glow * PI, 5.0) * 0.6);
}

```

```

float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #450 - fractal texture no wrap

File: fractal_texture_no_wrap.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. CHANGED: Reduced iterations from 6 to 3. This keeps the diamonds large and prevents the geometry from becoming "noisy" or pixelated. Darkened texture return

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {

```

```

    vec2 p = uv;
    for (int i = 0; i < 3; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    return tex * 0.7;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec2 xuv = tc;
    vec4 baseTex = texture(samp, xuv);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.5 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.15 + (0.15 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);

```

```

float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw * 0.5;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.6 + 0.4 * pulse);
outCol *= vign;
outCol = clamp(outCol, vec3(0.0), vec3(1.0));
vec3 finalRGB = mix(baseTex.rgb * 0.8, outCol, pingPong(glow * PI, 5.0) * 0.6);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #451 - fractal zoom e

File: fractal_zoom_e.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){

```

```

    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalZoom(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<5;i++){
        p = abs((p - c) * zoom) - 0.5 + c;
        p = rotateUV(p, t*0.1, c, aspect);
    }
    return p;
}

void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    vec2 kaleidoUV = reflectUV(uv, 6.0, m, aspect);
    float zoom = 1.5 + (0.5 + amp_low * 0.4) * sin(time_f * 0.5);
    kaleidoUV = fractalZoom(kaleidoUV, zoom, time_f, m, aspect);
    kaleidoUV = rotateUV(kaleidoUV, time_f * 0.2, m, aspect);
    vec2 p = (kaleidoUV - m) * ar;
    float base = 1.72;
    float period = log(base);
    float tz = time_f * 0.5;
    float r = length(p) + 1e-6;
    float ang = atan(p.y, p.x) + tz * 0.3;
    float k = fract((log(r) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 zoomUV = fract(pwrap / ar + m);
    vec4 kaleidoColor = texture(samp, zoomUV);
    float blendFactor = 0.6;
    vec4 blendedColor = mix(kaleidoColor, originalTexture, blendFactor);
    blendedColor.rgb *= 0.5 + 0.5 * sin(zoomUV.xy * time_f);
    color = sin(blendedColor * pingPong(time_f, 10.0));
    vec4 t = texture(samp, tc);
    color = color * t * 0.8;
    color = sin(color * pingPong(time_f, 15.0));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color.a = 1.0;
}

```

Shader #594 - julia

File: julia.glsl

Renders a fractal pattern using iterative computation (30 iterations). Responds to mouse position and audio input (amplitude/volume) for interactive control. Pseudo-random helper: returns a float in [0..1] (Classic 2D hash based on UV coords) A simple hashing function

Technique: Iterative fractal computation with complex-plane mapping - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float hash12(vec2 p) {
    vec3 p3 = fract(vec3(p.xy * 0.1031));
    p3 += dot(p3, p3.yzx + 19.19);
    return fract((p3.x + p3.y) * p3.z);
}
vec2 genSeed(float key) {
    float r1 = hash12(vec2(key, 13.789));
    float r2 = hash12(vec2(key, 37.239));
    return vec2(r1, r2);
}
mat2 rotate2D(float angle){
    return mat2(cos(angle), -sin(angle),
                sin(angle), cos(angle));
}
vec4 drawJulia(vec2 uv, vec2 center, vec2 c) {
    float zoom = 2.0;
    uv = (uv - center) * zoom;
    const int MAX_ITER = 30;
    vec2 z = uv;
    float m = 0.0;
    for (int i = 0; i < MAX_ITER; i++) {
        z = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + c;
        if (dot(z,z) > 4.0) {
            m = float(i) / float(MAX_ITER);
            break;
        }
    }
    float colVal = m;
    vec3 col = vec3(sin(colVal * 6.2831),
                    colVal,
                    1.0 - colVal);
    return vec4(col, smoothstep(0.0, 1.0, m));
}
vec4 drawPulsingCircle(vec2 uv, vec2 center, float radius) {
    float dist = length(uv - center);
    float pulse = 0.5 + 0.4 * sin(time_f * 2.0);
    float finalRadius = radius * pulse;
    float edge = smoothstep(finalRadius, finalRadius - 0.01, dist);
    float t = dist / finalRadius;
```

```

    vec3 baseColor = mix(vec3(1.0, 0.5, 0.0),
                        vec3(0.0, 0.5, 1.0),
                        t);
    float alpha = 1.0 - edge;
    return vec4(baseColor, alpha);
}

void main(void)
{
    color = texture(samp, tc);
    float cycleDuration = 13.0;
    float cycleTime = mod(time_f, cycleDuration);
    float cycleIndex = floor(time_f / cycleDuration);
    float fractalFadeInStart = 0.0;
    float fractalFadeInEnd = 1.0;
    float fractalHoldEnd = 3.0;
    float fractalFadeOutEnd = 4.0;
    float circleFadeInStart = 5.0;
    float circleFadeInEnd = 6.0;
    float circleHoldEnd = 8.0;
    float circleFadeOutEnd = 9.0;
    vec2 fractalRand = genSeed(cycleIndex * 3.17);
    vec2 fractalCenter = fractalRand * 1.6 - 0.8;
    vec2 fractalSeed = genSeed(cycleIndex * 1.93) * 2.0 - 1.0;
    vec2 circleRand = genSeed(cycleIndex * 7.87 + 1.234);
    vec2 circleCenter = circleRand * 1.6 - 0.8;
    float circleRadius = 0.2 + 0.2 * hash12(circleRand + 0.123);
    float fractalAlpha = 0.0;
    if (cycleTime >= fractalFadeInStart && cycleTime < fractalFadeInEnd) {
        float t = (cycleTime - fractalFadeInStart) / (fractalFadeInEnd - fractalFadeInStart);
        fractalAlpha = smoothstep(0.0, 1.0, t);
    } else if (cycleTime >= fractalFadeInEnd && cycleTime < fractalHoldEnd) {
        fractalAlpha = 1.0;
    } else if (cycleTime >= fractalHoldEnd && cycleTime < fractalFadeOutEnd) {
        float t = (cycleTime - fractalHoldEnd) / (fractalFadeOutEnd - fractalHoldEnd);
        fractalAlpha = 1.0 - smoothstep(0.0, 1.0, t);
    }
    float circleAlpha = 0.0;
    if (cycleTime >= circleFadeInStart && cycleTime < circleFadeInEnd) {
        float t = (cycleTime - circleFadeInStart) / (circleFadeInEnd - circleFadeInStart);
        circleAlpha = smoothstep(0.0, 1.0, t);
    } else if (cycleTime >= circleFadeInEnd && cycleTime < circleHoldEnd) {
        circleAlpha = 1.0;
    } else if (cycleTime >= circleHoldEnd && cycleTime < circleFadeOutEnd) {
        float t = (cycleTime - circleHoldEnd) / (circleFadeOutEnd - circleHoldEnd);
        circleAlpha = 1.0 - smoothstep(0.0, 1.0, t);
    }
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    vec4 fractalCol = drawJulia(uv, fractalCenter, fractalSeed);
    fractalCol.a *= fractalAlpha;
    vec4 circleCol = drawPulsingCircle(uv, circleCenter, circleRadius);
    circleCol.a *= circleAlpha;
    color = mix(color, fractalCol, fractalCol.a);
    color = mix(color, circleCol, circleCol.a);
}

```


Shader #614 - Liquid Fractal Tunnel

File: Liquid_Fractal_Tunnel.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections. Responds to mouse position and audio input (amplitude/volume) for interactive control. Distortion amount Chromatic aberration amount Logic from Shader 2: Creates the kaleidoscope geometry

Technique: Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - 2D rotation matrix - UV mirroring/folding for symmetry - Hermite smoothstep interpolation - Radial vignette falloff - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float time_f;
uniform float amp;
uniform float uamp;

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 2; i++) {
        p = abs((p - c) * (zoom + 0.10 * sin(t * 0.35 + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 uv = tc;
    float A = clamp(amp, 0.0, 1.0);
    float U = clamp(uamp, 0.0, 1.0);
    vec2 m = (iMouse.z > 0.5 || iMouse.w > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 normPos = (uv - m) * vec2(aspect, 1.0);
```

```

float dist = length(normPos);
float phase = sin(dist * 8.0 - time_f * 2.0);
float rippleStrength = 0.02 + (0.05 * A);
vec2 rippledUV = uv + (normPos * phase * rippleStrength);
float seg = 4.0 + 2.0 * sin(time_f * 0.1);
vec2 kUV = reflectUV(rippledUV, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.05 + 0.1 * sin(time_f * 0.2);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
vec2 mapUV = (kUV - m) * vec2(aspect, 1.0);
mapUV *= 0.8;
float rot = time_f * 0.1;
float s = sin(rot);
float c = cos(rot);
mapUV = mat2(c, -s, s, c) * mapUV;
float dispersion = 0.01 + (U * 0.05);
vec2 dispOffset = normalize(mapUV) * dispersion * length(mapUV);
vec2 centerBase = mapUV + m;
vec2 uvR = centerBase - dispOffset;
vec2 uvG = centerBase;
vec2 uvB = centerBase + dispOffset;
vec2 texR = abs(mod(uvR - 1.0, 2.0) - 1.0);
vec2 texG = abs(mod(uvG - 1.0, 2.0) - 1.0);
vec2 texB = abs(mod(uvB - 1.0, 2.0) - 1.0);
float r = texture(samp, texR).r;
float g = texture(samp, texG).g;
float b = texture(samp, texB).b;
float light = 1.0 + 0.5 * sin(dist * 20.0 - time_f * 5.0);
vec3 finalCol = vec3(r, g, b) * light;
float vign = 1.0 - smoothstep(0.5, 1.5, dist);
finalCol *= vign;
color = vec4(finalCol, 1.0);
}

```

Shader #625 - mandella1

File: mandella1.glsl

Applies a time-animated visual transformation to the input texture. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
void main(void) {
    color = texture(samp, tc);
}

```

Shader #662 - mirror mandella

File: mirror-mandella.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - UV mirroring/folding for symmetry
- Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    float loopDuration = 25.0;
    float t = mod(time_f, loopDuration);
    vec2 aspect = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 uv = 1.0 - abs(1.0 - 2.0 * tc);
    uv = uv - floor(uv);
    vec2 nc = (uv * 2.0 - 1.0) * sin(aspect * time_f);
    nc.x = abs(nc.x);
    float d = length(nc);
    float a = atan(nc.y, nc.x);
    float spiralSpeed = 5.0;
    float inward = t / loopDuration;
    a += (1.0 - smoothstep(0.0, 8.0, d)) * t * spiralSpeed;
    d *= 1.0 - inward;
    vec2 spiral = vec2(cos(a), sin(a)) * tan(d);
    vec2 uv0 = (spiral / aspect + 1.0) * 0.5;
    vec2 p = (uv0 * 2.0 - 1.0) * aspect;
    float r = length(p);
    float ang = atan(p.y, p.x);
    float N = 10.0;
    float tau = 6.28318530718;
    float sector = tau / N;
    ang = mod(ang + 0.5 * sector, sector);
    ang = abs(ang - 0.5 * sector);
    float ringFreq = 6.0;
    float ring = fract(r * ringFreq + 0.15 * sin(time_f * 0.5));
    float ringMirror = abs(ring - 0.5) * 2.0;
    float swirl = 0.25 * sin(time_f * 0.3);
    ang += swirl * r;
    float zoom = 0.85 + 0.1 * sin(time_f * 0.27);
    vec2 m = vec2(cos(ang), sin(ang)) * (r * zoom * (0.85 + 0.15 * ringMirror));
    vec2 uvx = (m / aspect + 1.0) * 0.5;
    vec2 px = 1.0 / iResolution;
    vec3 c = texture(samp, uv).rgb;
    c += texture(samp, uvx + vec2(px.x, 0)).rgb;
    c += texture(samp, uvx + vec2(-px.x, 0)).rgb;
    c += texture(samp, uvx + vec2(0, px.y)).rgb;
    c += texture(samp, uvx + vec2(0, -px.y)).rgb;
    c *= 0.2;
    float glow = smoothstep(0.95, 0.2, r) * (0.6 + 0.4 * ringMirror);
    vec3 base = c * (0.7 + 0.3 * glow);
    float hue = fract(ang / tau + 0.5 + 0.03 * time_f);
    float rC = clamp(abs(hue * 6.0 - 3.0) - 1.0, 0.0, 1.0);
    float gC = clamp(2.0 - abs(hue * 6.0 - 2.0), 0.0, 1.0);
    float bC = clamp(2.0 - abs(hue * 6.0 - 4.0), 0.0, 1.0);
    vec3 rainbow = vec3(rC, gC, bC);
    float rbAmt = 0.35 * smoothstep(0.15, 0.85, ringMirror);
```

```

    vec3 outCol = mix(base, base * rainbow, rbAmt);
    float vign = smoothstep(1.2, 0.4, length((tc - 0.5) * aspect));
    outCol *= vign;
    color = vec4(outCol, 1.0);
}

```

Shader #723 - new fractal

File: new_fractal.glsl

Renders a fractal pattern using iterative computation (many iterations).

Technique: Iterative fractal computation with complex-plane mapping - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,

```

```

        1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
        1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
        0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

float fractal(vec2 uv, float time) {
    vec2 z = uv;
    float iterations = 0.0;
    const float maxIterations = 50.0;
    for (float i = 0.0; i < maxIterations; i++) {
        z = vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + vec2(sin(time), cos(time));
        if (length(z) > 2.0) break;
        iterations += 1.0;
    }
    return iterations / maxIterations;
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float fractalValue = fractal(cos(uv*pingPong(time_f, 30.0)+1.0), time_f);
    vec3 fractalColor = rainbow(fractalValue);
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float r = length(uv);
    float theta = atan(uv.y, uv.x);
    theta += time_f * 5.0 + r * 15.0;
    uv = vec2(cos(theta), sin(theta)) * sin(r * time_f);
    vec4 blurred_color = blur(samp, tc, iResolution);
    vec3 blended_color = mix(blurred_color.rgb, fractalColor, 0.5);
    color = vec4(sin(blended_color * time_t), blurred_color.a);
}

```

Shader #724 - new fractal2

File: new_fractal2.glsl

Renders a fractal pattern using iterative computation (many iterations).

Technique: Iterative fractal computation with complex-plane mapping - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

```

```

float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}

vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}

vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}

vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

float fractal(vec2 uv, float time) {
    vec2 z = uv;
    float iterations = 0.0;
    const float maxIterations = 50.0;
    for (float i = 0.0; i < maxIterations; i++) {
        z = vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + vec2(sin(time), cos(time));
    }
}

```

```

        if (length(z) > 2.0) break;
        iterations += 1.0;
    }
    return iterations / maxIterations;
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float fractalValue = fractal(cos(uv*pingPong(time_f, 30.0)+1.0), time_f);
    vec3 fractalColor = rainbow(fractalValue);
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float r = length(uv);
    float theta = atan(uv.y, uv.x);
    theta += time_f * 5.0 + r * 15.0;
    uv = vec2(cos(theta * time_f), sin(theta * time_f)) * sin(r * time_f);
    vec4 blurred_color = blur(samp, tc, iResolution);
    vec3 blended_color = mix(blurred_color.rgb, fractalColor, 0.5);
    color = vec4(sin(blended_color * time_t), blurred_color.a);
}

```

Shader #820 - rainbow fractal

File: rainbow_fractal.glsl

Renders a fractal pattern using iterative computation (many iterations).

Technique: Iterative fractal computation with complex-plane mapping - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {

```

```

    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
float fractal(vec2 uv, float time) {
    vec2 z = uv;
    float iterations = 0.0;
    const float maxIterations = 50.0;
    for (float i = 0.0; i < maxIterations; i++) {
        z = vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + vec2(sin(time), cos(time));
        if (length(z) > 2.0) break;
        iterations += 1.0;
    }
    return iterations / maxIterations;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float fractalValue = fractal(uv, time_f);
    vec3 fractalColor = rainbow(fractalValue);
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float r = length(uv);
    float theta = atan(uv.y, uv.x);
    theta += time_f * 5.0 + r * 15.0;
    uv = vec2(cos(theta), sin(theta)) * r;
    vec4 blurred_color = blur(samp, tc, iResolution);
    vec3 blended_color = mix(blurred_color.rgb, fractalColor, 0.5);
    color = vec4(sin(blended_color * time_t), blurred_color.a);
}

```


Shader #821 - rainbow fractal random

File: rainbow_fractal_random.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. resolution of the window Utility functions Basic blur (kept as-is)

Technique: Bitwise XOR on integer color channels - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return (modVal <= length) ? modVal : (length * 2.0 - modVal);
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](
        0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
        1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
        1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
        2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
        2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
        2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
        2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
        1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
        1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
        0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5
    );
    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
```

```

    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
float fractal(vec2 uv, vec2 c)
{
    vec2 z = uv;
    const float maxIterations = 50.0;
    float iteration = 0.0;
    for (float i = 0.0; i < maxIterations; i++)
    {
        vec2 z_sq = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y);
        z = z_sq + c;
        if (length(z) > 2.0) break;
        iteration += 1.0;
    }
    return iteration / maxIterations;
}
vec2 getFractalUV(vec2 uv, float t)
{
    float cycleDuration = 6.0;
    float cycleIndex    = floor(t / cycleDuration);
    float cycleTime     = fract(t / cycleDuration) * cycleDuration;
    float zoomSpeed     = 0.5;
    float zoom          = pow(1.3, cycleTime * zoomSpeed);
    vec2 randOffset = random2(vec2(cycleIndex * 37.1234));
    randOffset *= 0.5;
    uv -= randOffset;
    uv /= zoom;
    uv += randOffset;
    return uv;
}
void main(void)
{
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float cycleIndex = floor(time_f / 6.0);
    vec2 cShift = random2(vec2(cycleIndex * 13.97)) * 1.0;
    vec2 c = vec2(sin(time_f * 0.7), cos(time_f * 0.9)) + cShift;
    vec2 zoomedUV = getFractalUV(uv, time_f);
    float fVal = fractal(zoomedUV, c);
    vec3 fractalColor = rainbow(fVal);
    vec4 blurred_color = blur(samp, tc, iResolution);
    float blendFactor = 0.5;
    vec3 blended_color = mix(blurred_color.rgb, fractalColor, blendFactor);
    float time_t = pingPong(time_f, 15.0) + 1.0;
    color = vec4(sin(blended_color * time_t), blurred_color.a);
}

```

Shader #826 - rainbow metal fractal

File: rainbow_metal_fractal.glsl

Generates procedural noise patterns combined with sinusoidal distortion. High-fidelity spectrum for the oily "fringe" colors Simple fractal noise function Scale and wrap coordinates to create fractal pattern

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
vec3 spectrum(float t) {
    return vec3(0.5 + 0.5 * cos(6.28318 * (t + vec3(0.0, 0.33, 0.67))));
}
float fractalNoise(vec2 uv) {
    float t = time_f * 0.1;
    float n = 0.0;
    float a = 1.0;
    for (int i = 0; i < 5; ++i) {
        n += a * texture(samp, uv).r;
        uv = fract(uv * 2.0);
        a *= 0.5;
    }
    return n / (1 - 0.5);
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;
    float r = length(uv);
    float angle = atan(uv.y, uv.x);
    float noiseValue = fractalNoise(tc * 5.0) * 0.1;
    float ripple = sin(angle * 10.0 + time_f + noiseValue) * 0.03;
    ripple += sin(angle * 25.0 - time_f * 2.0) * 0.01;
    float wave = sin(r * 20.0 - time_f * 4.0 + ripple * 10.0);
    float shift = ripple * 0.5 + wave * 0.01;
    float r_chan = texture(samp, tc + vec2(shift, 0.0)).r;
    float g_chan = texture(samp, tc).g;
    float b_chan = texture(samp, tc - vec2(shift, 0.0)).b;
    vec3 baseTex = vec3(r_chan, g_chan, b_chan);
    vec3 rainbow = spectrum(r - time_f * 0.5 + ripple + noiseValue * 0.2);
    float glowMask = smoothstep(0.5, 1.0, wave);
    vec3 coreGlow = vec3(1.0, 0.98, 0.9) * pow(wave, 2.0) * 2.0;
    vec3 finalColor = mix(baseTex, rainbow, glowMask * 0.35);
    finalColor = finalColor * sin(coreGlow * time_f);
    finalColor += (wave * ripple * 2.0) + noiseValue * 0.1;
    color = vec4(finalColor, 1.0);
}
```

Scale & Transform (29 shaders)

Shader #14 - af scale

File: af_scale.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
vec4 colorShift(vec4 col) {
    ivec4 i_col = ivec4(col * 255);
    for(int i = 0; i < 3; ++i) {
        i_col[i] = i_col[i]%255;
    }
    return vec4(i_col / 255.0);
}
void main(void) {
    float time_t = pingPong(time_f, 10.0) + 1.0;
    vec4 pix = blur(samp, tc, iResolution);
    pix = pix * time_t;
    pix = colorShift(pix);
    pix = cos(sin(pix * length(time_t)));
    color = pix;
}

```

Shader #15 - af scale2

File: af_scale2.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

vec4 colorShift(vec4 col) {
    return vec4(
        0.5 + 0.5 * cos(col.r * 3.14159265 * 0.5),
        0.5 + 0.5 * cos(col.g * 3.14159265 * 0.5),
        0.5 + 0.5 * cos(col.b * 3.14159265 * 0.5),
        col.a
    );
}

void main(void) {
    float time_t = pingPong(time_f, 10.0) + 1.0;
    vec4 pix = blur(samp, tc, iResolution);
    pix = pix * time_t;
    pix = colorShift(pix);
    pix.rgb = mix(vec3(1.0), pix.rgb, 0.8);
    color = pix;
}
```

Shader #16 - af scale3

File: af_scale3.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

vec4 colorShift(vec4 col) {
    return vec4(
        0.5 + 0.5 * cos(col.r * 3.14159265 * 0.5),
        0.5 + 0.5 * cos(col.g * 3.14159265 * 0.5),
        0.5 + 0.5 * cos(col.b * 3.14159265 * 0.5),
        col.a
    );
}

void main(void) {
    float time_t = pingPong(time_f, 10.0) + 1.0;
    vec4 pix = texture(samp, tc);
    pix = pix * 2.0;
    pix = colorShift(pix);
    pix.rgb = mix(vec3(1.0), pix.rgb, 0.8);
    color = pix;
}
```

Shader #17 - af scale puple

File: af_scale_puple.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations. Compute zoom in/out Rotation along Y-axis Camera transformation matrix

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

vec4 colorShift(vec4 col) {
    return vec4(
        0.5 + 0.5 * cos(col.r * 3.14159265 * 0.5),
        0.5 + 0.5 * cos(col.g * 3.14159265 * 0.5),
        0.5 + 0.5 * cos(col.b * 3.14159265 * 0.5),
        col.a
    );
}

mat3 getCameraMatrix(float time) {
    float zoom = 0.5 + 0.5 * sin(time * 0.5);
    float angleY = time;
    float c = cos(angleY);
    float s = sin(angleY);
```

```

    return mat3(
        zoom * c, 0.0, zoom * s,
        0.0, zoom, 0.0,
        -zoom * s, 0.0, zoom * c
    );
}

void main(void) {
    float time_t = pingPong(time_f, 10.0) + 1.0;
    mat3 camera = getCameraMatrix(time_f);
    vec2 transformedUV = (camera * vec3(tc * 2.0 - 1.0, 1.0)).xy * 0.5 + 0.5;
    vec4 pix = blur(samp, transformedUV, iResolution);
    pix = pix * time_t;
    pix = colorShift(pix);
    pix.rgb = mix(vec3(1.0), pix.rgb, 0.8);
    vec3 purple = vec3(0.6, 0.0, 0.6);
    pix.rgb = mix(pix.rgb, purple, 0.3);
    color = pix;
}

```

Shader #18 - af scale spectrum

File: af_scale_spectrum.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns. Adjust the phase shift based on time Adjust the phase shift here too Pass the current time to the color shift function

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
}

```



```

    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

vec4 colorShift(vec4 col, float time) {
    return vec4(
        0.5 + 0.5 * cos(col.r * 3.14159265 * (time + 1.0)),
        0.5 + 0.5 * cos(col.g * 3.14159265 * (time + 1.0)),
        0.5 + 0.5 * cos(col.b * 3.14159265 * (time + 1.0)),
        col.a
    );
}

void main(void) {
    float time_t = pingPong(time_f, 5.0) + 1.0;
    vec4 pix = blur(samp, tc, iResolution);
    pix = pix * time_t;
    pix = colorShift(pix, time_t);
    pix.rgb = mix(vec3(1.0), pix.rgb, 0.8);
    color = pix;
}

```

Shader #95 - cd zoom

File: cd_zoom.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 centeredCoord = tc - 0.5;
    float angle = time_f;
    float zoom = abs(sin(time_f)) * 2.0 + 0.5;
    centeredCoord *= zoom;
    vec2 rotatedCoord;
    rotatedCoord.x = centeredCoord.x * cos(angle) - centeredCoord.y * sin(angle);
    rotatedCoord.y = centeredCoord.x * sin(angle) + centeredCoord.y * cos(angle);
    rotatedCoord += 0.5;
    color = texture(samp, rotatedCoord);
}

```

Shader #96 - cd zoom out

File: cd_zoom_out.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 centeredCoord = tc - 0.5;
    float angle = time_f;
    float zoom = 2.5 - abs(sin(time_f)) * 4.0;
    centeredCoord *= zoom;
    vec2 rotatedCoord;
    rotatedCoord.x = centeredCoord.x * cos(angle) - centeredCoord.y * sin(angle);
    rotatedCoord.y = centeredCoord.x * sin(angle) + centeredCoord.y * cos(angle);
    rotatedCoord += 0.5;
    color = texture(samp, rotatedCoord);
}

```

Shader #145 - comp zoom

File: comp_zoom.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Scanline overlay using modular y-coordinate - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 compositeEffect(vec2 uv) {
    float offset = 0.01;
    vec3 col;
    col.r = texture(samp, uv + vec2(offset, 0.0)).r;
    col.g = texture(samp, uv).g;
    col.b = texture(samp, uv - vec2(offset, 0.0)).b;
    float noise = fract(sin(dot(uv.xy, vec2(12.9898, 78.233))) * 43758.5453);
    col += noise * 0.05;
    float scanline = sin(uv.y * iResolution.y * 1.5) * 0.1;
    col -= scanline;
    float bleed = sin(uv.y * iResolution.y * 0.2 + time_f * 5.0) * 0.005;
    col.r += bleed * 0.002;
    col.b -= bleed * 0.002;
    return col;
}
void main(void) {
    vec2 ar = vec2(iResolution.x / iResolution.y, 1.0);
    vec2 c = vec2(0.5);
    float amp = 0.2;
    float speed = 0.8;
    float zoom = 1.0 + amp * sin(time_f * speed);
    vec2 p = (tc - c) * ar;
    p /= zoom;
    vec2 uv = p / ar + c;
    uv = clamp(uv, 0.0, 1.0);
    vec3 col = compositeEffect(uv);
}

```

```
    color = vec4(col, 1.0);  
}
```

Shader #177 - cyclone zoom

File: cyclone_zoom.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core  
out vec4 color;  
uniform sampler2D samp;  
uniform vec2 iResolution;  
uniform float time_f;  
void main(void){  
    vec2 normCoord = gl_FragCoord.xy / iResolution.xy;  
    float aspect = iResolution.x / iResolution.y;  
    vec2 p = normCoord - 0.5;  
    p.x *= aspect;  
    float eps = 1e-6;  
    float r = length(p) + eps;  
    float theta = atan(p.y, p.x);  
    float base = 1.7;  
    float period = log(base);  
    float t = time_f * 0.5;  
    float k = fract((log(r) - t) / period);  
    float rw = exp(k * period);  
    float twistAmount = 15.0;  
    theta += (1.0 - clamp(rw, 0.0, 1.0)) * twistAmount * sin(time_f) + t * 1.5707963;  
    vec2 q = vec2(cos(theta), sin(theta)) * rw;  
    q.x /= aspect;  
    vec2 uv = fract(q + 0.5);  
    color = texture(samp, uv);  
}
```

Shader #326 - expand contract

File: expand_contract.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core  
out vec4 color;  
in vec2 tc;  
uniform sampler2D samp;  
uniform vec2 iResolution;  
uniform float time_f;  
uniform float amp_peak;  
uniform float amp_rms;  
uniform float amp_smooth;  
uniform float amp_low;  
uniform float amp_mid;  
uniform float amp_high;
```

```

uniform float iamp;
void main(void) {
    vec2 normCoord = (tc * 2.0) - vec2(1.0, 1.0);
    normCoord.x *= iResolution.x / iResolution.y;
    float dist = length(normCoord);
    float stretchMagnitude = (0.5 + amp_low * 0.5) + sin(time_f) * 0.50;
    float stretchFactor = 1.0 + pow(dist, 2.0) * stretchMagnitude;
    vec2 stretchedCoord = normCoord / stretchFactor;
    stretchedCoord.x *= iResolution.y / iResolution.x;
    stretchedCoord = (stretchedCoord + vec2(1.0, 1.0)) / 2.0;
    stretchedCoord = clamp(stretchedCoord, 0.0, 1.0);
    color = texture(samp, stretchedCoord);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #327 - expand mul

File: expand_mul.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    float t = sin(time_f) * 0.5 + 0.5;
    vec2 center = vec2(0.5, 0.5);
    vec2 normTC = tc - center;
    float time_t = mod(time_f, 25);
    vec2 squeezed = sin(normTC * time_t) * (1.0 - t) + center;
    vec4 texColor = texture(samp, squeezed);
    color = texColor;
}

```

Shader #328 - expand mul fast

File: expand_mul_fast.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {

```

```

float t = sin(time_f) * 0.5 + 0.5;
vec2 center = vec2(0.5, 0.5);
vec2 normTC = tc - center;
float time_t = mod(sin(time_f * t), 25);
vec2 squeezed = sin(normTC * time_t) * (1.0 - t) + center;
vec4 texColor = texture(samp, squeezed);
color = texColor;
}

```

Shader #329 - expand spot

File: expand_spot.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
float rand(float seed) {
    return fract(sin(seed) * 43758.5453123);
}
vec2 randomPos(float seed) {
    return vec2(rand(seed), rand(seed + 1.0));
}
vec2 expand(vec2 pos, vec2 center) {
    float t = mod(time_f, 4.0);
    float scale = abs(sin(t * 3.14159 / 2.0));
    return mix(center, pos, scale);
}
void main(void) {
    float cycle = floor(time_f / 4.0);
    vec2 center = randomPos(cycle);
    vec2 pos = expand(tc, center);
    vec4 tcolor = texture(samp, pos);
    color = tcolor;
}

```

Shader #384 - frac shader02 dmdi6i zoom

File: frac_shader02_dmdi6i_zoom.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {

```

```

    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    float zoomPhase = time_f * 0.12;
    float zCycle = floor(zoomPhase);
    float zLocal = fract(zoomPhase);
    float tri = 1.0 - abs(zLocal * 2.0 - 1.0);
    float sgn = mix(-1.0, 1.0, step(0.5, mod(zCycle, 2.0)));
    float depth = 1.0 + zCycle * 0.35;
    float zoomExp = sgn * tri * depth;
    float zoom = pow(1.45, zoomExp);
    vec2 z = tc - m;
    z.x *= aspect;
    z /= zoom;
    z.x /= aspect;
    vec2 zoomTC = fract(z + m);
    vec4 baseTex = texture(samp, zoomTC);
    vec2 uv = zoomTC * 2.0 - 1.0;

```

```

uv.x *= aspect;
float r = pingPong(sin(length(uv) * time_f), 5.0);
float radius = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
vec3 baseCol = preBlendColor(zoomTC);
float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
vec2 kUV = reflectUV(zoomTC, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((zoomTC - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #422 - frac zoom1

File: frac_zoom1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
 - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation -
 Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
```

```

    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    vec2 pa = abs(p);
    vec2 ps = sin(abs(p));
    float mixAmt = 0.5 + 0.5 * sin(time_f * 0.2);
    vec2 pm = mix(pa, ps, mixAmt);
    return max(pm.x, pm.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec2 uv = tc * 2.0 - 1.0;

```

```

uv.x *= aspect;
float rLen = length(uv);
float radialWave = pingPong(sin(rLen * time_f), 5.0);
float radius = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radius, radius - 0.25, radialWave);
vec3 baseCol = preBlendColor(tc);
float audioLevel = clamp(amp * 0.75 + uamp * 0.40, 0.0, 2.0);
float tZoom = pingPong(time_f * 0.12 + audioLevel * 0.35, 1.0);
float zoomBase = mix(0.90, 2.40, tZoom);
float zoomAudio = 1.0 + audioLevel * 0.75;
float foldZoom = zoomBase * zoomAudio;
float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33 + audioLevel * 0.25);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + audioLevel * 0.05), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0 + audioLevel);
float tz = time_f * 0.65 + audioLevel * 0.35;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6 + audioLevel * 3.0);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
float zoomShell = 1.0 + audioLevel * 0.8;
vec2 pwrap = vec2(cos(ang), sin(ang)) * (rw * zoomShell);
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3 + audioLevel)) * vec2(1.0,
↳ 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2 + audioLevel * 2.0));
ring *= pingPong(time_f * PI, 5.0);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0 + audioLevel *
↳ 6.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
float mixGlow = pingPong(glow * PI, 5.0) * (0.4 + 0.4 * clamp(audioLevel, 0.0, 1.0));
vec3 finalRGB = mix(baseTex.rgb, outCol, mixGlow);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #423 - frac zoom2

File: frac_zoom2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
```

```

    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    vec2 pa = abs(p);
    vec2 ps = sin(abs(p));
    float mixAmt = 0.5 + 0.5 * sin(time_f * 0.2);
    vec2 pm = mix(pa, ps, mixAmt);
    return max(pm.x, pm.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);

```

```

    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
vec2 triUV(vec2 uv, float scale) {
    vec2 p = uv * scale;
    vec2 g = floor(p);
    vec2 f = fract(p);
    if (f.x + f.y > 1.0) {
        g += 1.0;
        f = 1.0 - f.yx;
    }
    return (g + f) / scale;
}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= aspect;
    float rLen = length(uv);
    float radialWave = pingPong(sin(rLen * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - 0.25, radialWave);
    vec3 baseCol = preBlendColor(tc);
    float audioLevel = clamp(amp * 0.75 + uamp * 0.40, 0.0, 2.0);
    float tZoom = pingPong(time_f * 0.12 + audioLevel * 0.35, 1.0);
    float zoomBase = mix(0.95, 1.85, tZoom);
    float zoomAudio = 1.0 + audioLevel * 0.45;
    float foldZoom = zoomBase * zoomAudio;
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33 + audioLevel * 0.25);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + audioLevel * 0.05), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0 + audioLevel);
    float tz = time_f * 0.65 + audioLevel * 0.35;
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6 + audioLevel * 3.0);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    rw = clamp(rw, 0.45, 2.5);
    float zoomShell = 1.0 + audioLevel * 0.5;
    vec2 pwrap = vec2(cos(ang), sin(ang)) * (rw * zoomShell);
    float triScale = 80.0 * (1.0 + 0.3 * audioLevel);
    vec2 baseTri = triUV(fract(pwrap / ar + m), triScale);
    vec2 tri0 = baseTri;
    vec2 tri1 = triUV(fract(pwrap * 1.045 / ar + m), triScale);
    vec2 tri2 = triUV(fract(pwrap * 0.955 / ar + m), triScale);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * (((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3 + audioLevel)) * vec2(1.0,
    ↪ 1.0 / aspect));
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(tri0 + off);
    vec3 gC = preBlendColor(tri1);

```

```

    vec3 bC = preBlendColor(tri2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2 + audioLevel * 2.0));
    ring *= pingPong(time_f * PI, 5.0);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0 + audioLevel *
    ↪ 6.0);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
    vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
    outCol += bloom;
    outCol = mix(outCol, baseCol, 0.18);
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    float mixGlow = pingPong(glow * PI, 5.0) * (0.4 + 0.4 * clamp(audioLevel, 0.0, 1.0));
    vec3 finalRGB = mix(baseTex.rgb, outCol, mixGlow);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #424 - frac zoom3

File: frac_zoom3.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

```

```

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
}

```



```

float aspect = iResolution.x / iResolution.y;
vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
float r = length(p);
float t = time_f;
vec3 neon = neonPalette(t + r * 1.3);
float neonAmt = smoothstep(0.1, 0.8, r);
neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
vec3 grad = mix(tex, neon, neonAmt);
grad = mix(grad, tex, 0.2);
grad = softTone(grad);
return grad;
}

float diamondRadius(vec2 p) {
    vec2 pa = abs(p);
    vec2 ps = sin(abs(p));
    float mixAmt = 0.5 + 0.5 * sin(time_f * 0.2);
    vec2 pm = mix(pa, ps, mixAmt);
    return max(pm.x, pm.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 triUV(vec2 uv, float scale) {
    vec2 p = uv * scale;
    vec2 g = floor(p);
    vec2 f = fract(p);
    if (f.x + f.y > 1.0) {
        g += 1.0;
        f = 1.0 - f.yx;
    }
    return (g + f) / scale;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    float aspect = iResolution.x / iResolution.y;
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec2 uv = tc * 2.0 - 1.0;
    uv.x *= aspect;
    float rLen = length(uv);
    float radialWave = pingPong(sin(rLen * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - 0.25, radialWave);
    vec3 baseCol = preBlendColor(tc);
    float audioLevel = clamp(amp * 0.75 + uamp * 0.40, 0.0, 2.0);
    float tZoom = pingPong(time_f * 0.12 + audioLevel * 0.35, 1.0);
    float zoomBase = mix(0.98, 1.35, tZoom);
    float zoomAudio = 1.0 + audioLevel * 0.30;
    float foldZoom = zoomBase * zoomAudio;
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33 + audioLevel * 0.25);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + audioLevel * 0.05), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
}

```

```

float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0 + audioLevel);
float tz = time_f * 0.65 + audioLevel * 0.35;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6 + audioLevel * 3.0);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
rw = clamp(rw, 0.7, 1.6);
float zoomShell = 1.0 + audioLevel * 0.35;
vec2 pwrap = vec2(cos(ang), sin(ang)) * (rw * zoomShell * 0.85);
float triScale = 24.0 * (1.0 + 0.15 * audioLevel);
vec2 baseTri = triUV(fract(pwrap / ar + m), triScale);
vec2 tri0 = baseTri;
vec2 tri1 = triUV(fract(pwrap * 1.045 / ar + m), triScale);
vec2 tri2 = triUV(fract(pwrap * 0.955 / ar + m), triScale);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3 + audioLevel)) * vec2(1.0,
↳ 1.0 / aspect);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(tri0 + off);
vec3 gC = preBlendColor(tri1);
vec3 bC = preBlendColor(tri2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2 + audioLevel * 2.0));
ring *= pingPong(time_f * PI, 5.0);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0 + audioLevel *
↳ 6.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
float mixGlow = pingPong(glow * PI, 5.0) * (0.4 + 0.4 * clamp(audioLevel, 0.0, 1.0));
vec3 finalRGB = mix(baseTex.rgb, outCol, mixGlow);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #425 - frac zoom4

File: frac_zoom4.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;

```

```

uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {

```

```

    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 complexIter(vec2 z, vec2 c) {
    return vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + c;
}

vec2 fractalZoomUV(vec2 uv, vec2 center, float aspect, float t) {
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    float tZoom = t * 0.23;
    float level = floor(tZoom);
    float local = pingPong(fract(tZoom), 1.0);
    float zoom = pow(2.15, level + local * 0.98);
    zoom = min(zoom, 50000.0);
    p *= zoom;
    vec2 z = p * 0.42;
    vec2 c = vec2(0.32 + 0.02 * sin(t * 0.17),
        0.043 + 0.015 * cos(t * 0.21));
    for (int i = 0; i < 7; i++) {
        z = complexIter(z, c);
    }
    float warp = 0.30 + 0.15 * sin(t * 0.5);
    float lenZ = max(1.0, length(z));

```

```

    p += warp * (z / lenZ);
    p /= zoom;
    p.x /= aspect;
    return p + center;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 baseUV0 = pwrap / ar + m;
    vec2 baseUV1 = (pwrap * 1.045) / ar + m;
    vec2 baseUV2 = (pwrap * 0.955) / ar + m;
    vec2 u0 = fract(fractalZoomUV(baseUV0, m, aspect, time_f));
    vec2 u1 = fract(fractalZoomUV(baseUV1, m, aspect, time_f + 0.7));
    vec2 u2 = fract(fractalZoomUV(baseUV2, m, aspect, time_f + 1.4));
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↪ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
    vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
    outCol += bloom;
    outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
}

```

```

    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #426 - frac zoom5

File: frac_zoom5.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

```

```

}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);

```

```

    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
vec2 complexIter(vec2 z, vec2 c) {
    return vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + c;
}
vec2 fractalZoomUV(vec2 uv, vec2 center, float aspect, float t) {
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    float tZoom = t * 0.23;
    float level = floor(tZoom);
    float local = pingPong(fract(tZoom), 1.0);
    float zoom = pow(2.15, level + local * 0.98);
    zoom = min(zoom, 50000.0);
    p *= zoom;
    vec2 z = p * 0.42;
    vec2 c = vec2(0.32 + 0.02 * sin(t * 0.17),
        0.043 + 0.015 * cos(t * 0.21));
    for (int i = 0; i < 7; i++) {
        z = complexIter(z, c);
    }
    float warp = 0.30 + 0.15 * sin(t * 0.5);
    float lenZ = max(1.0, length(z));
    p += warp * (z / lenZ);
    float ang = t * 0.45 * 6.28318530718;
    float cs = cos(ang);
    float sn = sin(ang);
    p = mat2(cs, -sn, sn, cs) * p;
    p /= zoom;
    p.x /= aspect;
    return p + center;
}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 baseUV0 = pwrap / ar + m;

```



```

vec2 baseUV1 = (pwrap * 1.045) / ar + m;
vec2 baseUV2 = (pwrap * 0.955) / ar + m;
vec2 u0 = fract(fractalZoomUV(baseUV0, m, aspect, time_f));
vec2 u1 = fract(fractalZoomUV(baseUV1, m, aspect, time_f + 0.7));
vec2 u2 = fract(fractalZoomUV(baseUV2, m, aspect, time_f + 1.4));
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #427 - frac zoom6

File: frac_zoom6.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {

```

```

    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.10, 0.85);
    vec3 blue = vec3(0.00, 0.60, 1.0);
    vec3 green = vec3(0.05, 1.05, 0.50);
    float ph = fract(t * 0.15);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    vec3 col = a * k1 + b * k2 + c * k3;
    col = pow(col, vec3(0.85));
    return normalize(col) * 1.4;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.9));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.7);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;

```

```

    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

vec2 complexIter(vec2 z, vec2 c) {
    return vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + c;
}

vec2 fractalZoomUV(vec2 uv, vec2 center, float aspect, float t) {
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    float tZoom = t * 0.23;
    float level = floor(tZoom);
    float local = pingPong(fract(tZoom), 1.0);
    float zoom = pow(2.15, level + local * 0.98);
    zoom = min(zoom, 50000.0);
    p *= zoom;
    vec2 z = p * 0.42;
    vec2 c = vec2(0.32 + 0.02 * sin(t * 0.17),
        0.043 + 0.015 * cos(t * 0.21));
    for (int i = 0; i < 7; i++) {
        z = complexIter(z, c);
    }
    float warp = 0.30 + 0.15 * sin(t * 0.5);
    float lenZ = max(1.0, length(z));
    p += warp * (z / lenZ);
    float ang = t * 0.45 * 6.28318530718;
    float cs = cos(ang);
    float sn = sin(ang);
    p = mat2(cs, -sn, sn, cs) * p;
    p /= zoom;
    p.x /= aspect;
    return p + center;
}

void main(void) {

```

```

vec4 baseTex = texture(samp, tc);
vec2 uv = tc * 2.0 - 1.0;
float aspect = iResolution.x / iResolution.y;
uv.x *= aspect;
float r = pingPong(sin(length(uv) * time_f), 5.0);
float radius = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
vec2 ar = vec2(aspect, 1.0);
vec3 baseCol = preBlendColor(tc);
float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 baseUV0 = pwrap / ar + m;
vec2 baseUV1 = (pwrap * 1.045) / ar + m;
vec2 baseUV2 = (pwrap * 0.955) / ar + m;
vec2 u0 = fract(fractalZoomUV(baseUV0, m, aspect, time_f));
vec2 u1 = fract(fractalZoomUV(baseUV1, m, aspect, time_f + 0.7));
vec2 u2 = fract(fractalZoomUV(baseUV2, m, aspect, time_f + 1.4));
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, 1.18, vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + 0.5 * sin(time_f * 2.3 + rD * 30.0 + k * 14.0);
vec3 neonGlobal = neonPalette(time_f + rD * 0.6 + k * 0.8);
vec3 outCol = kaleidoRGB;
outCol = mix(outCol, neonGlobal, 0.22 + 0.25 * ring);
outCol *= (0.78 + 0.28 * ring) * (0.85 + 0.18 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.26 + pow(max(outCol - 0.55, 0.0), vec3(2.2)) * 0.22;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.14);
outCol = clamp(outCol, vec3(0.03), vec3(1.15));
vec3 finalRGB = mix(baseTex.rgb, outCol, min(0.95, pingPong(glow * PI, 5.0) * 0.95));
finalRGB = clamp(finalRGB, 0.0, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);

```

```
}
```

Shader #428 - frac zoom7

File: frac_zoom7.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - RGB $\leftrightarrow$ HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
```

```

    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

```

```

}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    vec3 outCol = kaleidoRGB;
    outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
    vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
    outCol += bloom;
    outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
    outCol = clamp(outCol, vec3(0.05), vec3(0.97));
    vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    finalRGB *= 1.0 + _ab * 0.6;
    finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↵ amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(finalRGB, baseTex.a);
}

```

Shader #429 - frac zoom8

File: frac_zoom8.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}
```



```

}
vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}
vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}
vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}
vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}
float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;

```

```

uv.x *= aspect;
float r = pingPong(sin(length(uv) * time_f), 5.0);
float radius = sqrt(aspect * aspect + 1.0) + 0.5;
float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
vec2 ar = vec2(aspect, 1.0);
vec3 baseCol = preBlendColor(tc);
float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
vec2 kUV = reflectUV(tc, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = abs(p);
if (q.y > q.x) q = q.yx;
float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
float period = log(base) * pingPong(time_f * PI, 5.0);
float tz = pingPong(time_f * PI/4, 3.0) * 0.65;
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + pingPong(time_f * PI/2, 3.0) * 0.6)
↳ ;
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap * 1.045) / ar + m);
vec2 u2 = fract((pwrap * 0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect)
↳ ;
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
vec3 rC = preBlendColor(u0 + off);
vec3 gC = preBlendColor(u1);
vec3 bC = preBlendColor(u2 - off);
vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + pingPong(time_f * PI, 5.0) * 1.2));
ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(pingPong(time_f * PI, 5.0) * 2.0 + rD * 28.0 + k *
↳ 12.0);
vec3 outCol = kaleidoRGB;
outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(baseTex.rgb, outCol, pingPong(glow * PI, 5.0) * 0.8);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
finalRGB *= 1.0 + _ab * 0.6;
finalRGB = mix(finalRGB, finalRGB * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
↳ amp_high, 0.0, 1.0) * 0.25), _ab);
color = vec4(finalRGB, baseTex.a);
}

```

Shader #561 - gt rotate

File: gt_rotate.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    float angle = time_f;
    vec2 tc_centered = tc - center;
    float cos_theta = cos(angle);
    float sin_theta = sin(angle);
    vec2 tc_rotated;
    tc_rotated.x = tc_centered.x * cos_theta - tc_centered.y * sin_theta;
    tc_rotated.y = tc_centered.x * sin_theta + tc_centered.y * cos_theta;
    tc_rotated += center;
    vec4 texColor = texture(samp, tc_rotated);
    color = texColor;
}
```

Shader #875 - ResizeVertical cache

File: ResizeVertical_cache.glsl

Blends the current frame with cached previous frames to create motion trails and temporal echoes.
History/Cache Frames (1-8) Updated to 8 slots Speed of the cycling effect

Technique: Temporal feedback using cached frame buffers - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform sampler2D samp1;
uniform sampler2D samp2;
uniform sampler2D samp3;
uniform sampler2D samp4;
uniform sampler2D samp5;
uniform sampler2D samp6;
uniform sampler2D samp7;
uniform sampler2D samp8;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    int square_size = 64;
    int total_cache_slots = 8;
    int start_index = int(time_f * 5.0);
    int start_dir = 1;
    int y = int(tc.y * iResolution.y);
```

```

int block_row = y / square_size;
int period = 2 * (total_cache_slots - 1);
if (period <= 0) period = 1;
int start_pos = (start_dir == 1) ? start_index : (2 * (total_cache_slots - 1)) - start_index;
int pos = (start_pos + block_row) % period;
if (pos < 0) pos += period;
int frame_index = (pos < total_cache_slots) ? pos : period - pos;
if (frame_index < 0) frame_index = 0;
if (frame_index >= total_cache_slots) frame_index = total_cache_slots - 1;
vec4 currentFrame = texture(samp, tc);
vec4 historyFrame;
if (frame_index == 0) {
    historyFrame = texture(samp1, tc);
} else if (frame_index == 1) {
    historyFrame = texture(samp2, tc);
} else if (frame_index == 2) {
    historyFrame = texture(samp3, tc);
} else if (frame_index == 3) {
    historyFrame = texture(samp4, tc);
} else if (frame_index == 4) {
    historyFrame = texture(samp5, tc);
} else if (frame_index == 5) {
    historyFrame = texture(samp6, tc);
} else if (frame_index == 6) {
    historyFrame = texture(samp7, tc);
} else {
    historyFrame = texture(samp8, tc);
}
color = (currentFrame * 0.5) + (historyFrame * 0.5);
color.a = 1.0;
}

```

Shader #907 - rotate xyz

File: rotate_xyz.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: 2D rotation matrix - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
mat4 rotationMatrixX(float angle) {
    return mat4(
        1.0, 0.0, 0.0, 0.0,
        0.0, cos(angle), -sin(angle), 0.0,
        0.0, sin(angle), cos(angle), 0.0,
        0.0, 0.0, 0.0, 1.0
    );
}
mat4 rotationMatrixY(float angle) {
    return mat4(
        cos(angle), 0.0, sin(angle), 0.0,
        0.0, 1.0, 0.0, 0.0,
        -sin(angle), 0.0, cos(angle), 0.0,
        0.0, 0.0, 0.0, 1.0
    );
}

```

```

    );
}
mat4 rotationMatrixZ(float angle) {
    return mat4(
        cos(angle), -sin(angle), 0.0, 0.0,
        sin(angle), cos(angle), 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    );
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 pos = tc - center;
    float angleX = time_f * 0.1;
    float angleY = time_f * 0.2;
    float angleZ = time_f * 0.3;
    mat4 rotationX = rotationMatrixX(angleX);
    mat4 rotationY = rotationMatrixY(angleY);
    mat4 rotationZ = rotationMatrixZ(angleZ);
    vec4 rotatedPos = rotationZ * rotationY * rotationX * vec4(pos, 0.0, 1.0);
    vec2 rotatedTc = rotatedPos.xy + center;
    color = texture(samp, rotatedTc);
}

```

Shader #908 - rotate xyz zoom

File: rotate_xyz_zoom.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
mat4 rotationMatrixX(float angle) {
    return mat4(
        1.0, 0.0, 0.0, 0.0,
        0.0, cos(angle), -sin(angle), 0.0,
        0.0, sin(angle), cos(angle), 0.0,
        0.0, 0.0, 0.0, 1.0
    );
}
mat4 rotationMatrixY(float angle) {
    return mat4(
        cos(angle), 0.0, sin(angle), 0.0,
        0.0, 1.0, 0.0, 0.0,
        -sin(angle), 0.0, cos(angle), 0.0,
        0.0, 0.0, 0.0, 1.0
    );
}
mat4 rotationMatrixZ(float angle) {
    return mat4(
        cos(angle), -sin(angle), 0.0, 0.0,
        sin(angle), cos(angle), 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
    );
}

```

```

        0.0, 0.0, 0.0, 1.0
    );
}
mat4 scaleMatrix(float scale) {
    return mat4(
        scale, 0.0, 0.0, 0.0,
        0.0, scale, 0.0, 0.0,
        0.0, 0.0, scale, 0.0,
        0.0, 0.0, 0.0, 1.0
    );
}
void main(void) {
    vec2 center = vec2(0.5, 0.5);
    vec2 pos = tc - center;
    float angleX = time_f * 0.1;
    float angleY = time_f * 0.2;
    float angleZ = time_f * 0.3;
    float scale = 1.0 + 0.5 * sin(time_f * 0.5);
    mat4 rotationX = rotationMatrixX(angleX);
    mat4 rotationY = rotationMatrixY(angleY);
    mat4 rotationZ = rotationMatrixZ(angleZ);
    mat4 scaleMat = scaleMatrix(scale);
    vec4 transformedPos = scaleMat * rotationZ * rotationY * rotationX * vec4(pos, 0.0, 1.0);
    vec2 transformedTc = transformedPos.xy + center;
    color = texture(samp, transformedTc);
}

```

Shader #912 - scale

File: scale.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
}

```

```

    color[0] += color[0]*alpha;
    color[1] += color[1]*alpha;
    color[2] += color[2]*alpha;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #1091 - zoom fish

File: zoom_fish.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Applies matrix transformations.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
mat3 rotX(float a){float s=sin(a),c=cos(a);return mat3(1,0,0, 0,c,-s, 0,s,c);}
mat3 rotY(float a){float s=sin(a),c=cos(a);return mat3(c,0,s, 0,1,0, -s,0,c);}
mat3 rotZ(float a){float s=sin(a),c=cos(a);return mat3(c,-s,0, s,c,0, 0,0,1);}
void main(void){
    float aspect=iResolution.x/iResolution.y;
    vec2 ar=vec2(aspect,1.0);
    vec2 m=(iMouse.z>0.5)?(iMouse.xy/iResolution):vec2(0.5);
    vec2 p2=(tc-m)*ar;
    float ax=0.25*sin(time_f*0.7);
    float ay=0.25*cos(time_f*0.6);
    float az=time_f*0.5;
    vec3 p3=vec3(p2,1.0);
    mat3 R=rotZ(az)*rotY(ay)*rotX(ax);
    vec3 r=R*p3;
    float k=0.6;
    float zf=1.0/(1.0+r.z*k);
    vec2 q=r.xy*zf;
    float dist=length(p2);
    float scale=1.0+0.2*sin(dist*15.0-time_f*2.0);
    q*=scale;
    float eps=1e-6;
    float base=1.7;
    float period=log(base);
    float t=time_f*0.45;
    float rad=length(q)+eps;
    float ang=atan(q.y,q.x)+t*0.35*6.2831853;
    float kf=fract((log(rad)-t)/period);
    float rw=exp(kf*period);
    vec2 qwrap=vec2(cos(ang),sin(ang))*rw;
    vec2 uv=qwrap/ar+m;
    uv=fract(uv);
}

```

```
    color=texture(samp,uv);  
}
```

Shader #1093 - zoom pong

File: zoom_pong.glsl

Applies a time-animated visual transformation to the input texture. Responds to mouse position for interactive control.

Technique: Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core  
out vec4 color;  
in vec2 tc;  
uniform sampler2D samp;  
uniform vec2 iResolution;  
uniform float time_f;  
uniform vec4 iMouse;  
const float PI = 3.1415926535897932384626433832795;  
void main(void) {  
    float aspect = iResolution.x / iResolution.y;  
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);  
    float zoomPhase = time_f * 0.12;  
    float zLocal = fract(zoomPhase);  
    float tri = 1.0 - abs(zLocal * 2.0 - 1.0);  
    float minZoom = 0.3;  
    float maxZoom = 4.0;  
    float zoom = mix(minZoom, maxZoom, tri);  
    vec2 z = tc - m;  
    z.x *= aspect;  
    z /= zoom;  
    z.x /= aspect;  
    vec2 zoomTC = fract(z + m);  
    vec4 baseTex = texture(samp, zoomTC);  
    color = baseTex;  
}
```

Blur & Smooth (19 shaders)

Shader #30 - and smooth

File: and_smooth.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```
#version 330  
in vec2 tc;  
out vec4 color;  
uniform sampler2D samp;  
uniform float time_f;  
uniform vec2 iResolution;  
vec4 add_RGB(vec4 icolor, vec4 source) {  
    ivec3 int_color;  
    ivec4 isource = ivec4(source * 255.0);  
    for(int i = 0; i < 3; ++i) {
```



```

        int_color[i] = int(255.0 * icolor[i]);
        int_color[i] = (int_color[i] + isource[i]) % 255;
        icolor[i] = float(int_color[i])/255.0;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[49];
    kernel[0] = 0.5; kernel[1] = 1.0; kernel[2] = 2.0; kernel[3] = 2.5; kernel[4] = 2.0; kernel[5] =
    ↪ 1.0; kernel[6] = 0.5;
    kernel[7] = 1.0; kernel[8] = 2.0; kernel[9] = 3.0; kernel[10] = 3.5; kernel[11] = 3.0; kernel[12] =
    ↪ 2.0; kernel[13] = 1.0;
    kernel[14] = 2.0; kernel[15] = 3.0; kernel[16] = 4.0; kernel[17] = 4.5; kernel[18] = 4.0; kernel[19]
    ↪ = 3.0; kernel[20] = 2.0;
    kernel[21] = 2.5; kernel[22] = 3.5; kernel[23] = 4.5; kernel[24] = 5.0; kernel[25] = 4.5; kernel[26]
    ↪ = 3.5; kernel[27] = 2.5;
    kernel[28] = 2.0; kernel[29] = 3.0; kernel[30] = 4.0; kernel[31] = 4.5; kernel[32] = 4.0; kernel[33]
    ↪ = 3.0; kernel[34] = 2.0;
    kernel[35] = 1.0; kernel[36] = 2.0; kernel[37] = 3.0; kernel[38] = 3.5; kernel[39] = 3.0; kernel[40]
    ↪ = 2.0; kernel[41] = 1.0;
    kernel[42] = 0.5; kernel[43] = 1.0; kernel[44] = 2.0; kernel[45] = 2.5; kernel[46] = 2.0; kernel[47]
    ↪ = 1.0; kernel[48] = 0.5;
    float kernelSum = 272.0;
    for (int x = -3; x <= 3; ++x) {
        for (int y = -3; y <= 3; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 3) * 7 + (x + 3)];
        }
    }
    return result / kernelSum;
}
void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = add_RGB(tcolor, tcolor * time_t);
}

```

Shader #66 - bloom

File: bloom.glsl

Converts to HSV color space for hue/saturation manipulation, then converts back to RGB.

Technique: RGB <-> HSV color space conversion - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp_peak;

```

```

uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}
vec3 rgb2hsv(vec3 c) {
    vec4 K = vec4(0.0, -1.0/3.0, 2.0/3.0, -1.0);
    vec4 p = mix(vec4(c.bg, K.wz), vec4(c.gb, K.xy), step(c.b, c.g));
    vec4 q = mix(vec4(p.xyw, c.r), vec4(c.r, p.yzx), step(p.x, c.r));
    float d = q.x - min(q.w, q.y);
    float e = 1.0e-10;
    return vec3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e), q.x);
}
vec3 hsv2rgb(vec3 hsv) {
    vec3 rgb = clamp(abs(mod(hsv.x*6.0 + vec3(0.0,4.0,2.0), 6.0) - 3.0) - 1.0, 0.0, 1.0);
    return hsv.z * mix(vec3(1.0), rgb, hsv.y);
}
void main(void) {
    vec4 originalColor = blur(samp, tc, iResolution);
    vec3 baseColor = rgb2hsv(originalColor.rgb);
    float bloomSpeed = 1.0 + amp_smooth * 2.0;
    baseColor = 0.5 + sin(baseColor * time_f * bloomSpeed);
}

```

```

    vec3 rgbColor = hsv2rgb(baseColor);
    color = vec4(rgbColor.rgb, originalColor.a);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
    ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
};

```

Shader #73 - blur offset

File: blur_offset.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
void main(void)
{
    color = texture(samp, tc);
    vec4 color_one = texture(samp, tc-0.01);
    vec4 color_two = texture(samp, tc-0.02);
    vec4 color_three = texture(samp, tc-0.03);
    vec4 color_off = color_one + color_two + color_three + color;
    color_off = color_off / 4;
    color = (color * 0.3) + (color_two * 0.5) + (color_one * 0.3);
    color = (0.5 * color) + (0.5 * color_off);
}

```

Shader #74 - blur offset color fade

File: blur_offset_color_fade.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330

```

```

in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
vec4 color_blend(vec4 color) {
    vec4 color2 = color;
    ivec4 color_source = ivec4(color * 255);
    color = color*alpha;
    ivec4 color_i = ivec4(color * 255);
    for(int i = 0; i < 3; ++i) {
        if(color_i[i] >= 255)
            color_i[i] = color_i[i]%255;
        if(color_source[i] >= 255)
            color_source[i] = color_source[i]%255;
        color_i[i] = color_i[i] ^ color_source[i];
        color[i] = float(color_i[i])/255;
    }
    for(int i = 0; i < 3; ++i)
        if(color[i] < 0.2) color[i] = color2[i];
    return color;
}
void main(void)
{
    color = texture(samp, tc);
    vec4 color_one = texture(samp, tc-0.01);
    vec4 color_two = texture(samp, tc-0.02);
    vec4 color_three = texture(samp, tc-0.03);
    vec4 color_off = color_one + color_two + color_three + color;
    color_off = color_off / 4;
    color = (color * 0.3) + (color_two * 0.5) + (color_one * 0.3);
    color = (0.5 * color) + (0.5 * color_off);
    color = color_blend(color);
}

```

Shader #75 - blurry

File: blurry.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;

```

```

uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
void main(void)
{
    color = texture(samp, tc);
    vec4 color_one = texture(samp, tc-0.01);
    vec4 color_two = texture(samp, tc-0.02);
    color = (color * 0.3) + (color_two * 0.5) + (color_one * 0.3);
}

```

Shader #104 - color bloom

File: color-bloom.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Computes luminance for brightness-based effects. Sample the texture at the current UV coordinates Determine the brightness of the pixel Apply a simple threshold for bright pixels to determine blooming effect

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Hermite smoothstep interpolation
 - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 uv = tc * iResolution / vec2(iResolution.y);
    float time = time_f * 0.5;
    float angle = time;
    vec2 center = vec2(0.5, 0.5) * iResolution / vec2(iResolution.y);
    vec2 toCenter = uv - center;
    float radius = length(toCenter);
    float theta = atan(toCenter.y, toCenter.x) + time;
    float pattern = abs(sin(12.0 * theta) * cos(12.0 * radius));
    vec4 texColor = texture(samp, tc);
    float brightness = dot(texColor.rgb, vec3(0.299, 0.587, 0.114));
    float bloomThreshold = 0.6;
    vec3 bloomColor = texColor.rgb * pattern;
    vec3 finalColor = mix(texColor.rgb, bloomColor, smoothstep(bloomThreshold - 0.1, bloomThreshold +
    ↪ 0.1, brightness)).rgb;
    color = vec4(finalColor, texColor.a);
}

```

Shader #246 - dys

File: dys.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (5 texture samples) - Color quantization via floor rounding
- Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for (int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i] ^ isource[i];
        if (int_color[i] > 255)
            int_color[i] = int_color[i] % 255;
        icolor[i] = float(int_color[i]) / 255;
    }
    icolor.a = 1.0;
    return icolor;
}

void main() {
    vec2 uv = tc;
    float timeOffset = 0.05 * sin(time_f);
    vec3 redLayer = texture(samp, uv + vec2(timeOffset, 0.0)).rgb;
    vec3 greenLayer = texture(samp, uv).rgb;
    vec3 blueLayer = texture(samp, uv - vec2(timeOffset, 0.0)).rgb;
    bool strobe = mod(floor(time_f * 2.0), 2.0) > 0.5;
    vec3 rgbStrobe = vec3(
        strobe ? redLayer.r : blueLayer.r,
        greenLayer.g,
        strobe ? blueLayer.b : redLayer.b
    );
    vec4 echoEffect = texture(samp, uv + vec2(0.01 * sin(time_f), 0.01 * cos(time_f)));
    vec4 modified = mix(vec4(rgbStrobe, 1.0), echoEffect, 0.5);
    vec4 original = texture(samp, uv);
    color = xor_RGB(original, modified);
}
```

Shader #361 - frac shader01 smooth

File: frac_shader01_smooth.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion
- 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    vec2 d = normalize(vec2(cos(t*0.27), sin(t*0.31)));
    float s = dot(p, d);
    float band = 0.5 + 0.5*sin(s*6.28318530718*0.35 + t*0.9);
    float h = fract(s*0.22 + t*0.07 + 0.15*sin(t*0.33));
    float S = 0.75 + 0.25*sin(t*0.21 + s*2.0);
    float V = 0.75 + 0.25*band;
    vec3 base = hsv2rgb(vec3(h, S, V));
    float edge = smoothstep(0.2, 0.8, band);

```

```

    return mix(base*0.6, base, edge);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}
float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}
float luma(vec3 c){ return dot(c, vec3(0.299,0.587,0.114)); }
vec3 bilateral9(vec2 uv, float radiusScale, float sigma_r){
    vec2 texel = 1.0 / iResolution;
    vec2 o = texel * radiusScale;
    vec3 c0 = texture(samp, uv).rgb;
    float L0 = luma(c0);
    vec2 offs[9] = vec2[](
        vec2(0,0),
        vec2( o.x, 0), vec2(-o.x, 0),
        vec2(0, o.y), vec2(0, -o.y),
        vec2( o.x, o.y), vec2(-o.x, o.y),
        vec2( o.x, -o.y), vec2(-o.x, -o.y)
    );
    float wsum = 0.0;
    vec3 acc = vec3(0.0);
    for(int i=0;i<9;i++){
        vec3 c = texture(samp, uv + offs[i]).rgb;
        float dl = luma(c) - L0;
        float wr = exp(-(dl*dl)/(2.0*sigma_r*sigma_r));
        float dsq = dot(offs[i]/texel, offs[i]/texel);
        float ws = exp(-dsq/(2.0*1.0*1.0));
        float w = wr*ws;
        acc += c*w;
        wsum += w;
    }
    return acc / max(wsum, 1e-6);
}
float localVar9(vec2 uv, float radiusScale){
    vec2 texel = 1.0 / iResolution;
    vec2 o = texel * radiusScale;
    float m = 0.0;
    float s = 0.0;
    int n = 0;
    for(int y=-1;y<=1;y++){
        for(int x=-1;x<=1;x++){
            vec3 c = texture(samp, uv + vec2(x,y)*o).rgb;
            float L = luma(c);
            m += L;
            s += L*L;
            n++;
        }
    }
    m /= float(n);
    s /= float(n);
    return max(s - m*m, 0.0);
}
void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);

```



```

vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
vec2 uv = tc;
vec4 originalTexture = texture(samp, tc);
float seg = 4.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
vec2 kUV = reflectUV(uv, seg, m, aspect);
kUV = diamondFold(kUV, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = p;
q = abs(q);
if(q.y > q.x) q = q.yx;
float base = 1.82 + 0.18*sin(time_f*0.2);
float period = log(base);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + time_f*0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap*1.045) / ar + m);
vec2 u2 = fract((pwrap*0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
float hue = fract(ang*0.25 + time_f*0.08 + k*0.5);
float sat = 0.75 - 0.25*cos(time_f*0.7 + rD*10.0);
float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
vec3 tint = hsv2rgb(vec3(hue, sat, val));
float ring = smoothstep(0.0, 0.7, sin(log(rD*1e-3)*9.5 + time_f*1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + rD*28.0 + k*12.0);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
vign = mix(0.85, 1.2, vign);
float blendFactor = 0.58;
float v0 = localVar9(u0, 1.0);
float v1 = localVar9(u1, 1.0);
float v2 = localVar9(u2, 1.0);
float pix = (v0+v1+v2)/3.0;
float sStrong = smoothstep(0.02, 0.20, pix);
float radA = mix(0.75, 2.25, sStrong);
float sigma_r = mix(0.10, 0.22, sStrong);
vec3 bc0 = bilateral9(u0 + off, radA, sigma_r);
vec3 bc1 = bilateral9(u1, radA, sigma_r);
vec3 bc2 = bilateral9(u2 - off, radA, sigma_r);
float rC = bc0.r;
float gC = bc1.g;
float bC = bc2.b;
vec3 kaleidoRGB = vec3(rC, gC, bC);
vec4 kaleidoColor = vec4(kaleidoRGB, 1.0) * vec4(tint, 1.0);
vec4 merged = mix(kaleidoColor, originalTexture, blendFactor);
merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
vec3 outCol = merged.rgb + bloom;
float wob = 0.9 + 0.1*sin(time_f + rD*14.0 + k*9.0);
outCol *= wob;
vec3 grad = movingGradient(tc, m, time_f, aspect);
float gradBase = 0.35 + 0.25*sin(time_f*0.5 + rD*7.0 + k*9.0);
float gradAmt = mix(gradBase*0.5, 0.95, sStrong);
vec3 screenBlend = 1.0 - (1.0 - outCol) * (1.0 - grad);
outCol = mix(outCol, screenBlend, gradAmt);

```

```

    vec4 t = texture(samp, tc);
    outCol = mix(outCol, outCol*t.rgb, 0.8);
    outCol = sin(outCol * (0.5 + 0.5*pingPong(time_f, 12.0)));
    outCol = clamp(outCol, vec3(0.08), vec3(0.96));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    outCol *= 1.0 + _ab * 0.6;
    outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
    ↪ 0.0, 1.0) * 0.25), _ab);
    color = vec4(outCol, 1.0);
}

```

Shader #462 - gblur

File: gblur.glsl

Processes the input texture to produce the "gblur" visual effect.

Technique: Driven by: resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform vec2 iResolution;
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

void main(void) {
    color = blur(samp, tc, iResolution);
}

```

Shader #532 - gptsmooth

File: gptsmooth.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Adjust the blur radius for a stronger effect Introduce a subtle wave effect for more dynamism Modify time variable for smoother transitions

Technique: Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color = ivec3(icolor.rgb * 255.0);
    ivec3 isource = ivec3(source.rgb * 255.0);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int_color[i] ^ isource[i];
        int_color[i] = int_color[i] % 256;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 enhancedBlur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float totalWeight = 0.0;
    int radius = 5;
    for (int x = -radius; x <= radius; ++x) {
        for (int y = -radius; y <= radius; ++y) {
            float distance = length(vec2(x, y));
            float weight = exp(-distance * distance / (2.0 * radius * radius));
            vec2 offset = vec2(x, y) * texelSize;
            result += texture(image, uv + offset) * weight;
            totalWeight += weight;
        }
    }
    return result / totalWeight;
}
void main(void) {
    vec2 uv = tc;
    uv.x += sin(uv.y * 20.0 + time_f * 2.0) * 0.005;
    uv.y += cos(uv.x * 20.0 + time_f * 2.0) * 0.005;
    vec4 tcolor = enhancedBlur(samp, uv, iResolution);
    float time_t = pingPong(time_f * 0.3, 5.0) + 1.0;
    vec4 modColor = vec4(
        tcolor.r * (0.5 + 0.5 * sin(time_f + uv.x * 10.0)),
        tcolor.g * (0.5 + 0.5 * sin(time_f + uv.y * 10.0)),
        tcolor.b * (0.5 + 0.5 * sin(time_f + (uv.x + uv.y) * 5.0)),
        tcolor.a
    );
    color = xor_RGB(sin(tcolor * time_f), modColor);
    color.rgb = pow(color.rgb, vec3(1.2));
}
```

Shader #533 - gptsmooth2

File: gptsmooth2.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec3 hue2rgb(float h) {
    h = mod(h, 1.0);
    float r = abs(h * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(h * 6.0 - 2.0);
    float b = 2.0 - abs(h * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}

vec4 enhancedBlur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float totalWeight = 0.0;
    int radius = 5;
    for (int x = -radius; x <= radius; ++x) {
        for (int y = -radius; y <= radius; ++y) {
            float distance = length(vec2(x, y));
            float sigma = float(radius) / 2.0;
            float weight = exp(-distance * distance / (2.0 * sigma * sigma));
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * weight;
            totalWeight += weight;
        }
    }
    return result / totalWeight;
}

vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color = ivec3(icolor.rgb * 255.0);
    ivec3 isource = ivec3(source.rgb * 255.0);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int_color[i] ^ isource[i];
        int_color[i] = int_color[i] % 256;
        icolor[i] = float(int_color[i]) / 255.0;
    }
    icolor.a = 1.0;
    return icolor;
}

void main(void) {
    vec2 uv = tc;
    uv.x += sin(uv.y * 20.0 + time_f * 2.0) * 0.005;
    uv.y += cos(uv.x * 20.0 + time_f * 2.0) * 0.005;
    vec4 tcolor = enhancedBlur(samp, uv, iResolution);
    vec4 modulatedColor = vec4(tcolor.rgb * time_f, tcolor.a);
    color = xor_RGB(tcolor, modulatedColor);
    color.rgb = pow(color.rgb, vec3(1.2));
}
```

```
}
```

Shader #806 - rainbow blur

File: rainbow_blur.glsl

Applies geometric transformations including rotation and scaling to distort the image.

Technique: 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending -
Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);
    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
}
```

```

    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

vec4 colorRotate(vec4 color) {
    ivec4 col = ivec4(color * 255);
    for (int i = 0; i < 3; ++i) {
        col[i] = col[i] % 0xFF;
    }
    return vec4(col / 255.0);
}

void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    vec3 rainbow_color = rainbow(uv.x + sin(time_f * 0.5 + uv.y * 5.0) * 0.1);
    vec4 blurred_color = blur(samp, tc, iResolution);
    vec3 blended_color = mix(blurred_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_t), blurred_color.a);
}

```

Shader #881 - rgb blur

File: rgb_blur.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;

```

```

uniform vec4 inc_valuex;
uniform vec4 inc_value;
void main(void)
{
    color = texture(samp, tc);
    vec2 pos1 = tc;
    vec2 pos2 = tc;
    pos1 += 0.01;
    pos2 += 0.02;
    vec4 color2 = texture(samp, pos1);
    vec4 color3 = texture(samp, pos2);
    color[1] = color2[1];
    color[2] = color3[2];
}

```

Shader #882 - rgb blur time

File: rgb_blur_time.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
void main(void)
{
    color = texture(samp, tc);
    vec2 pos1 = tc;
    vec2 pos2 = tc;
    pos1 += (fract(timeval)+0.01)*0.02;
    pos2 += (fract(timeval)+0.02)*0.03;
    vec4 color2 = texture(samp, pos1);
    vec4 color3 = texture(samp, pos2);
    color[1] = color2[1];
    color[2] = color3[2];
}

```

Shader #950 - smooth gl

File: smooth_gl.glsl

Blends multiple texture samples together using interpolation for a composite effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float luma(vec3 c){ return dot(c, vec3(0.299,0.587,0.114)); }
void main(void) {
    vec2 texel = 1.0 / iResolution;
    vec4 c0 = texture(samp, tc);
    vec2 o1 = texel * 1.0;
    vec2 o2 = texel * 1.5;
    vec2 k[9] = vec2[](
        vec2( 0.0,  0.0),
        vec2( 1.0,  0.0), vec2(-1.0,  0.0),
        vec2( 0.0,  1.0), vec2( 0.0, -1.0),
        vec2( 1.0,  1.0), vec2(-1.0,  1.0),
        vec2( 1.0, -1.0), vec2(-1.0, -1.0)
    );
    float sigma_s = 1.0;
    float sigma_r = 0.15;
    float wsum = 0.0;
    vec3 acc = vec3(0.0);
    for(int i=0;i<9;i++){
        vec2 dp = k[i] * o1;
        vec4 c = texture(samp, tc + dp);
        float dsq = dot(dp/(texel*1.0), dp/(texel*1.0));
        float wr = exp(-pow(max(0.0, luma(c.rgb)-luma(c0.rgb)), 2.0)/(2.0*sigma_r*sigma_r));
        float ws = exp(-dsq/(2.0*sigma_s*sigma_s));
        float w = wr * ws;
        acc += c.rgb * w;
        wsum += w;
    }
    vec3 bilateral = acc / max(wsum, 1e-6);
    vec4 cN = texture(samp, tc + vec2(0.0,  o1.y));
    vec4 cS = texture(samp, tc + vec2(0.0, -o1.y));
    vec4 cE = texture(samp, tc + vec2( o1.x, 0.0));
    vec4 cW = texture(samp, tc + vec2(-o1.x, 0.0));
    float edge = abs(luma(cE.rgb)-luma(cW.rgb)) + abs(luma(cN.rgb)-luma(cS.rgb));
    vec4 wide =
        ( texture(samp, tc + vec2( 0.0,  0.0)) * 4.0 +
          ( texture(samp, tc + vec2( o2.x, 0.0)) + texture(samp, tc + vec2(-o2.x, 0.0)) +
            texture(samp, tc + vec2(0.0,  o2.y)) + texture(samp, tc + vec2(0.0, -o2.y)) ) * 2.0 +
          ( texture(samp, tc + vec2( o2.x,  o2.y)) + texture(samp, tc + vec2(-o2.x,  o2.y)) +
            texture(samp, tc + vec2( o2.x, -o2.y)) + texture(samp, tc + vec2(-o2.x, -o2.y)) ) ) / 16.0;
    float localVar =
        ( abs(luma(c0.rgb) - luma(wide.rgb)) +
          abs(luma(cN.rgb) - luma(wide.rgb)) +
          abs(luma(cS.rgb) - luma(wide.rgb)) +
```



```

        abs(luma(cE.rgb) - luma(wide.rgb)) +
        abs(luma(cW.rgb) - luma(wide.rgb)) ) / 5.0;
float baseStrength = smoothstep(0.02, 0.12, localVar);
float edgeProtect  = 1.0 - smoothstep(0.05, 0.20, edge);
float userGain     = clamp(amp + uamp, 0.0, 1.0);
float strength     = clamp(baseStrength * edgeProtect * userGain, 0.0, 1.0);
float t = 0.5 + 0.5 * sin(time_f * 0.6);
vec3 melted = mix(bilateral, wide.rgb, t);
vec3 outRGB = mix(c0.rgb, melted, strength);
color = vec4(outRGB, 1.0);
}

```

Shader #952 - smooth var

File: smooth_var.glsl

Blends multiple texture samples together using interpolation for a composite effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
void main(void) {
    vec2 texel = 1.0 / iResolution;
    float spread = 1.5;
    vec2 o = texel * spread;
    vec4 c0 = texture(samp, tc);
    vec4 cN = texture(samp, tc + vec2(0.0, o.y));
    vec4 cS = texture(samp, tc + vec2(0.0, -o.y));
    vec4 cE = texture(samp, tc + vec2(o.x, 0.0));
    vec4 cW = texture(samp, tc + vec2(-o.x, 0.0));
    vec4 cNE = texture(samp, tc + vec2(o.x, o.y));
    vec4 cNW = texture(samp, tc + vec2(-o.x, o.y));
    vec4 cSE = texture(samp, tc + vec2(o.x, -o.y));
    vec4 cSW = texture(samp, tc + vec2(-o.x, -o.y));
    vec4 mean = (c0 + cN + cS + cE + cW + cNE + cNW + cSE + cSW) / 9.0;
    float variance =
        dot(abs(c0 - mean).rgb, vec3(0.333)) +
        dot(abs(cN - mean).rgb, vec3(0.333)) +
        dot(abs(cS - mean).rgb, vec3(0.333)) +
        dot(abs(cE - mean).rgb, vec3(0.333)) +
        dot(abs(cW - mean).rgb, vec3(0.333)) +
        dot(abs(cNE - mean).rgb, vec3(0.333)) +
        dot(abs(cNW - mean).rgb, vec3(0.333)) +
        dot(abs(cSE - mean).rgb, vec3(0.333)) +
        dot(abs(cSW - mean).rgb, vec3(0.333));
    variance /= 9.0;
    float strength = smoothstep(0.05, 0.15, variance) * clamp(amp + uamp, 0.0, 1.0);
    vec4 tent = (c0 * 4.0 +
        (cN + cS + cE + cW) * 2.0 +
        (cNE + cNW + cSE + cSW)) / 16.0;
}

```

```

    color = mix(c0, tent, strength);
}

```

Shader #990 - sum

File: sum.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging.

Technique: Multi-tap convolution blur (5 texture samples) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 0.0;
    for (int i = 0; i < 100; i++) {
        kernelSum += kernel[i];
    }
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
}

```

```

    }
}
return result / kernelSum;
}

void main() {
    vec2 texelSize = 1.0 / iResolution;
    vec4 sum = texture(samp, tc) * 4.0;
    sum += texture(samp, tc + vec2(texelSize.x, 0.0));
    sum += texture(samp, tc - vec2(texelSize.x, 0.0));
    sum += texture(samp, tc + vec2(0.0, texelSize.y));
    sum += texture(samp, tc - vec2(0.0, texelSize.y));
    sum /= 8.0;
    float time_t = pingPong(time_f, 10.0) + 2.0;
    float time_x = pingPong(time_f, 5.0) + 2.0;
    vec4 b_tex = blur(samp, tc, iResolution);
    if (sum.r + sum.g + sum.b > 1.5) {
        color = mix(sin(b_tex * time_t), sin(sum * time_x), 0.5);
    } else {
        color = xor_RGB(sin(b_tex * time_t), sin(sum * time_x));
    }
}
}

```

Shader #1079 - xor rgb smooth

File: xor_rgb_smooth.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;
    vec4 result = vec4(0.0);
    float kernel[49];
    kernel[0] = 0.5; kernel[1] = 1.0; kernel[2] = 2.0; kernel[3] = 2.5; kernel[4] = 2.0; kernel[5] =
    ↪ 1.0; kernel[6] = 0.5;

```

```

kernel[7] = 1.0; kernel[8] = 2.0; kernel[9] = 3.0; kernel[10] = 3.5; kernel[11] = 3.0; kernel[12] =
↳ 2.0; kernel[13] = 1.0;
kernel[14] = 2.0; kernel[15] = 3.0; kernel[16] = 4.0; kernel[17] = 4.5; kernel[18] = 4.0; kernel[19]
↳ = 3.0; kernel[20] = 2.0;
kernel[21] = 2.5; kernel[22] = 3.5; kernel[23] = 4.5; kernel[24] = 5.0; kernel[25] = 4.5; kernel[26]
↳ = 3.5; kernel[27] = 2.5;
kernel[28] = 2.0; kernel[29] = 3.0; kernel[30] = 4.0; kernel[31] = 4.5; kernel[32] = 4.0; kernel[33]
↳ = 3.0; kernel[34] = 2.0;
kernel[35] = 1.0; kernel[36] = 2.0; kernel[37] = 3.0; kernel[38] = 3.5; kernel[39] = 3.0; kernel[40]
↳ = 2.0; kernel[41] = 1.0;
kernel[42] = 0.5; kernel[43] = 1.0; kernel[44] = 2.0; kernel[45] = 2.5; kernel[46] = 2.0; kernel[47]
↳ = 1.0; kernel[48] = 0.5;
float kernelSum = 272.0;
for (int x = -3; x <= 3; ++x) {
    for (int y = -3; y <= 3; ++y) {
        vec2 offset = vec2(float(x), float(y)) * texelSize;
        result += texture(image, uv + offset) * kernel[(y + 3) * 7 + (x + 3)];
    }
}
return result / kernelSum;
}

void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = xor_RGB(tcolor, tcolor * time_t);
}

```

Shader #1080 - xor rgb smoother

File: xor_rgb_smoother.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}

float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}

vec4 blur(sampler2D image, vec2 uv, vec2 resolution) {
    vec2 texelSize = 1.0 / resolution;

```

```

    vec4 result = vec4(0.0);
    float kernel[100];
    float kernelVals[100] = float[](0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.5, 3.5, 4.0, 4.5, 5.0, 5.0, 4.5, 4.0, 3.5, 2.5,
                                     2.0, 3.0, 3.5, 4.0, 4.5, 4.5, 4.0, 3.5, 3.0, 2.0,
                                     1.5, 2.5, 3.0, 3.5, 4.0, 4.0, 3.5, 3.0, 2.5, 1.5,
                                     1.0, 2.0, 2.5, 3.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.0,
                                     0.5, 1.0, 1.5, 2.0, 2.5, 2.5, 2.0, 1.5, 1.0, 0.5);

    for (int i = 0; i < 100; i++) {
        kernel[i] = kernelVals[i];
    }
    float kernelSum = 842.0;
    for (int x = -5; x <= 4; ++x) {
        for (int y = -5; y <= 4; ++y) {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(image, uv + offset) * kernel[(y + 5) * 10 + (x + 5)];
        }
    }
    return result / kernelSum;
}

void main(void) {
    vec4 tcolor = blur(samp, tc, iResolution);
    float time_t = pingPong(time_f, 10.0) + 2.0;
    color = xor_RGB(tcolor, tcolor * time_t);
}

```

Light & Electric (18 shaders)

Shader #226 - distort light

File: distort_light.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Linear mix/lerp blending - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)

```

```

{
    color = texture(samp, tc);
    vec4 color_backup = color;
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec2 cord1 = vec2(tc[0]/2, tc[1]/2);
    vec2 cord2 = vec2(tc[0]/4, tc[1]/4);
    vec2 cord3 = vec2(tc[0]/8, tc[1]/8);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    color[0] = (color[0] * col1[0]) * timeval;
    color[1] = (color[1] * col2[1]) * timeval;
    color[2] = (color[2] * col3[2]) * timeval;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
    color[0] += 0.2;
    color[1] += 0.2;
    color[2] += 0.2;
    color = mix(color, color_backup, 0.5);
}

```

Shader #362 - frac shader01 smooth neon

File: frac_shader01_smooth_neon.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;

```

```

uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}
vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    vec2 d = normalize(vec2(cos(t*0.27), sin(t*0.31)));
    float s = dot(p, d);
    float band = 0.5 + 0.5*sin(s*6.28318530718*0.35 + t*0.9);
    float h = fract(s*0.22 + t*0.07 + 0.15*sin(t*0.33));
    float S = 0.75 + 0.25*sin(t*0.21 + s*2.0);
    float V = 0.75 + 0.25*band;
    vec3 base = hsv2rgb(vec3(h, S, V));
    float edge = smoothstep(0.2, 0.8, band);
    return mix(base*0.6, base, edge);
}
vec2 diamondFold(vec2 uv, vec2 c, float aspect){
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if(p.y > p.x) p = p.yx;
    p.x /= aspect;

```

```

    return p + c;
}
float diamondRadius(vec2 p){
    p = abs(p);
    return max(p.x, p.y);
}
float luma(vec3 c){ return dot(c, vec3(0.299,0.587,0.114)); }
vec3 bilateral9(vec2 uv, float radiusScale, float sigma_r){
    vec2 texel = 1.0 / iResolution;
    vec2 o = texel * radiusScale;
    vec3 c0 = texture(samp, uv).rgb;
    float L0 = luma(c0);
    vec2 offs[9] = vec2[](
        vec2(0,0),
        vec2( o.x, 0), vec2(-o.x, 0),
        vec2(0,  o.y), vec2(0, -o.y),
        vec2( o.x,  o.y), vec2(-o.x,  o.y),
        vec2( o.x, -o.y), vec2(-o.x, -o.y)
    );
    float wsum = 0.0;
    vec3 acc = vec3(0.0);
    for(int i=0;i<9;i++){
        vec3 c = texture(samp, uv + offs[i]).rgb;
        float dl = luma(c) - L0;
        float wr = exp(-(dl*dl)/(2.0*sigma_r*sigma_r));
        float dsq = dot(offs[i]/texel, offs[i]/texel);
        float ws = exp(-dsq/(2.0*1.0*1.0));
        float w = wr*ws;
        acc += c*w;
        wsum += w;
    }
    return acc / max(wsum, 1e-6);
}
float localVar9(vec2 uv, float radiusScale){
    vec2 texel = 1.0 / iResolution;
    vec2 o = texel * radiusScale;
    float m = 0.0;
    float s = 0.0;
    int n = 0;
    for(int y=-1;y<=1;y++){
        for(int x=-1;x<=1;x++){
            vec3 c = texture(samp, uv + vec2(x,y)*o).rgb;
            float L = luma(c);
            m += L;
            s += L*L;
            n++;
        }
    }
    m /= float(n);
    s /= float(n);
    return max(s - m*m, 0.0);
}
void main(){
    float aspect = iResolution.x / iResolution.y;
    vec2 ar = vec2(aspect, 1.0);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 uv = tc;
    vec4 originalTexture = texture(samp, tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
    vec2 kUV = reflectUV(uv, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);

```



```

kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
kUV = diamondFold(kUV, m, aspect);
vec2 p = (kUV - m) * ar;
vec2 q = p;
q = abs(q);
if(q.y > q.x) q = q.yx;
float base = 1.82 + 0.18*sin(time_f*0.2);
float period = log(base);
float tz = time_f * (0.65 + amp_low * 0.3);
float rD = diamondRadius(p) + 1e-6;
float ang = atan(q.y, q.x) + tz * 0.35 + 0.35*sin(rD*18.0 + time_f*0.6);
float k = fract((log(rD) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap*1.045) / ar + m);
vec2 u2 = fract((pwrap*0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
float hue = fract(ang*0.25 + time_f*0.08 + k*0.5);
float sat = 0.75 - 0.25*cos(time_f*0.7 + rD*10.0);
float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
vec3 tint = hsv2rgb(vec3(hue, sat, val));
float ring = smoothstep(0.0, 0.7, sin(log(rD+1e-3)*9.5 + time_f*1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + rD*28.0 + k*12.0);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
vign = mix(0.85, 1.2, vign);
float blendFactor = 0.58;
float v0 = localVar9(u0, 1.0);
float v1 = localVar9(u1, 1.0);
float v2 = localVar9(u2, 1.0);
float pix = (v0+v1+v2)/3.0;
float sStrong = smoothstep(0.02, 0.20, pix);
float radA = mix(0.75, 2.25, sStrong);
float sigma_r = mix(0.10, 0.22, sStrong);
vec3 bc0 = bilateral9(u0 + off, radA, sigma_r);
vec3 bc1 = bilateral9(u1, radA, sigma_r);
vec3 bc2 = bilateral9(u2 - off, radA, sigma_r);
float rC = bc0.r;
float gC = bc1.g;
float bC = bc2.b;
vec3 kaleidoRGB = vec3(rC, gC, bC);
vec4 kaleidoColor = vec4(kaleidoRGB, 1.0) * vec4(tint, 1.0);
vec4 merged = mix(kaleidoColor, originalTexture, blendFactor);
merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
vec3 outCol = merged.rgb + bloom;
float wob = 0.9 + 0.1*sin(time_f + rD*14.0 + k*9.0);
outCol *= wob;
vec3 grad = movingGradient(tc, m, time_f, aspect);
float gradBase = 0.35 + 0.25*sin(time_f*0.5 + rD*7.0 + k*9.0);
float gradAmt = mix(gradBase*0.5, 0.95, sStrong);
vec3 screenBlend = 1.0 - (1.0 - outCol) * (1.0 - grad);
outCol = mix(outCol, screenBlend, gradAmt);
vec4 t = texture(samp, tc);
outCol = mix(outCol, outCol*t.rgb, 0.8);
float lum = dot(outCol, vec3(0.299, 0.587, 0.114));
float tHue = time_f * 0.15 + uamp * 0.1;
float hueBase = fract(lum * 0.8 + tHue);
vec3 neon1 = hsv2rgb(vec3(hueBase, 1.0, 1.0));
vec3 neon2 = hsv2rgb(vec3(fract(hueBase + 0.33), 1.0, 1.0));

```

```

float wave = pingPong(time_f * 0.25 + amp * 2.0, 1.0);
vec3 neon = mix(neon1, neon2, wave);
float audio = clamp(amp * 1.5 + uamp * 0.25, 0.0, 1.5);
float strength = clamp(0.2 + audio, 0.0, 1.0);
vec3 mixed = mix(outCol, neon, strength);
mixed = pow(mixed, vec3(0.8));
mixed = clamp(mixed, 0.0, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
mixed *= 1.0 + _ab * 0.6;
mixed = mix(mixed, mixed * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high, 0.0,
↳ 1.0) * 0.25), _ab);
color = vec4(mixed, 1.0);
}

```

Shader #377 - frac shader02 dmd6i neon

File: frac_shader02_dmd6i_neon.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control. Computes luminance for brightness-based effects.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Indexed color palette look-up table - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
const float PI = 3.1415926535897932384626433832795;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect) {
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

```

```

}
vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect) {
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect) {
    vec2 p = uv;
    for (int i = 0; i < 6; i++) {
        p = abs((p - c) * (zoom + 0.15 * sin(t * (0.35 + amp_low * 0.2) + float(i)))) - 0.5 + c;
        p = rotateUV(p, t * 0.12 + float(i) * 0.07, c, aspect);
    }
    return p;
}

vec3 neonPalette(float t) {
    vec3 pink = vec3(1.0, 0.15, 0.75);
    vec3 blue = vec3(0.10, 0.55, 1.0);
    vec3 green = vec3(0.10, 1.00, 0.45);
    float ph = fract(t * 0.08);
    vec3 k1 = mix(pink, blue, smoothstep(0.00, 0.33, ph));
    vec3 k2 = mix(blue, green, smoothstep(0.33, 0.66, ph));
    vec3 k3 = mix(green, pink, smoothstep(0.66, 1.00, ph));
    float a = step(ph, 0.33);
    float b = step(0.33, ph) * step(ph, 0.66);
    float c = step(0.66, ph);
    return normalize(a * k1 + b * k2 + c * k3) * 1.05;
}

vec3 softTone(vec3 c) {
    c = pow(max(c, 0.0), vec3(0.95));
    float l = dot(c, vec3(0.299, 0.587, 0.114));
    c = mix(vec3(l), c, 0.9);
    return clamp(c, 0.0, 1.0);
}

vec3 tentBlur3(sampler2D img, vec2 uv, vec2 res) {
    vec2 ts = 1.0 / res;
    vec3 s00 = textureGrad(img, uv + ts * vec2(-1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s10 = textureGrad(img, uv + ts * vec2(0.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s20 = textureGrad(img, uv + ts * vec2(1.0, -1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s01 = textureGrad(img, uv + ts * vec2(-1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s11 = textureGrad(img, uv, dFdx(uv), dFdy(uv)).rgb;
    vec3 s21 = textureGrad(img, uv + ts * vec2(1.0, 0.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s02 = textureGrad(img, uv + ts * vec2(-1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s12 = textureGrad(img, uv + ts * vec2(0.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    vec3 s22 = textureGrad(img, uv + ts * vec2(1.0, 1.0), dFdx(uv), dFdy(uv)).rgb;
    return (s00 + 2.0 * s10 + s20 + 2.0 * s01 + 4.0 * s11 + 2.0 * s21 + s02 + 2.0 * s12 + s22) / 16.0;
}

vec3 preBlendColor(vec2 uv) {
    vec3 tex = tentBlur3(samp, uv, iResolution);
    float aspect = iResolution.x / iResolution.y;
    vec2 p = (uv - 0.5) * vec2(aspect, 1.0);
    float r = length(p);
    float t = time_f;
    vec3 neon = neonPalette(t + r * 1.3);
    float neonAmt = smoothstep(0.1, 0.8, r);
    neonAmt = (0.3 + amp_mid * 0.15) + 0.4 * (1.0 - neonAmt);
}

```

```

    vec3 grad = mix(tex, neon, neonAmt);
    grad = mix(grad, tex, 0.2);
    grad = softTone(grad);
    return grad;
}

float diamondRadius(vec2 p) {
    p = sin(abs(p));
    return max(p.x, p.y);
}

vec2 diamondFold(vec2 uv, vec2 c, float aspect) {
    vec2 p = (uv - c) * vec2(aspect, 1.0);
    p = abs(p);
    if (p.y > p.x) p = p.yx;
    p.x /= aspect;
    return p + c;
}

void main(void) {
    vec4 baseTex = texture(samp, tc);
    vec3 origRGB = baseTex.rgb;
    vec2 uv = tc * 2.0 - 1.0;
    float aspect = iResolution.x / iResolution.y;
    uv.x *= aspect;
    float r = pingPong(sin(length(uv) * time_f), 5.0);
    float radius = sqrt(aspect * aspect + 1.0) + 0.5;
    float glow = smoothstep(radius, radius - (0.25 + amp_low * 0.3), r);
    vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
    vec2 ar = vec2(aspect, 1.0);
    vec3 baseCol = preBlendColor(tc);
    float seg = 4.0 + (2.0 + amp_mid * 2.0) * sin(time_f * 0.33);
    vec2 kUV = reflectUV(tc, seg, m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
    kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
    kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
    kUV = diamondFold(kUV, m, aspect);
    vec2 p = (kUV - m) * ar;
    vec2 q = abs(p);
    if (q.y > q.x) q = q.yx;
    float base = 1.82 + 0.18 * pingPong(sin(time_f * 0.2) * (PI * time_f), 5.0);
    float period = log(base) * pingPong(time_f * PI, 5.0);
    float tz = time_f * (0.65 + amp_low * 0.3);
    float rD = diamondRadius(p) + 1e-6;
    float ang = atan(q.y, q.x) + tz * 0.35 + 0.35 * sin(rD * 18.0 + time_f * 0.6);
    float k = fract((log(rD) - tz) / period);
    float rw = exp(k * period);
    vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
    vec2 u0 = fract(pwrap / ar + m);
    vec2 u1 = fract((pwrap * 1.045) / ar + m);
    vec2 u2 = fract((pwrap * 0.955) / ar + m);
    vec2 dir = normalize(pwrap + 1e-6);
    vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f * 1.3)) * vec2(1.0, 1.0 / aspect);
    ↵ ;
    float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m) * ar));
    vign = mix(0.9, (1.15 + amp_smooth * 0.2), vign);
    vec3 rC = preBlendColor(u0 + off);
    vec3 gC = preBlendColor(u1);
    vec3 bC = preBlendColor(u2 - off);
    vec3 kaleidoRGB = vec3(rC.r, gC.g, bC.b);
    float ring = smoothstep(0.0, 0.7, sin(log(rD + 1e-3) * 9.5 + time_f * 1.2));
    ring = ring * pingPong((time_f * PI), 5.0) * (1.0 + amp_mid * 0.3);
    float pulse = 0.5 + (0.5 + amp_rms * 0.3) * sin(time_f * 2.0 + rD * 28.0 + k * 12.0);
    vec3 outCol = kaleidoRGB;

```

```

outCol *= (0.75 + 0.25 * ring) * (0.85 + 0.15 * pulse) * vign;
vec3 bloom = outCol * outCol * 0.18 + pow(max(outCol - 0.6, 0.0), vec3(2.0)) * 0.12;
outCol += bloom;
outCol = mix(outCol, baseCol, pingPong(pulse * PI, 5.0) * 0.18);
outCol = clamp(outCol, vec3(0.05), vec3(0.97));
vec3 finalRGB = mix(origRGB, outCol, pingPong(glow * PI, 5.0) * 0.8);
float lum = dot(finalRGB, vec3(0.299, 0.587, 0.114));
float audio = clamp(amp * 1.5 + uamp * 0.35, 0.0, 2.0);
float tHue = time_f * 0.15 + uamp * 0.12;
float hueBase = fract(lum * 0.8 + tHue);
vec3 neon1 = hsv2rgb(vec3(hueBase, 1.0, 1.0));
vec3 neon2 = hsv2rgb(vec3(fract(hueBase + 0.33), 1.0, 1.0));
float wave = pingPong(time_f * 0.25 + amp * 2.0, 1.0);
vec3 neon = mix(neon1, neon2, wave);
float strength = clamp(0.2 + audio * 0.6, 0.0, 1.0);
vec3 texNeon = mix(finalRGB, neon, strength);
vec3 intertwined = mix(texNeon * origRGB, texNeon, 0.4 + 0.5 * clamp(audio, 0.0, 1.0));
vec3 mixed = pow(intertwined, vec3(0.8));
mixed = clamp(mixed, 0.0, 1.0);
float _ab = clamp(amp_peak, 0.0, 1.0);
float _abass = clamp(amp_low, 0.0, 1.0);
mixed *= 1.0 + _ab * 0.6;
mixed = mix(mixed, mixed * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high, 0.0,
↳ 1.0) * 0.25), _ab);
color = vec4(mixed, baseTex.a);
}

```

Shader #472 - gem light frac

File: gem-light-frac.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions. Store original UVs for global glow This is where the magic happens. We iterate to create layers. This 'fract' creates the repetition. The '-0.5' re-centers each "tile".

Technique: Indexed color palette look-up table - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.263, 0.416, 0.557);
    return a + b * cos(6.28318 * (c * t + d));
}
void main(void) {
    vec2 uv = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    vec2 uv0 = uv;
    vec3 finalColor = vec3(0.0);
    for (float i = 0.0; i < 4.0; i++) {
        uv = fract(uv * 1.5) - 0.5;
        float d = length(uv) * exp(-length(uv0));
        vec3 col = palette(length(uv0) + i * 0.4 + time_f * 0.4);
        d = sin(d * 8.0 + time_f) / 8.0;
    }
}

```

```

        d = abs(d);
        d = pow(0.01 / d, 1.2);
        finalColor += col * d;
    }
    vec2 distortion = uv * 0.1;
    vec4 texColor = texture(samp, tc + distortion);
    vec3 composite = mix(texColor.rgb, finalColor, 0.6);
    color = vec4(composite, texColor.a);
}

```

Shader #602 - light rings

File: light-rings.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float random(vec2 st) {
    return fract(sin(dot(st.xy, vec2(12.9898,78.233))) * 43758.5453123);
}
float noise(vec2 st) {
    vec2 i = floor(st);
    vec2 f = fract(st);
    float a = random(i);
    float b = random(i + vec2(1.0, 0.0));
    float c = random(i + vec2(0.0, 1.0));
    float d = random(i + vec2(1.0, 1.0));
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(a, b, u.x) + (c - a) * u.y * (1.0 - u.x) + (d - b) * u.x * u.y;
}
float fbm(vec2 st) {
    float value = 0.0;
    float amplitude = 0.5;
    float frequency = 0.0;
    for (int i = 0; i < 5; i++) {
        value += amplitude * noise(st);
        st *= 2.0;
        amplitude *= 0.5;
    }
    return value;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc;
    vec2 center = iResolution * 0.5;
    vec2 pos = uv * iResolution - center;
    float dist = length(pos);
    float angle = atan(pos.y, pos.x);
}

```

```

float lightning = smoothstep(0.1, 0.12, fbm(vec2(angle * 8.0, time_f * 0.5) + vec2(dist * 0.1)));
float brightness = lightning * (1.0 - dist / (iResolution.x * 0.5));
float time_z = pingPong(time_f, 5.0) + 2.0;
brightness = sin(brightness * time_z);
vec4 texColor = texture(samp, uv);
float time_t = pingPong(time_f, 10.0) + 2.0;
color = texColor + sin(vec4(vec3(brightness), 1.0) * time_t);
color.a = 1.0;
}

```

Shader #603 - light pulse

File: light_pulse.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main(void) {
    float lightCycleSpeed = 2.0;
    float lightIntensity = 0.5 + (0.5 + amp_rms * 0.5) * sin(time_f * lightCycleSpeed);
    vec3 rainbow = vec3(
        sin(time_f + 0.0) * 0.5 + 0.5,
        sin(time_f + 2.094) * 0.5 + 0.5,
        sin(time_f + 4.188) * 0.5 + 0.5
    );
    vec4 rainbowLight = vec4(rainbow, 1.0) * lightIntensity;
    vec4 originalColor = texture(samp, tc);
    color = mix(originalColor, rainbowLight, 0.5 * lightIntensity);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #604 - Light Rainbow Swirl

File: Light_Rainbow_Swirl.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Responds to mouse position for interactive control. Computes luminance for brightness-based effects. Center the coordinate system for the Spiral calculation 'r' is the fluid vector field. We use it to distort everything. This makes the spiral look like it is made of liquid.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - Per-channel UV offset for chromatic aberration - 2D rotation matrix - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform vec4 iMouse;
const float iAmplitude = 1.0;
const float iFrequency = 1.0;
const float iBrightness = 1.0;
const float iContrast = 1.1;
const float iSaturation = 1.3;
const float iZoom = 1.0;
const float iRotation = 0.0;
const float PI = 3.14159;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
vec3 adjustBrightness(vec3 col, float b) { return col * b; }
vec3 adjustContrast(vec3 col, float c) { return (col - 0.5) * c + 0.5; }
vec3 adjustSaturation(vec3 col, float s) {
    float gray = dot(col, vec3(0.299, 0.587, 0.114));
    return mix(vec3(gray), col, s);
}
vec2 wrapUV(vec2 tc) {
    return 1.0 - abs(1.0 - 2.0 * fract(tc * 0.5));
}
vec4 mxTexture(sampler2D tex, vec2 tc) {
    vec2 ts = vec2(textureSize(tex, 0));
    vec2 eps = 0.5 / ts;
    vec2 uv = wrapUV(tc);
    vec2 sampleUV = clamp(uv, eps, 1.0 - eps);
    return texture(tex, sampleUV);
}
vec2 rotate2D(vec2 p, float a) {
    float c = cos(a); float s = sin(a);
    return mat2(c, -s, s, c) * p;
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash(i), hash(i + vec2(1.0, 0.0)), u.x),
        mix(hash(i + vec2(0.0, 1.0)), hash(i + vec2(1.0, 1.0)), u.x), u.y);
}
float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    mat2 rot = mat2(cos(0.5), sin(0.5), -sin(0.5), cos(0.5));
    for (int i = 0; i < 5; i++) {
        v += a * noise(p);
    }
}
```



```

        p = rot * p * 2.0 + vec2(2.0);
        a *= 0.5;
    }
    return v;
}

vec3 sampleMergedLiquid(vec2 uv, float t, float strength, vec2 center, vec2 res) {
    float aspect = res.x / res.y;
    vec2 p = (uv - center) * vec2(aspect, 1.0);
    p = rotate2D(p, iRotation);
    float zoom = max(iZoom, 0.1);
    p /= zoom;
    vec2 q = vec2(0.0);
    q.x = fbm(p + vec2(0.0, 0.0) + 0.05*t);
    q.y = fbm(p + vec2(5.2, 1.3) + 0.05*t);
    vec2 r = vec2(0.0);
    r.x = fbm(p + 4.0*q + vec2(t * 0.2));
    r.y = fbm(p + 4.0*q + vec2(t * 0.1, 2.8));
    vec2 distortion = r * 0.3 * strength * iAmplitude;
    vec2 spiralCoords = p + distortion;
    float angle = atan(spiralCoords.y, spiralCoords.x);
    float spiralTwist = angle + length(spiralCoords) * 3.0 - t * 0.5;
    vec3 spiralColor = rainbow(spiralTwist / (2.0 * PI));
    vec2 fluidUV = uv + distortion;
    vec3 texCol = mxTexture(samp, fluidUV).rgb;
    float blendFactor = 0.6 + 0.2 * sin(length(r) * 5.0 + t);
    vec3 finalCol = mix(texCol, spiralColor, 0.5);
    finalCol += spiralColor * 0.2 * strength;
    return finalCol;
}

void main() {
    vec2 uv = tc;
    uv = wrapUV(uv);
    float t = time_f * (0.2 + iFrequency * 0.1);
    float ampControl = clamp(iAmplitude, 0.0, 2.0);
    float strength = 1.0 + (ampControl * 0.5);
    vec2 center = vec2(0.5);
    if (iMouse.z > 0.0) {
        center = iMouse.xy / iResolution;
    }
    vec2 offset = vec2(0.005 * strength, 0.0);
    vec3 col;
    col.r = sampleMergedLiquid(uv + offset, t, strength, center, iResolution).r;
    col.g = sampleMergedLiquid(uv, t, strength, center, iResolution).g;
    col.b = sampleMergedLiquid(uv - offset, t, strength, center, iResolution).b;
    col = adjustBrightness(col, iBrightness);
    col = adjustContrast(col, iContrast);
    col = adjustSaturation(col, iSaturation);
    vec2 vUV = uv * (1.0 - uv.yx);
    float vig = vUV.x * vUV.y * 15.0;
    vig = pow(vig, 0.2);
    col *= vig;
    color.rgb = sin(col * time_f);
    color.a = 1.0;
}

```

Shader #605 - light spiral

File: light_spiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling

effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main(void) {
    vec2 centeredCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float angle = atan(centeredCoord.y, centeredCoord.x) + time_f;
    float radius = length(centeredCoord);
    float spiral = sin((10.0 + amp_mid * 15.0) * angle - 3.0 * time_f) * exp(-3.0 * radius);
    vec3 lightColor = vec3(0.1, 0.5, 0.8) * 0.5 * (1.0 + spiral);
    lightColor = sin(lightColor * time_f);
    vec4 texColor = texture(samp, tc);
    color = vec4(texColor.rgb * lightColor, texColor.a);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}
```

Shader #606 - lightfade

File: lightfade.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations. Assign a random value based on the integer coord

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
```

```

float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}

void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec3 st = gl_FragCoord.xyz;
    for(int i = 0; i < 3; ++i) {
        float f = 1024/st[i];
        color[i] = color[i]*f;
    }
}

```

Shader #607 - lightfade xor

File: lightfade_xor.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Applies matrix transformations.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Bitwise XOR on integer color channels - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
in vec2 iResolution;
float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233))) *
                43758.5453123);
}

void main(void)
{
    color = texture(samp, tc);
    vec2 pos = gl_FragCoord.xy / iResolution.xy * timeval;
    float r = random(pos);
    ivec4 source = ivec4(color * 255);
    vec4 c = vec4(0.2, 0.5, 0.8, 1);
}

```

```

    color = color * c * alpha/2;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #608 - lightfadeddark

File: lightfadeddark.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec2 pos = iResolution_.xy;
    vec3 st = gl_FragCoord.zyx;
    for(int i = 0; i < 3; ++i) {
        float f = (1024/st[i]);
        color[i] = color[i]*f;
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

```
}
```

Shader #721 - neon

File: neon.glsl

Applies a time-animated visual transformation to the input texture.

Technique: Polar coordinate transformation ($\text{atan2} + \text{length}$) - Hermite smoothstep interpolation
- Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
vec3 overlayBlend(vec3 base, vec3 blend, float opacity) {
    vec3 c2 = blend * 2.0;
    vec3 c1 = 1.0 - 2.0 * (1.0 - blend);
    vec3 result = mix(base * c2, c1, step(0.5, base));
    return mix(base, result, opacity);
}
void main(void) {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 normCoord = 2.0 * (uv - center);
    float dist = length(normCoord);
    float angle = atan(normCoord.y, normCoord.x) + time_f * (5.0 + amp_low * 10.0);
    float spiral = cos((10.0 + amp_mid * 10.0) * dist - angle);
    float mask = smoothstep(0.1, 0.2, abs(spiral) - dist * 0.5);
    vec3 neonPurple = vec3(value_alpha_r, value_alpha_g, value_alpha_b);
    vec3 originalTexture = texture(samp, uv).rgb;
    vec3 blendedColor = overlayBlend(originalTexture, neonPurple, mask);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    blendedColor *= 1.0 + _ab * 0.6;
    blendedColor = mix(blendedColor, blendedColor * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 +
    ↪ clamp(amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(blendedColor, 1.0);
}
```

Shader #775 - pos light

File: pos_light.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
vec4 xor_RGB(vec4 icolor, ivec4 isource) {
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
void main(void)
{
    color = texture(samp, tc);
    vec2 pos = gl_FragCoord.xy/iResolution.xy;
    color = color * vec4(pos*alpha, 1, 1);
}

```

Shader #819 - rainbow expand noise xor light

File: rainbow_expand_noise_xor_light.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec4 xor_RGB(vec4 icolor, vec3 source) {
    ivec3 int_color;
    ivec3 isource = ivec3(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)

```

```

        int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
return icolor;
}
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
        dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0) * 0.5 + 0.5;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float rotation_period = 3.0;
    float rotation_angle = mod(time_f, rotation_period * 2.0) < rotation_period ? time_t : -time_t;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + rotation_angle * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec4 blended_color = xor_RGB(original_color, rainbow_color);
    color = blended_color;
    color.a = 1.0;
}

```

Shader #825 - rainbow light swirl

File: rainbow_light_swirl.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 centeredCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
    float angle = atan(centeredCoord.y, centeredCoord.x) + time_f;
    float radius = length(centeredCoord);
    float spiral = sin(10.0 * angle - 3.0 * time_f) * exp(-3.0 * radius);
    vec3 lightColor = vec3(0.1, 0.5, 0.8) * 0.5 * (1.0 + spiral);
    lightColor = sin(lightColor * time_f);
    vec4 texColor = texture(samp, tc);
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float t = mod(time_f, 15.0);
    float wave = sin(uv.x * 10.0 + t * 2.0) * 0.1;
    float expand = 0.5 + 0.5 * sin(t * 2.0);
    vec2 spiral_uv = uv * expand + vec2(cos(t), sin(t)) * 0.2;
    float new_angle = atan(spiral_uv.y + wave, spiral_uv.x) + t * 2.0;
    vec3 rainbow_color = rainbow(new_angle / (2.0 * 3.14159));
    vec3 blended_color = mix(texColor.rgb * lightColor, rainbow_color, 0.5);
    color = vec4(sin(blended_color * t), texColor.a);
}

```

Shader #988 - strobe light

File: strobe_light.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)

```



```
{
    color = texture(samp, tc);
    vec4 color_rand = vec4(alpha_r, alpha_g, alpha_b, 1.0);
    color = (0.6 * color) + (0.6 * color_rand);
}
```

Shader #1061 - whitelight

File: whitelight.glsl

Applies a time-animated visual transformation to the input texture. Applies matrix transformations.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    color = color * alpha;
}
```

Shader #1062 - wlight

File: wlight.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    float speed = 1.5;
    float amplitude = 0.02;
    float frequency = 6.0;
    float twist = 8.0;
```

```

vec2 vortexCenter = vec2(-0.5, 1.5);
vec2 adjustedTC = tc - vortexCenter;
float angle = atan(adjustedTC.y, adjustedTC.x);
float radius = length(adjustedTC);
float vortex = sin(twist * angle + radius * frequency - time_f * speed) * amplitude * radius;
vec2 vortexDistortedTC = tc + normalize(adjustedTC) * vortex;
float whirlSpeed = 2.0;
float whirlFrequency = 8.0;
float whirlAmplitude = 0.03;
float whirlScale = 0.5 + 0.5 * sin(time_f * whirlSpeed);
vec2 whirlCenter = vec2(0.5, 0.5);
vec2 whirlAdjustedTC = (tc - whirlCenter) * whirlScale;
float whirlAngle = atan(whirlAdjustedTC.y, whirlAdjustedTC.x);
float whirlRadius = length(whirlAdjustedTC);
float whirlDistortion = sin(whirlFrequency * whirlAngle + whirlRadius * whirlFrequency - time_f *
↳ whirlSpeed) * whirlAmplitude;
vec2 whirlDistortedTC = tc + normalize(whirlAdjustedTC) * whirlDistortion;
vec4 originalColor = texture(samp, tc);
vec4 vortexColor = texture(samp, vortexDistortedTC);
vec4 whirlColor = texture(samp, whirlDistortedTC);
float lightEffect = sin(whirlRadius * 10.0 - time_f * 5.0) * 0.5 + 0.5;
vec3 rainbow = vec3(
    sin(time_f + 0.0) * 0.5 + 0.5,
    sin(time_f + 2.094) * 0.5 + 0.5,
    sin(time_f + 4.188) * 0.5 + 0.5
);
vec4 rainbowLight = vec4(rainbow, 1.0) * lightEffect;
color = mix(originalColor, mix(vortexColor, whirlColor, 0.5), 0.6 + 0.4 * sin(time_f));
color += rainbowLight * (1.0 - lightEffect) * 0.3;
}

```

Wave & Oscillation (17 shaders)

Shader #72 - blue wave

File: blue_wave.glsl

Uses modular arithmetic with time-based animation to create repeating, evolving patterns. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Linear mix/lerp blending - Distance field computation - Driven by: time, mouse, audio, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float sat(float x){return clamp(x,0.0,1.0);}
void main(void){
    vec2 uv = tc;
    vec2 m = (iMouse.z>0.0||iMouse.w>0.0)? iMouse.xy/iResolution : vec2(0.5);
    float v = 0.4 + amp*0.2;
    float s = 0.22;
    float width = 0.10;
    float base = mod(time_f*v, s);

```

```

    vec2 dir = normalize(uv - m + 1e-6);
    float r = distance(uv, m);
    float waveSum = 0.0;
    for(int k=0;k<4;k++){
        float rc = base + float(k)*s;
        float a = (r - rc)/width;
        waveSum += exp(-a*a);
    }
    vec2 duv = dir * waveSum * (0.015 + 0.015*uamp);
    vec4 tex = texture(samp, uv + duv);
    float blend = sat(waveSum * (0.45 + 0.55*uamp));
    vec3 blue = mix(tex.rgb, vec3(0.06, 0.28, 1.0), blend);
    color = vec4(blue, 1.0);
}

```

Shader #101 - code wave

File: code_wave.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    float yPos = gl_FragCoord.y / iResolution.y;
    float wavePhase = sin(yPos * 10.0 + time_f * 2.0);
    vec2 tcAdjusted = tc + vec2(0, wavePhase * 0.305);
    color = texture(samp, tcAdjusted);
}

```

Shader #102 - code wave rev

File: code_wave_rev.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    float yPos = 1.0 - (gl_FragCoord.y / iResolution.y);
    float wavePhase = sin(yPos * 10.0 + time_f * 2.0);
    vec2 tcAdjusted = tc + vec2(0, wavePhase * 0.305);
    color = texture(samp, tcAdjusted);
}

```

Shader #103 - code wave side

File: code_wave_side.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    float xNorm = (gl_FragCoord.x / iResolution.x) * 2.0 - 1.0;
    float phase = abs(sin(time_f * 2.0));
    vec2 tcAdjusted = tc + vec2(phase * xNorm * 0.305, 0);
    color = texture(samp, tcAdjusted);
}
```

Shader #173 - cwave

File: cwave.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec2 uv = tc * iResolution;
    uv -= 0.5 * iResolution;
    float dist = length(uv);
    float angle = atan(uv.y, uv.x);
    float wave = sin(dist * 10.0 - time_f * 5.0 + angle * 5.0);
    vec4 texColor = texture(samp, tc);
    color = texColor * (0.5 + 0.5 * wave);
}
```

Shader #174 - cwave2

File: cwave2.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
```

```

uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * iResolution;
    uv -= 0.5 * iResolution;
    float dist = length(uv);
    float angle = atan(uv.y, uv.x);
    float time_t = pingPong(time_f, 20.0);
    float wave = sin(dist * 10.0 - time_f * 5.0 + angle * 5.0);
    wave = sin(wave * time_t);
    vec4 texColor = texture(samp, tc);
    color = texColor * (0.5 + 0.5 * wave);
}

```

Shader #184 - dark psych wave

File: dark_psych_wave.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform vec2 iResolution;
uniform sampler2D samp;
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec2 uv = (tc - 0.5) * 2.0;
    uv.x *= iResolution.x / iResolution.y;
    float time = time_f * 0.5;
    float sine1 = sin(uv.x * 10.0 + time) * cos(uv.y * 10.0 + time);
    float sine2 = sin(uv.y * 15.0 - time) * cos(uv.x * 15.0 - time);
    float sine3 = sin(sqrt(uv.x * uv.x + uv.y * uv.y) * 20.0 + time);
    float pattern = sine1 + sine2 + sine3;
    float colorIntensity = pattern * 0.5 + 0.5;
    vec3 psychedelicColor = vec3(
        sin(colorIntensity * 6.28318 + 0.0) * 0.5 + 0.5,
        sin(colorIntensity * 6.28318 + 2.09439) * 0.5 + 0.5,
        sin(colorIntensity * 6.28318 + 4.18879) * 0.5 + 0.5
    );
    uv = tc * 2.0 - 1.0;
    float t = mod(time_f, 20);
    uv.y *= iResolution.y / iResolution.x;
    float angle = atan(uv.y, uv.x) + t * 20.0;
    vec3 rainbowColor = rainbow(angle / (2.0 * 3.14159));
}

```

```

    vec4 originalColor = texture(samp, tc);
    vec3 blendedColor = mix(originalColor.rgb, rainbowColor, 0.5);
    vec4 finalColor = vec4(sin(blendedColor * t), originalColor.a);
    finalColor = mix(finalColor, vec4(psychedelicColor, 1.0), 0.5);
    vec2 normCoord = ((gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0) * vec2(iResolution.x / iResolution
    ↪ .y, 1.0);
    float distanceFromCenter = length(normCoord);
    float wave = sin(distanceFromCenter * 12.0 - time_f * 4.0);
    vec2 tcAdjusted = tc + (normCoord * 0.301 * wave);
    vec4 textureColor = texture(samp, tcAdjusted);
    color = mix(textureColor, finalColor, 0.5);
}

```

Shader #574 - heat wave

File: heat-wave.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0) + 1.0;
    float time_s = pingPong(time_f, 5.0);
    float frequency = 10.0;
    float amplitude = sin(0.02 * time_t);
    float speed = sin(0.3 * time_s);
    vec2 uv = tc;
    uv.y += sin(uv.x * frequency + time_f * speed) * amplitude;
    color = texture(samp, uv);
}

```

Shader #795 - pulse1

File: pulse1.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;

```

```

uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main() {
    vec2 uv = tc;
    vec2 center = vec2(0.5, 0.5);
    vec2 offset = uv - center;
    float ripple = sin(length(offset) * (20.0 + amp_mid * 15.0) - time_f * 5.0) * (0.05 + amp_low * 0.1)
    ↵ ;
    float angle = atan(offset.y, offset.x) + ripple * sin(time_f);
    float radius = length(offset);
    vec2 swirlUV = center + radius * vec2(cos(angle), sin(angle));
    float pulse = (0.2 + amp_low * 0.3) * sin(time_f * 3.0);
    swirlUV += pulse * normalize(offset);
    vec4 texColor = texture(samp, swirlUV);
    vec3 shiftedColor = texColor.rgb * (0.8 + 0.2 * sin(time_f * 2.0));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    shiftedColor *= 1.0 + _ab * 0.6;
    shiftedColor = mix(shiftedColor, shiftedColor * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 +
    ↵ clamp(amp_high, 0.0, 1.0) * 0.25), _ab);
    color = vec4(shiftedColor, texColor.a);
}

```

Shader #970 - spiral wave

File: spiral_wave.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
float h1(float n){return fract(sin(n)*43758.5453123);}
vec2 h2(float n){return fract(sin(vec2(n, n+1.0))*vec2(43758.5453,22578.1459));}
void main(void){
    vec2 uv = tc;
    float aspect = iResolution.x / iResolution.y;
    float t = time_f * 0.85;
    float t0 = floor(t);
    float a = fract(t);
    float w = a * a * (3.0 - 2.0 * a);
    vec2 p0 = vec2(0.15) + h2(t0) * 0.7;
    vec2 p1 = vec2(0.15) + h2(t0 + 1.0) * 0.7;
    vec2 autoP = mix(p0, p1, w);
    vec2 m = iMouse.z > 0.5 ? (iMouse.xy / iResolution) : autoP;
}

```

```

    vec2 p = (uv - m) * vec2(aspect, 1.0);
    float d = length(p);
    float fall = smoothstep(0.6, 0.0, d);
    float k = amp * 0.25;
    float bend = k * fall;
    vec2 perp = normalize(vec2(-p.y, p.x));
    if (!all(greaterThan(abs(p), vec2(0.0)))) perp = vec2(0.0);
    vec2 bendOff = perp * bend * (0.25 + 0.75 * sin(3.0 * d + time_f * 2.0));
    float stretch = uamp * 0.7 * fall;
    vec2 stretchOff = p * stretch * (0.6 + 0.4 * cos(4.0 * d - time_f * 1.6));
    float spiral = 0.9 * fall;
    float ang = atan(p.y, p.x);
    float rot = spiral * (0.8 + 0.2 * sin(time_f * 0.7)) * sin(time_f * 1.2 + 3.0 * d);
    float r = length(p);
    vec2 spin = vec2(cos(ang + rot), sin(ang + rot)) * r - p;
    vec2 off = (bendOff + stretchOff + spin) / vec2(aspect, 1.0);
    float wave = sin((uv.x + uv.y) * 18.0 + time_f * 12.0) * 0.003 * fall;
    vec2 wobble = vec2(0.0, wave);
    vec2 tuv = uv + off + wobble;
    color = texture(samp, tuv);
}

```

Shader #971 - spiral wave 1

File: spiral_wave_1.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0);
    vec2 center = vec2(0.5, 0.5);
    float dist = sqrt((tc.x - center.x) * (tc.x - center.x) + (tc.y - center.y) * (tc.y - center.y));
    dist = sin(dist * time_t);
    float angle = atan(tc.y - center.y, tc.x - center.x);
    float wave = sin(dist * 10.0 + angle * 5.0 - time_f * 5.0);
    vec2 offset = tc + wave * 0.02 * vec2(cos(angle), sin(angle));
    vec4 texColor = texture(samp, offset);
    color = texColor;
}

```

Shader #1056 - wave 1

File: wave-1.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution


```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main() {
    vec2 pos = tc;
    float wave_x = sin((tc.x + time_f) * 5.0);
    float wave_y = cos((tc.y + time_f) * 8.0);
    pos.x += wave_x;
    pos.y += wave_y;
    color = texture(samp, sin(pos * time_f));
}

```

Shader #1057 - wave diag

File: wave_diag.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void) {
    vec2 normCoord = gl_FragCoord.xy / iResolution.xy;
    float diagonalDistance = (normCoord.x + normCoord.y - 1.0) * sqrt(2.0);
    float antiDiagonalDistance = (normCoord.x - normCoord.y) * sqrt(2.0);
    float diagonalWave = sin((diagonalDistance + time_f) * 5.0);
    float antiDiagonalWave = cos((antiDiagonalDistance + time_f) * 5.0);
    float combinedWave = (diagonalWave + antiDiagonalWave) * 0.5;
    vec2 waveAdjusted = vec2(tc.x + combinedWave * 0.202, tc.y + combinedWave * 0.202);
    color = texture(samp, waveAdjusted);
}

```

Shader #1058 - wave spiral

File: wave_spiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
void main(void) {
    vec2 normCoord = (tc * 2.0 - 1.0) * vec2(iResolution.x / iResolution.y, 1.0);
}

```

```

float waveStrength = 0.05;
float waveFrequency = 3.0;
vec2 wave = vec2(sin(normCoord.y * waveFrequency + time_f) * waveStrength,
                 cos(normCoord.x * waveFrequency + time_f) * waveStrength);
normCoord += wave;
float dist = length(normCoord);
float angle = atan(normCoord.y, normCoord.x);\
float spiralAmount = tan(time_f) * 3.0;
angle += dist * spiralAmount;
vec2 spiralCoord = vec2(cos(angle), sin(angle)) * dist;
spiralCoord = (spiralCoord / vec2(iResolution.x / iResolution.y, 1.0) + 1.0) / 2.0;
color = texture(samp, spiralCoord);
}

```

Shader #1094 - zoom pulse

File: zoom_pulse.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
void main(void) {
    vec2 uv = tc - 0.5;
    float distance = length(uv);
    float audioBoost = 0.5 + amp_low * 1.0;
    float pulse = 1.0 + audioBoost * sin(time_f * 2.0 + distance * 20.0);
    uv *= pulse;
    uv += 0.5;
    color = texture(samp, uv);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #1095 - zoom pulse2

File: zoom_pulse2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 10.0);
    vec2 uv = tc - 0.5;
    float distance = length(uv);
    float audioBoost = 0.5 + amp_low * 1.0;
    float pulse = 1.0 + audioBoost * sin(time_t * 2.0 + distance * 20.0);
    uv *= pulse;
    uv += 0.5;
    color = texture(samp, uv);
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    color.rgb *= 1.0 + _ab * 0.6;
    color.rgb = mix(color.rgb, color.rgb * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(
        ↪ amp_high, 0.0, 1.0) * 0.25), _ab);
}

```

Shader #1096 - zoom pulse3

File: zoom_pulse3.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    float time_t = pingPong(time_f, 7.0);
    vec2 uv = tc - 0.5;
    float distance = length(uv);
    float pulse = 1.0 + 0.5 * sin(time_t * 2.0 + distance * 20.0);
    uv *= pulse;
    uv += sin(0.5 * time_t);
    color = texture(samp, uv);
}

```

Noise & Random (15 shaders)

Shader #1 - 4ac rand

File: 4ac_rand.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
vec4 multiLevelBlend(vec2 tc, sampler2D tex) {
    vec2 cord1 = vec2(tc.x / 2.0, tc.y / 2.0);
    vec2 cord2 = vec2(tc.x / 4.0, tc.y / 4.0);
    vec2 cord3 = vec2(tc.x / 8.0, tc.y / 8.0);
    vec4 col1 = texture(tex, cord1);
    vec4 col2 = texture(tex, cord2);
    vec4 col3 = texture(tex, cord3);
    return vec4(col1.r + col2.r, col1.g + col2.g, col1.b + col3.b, 1.0);
}
void main(void) {
    vec2 normCoord = tc;
    float distortionStrength = 0.02;
    float distortionFrequency = 15.0;
    float distortion1 = sin(normCoord.y * distortionFrequency + time_f) * distortionStrength;
    float distortion2 = cos((normCoord.x + normCoord.y) * distortionFrequency + time_f) *
        ↪ distortionStrength;
    vec2 distortedTC = normCoord + vec2(distortion1, distortion2);
    vec4 baseColor = texture(samp, distortedTC);
    vec4 blendedColor = multiLevelBlend(distortedTC, samp);
    float val = (blendedColor.r + blendedColor.g + blendedColor.b) * 0.5;
    for (int q = 0; q < 3; ++q) {
        blendedColor[q] *= sin(val * blendedColor[q] / 0.3);
    }
    color = (0.5 * blendedColor) + (0.5 * baseColor);
}
```

Shader #2 - 4rand

File: 4rand.glsl

Quantizes colors to a restricted palette, producing a retro/pixelated aesthetic. Function to quantize colors to an 8-bit palette Function to apply a feedback loop Apply deformation effect

Technique: Indexed color palette look-up table - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 NESPalette(vec3 col) {
    float[16] rVals = float[16](0.0, 0.26, 0.5, 0.74, 1.0, 0.88, 0.62, 0.38, 0.14, 0.08, 0.22, 0.54,
        ↪ 0.78, 0.92, 0.96, 0.1);
```

```

float[16] gVals = float[16](0.0, 0.13, 0.25, 0.37, 0.5, 0.44, 0.31, 0.19, 0.08, 0.13, 0.37, 0.44,
↪ 0.56, 0.69, 0.82, 0.96);
float[16] bVals = float[16](0.0, 0.06, 0.13, 0.25, 0.38, 0.44, 0.56, 0.69, 0.82, 0.96, 0.74, 0.5,
↪ 0.25, 0.13, 0.06, 0.0);
int idxR = int(floor(col.r * 15.0 + 0.5));
int idxG = int(floor(col.g * 15.0 + 0.5));
int idxB = int(floor(col.b * 15.0 + 0.5));
return vec3(rVals[idxR], gVals[idxG], bVals[idxB]);
}
vec4 echo_loop(vec4 baseColor, int count) {
    vec4 col = baseColor;
    for (int i = 2; i < count; i += 2) {
        vec4 pos = texture(samp, tc / (i * fract(time_f)));
        col = (0.6 * col) + (0.3 * pos);
    }
    return col;
}
void main(void) {
    vec2 normCoord = tc;
    float warpAmount = tan(time_f);
    vec2 warp = vec2(
        sin(normCoord.y * 10.0 + time_f) * warpAmount,
        cos(normCoord.x * 10.0 + time_f) * warpAmount
    );
    vec2 warpedCoord = normCoord + warp;
    vec4 texColor = texture(samp, warpedCoord);
    vec3 quantizedColor = NESPalette(texColor.rgb);
    vec4 quantizedTexColor = vec4(quantizedColor, texColor.a);
    vec4 echoedColor = echo_loop(quantizedTexColor, 14);
    color = vec4(1.0 - echoedColor.rgb, echoedColor.a);
}

```

Shader #147 - composite static

File: composite-static.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Computes luminance for brightness-based effects.

Technique: Color quantization via floor rounding - Scanline overlay using modular y-coordinate - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float PI(){return 3.14159265358979323846;}
float hash(vec2 p){
    p=fract(p*vec2(123.34,345.45));
    p+=dot(p,p*34.345);
    return fract(p.x*p.y);
}
vec3 triadMask(vec2 uv){
    float px=floor(uv.x*iResolution.x);
    float m=mod(px,3.0);
    vec3 a=vec3(1.10,0.88,0.88);
    vec3 b=vec3(0.88,1.10,0.88);
    vec3 c=vec3(0.88,0.88,1.10);

```

```

    return mix(mix(c,b,step(1.0,m)),a,step(2.0,m));
}
float scanline(vec2 uv){
    float y=uv.y*iResolution.y;
    return 0.84+0.16*sin(PI()*y);
}
vec3 halation(vec2 uv){
    vec2 px=1.0/iResolution;
    vec3 s=vec3(0.0);
    s+=texture(samp,uv+vec2( px.x, 0.0)).rgb*0.10;
    s+=texture(samp,uv+vec2(-px.x, 0.0)).rgb*0.10;
    s+=texture(samp,uv+vec2( 0.0, px.y)).rgb*0.10;
    s+=texture(samp,uv+vec2( 0.0,-px.y)).rgb*0.10;
    s+=texture(samp,uv+vec2( px.x, px.y)).rgb*0.06;
    s+=texture(samp,uv+vec2(-px.x, px.y)).rgb*0.06;
    s+=texture(samp,uv+vec2( px.x,-px.y)).rgb*0.06;
    s+=texture(samp,uv+vec2(-px.x,-px.y)).rgb*0.06;
    return s;
}
vec3 rgb2yiq(vec3 c){
    float Y=dot(c,vec3(0.299,0.587,0.114));
    float I=dot(c,vec3(0.596,-0.274,-0.322));
    float Q=dot(c,vec3(0.211,-0.523,0.312));
    return vec3(Y,I,Q);
}
vec3 yiq2rgb(vec3 yIQ){
    float Y=yIQ.x, I=yIQ.y, Q=yIQ.z;
    return mat3(1.0, 1.0, 1.0,
                0.956,-0.272,-1.106,
                0.621,-0.647, 1.703) * vec3(Y,I,Q);
}
void main(){
    vec2 uv=tc;
    vec2 px=1.0/iResolution;
    float hs=(sin(time_f*120.0)+sin((uv.y+time_f*0.61)*87.0))*0.0006;
    uv.x+=hs;
    float jitter=(hash(vec2(floor(gl_FragCoord.y*0.5), time_f))*2.0-1.0)*0.00045;
    uv.x+=jitter;
    vec3 s0=texture(samp,uv).rgb;
    vec3 sL1=texture(samp,uv-vec2(px.x,0)).rgb;
    vec3 sR1=texture(samp,uv+vec2(px.x,0)).rgb;
    vec3 sL2=texture(samp,uv-vec2(px.x*2.0,0)).rgb;
    vec3 sR2=texture(samp,uv+vec2(px.x*2.0,0)).rgb;
    vec3 y0=rgb2yiq(s0);
    vec3 yL1=rgb2yiq(sL1);
    vec3 yR1=rgb2yiq(sR1);
    vec3 yL2=rgb2yiq(sL2);
    vec3 yR2=rgb2yiq(sR2);
    float Y = 0.40*y0.x + 0.24*(yL1.x+yR1.x) + 0.06*(yL2.x+yR2.x);
    float Ih = 0.40*y0.y + 0.24*(yL1.y+yR1.y) + 0.06*(yL2.y+yR2.y);
    float Qh = 0.36*y0.z + 0.12*(yL1.z+yR1.z) + 0.02*(yL2.z+yR2.z);
    float sub=227.5;
    float line=floor(gl_FragCoord.y);
    float phase=6.2831853*(gl_FragCoord.x*sub/iResolution.x + mod(line,2.0)*0.5 + time_f*0.03);
    mat2 R=mat2(cos(phase),-sin(phase),sin(phase),cos(phase));
    vec2 IQm = R*vec2(Ih,Qh);
    float Yhp = Y - 0.5*(yL1.x+yR1.x);
    vec2 cc = vec2(cos(phase),sin(phase))*Yhp*0.18;
    IQm += cc;
    vec2 IQd = R*vec2(
        0.50*y0.y + 0.30*yL1.y + 0.20*yR1.y,
        0.55*y0.z + 0.28*yL1.z + 0.17*yR1.z
    );
}

```

```

);
float tHop = 0.5;
vec2 IQdelay = vec2(
    texture(samp, uv-vec2(px.x*tHop,0)).rg
);
vec3 yDelay = rgb2yiq(texture(samp, uv-vec2(px.x*tHop,0)).rgb);
IQd = mix(IQd, vec2(yDelay.y,yDelay.z), 0.35);
vec2 IQ = mix(IQm, IQd, 0.5);
IQ *= 0.92;
vec3 ntsc = yiq2rgb(vec3(Y, IQ.x, IQ.y));
vec3 bleedH = (texture(samp,uv+vec2(px.x*1.2,0)).rgb + texture(samp,uv-vec2(px.x*1.2,0)).rgb)*0.25;
vec3 ghost = texture(samp, uv-vec2(px.x*2.2,0)).rgb*0.10;
vec3 base = mix(ntsc, ntsc*0.6 + bleedH*0.4 + ghost, 0.45);
float sl=scanline(uv);
vec3 tri=triadMask(uv);
float grille=0.95+0.05*sin(uv.x*iResolution.x*PI()*0.5);
float flick=0.992+0.008*sin(time_f*360.0);
vec3 bloom=halation(uv)*0.28;
float n = (hash(gl_FragCoord.xy+time_f*vec2(173.1,91.7))*2.0-1.0)*0.018;
n = sin(n * (time_f * PI()));
base += n;
vec3 linear=pow(clamp(base,0.0,1.0),vec3(2.2))+pow(bloom,vec3(2.2))*0.55;
vec3 shaped=linear*tri*sl*grille*flick;
vec3 outc=pow(shaped,vec3(1.0/2.2));
float black=0.02;
float white=1.05;
outc=clamp((outc-black)/(white-black),0.0,1.0);
color=vec4(outc,1.0);
}

```

Shader #289 - echo rand

File: echo_rand.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);

```

```

    vec4 color2 = texture(samp, tc / 2);
    vec2 pos1 = tc;
    vec2 pos2 = tc;
    vec4 iv1 = inc_valuex/255;
    vec4 iv2 = inc_value/255;
    pos1[0] = iv1[0]-pos1[0];
    pos1[1] = iv1[1]-pos1[1];
    pos2[0] = iv2[0]-pos2[0];
    pos2[1] = iv2[1]-pos2[1];
    vec4 color3 = texture(samp, (pos1/4)*0.9);
    vec4 color4 = texture(samp, (pos2/8)*0.9);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```

Shader #290 - echo rand 2

File: echo_rand_2.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec4 inc_valuex;
uniform vec4 inc_value;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 color2 = texture(samp, tc / 2);
    vec2 pos1 = tc;
    vec2 pos2 = tc;
    vec4 iv1 = inc_valuex/255;
    vec4 iv2 = inc_value/255;
    pos1[0] = iv1[0];
    pos1[1] = iv1[1];
    pos2[0] = iv2[0];
    pos2[1] = iv2[1];
    vec4 color3 = texture(samp, pos1/4);
    vec4 color4 = texture(samp, pos2/8);
    color = (color * 0.4) + (color2 * 0.4) + (color3 * 0.4) + (color4 * 0.4) ;
}

```


Shader #817 - rainbow expand noise

File: rainbow_expand_noise.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(float n) {
    return fract(sin(n) * 43758.5453123);
}
vec2 random2(vec2 st) {
    st = vec2(dot(st, vec2(127.1, 311.7)),
              dot(st, vec2(269.5, 183.3)));
    return -1.0 + 2.0 * fract(sin(st) * 43758.5453123);
}
vec2 smoothRandom2(float t) {
    float t0 = floor(t);
    float t1 = t0 + 1.0;
    vec2 rand0 = random2(vec2(t0));
    vec2 rand1 = random2(vec2(t1));
    float mix_factor = fract(t);
    return mix(rand0, rand1, smoothstep(0.0, 1.0, mix_factor));
}
vec3 rainbow(float t) {
    t = fract(t);
    float r = abs(t * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(t * 6.0 - 2.0);
    float b = 2.0 - abs(t * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main(void) {
    vec2 uv = tc * 2.0 - 1.0;
    uv.y *= iResolution.y / iResolution.x;
    float time_t = pingPong(time_f, 15.0) + 1.0;
    float wave = sin(uv.x * 10.0 + time_t * 2.0) * 0.1;
    vec2 random_direction = smoothRandom2(time_t) * 0.5;
    float expand = 0.5 + 0.5 * sin(time_t * 2.0);
    vec2 spiral_uv = uv * expand + random_direction;
    float angle = atan(spiral_uv.y + wave, spiral_uv.x) + time_t * 2.0;
    vec3 rainbow_color = rainbow(angle / (2.0 * 3.14159));
    vec4 original_color = texture(samp, tc);
    vec3 blended_color = mix(original_color.rgb, rainbow_color, 0.5);
    color = vec4(sin(blended_color * time_t), original_color.a);
}
```

Shader #840 - random colors

File: random_colors.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    for(int i = 0; i < 3; ++i) {
        color[i] = 255/random_value[i];
    }
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #842 - random pos fish

File: random_pos_fish.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Linear mix/lerp blending - Driven by: time

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : (length * 2.0 - m);
}
float h1(float n){ return fract(sin(n)*43758.5453123); }
```

```

vec2 h2(float n){ return fract(sin(vec2(n,n+1.0))*vec2(43758.5453,22578.1459)); }
void main(void) {
    float time_t = pingPong(time_f, 10.0) + 2.0;
    float rate = 0.8;
    float t = time_f * rate;
    float t0 = floor(t);
    float a = fract(t);
    float w = a*a*(3.0-2.0*a);
    vec2 p0 = vec2(0.15) + h2(t0) * 0.7;
    vec2 p1 = vec2(0.15) + h2(t0 + 1.0) * 0.7;
    vec2 center = mix(p0, p1, w);
    vec2 uv = tc - center;
    float lenv = length(uv);
    float factor = sqrt(lenv) * 0.5;
    float s = 1.0 + sin(factor * time_t);
    uv *= s;
    color = texture(samp, uv + center);
}

```

Shader #843 - random resize

File: random_resize.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 resizeVertical(vec2 uv, float blockSize, float timeFactor) {
    float blockHeight = blockSize;
    float offset = sin(time_f * timeFactor + uv.x * 10.0) * 0.05;
    uv.y += step(mod(uv.y, blockHeight), blockHeight * 0.5) * offset;
    return fract(uv);
}
vec2 randomShift(vec2 uv, float intensity) {
    float shiftX = rand(uv) * intensity * cos(time_f * 2.0);
    float shiftY = rand(uv + vec2(1.0, 1.0)) * intensity * sin(time_f * 2.0);
    return uv + vec2(shiftX, shiftY);
}
void main() {
    vec2 uv = tc;
    float blockSize = 0.1;
    uv = resizeVertical(uv, blockSize, 5.0);
    if (rand(floor(uv / blockSize)) > 0.5) {
        uv = randomShift(uv, 0.02);
    }
    vec4 texColor = texture(samp, fract(uv));
    vec3 glitchColor = texColor.rgb * (1.0 + 0.3 * sin(time_f * 10.0 + uv.y * 50.0));
    color = mix(texColor, vec4(glitchColor, texColor.a), 0.5);
}

```

Shader #844 - random rgb

File: random_rgb.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec2 uv = tc;
    uv[0] = uv[0] / alpha_r;
    uv[1] = uv[1] / alpha_g;
    uv[0] = tc[0]/uv[0];
    uv[1] = tc[1]/uv[1];
    vec4 color2 = texture(samp, uv);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    color[0] = (color[0] * color2[0]) / uv[0];
    color[1] = (color[1] * color2[1]) / uv[1];
    color[2] = (color[0] * color2[0]) / (uv[0]+uv[1]);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #845 - random soul

File: random_soul.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling

effect.

Technique: Polar coordinate transformation (atan2 + length) - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float len){float m=mod(x,len*2.0);return m<=len?m:len*2.0-m;}
float h(float n){return fract(sin(n)*43758.5453123);}
vec2 h2(float n){return vec2(h(n),h(n+1.23));}
vec3 rainbow(float t){
    t=fract(t);
    float r=abs(t*6.0-3.0)-1.0;
    float g=2.0-abs(t*6.0-2.0);
    float b=2.0-abs(t*6.0-4.0);
    return clamp(vec3(r,g,b),0.0,1.0);
}
void main(){
    vec2 res=iResolution;
    vec2 uv=tc*2.0-1.0;
    uv.y*=res.y/res.x;
    float t=time_f*0.25;
    float seg=floor(t);
    float a=fract(t);
    vec2 p0=-0.5+h2(seg)*1.0;
    vec2 p1=-0.5+h2(seg+1.0)*1.0;
    a=a*(3.0-2.0*a);
    vec2 swirlC=mix(p0,p1,a);
    vec2 d=uv-swirlC;
    float r=length(d);
    float k=0.85;
    float s=0.45;
    float theta=k*exp(-r*2.5)*(1.0+0.5*sin(time_f*0.6));
    float ct=cos(theta);
    float st=sin(theta);
    vec2 rot=vec2(d.x*ct-d.y*st,d.x*st+d.y*ct)+swirlC;
    float angle=atan(uv.y,uv.x)+time_f*20.0;
    vec3 rain=rainbow(angle/(2.0*3.1415926535));
    float bulge_strength=0.2;
    float distortion=pow(length(rot),2.0)*bulge_strength;
    vec2 distorted=rot*(1.0+distortion);
    vec2 tex=distorted*0.5+0.5;
    vec4 base=texture(samp,tex);
    vec3 blend=mix(base.rgb,rain,0.5);
    color=vec4(sin(blend*time_f),base.a);
}
```

Shader #847 - random square

File: random_square.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec2 posv = gl_FragCoord.xy / random_value[0];
    vec2 posi = fract(posv)*0.5;
    color[0] = (color[0] * posi[0])*alpha_r;
    color[1] = (color[1] * posi[1])*alpha_g;
    color[2] = (color[2] * posi[0]+posi[1])*alpha_b;
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #848 - random strobe

File: random_strobe.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;

```

```

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    for(int i = 0; i < 3; ++i) {
        color[i] += random_value[i]/255;
    }
}

```

Shader #849 - random uneven square

File: random_uneven_square.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
in float timeval;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform vec2 iResolution;
uniform float restore_black;
in vec2 iResolution_;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    float x = iResolution_.x / (gl_FragCoord.x / alpha_r);
    float y = iResolution_.y / (gl_FragCoord.y / alpha_g);
    vec2 f = fract(vec2(x,y)) / timeval;
    color[0] = (color[0]*15.5 * (f[0]+f[1]))*(alpha_r+0.1);
    color[1] = (color[1]*15.5 * (f[0]+f[1]))*(alpha_g+0.1);
    color[2] = (color[2]*15.5 * (f[0]+f[1]))*(alpha_b+0.1);
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
    }
}

```

```

        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #945 - singleton noise

File: singleton-noise.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Texture coordinates passed from vertex shader Single texture unit for all mappings and manipulations Screen resolution, useful for dynamic effects

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform float alpha_r;
const float PI = 3.1415926535897932384626433832795;
float noise(vec2 p) {
    vec2 ip = floor(p);
    vec2 fp = fract(p);
    float a = dot(sin(ip), vec2(1.0, 17.0));
    float b = dot(cos(ip), vec2(1.0, 17.0));
    float n_x = mix(mix(a, b, fp.x), mix(sin(a), cos(b), fp.x), fp.y);
    return n_x;
}
float pingPong(float x, float length) {
    float modVal = mod(x, length * 2.0);
    return modVal <= length ? modVal : length * 2.0 - modVal;
}
void main() {
    vec2 uv = tc;
    int OCTAVES = int(alpha_r * pingPong(time_f,10.0));
    float detailNoise = 0.0;
    for (int i = 0; i < OCTAVES; ++i) {
        float freq = pow(2.0, float(i));
        float amp = pow(0.5, float(OCTAVES - i));
        detailNoise += noise(uv * freq) * amp;
    }
    uv += vec2(detailNoise) * 0.1;
    color = texture(samp, uv);
}

```

Tile & Mosaic (11 shaders)

Shader #411 - frac shader04 grid

File: frac_shader04_grid.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect. Responds to mouse position for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - 2D rotation matrix - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, mouse, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp_peak;
uniform float amp_rms;
uniform float amp_smooth;
uniform float amp_low;
uniform float amp_mid;
uniform float amp_high;
uniform float iamp;

float pingPong(float x, float length){
    float m = mod(x, length*2.0);
    return m <= length ? m : length*2.0 - m;
}

vec2 rotateUV(vec2 uv, float angle, vec2 c, float aspect){
    float s = sin(angle), cc = cos(angle);
    vec2 p = uv - c;
    p.x *= aspect;
    p = mat2(cc, -s, s, cc) * p;
    p.x /= aspect;
    return p + c;
}

vec2 reflectUV(vec2 uv, float segments, vec2 c, float aspect){
    vec2 p = uv - c;
    p.x *= aspect;
    float ang = atan(p.y, p.x);
    float rad = length(p);
    float stepA = 6.28318530718 / segments;
    ang = mod(ang, stepA);
    ang = abs(ang - stepA * 0.5);
    vec2 r = vec2(cos(ang), sin(ang)) * rad;
    r.x /= aspect;
    return r + c;
}

vec2 fractalFold(vec2 uv, float zoom, float t, vec2 c, float aspect){
    vec2 p = uv;
    for(int i=0;i<6;i++){
        p = abs((p - c) * (zoom + 0.15*sin(t*(0.35 + amp_low * 0.2)+float(i)))) - 0.5 + c;
        p = rotateUV(p, t*0.12 + float(i)*0.07, c, aspect);
    }
    return p;
}

vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec3 movingGradient(vec2 uv, vec2 c, float t, float aspect){
    vec2 ar = vec2(aspect, 1.0);
    vec2 p = (uv - c) * ar;
    vec2 d = normalize(vec2(cos(t*0.27), sin(t*0.31)));
```

```

float s = dot(p, d);
float band = 0.5 + 0.5*sin(s*6.28318530718*0.35 + t*0.9);
float h = fract(s*0.22 + t*0.07 + 0.15*sin(t*0.33));
float S = 0.75 + 0.25*sin(t*0.21 + s*2.0);
float V = 0.75 + 0.25*band;
vec3 base = hsv2rgb(vec3(h, S, V));
float edge = smoothstep(0.2, 0.8, band);
return mix(base*0.6, base, edge);
}

void main(){
float aspect = iResolution.x / iResolution.y;
vec2 ar = vec2(aspect, 1.0);
vec2 m = (iMouse.z > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5);
vec2 uv = tc;
vec4 originalTexture = texture(samp, tc);
float seg = 6.0 + (2.0 + amp_mid * 2.0)*sin(time_f*0.33);
vec2 kUV = reflectUV(uv, seg, m, aspect);
float foldZoom = 1.45 + (0.55 + amp_low * 0.4) * sin(time_f * 0.42);
kUV = fractalFold(kUV, foldZoom, time_f, m, aspect);
kUV = rotateUV(kUV, time_f * (0.23 + amp_low * 0.15), m, aspect);
vec2 p = (kUV - m) * ar;
float base = 1.82 + 0.18*sin(time_f*0.2);
float period = log(base);
float tz = time_f * (0.65 + amp_low * 0.3);
float r = length(p) + 1e-6;
float ang = atan(p.y, p.x) + tz * 0.35 + 0.35*sin(r*9.0 + time_f*0.8);
float k = fract((log(r) - tz) / period);
float rw = exp(k * period);
vec2 pwrap = vec2(cos(ang), sin(ang)) * rw;
vec2 u0 = fract(pwrap / ar + m);
vec2 u1 = fract((pwrap*1.045) / ar + m);
vec2 u2 = fract((pwrap*0.955) / ar + m);
vec2 dir = normalize(pwrap + 1e-6);
vec2 off = dir * ((0.0015 + amp_high * 0.002) + 0.001 * sin(time_f*1.3)) * vec2(1.0, 1.0/aspect);
float hue = fract(ang*0.15 + time_f*0.08 + k*0.5);
float sat = 0.8 - 0.2*cos(time_f*0.7 + r*6.0);
float val = 0.8 + 0.2*sin(time_f*0.9 + k*6.28318530718);
vec3 tint = hsv2rgb(vec3(hue, sat, val));
float ring = smoothstep(0.0, 0.7, sin(log(r+1e-3)*8.0 + time_f*1.2));
float pulse = 0.5 + (0.5 + amp_rms * 0.3)*sin(time_f*2.0 + r*18.0 + k*12.0);
float vign = 1.0 - smoothstep(0.75, 1.2, length((tc - m)*ar));
vign = mix(0.85, 1.2, vign);
float blendFactor = 0.58;
float rC = texture(samp, u0 + off).r;
float gC = texture(samp, u1).g;
float bC = texture(samp, u2 - off).b;
vec3 kaleidoRGB = vec3(rC, gC, bC);
vec4 kaleidoColor = vec4(kaleidoRGB, 1.0) * vec4(tint, 1.0);
vec4 merged = mix(kaleidoColor, originalTexture, blendFactor);
merged.rgb *= (0.75 + 0.25*ring) * (0.85 + 0.15*pulse) * vign;
vec3 bloom = merged.rgb * merged.rgb * 0.18 + pow(max(merged.rgb-0.6,0.0), vec3(2.0))*0.12;
vec3 outFractal = merged.rgb + bloom;
float wob = 0.9 + 0.1*sin(time_f + r*12.0 + k*9.0);
outFractal *= wob;
vec3 grad = movingGradient(tc, m, time_f, aspect);
float gradAmt = 0.35 + 0.25*sin(time_f*0.5 + r*5.0 + k*9.0);
vec3 screenBlend = 1.0 - (1.0 - outFractal) * (1.0 - grad);
outFractal = mix(outFractal, screenBlend, gradAmt);
outFractal = mix(outFractal, outFractal * originalTexture.rgb, 0.8);
float sparkle = abs(sin(time_f * 10.0 + tc.x * 100.0) * cos(time_f * 15.0 + tc.y * 100.0));
vec3 gridMixed = mix(originalTexture.rgb, outFractal, sparkle);
vec3 outCol = sin(gridMixed * (0.5 + 0.5*pingPong(time_f, 12.0)));

```

```

    outCol = clamp(outCol, vec3(0.08), vec3(0.96));
    float _ab = clamp(amp_peak, 0.0, 1.0);
    float _abass = clamp(amp_low, 0.0, 1.0);
    outCol *= 1.0 + _ab * 0.6;
    outCol = mix(outCol, outCol * vec3(1.0 + _abass * 0.3, 1.0 - _abass * 0.15, 1.0 + clamp(amp_high,
    ↪ 0.0, 1.0) * 0.25), _ab);
    color = vec4(outCol, 1.0);
}

```

Shader #553 - grid spiral

File: grid-spiral.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Hermite smoothstep interpolation
 - Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
void main(void) {
    vec4 base = texture(samp, tc);
    float sparkle = abs(sin(time_f * 10.0 + tc.x * 100.0) * cos(time_f * 15.0 + tc.y * 100.0));
    vec3 magicalColor = vec3(sin(time_f * 2.0) * 0.5 + 0.5, cos(time_f * 3.0) * 0.5 + 0.5, sin(time_f *
    ↪ 4.0) * 0.5 + 0.5);
    vec3 glow = magicalColor * sparkle;
    vec2 p = tc - 0.5;
    p.x *= iResolution.x / iResolution.y;
    float r = length(p);
    float ang = atan(p.y, p.x) + time_f * 0.5 + r * 8.0;
    vec2 s = vec2(cos(ang), sin(ang)) * r;
    s.x /= (iResolution.x / iResolution.y);
    s += 0.5;
    float N = 18.0;
    vec2 q = s * N;
    vec2 g = abs(fract(q) - 0.5);
    vec2 w = fwidth(q);
    float lx = 1.0 - smoothstep(0.0, w.x * 1.5, g.x);
    float ly = 1.0 - smoothstep(0.0, w.y * 1.5, g.y);
    float grid = max(lx, ly);
    vec3 spiralColor = 0.5 + 0.5 * cos(vec3(0.0, 2.0, 4.0) + ang * 4.0 + time_f * 2.0);
    vec3 gridRGB = spiralColor * grid * 0.65;
    vec3 rgb = clamp(base.rgb + glow + gridRGB, 0.0, 1.0);
    color = vec4(rgb, base.a);
}

```

Shader #555 - grid pattern

File: grid_pattern.glsl

Animates the image using time-driven sinusoidal and cosinusoidal wave distortions.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
void main(void) {
    vec4 texColor = texture(samp, tc);
    float sparkle = abs(sin(time_f * 10.0 + tc.x * 100.0) * cos(time_f * 15.0 + tc.y * 100.0));
    vec3 magicalColor = vec3(sin(time_f * 2.0) * 0.5 + 0.5, cos(time_f * 3.0) * 0.5 + 0.5, sin(time_f *
    ↪ 4.0) * 0.5 + 0.5);
    color = texColor + vec4(magicalColor * sparkle, 1.0);
}

```

Shader #556 - grid rand strobe

File: grid_rand_strobe.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec2 cord1 = vec2(tc[0]/3, tc[1]/3);
    vec2 cord2 = vec2(tc[0]/6, tc[1]/6);
    vec2 cord3 = vec2(tc[0]/9, tc[1]/9);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    vec2 val = gl_FragCoord.xy / 3 * random_value[0];
    vec2 f = fract(val);
    color[0] = (color[0] + col1[2]) * f[0];
    color[1] = (color[1] + col2[1]) * f[1];
    color[2] = (color[2] + col3[0]) * f[0]+f[1];
    ivec3 int_color;

```

```

    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #557 - grid strobe

File: grid_strobe.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec2 cord1 = vec2(tc[0]/3, tc[1]/3);
    vec2 cord2 = vec2(tc[0]/6, tc[1]/6);
    vec2 cord3 = vec2(tc[0]/9, tc[1]/9);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    vec2 val = gl_FragCoord.xy / 3 * alpha_r;
    vec2 f = fract(val);
    color[0] = (color[0] + col1[2]) * f[0];
    color[1] = (color[1] + col2[1]) * f[1];
    color[2] = (color[2] + col3[0]) * f[0]+f[1];
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

```
}  
}
```

Shader #558 - gridcolor gradient

File: gridcolor_gradient.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time, resolution

```
#version 330  
in vec2 tc;  
out vec4 color;  
uniform float alpha_r;  
uniform float alpha_g;  
uniform float alpha_b;  
uniform float alpha;  
uniform vec4 optx;  
uniform vec4 random_var;  
uniform float alpha_value;  
uniform mat4 mv_matrix;  
uniform mat4 proj_matrix;  
uniform sampler2D samp;  
uniform float value_alpha_r, value_alpha_g, value_alpha_b;  
uniform float index_value;  
uniform float time_f;  
uniform float restore_black;  
in vec2 iResolution;  
vec4 xor_RGB(vec4 icolor, ivec4 isource) {  
    ivec3 int_color;  
    for(int i = 0; i < 3; ++i) {  
        int_color[i] = int(255 * icolor[i]);  
        int_color[i] = int_color[i]^isource[i];  
        if(int_color[i] > 255)  
            int_color[i] = int_color[i]%255;  
        icolor[i] = float(int_color[i])/255;  
    }  
    icolor.a = 1.0;  
    return icolor;  
}  
void main(void)  
{  
    color = texture(samp, tc);  
    ivec4 source = ivec4(255 * color);  
    vec2 pos = gl_FragCoord.xy/iResolution.xy;  
    pos[0] += 0.01;  
    pos[1] += 0.01;  
    vec4 color2 = texture(samp, pos);  
    color *= vec4(tc, tc[0]+tc[1], 1);  
    color = (color * 0.5) + (0.5 * color2);  
    color = xor_RGB(color, source);  
}
```

Shader #759 - pigrid

File: pigrid.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time, resolution

```
#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
in vec2 iResolution;
void main(void)
{
    color = texture(samp, tc);
    ivec4 source = ivec4(255 * color);
    vec2 pos = gl_FragCoord.xy/iResolution.xy;
    color = color / sin(3.1415926535/alpha);
    color[0] *= pos[0];
    color[1] *= pos[1];
    color[2] *= pos[0]+pos[1];
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}
```

Shader #839 - rand blocks

File: rand_blocks.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Linear mix/lerp blending -
Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
```

```

float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
vec2 glitchOffset(vec2 uv, float intensity) {
    float glitchX = rand(uv) * intensity * sin(time_f);
    float glitchY = rand(uv + vec2(1.0, 1.0)) * intensity * cos(time_f);
    return uv + vec2(glitchX, glitchY);
}
void main() {
    vec2 uv = tc;
    float blockSize = 0.1;
    vec2 blockCoord = floor(uv / blockSize);
    vec2 localCoord = fract(uv / blockSize);
    float randomSeed = rand(blockCoord);
    if (randomSeed > 0.5) {
        uv = glitchOffset(uv, 0.05);
    }
    vec3 texColor = texture(samp, uv).rgb;
    vec3 glitchColor = texColor * (1.0 + 0.5 * sin(time_f * 10.0 + randomSeed * 6.28));
    color = mix(vec4(texColor, 1.0), vec4(glitchColor, 1.0), 0.5);
}

```

Shader #885 - rgbchecker

File: rgbchecker.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns. Applies matrix transformations.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec2 st = gl_FragCoord.xy;
    color[0] = cos(st[0]*alpha_b);
    color[1] = sin(st[1]*alpha_g);
    color[2] = (st[0]+st[1]*alpha_r);
    ivec3 int_color;

```



```

    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #975 - square grid

File: square_grid.glsl

Applies an iterative, time-animated effect that evolves the image over time.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
vec4 randomColor(ivec2 cell_coord) {
    int n = cell_coord.x + cell_coord.y * 10 + int(time_f);
    n = (n << 13) ^ n;
    n = (n * (n * n * 15731 + 789221) + 1376312589) & 0x7fffffff;
    return vec4(float((n & 0xFF0000) >> 16) / 255.0,
                float((n & 0x00FF00) >> 8) / 255.0,
                float(n & 0x0000FF) / 255.0,
                1.0);
}
void main(void) {
    int grid_size = 10;
    float cell_size = 1.0 / float(grid_size);
    ivec2 cell_coord = ivec2(floor(tc.x * float(grid_size)), floor(tc.y * float(grid_size)));
    vec2 cell_center = (vec2(cell_coord) + 0.5) * cell_size;
    vec2 offset = tc - cell_center;
    float border = 0.05 * cell_size;
    if (abs(offset.x) < border || abs(offset.y) < border) {
        color = vec4(0, 0, 0, 1);
    } else {
        vec4 rand_color = randomColor(cell_coord);
        color = xor_RGB(texture(samp, tc), rand_color);
    }
}

```

Shader #976 - square grid2

File: square_grid2.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time

```
#version 330
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
vec4 xor_RGB(vec4 icolor, vec4 source) {
    ivec3 int_color;
    ivec4 isource = ivec4(source * 255);
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * icolor[i]);
        int_color[i] = int_color[i]^isource[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        icolor[i] = float(int_color[i])/255;
    }
    icolor.a = 1.0;
    return icolor;
}
vec4 randomColor(ivec2 cell_coord) {
    int n = cell_coord.x + cell_coord.y * 10 + int(time_f);
    n = (n << 13) ^ n;
    n = (n * (n * n * 15731 + 789221) + 1376312589) & 0x7fffffff;
    return vec4(float((n & 0xFF0000) >> 16) / 255.0,
                float((n & 0x00FF00) >> 8) / 255.0,
                float(n & 0x0000FF) / 255.0,
                1.0);
}
void main(void) {
    int grid_size = 10;
    float cell_size = 1.0 / float(grid_size);
    ivec2 cell_coord = ivec2(floor(tc.x * float(grid_size)), floor(tc.y * float(grid_size)));
    vec2 cell_center = (vec2(cell_coord) + 0.5) * cell_size;
    vec2 offset = tc - cell_center;
    float border = 0.05 * cell_size;
    if (abs(offset.x) < border || abs(offset.y) < border) {
        color = vec4(0, 0, 0, 1);
    } else {
        vec4 rand_color = sin(randomColor(cell_coord) * time_f);
        color = xor_RGB(texture(samp, tc), rand_color);
    }
}
```

Acid & Psychedelic (9 shaders)

Shader #8 - acid color2

File: acid_color2.glsl

Applies bitwise XOR operations on pixel color channels, creating digital-art style interference patterns.

Technique: Bitwise XOR on integer color channels - Driven by: time

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform float alpha_value;
uniform sampler2D samp;
uniform float time_f;
void main(void)
{
    color = texture(samp, tc);
    ivec3 source;
    for(int i = 0; i < 3; ++i) {
        source[i] = int(255 * color[i]);
    }
    vec2 cord1 = vec2(tc[0]/2, tc[1]/2);
    vec2 cord2 = vec2(tc[0]/4, tc[1]/4);
    vec2 cord3 = vec2(tc[0]/8, tc[1]/8);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    color[0] = color[0] * col1[0];
    color[1] = color[1] * col2[1];
    color[2] = color[2] * col3[2];
    ivec3 int_color;
    for(int i = 0; i < 3; ++i) {
        int_color[i] = int(255 * color[i]);
        int_color[i] = int_color[i]^source[i];
        if(int_color[i] > 255)
            int_color[i] = int_color[i]%255;
        color[i] = float(int_color[i])/255;
    }
}

```

Shader #9 - acidcam

File: acidcam.glsl

Processes the input texture to produce the "acidcam" visual effect.

Technique: Direct texture color manipulation with arithmetic operations

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
void main(void) {
    color = texture(samp, tc);
}

```

Shader #10 - acidcolor

File: acidcolor.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;

```

```

out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec2 cord1 = vec2(tc[0]/2, tc[1]/2);
    vec2 cord2 = vec2(tc[0]/4, tc[1]/4);
    vec2 cord3 = vec2(tc[0]/8, tc[1]/8);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    color[0] = color[0]+col1[0];
    color[1] = color[1]+col2[1];
    color[2] = color[2]+col3[2];
}

```

Shader #11 - acidcolorrand

File: acidcolorrand.glsl

Samples the texture at multiple coordinates and composites the results. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
in vec4 random_value;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
int tosmall(float f) {
    if(f >= 0 && f <= 75)
        return 0;
    if(f >= 76 && f <= 140)
        return 1;
}

```

```

    if(f >= 141 && f <= 255)
        return 2;
}
void main(void)
{
    color = texture(samp, tc);
    vec2 cord1 = vec2(tc[0]/2, tc[1]/2);
    vec2 cord2 = vec2(tc[0]/4, tc[1]/4);
    vec2 cord3 = vec2(tc[0]/8, tc[1]/8);
    vec4 col1 = texture(samp, cord1);
    vec4 col2 = texture(samp, cord2);
    vec4 col3 = texture(samp, cord3);
    color[0] = color[0]+col1[tosmall(random_value[0])];
    color[1] = color[1]+col2[tosmall(random_value[1])];
    color[2] = color[2]+col3[tosmall(random_value[2])];
}

```

Shader #242 - dream tunnel

File: dream-tunnel.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: RGB <-> HSV color space conversion - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
void main(void){
    vec2 uvScreen = gl_FragCoord.xy / iResolution.xy;
    float aspect = iResolution.x / iResolution.y;
    vec2 center = (iMouse.z > 0.5 || iMouse.w > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5, 0.5);
    vec2 p = uvScreen - center;
    p.x *= aspect;
    float r = length(p);
    float audio = clamp(uamp * 1.2 + amp * 0.5, 0.0, 2.0);
    float t = time_f;

```

```

float swirlPhase = t * (0.4 + 0.2 * audio) + r * 7.0;
float cs = cos(swirlPhase);
float sn = sin(swirlPhase);
vec2 q = vec2(cs * p.x - sn * p.y, sn * p.x + cs * p.y);
float zoomWave = 0.45 + 0.35 * sin(t * 0.35 + audio * 1.3);
float tunnelDepth = 0.25 + zoomWave + r * 2.2;
vec2 tunnelUV = center + q / tunnelDepth;
tunnelUV.x /= aspect;
tunnelUV = clamp(tunnelUV, vec2(0.001), vec2(0.999));
vec4 texTunnel = texture(samp, tunnelUV);
vec4 texBase = texture(samp, tc);
float ring = 0.5 + 0.5 * sin(r * 40.0 - t * (5.0 + audio * 6.0));
float hue = fract(t * 0.03 + r * 1.2);
float sat = 0.85;
float val = 0.55 + 0.35 * ring;
vec3 neon = hsv2rgb(vec3(hue, sat, val));
float tunnelMix = smoothstep(0.0, 0.7, r) * 0.9 + 0.1;
vec3 baseMixed = mix(texBase.rgb, texTunnel.rgb, tunnelMix);
float audioGlow = clamp(audio * 0.7, 0.0, 0.9);
vec3 withNeon = mix(baseMixed, neon, ring * (0.2 + 0.6 * audioGlow));
float vignette = 1.0 - 0.55 * r * r;
vignette = clamp(vignette, 0.35, 1.0);
vec3 finalCol = withNeon * vignette;
finalCol = clamp(finalCol, 0.0, 1.0);
color = vec4(finalCol, 1.0);
}

```

Shader #243 - dream tunnel2

File: dream-tunnel2.glsl

Applies a vignette effect, darkening the image edges while keeping the center bright. Responds to mouse position and audio input (amplitude/volume) for interactive control.

Technique: RGB <-> HSV color space conversion - Hermite smoothstep interpolation - Linear mix/lerp blending - Radial vignette falloff - Driven by: time, mouse, audio, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
uniform vec4 iMouse;
uniform float amp;
uniform float uamp;
uniform float iTime;
uniform int iFrame;
uniform float iTimeDelta;
uniform vec4 iDate;
uniform vec2 iMouseClicked;
uniform float iFrameRate;
uniform vec3 iChannelResolution[4];
uniform float iChannelTime[4];
uniform float iSampleRate;
const float PI = 3.1415926535897932384626433832795;
vec3 hsv2rgb(vec3 c){
    vec4 K = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

```

```

}
void main(void){
    vec2 uvScreen = gl_FragCoord.xy / iResolution.xy;
    float aspect = iResolution.x / iResolution.y;
    vec2 center = (iMouse.z > 0.5 || iMouse.w > 0.5) ? (iMouse.xy / iResolution) : vec2(0.5, 0.5);
    vec2 p = uvScreen - center;
    p.x *= aspect;
    float r = length(p);
    float audio = clamp(uamp * 1.6 + amp * 0.8, 0.0, 3.0);
    float t = time_f;
    float swirlPhase = t * (0.5 + 0.45 * audio) + r * (7.0 + 4.0 * audio);
    float cs = cos(swirlPhase);
    float sn = sin(swirlPhase);
    vec2 q = vec2(cs * p.x - sn * p.y, sn * p.x + cs * p.y);
    float zoomWave = 0.35 + 0.55 * sin(t * 0.4 + audio * 0.7);
    float tunnelDepth = 0.18 + zoomWave + r * (1.6 - 0.4 * audio);
    tunnelDepth = max(tunnelDepth, 0.12);
    vec2 tunnelUV = center + q / tunnelDepth;
    tunnelUV.x /= aspect;
    tunnelUV = clamp(tunnelUV, vec2(0.001), vec2(0.999));
    vec4 texTunnel = texture(samp, tunnelUV);
    vec4 texBase = texture(samp, tc);
    float ring = 0.5 + 0.5 * sin(r * (40.0 + 20.0 * audio) - t * (5.0 + audio * 8.0));
    float hue = fract(t * 0.04 + r * (1.2 + 0.8 * audio));
    float sat = 0.9;
    float val = 0.55 + 0.35 * ring;
    vec3 neon = hsv2rgb(vec3(hue, sat, val));
    float tunnelMix = smoothstep(0.0, 0.8, r * (1.3 + 0.5 * audio));
    tunnelMix = clamp(tunnelMix * (0.7 + 0.5 * audio), 0.0, 1.0);
    vec3 baseMixed = mix(texBase.rgb, texTunnel.rgb, tunnelMix);
    float audioGlow = clamp(audio * 0.75, 0.0, 1.2);
    vec3 withNeon = mix(baseMixed, neon, ring * (0.25 + 0.6 * audioGlow));
    float vigStrength = 0.55 + 0.25 * audioGlow;
    float vignette = 1.0 - vigStrength * r * r;
    vignette = clamp(vignette, 0.3, 1.0);
    vec3 finalCol = withNeon * vignette;
    finalCol = clamp(finalCol, 0.0, 1.0);
    color = vec4(finalCol, 1.0);
}

```

Shader #530 - gpt trip

File: gpt_trip.glsl

Performs edge detection on the input image, highlighting boundaries and contours. Computes luminance for brightness-based effects.

Technique: Sobel/Laplacian convolution kernel for edge detection - Pseudo-random noise via $\text{fract}(\sin(\text{dot}()))$ hash function - Polar coordinate transformation ($\text{atan2} + \text{length}$) - RGB <-> HSV color space conversion - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float lum(vec3 c) {
    return dot(c, vec3(0.2126, 0.7152, 0.0722));
}

```

```

}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    float a = hash(i);
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(a, b, u.x) + (c - a) * u.y * (1.0 - u.x) + (d - b) * u.x * u.y;
}
vec3 rgb2hsv(vec3 c) {
    vec4 K = vec4(0.0, -0.3333333, 0.6666667, -1.0);
    vec4 p = mix(vec4(c.bg, K.wz), vec4(c.gb, K.xy), step(c.b, c.g));
    vec4 q = mix(vec4(p.xyw, c.r), vec4(c.r, p.yzx), step(p.x, c.r));
    float d = q.x - min(q.w, q.y);
    float e = 0.00001;
    return vec3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e), q.x);
}
vec3 hsv2rgb(vec3 c) {
    vec3 p = abs(fract(c.xxx + vec3(0.0, 0.6666667, 0.3333333)) * 6.0 - 3.0);
    vec3 rgb = clamp(p - 1.0, 0.0, 1.0);
    return c.z * mix(vec3(1.0), rgb, c.y);
}
vec2 warp(vec2 uv) {
    vec2 p = uv * 2.0 - 1.0;
    float r = length(p);
    float a = atan(p.y, p.x);
    float n1 = noise(uv * 5.0 + vec2(time_f * 0.08, -time_f * 0.06));
    float n2 = noise(uv * 9.0 + vec2(-time_f * 0.05, time_f * 0.07));
    a += 0.04 * sin(r * 8.0 - time_f * 0.5) + 0.03 * (n1 - 0.5);
    r += 0.012 * sin(a * 6.0 + time_f * 0.4) + 0.01 * (n2 - 0.5);
    p = vec2(cos(a), sin(a)) * r;
    uv = p * 0.5 + 0.5;
    uv += 0.004 * vec2(
        sin(uv.y * 22.0 + time_f * 0.7),
        cos(uv.x * 18.0 - time_f * 0.6)
    );
    return uv;
}
float edgeSobel(vec2 uv, vec2 px) {
    float tl = lum(texture(samp, uv + px * vec2(-1.0, -1.0)).rgb);
    float tc0 = lum(texture(samp, uv + px * vec2( 0.0, -1.0)).rgb);
    float tr = lum(texture(samp, uv + px * vec2( 1.0, -1.0)).rgb);
    float ml = lum(texture(samp, uv + px * vec2(-1.0,  0.0)).rgb);
    float mr = lum(texture(samp, uv + px * vec2( 1.0,  0.0)).rgb);
    float bl = lum(texture(samp, uv + px * vec2(-1.0,  1.0)).rgb);
    float bc = lum(texture(samp, uv + px * vec2( 0.0,  1.0)).rgb);
    float br = lum(texture(samp, uv + px * vec2( 1.0,  1.0)).rgb);
    float gx = -tl - 2.0 * ml - bl + tr + 2.0 * mr + br;
    float gy = -tl - 2.0 * tc0 - tr + bl + 2.0 * bc + br;
    return sqrt(gx * gx + gy * gy);
}
void main() {
    vec2 uv = tc;
    vec2 px = 1.0 / iResolution;
    vec2 wuv = warp(uv);
    vec3 src = texture(samp, wuv).rgb;
    vec3 s1 = texture(samp, wuv + px * vec2(1.0, 0.0)).rgb;

```



```

vec3 s2 = texture(samp, wuv + px * vec2(-1.0, 0.0)).rgb;
vec3 s3 = texture(samp, wuv + px * vec2(0.0, 1.0)).rgb;
vec3 s4 = texture(samp, wuv + px * vec2(0.0, -1.0)).rgb;
vec3 soft = (src + s1 + s2 + s3 + s4) / 5.0;
float edge = edgeSobel(wuv, px);
edge = clamp(edge * 1.6, 0.0, 1.0);
float detail = length(src - soft);
detail = clamp(detail * 3.0, 0.0, 1.0);
vec3 hsv = rgb2hsv(src);
hsv.x = fract(hsv.x + 0.08 * sin(lum(src) * 8.0 + time_f * 0.25) + 0.12 * edge);
hsv.y = clamp(hsv.y * 1.4 + 0.2 * detail + 0.2 * edge, 0.0, 1.0);
hsv.z = clamp(hsv.z * 1.05 + 0.12 * detail, 0.0, 1.0);
vec3 neonBase = hsv2rgb(hsv);
vec3 edgeTint1 = vec3(1.0, 0.1, 0.9);
vec3 edgeTint2 = vec3(0.1, 1.0, 0.5);
vec3 edgeTint3 = vec3(0.2, 0.6, 1.0);
vec3 contour = vec3(0.0);
contour += edgeTint1 * smoothstep(0.10, 0.35, edge);
contour += edgeTint2 * smoothstep(0.25, 0.60, edge);
contour += edgeTint3 * smoothstep(0.45, 0.90, edge);
float rings = 0.5 + 0.5 * sin((lum(src) + detail * 0.7) * 20.0 + time_f * 0.2);
vec3 ringTint = mix(vec3(0.8, 0.1, 1.0), vec3(0.1, 0.9, 1.0), rings);
vec3 finalColor = mix(src, neonBase, 0.45);
finalColor += contour * 0.35;
finalColor += ringTint * detail * 0.12;
finalColor = mix(finalColor, src, smoothstep(0.75, 1.0, lum(src)) * 0.45);
finalColor = clamp(finalColor, 0.0, 1.0);
finalColor = pow(finalColor, vec3(0.95));
color = vec4(finalColor, 1.0);
}

```

Shader #531 - gpt trip2

File: gpt_trip2.glsl

Performs edge detection on the input image, highlighting boundaries and contours. Computes luminance for brightness-based effects.

Technique: Sobel/Laplacian convolution kernel for edge detection - Pseudo-random noise via fract(sin(dot())) hash function - Polar coordinate transformation (atan2 + length) - RGB <-> HSV color space conversion - Color quantization via floor rounding - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
in vec2 tc;
out vec4 color;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float lum(vec3 c) {
    return dot(c, vec3(0.2126, 0.7152, 0.0722));
}
float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7)))) * 43758.5453123;
}
float noise(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    float a = hash(i);
    float b = hash(i + vec2(1.0, 0.0));

```

```

    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(a, b, u.x) + (c - a) * u.y * (1.0 - u.x) + (d - b) * u.x * u.y;
}

float fbm(vec2 p) {
    float v = 0.0;
    float a = 0.5;
    for (int i = 0; i < 6; ++i) {
        v += noise(p) * a;
        p *= 2.0;
        a *= 0.5;
    }
    return v;
}

vec3 rgb2hsv(vec3 c) {
    vec4 K = vec4(0.0, -0.3333333, 0.6666667, -1.0);
    vec4 p = mix(vec4(c.bg, K.wz), vec4(c.gb, K.xy), step(c.b, c.g));
    vec4 q = mix(vec4(p.xyw, c.r), vec4(c.r, p.yzx), step(p.x, c.r));
    float d = q.x - min(q.w, q.y);
    float e = 0.00001;
    return vec3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e), q.x);
}

vec3 hsv2rgb(vec3 c) {
    vec3 p = abs(fract(c.xxx + vec3(0.0, 0.6666667, 0.3333333)) * 6.0 - 3.0);
    vec3 rgb = clamp(p - 1.0, 0.0, 1.0);
    return c.z * mix(vec3(1.0), rgb, c.y);
}

vec2 warp(vec2 uv) {
    vec2 p = uv * 2.0 - 1.0;
    float r = length(p);
    float a = atan(p.y, p.x);
    float n1 = fbm(uv * 3.2 + vec2(time_f * 0.05, -time_f * 0.04));
    float n2 = fbm(uv * 6.4 + vec2(-time_f * 0.03, time_f * 0.06));
    a += 0.06 * sin(r * 9.0 - time_f * 0.25) + 0.08 * (n1 - 0.5);
    r += 0.018 * sin(a * 7.0 + time_f * 0.18) + 0.015 * (n2 - 0.5);
    p = vec2(cos(a), sin(a)) * r;
    uv = p * 0.5 + 0.5;
    uv += 0.006 * vec2(
        sin(uv.y * 24.0 + time_f * 0.30),
        cos(uv.x * 20.0 - time_f * 0.28)
    );
    return uv;
}

float sobel(vec2 uv, vec2 px) {
    float tl = lum(texture(samp, uv + px * vec2(-1.0, -1.0)).rgb);
    float tc0 = lum(texture(samp, uv + px * vec2( 0.0, -1.0)).rgb);
    float tr = lum(texture(samp, uv + px * vec2( 1.0, -1.0)).rgb);
    float ml = lum(texture(samp, uv + px * vec2(-1.0,  0.0)).rgb);
    float mr = lum(texture(samp, uv + px * vec2( 1.0,  0.0)).rgb);
    float bl = lum(texture(samp, uv + px * vec2(-1.0,  1.0)).rgb);
    float bc = lum(texture(samp, uv + px * vec2( 0.0,  1.0)).rgb);
    float br = lum(texture(samp, uv + px * vec2( 1.0,  1.0)).rgb);
    float gx = -tl - 2.0 * ml - bl + tr + 2.0 * mr + br;
    float gy = -tl - 2.0 * tc0 - tr + bl + 2.0 * bc + br;
    return sqrt(gx * gx + gy * gy);
}

vec3 pastelBase(float t) {
    vec3 a = vec3(0.84, 0.82, 0.86);
    vec3 b = vec3(0.78, 0.92, 0.74);
    vec3 c = vec3(0.46, 0.86, 0.94);
    vec3 d = vec3(0.96, 0.86, 0.38);
}

```

```

    vec3 e = vec3(0.92, 0.28, 0.86);
    if (t < 0.25) return mix(a, b, t / 0.25);
    if (t < 0.50) return mix(b, c, (t - 0.25) / 0.25);
    if (t < 0.75) return mix(c, d, (t - 0.50) / 0.25);
    return mix(d, e, (t - 0.75) / 0.25);
}

void main() {
    vec2 uv = tc;
    vec2 px = 1.0 / iResolution;
    vec2 wuv = warp(uv);
    vec3 src = texture(samp, wuv).rgb;
    vec3 sx1 = texture(samp, wuv + px * vec2( 1.0,  0.0)).rgb;
    vec3 sx2 = texture(samp, wuv + px * vec2(-1.0,  0.0)).rgb;
    vec3 sy1 = texture(samp, wuv + px * vec2( 0.0,  1.0)).rgb;
    vec3 sy2 = texture(samp, wuv + px * vec2( 0.0, -1.0)).rgb;
    vec3 soft = (src + sx1 + sx2 + sy1 + sy2) / 5.0;
    float e1 = sobel(wuv, px * 1.0);
    float e2 = sobel(wuv, px * 2.0);
    float edge = clamp(e1 * 1.1 + e2 * 0.55, 0.0, 1.0);
    float detail = clamp(length(src - soft) * 4.5, 0.0, 1.0);
    float l = lum(src);
    float n1 = fbm(wuv * 7.0 + edge * 3.0);
    float n2 = fbm(wuv * 18.0 + vec2(time_f * 0.04, -time_f * 0.03));
    float n3 = fbm(wuv.yx * 28.0 + vec2(-time_f * 0.02, time_f * 0.05));
    float field = l * 1.15 + edge * 1.35 + detail * 0.65 + n1 * 0.55;
    float q = floor(field * 9.0) / 9.0;
    vec3 base = pastelBase(clamp(q, 0.0, 1.0));
    vec3 cyan = vec3(0.05, 0.76, 0.92);
    vec3 aqua = vec3(0.20, 0.92, 0.72);
    vec3 lime = vec3(0.55, 0.92, 0.18);
    vec3 yellow = vec3(0.98, 0.86, 0.12);
    vec3 magenta = vec3(0.88, 0.12, 0.82);
    vec3 violet = vec3(0.48, 0.28, 0.92);
    vec3 grayLav = vec3(0.80, 0.78, 0.82);
    float flow1 = 0.5 + 0.5 * sin((wuv.x * 12.0 + wuv.y * 16.0) + n1 * 8.0 + field * 7.0);
    float flow2 = 0.5 + 0.5 * sin((wuv.x * 20.0 - wuv.y * 13.0) + n2 * 10.0 + field * 11.0 + 1.4);
    float flow3 = 0.5 + 0.5 * sin((wuv.y * 24.0 + wuv.x * 9.0) + n3 * 12.0 + field * 15.0 + 2.1);
    vec3 river = mix(cyan, aqua, flow1);
    river = mix(river, lime, flow2 * 0.65);
    river = mix(river, yellow, smoothstep(0.58, 0.95, flow3) * 0.45);
    vec3 contour = vec3(0.0);
    contour += magenta * (smoothstep(0.08, 0.25, edge) - smoothstep(0.25, 0.45, edge)) * 0.90;
    contour += violet * (smoothstep(0.28, 0.50, edge) - smoothstep(0.50, 0.72, edge)) * 0.70;
    contour += cyan * smoothstep(0.50, 0.95, edge) * 0.45;
    float cells = 0.5 + 0.5 * sin((field + n2 * 0.7) * 42.0);
    float rings = smoothstep(0.72, 0.98, cells) - smoothstep(0.98, 1.0, cells);
    vec3 ringColor = mix(magenta, lime, 0.5 + 0.5 * sin(field * 5.0 + n3 * 3.0));
    ringColor = mix(ringColor, yellow, smoothstep(0.60, 1.00, detail) * 0.35);
    vec3 hsvShift = hsv2rgb(vec3(
        fract(rgb2hsv(src).x + 0.12 * edge + 0.06 * flow2),
        clamp(rgb2hsv(src).y * 1.25 + 0.20 * detail, 0.0, 1.0),
        clamp(rgb2hsv(src).z * 1.05 + 0.08 * edge, 0.0, 1.0)
    ));
    vec3 finalColor = mix(grayLav, base, 0.72);
    finalColor = mix(finalColor, hsvShift, 0.20);
    finalColor = mix(finalColor, river, 0.36 * smoothstep(0.10, 0.80, detail + edge * 0.3));
    finalColor += contour;
    finalColor += ringColor * rings * 0.65;
    float subjectMask = smoothstep(0.06, 0.38, edge + detail * 0.55);
    finalColor = mix(finalColor, mix(river, magenta, 0.18), subjectMask * 0.22);
    float speck = smoothstep(0.78, 0.98, fbm(wuv * 60.0 + edge * 8.0));
    finalColor += magenta * speck * 0.12;
}

```

```

    finalColor -= vec3(0.08, 0.05, 0.10) * speck * 0.10;
    finalColor = mix(finalColor, grayLav, smoothstep(0.72, 1.0, 1) * 0.25);
    finalColor = clamp(finalColor, 0.0, 1.0);
    finalColor = pow(finalColor, vec3(0.96));
    color = vec4(finalColor, 1.0);
}

```

Shader #792 - psyche wave

File: psyche_wave.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Technique: Driven by: time, resolution

```

#version 330
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
void main(void)
{
    vec2 normCoord = ((gl_FragCoord.xy / iResolution.xy) * 2.0 - 1.0) * vec2(iResolution.x / iResolution
    ↪ .y, 1.0);
    float distanceFromCenter = length(normCoord);
    float wave = sin(distanceFromCenter * 12.0 - time_f * 4.0);
    vec2 tcAdjusted = tc + (normCoord * 0.301 * wave);
    vec4 textureColor = texture(samp, tcAdjusted);
    color = textureColor;
}

```

Blend & Composite (7 shaders)

Shader #12 - addup blend

File: addup_blend.glsl

Applies time-animated sinusoidal transformations to the image for a flowing, organic look.

Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;

```

```

uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    vec4 source = color;
    float val = (color[0]+color[1]+color[2])*alpha;
    for(int q = 0; q < 3; ++q)
        color[q] = color[q] * (sin(val * color[q]/0.3));
    color = (0.5 * color) + (0.5 * source);
}

```

Shader #148 - composite

File: composite.glsl

Generates procedural noise patterns combined with sinusoidal distortion.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Scanline overlay using modular y-coordinate - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
vec3 compositeEffect(vec2 uv) {
    float offset = 0.01;
    vec3 col;
    col.r = texture(samp, uv + vec2(offset, 0.0)).r;
    col.g = texture(samp, uv).g;
    col.b = texture(samp, uv - vec2(offset, 0.0)).b;
    float noise = fract(sin(dot(uv.xy, vec2(12.9898, 78.233))) * 43758.5453);
    col += noise * 0.05;
    float scanline = sin(uv.y * iResolution.y * 1.5) * 0.1;
    col -= scanline;
    float bleed = sin(uv.y * iResolution.y * 0.2 + time_f * 5.0) * 0.005;
    col.r += bleed * 0.002;
    col.b -= bleed * 0.002;
    return col;
}
void main(void) {
    vec2 uv = tc;
    vec3 col = compositeEffect(uv);
    color = vec4(col, 1.0);
}

```

Shader #149 - composite2

File: composite2.glsl

Generates procedural noise patterns combined with sinusoidal distortion. Computes luminance for brightness-based effects.

Technique: Pseudo-random noise via fract(sin(dot())) hash function - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float hash(vec2 p){ return fract(sin(dot(p, vec2(127.1,311.7)))) * 43758.5453; }
vec3 compositeEffect(vec2 uv) {
    vec2 c = vec2(0.5);
    vec2 d = uv - c;
    float r2 = dot(d,d);
    vec2 uvd = c + d * (1.0 + 0.15*r2 + 0.15*r2*r2);
    float wob = (hash(vec2(time_f*0.73, uv.y*13.7)) - 0.5) * 0.003;
    uvd.x += wob;
    uvd = clamp(uvd, 0.001, 0.999);
    float ca = 0.002 + 0.002*sin(time_f*0.37);
    vec2 shift = d * ca;
    float bleed = sin(uvd.y * iResolution.y * 0.2 + time_f * 5.0) * 0.003;
    float off = 0.01;
    vec3 col;
    col.r = texture(samp, uvd + vec2(off + bleed, 0.0) + shift).r;
    col.g = texture(samp, uvd).g;
    col.b = texture(samp, uvd - vec2(off + bleed, 0.0) - shift).b;
    float grain = hash(uvd * iResolution.xy + time_f*41.0) * 2.0 - 1.0;
    col += grain * 0.03;
    float scan = sin(uvd.y * iResolution.y * 3.14159) * 0.06;
    col -= scan;
    float l = dot(col, vec3(0.2126,0.7152,0.0722));
    col = mix(vec3(l), col, 0.92);
    float vig = 1.0 - smoothstep(0.25, 0.8, length(d));
    col *= vig;
    col = pow(max(col, 0.0), vec3(1.0/1.7));
    return clamp(col, 0.0, 1.0);
}
void main(void) {
    vec3 col = compositeEffect(tc);
    color = vec4(col, 1.0);
}

```

Shader #150 - composite3

File: composite3.glsl

Applies a blur/smoothing convolution to the input image using multi-sample averaging. Computes luminance for brightness-based effects.

Technique: Multi-tap convolution blur (21 texture samples) - Pseudo-random noise via fract(sin(dot())) hash function - Color quantization via floor rounding - Scanline overlay using modular y-coordinate - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float PI(){return 3.14159265358979323846;}
float hash(vec2 p){
    p=fract(p*vec2(123.34,345.45));

```

```

    p+=dot(p,p+34.345);
    return fract(p.x*p.y);
}
vec3 triadMask(vec2 uv){
    float px=floor(uv.x*iResolution.x);
    float m=mod(px,3.0);
    vec3 a=vec3(1.10,0.88,0.88);
    vec3 b=vec3(0.88,1.10,0.88);
    vec3 c=vec3(0.88,0.88,1.10);
    return mix(mix(c,b,step(1.0,m)),a,step(2.0,m));
}
float scanline(vec2 uv){
    float y=uv.y*iResolution.y;
    return 0.84+0.16*sin(PI()*y);
}
vec3 halation(vec2 uv){
    vec2 px=1.0/iResolution;
    vec3 s=vec3(0.0);
    s+=texture(samp,uv+vec2( px.x, 0.0)).rgb*0.10;
    s+=texture(samp,uv+vec2(-px.x, 0.0)).rgb*0.10;
    s+=texture(samp,uv+vec2( 0.0, px.y)).rgb*0.10;
    s+=texture(samp,uv+vec2( 0.0,-px.y)).rgb*0.10;
    s+=texture(samp,uv+vec2( px.x, px.y)).rgb*0.06;
    s+=texture(samp,uv+vec2(-px.x, px.y)).rgb*0.06;
    s+=texture(samp,uv+vec2( px.x,-px.y)).rgb*0.06;
    s+=texture(samp,uv+vec2(-px.x,-px.y)).rgb*0.06;
    return s;
}
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
vec3 rgb2yiq(vec3 c){
    float Y=dot(c,vec3(0.299,0.587,0.114));
    float I=dot(c,vec3(0.596,-0.274,-0.322));
    float Q=dot(c,vec3(0.211,-0.523,0.312));
    return vec3(Y,I,Q);
}
vec3 yiq2rgb(vec3 yIQ){
    float Y=yIQ.x, I=yIQ.y, Q=yIQ.z;
    return mat3(1.0, 1.0, 1.0,
                0.956,-0.272,-1.106,
                0.621,-0.647, 1.703) * vec3(Y,I,Q);
}
vec3 compositeEffect(vec2 uv) {
    float offset = 0.01;
    vec3 col;
    col.r = texture(samp, uv + vec2(offset, 0.0)).r;
    col.g = texture(samp, uv).g;
    col.b = texture(samp, uv - vec2(offset, 0.0)).b;
    float noise = fract(sin(dot(uv.xy, vec2(12.9898, 78.233)))) * 43758.5453;
    col += noise * 0.05;
    float scanline = sin(uv.y * iResolution.y * 1.5) * 0.1;
    col -= scanline;
    float bleed = sin(uv.y * iResolution.y * 0.2 + time_f * 5.0) * 0.005;
    col.r += bleed * 0.002;
    col.b -= bleed * 0.002;
    return col;
}
void main(){
    vec2 uv=tc;
    vec2 px=1.0/iResolution;

```

```

float hs=(sin(time_f*120.0)+sin((uv.y+time_f*0.61)*87.0))*0.0006;
uv.x+=hs;
float jitter=(hash(vec2(floor(gl_FragCoord.y*0.5), time_f))*2.0-1.0)*0.00045;
uv.x+=jitter;
vec3 s0=texture(samp,uv).rgb;
vec3 sL1=texture(samp,uv-vec2(px.x,0)).rgb;
vec3 sR1=texture(samp,uv+vec2(px.x,0)).rgb;
vec3 sL2=texture(samp,uv-vec2(px.x*2.0,0)).rgb;
vec3 sR2=texture(samp,uv+vec2(px.x*2.0,0)).rgb;
vec3 y0=rgb2yiq(s0);
vec3 yL1=rgb2yiq(sL1);
vec3 yR1=rgb2yiq(sR1);
vec3 yL2=rgb2yiq(sL2);
vec3 yR2=rgb2yiq(sR2);
float Y = 0.40*y0.x + 0.24*(yL1.x+yR1.x) + 0.06*(yL2.x+yR2.x);
float Ih = 0.40*y0.y + 0.24*(yL1.y+yR1.y) + 0.06*(yL2.y+yR2.y);
float Qh = 0.36*y0.z + 0.12*(yL1.z+yR1.z) + 0.02*(yL2.z+yR2.z);
float sub=227.5;
float line=floor(gl_FragCoord.y);
float phase=6.2831853*(gl_FragCoord.x*sub/iResolution.x + mod(line,2.0)*0.5 + time_f*0.03);
mat2 R=mat2(cos(phase),-sin(phase),sin(phase),cos(phase));
vec2 IQm = R*vec2(Ih,Qh);
float Yhp = Y - 0.5*(yL1.x+yR1.x);
Yhp = sin(Yhp * PI());
vec2 cc = vec2(cos(phase),sin(phase))*Yhp*0.18;
IQm += cc;
vec2 IQd = R*vec2(
    0.50*y0.y + 0.30*yL1.y + 0.20*yR1.y,
    0.55*y0.z + 0.28*yL1.z + 0.17*yR1.z
);
float tHop = 0.5;
vec2 IQdelay = vec2(
    texture(samp, uv-vec2(px.x*tHop,0)).rg
);
vec3 yDelay = rgb2yiq(texture(samp, uv-vec2(px.x*tHop,0)).rgb);
IQd = mix(IQd, vec2(yDelay.y,yDelay.z), 0.35);
vec2 IQ = mix(IQm, IQd, 0.5);
IQ *= 0.92;
vec3 ntsc = yiq2rgb(vec3(Y, IQ.x, IQ.y));
vec3 bleedH = (texture(samp,uv+vec2(px.x*1.2,0)).rgb + texture(samp,uv-vec2(px.x*1.2,0)).rgb)*0.25;
vec3 ghost = texture(samp, uv-vec2(px.x*2.2,0)).rgb*0.10;
vec3 base = mix(ntsc, ntsc*0.6 + bleedH*0.4 + ghost, 0.45);
float sl=scanline(uv);
vec3 tri=triadMask(uv);
float grille=0.95+0.05*sin(uv.x*iResolution.x*PI()*0.5);
float flick=0.992+0.008*sin(time_f*360.0);
vec3 bloom=halation(uv)*0.28;
float n = (hash(gl_FragCoord.xy+time_f*vec2(173.1,91.7))*2.0-1.0)*0.018;
base += n;
vec3 linear=pow(clamp(base,0.0,1.0),vec3(2.2))+pow(bloom,vec3(2.2))*0.55;
vec3 shaped=linear*tri*sl*grille*flick;
vec3 outc=pow(shaped,vec3(1.0/2.2));
float black=0.02;
float white=1.05;
outc=clamp((outc-black)/(white-black),0.0,1.0);
color = mix(vec4(compositeEffect(uv), 1.0), vec4(outc, 1.0), 0.5);
}

```


Shader #151 - composite crt

File: composite_crt.glsl

Overlays CRT-style scanlines onto the image for a retro display effect.

Technique: Scanline overlay using modular y-coordinate - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float PI(){return 3.14159265358979323846;}
vec3 triadMask(vec2 uv){
    float px=floor(uv.x*iResolution.x);
    float m=mod(px,3.0);
    vec3 a=vec3(1.10,0.88,0.88);
    vec3 b=vec3(0.88,1.10,0.88);
    vec3 c=vec3(0.88,0.88,1.10);
    return mix(mix(c,b,step(1.0,m)),a,step(2.0,m));
}
float scanline(vec2 uv){
    float y=uv.y*iResolution.y;
    return 0.84+0.16*sin(PI()*y);
}
vec3 halation(vec2 uv){
    vec2 px=1.0/iResolution;
    vec3 s=vec3(0.0);
    s+=texture(samp,uv+vec2( px.x, 0.0)).rgb*0.10;
    s+=texture(samp,uv+vec2(-px.x, 0.0)).rgb*0.10;
    s+=texture(samp,uv+vec2( 0.0, px.y)).rgb*0.10;
    s+=texture(samp,uv+vec2( 0.0,-px.y)).rgb*0.10;
    s+=texture(samp,uv+vec2( px.x, px.y)).rgb*0.06;
    s+=texture(samp,uv+vec2(-px.x, px.y)).rgb*0.06;
    s+=texture(samp,uv+vec2( px.x,-px.y)).rgb*0.06;
    s+=texture(samp,uv+vec2(-px.x,-px.y)).rgb*0.06;
    return s;
}
void main(){
    vec2 uv=tc;
    float jitter=(sin(time_f*120.0)+sin((uv.y+time_f*0.61)*87.0))*0.0005;
    uv.x+=jitter;
    vec2 center=vec2(0.5);
    vec2 dir=uv-center;
    float aberr=0.0016+0.0010*sin(time_f*0.8);
    float r=texture(samp, uv+dir*aberr).r;
    float g=texture(samp, uv).g;
    float b=texture(samp, uv-dir*aberr).b;
    vec3 base=vec3(r,g,b);
    vec3 bloom=halation(uv)*0.35;
    vec3 linear=pow(base,vec3(2.2))+pow(bloom,vec3(2.2));
    float sl=scanline(uv);
    vec3 tri=triadMask(uv);
    float grille=0.94+0.06*sin(uv.x*iResolution.x*PI()*0.5);
    float flick=0.992+0.008*sin(time_f*360.0);
    vec3 shaped=linear*tri*sl*grille*flick;
    vec3 outc=pow(shaped,vec3(1.0/2.2));
    color=vec4(outc,1.0);
}
```

Shader #168 - crystalblend2

File: crystalblend2.glsl

Reduces the color depth by quantizing pixel values, creating a posterized look. Responds to audio input (amplitude/volume) for interactive control.

Technique: Polar coordinate transformation (atan2 + length) - Color quantization via floor rounding - Hermite smoothstep interpolation - Linear mix/lerp blending - Driven by: time, audio, resolution

```
#version 330 core
in vec2 tc;
out vec4 color;
uniform float time_f;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float amp;
uniform float uamp;
uniform int blendMode;
uniform float blendAmt;
float PI = 3.141592653589793;
float mirror1(float x, float m){
    float p = mod(x, m*2.0);
    return p > m ? (2.0*m - p) : p;
}
vec2 kaleido(vec2 p, float seg){
    vec2 c = vec2(0.5);
    vec2 d = p - c;
    float a = atan(d.y, d.x);
    float r = length(d);
    float m = PI / max(2.0, seg);
    a = mirror1(a, m);
    return c + vec2(cos(a), sin(a)) * max(r, 1e-6);
}
uvec3 to8(vec3 c){
    return uvec3(clamp(round(c*255.0), 0.0, 255.0));
}
vec3 from8(uvec3 x){
    return vec3(x) / 255.0;
}
vec3 xorBlend(vec3 a, vec3 b){
    uvec3 ai = to8(a);
    uvec3 bi = to8(b);
    return from8(ai ^ bi);
}
vec3 median3(vec3 a, vec3 b, vec3 c){
    vec3 mn = min(a, min(b, c));
    vec3 mx = max(a, max(b, c));
    return a + b + c - mn - mx;
}
void main(){
    vec2 uv = tc;
    float drive = clamp(amp + uamp, 0.0, 2.0);
    float seg = floor(mix(5.0, 12.0, clamp(drive*0.5, 0.0, 1.0)));
    vec2 k0 = kaleido(sin(uv*time_f), seg);
    float r0 = length(k0 - 0.5);
    vec2 dir0 = normalize(k0 - 0.5 + vec2(1e-6, 0.0));
    float bend0 = (0.05 + 0.12*uamp) / (r0 + 0.04);
    vec2 wob0 = 0.010 * (0.3 + amp) * vec2(sin(30.0*r0 - 1.6*time_f), cos(26.0*r0 - 1.2*time_f));
    vec2 wuv0 = k0 + dir0 * bend0 + wob0;
```

```

    vec2 k1 = kaleido(uv + 0.003*vec2(sin(time_f*0.7), cos(time_f*0.6)), seg);
    float r1 = length(k1 - 0.5);
    vec2 dir1 = normalize(k1 - 0.5 + vec2(1e-6, 0.0));
    float bend1 = (0.04 + 0.10*uamp) / (r1 + 0.05);
    vec2 wob1 = 0.009 * (0.3 + amp) * vec2(cos(27.0*r1 - 1.3*time_f), sin(25.0*r1 - 1.1*time_f));
    vec2 wuv1 = k1 + dir1 * bend1 + wob1;
    vec3 a = texture(samp, uv).rgb;
    vec3 b = texture(samp, wuv0).rgb;
    vec3 c3 = texture(samp, wuv1).rgb;
    vec3 blended = (blendMode == 1) ? xorBlend(a, b) : median3(a, b, c3);
    float s = clamp(mix(0.35, 0.85, blendAmt) * (0.6 + 0.4*clamp(drive*0.5, 0.0, 1.0)), 0.0, 1.0);
    float edge = smoothstep(0.0, 0.9, length(uv-0.5));
    float adaptive = mix(s*0.7, s, edge);
    vec3 outc = mix(a, blended, adaptive);
    color = vec4(outc, 1.0);
}

```

Shader #918 - self alphablend scale

File: self_alphablend_scale.glsl

Applies an iterative, time-animated effect that evolves the image over time. Applies matrix transformations.

Technique: Driven by: time

```

#version 330
in vec2 tc;
out vec4 color;
uniform float alpha_r;
uniform float alpha_g;
uniform float alpha_b;
uniform float alpha;
uniform vec4 optx;
uniform vec4 random_var;
uniform float alpha_value;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
uniform float value_alpha_r, value_alpha_g, value_alpha_b;
uniform float index_value;
uniform float time_f;
uniform float restore_black;
void main(void)
{
    color = texture(samp, tc);
    color = color*alpha;
    ivec4 colori = ivec4(color * 255);
    for(int i = 0; i < 3; ++i) {
        if(colori[i] > 255)
            colori[i] = colori[i]%255;
        color[i] = float(colori[i])/255;
    }
}

```

Edge & Outline (7 shaders)

Shader #311 - edge black

File: edge_black.glsl

Performs edge detection on the input image, highlighting boundaries and contours.

Technique: Sobel/Laplacian convolution kernel for edge detection - Driven by: resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
vec3 edgeDetection(sampler2D tex, vec2 uv) {
    vec2 offset = vec2(1.0 / iResolution.x, 1.0 / iResolution.y);
    float gx =
        -1.0 * texture(tex, uv + vec2(-offset.x, -offset.y)).r
        + -2.0 * texture(tex, uv + vec2(-offset.x, 0.0)).r
        + -1.0 * texture(tex, uv + vec2(-offset.x, offset.y)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, -offset.y)).r
        + 2.0 * texture(tex, uv + vec2(offset.x, 0.0)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, offset.y)).r;
    float gy =
        -1.0 * texture(tex, uv + vec2(-offset.x, -offset.y)).r
        + -2.0 * texture(tex, uv + vec2(0.0, -offset.y)).r
        + -1.0 * texture(tex, uv + vec2(offset.x, -offset.y)).r
        + 1.0 * texture(tex, uv + vec2(-offset.x, offset.y)).r
        + 2.0 * texture(tex, uv + vec2(0.0, offset.y)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, offset.y)).r;
    float edge = sqrt(gx * gx + gy * gy);
    float threshold = 0.3;
    vec3 color = (edge > threshold) ? vec3(1.0) : vec3(0.0);
    return color;
}
void main(void) {
    vec3 edgeColor = edgeDetection(samp, tc);
    color = vec4(edgeColor, 1.0);
}
```

Shader #312 - edge face

File: edge_face.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - 2D rotation matrix - Linear mix/lerp blending - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
uniform float uDistortion = 0.5;
uniform float uRotateSpeed = 1.0;
uniform float uWarpSpeed = 0.1;
uniform float uRandRate = 0.2;
```

```

uniform float uPhaseRate = 0.1;
float pingPong(float x, float len) {
    float m = mod(x, len * 2.0);
    return abs(m - len);
}
float rand(float n) {
    return fract(sin(n) * 43758.5453123);
}
void main(void) {
    vec2 center = vec2(0.5);
    vec2 off = tc - center;
    float maxR = length(vec2(0.5));
    float r = length(off);
    float nr = r / maxR;
    float ang = atan(off.y, off.x);
    float dr = nr + uDistortion * (nr * nr);
    dr = clamp(dr, 0.0, 1.0) * maxR;
    vec2 d = center + dr * vec2(cos(ang), sin(ang));
    vec2 dp = d - center;
    float tr = time_f * uRandRate;
    float tp = time_f * uPhaseRate;
    float nR = floor(tr);
    float nP = floor(tp);
    float ar = fract(tr);
    float ap = fract(tp);
    float dir0 = sign(rand(nR) - 0.5);
    float dir1 = sign(rand(nR + 1.0) - 0.5);
    float dir = mix(dir0, dir1, ar*ar*(3.0-2.0*ar));
    float ph0 = rand(nP) * 6.2831853;
    float ph1 = rand(nP + 1.0) * 6.2831853;
    float phase = mix(ph0, ph1, ap*ap*(3.0-2.0*ap));
    float rotA = dir * (phase + time_f * uRotateSpeed);
    mat2 rot = mat2(cos(rotA), -sin(rotA), sin(rotA), cos(rotA));
    vec2 rotp = rot * dp + center;
    float t = time_f * uWarpSpeed;
    vec2 w = vec2(pingPong(rotp.x + t, 1.0), pingPong(rotp.y + t, 1.0));
    color = texture(samp, w);
}

```

Shader #313 - edge fill

File: edge_fill.glsl

Performs edge detection on the input image, highlighting boundaries and contours.

Technique: Sobel/Laplacian convolution kernel for edge detection - Linear mix/lerp blending -
Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 edgeDetection(sampler2D tex, vec2 uv) {
    vec2 offset = vec2(1.0 / iResolution.x, 1.0 / iResolution.y);
    float gx =
        -1.0 * texture(tex, uv + vec2(-offset.x, -offset.y)).r
        + -2.0 * texture(tex, uv + vec2(-offset.x, 0.0)).r
        + -1.0 * texture(tex, uv + vec2(-offset.x, offset.y)).r

```

```

        + 1.0 * texture(tex, uv + vec2(offset.x, -offset.y)).r
        + 2.0 * texture(tex, uv + vec2(offset.x, 0.0)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, offset.y)).r;
float gy =
    -1.0 * texture(tex, uv + vec2(-offset.x, -offset.y)).r
    + -2.0 * texture(tex, uv + vec2(0.0, -offset.y)).r
    + -1.0 * texture(tex, uv + vec2(offset.x, -offset.y)).r
    + 1.0 * texture(tex, uv + vec2(-offset.x, offset.y)).r
    + 2.0 * texture(tex, uv + vec2(0.0, offset.y)).r
    + 1.0 * texture(tex, uv + vec2(offset.x, offset.y)).r;
float edge = sqrt(gx * gx + gy * gy);
float threshold = 0.3;
vec3 edgeColor = (edge > threshold) ? vec3(1.0) : vec3(0.0);
return edgeColor;
}
vec3 rainbowBackground(vec2 uv, float time_f) {
    float hue = mod(time_f + length(uv), 1.0);
    float r = abs(hue * 6.0 - 3.0) - 1.0;
    float g = 2.0 - abs(hue * 6.0 - 2.0);
    float b = 2.0 - abs(hue * 6.0 - 4.0);
    return clamp(vec3(r, g, b), 0.0, 1.0);
}
void main(void) {
    vec3 edgeColor = edgeDetection(samp, tc);
    vec3 bgColor = rainbowBackground(tc, time_f);
    vec3 finalColor = mix(bgColor, vec3(1.0), step(0.5, edgeColor.r));
    color = vec4(finalColor, 1.0);
}

```

Shader #314 - edge pencil

File: edge_pencil.glsl

Performs edge detection on the input image, highlighting boundaries and contours.

Technique: Sobel/Laplacian convolution kernel for edge detection - Pseudo-random noise via fract(sin(dot())) hash function - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float random(vec2 st) {
    return fract(sin(dot(st.xy, vec2(12.9898, 78.233))) * 43758.5453123);
}
vec3 pencilSketch(sampler2D tex, vec2 uv) {
    float edgeThreshold = 0.3;
    vec2 offset = vec2(1.0 / iResolution.x, 1.0 / iResolution.y);
    float gx =
        -1.0 * texture(tex, uv + vec2(-offset.x, -offset.y)).r
        + -2.0 * texture(tex, uv + vec2(-offset.x, 0.0)).r
        + -1.0 * texture(tex, uv + vec2(-offset.x, offset.y)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, -offset.y)).r
        + 2.0 * texture(tex, uv + vec2(offset.x, 0.0)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, offset.y)).r;
    float gy =
        -1.0 * texture(tex, uv + vec2(-offset.x, -offset.y)).r
        + -2.0 * texture(tex, uv + vec2(0.0, -offset.y)).r

```

```

        + -1.0 * texture(tex, uv + vec2(offset.x, -offset.y)).r
        + 1.0 * texture(tex, uv + vec2(-offset.x, offset.y)).r
        + 2.0 * texture(tex, uv + vec2(0.0, offset.y)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, offset.y)).r;
float edge = sqrt(gx * gx + gy * gy);
float noise = random(uv * time_f) * 0.1;
float sketch = (edge > edgeThreshold) ? 0.0 : 1.0;
sketch += noise;
vec3 color = vec3(sketch);
return color;
}
void main(void) {
    vec3 sketchColor = pencilSketch(samp, tc);
    color = vec4(sketchColor, 1.0);
}

```

Shader #315 - edge rainbow

File: edge_rainbow.glsl

Performs edge detection on the input image, highlighting boundaries and contours.

Technique: Sobel/Laplacian convolution kernel for edge detection - Driven by: time, resolution

```

#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform vec2 iResolution;
uniform float time_f;
vec3 edgeDetection(sampler2D tex, vec2 uv) {
    vec2 offset = vec2(1.0 / iResolution.x, 1.0 / iResolution.y);
    float gx =
        -1.0 * texture(tex, uv + vec2(-offset.x, -offset.y)).r
        + -2.0 * texture(tex, uv + vec2(-offset.x, 0.0)).r
        + -1.0 * texture(tex, uv + vec2(-offset.x, offset.y)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, -offset.y)).r
        + 2.0 * texture(tex, uv + vec2(offset.x, 0.0)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, offset.y)).r;
    float gy =
        -1.0 * texture(tex, uv + vec2(-offset.x, -offset.y)).r
        + -2.0 * texture(tex, uv + vec2(0.0, -offset.y)).r
        + -1.0 * texture(tex, uv + vec2(offset.x, -offset.y)).r
        + 1.0 * texture(tex, uv + vec2(-offset.x, offset.y)).r
        + 2.0 * texture(tex, uv + vec2(0.0, offset.y)).r
        + 1.0 * texture(tex, uv + vec2(offset.x, offset.y)).r;
    float edge = sqrt(gx * gx + gy * gy);
    float threshold = 0.3;
    vec3 edgeColor = vec3(0.0);
    if (edge > threshold) {
        float hue = mod(time_f + length(uv), 1.0);
        float r = abs(hue * 6.0 - 3.0) - 1.0;
        float g = 2.0 - abs(hue * 6.0 - 2.0);
        float b = 2.0 - abs(hue * 6.0 - 4.0);
        edgeColor = clamp(vec3(r, g, b), 0.0, 1.0);
    }
    return edgeColor;
}
void main(void) {
    vec3 edgeColor = edgeDetection(samp, tc);
}

```

```
    color = vec4(edgeColor, 1.0);  
}
```

Shader #781 - prism glass wall noedge

File: prism_glass_wall_noedge.glsl

Simulates chromatic aberration by offsetting color channels, mimicking lens imperfections.
High-fidelity spectral helper for the rainbow smear Dividing the screen into 3 prism regions
without visual borders Line from (0,0) to (0.5, 1)

Technique: Per-channel UV offset for chromatic aberration - Linear mix/lerp blending - Driven
by: time, resolution

```
#version 330 core  
out vec4 color;  
in vec2 tc;  
uniform sampler2D samp;  
uniform vec2 iResolution;  
uniform float time_f;  
vec3 spectrum(float w) {  
    vec3 c;  
    if (w < 0.2) c = mix(vec3(1.0, 0.0, 1.0), vec3(0.0, 0.0, 1.0), w * 5.0);  
    else if (w < 0.4) c = mix(vec3(0.0, 0.0, 1.0), vec3(0.0, 1.0, 0.0), (w - 0.2) * 5.0);  
    else if (w < 0.6) c = mix(vec3(0.0, 1.0, 0.0), vec3(1.0, 1.0, 0.0), (w - 0.4) * 5.0);  
    else if (w < 0.8) c = mix(vec3(1.0, 1.0, 0.0), vec3(1.0, 0.0, 0.0), (w - 0.6) * 5.0);  
    else c = vec3(1.0, 0.0, 0.0);  
    return c;  
}  
void main(void) {  
    vec2 uv = tc;  
    vec2 refractDir;  
    float edge1 = 2.0 * uv.x;  
    float edge2 = -2.0 * uv.x + 2.0;  
    if (uv.y > edge1) {  
        refractDir = normalize(uv - vec2(0.0, 1.0));  
    } else if (uv.y > edge2) {  
        refractDir = normalize(uv - vec2(1.0, 1.0));  
    } else {  
        refractDir = normalize(uv - vec2(0.5, 0.0));  
    }  
    vec3 accumulatedColor = vec3(0.0);  
    float dispersion = 0.08;  
    int samples = 20;  
    for (int i = 0; i < samples; i++) {  
        float w = float(i) / float(samples - 1);  
        float shift = (w - 0.5) * dispersion;  
        vec2 sampleUV = uv + (refractDir * shift);  
        accumulatedColor += texture(samp, clamp(sampleUV, 0.0, 1.0)).rgb * spectrum(w);  
    }  
    vec3 finalRGB = accumulatedColor / (float(samples) * 0.48);  
    finalRGB += vec3(0.015, 0.02, 0.03);  
    color = vec4(finalRGB, 1.0);  
}
```


Shader #963 - spiral edge

File: spiral_edge.glsl

Transforms the image into polar coordinates and applies wave-based distortion for a swirling effect.

Technique: Polar coordinate transformation (atan2 + length) - Driven by: time, resolution

```
#version 330 core
out vec4 color;
in vec2 tc;
uniform sampler2D samp;
uniform float time_f;
uniform vec2 iResolution;
float pingPong(float x, float length) {
    float m = mod(x, length * 2.0);
    return m <= length ? m : length * 2.0 - m;
}
void main(void) {
    float t = time_f;
    vec2 p = tc * 2.0 - 1.0;
    float k = 10.0;
    float a = 0.35 + 0.25 * sin(t * 1.7);
    p += a * vec2(sin((p.y + t) * k), cos((p.x - t) * k));
    float r = length(p);
    float ang = atan(p.y, p.x) + a * 4.0 * r * r + t * 1.2;
    vec2 s = vec2(cos(ang), sin(ang)) * r;
    vec2 uv = s * 0.5 + 0.5;
    uv.x = pingPong(uv.x + t * 1.8, 1.0);
    uv.y = pingPong(uv.y + t * 1.8, 1.0);
    color = texture(samp, uv);
}
```