GRASS

Coding your own video effects in Sandin FX — a tutorial of the language

The GRASS module of Sandin FX pays tribute to Tom DeFanti's **GRAphics Symbiosis System** (1974), the language that let artists animate images in real time before sending them into the Sandin Image Processor. Same gesture here: you write a few lines of code, they are compiled on the fly, and the image they produce then travels through every module of the application.

1. Where it happens

In Sandin FX, click the **GRASS** button in the transport bar: an editor window opens. There you write the **body of an effect function**, then **Compile and apply**: the effect is active immediately in the preview and in the export, and it is saved (reloaded at the next launch). The **Examples** menu provides a dozen ready-to-tweak effects.

Your effect sits **at the head of the chain**: what it returns then flows through the Adder/Multiplier, the Comparator, the Colorizer, the feedback… In the sidebar, the **GRASS — coded effect** module (green) sets its **opacity** (mix with the video), its **blend mode** (Normal, Add, Multiply, Screen) and the two **knobs p1 / p2** exposed to your code.

2. The language

The code is **Metal Shading Language** (very close to C). Key point: your code runs **once per pixel**, for every frame. You do not draw “onto” the image: you answer the question “which colour for this pixel?” — and you always finish with `return` of a `float3` (red, green, blue, each from 0.0 to 1.0).

The variables at your disposal

Variable	Type	Role
`col`	`float3`	The colour of the current video pixel (after chroma separation). The simplest starting point.
`uv`	`float2`	The pixel's position in the image: `uv.x` from 0 (left) to 1 (right), `uv.y` from 0 (top) to 1 (bottom). The centre is `(0.5, 0.5)`.
`t`	`float`	Time in seconds. This is what animates everything.
`p1`, `p2`	`float`	The two knobs of the GRASS module (0 to 1) — your live controls, no recompiling needed. MIDI-mappable.
`video`, `smp`	texture	The whole source video: `video.sample(smp, coord).rgb` reads the colour at any coordinate `coord` (a `float2` in 0–1).

Types and how to write them

```
float  a = 0.5;                       // a number
float2 p = float2(0.3, 0.7);          // a 2D point / vector (x, y)
float3 c = float3(1.0, 0.5, 0.0);     // a colour (r, g, b) — orange here
float3 grey = float3(0.5);            // shorthand: (0.5, 0.5, 0.5)

// "swizzle": accessing the components
float red = c.r;                      // also .g .b, or .x .y for a float2
float3 reversed = c.bgr;              // reorders the channels in one go
```

Pitfall #1: always write numbers with a decimal — `2.0`, not `2`. Metal is strict: `uv * 2` may refuse to compile where `uv * 2.0` goes through. **Pitfall #2:** every statement ends with a semicolon. **Pitfall #3:** you must always reach a `return`.

Vector operations

Operators work component by component: `col * 0.5` darkens, `1.0 - col` inverts, `col + float3(0.1, 0.0, 0.0)` reddens. This is what keeps the code so short.

The toolbox (the functions you use all the time)

Function	What it does
`sin(x)`, `cos(x)`	Oscillation — the basis of all animation. `sin(t)` swings between −1 and 1; `0.5 + 0.5*sin(t)` brings it back to 0–1.
`fract(x)`, `floor(x)`	Fractional / integer part. `fract` loops (ramps, scrolling), `floor` quantises (blocks, steps).
`mix(a, b, k)`	Blend from `a` to `b` by `k` (0–1). The #1 tool for dosing an effect.
`clamp(x, min, max)`	Bounds a value — handy for keeping colours in 0–1.
`step(threshold, x)`, `smoothstep(a, b, x)`	Hard / soft threshold — for Comparator-style cut-outs.
`length(v)`, `distance(a, b)`	Length of a vector, distance between two points — circles, halos, tunnels.
`atan2(y, x)`	The angle of a point around the centre — spirals, kaleidoscopes.
`dot(col, float3(0.299, 0.587, 0.114))`	The **luminance** of a colour — to think in terms of “light/dark”.
`abs(x)`, `min`, `max`, `pow`, `fmod`	The classics: absolute value, bounds, power, modulo.

3. First steps — five effects from scratch

a. Touching the colours

```
return 1.0 - col;                    // negative
```

```
float l = dot(col, float3(0.299, 0.587, 0.114));   // luminance
return float3(l * 1.1, l * 0.8, l * 0.5);          // sepia
```

b. Animating with time

```
// pulse: the image breathes at the rate set by p1
float pulse = 0.75 + 0.25 * sin(t * (1.0 + p1 * 8.0));
return col * pulse;
```

c. Displacing where the video is read (the great classic)

Instead of reading the video “here” (`uv`), read it **somewhere else**: every distortion comes from this.

```
// horizontal waves: each line is shifted according to its height
float2 d = float2(sin(uv.y * 30.0 + t * 2.0) * 0.02 * p1, 0.0);
return video.sample(smp, uv + d).rgb;
```

d. Cutting up space

```
// split-screen: left half normal, right half negative
if (uv.x > 0.5) { return 1.0 - col; }
return col;
```

e. Generating without the video (pure GRASS)

```
// animated concentric rings — ignore col, create the image
float r = distance(uv, float2(0.5, 0.5));
float v = 0.5 + 0.5 * sin(r * 40.0 - t * 3.0);
return float3(v * 0.9, v * 0.4, 1.0 - v);
```

A pure generator covers the video: lower the GRASS module's **Opacity** or switch its **Blend** to Add/Screen to overlay it — then let the Colorizer and the feedback do the rest.

4. Making good use of p1 and p2

Write your interesting constants as functions of `p1` / `p2` so you can play the effect live:

```
float speed    = 1.0 + p1 * 8.0;          // p1 at 0 → slow, at 1 → fast
float sectors  = 2.0 + floor(p2 * 10.0);  // p2 quantised: 2 to 12 sectors
```

Tip: `floor(p * n)` turns a continuous knob into a notched selector.

5. Reading compile errors

When something fails, the compiler message appears in red below the editor. Look for `error:` followed by a description and a snippet. The most common ones:

Message (excerpt)	Likely cause
`expected ';'`	Semicolon forgotten on the previous line.
`cannot convert ... 'int'`	Number written without a decimal (`2` instead of `2.0`).
`use of undeclared identifier`	Misspelled name (`uv` , `col` , `video` …) or undeclared variable.
`control reaches end of non-void function`	One path of the code never reaches a `return` .

6. Performance

The code runs for every pixel, 60 times per second — keep it light: avoid long loops (a few dozen iterations at most, like the Lissajous example and its 48 points) and large numbers of `sample` calls. Everything else — `sin` , `mix` , arithmetic — is almost free on the GPU.

7. The bundled examples

Example	What it teaches	p1 / p2
Negative	The simplest operation on `col`	—
Pixelation	Quantising `uv` with `floor`	pixel size
Ripples	Radial displacement of the video read	strength / frequency
Mirror kaleidoscope	Folding `uv` with `abs`	—
Angular kaleidoscope	Polar coordinates (`atan2` , `fmod`)	sectors / rotation
Block glitch	Deterministic randomness per cell, channel swap	intensity / size
Detuned TV	Scrolling with `fract` , events on time steps	speed / tearing
Plasma	Layered sines, signal modulation	frequency / intensity
Psychedelic tunnel	Pure polar generator	speed / branches
Glowing Lissajous	Loop, accumulation of halos	X / Y ratios
Julia (animated fractal)	Iterating z^2+c, smooth colouring	constant c (x / y)
Logarithmic spiral	Polar coordinates + log	arms / tightness

Recommended method: start from an example close to what you have in mind, change *one* constant, recompile, observe. That was the pedagogy of GRASS in 1974 — it still works.

8. Appendix — full code of the examples

The examples of the menu, as shipped (copy-paste them into the editor). Comments are translated here; the code is identical.

Negative

```
// Video negative
return 1.0 - col;
```

Pixelation

```
// Mosaic pixelation — p1: pixel size
float n = 8.0 + (1.0 - p1) * 120.0;
float2 q = (floor(uv * n) + 0.5) / n;
return video.sample(smp, q).rgb;
```

Ripples

```
// Concentric ripples distorting the video — p1: strength, p2: frequency
float2 d = uv - 0.5;
float r = length(d) + 0.0001;
float rip = sin(r * (20.0 + p2 * 70.0) - t * 4.0) * (0.002 + p1 * 0.03);
return video.sample(smp, uv + (d / r) * rip).rgb;
```

Mirror kaleidoscope

```
// Kaleidoscope: the image folded onto itself
float2 m = float2(abs(uv.x * 2.0 - 1.0), abs(uv.y * 2.0 - 1.0));
return video.sample(smp, m).rgb;
```

Angular kaleidoscope

```
// Angular kaleidoscope — p1: number of sectors, p2: rotation
float2 d = uv - 0.5;
float n = 2.0 + floor(p1 * 10.0);
float a = atan2(d.y, d.x) + p2 * 6.28318;
a = abs(fmod(a, 6.28318 / n) - 3.14159 / n);
float r = length(d);
return video.sample(smp, 0.5 + r * float2(cos(a), sin(a))).rgb;
```

Block glitch

```
// Block glitch, corrupted-signal style — p1: intensity, p2: block size
float2 cell = floor(uv * (4.0 + (1.0 - p2) * 28.0));
float h = fract(sin(dot(cell, float2(12.9898, 78.233)) + floor(t * 8.0)) * 43758.5453);
float2 off = (h < p1 * 0.5) ? float2(h - 0.5, fract(h * 7.0) - 0.5) * 0.2 : float2(0.0);
float3 c = video.sample(smp, uv + off).rgb;
if (h < p1 * 0.15) { c = c.bgr; }   // some blocks swap their channels
return c;
```

Detuned TV

```
// Detuned TV: vertical roll + tearing — p1: speed, p2: tearing
float roll = fract(uv.y + t * p1 * 1.5);
float tear = (fract(sin(floor(t * 6.0) * 7.13) * 43758.5453) - 0.5) * p2 * 0.25;
float x = uv.x + ((roll < 0.07) ? tear : 0.0);
float3 c = video.sample(smp, float2(x, roll)).rgb;
if (roll < 0.02) { c *= 0.3; }      // black sync bar
return c;
```

Plasma

```
// Plasma modulated by the signal — p1: frequency, p2: intensity
float f = 8.0 + p1 * 30.0;
float v = sin(uv.x * f + t) * sin(uv.y * f * 0.75 - t * 1.3)
        + sin((uv.x + uv.y) * f * 0.6 + t * 0.7);
return col * (1.0 - p2 * 0.5 + p2 * 0.4 * v);
```

Psychedelic tunnel

```
// Generated psychedelic tunnel (ignores the source, like pure GRASS)
// p1: speed — p2: number of branches
float2 d = uv - 0.5;
float a = atan2(d.y, d.x);
float r = length(d);
float v = sin(10.0 / (r + 0.1) - t * (1.0 + p1 * 6.0))
        * sin(a * floor(2.0 + p2 * 10.0) + t);
return float3(0.5 + 0.5 * v, 0.2 + 0.3 * v, 0.7 - 0.4 * v);
```

Julia (animated fractal)

```
// Animated Julia fractal (z^2 + c) — p1/p2: shift of the constant c
float2 z = (uv - 0.5) * float2(3.2, 2.4);
float2 c = float2(-0.8 + p1 * 0.6 + 0.04 * sin(t * 0.31),
                  0.156 + (p2 - 0.5) * 0.4 + 0.04 * cos(t * 0.23));
float m = 0.0;
for (int i = 0; i < 48; i++) {
    z = float2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + c;
    if (dot(z, z) > 4.0) { m = float(i) / 48.0; break; }
}
if (m == 0.0) { return float3(0.02); }
float k = fract(m * 3.0 + t * 0.05);
float3 ca = float3(0.85, 0.05, 0.75);
float3 cb = float3(1.0, 0.85, 0.15);
float3 cc = float3(0.88, 0.9, 0.93);
return (k < 0.5) ? mix(ca, cb, k * 2.0) : mix(cb, cc, k * 2.0 - 1.0);
```

Logarithmic spiral

```
// Glowing logarithmic spiral — p1: number of arms, p2: tightness
float2 d = uv - 0.5;
float r = length(d) + 0.0001;
float a = atan2(d.y, d.x);
float arms = 2.0 + floor(p1 * 8.0);
float v = sin(a * arms + log(r) * (4.0 + p2 * 14.0) - t * 2.0);
float m = smoothstep(0.0, 0.35, v) * smoothstep(1.0, 0.55, r * 2.0);
return col * 0.3 + float3(0.75, 0.7, 1.0) * m;
```

Glowing Lissajous

```
// Glowing Lissajous curve over the dimmed video — p1/p2: X/Y ratios
float g = 0.0;
for (int i = 0; i < 48; i++) {
    float ph = t * 0.6 + float(i) * 0.13;
    float2 pt = 0.5 + 0.38 * float2(sin(ph * (2.0 + floor(p1 * 4.0))),
                                    sin(ph * (3.0 + floor(p2 * 4.0)) + 1.57));
    float d2 = distance(uv, pt);
    g += 0.003 / (0.003 + d2 * d2 * 400.0);
}
return col * 0.25 + float3(0.45, 1.0, 0.55) * g;
```