GRASS

Coder ses propres effets vidéo dans Sandin FX — tutoriel du langage

Le module GRASS de Sandin FX rend hommage au **GRAphics Symbiosis System** de Tom DeFanti (1974), le langage qui permettait aux artistes d'animer l'image en temps réel avant de l'envoyer dans le Sandin Image Processor. Ici, même geste : vous écrivez quelques lignes de code, elles sont compilées à chaud, et l'image qu'elles produisent traverse ensuite tous les modules de l'application.

1. Où ça se passe

Dans Sandin FX, cliquez le bouton **GRASS** de la barre de transport : une fenêtre d'édition s'ouvre. Vous y écrivez le **corps d'une fonction d'effet**, puis **Compiler et appliquer** : l'effet est actif immédiatement dans l'aperçu, à l'export, et il est sauvegardé (rechargé au prochain lancement). Le menu **Exemples** fournit dix effets prêts à modifier.

Votre effet s'insère **en tête de la chaîne** : ce qu'il retourne passe ensuite dans l'Adder/Multiplier, le Comparator, le Colorizer, le feedback… Dans la barre latérale, le module **GRASS — effet codé** (vert) règle son **opacité** (mix avec la vidéo), son **mode de fusion** (Normal, Addition, Multiplication, Écran) et les deux **potentiomètres p1 / p2** exposés à votre code.

2. Le langage

Le code est du **Metal Shading Language** (très proche du C). Point essentiel : votre code s'exécute **une fois par pixel**, à chaque image. Vous ne dessinez pas « sur » l'image : vous répondez à la question « quelle couleur pour ce pixel ? » — et vous terminez toujours par `return` d'un `float3` (rouge, vert, bleu, chacun de 0.0 à 1.0).

Les variables à votre disposition

Variable	Type	Rôle
`col`	`float3`	La couleur du pixel courant de la vidéo (après séparation chroma). Le point de départ le plus simple.
`uv`	`float2`	La position du pixel dans l'image : `uv.x` de 0 (gauche) à 1 (droite), `uv.y` de 0 (haut) à 1 (bas). Le centre est `(0.5, 0.5)`.
`t`	`float`	Le temps en secondes. C'est lui qui anime tout.
`p1`, `p2`	`float`	Les deux potentiomètres du module GRASS (0 à 1) — vos réglages en direct, sans recompiler. Mappables en MIDI.
`video`, `smp`	texture	La vidéo source entière : `video.sample(smp, coord).rgb` lit la couleur à n'importe quelle coordonnée `coord` (un `float2` en 0–1).

Les types et leur écriture

```
float  a = 0.5;                      // un nombre
float2 p = float2(0.3, 0.7);         // un point / vecteur 2D (x, y)
float3 c = float3(1.0, 0.5, 0.0);    // une couleur (r, v, b) — ici orange
float3 gris = float3(0.5);           // raccourci : (0.5, 0.5, 0.5)

// « swizzle » : accéder aux composantes
float rouge = c.r;                   // aussi .g .b, ou .x .y pour un float2
float3 inverse = c.bgr;              // réordonne les canaux d'un coup
```

Piège n°1 : écrivez toujours les nombres avec une décimale — `2.0` et non `2`. Metal est strict : `uv * 2` peut refuser de compiler là où `uv * 2.0` passe. **Piège n°2 :** chaque instruction finit par un point-virgule. **Piège n°3 :** il faut toujours atteindre un `return`.

Les opérations vectorielles

Les opérateurs travaillent composante par composante : `col * 0.5` assombrit, `1.0 - col` inverse, `col + float3(0.1, 0.0, 0.0)` rougit. C'est ce qui rend le code très court.

La boîte à outils (les fonctions qui servent tout le temps)

Fonction	Ce qu'elle fait
`sin(x)`, `cos(x)`	L'oscillation — la base de toute animation. `sin(t)` ondule entre –1 et 1 ; `0.5 + 0.5*sin(t)` le ramène en 0–1.
`fract(x)`, `floor(x)`	Partie fractionnaire / entière. `fract` fait boucler (rampes, défilements), `floor` quantifie (blocs, paliers).
`mix(a, b, k)`	Fondu de `a` vers `b` selon `k` (0–1). L'outil n°1 pour doser un effet.
`clamp(x, min, max)`	Borne une valeur — utile pour garder les couleurs en 0–1.
`step(seuil, x)`, `smoothstep(a, b, x)`	Seuil net / seuil adouci — pour les découpes façon Comparator.
`length(v)`, `distance(a, b)`	Longueur d'un vecteur, distance entre deux points — cercles, halos, tunnels.
`atan2(y, x)`	L'angle d'un point autour du centre — spirales, kaléidoscopes.
`dot(col, float3(0.299, 0.587, 0.114))`	La **luminance** d'une couleur — pour raisonner en « clair/sombre ».
`abs(x)`, `min`, `max`, `pow`, `fmod`	Les classiques : valeur absolue, bornes, puissance, modulo.

3. Premiers pas — cinq effets en partant de zéro

a. Toucher aux couleurs

```
return 1.0 - col;                    // négatif
```

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

b. Animer avec le temps

```
// pulsation : l'image respire au rythme de p1
float pulse = 0.75 + 0.25 * sin(t * (1.0 + p1 * 8.0));
return col * pulse;
```

c. Déplacer la lecture de la vidéo (le grand classique)

Au lieu de lire la vidéo « ici » (`uv`), lisez-la **ailleurs** : toute distorsion vient de là.

```
// vagues horizontales : chaque ligne est décalée selon sa hauteur
float2 d = float2(sin(uv.y * 30.0 + t * 2.0) * 0.02 * p1, 0.0);
return video.sample(smp, uv + d).rgb;
```

d. Découper l'espace

```
// split-screen : moitié gauche normale, moitié droite négative
if (uv.x > 0.5) { return 1.0 - col; }
return col;
```

e. Générer sans la vidéo (le GRASS pur)

```
// anneaux concentriques animés — ignorez col, créez l'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);
```

> Un générateur pur recouvre la vidéo : baissez l'**Opacité** du module GRASS ou passez la **Fusion** en Addition/Écran pour le superposer — puis laissez le Colorizer et le feedback faire le reste.

4. Bien utiliser p1 et p2

Codez vos constantes intéressantes en fonction de `p1` / `p2` pour jouer l'effet en direct :

```
float vitesse   = 1.0 + p1 * 8.0;          // p1 à 0 → lent, à 1 → rapide
float secteurs  = 2.0 + floor(p2 * 10.0);  // p2 quantifié : 2 à 12 secteurs
```

Astuce : `floor(p * n)` transforme un potentiomètre continu en sélecteur à crans.

5. Lire les erreurs de compilation

En cas d'erreur, le message du compilateur s'affiche en rouge sous l'éditeur. Repérez `error:` suivi d'une description et d'un extrait. Les plus courantes :

Message (extrait)	Cause probable
`expected ';'`	Point-virgule oublié à la ligne précédente.
`cannot convert ... 'int'`	Nombre écrit sans décimale (`2` au lieu de `2.0`).
`use of undeclared identifier`	Nom mal orthographié (`uv` , `col` , `video` …) ou variable non déclarée.
`control reaches end of non-void function`	Un chemin du code n'atteint pas de `return` .

6. Performance

Le code tourne pour chaque pixel, 60 fois par seconde — restez léger : évitez les boucles longues (quelques dizaines d'itérations maximum, comme l'exemple Lissajous et ses 48 points) et les `sample` en grand nombre. Tout le reste — `sin` , `mix` , arithmétique — est quasi gratuit sur GPU.

7. Les exemples fournis

Exemple	Ce qu'il enseigne	p1 / p2
Négatif	L'opération la plus simple sur `col`	—
Pixellisation	Quantifier `uv` avec `floor`	taille des pixels
Ondulations	Déplacement radial de la lecture vidéo	force / fréquence
Kaléidoscope miroir	Replier `uv` avec `abs`	—
Kaléidoscope angulaire	Coordonnées polaires (`atan2` , `fmod`)	secteurs / rotation
Glitch en blocs	Hasard déterministe par cellule, échange de canaux	intensité / taille
TV déréglée	Défilement avec `fract` , événements par paliers de temps	vitesse / déchirure
Plasma	Superposition de sinus, modulation du signal	fréquence / intensité
Tunnel psychédélique	Générateur pur en polaire	vitesse / branches
Lissajous lumineux	Boucle, accumulation de halos	rapports X / Y
Julia (fractale animée)	Itération z^2+c, coloration lisse	constante c (x / y)
Spirale logarithmique	Coordonnées polaires + log	bras / resserrement

Méthode conseillée : partez d'un exemple proche de ce que vous imaginez, changez *une* constante, recompilez, observez. C'était la pédagogie de GRASS en 1974 — elle marche toujours.

8. Annexe — le code complet des exemples

Les dix exemples du menu, tels qu'ils sont livrés (copiables-collables dans l'éditeur).

Négatif

```
// Negatif video
return 1.0 - col;
```

Pixellisation

```
// Pixellisation mosaique — p1 : taille des pixels
float n = 8.0 + (1.0 - p1) * 120.0;
float2 q = (floor(uv * n) + 0.5) / n;
return video.sample(smp, q).rgb;
```

Ondulations

```
// Ondulations concentriques deformant la video — p1 : force, p2 : frequence
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;
```

Kaléidoscope miroir

```
// Kaleidoscope : l'image repliee sur elle-meme
float2 m = float2(abs(uv.x * 2.0 - 1.0), abs(uv.y * 2.0 - 1.0));
return video.sample(smp, m).rgb;
```

Kaléidoscope angulaire

```
// Kaleidoscope angulaire — p1 : nombre de secteurs, 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;
```

Glitch en blocs

```
// Glitch en blocs façon signal corrompu — p1 : intensite, p2 : taille des blocs
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; }   // certains blocs echangent leurs canaux
return c;
```

TV déréglée

```
// TV dereglee : defilement vertical + dechirure — p1 : vitesse, p2 : dechirure
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; }      // barre noire de synchro
return c;
```

Plasma

```
// Plasma module par le signal — p1 : frequence, p2 : intensite
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);
```

Tunnel psychédélique

```
// Tunnel psychedelique genere (ignore la source, comme un GRASS pur)
// p1 : vitesse — p2 : nombre de 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 (fractale animée)

```
// Fractale de Julia animee (z^2 + c) — p1/p2 : deplacement de la constante 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);
```

Spirale logarithmique

```
// Spirale logarithmique lumineuse — p1 : nombre de bras, p2 : resserrement
float2 d = uv - 0.5;
float r = length(d) + 0.0001;
float a = atan2(d.y, d.x);
float bras = 2.0 + floor(p1 * 8.0);
float v = sin(a * bras + 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;
```

Lissajous lumineux

```
// Courbe de Lissajous lumineuse sur la video attenuee — p1/p2 : rapports X/Y
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;
```