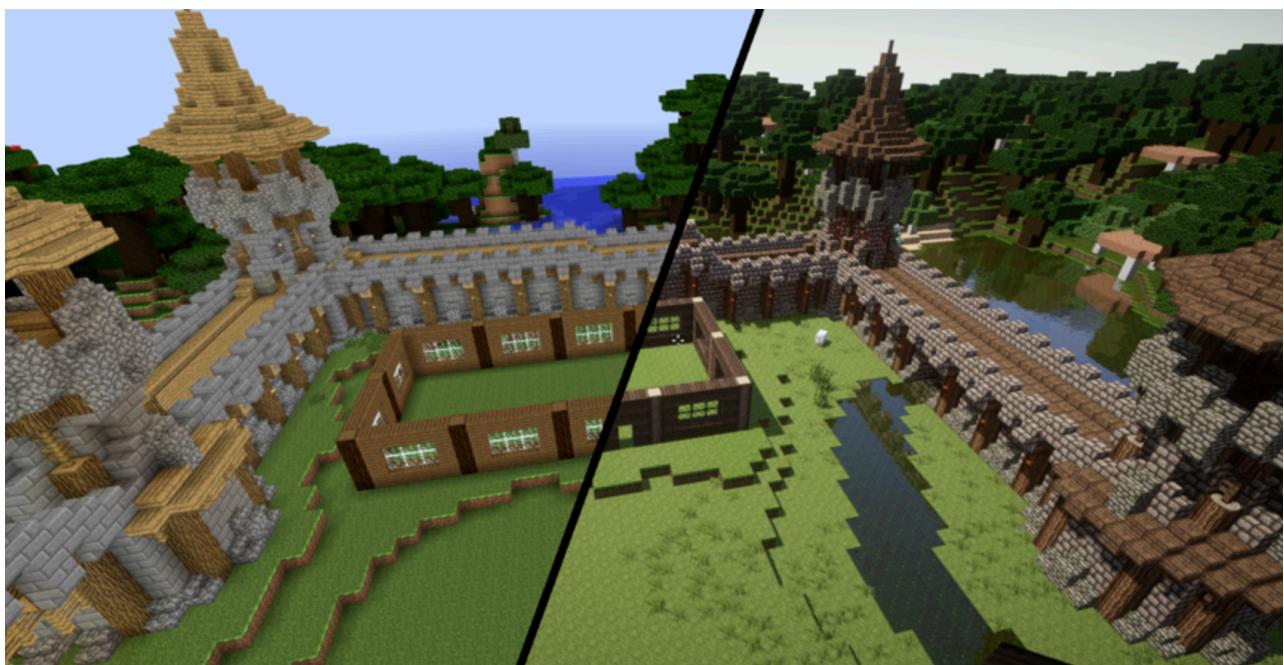


Les Shaders

Une introduction

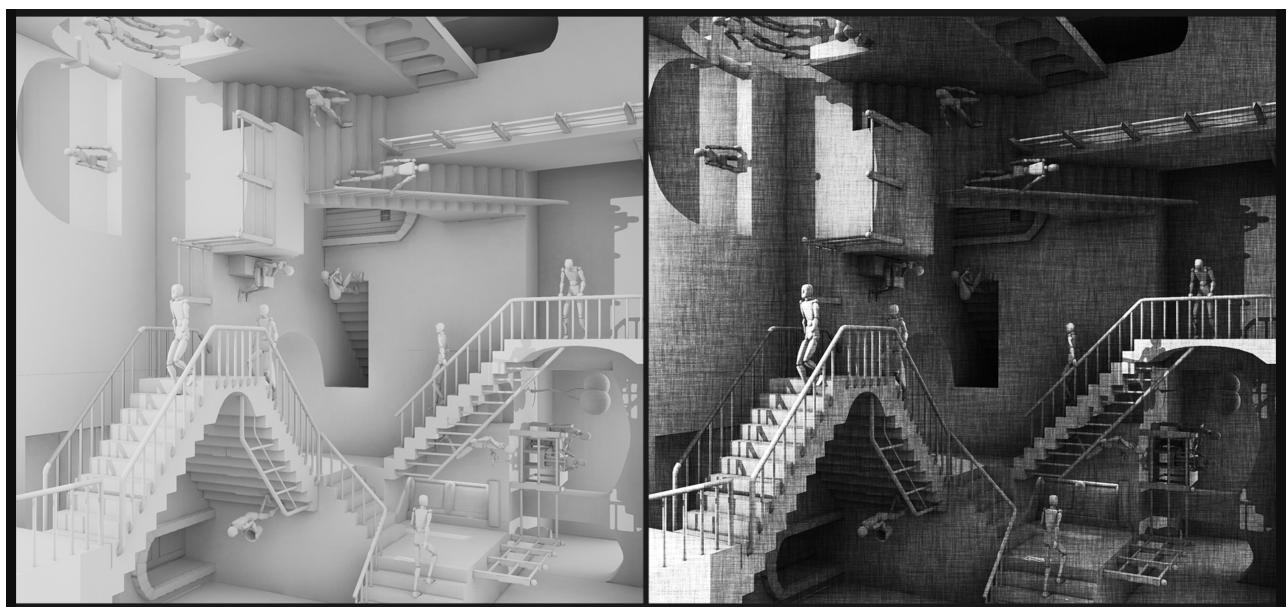
Rudi Giot - Mise à jour le 20 août 2017



**Attribution-NonCommercial-ShareAlike
4.0 International (CC BY-NC-SA 4.0)**

Avant propos

L'image ci-dessous a été créée à partir d'une scène 3D (à gauche) et ensuite passée dans un *Shader* qui l'a transformée (coloriée dans ce cas particulier). Le « *Shading* » est une technique qui consiste à travers un code informatique à réaliser un « rendu » en deux dimensions (souvent à partir d'une scène en trois dimensions). Généralement, ces programmes informatiques profitent des architectures massivement parallèles des processeurs des cartes graphiques (*GPU*) pour réaliser ces opérations très rapidement. Ils sont surtout utilisés dans les domaines du cinéma (effets spéciaux et films d'animation) et du jeu vidéo.



Exemple de Shader appliqué à une scène 3D

Ce cours d'introduction aux *Shaders* est construit à partir de plusieurs documents (livres, articles) mais aussi des sites Internet : <http://thebookofshaders.com/>, <https://gamedevelopment.tutsplus.com/tutorials/a-beginners-guide-to-coding-graphics-shaders--cms-23313>, ... Consultez la bibliographie pour plus de détails. Vous pouvez à tout moment vous référer à ces documents mais, en principe, toutes les informations nécessaires à la compréhension des notions théoriques et des exemples ainsi qu'à la réalisation des exercices sont disponibles dans le présent document.

Table des matières

1. Introduction	4
2. Fragment Shaders	5
2.1. Bac à sable	5
2.2. Couleurs	6
2.3. Step et SmoothStep	10
2.4. Tracé d'une droite	14
2.5. Fonctions	17
2.6. Translation	18
2.7. Mouvement	20
2.8. Rotation	22
2.9. Grille	23
2.10. Mixage de couches	25
2.11. Disque	26
2.12. Bitmap	27
2.13. Vidéo	29
Solution des exercices	30

1. Introduction

Il existe plusieurs façon d'aborder les *Shaders* et plusieurs façon de les programmer : HLSL, GLSL, CG, ... Il faut donc faire un choix. Même si les principes de base restent identiques, la manière de coder est un peu différentes. Le GLSL (openGL Shading Language) étant le système le plus ouvert (multiplateforme) et le plus répandu, c'est celui que nous décrirons dans la suite de ce cours.

The screenshot shows three code editors side-by-side. The first editor contains `RaycastTerrain.fx`, which includes HLSL code for a basic effect. The second editor contains `reflection_fragment.ghl`, which is a GLSL fragment shader for applying reflections. The third editor contains `curves.cg`, which includes various tessellation functions. The code is color-coded by language.

```
RaycastTerrain.fx
//----- File: BasicHLSL10.fx
//----- The effect file for the BasicHLSL sample.
//----- Copyright (c) Microsoft Corporation. All rights reserved.
//----- 

// Maximum number of binary searches to make
#define BINARY_STEPS 8

// Maximum number of steps to find any intersection for shadow
#define MAX_ANY_STEPS 256

// Maximum number of steps to make for relief mapping of detail
#define MAX_DETAIL_STEPS 128

// Maximum number of steps to take when cone-step mapping
#define MAX_CONE_STEPS 512

//----- Global variables
//-----
cbuffer cbOnRender
{
    float3 g_LightDir;
    float3 g_LightDirTex;
    float4 g_LightDiffuse;
    float4x4 g_mWorldViewProjection;
    float4x4 g_mWorld;
    float3 g_VTextureEyePt;
    float4x4 g_mWorldToTerrain;
    float4x4 g_mTexEyeViewProj;
    float4x4 g_mLightViewProj;
    float4x4 g_mTexLightViewProj;
};

reflection_fragment.ghl
#version 120
#extension GL_ARB_draw_buffers : require
#extension GL_EXT_gpu_shader4 : require

//
// Fragment shader for applying reflections to the deferred shader
//
// Author: Evan Hart
// Email: sdkfeedback@nvidia.com
//
// Copyright (c) NVIDIA Corporation. All rights reserved.
//-----

varying vec2 texCoord;

uniform sampler2D normalTex;
uniform sampler2D materialTex;
uniform sampler2D positionTex;
uniform samplerCube cubeTex;

void main() {

    vec3 normal = texture2D(normalTex, texCoord).xyz * 2.0 - 1.0;
    vec3 mat = texture2D(materialTex, texCoord);
    vec3 position = texture2D(positionTex, texCoord).xyz;

    vec3 spec = vec3(0.0);

    //only apply reflections, if material is reflective
    if (mat.w > 0.0f) {

        vec3 refl = reflect(normalize(position), normalize(normal));
        // bias the reflection to blur it for rougher surfaces
        vec3 env = textureCube(cubeTex, refl).rgb - log(mat.w);
    }
}

GLSL
// various curve tessellation functions
//
// Author: Simon Green
// Email: sdkfeedback@nvidia.com
//
// Copyright (c) NVIDIA Corporation. All rights reserved.

#include "common.cg"

float4 bezierBasis =
(
    { 1, -3, 3, -1 },
    { 0, 3, -6, 3 },
    { 0, 0, 3, -3 },
    { 0, 0, 0, 1 }
);

float4 evaluateBezierPosition(AttribArray<float4> v, float t)
{
    float4 tvec = float4(1, t, tt, tt*t);
    float4 b = mul(bezierBasis, tvec);
    return v[0]*b.x + v[1]*b.y + v[2]*b.z + v[3]*b.w;
}

float3 evaluateBezierPosition(float3 v[4], float t)
{
    float3 p;
    float cmt = 1.0 - t;
    float b0 = cmt*cmt*cmt;
    float b1 = 3.0*t*cmt*cmt;
    float b2 = 3.0*t*t*cmt;
    float b3 = t*t*t;
    return b0*v[0] + b1*v[1] + b2*v[2] + b3*v[3];
}

float3 evaluateBezierTangent(float3 v[4], float t)
{
    float cmt = 1.0 - t;
```

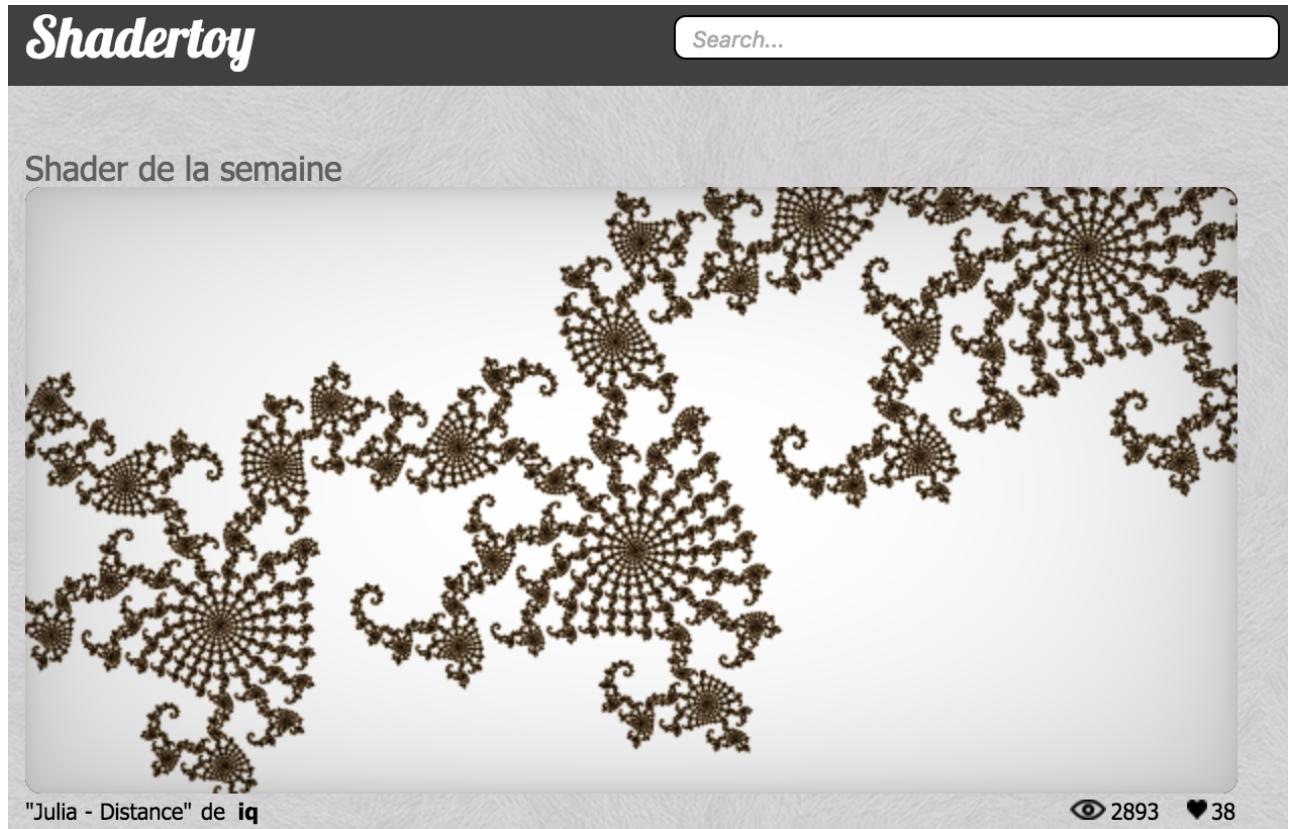
Différentes façons d'appréhender les Shaders

Il existe trois types de Shaders : fragments, pixel et geometric

2. Fragment Shaders

2.1. Bac à sable

Pour nos premiers exemples nous allons utiliser un environnement de test (bac à sable ou *Sandbox*) disponible sur Internet à l'adresse : <https://www.shadertoy.com>



Sandbox ShaderToy

Il suffit de vous inscrire gratuitement pour pouvoir créer vos propres *Fragment Shaders* et les sauvegarder. Vous pourrez également accéder aux exemples de ce chapitre ainsi qu'aux résolutions des exercices proposés. Vous pourrez enfin y parcourir un grand nombre d'exemples réalisés par une très large communauté. Les codes sont disponibles, vous pourrez donc vous en inspirer pour continuer à progresser par la suite.

2.2. Couleurs

Un *Fragment Shader* va attribuer des valeurs *RGBA* (Red, Green, Blue, Alpha) à chaque pixels de l'écran. Le plus simple *Shader* que l'on puisse écrire est le suivant :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Vous voyez que la fonction *mainImage()* va renvoyer un paramètre *fragColor* de type *vec4*. Ce type *vec4* correspond à un vecteur comportant quatre valeurs correspondants aux quatre composantes de couleurs *Red, Green, Blue, Alpha*. Nous pouvons donc créer une variable de type *vec4* et ensuite l'assigner à *fragColor*. Cela ne change rien au résultat mais nous permet d'aborder la déclaration d'une variable de type *vec4* et l'assignation d'une valeur à cette variable.

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec4 solidRed = vec4(1.0, 0.0, 0.0, 1.0);
    fragColor = solidRed;
}
```

Si vous visualisez le résultat, vous voyez que c'est **tout** l'écran de travail qui est en rouge, alors que nous n'avons écrit qu'une seule ligne de code et **aucune** boucle. C'est donc chaque pixel de l'écran qui est impacté par ce code. Vous pouvez déjà imaginer la puissance de cette technique qui permet en une seule ligne de code de remplir un écran complet uniformément avec une couleur.

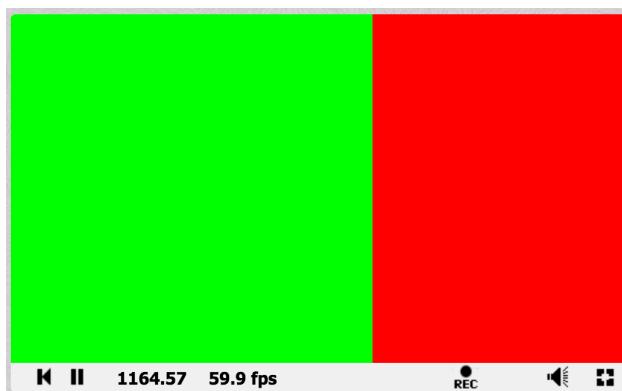
A titre d'exercice, essayez de changer la couleur uniforme rouge en une couleur uniforme grise.

Indication : un niveau de gris s'obtient avec des valeurs égales des trois composantes *R, G* et *B*.

Vous avez sans doute remarqué dans la fonction `mainImage()` qu'il y a un autre paramètre que le `fragColor` : le `fragCoord`. Ce paramètre est de type `vec2` (un vecteur avec deux valeurs), il contient les deux coordonnées (en *x* et *y*) du pixel qui est modifié. Nous pouvons donc changer la couleur de l'écran de travail en tenant compte de la position du pixel modifié avec une instruction conditionnelle (`if`). Par exemple, divisons l'écran verticalement en deux parties : vert à gauche et à rouge à droite.

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    fragColor = vec4(0.0, 1.0, 0.0, 1.0); // vert par défaut
    if(xy.x > 300.0) fragColor = vec4(1.0, 0.0, 0.0, 1.0); // rouge si x>300
}
```

Le résultat est alors le suivant :



Exemple d'écran colorié en deux parties

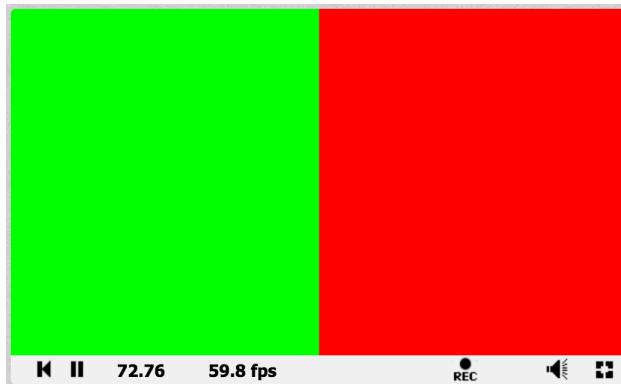
Si nous voulons diviser l'écran en deux parties **égales**, c'est plus difficile car nous ne connaissons pas, à priori, la largeur de l'écran en pixels. De ce fait, nous allons, la plupart du temps, travailler avec les coordonnées de l'image relativement par rapport à la dimension de l'écran (contenue dans la variable de type `vec2` : `iResolution`) en utilisant l'astuce suivante :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy.x = xy.x / iResolution.x;
    xy.y = xy.y / iResolution.y;
    fragColor = vec4(0.0, 1.0, 0.0, 1.0);
    if(xy.x > 0.5){
        fragColor = vec4(1.0, 0.0, 0.0, 1.0);
    }
}
```

Souvent, à la place des deux lignes en gras, vous verrez ce raccourci d'écriture :

```
xy /= iResolution.xy;
```

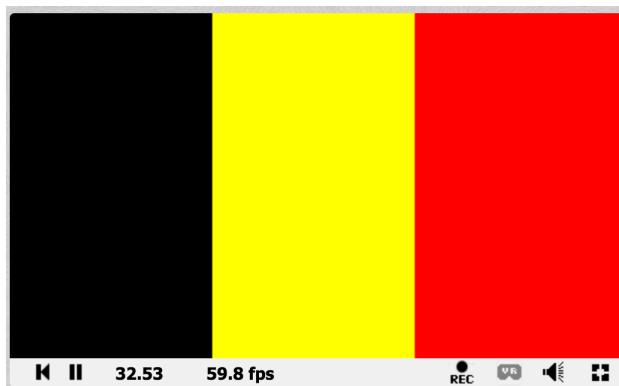
Le résultat est alors le suivant quel que soit la taille de votre écran :



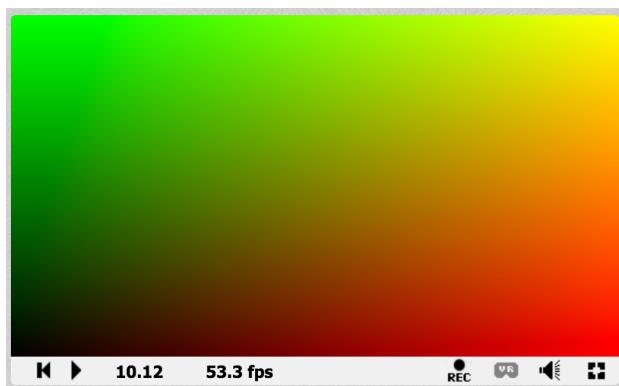
Ecran partagé en deux parties égales

Exercices :

- Dessinez « le drapeau belge » :

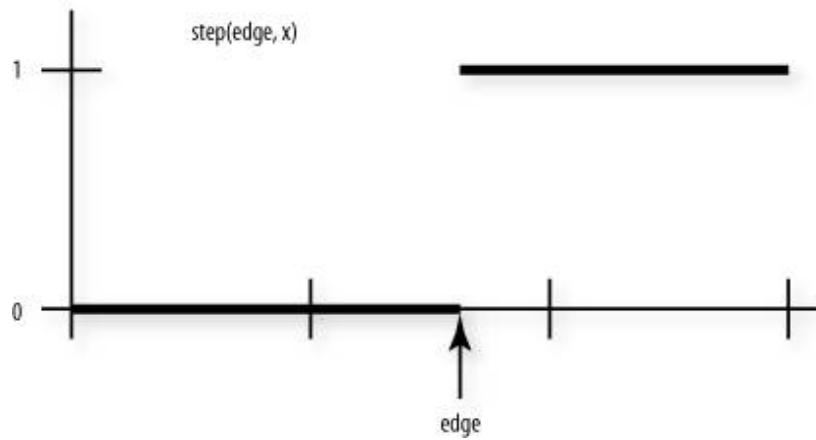


- Réalisez un écran « en dégradé de couleurs » dans le genre de celui-ci :



2.3. Step et SmoothStep

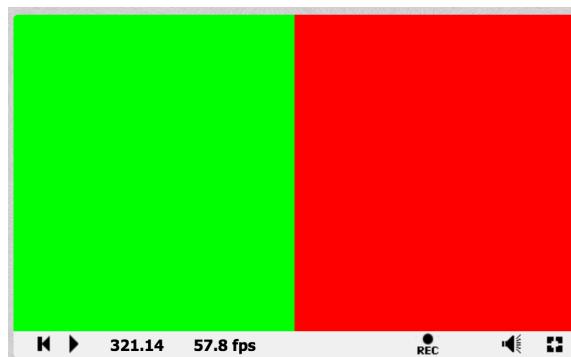
Les résultats précédents auraient pu être obtenus en utilisant une instruction très utile en *OpenGL* : *step(edge, x)*. Cette fonction possède deux paramètres : le seuil (*edge*) et la valeur (*x*) à tester. Si la valeur est inférieure au seuil la fonction va renvoyer 0, sinon elle renverra 1.



La fonction step(edge, x)

Grâce à cette fonction, nous pouvons réécrire le code de l'exemple « Ecran partagé en deux parties égales » de manière plus concise :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy /= iResolution.xy;
    float position = step(0.5, xy.x);
    fragColor = vec4(position, 1.0-position, 0.0, 1.0);
}
```

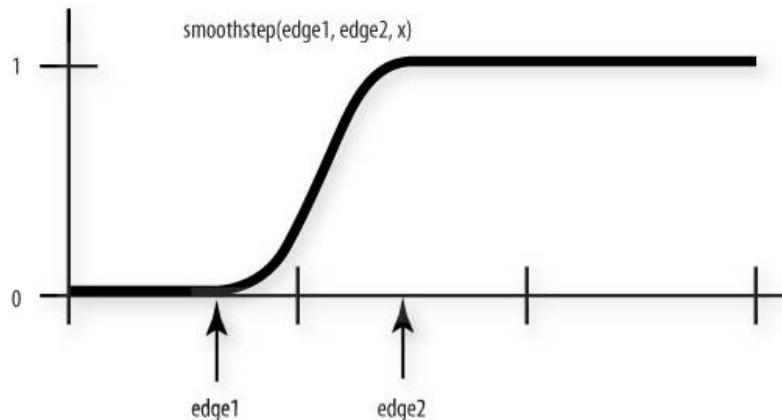


Ecran partagé en deux parties égales avec la fonction step()

On pourrait obtenir identiquement la même chose en utilisant la fonction `smoothstep(edge1, edge2, x)` de cette manière :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy /= iResolution.xy;
    float position = smoothstep(0.5, 0.5, xy.x);
    fragColor = vec4(position, position, position, 1.0);
}
```

L'avantage de cette dernière, c'est qu'elle possède deux paramètres de seuil (`edge1` et `edge2`). S'ils sont égaux on obtient, comme on l'a vu dans l'exemple ci-dessus, le même résultat que le `step()` mais si les valeurs prises par les paramètres sont différentes, la fonction calcule une interpolation non linéaire (comprise entre 0 et 1) entre ces deux seuils.



Représentation de la fonction `smoothstep()`

Le code suivant :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy /= iResolution.xy;
    float greyScale = smoothstep(0.5, 0.5, xy.x);
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}
```

Donne le résultat suivant :

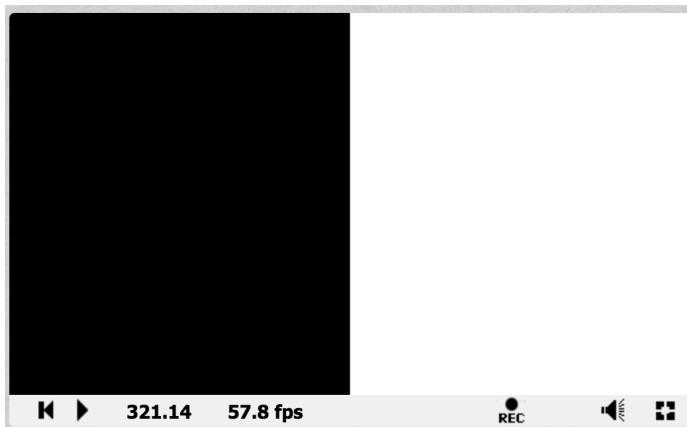


Illustration du smoothstep() à la place du step()

Mais si on différencie les deux paramètres $edge1$ et $edge2$, on obtient alors avec ce code :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy /= iResolution.xy;
    float greyScale = smoothstep(0.3, 0.7, xy.x);
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}
```

Le résultat suivant :

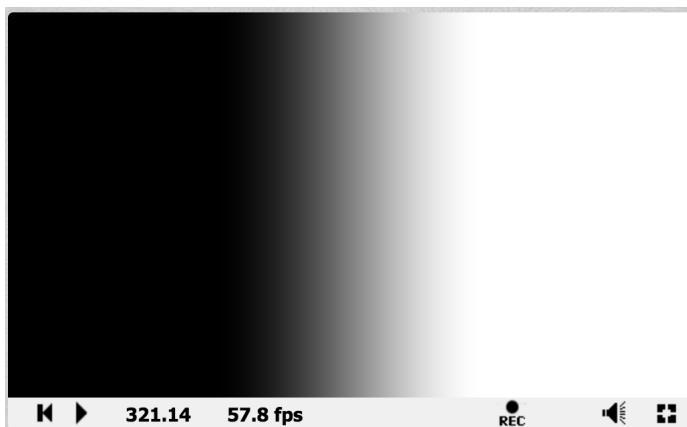


Illustration du smoothstep() avec des seuils différents

Vous pouvez essayer de modifier la ligne du calcul du niveau de gris avec d'autres opérateurs pour observer les motifs que vous pouvez créer.

```
float greyScale = sin(xy.x*10.5);
```



Utilisation de la fonction sin()

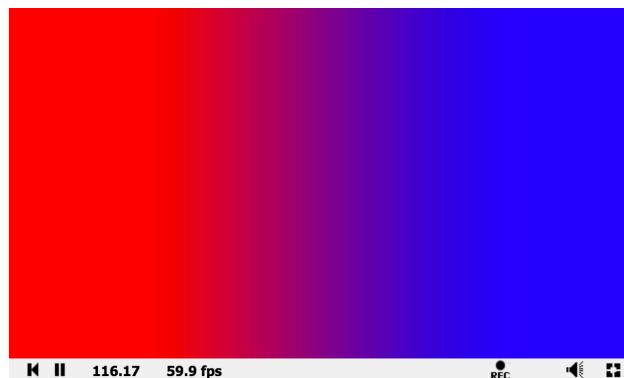
```
float greyScale = pow(xy.x, 6.0);
```



Utilisation de la fonction pow()

Exercice :

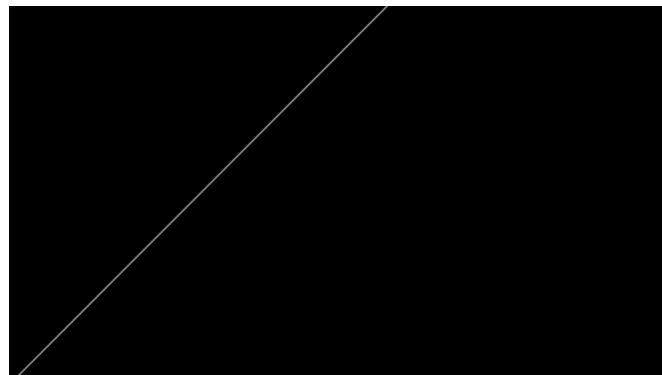
- Dessinez un dégradé similaire à celui-ci en utilisant la fonction *smoothstep()* :



2.4. Tracé d'une droite

Dans ce mode de programmation, il est relativement difficile de tracer une simple droite. On pourrait imaginer calculer si chaque point qu'on est en train de colorier appartient à la droite ou pas. Dans l'exemple suivant on trace la droite d'équation $y = x$:

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    fragColor = vec4(0.0,0.0,0.0,1.0);
    if (fragCoord.xy.x==fragCoord.xy.y) fragColor = vec4(1.0,1.0,1.0,1.0);
}
```

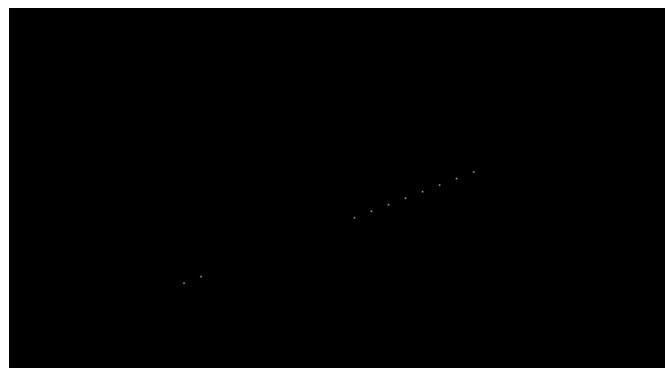


Résultat du code qui permet de tracer une droite $y = x$

Si nous voulons changer l'équation par exemple en $x = 2.6 y - 50$:

```
if (fragCoord.xy.x==fragCoord.xy.y*2.6-50.0) fragColor =
    vec4(1.0,1.0,1.0,1.0);
```

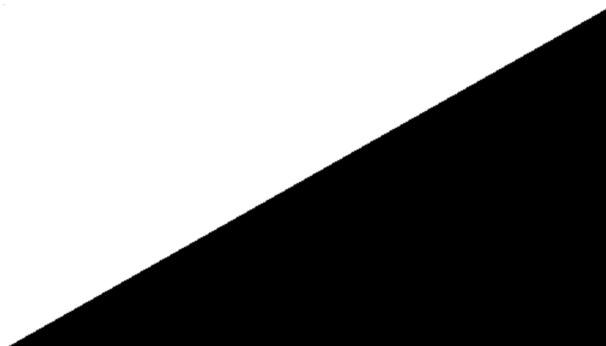
Le résultat malheureusement n'est pas très spectaculaire, on ne voit que quelques points de la droite calculée.



Tracé de la droite $x = 2.6 y - 50$

Ce problème de « pointillisme » est dû au fait que nous faisons une comparaison stricte (== dans le *if*). En fait, la technique classiquement utilisée pour afficher une droite ou une courbe d'équation quelconque est toute autre. On utilise deux fois la fonction *smoothstep()* vue plus haut et on combine ensuite les résultats. Décomposons cette technique en ses différentes étapes pour bien comprendre comment cela fonctionne. On va d'abord utiliser l'équation de la droite ou de la courbe pour séparer l'écran en deux parties noire et blanche.

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 coord = fragCoord.xy / iResolution.xy;
    float y = coord.x;
    float greyScale = smoothstep( y, y, coord.y);
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}
```



Première étape pour tracer une droite

Ensuite nous allons tracer le négatif (inverse) du même écran en changeant simplement la ligne :

```
float greyScale = 1.0 - smoothstep( y, y, coord.y);
```

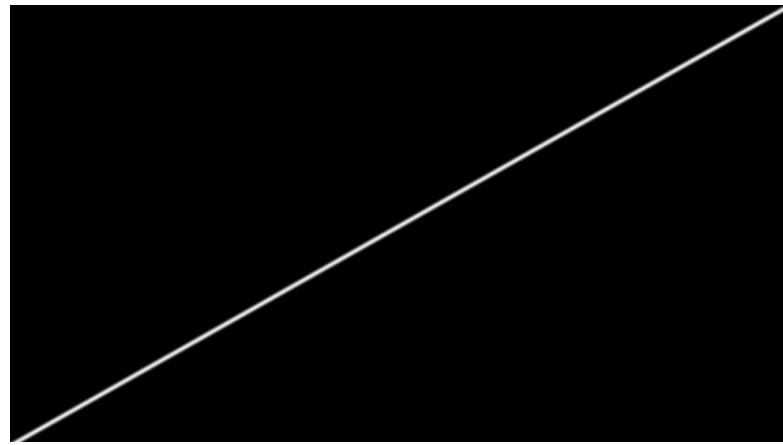


Deuxième étape pour tracer une droite

Si nous combinons (superposition) ensuite les deux parties en faisant un léger décalage de la première vers le bas-droit (-0.01) et de la seconde vers le haut-gauche (+0.01), nous obtenons la ligne de code suivante :

```
float greyScale = smoothstep( y-0.01, y, coord.y)
    - smoothstep( y, y+0.01, coord.y);
```

Et le résultat suivant :



Troisième étape pour tracer une droite

Vous remarquez qu'il est intéressant d'utiliser ce léger décalage (de 0.01 dans notre exemple) comme paramètre de « réglage » pour l'épaisseur du trait. Essayez de modifier cette valeur et observez les effets produits.

Exercice :

- Dessinez une droite en dégradé similaire à cet écran :



2.5. Fonctions

Comme toujours en programmation, il est primordial d'écrire du code réutilisable. Nous allons donc reprendre le code vu précédemment pour le généraliser dans une fonction *plot()* qui permettra d'afficher n'importe quelle courbe à l'écran. Gardez bien cette fonction de côté car nous allons la réutiliser souvent.

```
float plot(vec2 coord, float y, float thickness){  
    return smoothstep( y-thickness, y, coord.y) - smoothstep( y, y+thickness,  
        coord.y);  
}
```

Le code pour dessiner une parabole devient alors simplement :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )  
{  
    vec2 coord = fragCoord.xy / iResolution.xy;  
    float y = coord.x * coord.x;  
    float greyScale = plot( coord, y, 0.02);  
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);  
}
```



Dessin d'une parbole

2.6. Translation

Pour faire translater le repère et donc aussi une courbe ou une image, on utilise un simple déplacement des coordonnées en x et/ou y :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 coord = fragCoord.xy / iResolution.xy;
    ...
    coord.x -= 0.5; // translation en x du repère au centre de l'image
}
```

Ce qui donne, en l'appliquant à l'exemple de la parabole, le résultat suivant :



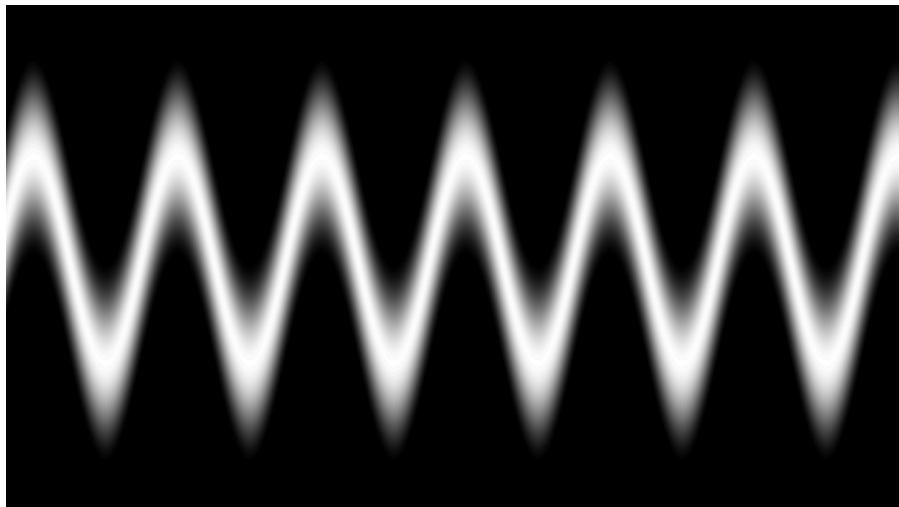
Parabole après translation

Avec le code suivant :

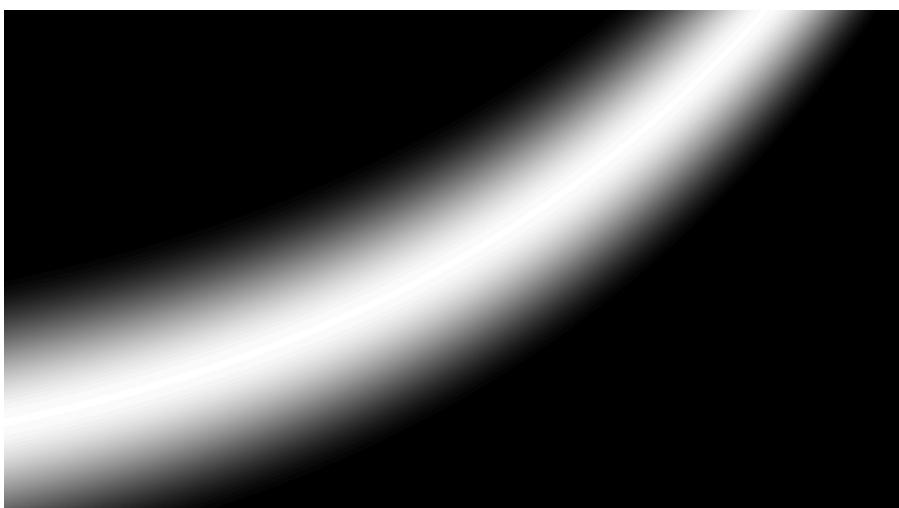
```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 coord = fragCoord.xy / iResolution.xy;
    coord.x -= 0.5;
    float y = 4.0 * coord.x * coord.x;
    float greyScale = plot( coord, y, 0.02 );
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}
```

Exercice :

- Dessiner une sinusoïde dans ce genre :



- Dessiner une exponentielle dans ce genre :



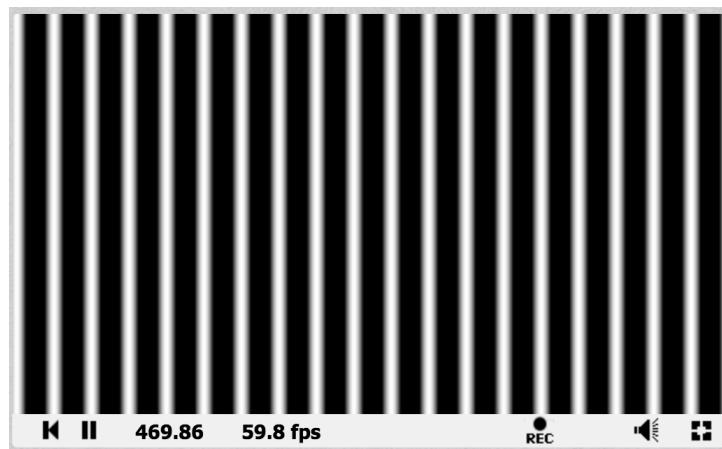
2.7. Mouvement

Pour ajouter un peu de vie, un peu de mouvement dans le résultat on peut utiliser la variable *iGlobalTime* qui évolue continuellement au court du temps. Si on calcule la valeur absolue du *sinus* (ou *cosinus*) de cette variable on obtient des valeurs entre 0 et 1 qui changent au cours du temps. Par exemple, on peut écrire :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy / iResolution.xy;
    fragColor = vec4(1.0-abs(cos(iGlobalTime)),
                     abs(cos(iGlobalTime)),
                     1.0-abs(cos(iGlobalTime)),
                     1.0);
}
```

Et si on combine ce qu'on a vu plus tôt avec cette nouvelle notion on peut commencer à réaliser des animations complexes et visuellement intéressantes. essayez par exemple, le code suivant :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy /= iResolution.xy;
    float greyScale = sin(xy.x*120.0+(iGlobalTime*100.0));
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}
```



Difficile de visualiser le mouvement sur du papier

Exercice :

- Combiner les deux chapitres précédent en réalisant un mouvement de translation sur une courbe exponentielle



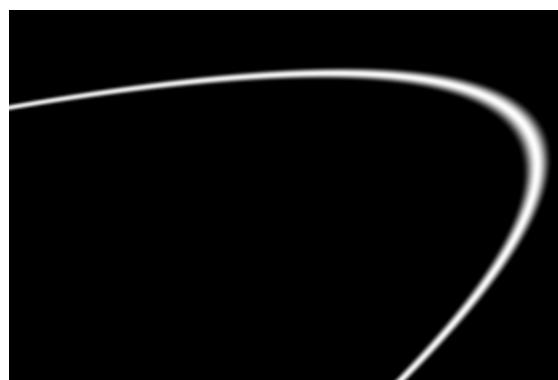
2.8. Rotation

Pour faire tourner le repère, on utilise les matrices de transformation (*mat2*) fournie par *OpenGL* :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 coord = fragCoord.xy / iResolution.xy;
    float rotation = 1.57; // Angle en radian
    coord -= 0.5; // translation du repère au centre de l'image
    coord = mat2( cos(rotation),
                  -sin(rotation),
                  sin(rotation),
                  cos(rotation)) * coord; // Rotation
    ...
}
```

A titre d'exemple, faisons tourner la parabole en fonction du temps (*iGlobalTime*) :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 coord = fragCoord.xy / iResolution.xy;
    coord -= 0.5;
    float rotation = iGlobalTime;
    coord = mat2( cos(rotation),
                  -sin(rotation),
                  sin(rotation),
                  cos(rotation)) * coord; // Rotation
    float y = 4.0 * coord.x * coord.x;
    float greyScale = plot( coord, y, 0.02);
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}
```



Instantané de la rotation de la parbole

2.9. Grille

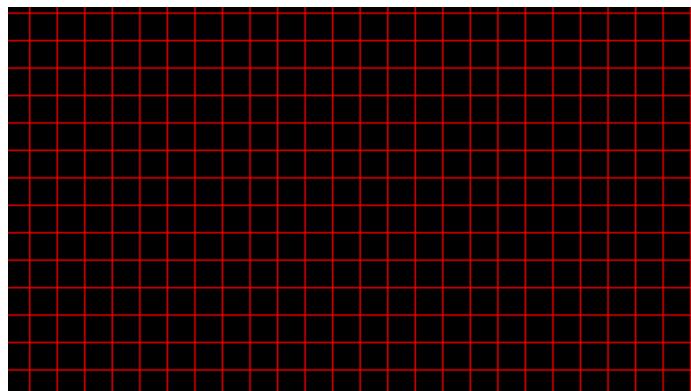
Pour tracer plusieurs lignes (pour par exemple faire une grille) on utilise fréquemment les instructions *fract(valeur)* qui retourne la partie fractionnelle d'une valeur (par exemple, *fract(2.5)* retourne 0.5) et *max(param1, param2)* qui retourne la valeur maximale entre deux valeurs passées en paramètre. A partir de ces deux nouvelles fonctions nous allons pourvoir construire la grille. Commençons par calculer si la coordonnée courante (*fragCoord*) est située sur la grille ou pas :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 gridSize = vec2(20.0, 20.0);
    vec2 coord = fragCoord.xy;
    vec2 gridCoord = fract(coord / gridSize) * gridSize;
    vec2 isALine = step(gridCoord, vec2(1.0));
    ...
}
```

Ensuite, si nous sommes sur une ligne horizontale ou verticale de la grille le pixel courant prend la valeur rouge :

```
float color = max(isALine.x, isALine.y);
fragColor = vec4(color, 0.0, 0.0, 1.0);
```

Ce qui donne le résultat suivant :



Dessin d'une grille

Exercice :

- Modifier la couleur, la taille et la position initiale de la grille dynamiquement pour créer une animation.

2.10. Mixage de couches

Il est souvent intéressant de superposer plusieurs dessins/images les uns sur les autres. On peut utiliser la fonction *max()* vue précédemment mais on préfère souvent la fonction *mix(x, y, a)* qui interpole la nouvelle valeur en fonction des deux valeurs *x* et *y* et du facteur d'interpolation (*a*) de la manière suivante : $mix = x.(1-a) + y.a$

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy /= iResolution.xy;
    float positionX = step(0.5, xy.x);
    float positionY = step(0.5, xy.y);
    vec4 layer1 = vec4(positionX, 1.0-positionX, 0.0, 1.0);
    vec4 layer2 = vec4(0.0, positionY, 1.0-positionY, 1.0);
    fragColor = mix(layer1, layer2, 0.5);
}
```

Vous voyez dans le code précédent que l'on crée d'abord deux layers qui divisent respectivement l'écran en deux couleurs verticalement et horizontalement et ensuite on les mélange avec la fonction *mix()* et un facteur d'interpolation égale à 0.5.

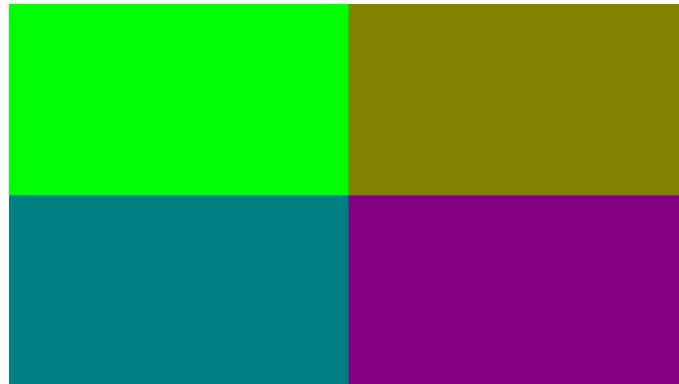


Illustration de la fonction mix()

On obtient avec cette fonction des combinaisons d'images qui peuvent rapidement devenir très puissantes. Essayez à titre d'exercice de modifier les loyers et d'appliquer des facteurs d'interpolation différents.

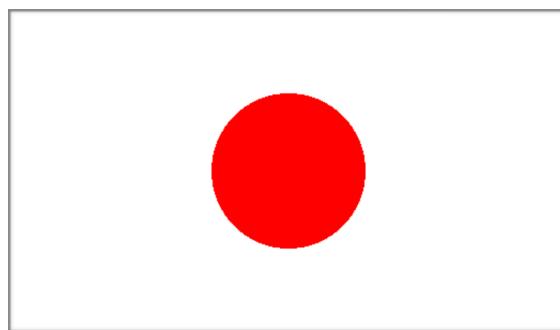
2.11. Disque

Le dessin du cercle va nous permettre d'aborder une fonction très intéressante : `length(vec2 ...)`. Elle permet, comme son nom l'indique, de calculer la longueur d'un vecteur. Pour dessiner un disque, il suffit donc de calculer la l'éloignement du pixel considéré par rapport au centre du disque pour déterminer sa couleur. On peut dès lors créer une fonction `disk()` que nous pourrons réutiliser facilement :

```
vec4 disk(vec2 uv, vec2 center, float radius, vec4 color) {  
    float dist = step(length(center - uv), radius);  
    return vec4(dist, dist, dist, 1.0)*color;  
}
```

Exercice :

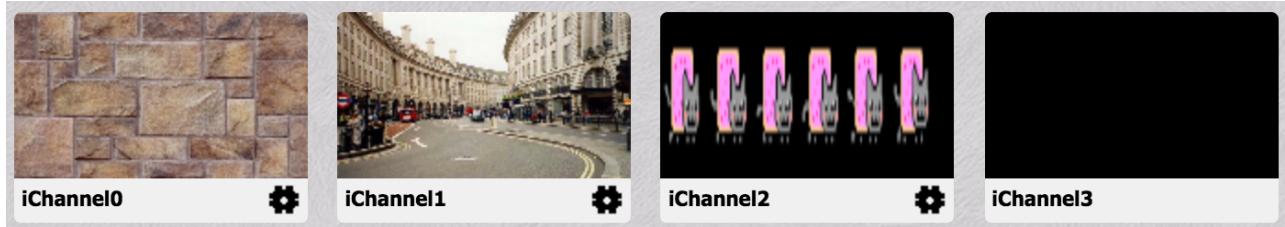
- Dessiner le drapeau nippon :



Utilisation de la fonction length()

2.12.Bitmap

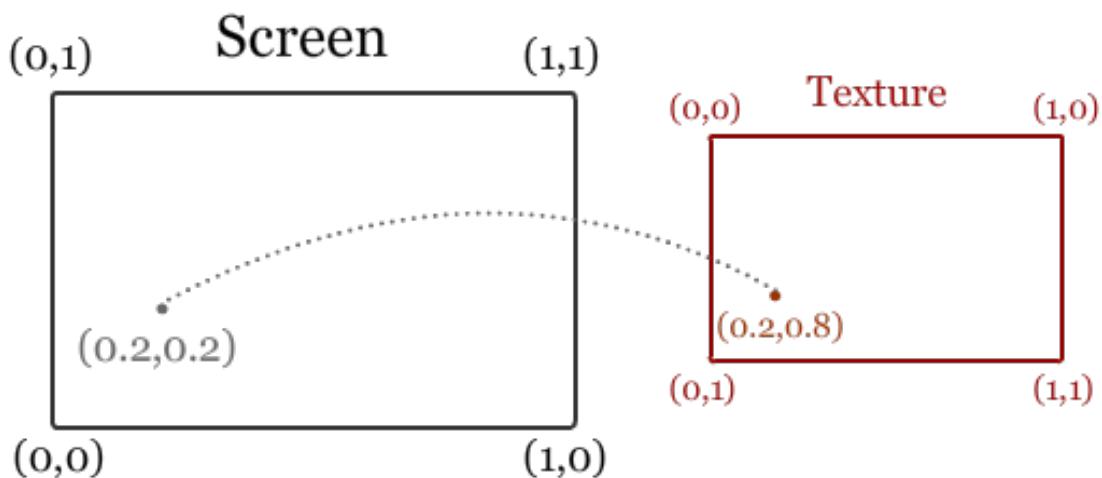
Nous allons maintenant partir d'une image (*Bitmap*) et la transformer. Dans le *SandBox* dans lequel nous « jouons », l'image de départ est référencée comme étant *iChannelx*. Le *x* dans *iChannel* représentant le numéro de l'image sélectionnée dans le bas de l'écran :



Pour accéder et recopier chaque pixel de l'entrée *iChannel0* vers l'écran il suffit d'écrire le code suivant :

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy / iResolution.xy;
    fragColor = texture(iChannel0, xy);
}
```

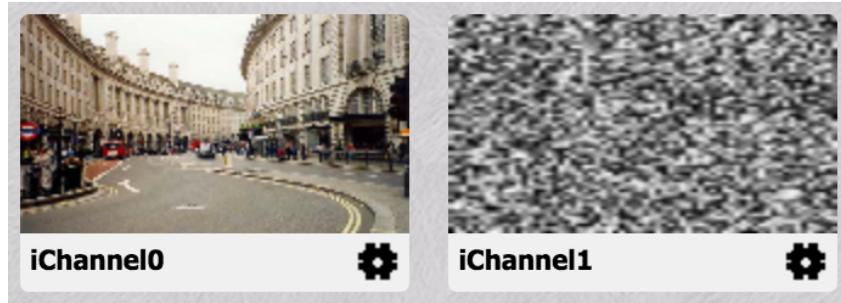
Dans cet exemple, on obtient la valeur de la couleur du pixel d'entrée grappe à la fonction *texture(iChannel0, xy)* que l'on recopie vers la sortie *fragColor*.



Recopie d'une image en entrée vers la sortie

Exercices :

Mettez d'abord ces deux images en *iChannel* 0 et 1 :



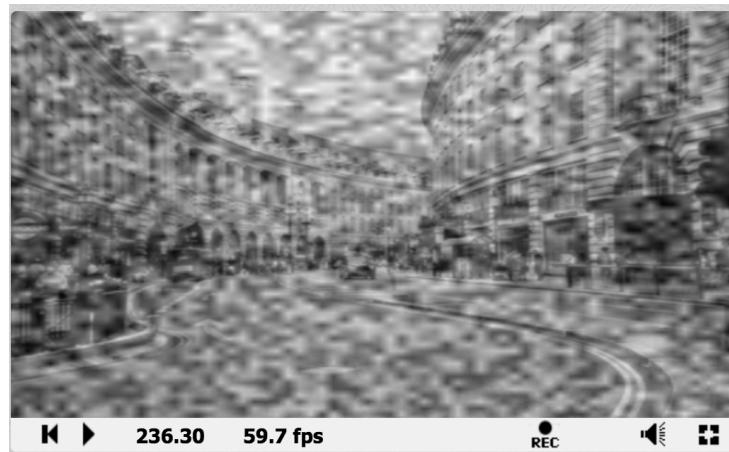
Images en entrée

Et ensuite :

- Transformez la première image couleur en niveau de gris :



- Mélangez les deux images (0 et 1) pour obtenir ce résultat :



2.13.Vidéo

On peut également mélanger une image fixe ou des couleurs à une vidéo. Il suffit pour cela de charger dans les *iChannelx* une des vidéos proposées par le *SandBox*.

Exercice :

- Intégrer la vidéo de *Jean-Claude Vandamme* sur fond vert dans l'image tel que la capture d'écran ci-dessous :



Mélange vidéo avec GreenKey et image fixe

Solution des exercices

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec4 solidRed = vec4(0.3,0.3,0.3,1.0);
    fragColor = solidRed;
}
```

Exercice de l'écran en niveau de gris uniforme

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy.x = xy.x / iResolution.x;
    xy.y = xy.y / iResolution.y;
    fragColor = vec4(0.0, 1.0, 0.0, 1.0);
    if(xy.x < 0.33){
        fragColor = vec4(0.0, 0.0, 0.0, 1.0);
    }
    else if (xy.x < 0.66){
        fragColor = vec4(1.0, 1.0, 0.0, 1.0);
    }
    else {
        fragColor = vec4(1.0, 0.0, 0.0, 1.0);
    }
}
```

Exercice du « Drapeau belge »

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy.x = xy.x / iResolution.x;
    xy.y = xy.y / iResolution.y;
    fragColor = vec4(xy.x, xy.y, 0.0, 1.0);
}
```

Exercice du « Dégradé de couleurs »

```

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;
    xy /= iResolution.xy;
    float position = smoothstep(0.1, 0.9, xy.x);
    fragColor = vec4(1.0-position, 0.0, position, 1.0);
}

```

Exercice du « Dégradé de couleurs avec smoothstep() »

```

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 coord = fragCoord.xy / iResolution.xy;
    float y = coord.x;
    float greyScale = smoothstep( y-sin(y)/2.0, y, coord.y)
                    - smoothstep( y, y+sin(y)/2.0, coord.y);
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}

```

Exercice de la « Droite en dégradé »

```

float plot(vec2 coord, float y, float thickness){
    return smoothstep( y-thickness, y, coord.y)
           - smoothstep( y, y+thickness, coord.y);
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 coord = fragCoord.xy / iResolution.xy;
    float y = sin(coord.x*40.0)/5.0+0.5;
    float greyScale = plot( coord, y, 0.2);
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}

```

Exercice de la « Sinusoïde en dégradé »

```

float plot(vec2 coord, float y, float thickness){
    return smoothstep( y-thickness, y, coord.y)
           - smoothstep( y, y+thickness, coord.y);
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 coord = fragCoord.xy / iResolution.xy;
    float y = exp(coord.x*2.0-1.0)/2.0;
    float greyScale = plot( coord, y, 0.3);
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}

```

Exercice de l' « Exponentielle en dégradé »

```

float plot(vec2 coord, float y, float thickness) {
    return smoothstep( y-thickness, y, coord.y) - smoothstep( y, y+thickness,
    coord.y);
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 coord = fragCoord.xy / iResolution.xy;
    coord -= abs(cos(iGlobalTime)/2.0);
    float y = exp(coord.x*1.0-cos(iGlobalTime))/2.0 - 0.4;
    float greyScale = plot( coord, y, 0.1);
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}

```

Exercice de l'« Exponentielle en mouvement »

```

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy / iResolution.xy;
    vec4 inputColor = texture(iChannel0, xy);
    float greyScale = (inputColor.r + inputColor.g + inputColor.b ) / 3.0;
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}

```

Exercice de l'image transformée en niveaux de gris

```

vec4 disk(vec2 uv, vec2 center, float radius, vec4 color) {
    float inDisk = step(length(center - uv), radius);
    if (inDisk==0.0) return vec4(1.0, 1.0, 1.0, 1.0);
    else return vec4(inDisk, inDisk, inDisk, 1.0)*color;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ){
    vec2 xy = fragCoord.xy;
    xy /= iResolution.xy;
    vec2 center = iResolution.xy * 0.5;
    float radius = 0.25 * iResolution.y;
    fragColor = disk(fragCoord.xy, center, radius, vec4(1.0, 0.0, 0.0, 1.0));
}

```

Exercice du « Drapeau Nippon »

```

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy / iResolution.xy;
    vec4 inputColor = texture(iChannel0, xy);
    float greyScale = (inputColor.r + inputColor.g + inputColor.b) / 3.0;
    greyScale += texture(iChannel1, xy).r;
    greyScale /= 2.0;
    fragColor = vec4(greyScale, greyScale, greyScale, 1.0);
}

```

Exercice du « Mélange d'images »

```

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    vec2 xy = fragCoord.xy / iResolution.xy;
    vec4 texColor = texture(iChannel0,xy);
    vec4 backgroundColor = vec4(13.0/255.0, 161.0/255.0, 37.0/255.0, 1.0);
    float difference = distance(texColor, backgroundColor);
    if( difference < 0.4 ) {
        texColor = texture(iChannel1,xy);
    }
    fragColor = texColor;
}

```

Exercice du GreenKey