

A wireframe landscape rendered in white lines on a black background. The foreground features a perspective grid that recedes into the distance. In the midground, there are jagged, mountain-like shapes composed of various polygons. The background is a dark sky filled with numerous small, bright white dots representing stars.

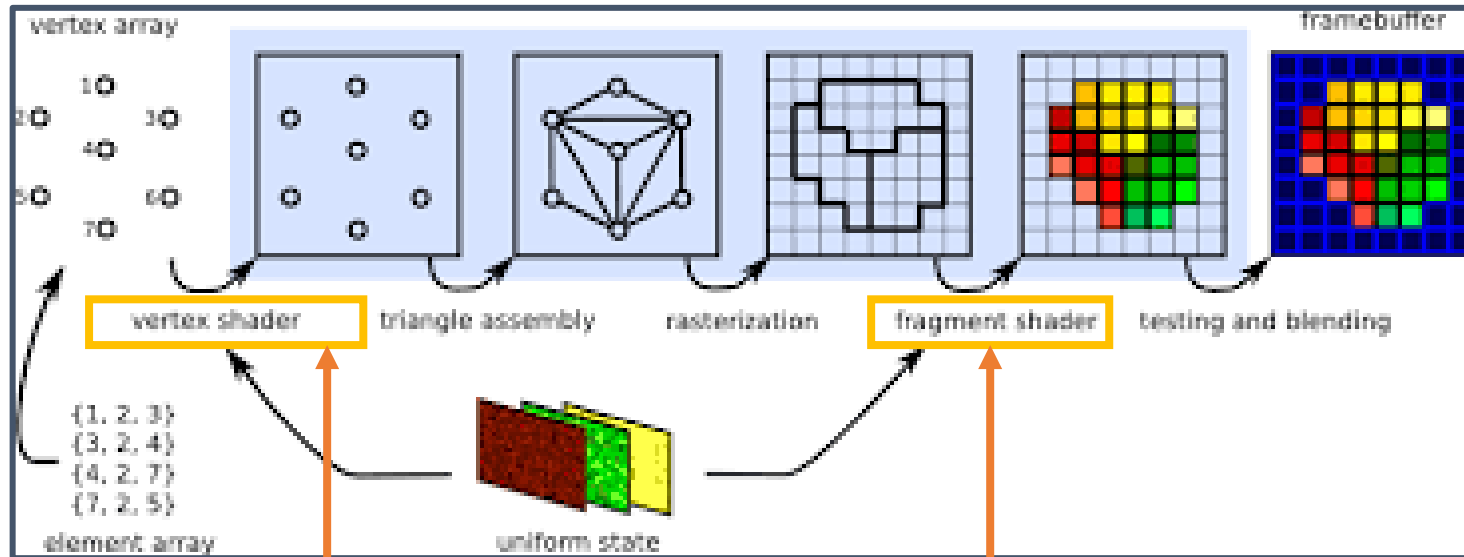
Shadertoy

OpenGL graphics programming

Mini-introduction

RAPPEL : pipeline graphique (OpenGL graphics pipeline)

Vous avez vu la programmation de “vertex shader”
et de “fragment shader” en cours de synthèse d’images.



Fragment shader

Après “rastérisation” des primitives graphiques (ex: triangles, lignes, points), chacun de leur pixel à l’écran génère un “fragment” (pixel + attributs interpolés entre les points [ex : couleurs, normales, profondeurs, etc...]).

Un **fragment shader** (programme `main()`) est appelé pour chaque fragment. En graphique, son but est de déterminer la couleur finale du pixel, qui sera ensuite envoyée dans le framebuffer (buffer/mémoire vidéo ensuite affiché à l’écran).

Vertex shader

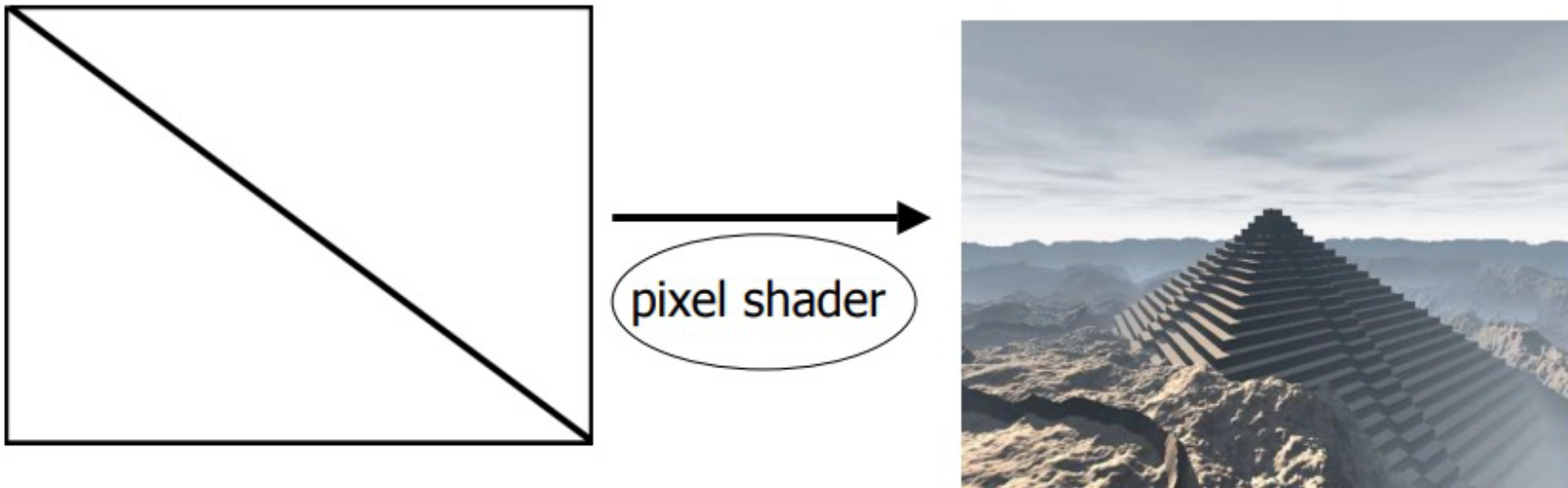
transforme vos points dans l’espace par des matrices :

- Model : translation, rotation, scale (world position)
- View : scène vue depuis une “caméra” virtuelle
- Projection : orthogonale ou pas (point de fuite, fish eye, sphérique...)

A bit of context

- The idea: draw two triangles that cover the entire screen area, and invoke a pixel shader that will create an animated or static image.

NOTE: pixel shader == fragment shader



- Make the complete demo self-contained in no more 4096 bytes (that includes the "engine", music, shaders, animations, textures and everything).

Pour simplifier :

Comme pour Python, C++, etc..., le GPU fournit une API avec des threads.

On aura 1 thread par pixel sur GPU (*simplification*). Le thread exécute une fonction main() (le fragment shader), et sera identifiable par son identifiant.

Fragment shader



thread function

output thread data (color)

kind of threadID
(2D position in window)

shadertoy.com/new

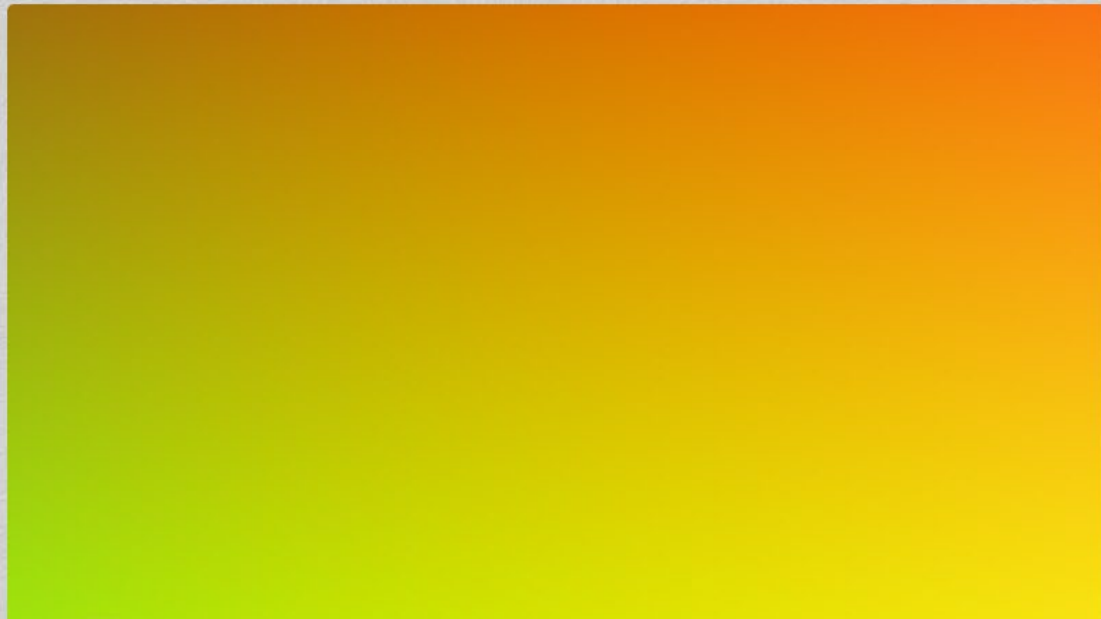
Shadertoy

Search...

Explorer

Nouveau

Connexion



4.95

60.0 fps

800 x 450



Shadertoy => programmation en "GLSL"
(OpenGL Shading Language)

+ Image

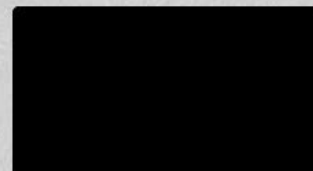
Données d'entrée du shader

```
1 void mainImage( out vec4 fragColor, in vec2 fragCoord )
2 {
3     // Normalized pixel coordinates (from 0 to 1)
4     vec2 uv = fragCoord / iResolution.xy;
5
6     // Time varying pixel color
7     vec3 col = 0.5 + 0.5 * cos( iTime + uv.xyx + vec3(0,2,4) );
8
9     // Output to screen
10    fragColor = vec4( col, 1.0 );
11 }
```

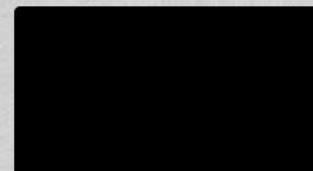
Compiled in 0.0 secs

158 chars

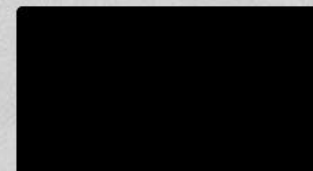
S ?



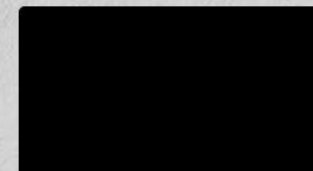
iChannel0



iChannel1



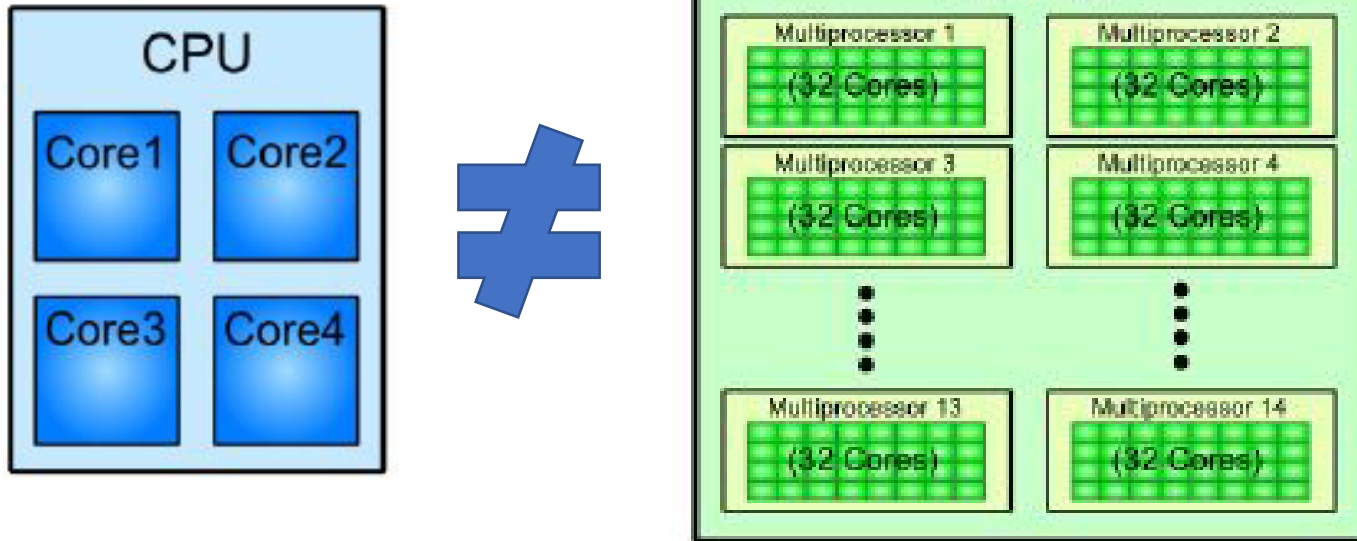
iChannel2



iChannel3

shader inputs (textures [from image or video], uniforms)

Principes de base derrière "Shadertoy"



CPU

Quelques coeurs. Peuvent faire des choses complexes (ex : branchements, récursivité, etc...). Context switch long.

GPU

Plusieurs centaines voire milliers de threads. Légers et context switch très rapide, mais peut faire moins de chose que CPU, et ressources limités (ex : registres).

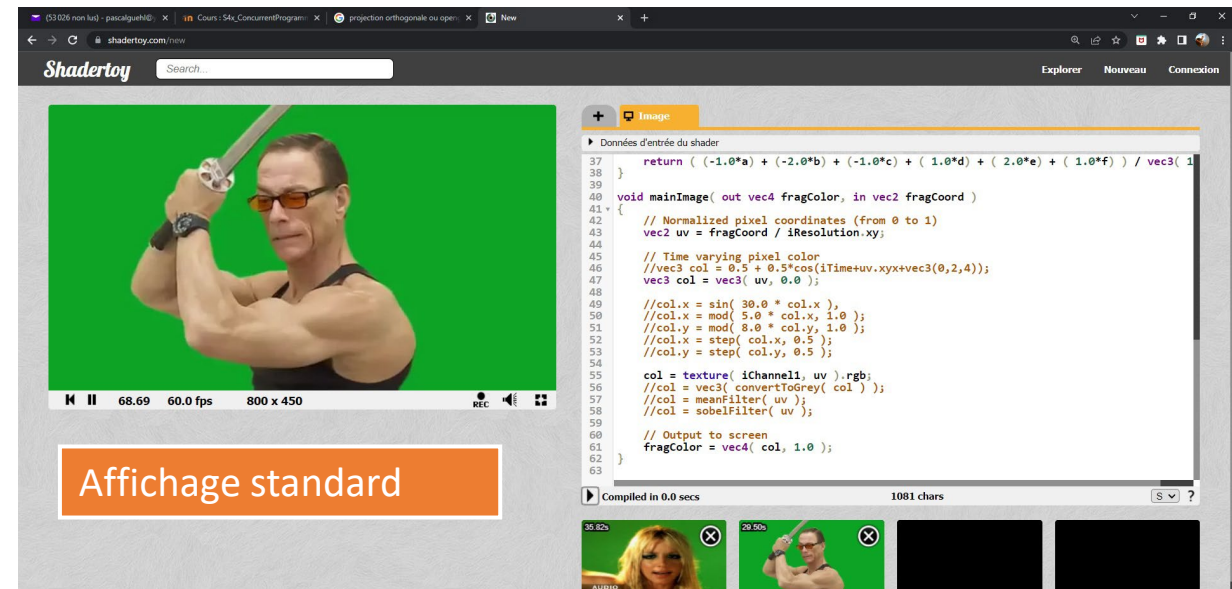
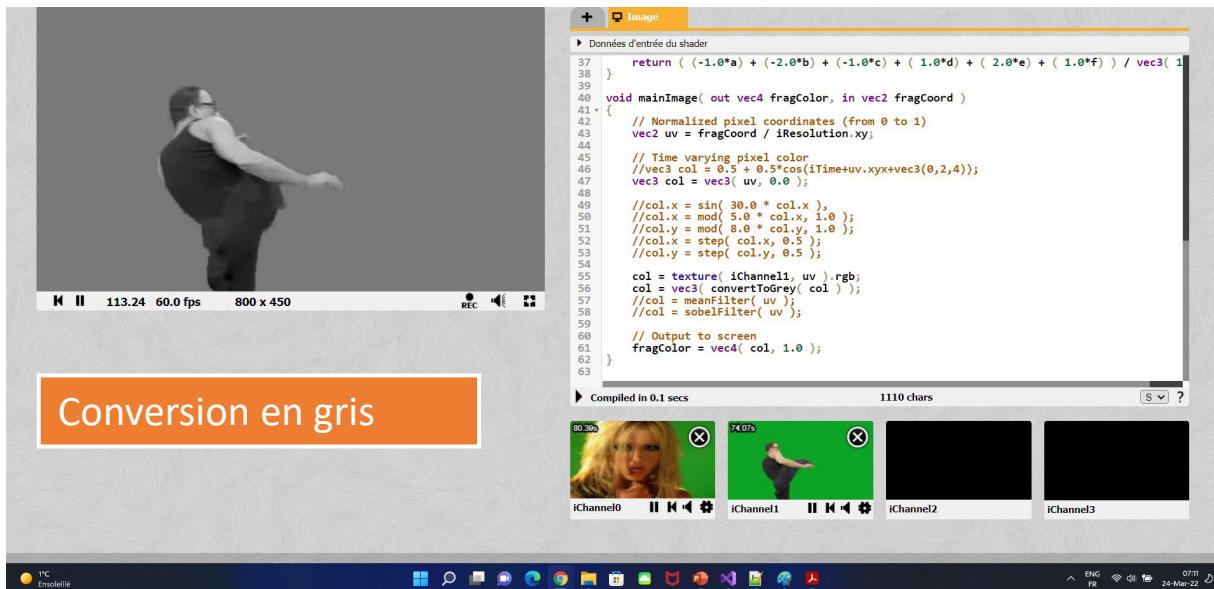
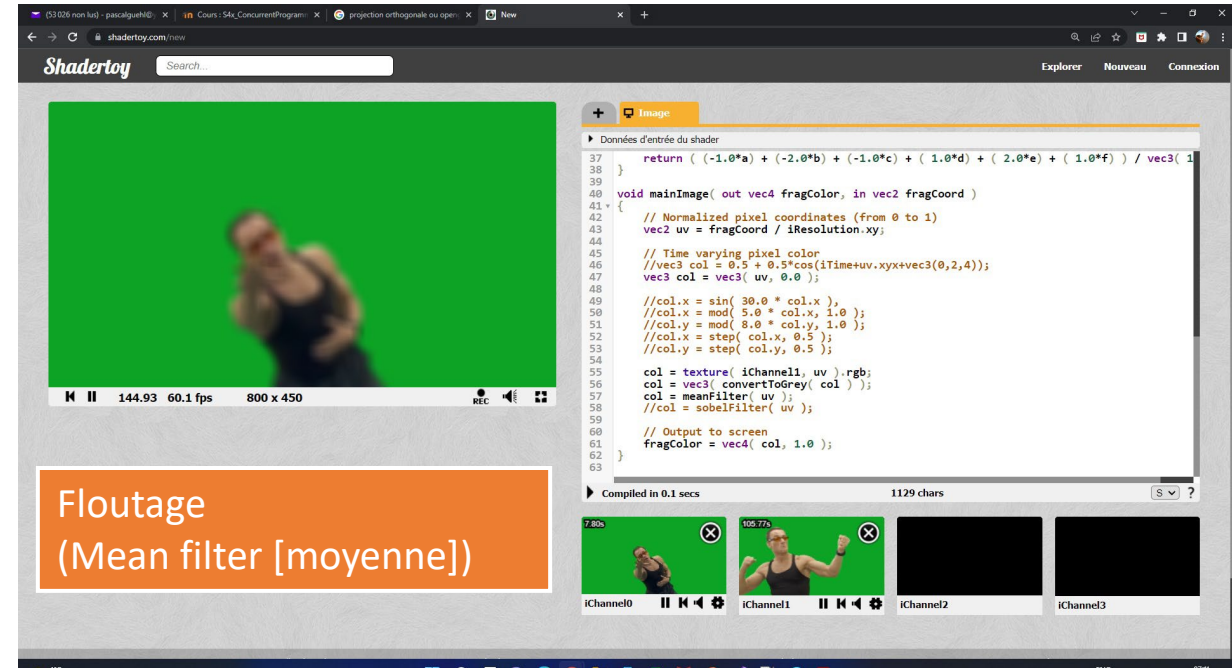
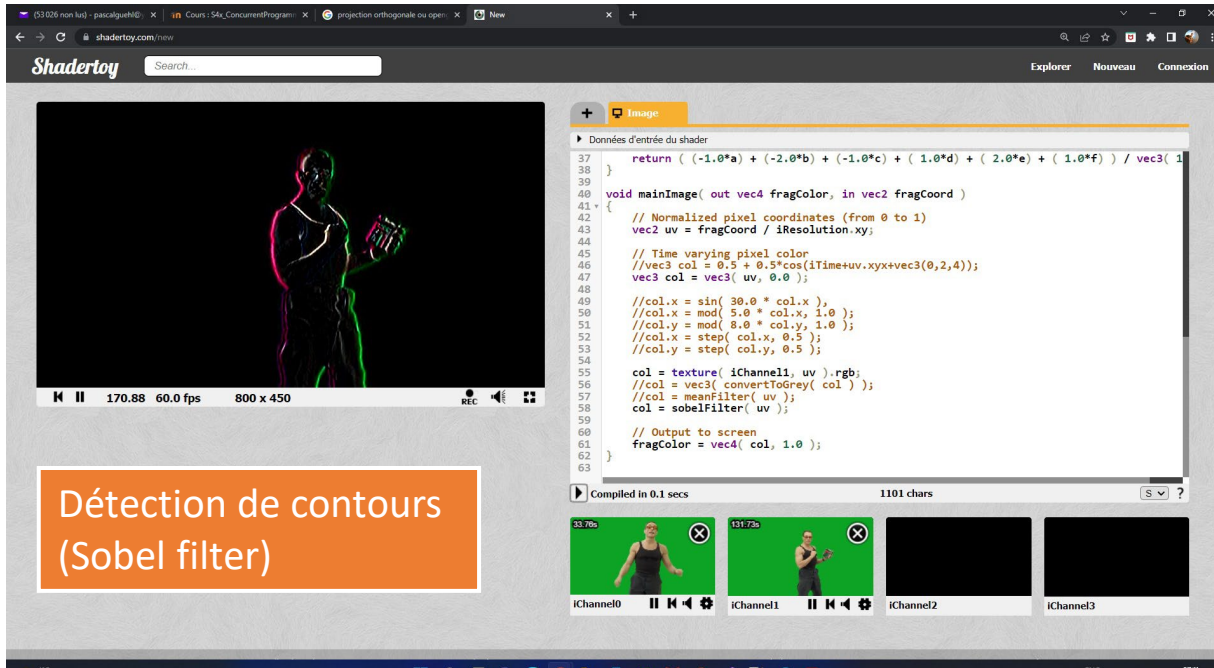
NOTE

Contrairement au CPU, ici les threads GPU s'exécutent tous en parallèle, synchronisé en suivant les ticks d'une horloge/clock hardware. S'il y a des branchements différents dans le code (ex : if, for, while...), certains threads attendent puis repartent ensemble.

NOTE :

En fait, les coeurs GPU sont beaucoup plus simples/légers que sur CPU et il peut y en avoir des milliers. Chacun va exécuter un thread. S'il n'y a pas assez de coeurs pour traiter les pixels d'une image, il y aura une autre passe de traitements sur les coeurs.

Quelques exemples de traitement d'images temps-reel, codés en TP sur des flux vidéos (ou images)



Fragment shader

thread function

output thread data (color)

kind of threadID
(2D position in window)

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
```

```
// Normalized pixel coordinates (from 0 to 1)
vec2 uv = fragCoord / iResolution.xy;
```

```
vec3 col = texture( iChannel1, uv ).rgb;
//col = vec3( convertToGrey( col ) );
//col = meanFilter( uv );
//col = sobelFilter( uv );
```

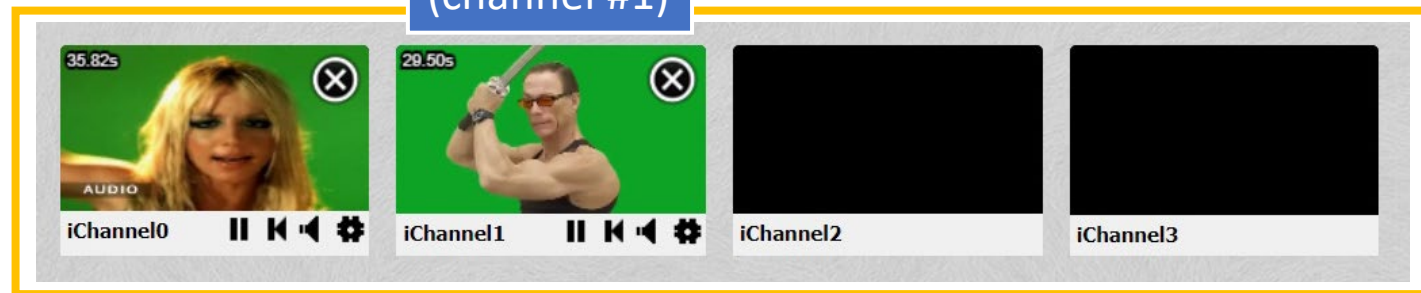
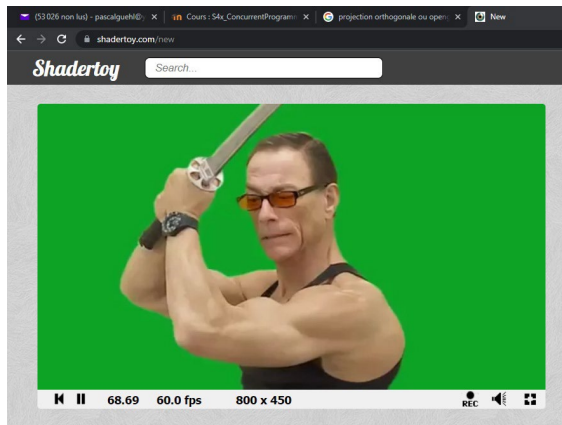
```
// Output to screen
fragColor = vec4( col, 1.0 );
```

“read texture”
fonction GLSL :

- texture() +
- texture ID +
- UV position (coordonnées de texture)

texture
(channel #1)

Affichage standard



shader inputs (textures [from image or video], uniforms)

```
float convertToGrey( in vec3 c )
{
    return 0.21 * c.r + 0.71 * c.g + 0.07 * c.b;
}
```

Conversion en gris

```
vec3 meanFilter( in vec2 uv )
{
    vec2 offset = vec2( 1.0 ) / iResolution.xy;

    vec3 result = vec3( 0.0 );
    int maskSize = int( 10.0 * ( 0.5*sin( 5.0 * iTime )+0.5 ) );
    for ( int j = -maskSize; j <= maskSize; j++ )
    {
        for ( int i = -maskSize; i <= maskSize; i++ )
        {
            vec2 newUV = uv + vec2( offset.x * float( i ), offset.y * float( j ) );
            result = result + texture( iChannel0, newUV ).rgb;
        }
    }
    return result / ( (2.0*float(maskSize)+1.0)*(2.0*float(maskSize)+1.0) );
}
```

Floutage
(Mean filter [moyenne])

```
vec3 sobelFilter( in vec2 uv )
{
    vec2 offset = vec2( 1.0 ) / iResolution.xy;

    // Left (top to bottom)
    vec3 a = texture( iChannel1, uv + vec2( -offset.x, offset.y ) ).rgb;
    vec3 b = texture( iChannel1, uv + vec2( -offset.x, 0.0 ) ).rgb;
    vec3 c = texture( iChannel1, uv + vec2( -offset.x, -offset.y ) ).rgb;

    // Right (top to bottom)
    vec3 d = texture( iChannel1, uv + vec2( offset.x, offset.y ) ).rgb;
    vec3 e = texture( iChannel1, uv + vec2( offset.x, 0.0 ) ).rgb;
    vec3 f = texture( iChannel1, uv + vec2( offset.x, -offset.y ) ).rgb;

    return ( (-1.0*a) + (-2.0*b) + (-1.0*c) + ( 1.0*d) + ( 2.0*e) + ( 1.0*f) ) / vec3( 1.0 );
}
```

Détection de contours
(Sobel filter)

A wireframe landscape with a grid floor and starry sky. The floor is a perspective grid of white lines on a black background. In the distance, there are jagged, mountain-like shapes made of white wireframes. The sky is black with many small white dots representing stars.

Shadertoy

On peut faire bien plus avec la programmation de shaders ! (ex : GPU Computing [le GPU est vu comme un accélérateur matériel générique pour la physique, la finance, le medical, les traitements audio et video, etc...])

On a vu juste des bases.