# TRABAJO PRÁCTICO N°5

***Programación Paralela (Shaders)***

**Docentes:**

**David Petrocelli**
**Nehuen Prados**
**Mariano Rapaport**
**Alejandro Romero**

**Integrantes de Grupo:**

Asenzo, Mateo
Bianchi, Enzo
Gatica Odato, Josué
Latessa, Lucas

# Trabajo Práctico Nº 5

## Programación Paralela (Shaders)

**Fecha de Entrega: 20/05/24**
**(1 a 6 obligatorios)**

La programación paralela es la respuesta a la necesidad de tener escalabilidad vertical. Es inherente a existencia de la programación paralela la existencia de 1 único nodo. En el núcleo de un sistema distribuido se encuentra el procesamiento sincrónico. El core de un procesamiento en paralelo, es la sincronización. El eje central de este procesamiento es la gestión de recursos compartidos.

**Comencemos con lo básico.**

Imaginemos un **proceso A** que debe realizar un cómputo paralelo, este cálculo normalmente se realiza en GPU, ya que la misma ofrece un elevado número de núcleos de bajo rendimiento individual mientras que la CPU ofrece muy pocos núcleos de alto rendimiento.
El procesamiento de una tarea en paralelo tiene algunas implicancias que se deben tener en cuenta ya que su funcionamiento es sustancialmente diferente al que estamos acostumbrados en CPU.

La forma más “visible” de entender estas diferencias es mediante el procesamiento de gráficos, más concretamente del vídeo en tiempo real.

- **Podemos pensar que un pixel es un vector de 4 unsigned byte, el primero representa el rojo, el segundo el verde, el tercero el azul y el cuarto la transparencia.**
- **Podemos pensar que una línea horizontal de píxeles es un vector de x píxeles, donde x es el ancho de la línea.**
- **Podemos pensar que una imagen es un vector de y líneas horizontales, donde y es el alto de la imagen.**
- **Podemos pensar que un video es un vector de f imágenes, donde f es la cantidad de fotogramas del video.**

En conclusión, procesar un video aplicando una operación como ser un filtro de escala de grises a cada píxel, **requiere f * y * x operaciones**, si esto lo hiciéramos en CPU con un único núcleo tendríamos que hacer 3 for anidados y sería costoso en tiempo computacional.

Cabe aclarar, que el procesamiento paralelo no se limita a procesamiento gráfico, comenzaremos esta guía con este enfoque ya que es la forma más divertida y didáctica de realizar una introducción.

**Hit #1**

Visite https://en.wikipedia.org/wiki/Shader#Pixel_shaders en su versión en inglés y lea atentamente los apartados. Comience a elaborar un informe donde documente someramente los tipos de shaders. En particular nos enfocaremos en los Píxel Shaders que operan en 2D.

Un shader es un programa que calcula los niveles apropiados de luz, oscuridad y color durante la representación de una escena 3D. Hoy día las gpu son las encargadas de realizar este trabajo y son más eficientes que las antiguas no solo por haber aumentado su poder de cómputo, sino también por permitir realizar el procesamiento de los tres tipos de shaders de forma unificada.

Pixel shaders: Esta técnica utilizada principalmente en 2D, calcula la posición, color, sombra, etc. de cada píxel de forma particular, pudiendo generar una única o, en aplicaciones más complejas, más de una salida a partir de esa única entrada. En el caso del 3D, al ser la entrada un único fragmento, no queda otra que obtener los píxeles de alrededor del fragmento de entrada para poder producir el efecto requerido.

Vertex shader: Este es el shader 3D más utilizado, transforma la posición 3D de cada vértice para manipularla en 2D. La salida de este proceso es utilizada en el geometry shader.

Geometry shader: Se toma como entrada la primitiva generada por el vertex shader, las procesa y genera como salida cero o más primitivas, dando lugar al pixel shader.

Visite https://medium.com/trabe/a-brief-introduction-to-webgl-5b584db3d6d6 y agregue a su informe información sobre el pipeline de renderizado. En particular busque relacionarlo con el artículo anterior.

**Consideración: Corto y conciso.**

El pipeline de renderizado es una secuencia de etapas para renderizar gráficos 3D. La GPU es un procesador altamente paralelo, es decir que las etapas del proceso se ejecutan simultáneamente en las unidades de procesamiento de la GPU.

Divida los 6 pasos del pipeline en aquellos que corresponden al procesamiento 3D, los que corresponden al 2D.

Etapas del renderizado que corresponde al procesamiento 3D:

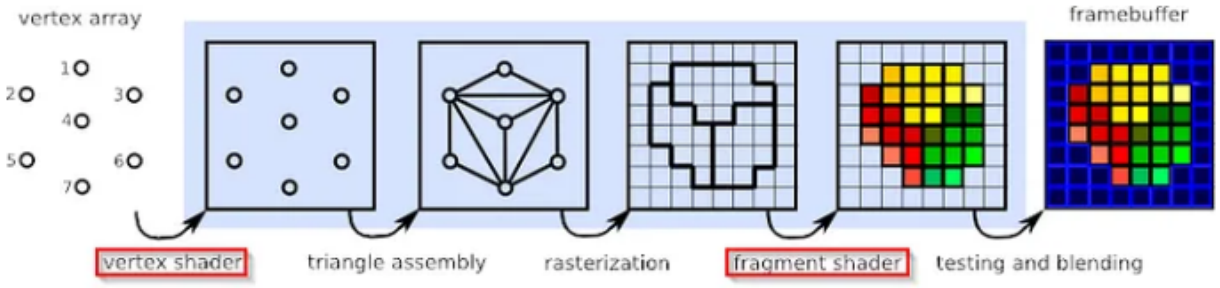
1. Vertex shader
2. Triangle assembly
3. Rasterization

Toda esta etapa implica el procesamiento de vectores de un gráfico dado, generando un único vector para la secuencia de vectores de salida. Aquí se aplican rotaciones, traslaciones o escalamiento de vectores.

Etapas del renderizado que corresponden al procesamiento 2D:

4. Fragment Shader
5. Testing and blending

Se procesa un fragmento generado por la etapa de Rasterización en un conjunto de colores y un único valor de profundidad, convirtiéndolo en pixeles.




Visite https://en.wikipedia.org/wiki/Video_post-processing y agregue a su informe conceptos básicos de post-processing, ¿en que etapa del pipeline se ejecutan?

**Consideración: Corto y conciso.**

El post-procesamiento son los efectos aplicados a la imagen final luego de que haya sido renderizada en el framebuffer, como puede ser:

- Motion blur: Desenfoque por movimiento rápido
- Color correction: Ajusta el balance de color, brillo y contraste
- Grayscale: Aplica escala de grises
- Blom: Añade brillo adicional

Todo esto se aplica una vez que la imagen ya se encuentra almacenada en el framebuffer, obteniendo así la presentación final en nuestra pantalla

Diríjase a la web https://www.shadertoy.com la cual nos permite programar de forma interactiva shaders gráficos que corren en GPU gracias al uso de WebGL. Toque en la opción “Nuevo” arriba a la derecha, amplíe las “Entradas del shader”, agregue a su informe un listado de las entradas posibles indicando su tipo, nombre y una descripción breve de qué representa cada una

Listado de entradas posibles:

- iResolution
  - Tipo: uniform vec3
  - Resolución de la ventana en pixeles
- iTime
  - Tipo: uniform float
  - Tiempo de reproduccion del shader, en segundos
- iTimeDelta
  - Tipo: uniform float
  - Tiempo de renderizado en segundos
- iFrameRate
  - Tipo: uniform float
  - Velocidad de fotogramas del shader

- iFrame
  - Tipo: uniform int
  - Cuadro de reproduccion (frame) del shader
- iChannelTime[4]
  - Tipo: uniform float
  - Tiempo de reproduccion del canal, en segundos
- iChanellResolution[4]
  - Tipo: uniform vec3
  - Resolucion del canal, en pixeles
- iMouse
  - Tipo: uniform vec4
  - Coordenadas de pixeles del mouse. Es XY por defecto, pero cambia a zw cuando se hace clic
- iChaneel0..3;
  - Tipo: uniform samplerXX
  - Canal de entrada (en 2D)
- iDate
  - Tipo: uniform vec4
  - Año, mes, dia y hora en segundos
- iSampleRate
  - Tipo: uniform float
  - Frecuencia de muestreo de sonido (44100)

.

Diríjase a https://www.shadertoy.com/howto y agregue a su informe un listado de las salidas posibles de los Pixel Shaders así como su tipo y una breve descripción de qué representa cada una.

Listado de salidas posibles:

- Shaders de imagen:
  - Se generan imagenes de procedimiento calculando un color para cada pixel
  - Prototipo: void mainImage( out vec4 fragColor, in vec2 fragCoord)
    - Esta funcion se ejecuta una vez por pixel.
    - fragCoord contiene las coordenadas de pixeles para los cuales el shader necesita calcular un color
    - El color resultante se reune en fragColor como un vector de cuatro componentes, mostrando el resultado en pantalla
- Shaders de sonido:
  - Generacion de sonido usando los shaders GLSL
  - Prototipo: vec2 mainSound( float time)
    - el Time indica el tiempo en segundos de la muestra de sonida para la cual se debe calcular la amplitud de la onda, que se muestreo a una frecuencia que sera de 44100 o 48000
- Shaders VR
  - Generacion de imagenes para medios de realidad virtual (VR)

- Prototipo: void mainVR(out vec4 fragColor, in vec2 fragCoord, in vec3 fragRayOri, in vec3 fragRayDir)
  - fragCoord y fracColor funciona igual que en los shaders 2D: contiene las coordenadas de los pixeles en el espacio de la superficie y el color del pixel de salida
  - fragRayOri y fragRayDir tienen el origen del rayo y la direccion que pasa por ese pixel en el mundo virtual. Se espera que el shader transforme esos valores en espacio de camara si se produce algun movimiento

Cuando crea un nuevo shadertoy el código de ejemplo que le sugiere la web es el siguiente:

```
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
  // Normalized pixel coordinates (from 0 to 1)
  vec2 uv = fragCoord/iResolution.xy;

  // Time varying pixel color
  vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));

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

Con apoyo de internet o lo que considere, explique en profundidad cada parte de este shader “hello world”. Debe explicar como mínimo:

- Que representa uv,

uv es un vector de 2 componentes que representa las coordenadas normalizadas del píxel actual.

El cálculo que se realiza para uv es la división entre fragCoord, que son las coordenadas del píxel actual en la pantalla, y iResolution, que es un vector de dos componentes que representa la resolución de la pantalla. De esta forma, se obtienen coordenadas normalizadas en el rango de 0 a 1.

- Porque es necesario trabajar en uv y no en xy,

Se trabaja con uv y no en xy, ya que se aplica una transformación a un sistema de coordenadas dado en otro sistema de coordenadas que va del 0 a 1, para poder representarlo.

- Cómo se logra que el resultado sea una animación siendo que las entradas son estáticas,

Se logra que el resultado sea una animación a través de iTime, que es un variable de entrada que se actualiza automáticamente a medida que pasa el tiempo desde que se ejecutó el shader.

- Cómo es posible que col sea de tipo vec3 siendo que esta igualado a una operacion aritmetica a priori entre flotantes,

Es posible que col sea de tipo vec3 (vector de tres componentes) ya que la expresión se evalúa componente por componente.

- Cuales son los constructores posibles para vec4, que representan los componentes de fragColor, uv se presenta como vec2 pero se utiliza su propiedad xyx, ¿que es eso? ¿Qué otras propiedades tiene vec2? ¿y vec3? ¿y vec4?

Al ser un vec4 o vector de 4 componentes, se puede construir de distintas formas:

- vec4(float x, float y, float z, flot w) → Usando 4 valores flotantes
- vec4(vec3 v, float x) → Usando un vector de 3 componentes y 1 flotante
- vec4(vec2 v, float x, float y) → Usando un vector de 2 componentes con 2 flotantes
- vec4(vec2 v1,vec v2) → Usando dos vectores de 2 componentes

Los componentes de fragcolor representan el color del fragmento o pixel resultante, que se consigue de la siguiente manera:

- r (rojo)

- g (verde)
- b (azul)
- a (alfa, o transparencia)

En este caso, el rgb se forma con el resultado de col, y el alfa va a ser 1 (no hay transparencia)

Al presentar uv como vec2 y utilizar su propiedad xyx, lo que significa es que se utilizan los componentes rojo,verde y rojo nuevamente. Osea, se crea un vec3 con componentes x-y-x, o r-g-r

Al igual que con vec2, se puede obtener valores xyz para vec3, y xyzw para vec4, y la posibilidad de hacer “Swizzling”, o reordenamiento y duplicacion de componentes, como puede ser vec3.xyx o vec4.xyzx

**Hit #2**

Vea el video de Inigo Quilez https://www.youtube.com/watch?v=0ifChJ0nJfM donde utilizando matemáticas, trigonometría, shaders y mucha creatividad dibuja una palmera en la playa.

Documente el video de forma somera, ¿qué conclusiones saca al respecto?

**Consideración: Corto y conciso.**

El video “The basics of Painting with Maths” explica cómo usar matemáticas y código para generar imágenes, mostrando con ejemplos prácticos en Shadertoy utilizando funciones para crear efectos visuales complejos. Tal es así que con estas funciones llega a dibujar esta imagen de una palmera con un fondo gradiente solo con 16 líneas de código:

Código documentado:

```
void main(void)
{
    //Normaliza las coordenadas de los fragmentos (puntos) de la pantalla
    vec2 p = gl_FragCoord.xy / iResolution.xy;

    // Define punto de referencia
    vec2 q = vec2(0.33, 0.7);

    // Define el color base usando dos colores anaranjados
    vec3 col = mix(vec3(1.0, 0.3, 0.1), vec3(1.0, 0.8, 0.3), sqrt(p.y));

    // Calcula un radio dinámico
    float r = 0.2 + 0.1 * cos(atan(q.y, q.x) * 10.0 + 20.0 * q.x + 1.0);

    // Ajusta el color utilizando una función de suavizado
    col *= smoothstep(r, r + 0.01, length(q));

    // Modifica el radio con función de coseno y exponencial
    r = 0.015;
    r += 0.002 * cos(120.0 * q.y);
    r += exp(-40.0 * p.y);

    // Ajusta el color nuevamente usando funciones de suavizado basadas en la posición q
    col *= 1.0 - (1.0 - smoothstep(r, r + 0.002, abs(q.x - 0.25 * sin(2.0 * q.y)))) * (1.0 - smoothstep(0.0, 0.1, q.y));
```

```
    // Asigna el color final
    gl_FragColor = vec4(col, 1.0);
}
```

Salida generada:

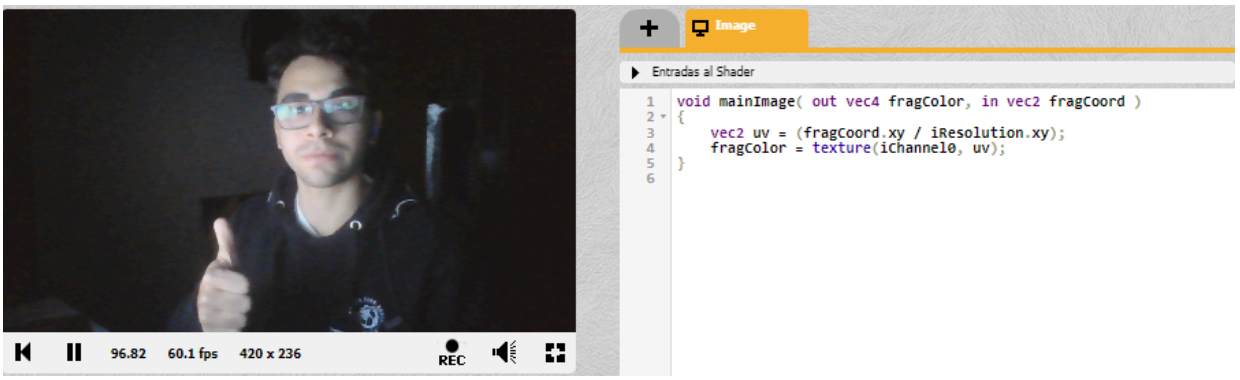
Como conclusión obtenemos que el uso de matemáticas y programación puede ser una herramienta poderosa para generar efectos visuales creativos en la creación de imágenes. A pesar de ser una manera quizá muy engorrosa, pero también ingeniosa, destacamos la importancia de entender y aplicar conceptos matemáticos en estos entornos.

**Hit #3**

Hora de ensuciarse las manos, utilizando shadertoy seleccione en iChannel0 una fuente de textura para poder continuar esta guía, puede ser una imagen de ejemplo, un video de ejemplo o para hacerlo más entretenido, su cámara web.

El siguiente shader muestra de forma trivial como copiar los pixeles desde el iChannel0 a la salida:

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

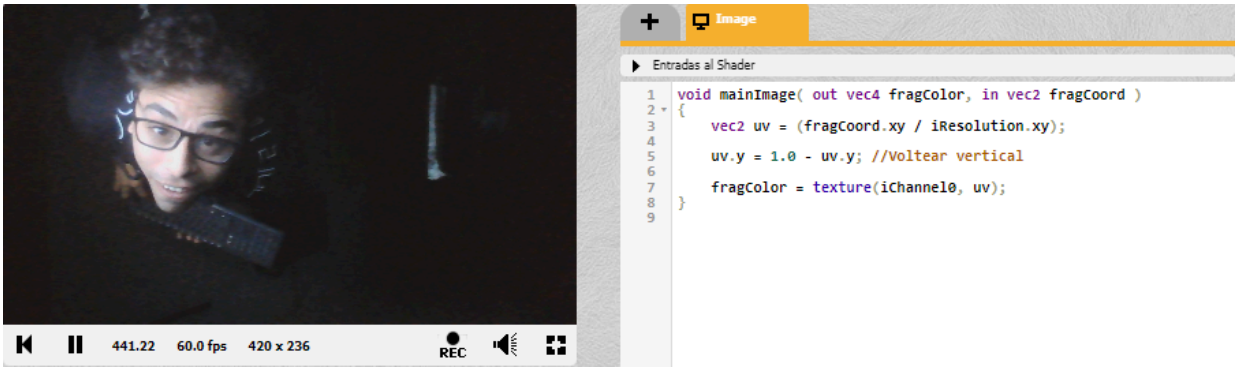


## Hit #4

Utilizando como punto de partida el **HIT#3** y sin perder de vista lo que documentó en los hits anteriores, **modifique el código para poner la imagen cabeza abajo, es decir aplicar lo que comúnmente se llama un efecto de FLIPY o voltear vertical. Luego haga el FLIPX o voltee horizontal (también llamado espejo).**

### Voltear vertical (FLIPY)

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

   uv.y = 1.0 - uv.y; //Voltear vertical

   fragColor = texture(iChannel0, uv);
}
```




### Voltear horizontal (FLIPX - Espejo)

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

    uv.x = 1.0 - uv.x; //Voltear horizontal

    fragColor = texture(iChannel0, uv);
}
```


Si no lo había hecho previamente, amplíe su informe sobre la potencialidad de UV.

**Consideración: Corto y conciso**

Al utilizar UV, estas coordenadas se encuentran normalizadas entre 0 y 1, logrando que los shaders funcionen de manera consistente en cualquier resolución de pantalla sin necesidad de ajustes adicionales. Además de ello, me permite tener gráficos 3D de diferentes texturas (aplicando en nuestro caso la cámara web), lograr animaciones, efectos visuales como gradientes o patrones de ruido, etc.

**Hit #5**

Tomando como base el ejemplo anterior, en iChannel1 agregue una fuente de textura de video, puede usar el de ejemplo de Britney Spears. Quite los efectos flips y muestre la nueva textura en lugar de la de iChannel0.


Va a implementar ahora un filtro chroma básico, su objetivo es cambiar el color de los píxeles verdes del fondo por el video proveniente de iChannel0 (su webcam) para poder bailar como malocorista detrás de Britney Spears (el baile es opcional pero altamente recomendable).
Para implementar un filtro chroma primero necesitará recordar algunos conceptos matemáticos en particular el que proviene de la geometría, y es la distancia pitagórica en N dimensiones (si, la fórmula esa del triángulo rectángulo pero en N dimensiones).
Vamos pasito a pasito como decía mostaza merlo hasta poder unir todas la piezas (https://www.youtube.com/watch?v=cGBvmbtpLcE).
En primer lugar tiene que definir el color del chroma dentro de un vec4.
Luego tiene que definir un umbral de chroma dentro de un float.
Luego tiene que calcular la distancia entre el color del iChannel1 y el color del chroma con el teorema de distancia pitagórica en n dimensiones, siendo n=3.
Finalmente elegir si mostrar la textura de un canal u otro en funcion de si la distancia supera o no el umbral establecido.
Modifique el umbral a diferentes valores. Analice los resultados. Documente la experiencia. Haga screenshots bailando como malocorista con el efecto chroma (opcional).

Video documentado la experiencia + interaccion con Van Damme: https://youtu.be/04uBFfXM-PQ

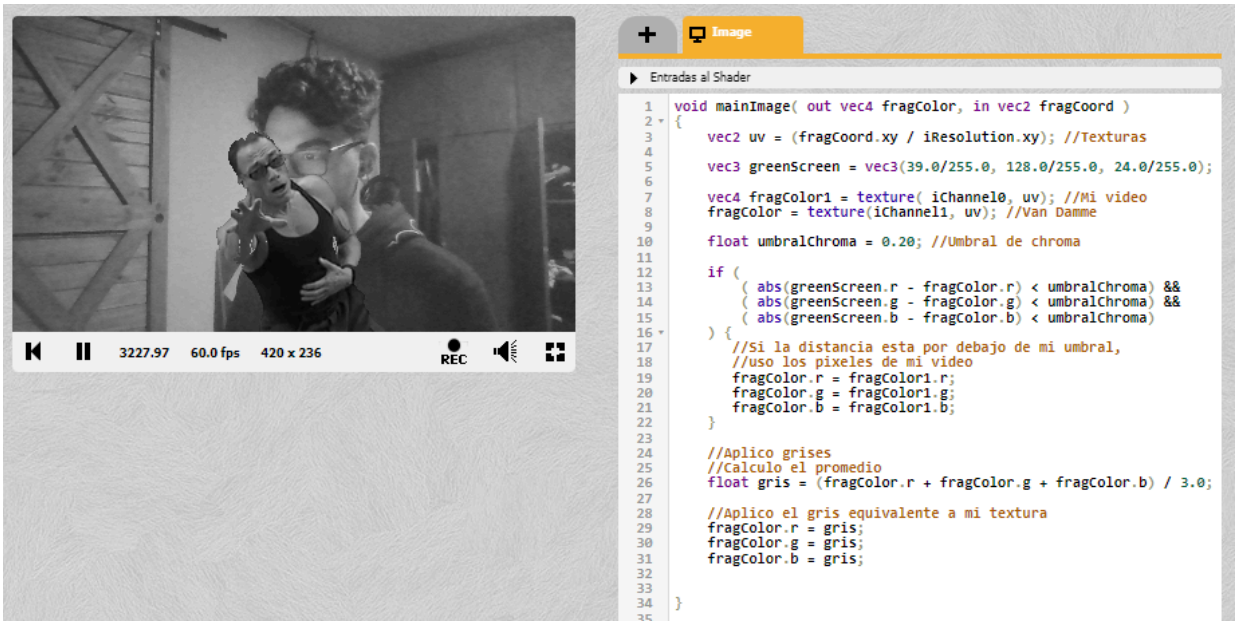
```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = (fragCoord.xy / iResolution.xy); //Texturas

        vec3 greenScreen = vec3(39.0/255.0, 128.0/255.0,
24.0/255.0); //Guardo el verde
```

```
    vec4 fragColor1 = texture( iChannel0, uv); //Mi video
    fragColor = texture(iChannel1, uv); //Van Damme

    float umbralChroma = 0.20; //Umbral de chroma

    if (
        ( abs(greenScreen.r - fragColor.r) < umbralChroma) &&
        ( abs(greenScreen.g - fragColor.g) < umbralChroma) &&
        ( abs(greenScreen.b - fragColor.b) < umbralChroma)
    ) {
       //Si la distancia esta por debajo de mi umbral,
       //uso los pixeles de mi video
       fragColor.r = fragColor1.r;
       fragColor.g = fragColor1.g;
       fragColor.b = fragColor1.b;
    }
}
```

**Hit #6**

Modifique el programa para aplicar un filtro de escala de grises acorde a https://en.wikipedia.org/wiki/Grayscale, luego documente los cambios realizados.

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = (fragCoord.xy / iResolution.xy); //Texturas

    vec3 greenScreen = vec3(39.0/255.0, 128.0/255.0,
24.0/255.0); //Guardo el verde

    vec4 fragColor1 = texture( iChannel0, uv); //Mi video
    fragColor = texture(iChannel1, uv); //Van Damme

    float umbralChroma = 0.20; //Umbral de chroma

    if (
        ( abs(greenScreen.r - fragColor.r) < umbralChroma) &&
        ( abs(greenScreen.g - fragColor.g) < umbralChroma) &&
        ( abs(greenScreen.b - fragColor.b) < umbralChroma)
    ) {
       //Si la distancia esta por debajo de mi umbral,
       //uso los pixeles de mi video
       fragColor.r = fragColor1.r;
       fragColor.g = fragColor1.g;
       fragColor.b = fragColor1.b;
    }

    //Aplico grises
    //Calculo el promedio
    float gris = (fragColor.r + fragColor.g + fragColor.b) /
3.0;

    //Aplico el gris equivalente a mi textura
    fragColor.r = gris;
    fragColor.g = gris;
    fragColor.b = gris;
}
```




```glsl
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = (fragCoord.xy / iResolution.xy); //Texturas

    vec3 greenScreen = vec3(39.0/255.0, 128.0/255.0, 24.0/255.0);

    vec4 fragColor1 = texture( iChannel0, uv); //Mi video
    fragColor = texture(iChannel1, uv); //Van Damme

    float umbralChroma = 0.20; //Umbral de chroma

    if (
        ( abs(greenScreen.r - fragColor.r) < umbralChroma) &&
        ( abs(greenScreen.g - fragColor.g) < umbralChroma) &&
        ( abs(greenScreen.b - fragColor.b) < umbralChroma)
    ) {
       //Si la distancia esta por debajo de mi umbral,
       //uso los pixeles de mi video
       fragColor.r = fragColor1.r;
       fragColor.g = fragColor1.g;
       fragColor.b = fragColor1.b;
    }

    //Aplico grises
    //Calculo el promedio
    float gris = (fragColor.r + fragColor.g + fragColor.b) / 3.0;

    //Aplico el gris equivalente a mi textura
    fragColor.r = gris;
    fragColor.g = gris;
    fragColor.b = gris;


}
```

Para la aplicación de un filtro de grises, solamente es necesario obtener el promedio de cada pixel del fragColor, y aplicar a cada componente de nuestra salida.

**Hit #7**

Realice un nuevo programa que tomando como iChannel0 su webcam aplique el efecto Sobel https://es.wikipedia.org/wiki/Operador_Sobel sobre la misma. Explique porque, al menos con lo que sabe hasta ahora, no podría aplicar el operador de sobel sobre el código anterior donde aplicaba el chroma pero si pudo aplicar el de escala de grises.

**HIT #8**

Agregue un framebuffer y coloque el código de chroma dentro de él. Luego agregue el código de sobel dentro del shader principal. Seleccione como entrada del shader principal el BufferA previamente agregado.
Analice lo sucedido, realice un gráfico del pipeline de renderizado donde se muestre que es lo que está pasando en este caso puntual. ¿A cuantos FPS corre?

**HIT #9**

Agregue otro framebuffer con el resultado de la máscara de sobel y escriba un shader que haga un zoom de 2x sobre el centro del resultado de sobel.
Ahora haga que sea sobre la esquina superior izquierda. Y ahora sobre la inferior derecha. Documente la experiencia.

**Hit #10**

Durante mucho tiempo, antes de que surgiera CUDA, los programadores solo podíamos hacer uso de las capacidades de la GPU mediante el pipeline gráfico que usted está utilizando ahora. Esto nos forzaba a tener que representar gráficamente cosas que no son gráficas para poder utilizarlas.
Volvamos al código de base del **Hit #3**, ponga como iChannel0 al keyboard. Haga click sobre la previsualizacion (inicialmnte en negro) y comience a precionar teclas. Experimente. ¿Observa las 3 filas y las 256 columnas?
¿Qué cree que son? Tome como iChannel1 su cámara y haga que cuando se apreta la tecla “g” se active el filtro de escala de grises sobre iChannel1, cuando esta se suelta el filtro debe quedar activado hasta que se presione nuevamente.
Haga que cuando se presione la letra “s” se aplique un filtro de sobel sobre iChannel1 pero que al soltar la tecla este filtro desaparezca.
¿Qué conclusiones puede sacar sobre esta forma de interactuar con los shaders?