

Trabajo Práctico N° 4

Programación Paralela (Shaders)

Sistemas Distribuidos y Programación Paralela -
Universidad Nacional de Luján

Fecha de Entrega: 31/05/24

Grupo 4 - Integrantes:

- Puyelli Thiago
- Herrneder Matías

Hit #1

Los shaders son programas hechos para correr en la unidad de procesamiento gráfico (GPU) y tienen como función modificar cómo se representan los gráficos en pantalla. Existen distintos tipos de shaders según la etapa del pipeline gráfico en la que actúan, y cada uno cumple una función específica.

2D

Los shaders 2D operan sobre imágenes digitales,. Su función principal es modificar atributos de los píxeles, por ejemplo, para alterar su color, transparencia o iluminación. En la actualidad, el único tipo de shader 2D es el *pixel shader*.

Los **Pixel Shaders** vienen de lo que hablamos en los 2D ya que son los shaders 2D que se usan hasta la actualidad, el cual permite generar una diversa cantidad de efectos visuales con la posibilidad de poder alterar los píxeles como por ejemplo generar sombras o translucidez. Además, tienen acceso a las coordenadas de los píxeles, el cual podría utilizarse para desenfoque o detección de bordes.

3D

Los **Vertex Shaders** sobre cada vértice de un modelo 3D, transformando su posición 3D a coordenadas 2D en pantalla. También pueden modificar atributos como el color o las coordenadas de textura. No generan nuevos vértices.

Los **Geometry Shaders** generar nuevas primitivas gráficas (puntos, líneas, triángulos) a partir de las primitivas originales. Se ejecutan después del *vertex shader* y pueden modificar la complejidad geométrica de una escena.

Los **Tessellation Shaders** añaden detalle a los modelos 3D subdividiendo sus meshes en tiempo real, en función de parámetros como la distancia a la cámara permitiendo que se logre manejar la distancia sin que se pierda calidad visual.

Los **Primitive Shaders**, similares a los compute shaders pero con acceso a los datos necesarios para procesar geometría. Nvidia introdujo los mesh y task shaders permitiendo al GPU manejar algoritmos más complejos y aliviando la carga del CPU.

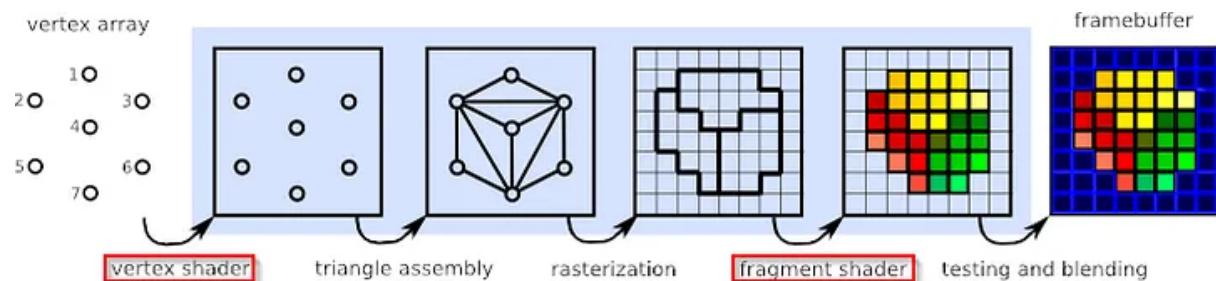
Los **Computers Shaders** si bien no están limitados al ámbito gráfico, pueden usarse para tareas como simulaciones físicas o cálculos de iluminación. Se ejecutan en paralelo en la GPU y comparten recursos con el pipeline gráfico.

WebGL

WebGL es una API utilizada para crear gráficos 3D en navegadores web. Las principales características son:

- _ Multiplataforma y multi browser: Funciona en diferentes sistemas operativos y navegadores web.
- _ Aceleración 3D por GPU: Utiliza la potencia de la GPU para renderizar gráficos 3D de manera eficiente.
- _ API nativa con soporte para GLSL: Permite escribir shaders personalizados utilizando GLSL.
- _ Funciona dentro de un elemento canvas: Los gráficos se renderizan dentro de un elemento <canvas> en HTML.
- _ Integración con interfaces DOM: Se puede integrar con otras partes del Document Object Model (DOM) de la página web.

Para poder funcionar, hace el proceso de renderizar el pipeline, con el objetivo de poder renderizar en 3D, donde para ello funciona muy bien con la GPU ya que gracias al procesamiento paralelo se puede lograr con una gran eficiencia. Para lograr este pipeline se pasan por 6 etapas, con esta imagen se pueden visualizar.



Las etapas de vertex shader y fragment shader son las que explicamos anteriormente, y pueden ser programados por nosotros.

Las tres primeras etapas involucran un espacio 3D. La etapa de Vertex Array definen de los vértices en un espacio 3D, luego Vertex Shader aplica una transformación a los vértices en ese espacio 3D. Después Triangle Assembly ensambla de los vértices en triángulos que conforman la geometría 3D. La Rasterization convierte los triángulos 3D en fragmentos 2D, es decir en pixeles. Fragment Shader procesa cada fragmento y determina su color y otros atributos. Por último el Testing and Blending realiza pruebas y mezclas de colores para determinar el color final de cada píxel en el framebuffer.

Post-processing

Se trata de métodos de procesamiento de imágenes con el fin de lograr efectos y una calidad de imagen mayor, donde antes de renderizar objetos 3D en la pantalla, primero se renderiza en una memoria buffer en la GPU, luego se usan pixel y vertex shaders para aplicarle un filtro post-procesado al buffer antes de mostrarlo.

Shadertoy

Parámetros posibles de entrada para un shader

- iResolution: Vector de 3 valores que establecen en pixeles la resolución del viewport
- iTIME: Valor flotante que lleva el tiempo de reproducción del shader en segundos.
- iTIMEDELTA: Valor flotante que establece en segundos el tiempo de renderizado entre frames consecutivos
- iFrameRate: Valor flotante que establece la tasa de frames del shader por segundo (fps)
- iFrame: Valor entero que cuenta cuantos frames se renderizaron desde el inicio
- iChannelTime[4]: Valor flotante que representa el tiempo de reproducción de cada canal en segundos. Puede haber hasta 4 canales
- iChannelResolution[4]: Vector de 3 posiciones que representa la resolución de cada canal en pixeles
- iMouse: Vector de 4 posiciones que representan las coordenadas del mouse y si se hizo click
- iChannel0..3: De tipo samplerXX, donde XX puede ser 2D o Cube. Representan canales de entrada
- iDate: Vector de 4 posiciones que representa el año, mes, día y hora

Salidas posibles

- fragColor (vec4): Es el color final del pixel que se va a renderizar en pantalla (RGBA).
- mainSound (vec2): Esta es la salida del shader de sonido (mainSound()). El vector de dos componentes representa la amplitud de la onda de sonido para los canales izquierdo y derecho de un sistema estéreo.
- fragColor (vec4) en VR: Similar al fragColor de los shaders de imagen, pero específico para VR.

Explicación del código

```

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.yxy+vec3(0,2,4));
}

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

```

Explicación

Este código lo que hace es obtener las coordenadas xy y transformarlas en uv, el cual **uv** trata de pasar los ejes a que vayan de 0 a 1. Luego de esto hace una variación de colores, donde usa **iTime** para que el valor varíe conforme pasa el tiempo, luego **uv.yxy** crea un **vec3** con **uv.x**, **uv.y** y **uv.x**, por último hace un **vec3** donde asigna los colores 0, 2 y 4, para que cada color no tenga el mismo valor, generando que sean distantes y se vea más clara la rotación entre rojo, verde y azul, además de los otros colores que se pueden apreciar. Modificando esos colores también se pueden hacer otros efectos con otros colores. Por último retorna el resultado. Cuestiones a explicar:

Qué representa uv

está explicado antes, transforma x e y, en su equivalente yendo de 0 a 1, donde si hablamos de x, 1 equivaldría al valor máximo de acuerdo a la resolución de la pantalla.

Porque es necesario trabajar en uv y no en xy

Es necesario ya que uv permite trabajar con cualquier resolución dando el mismo resultado, en cuanto se trabaje directamente con xy se generarían inconsistencias visuales.

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

Se logra una animación con lo que explicamos de **iTime** donde hace que conforme el valor que devuelva el tiempo varía el resultado.

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

Lo que sucede es que el lenguaje al hacer una operación con vec3, termina convirtiendo esos valores flotantes en vec3 por lo tanto termina haciendo que sea un resultado en vec3.

_ 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?

En si las propiedades de vec2, vec3 y vec4 son estas:

vec2: r, g / x, y

vec3: r, g, b / x, y, z

vec4: r, g, b, a / x, y, z, w

En sí estas propiedades son los valores en fin a asignar en caso de constructores, el cual se pueden hacer de diferentes maneras, como por ejemplo vec4(vec3 xyz, float w) o vec4(float x, float y, vec2 zw), pero en sí todo gira en torno a asignar estas propiedades, donde en el caso de rgb se trata de los colores bustamente rojo, verde y azul, luego a de alfa, es decir la opacidad del color, y luego x, y, z, w, son coordenadas espaciales. Luego eso de utilizar la propiedad xyx, se trata de que uno puede hacer uso del vector y armar otros vectores repitiendo valores, es decir si yo tengo este código:

```
vector2 = vec2(0, 2);
vector3 = vec3(vector2.xy);
```

Lo que estoy haciendo es crear vector3 de esta manera vec3(0, 2, 0), de esta manera shader toy es muy flexible a la hora de acceder a las propiedades que maneja. Luego faltó mencionar que fragColor, sería un vec4 que devuelve la respuesta final de como se representa el pixel.

Hit #2

En cuanto al video de Inigo Quilez, podemos concluir que el video muestra las posibilidades que tienen estas herramientas y el potencial que se les puede sacar gracias a un buen uso de las matemáticas, donde se ve como hace uso de funciones como lo es el coseno, seno y exponencial, donde algo que hace muy interesante es que el piso de la palmera lo pensó con una función exponencial, y luego jugó mucho cambiando valores demostrando que teniendo una buena abstracción y un buen uso de conceptos matemáticos realmente las posibilidades visuales son prácticamente infinitas.

Hit #3

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

El código toma como base la textura en iChannel0 simplemente la convierte en la salida sin hacer otras transformaciones.

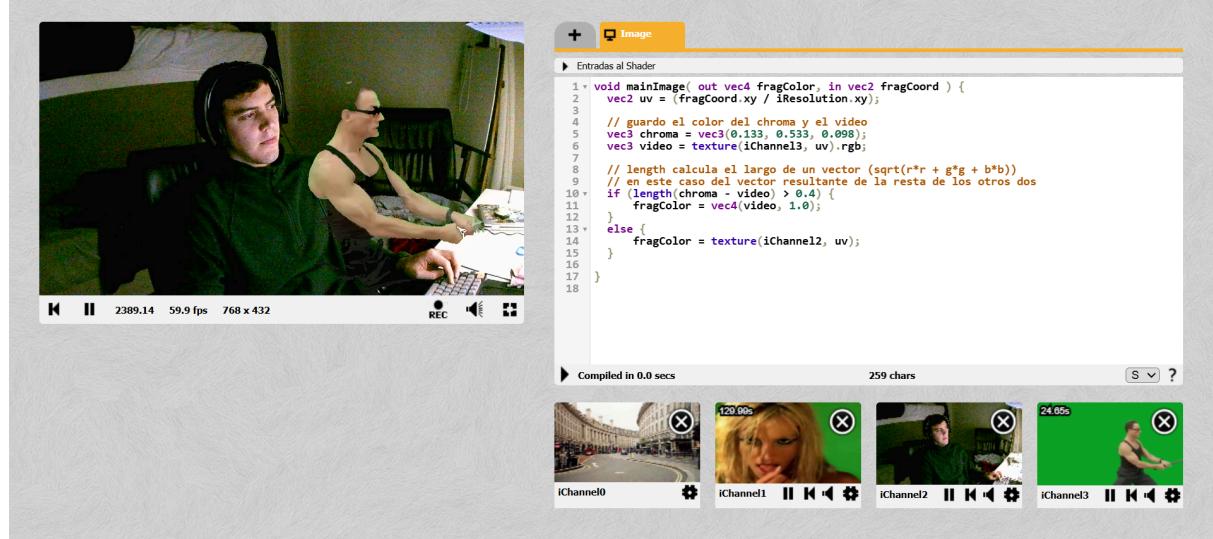
Hit #4

```
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {  
    vec2 uv = (fragCoord.xy / iResolution.xy);  
    fragColor = texture(iChannel0, 1.0 - uv);  
}
```

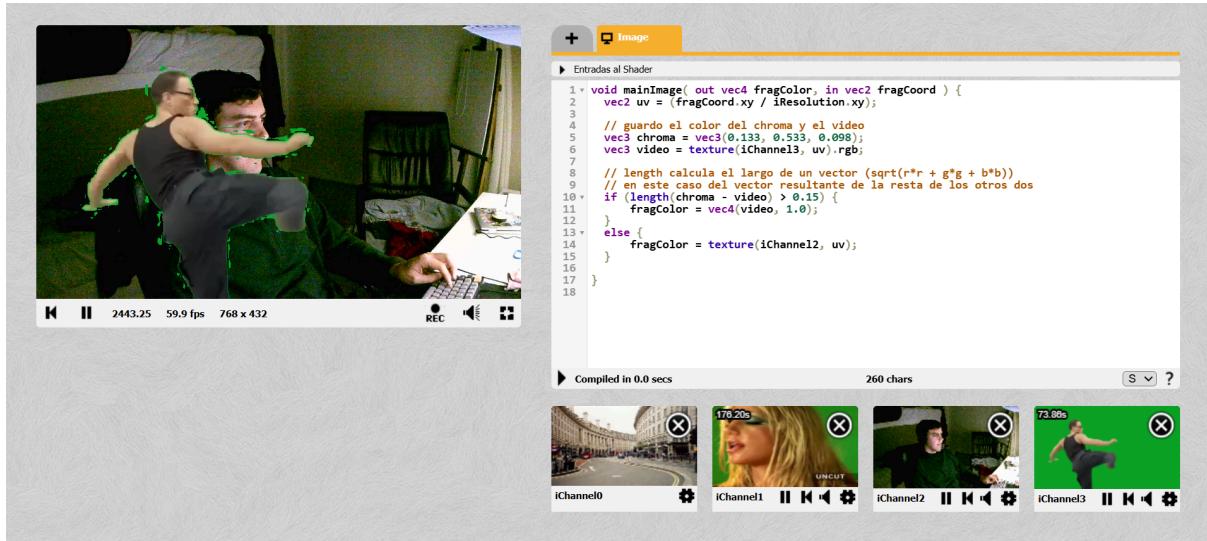
El código toma como base la textura en iChannel0 y a la hora de hacer la salida, mapea para cada dimensión de uv y toma su opuesto haciendo $\{1.0 - uv\}$, ya que uv puede tomar valores entre 0 y 1.

Hit #5

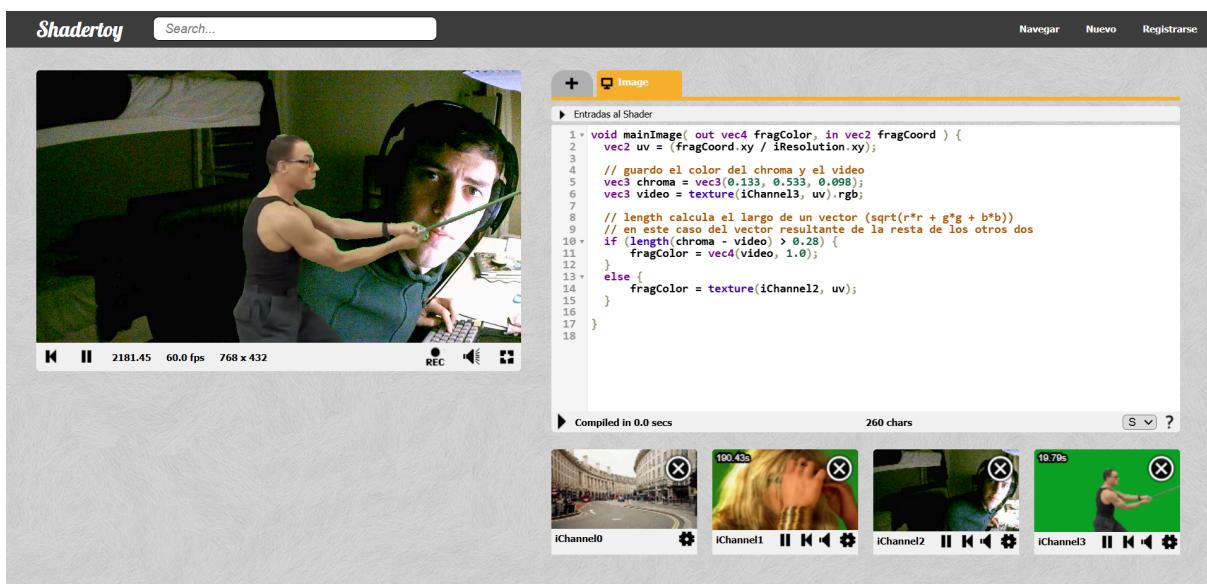
```
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {  
    vec2 uv = (fragCoord.xy / iResolution.xy);  
  
    // guardo el color del chroma y el video  
    vec3 chroma = vec3(0.133, 0.533, 0.098);  
    vec3 video = texture(iChannel3, uv).rgb;  
  
    // length calcula el largo de un vector (sqrt(r*r + g*g + b*b))  
    // en este caso del vector resultante de la resta de los otros dos  
    if (length(chroma - video) > 0.28) { // UMBRAL DE CHROMA  
        fragColor = vec4(video, 1.0);  
    }  
    else {  
        fragColor = texture(iChannel2, uv);  
    }  
}
```



umbral = 0.4



umbral = 0.15



umbral = 0.28 (ganador)

Luego de hacer algunas pruebas, tomamos 0.28 como umbral del chroma, ya que valores más altos hacen desaparecer el video del frente, y valores más bajos dejan entrever el chroma demasiado.

Hit #6

```
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {  
    vec2 uv = (fragCoord.xy / iResolution.xy);  
  
    // guardo el color del chroma y el video  
    vec3 chroma = vec3(0.133, 0.533, 0.098);  
    vec3 video = texture(iChannel3, uv).rgb;  
  
    // length calcula el largo de un vector (sqrt(r*r + g*g + b*b))  
    // en este caso del vector resultante de la resta de los otros dos  
    vec4 tx;  
  
    if (length(chroma - video) > 0.28) { // UMBRAL DE CHROMA  
        tx = vec4(video, 1.0);  
    }  
    else {  
        tx = texture(iChannel2, uv);  
    }  
  
    // AGREGAMOS GRayscale  
    float grayscale = (0.2126*tx.r) + (0.7152*tx.g) +  
(0.0722*tx.b);  
  
    // mostramos la salida  
    fragColor = vec4(grayscale);  
}
```

Agregamos al final un filtro de gris, lo que hace el filtro es tomar la variable tx, que puede contener la imagen de fondo o bien el video del frente y le aplica el filtro. Para aplicar el filtro transformamos un color RGB a un único valor, compuesto del 21,26% de ROJO, 71,52% de VERDE y 07,22% de AZUL, la razón por la que no se hace un promedio de los tres colores se debe a que estos valores resultan más apropiados para la creación de un filtro en escala de grises por razones biológicas de la interpretación humana del color.