Trabajo Práctico Nº 5
Programación Paralela (Shaders)
Grupo 4

HIT #1:

¿Qué es un shader?
Los shaders se tratan de software especializado en calcular niveles apropiados de luz, oscuridad y color durante el renderizado de escenas 3D, proceso llamado shading.

Tipos de shaders 2D
Los shaders 2D actúan sobre imágenes y son llamados texturas.

- Pixel shaders: se encargan de computar el color y otros atributos de cada fragmento, donde una unidad de renderizado afecta a un único píxel de salida. Pueden ir desde simplemente devolver siempre el mismo color, hasta también aplicar diferentes valores de iluminación, sombras, reflejos, translucidez, etc.

Tipos de shaders 3D
Estos actúan sobre modelos 3D, y también acceden a los colores y texturas utilizados para dibujar el modelo o malla.

- Shaders de vértice: Los shaders de vértice son el tipo de shader más común y se ejecutan una vez por cada vértice dado al procesador gráfico. Transforma la posición 3D de cada vértice en el espacio virtual a la coordenada 2D en la que aparece en la pantalla. Los shaders de vértice pueden manipular propiedades como la posición, el color y las coordenadas de textura, pero no pueden crear nuevos vértices.
- Shaders de geometría: Este tipo de shader puede generar nuevos primitivos gráficos, como puntos, líneas y triángulos, a partir de esos primitivos que se enviaron al comienzo del pipeline gráfico. Los usos típicos de un shader de geometría incluyen la generación de sprites de puntos (generar pequeños gráficos 2D (sprites) a partir de puntos individuales en una escena 3D), teselación de geometría (proceso que subdivide una malla en partes más pequeñas para agregar detalle a un modelo 3D, mejorando la apariencia de curvas), extrusión de volúmenes de sombra (utilizado para extender los bordes de los objetos en la dirección de la luz para crear volúmenes que representan sombras) y renderizado de un solo paso a un mapa cúbico (permite renderizar una escena desde seis vistas diferentes (frente, atrás, izquierda, derecha, arriba, abajo) en una sola pasada, generando un mapa cúbico utilizado para reflejos o iluminación ambiental).
- Shaders de teselación: Añade dos nuevas etapas de shader al modelo tradicional: shaders de control de teselación y shaders de evaluación de teselación, que juntos permiten que las mallas más simples se subdividen en mallas más finas en tiempo de ejecución según una función matemática. Esta funcion puede relacionarse con muchas variables, siendo la principal la distancia desde la cámara de visión. Permite

que los objetos cercanos a la cámara tengan detalles finos, mientras que los más alejados pueden tener mallas más toscas, pero parecer comparables en calidad.
- Shaders primitivos y de malla: permiten que la GPU maneje algoritmos más complejos, descargando más trabajo del CPU a la GPU y, en renderizados intensivos en algoritmos, aumentando la tasa de fotogramas.
- Shaders de cómputo: Los shaders de cómputo no están limitados a aplicaciones gráficas, pero utilizan los mismos recursos de ejecución para GPGPU (General-purpose computing on graphics processing units).

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 renderizar gráficos, WebGL sigue una secuencia de etapas conocida como Pipeline de Renderizado. Este pipeline se adapta bien a la GPU, ya que es un procesador altamente paralelo y las etapas del pipeline se ejecutan simultáneamente en las unidades de procesamiento de la GPU.
El pipeline de renderizado de WebGL incluye varias etapas, donde la salida de una etapa se utiliza como entrada para la siguiente. Dos de las etapas más importantes, que son programables, son el Shader de Vértice (transforman las coordenadas de los vértices y pueden aplicar transformaciones como rotaciones, traslaciones y escalado.) y el Shader de Fragmentos (estos shaders se encargan de calcular el color y otros atributos de cada fragmento/píxel).

Ejemplo de renderizado de un triángulo con WebGL:

- Definir un canvas: Crear un elemento <canvas> en HTML donde se renderizarán los gráficos.
- Declarar los vértices: Definir un array con las coordenadas de los tres vértices del triángulo.
- Inicializar el contexto WebGL: Obtener el contexto de WebGL del canvas.
- Definir y compilar los shaders: Escribir y compilar un shader de vértice y un shader de fragmentos.
- Proporcionar los datos de entrada: Crear un buffer con las coordenadas de los vértices y pasarlo al shader de vértice.
- Definir el color: Usar un uniforme para pasar el color al shader de fragmentos.
- Renderizar el triángulo: Limpiar el canvas y dibujar el triángulo usando el método gl.drawArrays.

Imaginando que estamos renderizando una figura en 3D, los pasos que formarían parte del procesamiento 3D sería la declaración de vértices (los cuales pueden ser declarados en un espacio tridimensional), la definición y compilación de los shaders (se transformarían las coordenadas 3D de los vértices en coordenadas 2D en pantalla, pudiendo aplicar transformaciones) y la creación de un buffer con los datos de entrada (trabajaríamos con los vértices e n3 dimensiones).

Post-procesamiento
Se refiere a los métodos de procesamiento de imágenes utilizados en la reproducción de video para mejorar la calidad, así como en el renderizado 3D en tiempo real, como en los videojuegos, para agregar efectos adicionales.
Se utilizan pixel shaders y, opcionalmente, shaders de vértices para aplicar filtros de post-procesamiento.
En el contexto de WebGL y la renderización 3D, el post-procesamiento se realiza en la última etapa del pipeline de renderizado, después de que la escena ha sido renderizada a un buffer en la memoria de la tarjeta gráfica.

Entradas de shader

- iResolution (vec3): resolución en pixeles del viewport.
- iTime (float): tiempo de reproducción del shader en segundos.
- iTimeDelta (float): tiempo de renderizado.
- iFrameRate (float): ratio de cuadros del shader.
- iFrame (int): cuadro de reproducción del shader.
- iChannelTime[4] (float): tiempo de reproducción del canal (segundos).
- iChannelResolution (vec3): resolución del canal en pixeles.
- iMouse (vec4): coordenadas del mouse (xy: actual, zw: click).
- iChannel10..3 (samplerXX): canal de entrada.
- iDate (vec4): año, mes, día, tiempo en segundos.
- iSampleRate (float): sound sample rate.

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.

```
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);
}
```

La función mainImage se llama para cada píxel, y tiene 2 parámetros, fragColor (la salida) y fragCoord (las coordenadas del píxel que se está procesando).

uv se trata de un vector que contiene la división entre las coordenadas del pixel con iResolution.xy, normalizando las coordenadas, es decir, obtenemos unas coordenadas que van del 0 al 1.

El uso de coordenadas normalizadas es útil para no generar dependencias con la resolución, haciéndolo más flexible para todo tipo de pantallas. Además nos facilita hacer matemáticas gracias a que se trabaja en un rango conocido.

col es un vector de 3 coordenadas, donde se realiza un cálculo: “0.5 + 0.5 * coseno()”, este nos permite ajustar el rango entre 0 y 1. Dentro del coseno hay una suma de 3 elementos:
iTime (tiempo transcurrido desde el inicio del shader)
uv.xyx (es un vector que reutiliza uv.x y uv.y para los componentes x, y, z del nuevo vector)
vec3(0,2,4) (añade un desfase diferente a cada componente de color)

Por último, fragColor se establece con el color calculado, donde vec4(col, 1.0) nos devuelve un vector de cuatro componentes donde col es el color RGB y 1.0 es el valor alfa (quedando opaco).

La animación es gracias a la variable iTime, ya que cambia continuamente, permitiendo cambiar los colores en todo momento.

En GLSL, las operaciones aritméticas entre vectores y flotantes están sobrecargadas para funcionar de manera intuitiva:
Las sumas de un flotante a un vector, el flotante se suma a cada componente del vector.
Las multiplicaciones entre un vector y un flotante, cada componente del vector se multiplica por ese flotante.

Los constructores para un vec4 pueden tomar distintas formas:

- cuatro floats
- un vec3 y un float
- dos vec2
- un float (todos los componentes terminan en el mismo valor)

La propiedad xyx de uv es una forma de “swizzling” en GLSL, que reordena y duplica componentes, creando un nuevo vec3 = (x,y,x).

Propiedades de vec2

- x, y: Acceso a los componentes individuales.
- r, g: Acceso a los componentes como colores.
- s, t: Acceso a los componentes como coordenadas de textura.

Propiedades de vec3

- x, y, z: Acceso a los componentes individuales.
- r, g, b: Acceso a los componentes como colores.
- s, t, p: Acceso a los componentes como coordenadas de textura.
- xyz, yzx, zxy, etc.: Swizzling para reordenar o duplicar componentes.

Propiedades de vec4

- x, y, z, w: Acceso a los componentes individuales.
- r, g, b, a: Acceso a los componentes como colores.
- s, t, p, q: Acceso a los componentes como coordenadas de textura.
- xyzw, yzwx, zwxy, etc.: Swizzling para reordenar o duplicar componentes.

HIT #2:

Lo primero que hace, es explicar el uso de gradientes, para luego, con smoothstep, reducir este degradado en dimensiones que el considere. Luego, le da una forma mediante el uso de una función coseno.
Con la función de arco tangente le da el efecto de “caída” a las hojas de la palmera.
Con ayuda de la función smoothstep crea el tronco de la palmera y usando el seno le da un movimiento de caída hacia la derecha al tronco. Para darle una textura indicada vuelve a recurrir a la función del coseno y lo termina con una exponencial que muestra la conexión de la palmera con el suelo.
Mediante el uso de los gradientes nuevamente, colorea el fondo con más detalle, generando que quede un degradado del color naranja hacia un amarillento.
Como conclusión del video, desconocía totalmente lo útiles que pueden ser las matemáticas a la hora de realizar todo tipo de gráficos. El uso de funciones trigonométricas y exponenciales, era algo que no se me cruzó por la cabeza para realizar distintas figuras como lo hizo él. Incluso, me interesé y vi algunos otros contenidos que tiene en su canal, y es increíble la cantidad de modelos tanto 2D como 3D que enseña, solo utilizando fórmulas matemáticas. Es un campo de las ciencias exactas el cual prácticamente desconocía.

HIT #4:

voltear verticalmente la imagen

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

voltear horizontalmente la imagen

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

HIT #5:

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = (fragCoord.xy / iResolution.xy);
    vec4 croma = texture(iChannel1, uv);
    float cr = croma.r;
    float cg = croma.g;
    float cb = croma.b;

    if ((cr <= 0.2) && (cg >= 0.4) && (cb <= 0.2)){
        fragColor = texture(iChannel0,uv);
    } else {
        fragColor = texture(iChannel1, uv);
    }
}
```

Fui cambiando los valores en las diferentes colores, y llegue a varias conclusiones:

- si R y B son menores a 2, no capta el color del croma, al parecer no es un verde puro.
- Si el umbral de G es muy pequeño, dejaremos de ver los bordes de Van Damme en verde, pero a cambio habrán partes de él que serán tomadas como croma (principalmente su pantalón verde oliva).


1982.78 60.0 fps 800 x 450
REC


1952.17 60.0 fps 800 x 450
REC


2090.69 8.3 fps 800 x 450
REC

Espero que entienda las razones por las que no quiero aparecer de fondo bailando (no quiero que Van Damme me mutile un brazo de un katanazo).

HIT #6:

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = (fragCoord.xy / iResolution.xy);
    vec4 croma = texture(iChannel1, uv);
    float cr = croma.r;
    float cg = croma.g;
    float cb = croma.b;

    if ((cr <= 0.2) && (cg >= 0.4) && (cb <= 0.2)){
        float r = texture(iChannel0,uv).r;
        float g = texture(iChannel0,uv).g;
        float b = texture(iChannel0,uv).b;
        float gray = (r + g + b) / 3.0;
        fragColor = vec4(gray,gray,gray,1.0);
    } else {
        float r = texture(iChannel1,uv).r;
        float g = texture(iChannel1,uv).g;
        float b = texture(iChannel1,uv).b;
        float gray = (r + g + b) / 3.0;
        fragColor = vec4(gray,gray,gray,1.0);
        fragColor = vec4(gray,gray,gray,1.0);
    }
}
```

El único cambio que requiere es promediar los 3 colores y conseguir el gris cambiando a cada componente (menos el alpha) por ese promedio.