Taller #4 - Shaders

Jesús Cibeira

Enero 2019

Agenda

- Introducción
- Programación de Shaders
- GLSL
- Ejemplos de Shaders
- Uso de Shaders
- Modelos de sombreado
- Explicación de código

Introducción

GPU (Graphics Processing Unit): es un circuito lógico especializado diseñado para manipular y alterar memoria de manera rápida, de forma de acelerar la creación de imágenes para su despliegue. Orientado originalmente al procesamiento de tareas en gráficos 3D. E.j.: procesamiento de vértices y píxeles, sombreado, mapeado de textura, rasterización, entre otros.

Programación de Shaders

El desarrollo de la programación de GPUs fue muy parecida a la programación de CPUs. Inicialmente la programación se hacía en lenguaje ensamblador específico para cada GPU.

Hoy en día existen diversos lenguajes para programación de shaders, entre ellos están:

- ***Cg*** (C for Graphics – NVidia).
- ***GLSL*** (OpenGL Shading Language).
- ***HLSL*** (High Level Shading Language – Microsoft).

Programación de Shaders

Un shader o sombreador es un programa que realiza cálculos gráficos escrito en un lenguaje de sombreado. Es una tecnología que ha experimentado una rápida evolución destinada a proporcionar al programador una interacción con la unidad de procesamiento gráfico (GPU).

Programación de Shaders

- ***Vertex Shader:*** también conocido como procesador de vértices, se encarga de llevar a cabo todas las operaciones sobre los vértices pertenecientes a los modelos que componen una escena dentro de una aplicación.
- ***Fragment Shader****:* también conocido como procesador de vértices, en este el programador tiene control total sobre cada uno de los fragmentos de la pantalla y toma decisiones acerca de qué color va a tener cada uno de ellos.

Programación de Shaders

Una vez que se haya decidido cuáles etapas se van a manejar, se deben tomar cada uno de estos shaders asociados a dichas etapas escritos en GLSL, para vincularlos con la aplicación, compilarlos y adjuntarlos a un programa que será el que posteriormente enlazaremos dentro de nuestra aplicación, para enviar los datos al pipeline gráfico. La clase encargada de representar al programa se llama ***CGLSLProgram***.

GLSL

- Es un lenguaje usado para crear shaders específicamente creado para trabajar con OpenGL.
- Basado en C.
- Fue originalmente una extensión de OpenGL 1.4.
- A partir de OpenGL 2.0 forma parte del OpenGL (el primer gran cambio en OpenGL desde 1992).

GLSL

OpenGL 1.0	1.0	1.1	1.2	1.3	1.4	1.5	(1992-2003)
OpenGL 2.0	2.0	2.1					(2004-2007)
OpenGL 3.0	3.0	3.1	3.2	3.3			(2008-2010)
OpenGL 4.0	4.0	4.1	4.2	4.3	4.4		(2010-201*)

GLSL

- En la versión 2.0 se añaden el vertex shader y el fragment shader.
- En la versión 3.1 se rompe la compatibilidad hacia atrás.
- En la versión 3.2 se introduce el geometry shader.
- En la versión 4.0 se añaden dos etapas más: teselletion control y teselletion evaluation shader.
- En la versión 4.3 se añade una nueva etapa: compute shader.

GLSL

- El GPU provee el compilador de GLSL en el driver. De esta forma, cuando se desea compilar código GLSL, el driver del GPU se encarga de compilar y generar el código objeto optimizado para ese GPU en particular.
- OpenGL únicamente se encarga de invocar al compilador.
- Una vez compilado el código es transferido al GPU para su ejecución.

GLSL

- Provee todos los operadores de C/C++ sin incluir apuntadores.
- Estructuras de control como if-else, for, do-while, break, continue …
- Es posible implementar funciones, pero no se permite la recursividad.
- Se proveen muchas funciones matemáticas como sin, cos, tan, y otras específicas del GPU como ***texture***.

GLSL

Ya que está basado C/C++ provee los tipos de dato elementales, además de los siguientes:

- ***Vectores:*** vec2, vec3, vec4, ivec2, ivec3, ivec4, bvec2, bvec3, bvec4, dvec2, dvec3, dvec4.
- ***Matrices:*** mat2, mat3, mat4, mat2x3, mat2x4…
- ***Texturas:*** sampler1D, sampler2D, sampler3D, samplerCube, sampler1Dshadow, sampler2Dshadow

Vertex Shader

```
#version 330
#extension GL_ARB_separate_shader_objects : enable

uniform mat4 mProjection, mModelView;

layout(location = 0) in vec4 vVertex;
layout(location = 1) in vec4 vColor;

out vec4 vVertexColor;

void main(){
vVertexColor = vColor;
gl_Position = mProjection * mModelView * vVertex;
}
```

Vertex Shader

- ***#version:*** es una directiva que indica que versión de GLSL requiere el shader. 140 indica la versión 1.40.
- ***#extension:*** indica extensiones que pueden modificar el comportamiento del programa GLSL si es necesario.
- ***uniform:*** indica que el dato no varía para todos los vértices (o fragmentos). En este caso la transformación (projection y modelview) es la misma para todos los vértices.

Vertex Shader

- ***in:*** indica una declaración de un parámetro de entrada. En este caso cada vez que el shader sea invocado se deben pasar coordenadas de mundo en la locación 0 y colores en la locación 1.
- ***out:*** son parámetros que servirán de entrada al próximo shader, en este caso el fragment shader. Los parámetros de cada vértice serán interpolados para cada fragmento.
- ***glPosition:*** es una variable built-in de GLSL que almacena la posición del vértice actual (sólo disponible en el vertex shader).

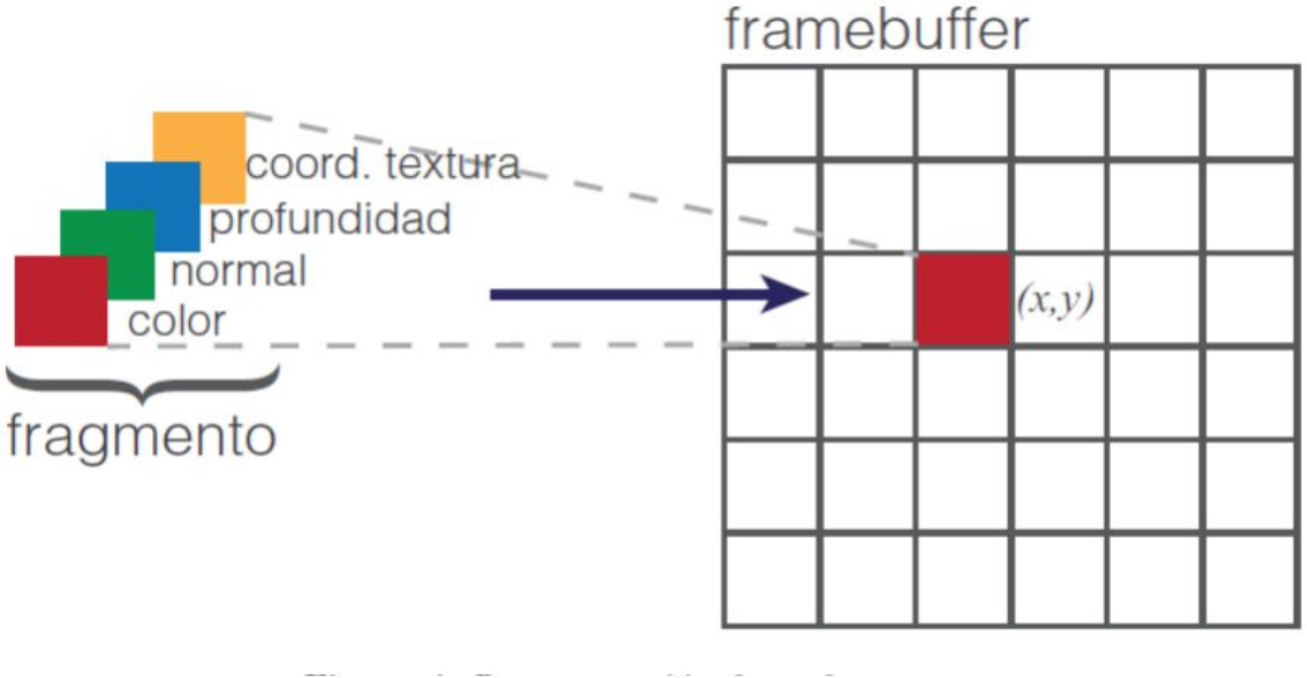
Fragment Shader

```
#version 330
#extension GL_ARB_separate_shader_objects :enable

in vec4 vVertexColor;

layout(location = 0) out vec4 vFragColor;

void main(void){
vFragColor = vVertexColor;
}
```

Fragment Shader

- Estructura parecida al programa de vértices.
- Toma como entrada la salida del shader anterior, en nuestro caso es el vertex shader.
- Como salida, este programa indica el color del fragmento correspondiente del framebuffer.

Fragment Shader



Fragment Shader

Para la carga correcta de shaders y su vinculación con el programa OpenGL se puede utilizar alguna clase con la siguiente estructura:

- ***GLSLProgram.cpp***
- ***GLSLProgram.h***

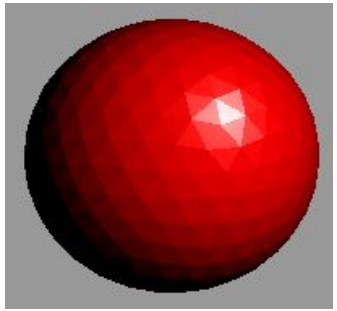
Uso de Shaders

Con esta librería podemos crear un programa de shaders de la siguiente manera:

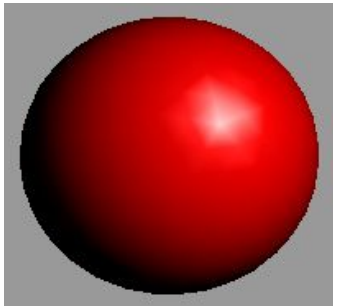
```
//Load the shaders
m_program.loadShader("shaders/basic.vert", CGLSLProgram::VERTEX);
m_program.loadShader("shaders/basic.frag", CGLSLProgram::FRAGMENT);
//Link the shaders in a program
m_program.create_link();
//Enable the program
m_program.enable();
//Link the attributes and the uniforms
m_program.addAttribute("vVertex");
m_program.addAttribute("vColor");
m_program.addUniform("mProjection");
m_program.addUniform("mModelView");
//Disable the program
m_program.disable();
```

Modelos de Sombreado

- ***Flat shading***: usa la normal por cara para calcular la iluminación, este modelo arroja resultados irreales.
- ***Gouraud shading***: usa la normal por vértice, calculamos la iluminación en el *vertex shader*.



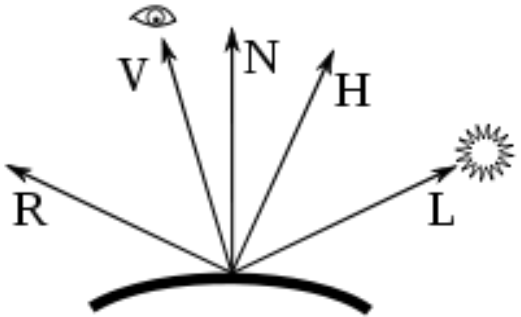
Flat



Gouraud

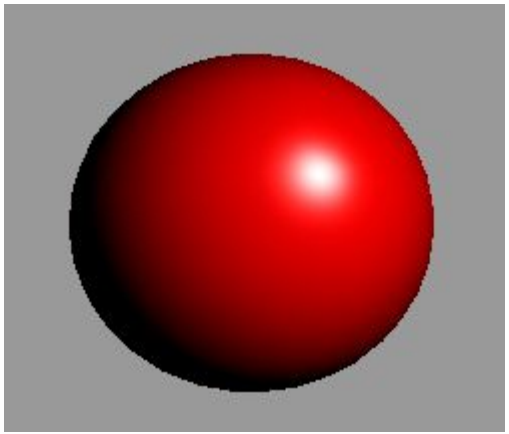
Modelos de Sombreado

- ***Phong:*** usa la normal por vértice y se realiza el cálculo de iluminación en el *fragment shader*.
- ***Blinn-Phong****:* se calcula el vector medio entre el vector vista y el vector fuente de luz (es una modificación del modelo de Phong).

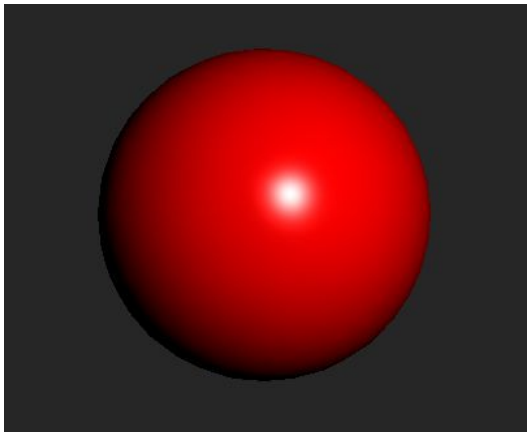


$$H = \frac{L+V}{\|L+V\|}$$

Modelos de Sombreado



Phong



Blinn-Phong

Explicación de código

http://shdr.bkcore.com/