

Introducción a la Programación en GPU



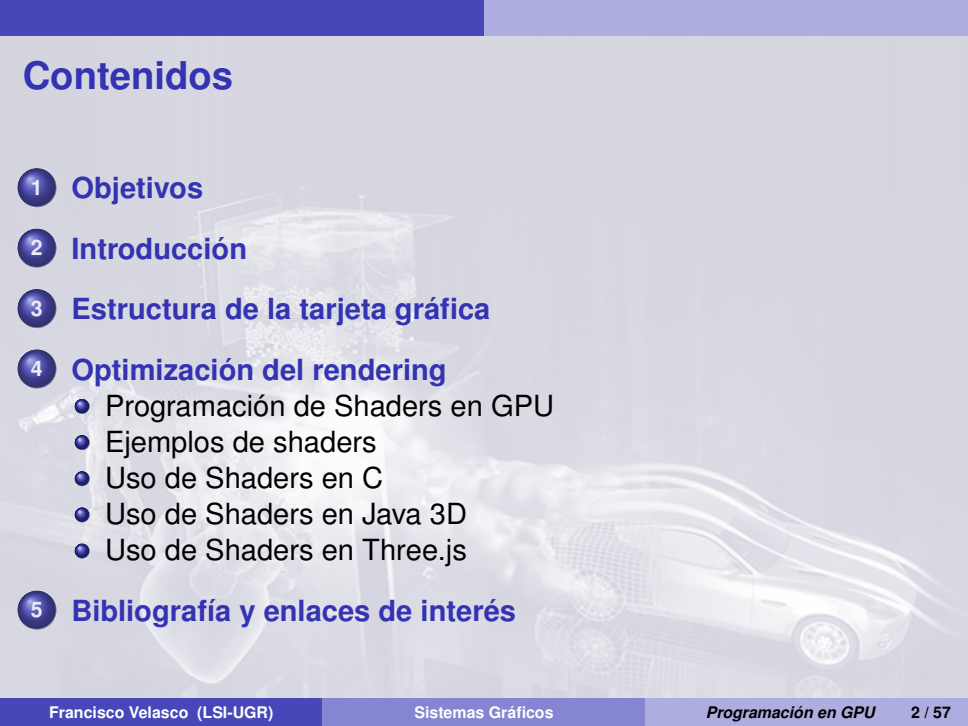
Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Sistemas Gráficos

Grado en Informática
Curso 2016-2017

Contenidos

- 
- 1 **Objetivos**
 - 2 **Introducción**
 - 3 **Estructura de la tarjeta gráfica**
 - 4 **Optimización del rendering**
 - Programación de Shaders en GPU
 - Ejemplos de shaders
 - Uso de Shaders en C
 - Uso de Shaders en Java 3D
 - Uso de Shaders en Three.js
 - 5 **Bibliografía y enlaces de interés**

Objetivos

- Conocer los fundamentos de las tarjetas gráficas y su funcionamiento
- Conocer los fundamentos de la programación con vertex y fragment shaders, según el estándar OpenGL 4.x

Introducción

- GPU: Graphics Processing Unit
 - ▶ Coprocesador dedicado al procesamiento de gráficos
 - ▶ No confundir con *Tarjeta Gráfica*, la placa que aloja a la GPU así como memoria y otra circuitería
 - ▶ Aligera de trabajo a la CPU
 - ★ La CPU se encarga de la gestión de la escena
 - ★ La GPU se encarga de la visualización de la misma



Evolución (1)

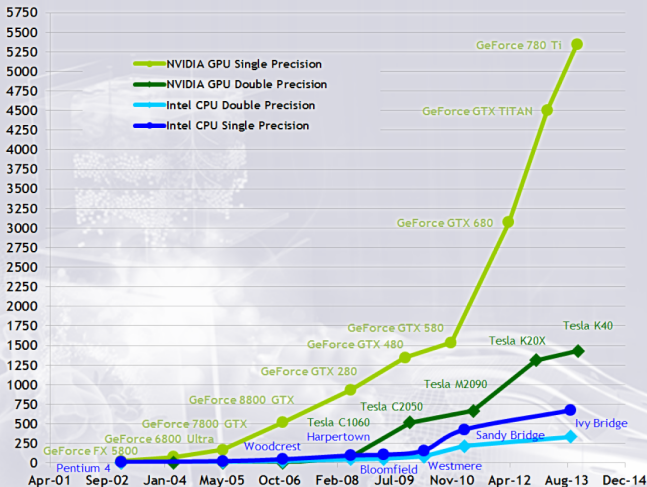
- Anteriores a las GPU existían tarjetas de vídeo (VGA, CGA, etc.)
 - ▶ Un controlador con memoria dedicada encargado de enviar imágenes a un monitor
- Se le fueron añadiendo tareas de rasterización, texturización, sombreado básico
- En 1999 nVIDIA presenta la GeForce 256
 - ▶ Es considerada la primera GPU
 - ▶ Realiza cálculos de geometría y de iluminación
- Las GPU se hacen cada vez más programables
 - ▶ Los programas, *Shaders*, se pueden aplicar a cada triángulo, vértice y píxel
 - ▶ Permite obtener efectos más complejos

Evolución (2)

- Han evolucionado mucho en la última década
- Por necesidades de los sistemas gráficos: videojuegos, cine, etc.
- En una tendencia iniciada por nVIDIA, se han ido transformando:
 - ▶ De una arquitectura orientada exclusivamente a la visualización 3D
 - ▶ Hacia una arquitectura paralela de propósito general
- GPGPU, General Purpose GPU
 - ▶ Usándose en áreas muy diversas:
Síntesis de proteínas, identificación de huellas, etc
 - ▶ Superando incluso a CPUs con varios núcleos

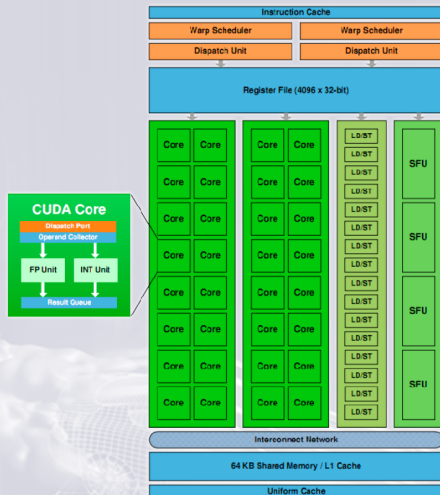
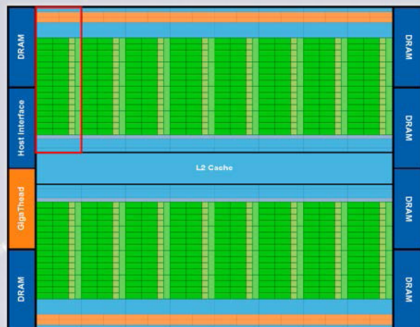
Evolución (y 3)

Theoretical GFLOP/s



Floating-Point Operations per Second - Nvidia CUDA C Programming Guide
Version 6.5 - 24/9/2014 - copyright Nvidia Corporation 2014

Estructura de la tarjeta gráfica (1)



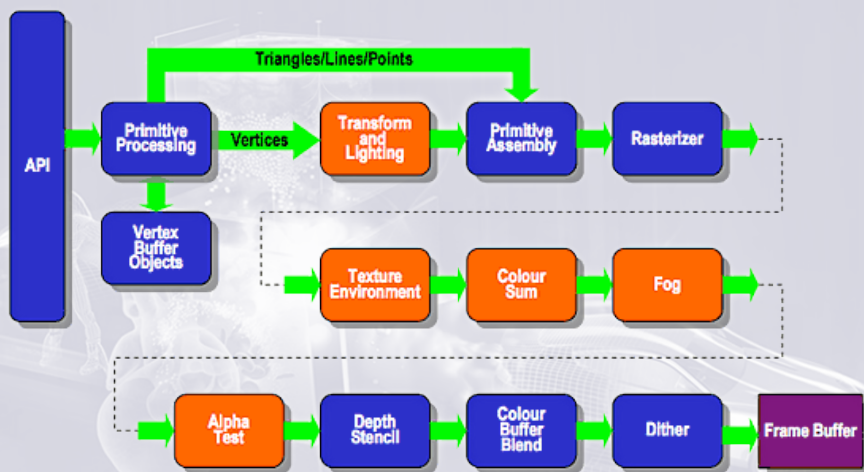
Estructura de una GPU de la serie 300 de nVIDIA GeForce

Estructura de la tarjeta gráfica (y 2)

- La GPU anterior consta de 512 cores
- Organizados en 16 multiprocesadores de 32 cores cada uno
- Cada core posee unidades de cálculo entero y en coma flotante
- Cada multiprocesador dispone de:
 - ▶ Registros,
 - ▶ Memoria compartida
 - ▶ Unidades SFU (Special Function Unit) que pueden realizar cálculos trigonométricos
- Las instrucciones GPU son SIMD (Single Instruction Multiple Data)
 - ▶ Cada Shader es ejecutado sobre miles de datos en paralelo
Cálculo de iluminación, aplicación de texturas, etc.

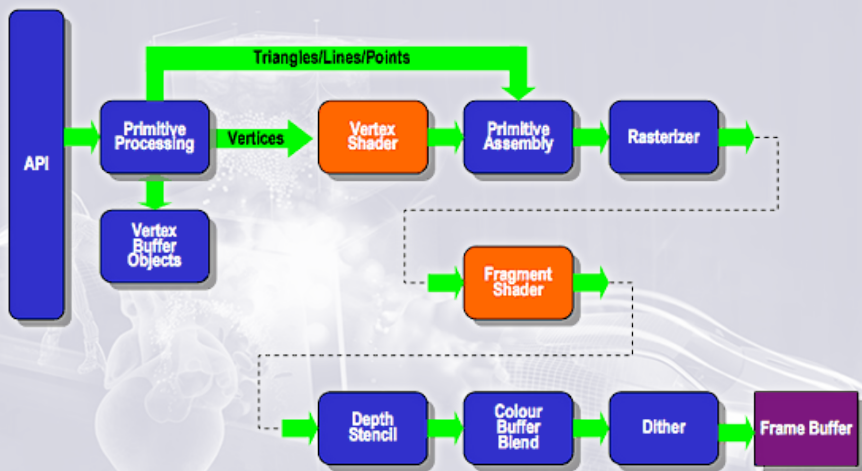
Optimización del rendering

Existing Fixed Function Pipeline



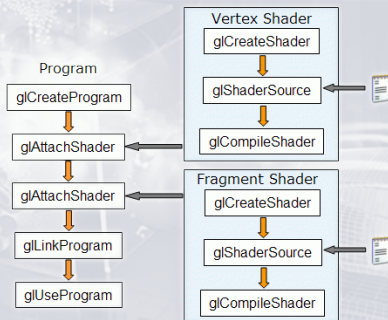
Optimización del rendering

ES2.0 Programmable Pipeline



Programación de Shaders en GPU

- Los Shaders se escriben en GLSL (OpenGL Shading Language)
 - ▶ Lenguaje inspirado en C
- Modo de uso
 - ▶ El código fuente de cada shader es compilado por OpenGL
 - ▶ Un programa objeto es creado que enlaza todos los shaders

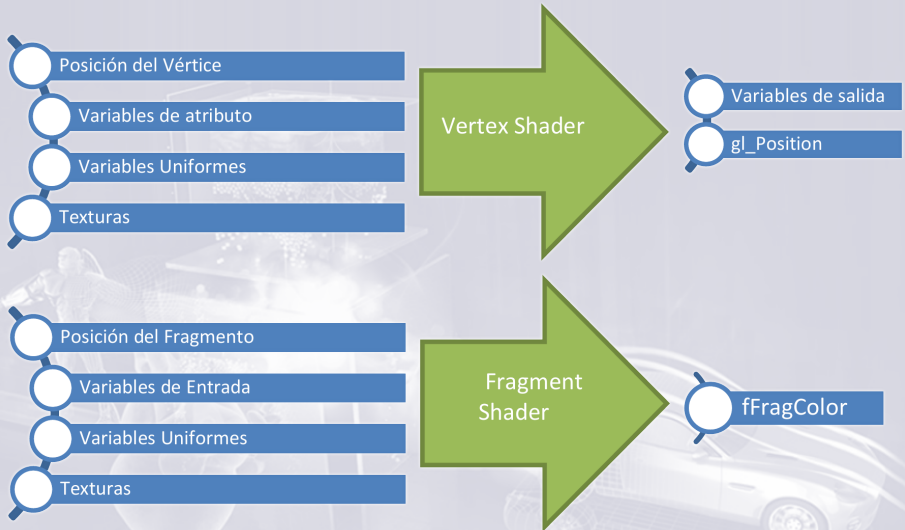


Vertex y fragment shaders

- El vertex shader es llamado para cada vértice
- Realiza cálculos asociados a los vértices
- En etapas posteriores se realiza la rasterización
- El fragment shader es llamado para cada píxel
- Realiza cálculos con el objetivo de obtener el color del píxel

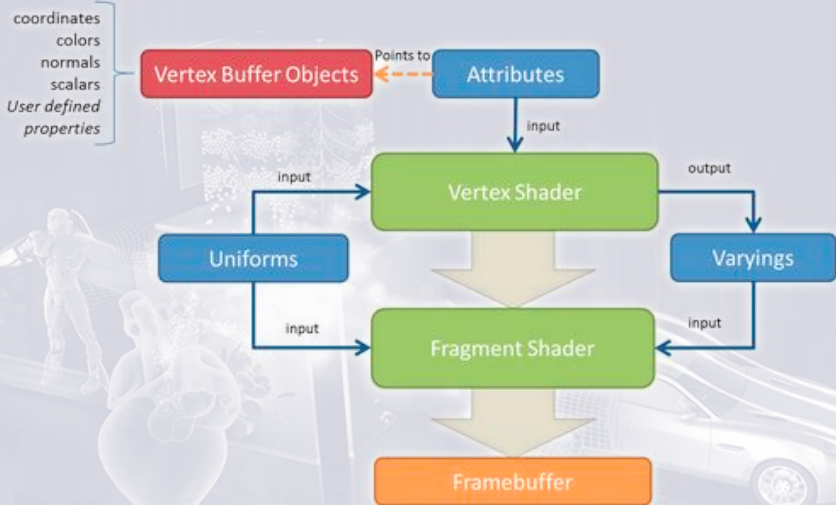


Flujo de datos en el vertex y fragment shader (1)



Flujo de datos en el vertex y fragment shader (y 2)

Rendering Pipeline Overview

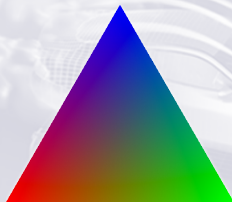
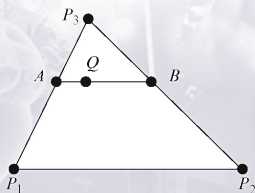


Vertex shader

- Se ejecuta sobre cada vértice de la escena
- Información de entrada
 - ▶ Vértices: Coordenadas, normales, color, etc.
 - ▶ Iluminación: Vectores a las fuentes, intensidades, etc.
- Operaciones que se pueden programar
 - ▶ Transformaciones de vértices y normales
 - ▶ Cálculo de la iluminación por vértice
 - ▶ Gestión de las coordenadas de textura
- Salidas
 - ▶ El vértice en coordenadas de dispositivo, `gl_Position`, obligatorio
 - ▶ Intensidad lumínica o color en los vértices
 - ▶ Normales, coordenadas de textura, etc.

Fragment shader

- Se ejecuta sobre cada píxel
- Información de entrada
 - ▶ Información *interpolada*
 - ▶ También se tiene acceso a información constante
- Operaciones que se pueden realizar
 - ▶ Principalmente, calcular el color del píxel
 - ▶ Mediante iluminación, texturas, etc.
- Información de salida: El color del píxel, `gl_FragColor`, obligatorio



Parámetros de entrada / salida

● uniform

- ▶ De solo lectura, constantes en todo el cauce de procesamiento de una primitiva
- ▶ Accesibles por ambos shaders
- ▶ Se establecen en la aplicación OpenGL
- ▶ Pueden especificar valores como:
 - ★ Matrices de transformación:
Modelado, vista, proyección, una composición de varias, etc.
 - ★ Planos de recorte
 - ★ Propiedades del material: componente ambiental, difuso, etc.
 - ★ Propiedades de las luces: color, posición, dirección, etc.
 - ★ Parámetros de efectos: niebla, densidad, etc.

Parámetros de entrada / salida (y 2)

● attribute

- ▶ De solo lectura
- ▶ Accesibles solo por el vertex shader
- ▶ Se establecen en la aplicación OpenGL
- ▶ Especifican información asociada a los vértices
 - ★ Coordenadas
 - ★ Color
 - ★ Normal
 - ★ Coordenadas de textura

● varying

- ▶ Variables de paso de información entre shaders
- ▶ De escritura/salida en el vertex y lectura/entrada en el fragment
- ▶ Deben estar en ambos

Otros tipos de datos en GLSL

- GLSL se parece mucho a C
 - ▶ Se pueden usar tipos como `float`, `int`, `bool` con los operadores habituales
- GLSL incluye nuevos tipos vectoriales y matriciales
 - ▶ Enteros: `ivec2`, `ivec3`, `ivec4`
 - ▶ Reales: `vec2`, `vec3`, `vec4`
 - ▶ Matrices cuadradas reales: `mat2`, `mat3`, `mat4`
 - ▶ Tienen los operadores `+` y `*` sobrecargados
 - ▶ Pueden accederse
 - ★ Con el operador tradicional `[]`
 - ★ Con nombres significativos, por ejemplo, `color.r`, `punto.x`
- GLSL incluye funciones como
 - ▶ `abs`, `max`, `sqrt`, `pow`, `log`, `cos`, `normalize`, `dot`, `cross`, etc.

Variables

- Convenciones para nombrar variables
 - ▶ `a_nombre`, para nombrar atributos de vértices
 - ▶ `u_nombre`, para nombrar variables uniformes
 - ▶ `v_nombre`, para las salidas del vertex shader
 - ▶ `f_nombre`, para las salidas del fragment shader
- Nombres usados habitualmente, prácticamente un estándar
 - ▶ `a_vertex`, el vértice actual, en coordenadas del modelo
 - ▶ `a_normal`, la normal del vértice
 - ▶ `a_color`, el color del vértice
- Se disponen en ambos shaders de diversas variables uniform
 - ▶ `u_ModelViewMatrix`, la matriz de modelado y vista
 - ▶ `u_ProjectionMatrix`, la matriz de proyección, etc.

Ejemplos: Gouraud (1)

Gouraud: Vertex shader: Parámetros de entrada / salida

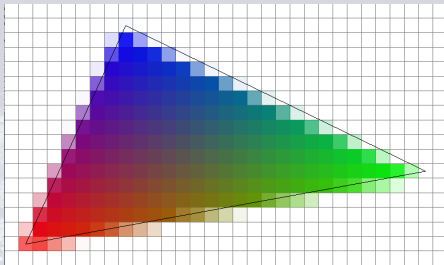
```
// _____ Constantes _____  
// Transformaciones para puntos y normales  
uniform mat4 u_mvp_matrix;  
uniform mat3 u_normalMatrix;  
  
// Parámetros de un material Lambertiano  
uniform vec4 u_ambient;      uniform vec4 u_diffuse;  
uniform vec4 u_specular;     uniform float u_shininess;  
  
// Una luz direccional  
uniform vec3 u_light_direction;  uniform vec4 u_light_color;  
  
// El vector al observador  
uniform vec3 u_observer  
  
// _____ Atributos de los vértices _____  
attribute vec4 a_position;  
attribute vec3 a_normal;  
  
// _____ Salida para el fragment shader _____  
varying vec4 v_color;
```

Gouraud (2)

Gouraud: Vertex shader: main

```
void main() {  
    // Variables locales para cálculos intermedios  
    float NdotL,    NdotHV;  
    vec3 HV,    normal;  
  
    // Cálculo del vértice en coordenadas de dispositivo  
    gl_Position = u_mvp_matrix * a_position;  
    // Cálculo de la normal en coordenadas de vista  
    normal = u_normalMatrix * a_normal;  
  
    // Cálculo del color por Lambert, cálculos intermedios  
    NdotL = max (dot (normal, u_light_direction), 0.0);  
    HV = normalize (u_observer + u_light_direction);  
    NdotHV = max (dot (normal, HV), 0.0);  
  
    // Cálculo del color por Lambert  
    v_color = min (u_ambient +  
        NdotL * u_diffuse * u_light_color +  
        pow (NdotHV, u_shininess) * u_specular * u_light_color,  
        vec4(1,1,1,1));  
}
```

Gouraud (y 3)



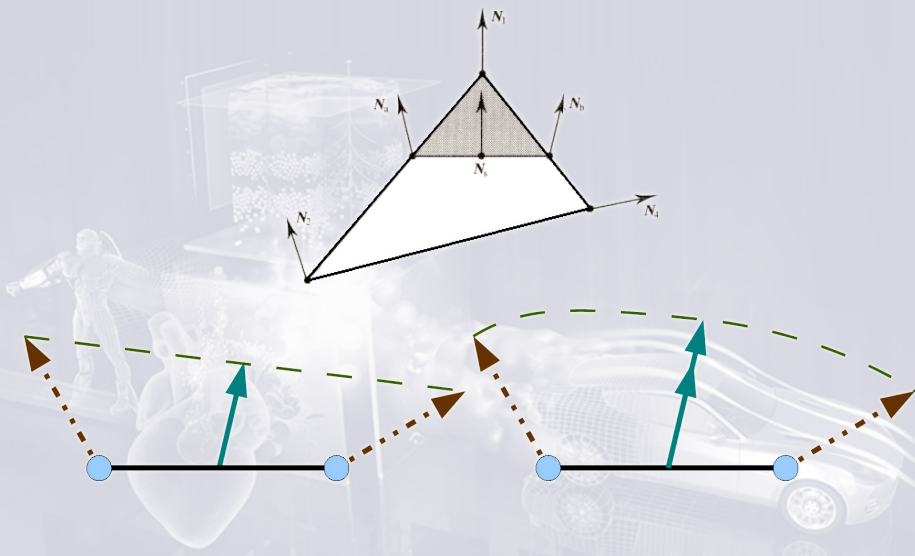
Fragment shader: Parámetros y main

```
// Entrada: Un color interpolado  
varying vec4 v_color;
```

```
// En el caso de Gouraud no son necesarios más cálculos
```

```
void main()  
{  
    gl_FragColor = v_color;  
}
```


Ejemplos: Phong (1)



Phong (2)

Phong: Vertex shader: Parámetros y main

```
// ———— Constantes ————  
// Transformación del modelo al dispositivo  
uniform mat4 u_mvp_matrix;  
uniform mat3 u_normalMatrix;  
  
// ———— Atributos de los vértices ————  
attribute vec4 a_position;  
attribute vec3 a_normal;  
  
// ———— Salida para el fragment shader ————  
varying vec3 v_normal;  
  
// ———— Programa ————  
void main()  
{  
    // Cálculo del vértice en coordenadas de dispositivo  
    gl_Position = u_mvp_matrix * a_position;  
  
    // Las normales son transmitidas para su interpolación  
    v_normal = u_normalMatrix * a_normal;  
}
```

Phong (3)

Phong: Fragment shader: Parámetros

```
// ———— Constantes ————  
  
// Parámetros de un material Lambertiano  
uniform vec4 u_ambient;    uniform vec4 u_diffuse;  
uniform vec4 u_specular;   uniform float u_shininess;  
  
// Una luz direccional  
uniform vec3 u_light_direction;    uniform vec4 u_light_color;  
  
// El vector al observador  
uniform vec3 u_observer  
  
// ———— Entrada desde el vertex shader ————  
varying vec3 v_normal;
```

Phong (y 4)

Phong: Fragment shader: main

```
void main()
{
    float NdotL;           // Para cálculos intermedios
    float NdotHV;
    vec3 HV;
    vec3 normal;           // Para normalizar la normal de entrada

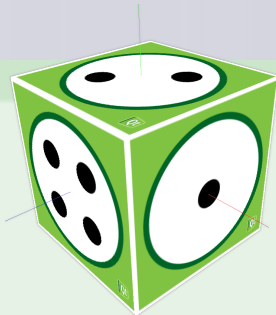
    // Cálculo del color por Lambert, cálculos intermedios
    normal = normalize (v_normal);
    NdotL = max (dot (normal, u_light_direction), 0.0);
    HV = normalize (u_observer + u_light_direction);
    NdotHV = max (dot (normal, HV), 0.0);

    // Cálculo del color por Lambert
    gl_FragColor = min (u_ambient +
        NdotL * u_diffuse * u_light_color +
        pow (NdotHV, u shininess) * u_specular * u_light_color,
        vec4(1,1,1,1));
}
```

Texturas (1)

Texturas: Vertex shader: Parámetros y main

```
// ———— Constantes ————  
// Transformación del modelo al dispositivo  
uniform mat4 u_mvp_matrix;  
  
// ———— Atributos de los vértices ————  
attribute vec4 a_position;  
attribute vec2 a_texcoord;  
  
// ———— Salida para el fragment shader ————  
varying vec2 v_texcoord;  
  
void main()  
{  
    // Cálculo del vértice en coordenadas del dispositivo  
    gl_Position = u_mvp_matrix * a_position;  
  
    // Las coords de textura son transmitidas para su interpolación  
    v_texcoord = a_texcoord;  
}
```

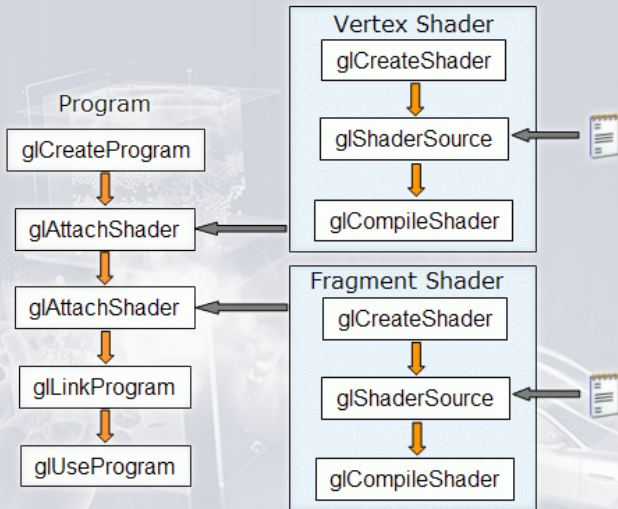


Texturas (y 2)

Texturas: Fragment shader: Parámetros y main

```
// ———— Constantes ————  
  
// La textura  
uniform sampler2D u_texture;  
  
// ———— Entrada desde el vertex shader ————  
  
// Las coordenadas de textura concretas de este píxel  
varying vec2 v_texcoord;  
  
void main()  
{  
    // Cálculo del color a partir de la textura  
    gl_FragColor = texture2D (u_texture , v_texcoord);  
}
```

Creación de un programa shader en C



Lectura y compilación de un shader en C

Lenguaje C: Lectura y compilación de un shader

// type – El tipo de shader a cargar, vertex o fragment
// shaderPath – El archivo que contiene el código fuente del shader

```
GLuint loadShader (GLenum type, const char *shaderPath)
{
    // Se almacena el texto del archivo en una cadena
    //        finalizada en '\0'
    const char *shaderSrc = readFromFile (shaderPath);

    // Se crea el shader
    GLuint shader = glCreateShader(type);

    // Se carga el código fuente
    glShaderSource(shader, 1, &shaderSrc, NULL);

    // Se compila el shader
    glCompileShader(shader);

    return shader;
}
```


Creación de un programa shader en C

Lenguaje C: Creación del programa shader

```
// Recibe los path de los ficheros con el código fuente
GLuint createShaderProgram (const char *vertexShaderPath ,
                           const char *fragmentShaderPath)
{
    GLuint vertexShader , fragmentShader , shaderProgram ;

    // Se cargan los shaders , compilados
    vertexShader  = loadShader (GL_VERTEX_SHADER, vertexShaderPath) ;
    fragmentShader = loadShader (GL_FRAGMENT_SHADER, fragmentShaderPath) ;

    // Se crea el programa, se le añaden los shaders y se linka
    shaderProgram = glCreateProgram () ;
    glAttachShader (shaderProgram , vertexShader) ;
    glAttachShader (shaderProgram , fragmentShader) ;
    glLinkProgram (shaderProgram) ;
    glUseProgram (shaderProgram) ;

    return shaderProgram ;
}
```

Envío de información a los shaders en C

Parámetros uniform

- Los parámetros de entrada/salida se declaran y usan en el shader
- Se les da valores en la aplicación

Envío de información a los shaders en C: Parámetros uniform

```
// Si en el shader se tiene

uniform vec3 u_light_direction

// En la aplicación se pone

GLfloat light_direction[] = {1.0f, 2.0f, 3.0f};

GLint location_light_direction =
    glGetUniformLocation (shaderProgram, "u_light_direction");

glUniform3fv (shaderProgram, location_light_direction ,
    1, light_direction)
```

Envío de información a los shaders en C

Texturas

Envío de información a los shaders en C: Texturas

```
// Si en el shader se tiene
uniform sampler2D u_texture

// En la aplicación se pone
// Se carga la textura

GLuint textureId;
glGenTextures (1, &textureId);
glActiveTexture (GL_TEXTURE0);
glBindTexture (GL_TEXTURE_2D, textureId);
glTexImage2D ( ... parámetros para cargar la imagen ... );

// Se vincula la textura con el shader

GLint location_u_texture =
    glGetUniformLocation (shaderProgram, "u_texture");

glUniform1i (shaderProgram, location_u_texture, 0);
```

Envío de información a los shaders en C

Parámetros attribute (1)

Envío de información a los shaders en C: Parámetros attribute (1)

```
// Si en el shader se tiene  
  
attribute vec4 a_position  
attribute vec3 a_normal  
  
// En la aplicación se pone  
  
// Se asume que los vértices están almacenados en un array.  
//     En cada posición del array hay un struct  
//     con la posición del vértice y su normal  
  
struct Vertex {  
    GLfloat position[4];  
    GLfloat normal[3];  
}  
Vertex *vertices;  
  
// Además se tiene un array indexes con los índices  
  
GLuint *indexes;
```

Envío de información a los shaders en C

Parámetros attribute (2)

Envío de información a los shaders en C: Parámetros attribute (2)

```
// Buffer objects para vértices e índices
```

```
GLuint vbo[2];  
glGenBuffers (2, vbo);  
  
glBindBuffer (GL_ARRAY_BUFFER, vbo[0]);  
glBufferData (GL_ARRAY_BUFFER,  
    totalVertices * sizeof (Vertex), vertices, GL_STATIC_DRAW);  
  
glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, vbo[1]);  
glBufferData (GL_ELEMENT_ARRAY_BUFFER,  
    totalIndexes * sizeof (GLuint), indexes, GL_STATIC_DRAW);
```

Envío de información a los shaders en C

Parámetros attribute (y 3)

Envío de información a los shaders en C: Parámetros attribute (y 3)

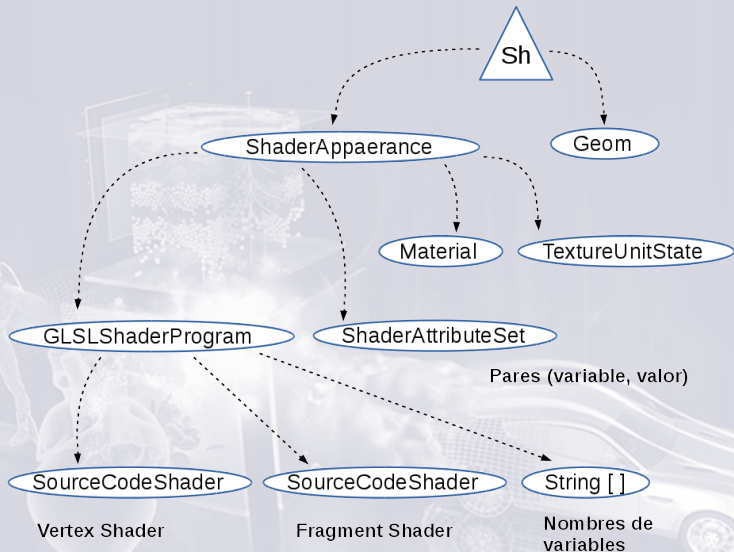
```
// Se vinculan los parámetros del shader con los buffer objetos

int offset = 0; // Desplazamiento del position en el struct
location_a_position = glGetAttribLocation (shaderProgram,
    "a_position");
glEnableVertexAttribArray (location_a_position);
glVertexAttribPointer (location_a_position, 4, GLfloat, GL_FALSE,
    sizeof(Vertex), (void *) offset);

offset += 4 * sizeof (GLfloat); // Desplazamiento de la normal
location_a_normal = glGetAttribLocation (shaderProgram, "a_normal");
glEnableVertexAttribArray (location_a_normal);
glVertexAttribPointer (location_a_normal, 3, GLfloat, GL_FALSE,
    sizeof(Vertex), (void *) offset);

glDrawElements (GL_TRIANGLE_STRIP, totalIndexes, GLuint, 0);
```

Uso de Shaders en Java 3D



Lectura y compilación de un shader en Java3D

Lenguaje Java: Lectura y compilación de un shader

```
String vertexSourceCode = readFile("pathToVertex.glsl");

SourceCodeShader vertexShader = new SourceCodeShader (
    Shader.SHADING_LANGUAGE_GLSL,
    Shader.SHADER_TYPE_VERTEX, vertexSourceCode);

String fragmentSourceCode = readFile("pathToFragment.glsl");

SourceCodeShader fragmentShader = new SourceCodeShader (
    Shader.SHADING_LANGUAGE_GLSL,
    Shader.SHADER_TYPE_FRAGMENT, fragmentSourceCode);

GLSLShaderProgram programShader = new GLSLShaderProgram();
Shader[] shaders = {vertexShader, fragmentShader};
programShader.setShaders(shaders);
```


Lectura y compilación de un shader en Java3D

- El código fuente tiene que estar en un String
- Se puede usar el siguiente método para guardar un archivo de texto en un solo String

Java: Lectura de un archivo en un String

```
String readFile (String path) {  
    String output = "";  
    try {  
        output = new String (Files.readAllBytes(Paths.get(path)),  
                               Charset.defaultCharset());  
    } catch (IOException e) {  
        System.err.println (e);  
    }  
    return output;  
}
```

Envío de información a los shaders en Java3D

Parámetros uniform (1)

- Los nombres de las variables se le indican al GLSLShaderProgram
- Los pares (nombre, valor) se le indican al ShaderAppearance

Envío de información a los shaders en Java3D: Parámetros uniform

// Si en el shader se tiene

```
uniform float unEscalar;  
uniform vec3 unVector;
```

// En la aplicación se pone

```
String[] shaderAttrNames = { "unEscalar", "unVector" };
```

```
Object[] shaderAttrValues = {  
    new Float (0.3),  
    new Vector3f (1f, 0f, 0f)  
};
```

Envío de información a los shaders en Java3D

Parámetros uniform (2)

Envío de información a los shaders en Java3D: Parámetros uniform

```
// Al GLSLShaderProgram se le indican los nombres
programShader.setShaderAttrNames (shaderAttrNames);

// Al ShaderAppearance se le dan los pares (nombre, valor)
ShaderAttributeSet shaderAttrSet = new ShaderAttributeSet();

for (int i = 0; i < shaderAttrNames.length; i++) {
    shaderAttrSet.put (new ShaderAttributeValue (
        shaderAttrNames[i], shaderAttrValues[i]));
}

shaderAppearance.setShaderAttributeSet (shaderAttrSet);
```

Envío de información a los shaders en Java3D

Texturas

Envío de información a los shaders en Java3D: Texturas

```
// Si en el shader se tiene
uniform sampler2D u_texture

// En la aplicación se envía como un uniform
String[] shaderAttrNames = { u_texture } ;
Object[] shaderAttrValues = { new Integer (0) };

// Y se asignan los nombres y los pares (nombre, valor)
// como se ha visto en la transparencia anterior
```

Acceso a información Java3D desde GLSL

Matrices de transformación y fuentes de luz

- Desde los shaders se tienen disponibles las siguientes variables
 - ▶ `gl_ModelViewProjectionMatrix`, necesaria para transformar los vértices a coordenadas de dispositivo
 - ▶ `gl_NormalMatrix`, para transformar las normales
 - ▶ `gl_MaxLights`, el total de luces que influyen en ese momento
 - ▶ `gl_LightSource[i]`, la i-ésima fuente de luz (la primera es la 0)
 - ★ `.position`, su posición o dirección
 - ★ `.halfVector`, para el cálculo de iluminación mediante Phong
 - ★ `.ambient`, su componente ambiental
 - ★ `.diffuse`, su componente difusa
 - ★ `.specular`, su componente especular

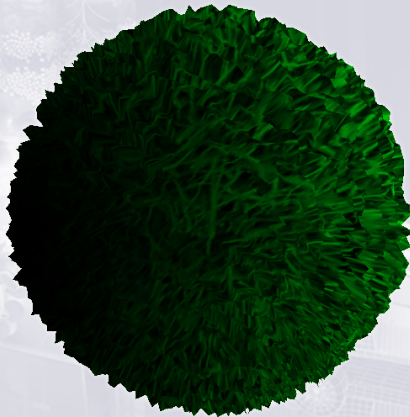
Acceso a información Java3D desde GLSL

Material y atributos de los vértices

- Desde los shaders se tienen disponibles las siguientes variables
 - ▶ `gl_FrontMaterial`, el material de la figura
 - ★ `.ambient`, su componente ambiental
 - ★ `.diffuse`, su componente difusa
 - ★ `.specular`, su componente especular
 - ★ `.shininess`, su exponente especular
- En el vertex shader se pueden acceder a los siguientes atributos de los vértices:
 - ▶ `gl_Vertex`, sus coordenadas
 - ▶ `gl_Normal`, su normal
 - ▶ `gl_MultiTexCoord0.st`,
Sus coordenadas de textura, primera textura
 - ▶ `gl_MultiTexCoord1.st`, idem con la segunda textura, etc.

Uso de Shaders en Three.js

- Cada material predefinido lleva asociado su shader
- Para usar shaders personalizados debemos usar la clase `ShaderMaterial`



Clase ShaderMaterial

Código fuente de los shaders

Shader: Código fuente del vertex shader

```
<script type="x-shader/x-vertex" id="vertexS">
    uniform float amplitude;
    attribute float displacement;
    varying vec3 v_normal;
    varying vec4 v_position;

    void main() {
        vec3 newPosition = position + normal * displacement * amplitude;
        v_position = modelViewMatrix * vec4 (newPosition, 1.0);
        gl_Position = projectionMatrix * v_position;

        // Cálculo de la normal en coordenadas de vista
        v_normal = normalize (vec3 (normalMatrix * normal));
    }
</script>
```


Clase ShaderMaterial

uniform y attribute disponibles

ShaderMaterial: uniform y attribute disponibles

// Datos globales

```
uniform mat4 modelMatrix;  
uniform mat4 viewMatrix;  
uniform mat4 modelViewMatrix;  
uniform mat4 projectionMatrix;  
uniform mat3 normalMatrix;  
uniform vec3 cameraPosition;
```

// Atributos para cada vértice

```
attribute vec3 position;  
attribute vec3 normal;  
attribute vec2 uv;  
attribute vec3 color;
```

Clase ShaderMaterial

Añadido de más `uniform` y `attribute` y `define`

ShaderMaterial: Añadido de más `uniform` y `attribute`

```
uniforms = {  
  amplitude: { type: "f", value: 1.0 }  
};  
// Se puede modificar cuando se desee  
uniforms.amplitude.value = 2.5;  
  
// Los atributos se asignan a la geometría, debe ser BufferGeometry  
var nVertices = model.geometry.attributes.position.count;  
var displacement = new Float32Array (nVertices);  
for (var v = 0; v < nVertices; v++) {  
  displacement[v] = Math.random();  
}  
model.geometry.addAttribute( 'displacement',  
  new THREE.BufferAttribute (displacement,1));  
  
// Cuando se cambian atributos hay que solicitar una actualización  
model.geometry.attributes.displacement.needsUpdate = true;
```

Clase ShaderMaterial

Tipos de datos

- Correspondencia entre tipos de datos de GLSL y Three.js

GLSL	type:	value
int	"i"	un entero
float	"f"	un real
vec2	"v2"	THREE.Vector2
vec3	"v3"	THREE.Vector3
vec3	"c"	THREE.Color
vec4	"v4"	THREE.Vector4
mat3	"m3"	THREE.Matrix3
mat4	"m4"	THREE.Matrix4
sampler2D	"t"	THREE.Texture

Clase ShaderMaterial

Creación del material

ShaderMaterial: Creación del material

```
var shaderMat = new THREE.ShaderMaterial ( {  
    uniforms:      uniforms ,  
    vertexShader:  document.getElementById ( 'vertexS' ).textContent ,  
    fragmentShader: document.getElementById ( 'fragmentS' ).textContent ,  
  
    // Se le pueden poner otros campos opcionales  
    wireframe: true ,    // y se mostraría en modo alambre  
    transparent: true ,  // obligatorio si se manejan transparencias  
    lights: true    // si se van a usar luces definidas en THREE  
});
```

Uso de luces en un shader personalizado

- Al definir el `ShaderMaterial` se debe
 - ▶ Poner el atributo `lights` a `true`
 - ▶ Añadir a nuestros `uniforms` los `uniforms` de las luces
- En el shader se debe declarar:
 - ▶ Una estructura con los campos adecuados según el tipo de luz que se desea usar
 - ★ Esa información se obtiene de los fuentes de la biblioteca
 - ★ En concreto de
`src/renderer/shaders/ShaderChunk/lights_pars.glsl`
 - ▶ Una variable `uniform` que sea un array de elementos de dicha estructura
- Se pueden copiar y pegar esas declaraciones desde ese archivo en nuestro Shader

Uso de luces en un shader personalizado

Ejemplo: Definición del ShaderMaterial

Luces en un shader personalizado: En la aplicación js

```
var shaderMat = new THREE.ShaderMaterial ({  
  // se mezclan los uniforms de las luces con otros que hayamos  
  // podido definir nosotros, en el ejemplo, amplitude  
  uniforms : THREE.UniformsUtils.merge ([  
    THREE.UniformsLib[ 'lights ' ],  
    {  
      amplitude : { type : 'f', value : 5.0 }  
    }  
  ]),  
  vertexShader: document.getElementById ( 'vertexS' ).textContent ,  
  fragmentShader: document.getElementById ( 'fragmentS' ).textContent ,  
  // se activa el atributo lights  
  lights : true  
});
```

Uso de luces en un shader personalizado

Ejemplo: Definición y uso en el Shader

Archivo `lights_pars.glsl`: Fragmento relativo a las luces direccionales

```
struct DirectionalLight {  
    vec3 direction;  
    vec3 color;  
    int shadow;  
    float shadowBias;  
    float shadowRadius;  
    vec2 shadowMapSize;  
};  
uniform DirectionalLight directionalLights[ NUM_DIR_LIGHTS ];
```

- Se copian las declaraciones en nuestro Shader y se usa el array de luces de la manera habitual

Nuestro shader: Uso de las luces

```
for (int i = 0; i < NUM_DIR_LIGHTS; i++) {  
    esteColor = directionalLights[i].color;  
    // se hacen los cálculos necesarios  
}
```

Bibliografía y enlaces de interés

- M. Bailey, S. Cunningham; **Graphics Shaders, Theory and Practice**; *CRC Press*; 2012
- Shreiner, Dave, et al.; **OpenGL programming guide: the official guide to learning OpenGL, version 4.1**; *Addison-Wesley*; 2013 (disponible online en la biblioteca)
- <https://developer.nvidia.com/content/gpu-gems>
- <https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/>
- <http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/>
- <http://webglfundamentals.org/>
- <http://www.html5rocks.com/en/tutorials/webgl/shaders/>

Introducción a la Programación en GPU

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Sistemas Gráficos

Grado en Informática
Curso 2016-2017

Parte de este material ha sido realizado en colaboración con Francisco Javier Melero Rus