



Programación de shaders en GPU (introducción)


Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática
Curso 2021-2022

Contenidos

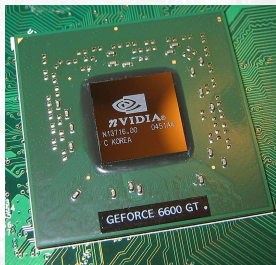
- 
- 1 **Introducción**
 - 2 **Estructura de un programa shader**
 - 3 **Ejemplos de shaders**
 - 4 **Uso de Shaders en Three.js**

Objetivos

- Conocer el cauce programable de OpenGL
- Conocer la responsabilidad del Vertex y Fragment Shader
- Tener nociones de la programación de shaders en GPU

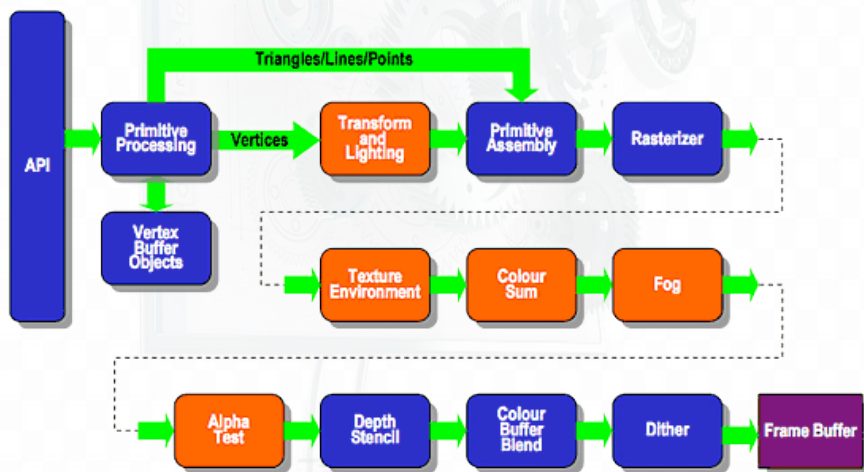
Introducción

- 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



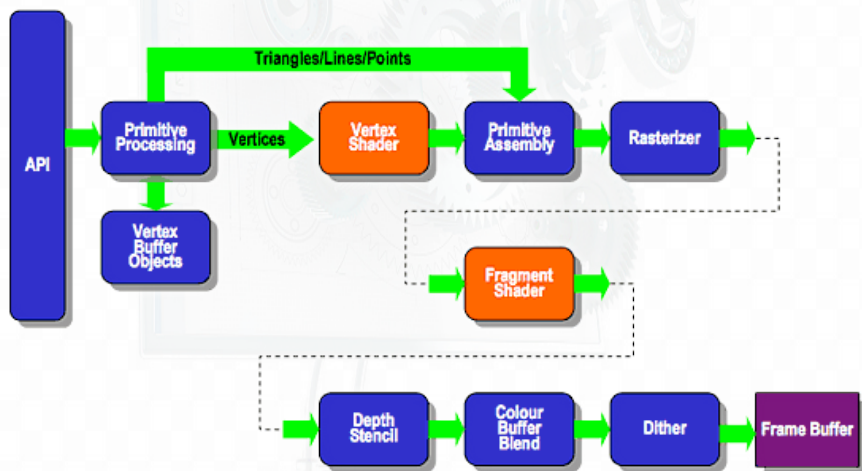
Optimización del rendering

Existing Fixed Function Pipeline



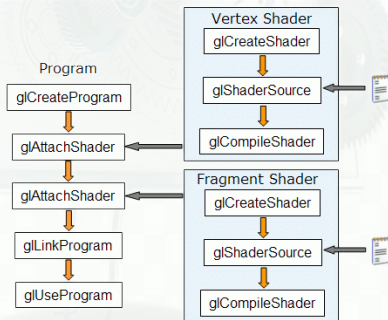
Optimización del rendering

ES2.0 Programmable Pipeline



Estructura de un programa shader

- 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

- **Vertex Shader (VS)**

- ▶ Es llamado para cada vértice
- ▶ Realiza cálculos asociados a los vértices

- **Rasterizador**

- ▶ Descompone el polígono en píxeles (fragmentos)
- ▶ Y le asigna los datos que le corresponden por interpolación entre la información que el VS ha calculado para los vértices

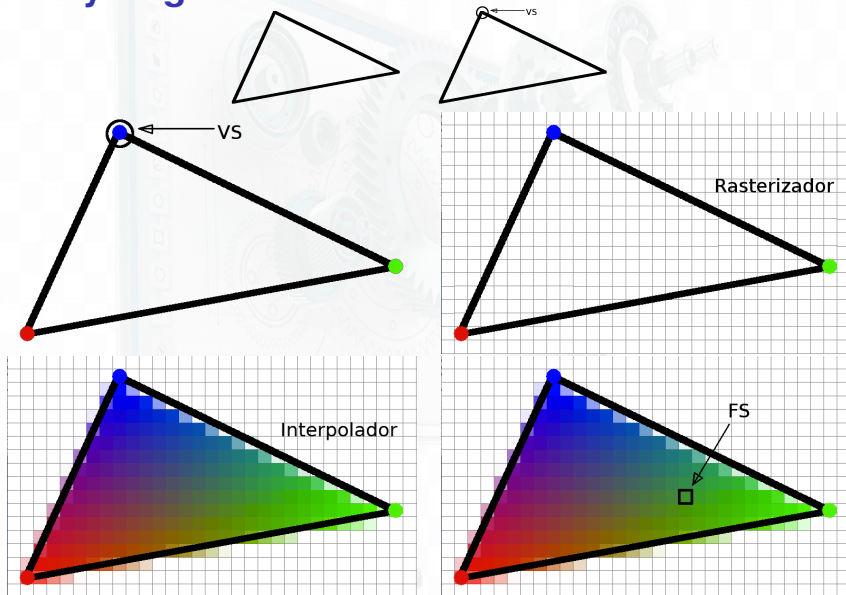
- **Fragment Shader (FS)**

- ▶ Es llamado para cada fragmento
- ▶ Realiza cálculos con el objetivo de obtener el color del fragmento

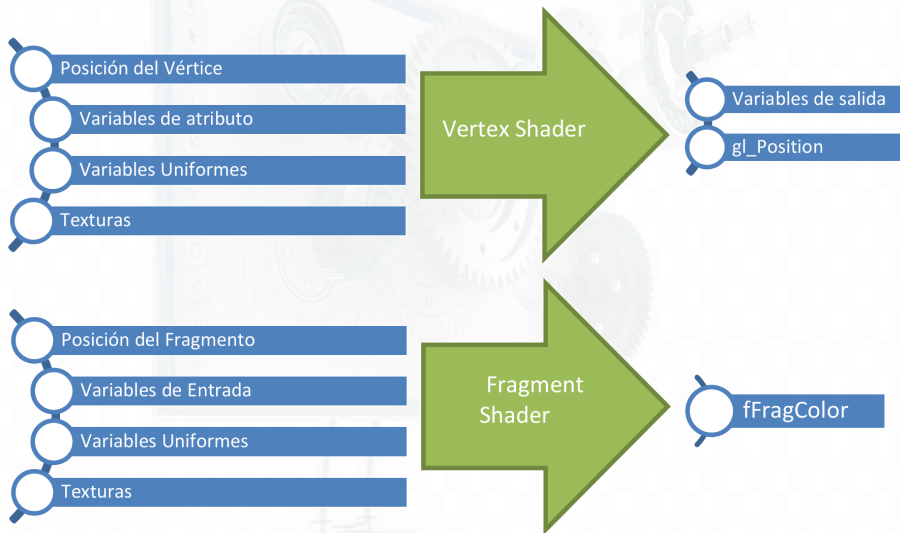
- **Mezclador de fragmentos**

- ▶ Los fragmentos que *caen* en el mismo píxel se combinan para obtener el color final del mismo

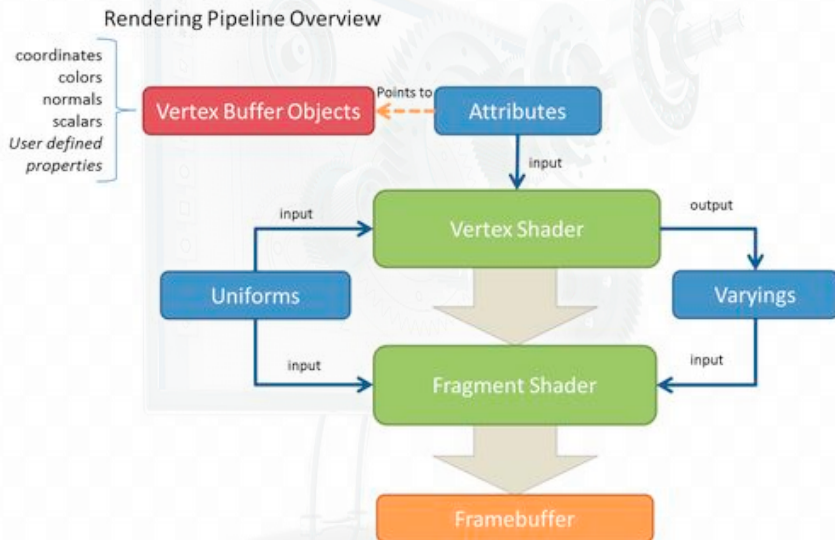
Vertex y fragment shaders



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



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

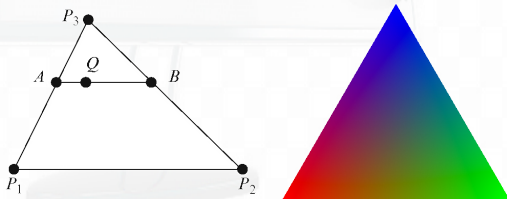


Vertex shader

- 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 transformado, `gl_Position`, obligatorio
 - ▶ Intensidad lumínica o color en los vértices
 - ▶ Normales, coordenadas de textura, etc.

Fragment shader

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



Parámetros de entrada / salida

GLSL

● 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

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)

GLSL

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

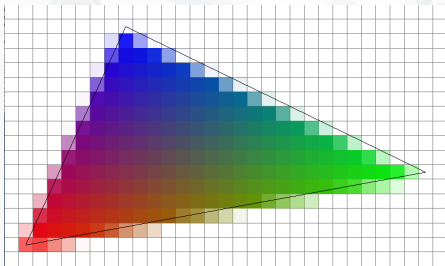
```
void main() {
    // Variables locales para cálculos intermedios
    float NdotL,    NdotHV;
    vec3 HV,    normal;

    // 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
```

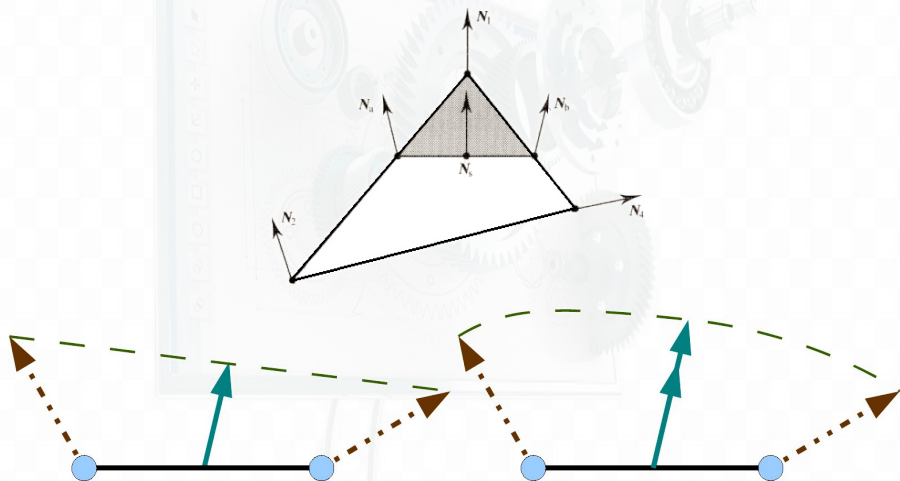
```
varying vec4 v_color;
```

```
// 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 fragmento  
varying vec2 v_texcoord;  
  
void main()  
{  
    // Cálculo del color a partir de la textura  
    gl_FragColor = texture2D (u_texture , v_texcoord);  
}
```

Uso de Shaders en Three.js

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



Textura con Desplazamiento



Clase ShaderMaterial

Código fuente de los shaders

- Se añade el código en el archivo `index.html`, entre etiquetas `script`

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`

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

```
uniforms = {
  amplitud: { type: "f", value: 1.0 }
};
// Se puede modificar cuando se desee
uniforms.amplitud.value = 2.5;

// Los atributos se asignan a la geometría, debe ser BufferGeometry
var nVertices = model.geometry.attributes.position.count;
var aux = new Float32Array (nVertices);
for (var v = 0; v < nVertices; v++) {
  aux[v] = Math.random();
}
model.geometry.setAttribute( 'displacement',
  new THREE.BufferAttribute (aux,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

Three.js

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/renderers/shaders/ShaderChunk/lights_pars_begin.glsl.js`
 - ▶ 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: Al definir el material

```
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_begin.glsl.js`: Struct de las luces direccionales

```
struct DirectionalLight {  
    vec3 direction;  
    vec3 color;  
};  
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  
}
```



Programación de shaders en GPU (introducción)

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática
Curso 2021-2022