PROGRAMACIÓN 3D

Máster en Programación de Videojuegos

TEMA 2: OpenGL

Juan Mira Núñez

U-tad
CENTRO UNIVERSITARIO
DE TECNOLOGÍA Y ARTE DIGITAL

Índice

- OpenGL
- Buffers
- Pintado
- Shaders (Vertex y Fragment)

¿Qué es OpenGL?

OpenGL

- Interfaz para interactuar con el procesador gráfico de un dispositivo.
- Lenguaje multiplataforma para dibujar gráficos 2D y 3D
- Creado en 1992 por Silicon Graphics Inc y gestionado actualmente por Khronos Group.
- Existen variantes de OpenGL como OpenGL ES para dispositivos móviles o WebGL para navegadores.

OpenGL

- La API de OpenGL va evolucionando para permitir mayor control sobre el pintado y mayor eficiencia pero mantiene compatibilidad con versiones anteriores
- Tiene dos perfiles o especificaciones:
  - **Core profile**: API moderna, más
  - **Compatibility profile**: Disponible para compatibilidad con versiones antiguas.

OpenGL

- OpenGL funciona como una máquina de estados que definen como OpenGL debe operar en ese momento (su estado lo llamaremos context).
  - Por ejemplo, se puede poner un color y OpenGL dibujará todos los objetos con ese color hasta que decidamos cambiarlo. Otros valores que podemos dar a la *máquina de estados son por ejemplo el modo de mezclado de los colores, si hay test de profundidad a la hora de pintar, etc.*
- Las transformaciones de los objetos 3D (**traslación, rotación, escalado**) se realizan utilizando matrices.
- Al dibujar un objeto, se transforman sus coordenadas multiplicándolas por una serie de matrices (**Modelo, Vista, Proyección**).
- El API moderno de OpenGL (Core) no incluye funciones para el manejo de matrices. Vamos a utilizar la librería GLM para este propósito.

OpenGL

OpenGL dibuja con dos elementos básicos: texturas y vértices.

- Las texturas definen el aspecto del objeto a dibujar (por ejemplo, se pueden utilizar texturas de madera, piedra, metal…)
- Los vértices definen la forma del objeto.

Al definir los vértices de un objeto, se debe indicar cómo se conectan unos con otros para formar caras, y el tipo de primitiva que dibujan (normalmente triángulos).

Las funciones en OpenGL tienen el prefijo gl, y muchas funciones además un sufijo Con el tipo de parámetros esperado. Ejemplo:

- **glUniform1i**: Establece el valor de una variable uniforme de tipo entero en el shader.
- **glUniform3f**: Establece los valores de un vector de 3 coordenadas de tipo float en el shader.

OpenGL

Los estados se activan y desactivan con las funciones glEnable y glDisable, respectivamente:

- **GL_BLEND**: Activa la mezcla de valores de color (entre los colores a pintar y los colores existentes en el buffer).

Para establecer el modo de mezcla, se usa la función glBlendFunc.

- **GL_DEPTH_TEST**: Activa la comprobación de profundidad en el pintado
- **GL_SCISSOR_TEST**: Restringe el pintado del color buffer a la región especificada por la función glScissor.

Buffers

Buffers

- Se puede definir la geometría de los elementos a dibujar de muchas formas en OpenGL. La mayor parte de estas se considera obsoleta en versiones recientes de OpenGL.
- Nosotros trabajaremos con Vertex Buffer Objects (VBO) que se pueden utilizar tanto en OpenGL ES 1 como en ES 2.
- Los VBO permiten enviar la información geométrica de los elementos a dibujar a la memoria de la tarjeta de vídeo.
- Esto hace el acceso a estos datos en el momento del pintado mucho más rápido.

Buffers

- Para usar un VBO tenemos que:
  - Generar un nombre para el buffer (un id).
  - Activar el buffer actual (Bind).
  - Guardar los datos en el buffer.
  - Usar el buffer para renderizar los datos.
  - Destruir los buffers.

Buffers

Las funciones de OpenGL para manejar VBO son:

- **void glGenBuffers(GLsizei n, GLuint* buffers)**

Genera el número de buffers especificado por el parámetro n, y coloca sus identificadores en el array buffers, que debe tener al menos n elementos.

- **void glDeleteBuffers(GLsizei n, const GLuint* buffers)**

Elimina n buffers cuyos identificadores serán leídos del array buffers.

Buffers

- **void glBindBuffer(GLenum target, GLuint buffer)**

Establece el buffer activo. A la hora de dibujar, OpenGL recibirá los datos de este buffer.

Los valores de target con los que trabajaremos son:

GL_ARRAY_BUFFER: El buffer especificado contiene información de los vértices (coordenadas, normales, color…)

GL_ELEMENT_ARRAY_BUFFER: El buffer especificado contiene los índices a los vértices con los que se forma la geometría de los elementos a dibujar

Buffers

- **void glBufferData(GLenum target, GLsizeiptr size, const GLvoid* data, GLenum usage)**

Envía los datos al buffer. Estos datos deben estar contenidos en el array apuntado por data, y el tamaño en bytes del bloque de memoria que los contiene se especifica en size.

Un buffer de índices recibirá el array con los índices a los vértices.

Un buffer de vértices recibirá el array con los datos de los vértices (normal, color…)

El parámetro usage toma los valores:

GL_STATIC_DRAW: Solicita que los datos se carguen en memoria de vídeo. Útil cuando la información contenida en data no va a ser modificada frecuentemente (por ejemplo, mallas estáticas o buffers de índices).

GL_DYNAMIC_DRAW: Solicita que los datos permanezcan en la memoria principal. Útil cuando estos datos van a cambiar frecuentemente (por ejemplo, algunos tipos de mallas animadas).

GL_STREAM_DRAW: Los datos no cambiarán a menudo, y no se van a utilizar frecuentemente,

con lo que se volcará en memoria de vídeo cuando sea necesario.

Buffers

- **void glDrawArrays(GLenum mode, GLint first, GLsizei count)**

Dibuja la geometría especificada por el buffer de vértices enlazado No utiliza el buffer de índices (asume que se ensamblan las primitivas en el orden en que los vértices aparecen en el buffer) Sus parámetros son:

  - **mode**: Tipo de primitiva a dibujar (normalmente GL_TRIANGLES).
  - **first**: Índice del primer vértice a procesar.
  - **count**: Número de vértices a procesar.

Buffers

- **void glDrawElements(Glenum mode, GLsizei count, GLenum type, const GLvoid* indices)**

Dibuja la geometría especificada por el buffer de índices y el buffer de vértices.

Sus parámetros son:

- **mode**: Tipo de primitiva a dibujar (normalmente GL_TRIANGLES).
- **count**: Número de índices contenido en el buffer de índices.
- **type**: Formato en que están definidos los índices en el buffer (normalmente GL_UNSIGNED_SHORT o GL_UNSIGNED_INT).
- **indices**: Cuando trabajamos con VBO, este parámetro debe ser nullptr.

Buffers

Ejemplo de uso (Definiendo un triángulo):

```
VertexVec vértices = {
        Vertex(glm::vec3(0, 0.5f, 0), glm::vec3(1, 0, 0)),
        Vertex(glm::vec3(-0.5f, -0.5f, 0), glm::vec3(0, 1, 0)),
        Vertex(glm::vec3(0.5f, -0.5f, 0), glm::vec3(0, 0, 1))
};
IndexVec indices = {0, 1, 2};
```

Buffers

Ejemplo de uso (Creando el buffer):

```
std::array<uint32_t, 2> m_ids;

glGenBuffers(2, m_ids.data());


glBindBuffer(GL_ARRAY_BUFFER, m_ids[0]);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_ids[1]);


glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex)*vertices.size(), vertices.data(),
GL_STATIC_DRAW);

glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(uint16_t)*indices.size(), indices.data(),
GL_STATIC_DRAW);
```

Buffers

Ejemplo de uso (Pintando el buffer):

```
glBindBuffer(GL_ARRAY_BUFFER, m_ids[0]);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_ids[1]);


shader.setupAttribs();


glDrawElements(GL_TRIANGLES, m_numIndices, GL_UNSIGNED_SHORT, nullptr);
```

Pintado

Pintado

Los vértices de un triángulo pueden definir distintos tipos de primitivas. En esta asignatura trabajaremos con triángulos, aunque existen otros tipos:

- **GL_POINTS**: Cada vértice dibuja un punto.
- **GL_LINES**: Cada par de vértices dibujan una línea.
- **GL_LINE_STRIP**: Dibuja una serie de líneas conectadas entre el punto n y el punto n+1.
- **GL_LINE_LOOP**: Dibuja una serie de líneas conectadas entre el punto n y el punto n+1, y conectando el último
- **GL_TRIANGLES**: Cada tres vértices dibujan un triángulo.
- **GL_TRIANGLE_STRIP**: Dibuja cada triángulo con los dos últimos vértices del triángulo anterior y el siguiente.
- **GL_TRIANGLE_FAN**: Dibuja cada triángulo con el primer vértice de la lista, el actual, y el siguiente.

Pintado

Ya hemos visto cómo pintar los buffers de vértices, pero hay otras tareas de pintado que se pueden efectuar sobre el color buffer:

- **void glViewport(GLint x, GLint y, GLsizei width, GLsizei height)**

Indica el rectángulo de la pantalla donde se volcará el color buffer. Si queremos que se dibujen a pantalla completa, pasaremos 0,0 como coordenadas de origen, y el ancho y alto de la pantalla como tamaño del rectángulo.

- **void glScissor(GLint x, GLint y, GLsizei width, GLsizei height)**

Si el estado GL_SCISSOR_TEST está activado, excluye del pintado (recorta) las partes del viewport que quedan fuera de este área. Normalmente se le dan los mismos valores que a glViewport.

Pintado

- **void glClearColor(GLclampf red, GLclampf green, GLclampf blue, Glclampf alpha)**

Define el color con el que se limpiará el color buffer. Normalmente se limpia antes de pintar el siguiente frame.

- **void glClear(GLbitfield mask)**

Limpia el buffer pasado como parámetro. Puede ser(se pueden combinar con |):

  - **GL_COLOR_BUFFER_BIT**: Limpia el buffer de color o backbuffer.
  - **GL_DEPTH_BUFFER_BIT**: Limpia el buffer de profundidad.
  - **GL_STENCIL_BUFFER_BIT**:Limpia el buffer de recorte (no lo utilizamos).

Pintado

Cuando se realiza el pintado de elementos, OpenGL tiene dos modos:

- **Flujo de pintado fijo** : Las operaciones de transformación de los vértices y rasterizado de fragmentos está preestablecida.
- **Shaders**: Codificamos la transformación de los vértices en un vertex shader, y el rasterizado de fragmentos en un fragment shader.

El flujo de pintado fijo se utiliza únicamente en el perfil de compatibilidad, así que nos limitaremos a renderizado con shaders.

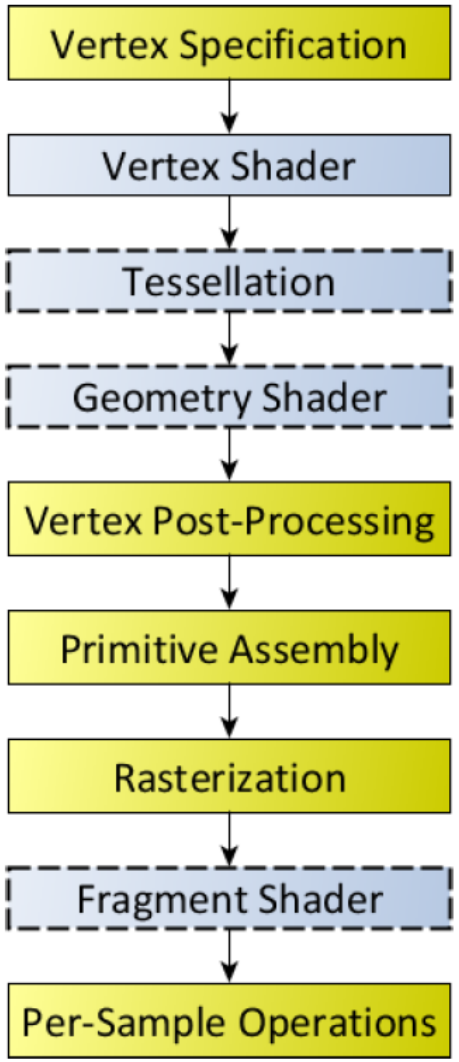
Shaders

Shaders

- En este tema, abordaremos los shaders, el lenguaje GLSL, y cómo se introducen éstos en el pipeline de OpenGL.
- Un shader es un programa compilable que se ejecuta en la GPU.
- Permite hacer programables tareas de las que anteriormente se encargaba el pipeline fijo de la API de renderizado.
- Los lenguajes de shaders más utilizados son:
  - Cg, de Nvidia, compatible con OpenGL y Direct3D.
  - HLSL, de Microsoft, compatible con Direct3D.
  - GLSL, del Khronos Group, compatible con OpenGL, OpenGL ES, WebGL y Vulkan.

En esta asignatura trataremos GLSL.

Shaders


Vertex Specification
Vertex Shader
Tessellation
Geometry Shader
Vertex Post-Processing
Primitive Assembly
Rasterization
Fragment Shader
Per-Sample Operations

Shaders

- Ya hemos visto un esquema del funcionamiento del pipeline fijo de OpenGL.
  - Los vertex shaders permiten programar el proceso de transformación de los vértices.
  - Los geometry shaders, tessellation control shaders y tessellation evaluation shaders se encargan del ensamblado de fragmentos (permiten definir nuevas primitivas y modificar las existentes).
  - Los fragment shaders se encargan del proceso de coloreado y texturizado de fragmentos.

En la asignatura, veremos vertex y fragment shaders.

- Cuando utilizamos shaders, es necesario definir varias etapas del proceso de renderizado, por lo que necesitaremos varios shaders (al menos el shader de vértices y el shader de fragmentos).

Toda la serie de shaders que definen un proceso de renderizado completo, sean del tipo que sean, forman un programa.

Shaders

- **GLuint glCreateShader(Glenum shaderType)**

Crea un nuevo shader. El valor de shaderType puede ser:

  - GL_VERTEX_SHADER
  - GL_FRAGMENT_SHADER

- **void glDeleteShader(GLuint shader)**

Elimina un shader. Una vez que un shader ha sido enlazado en un programa, podemos eliminarlo.

Shaders

- **Void glShaderSource(GLuint shader, GLsizei count, const GLchar** string, const GLint* length)**

Define el código fuente de un shader. El código puede estar dividido en varios strings.

El valor de count será el número de strings.

El valor de string será un puntero al array de strings, y length será un array con la longitud de cada string. Si los strings son terminados en cero (el formato estándar de C), se puede pasar nullptr como último parámetro.

Si queremos definir el shader desde un único string, lo haremos de la forma:

```
char* codigo
glShaderSource(shader, 1,&código, nullptr);
```

Shaders

- **void glCompileShader(Gluint shader)**

Compila el código fuente asignado a un shader. Para comprobar si la compilación ha sido correcta, haremos lo siguiente:

```
Glint retCode;
Char errorLog[1024];
glCompileShader(shader);
glGetShaderiv(shader, GL_COMPILE_STATUS, &retCode);
if (retCode ==GL_FALSE )
{
glGetShaderInfoLog(shader, sizeof(errorLog), NULL, errorLog);
}
```

Shaders

Ejemplo de creación de Vertex shader:

```
int retCode;

const char* cVertexShaderSource =  vertex.c_str();

uint32_t vertexShader = glCreateShader(GL_VERTEX_SHADER);

glShaderSource(vertexShader, 1,&cVertexShaderSource, nullptr);

glCompileShader(vertexShader);

glGetShaderiv(vertexShader, GL_COMPILE_STATUS, STATUS,&retCode);

If (retCode == GL_FALSE){

char errorLog[1024];

glGetShaderInfoLog(vertexShader, sizeof(errorLog), nullptr, errorLog);

m_error = errorLog;

glDeleteShader(vertexShader);

return;

}
```

Shaders

- **Gluint glCreateProgram(void)**

Crea un nuevo programa.

- **Void glDeleteProgram(GLuint program)**

Elimina el programa pasado como parámetro.

- **Void glAttachShader(GLuint program, GLuint shader)**

Añade un shader (previamente compilado) al programa especificado.

Debemos añadir al menos un vertex shader y un fragment shader.

Shaders

- **void glLinkProgram(GLuint program)**

Enlaza todos los shaders de un programa para crear el programa ejecutable final. Para comprobar si el enlazado ha sido correcto, haremos lo siguiente:

```
GIint retCode;
char errorLog[1024];
glLinkProgram(program);
glGetProgramiv(program, GL_LINK_STATUS, STATUS,&retCode);
If (retCode == GL_FALSE ){ glGetProgramInfoLog(program, sizeof(errorLog),
NULL, errorLog);}
```

- **Void glUseProgram(GLuint program)**

Reemplaza las tareas del pipeline de renderizado de OpenGL por las definidas en el programa.

Shaders

Ejemplo de creación de programa

```
m_id = glCreateProgram();
glAttachShader(m_id, vertexShader);
glAttachShader(m_id, fragmentShader);
glLinkProgram(m_id);
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
glGetProgramiv(m_id, GL_LINK_STATUS, &retCode);
If  (retCode == GL_FALSE){
        char errorLog[1024];
        glGetProgramInfoLog(m_id, sizeof(errorLog), nullptr, errorLog);
        m_error = errorLog;
        glDeleteProgram(m_id);
        m_id = 0;
        return;
}
```

Shaders: Vertex

- En el proceso de transformación de los vértices, se recibe cada vértice en el espacio del modelo, y debe devolverse en el espacio de proyección.
- El vertex shader se ejecuta una vez por vértice.
- Para realizar la transformación, se utilizan las matrices modelo, vista y proyección.

En muchos casos, se puede pasar al shader una matriz MVP con las tres transformaciones aplicadas.

- *Para calcular la iluminación, se suele trabajar en el espacio del observador, así que podemos pasar también una matriz MV.*

Shaders: Vertex

Un ejemplo de shader que realiza la transformación de vértices estándar:

```
uniform mat 4 MVP;
attribute vec 3 vpos;
void main(){
gl_Position =  MVP * vec4(vpos, 1);
}
```

gl_Position es una variable predefinida que define la posición final de un vértice del vertex shader.

Con attribute indicamos una variable que define datos de un vértice, y cuyo valor se lee del buffer de vértices enlazado.

Más adelante veremos el significado de uniform.

Shaders: Vertex

- **Glint glGetAttribLocation(GLuint program, const GLchar* name)**

Devuelve la localización (el identificador) de la variable attribute llamada name del vertex shader del programa indicado. Si la variable no se ha encontrado, devolverá -1.

Aunque hayamos definido una variable, si ésta no se utiliza en el shader, puede ser descartada por GLSL.

m_vposLoc = glGetAttribLocation(m_id,“vpos”);

Shaders: Vertex

- **Void glEnableVertexAttribArray(GLuint index)**

Activa la escritura en una variable attribute cuya localización se pasa en el parámetro index.

- **Void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid* pointer)**

Especifica cómo leer del buffer de vértices los datos de una variable attribute. Sus argumentos son:

  - ***Index**: Localización de la variable.*
  - ***Size:** Número de componentes a escribir por vértice (3 para posiciones y normales, 4 para colores, 2 para coordenadas de textura).*
  - ***Type:** Tipo con el que le pasamos los datos (por ejemplo, GL_FLOAT, GL_UNSIGNED_BYTE…).*
  - ***Normalized:** Indica si se deben normalizar los datos enviados. Por ejemplo, si definimos los colores con valores [255], se normalizarán automáticamente al rango [1] si este parámetro vale true.*
  - ***Stride:** Tamaño del paquete de datos completo para un vértice (el sizeof de la estructura donde guardamos los datos de un vértice).*
  - ***Pointer:** Offset dentro del buffer activo en que se encuentra el primer dato (es un entero que debemos castear a void*).*

Shaders: Vertex

```
If (m_vposLoc != -1)
{
        glEnableVertexAttribArray(m_vposLoc);
        glVertexAttribPointer(m_vposLoc, 3, GL_FLOAT, false, sizeof(Vertex), reinterpret_cast<const
                void*>(offsetof(Vertex, position)));
}
```

Shaders: Fragment

- El fragment shader recibirá los fragmentos que forman cada una de las primitivas que pasamos al proceso de renderizado.
- Es necesario pasar información desde el vertex shader hacia el fragment shader.
- Una variable usa el atributo attribute cuando es un valor de entrada al vertex shader, y varying cuando es de salida en el vertex shader y de entrada en el fragment shader (se utiliza para pasar datos de uno a otro).
- Como el vertex shader se ejecuta por vértice y el fragment shader por fragmento, para calcular el valor que recibe cada fragmento de las variables varying, se realiza una interpolación de los datos.
- Versiones más modernas de GLSL utilizan in para variables de entrada en cualquier shader, y out para variables de salida (esta sintaxis no es compatible con OpenGL ES 2 ni WebGL 1).

Shaders: Fragment

Ejemplo: el vertex shader pasa al fragment shader el color del vértice.

Para los fragmentos que no corresponden con ninguno de los vértices, el color recibido será la interpolación de los valores de color de todos los vértices de la primitiva.

Shaders: Fragment

Código del vertex shader:

```
uniform mat4 MVP;
attribute vec3 vpos;
attribute vec4 vcolor;
varying vec4 fcolor;
void main(){
gl_Position = MVP * vec4(vpos, 1);
fcolor = vcolor;
}
```

Código del fragment shader:

```
varying vec4 fcolor;
void main() {
gl_FragColor = fcolor; };
```

- La variable gl_FragColor del fragment shader guarda el color final del fragmento (del valor de salida del shader). Tiene tipo vec4 (vector de 4 coordenadas RGBA.

Shaders: Fragment

- Hemos explicado dos modificadores para las variables de los shaders:
  - **Attribute** son valores que se recibirán por cada vértice.
  - **Varying** son variables de salida en el vertex shader y de entrada en el fragment shader (los valores que salen se interpolan para generar la entrada en la siguiente fase).
- En los ejemplos se han visto otro tipo de variables, la variables **uniform** o uniformes.

El valor de estas variables se define desde la aplicación (en nuestro código en C++), y no varían durante toda la ejecución de los shaders (es decir, todos los vértices y fragmentos recibirán el mismo valor para estas variables)

- Un caso típico de variable uniform es la matriz MVP, que creamos antes de renderizar y pasamos a los shaders.

Shaders: Fragment

- **GLint glGetUniformLocation(GLuint program, const GLchar* name)**

Devuelve la localización (el identificador) de la variable uniform llamada name del programa indicado. Si la variable no se ha encontrado, devolverá -1.

- **void glUniform1i(GLint location, GLint v0)**
- **void glUniform1f(GLint location, GLfloat v0)**
- **Void glUniform2f(GLint location, GLfloat v0, GLfloat v1)**
- **Void glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2)**
- **Void glUniform4f(GLint location, GLfloat v 0 GLfloat v 1 GLfloat v 2 GLfloat v3)**
- **void glUniformMatrix 4 fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat* value)**

Establece el valor o valores de una variable uniform. Según el tipo de variable, se utilizará una función u otra.

Ejemplos: bool, int: glUniform1i

float: glUniform1f

vec3: glUniform3f

vec4: glUniform4f

mat4: glUniformMatrix4fv (con count =1)

Shaders: Fragment

- Nuestro motor implementará una clase Shader para representar un programa formado a partir de un vertex y un fragment shader (cuyo código se pasará al objeto en la construcción).
- Tendremos métodos para configurar las variables attribute, y establecer el valor de cada una de las variables uniform.
- Podremos activar también el uso del shader para las subsiguientes tareas de pintado.

¿Dudas?