Universidad de Huelva

Tema 4

El lenguaje GLSL

Índice

- A la hora de desarrollar un programa en OpenGL es necesario tener en cuenta que debemos programar en varios planos:
  - Por un lado utilizaremos las funciones de OpenGL como las de cualquier otra librería incluida en un programa escrito en C/C++.
  - Por otro lado incluiremos fragmentos de código a insertar en los shaders. Este código utiliza el lenguaje GLSL. De momento solo vamos a utilizar el Vertex_Shader y el Fragment_Shader.
  - Para distinguir los diferentes tipos de código vamos a utilizar distintos colores de fondo.

Código incluido en el programa fuente. A ejecutar en la CPU.	Código incluido en el Vertex Shader. A ejecutar en la GPU (tarjeta gráfica).	Código incluido en el Fragment Shader. A ejecutar en la GPU (tarjeta gráfica).

- Shaders
  - Para crear un shader se utiliza el comando

```
Gluint shader = glCreateShader( type );
```

  - donde *type* puede ser GL_VERTEX_SHADER, GL_FRAGMENT_SHADER, GL_GEOMETRY_SHADER, GL_TESS_CONTROL_SHADER o GL_TESS_EVALUATION_SHADER.
  - A continuación hay que cargar el código del shader. El código se añade en modo texto. El argumento *codeArrays* contiene un array de cadenas (GLchar*) con el texto. El argumento *numLines* indica el número de elementos de *codeArrays* y *length* es un array con la longitud de cada línea.

```
glShaderSource( shader, numLines, codeArrays, length);
```

- Shaders
  - Una vez cargado el código, hay que compilarlo

```
glCompileShader( shader );
```

  - Se puede comprobar el resultado de la compilación mediante el comando

```
GLint status;
glGetShaderiv( shader, GL_COMPILE_STATUS,&status);
```

  - En caso de error (status == GL_FALSE), se puede obtener la descripción

```
glGetShaderInfoLog( … )
```

  - Para eliminar un shader se utiliza

```
glDeleteShader( shader );
```

- Ejemplo de Vertex_Shader

```
#version 400

in vec3 VertexPosition;
in vec3 VertexColor;

out vec3 Color;

void main()
{
    Color = VertexColor;
    gl_Position = vec4( VertexPosition, 1.0);
}
```

- Ejemplo de Fragment_Shader

```
#version 400

in vec3 Color;

out vec4 FragmentColor;

void main()
{
    FragmentColor = vec4( Color, 1.0);
}
```

- Programas:
  - Para configurar el proceso de renderizado programable es necesario crear un *programa*. Un programa es, básicamente, un contenedor de shaders.

```
Gluint programHandle = glCreateProgram();
```

  - Una vez creado el programa, es necesario añadirle los shaders. Para que el programa sea correcto, al menos debe contener el Vertex_shader y el Fragment_shader.

```
glAttachShader( programHandle, vertexShader);
glAttachShader( programHandle, fragmentShader);
```

  - Una vez añadidos los shaders, hay que linkar el programa:

```
glLinkProgram( programHandle );
```

- Programas:
  - Si el linkado no da problemas, ya se puede instalar el programa en el pipeline de renderizado:

```
glUseProgram( programHandle );
```

  - Se puede conocer el estado de linkado con el comando

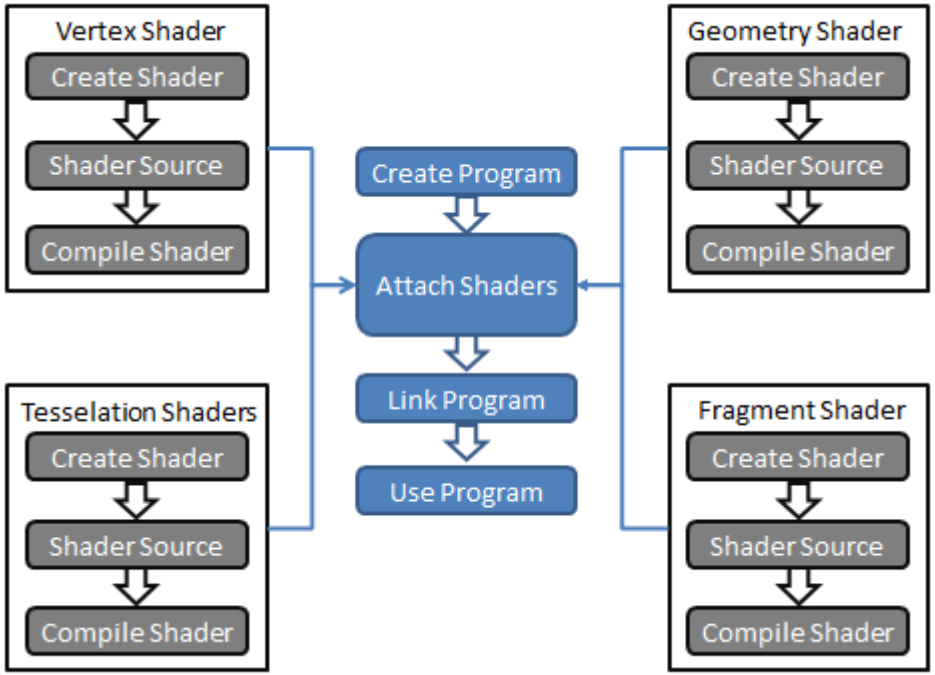
```
GLint status;
glGetProgramiv(programHandle, GL_LINK_STATUS,&status);
```

  - En caso de error (status == GL_FALSE), se puede obtener el mensaje de error

```
glGetProgramInfoLog( … )
```

- Programas
  - Para eliminar un programa, liberando la memoria que ocupa en la GPU, se utiliza el comando

```
glDeleteProgram( programHandle );
```



- Ejemplo de creación de un programa

```
public GLboolean Init() {
  GLuint status;
  char** vertexShaderSource = …;
  char** fragmentShaderSource = …;

  // Crea y compila el vertex shader
  GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
  glShaderSource(vertexShader, 1, vertexShaderSource, NULL);
  glCompileShader(vertexShader);
  glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &status);
  if (status == GL_FALSE) {
    GLint length;
    glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &length);
    char* logInfo = (char *)malloc(sizeof(char)*length);
    GLsizei written;
    glGetShaderInfoLog(vertexShader, length, &written, logInfo);
    return GL_FALSE;
  }
…
```

- Ejemplo de creación de un programa (sigue)

```
…

   // Crea y compila el fragment shader

   GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
   glShaderSource(fragmentShader, 1, fragmentShaderSource, NULL);
   glCompileShader(fragmentShader);
   glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &status);
   if (status == GL_FALSE)
   {
     GLint length;
     glGetShaderiv(fragmentShader, GL_INFO_LOG_LENGTH, &length);
     char* logInfo = (char *)malloc(sizeof(char)*length);
     GLsizei written;
     glGetShaderInfoLog(fragmentShader, length,&written,logInfo);
     return GL_FALSE;
   }

…
```

- Ejemplo de creación de un programa (fin)

```
…
   //Crea y enlaza el programa

   GLuint program = glCreateProgram();
   glAttachShader(program, vertexShader);
   glAttachShader(program, fragmentShader);
   glLinkProgram(program);
   glGetProgramiv(program, GL_LINK_STATUS, &status);
   if (status == GL_FALSE)
   {
     GLint length;
     glGetProgramiv(program, GL_INFO_LOG_LENGTH, &length);
     char* logInfo = (char *)malloc(sizeof(char)*length);
     GLsizei written;
     glGetProgramInfoLog(program, length, &written, logInfo);
     return GL_FALSE;
   }
   return GL_TRUE;
 }
```

- La estructura básica de un shader sigue el siguiente esquema:

```
#version VERSION_DE_GLSL_UTILIZADA

DECLARACIONES

void main()
{
    CÓDIGO
}
```

Las declaraciones pueden ser:

- Declaraciones de variables de entrada (***in***):
  - Definen las entradas del shader.
  - Cada shader tiene además una serie de entradas predefinidas que no hay que declarar.
  - En el Vertex_Shader las entradas corresponden a los atributos de cada vértice.
  - En el Fragment_Shader las entradas corresponden a los atributos de cada pixel, calculados mediante interpolación de las salidas del Vertex_Shader.

Las declaraciones pueden ser:

- Declaraciones de variables de salida (***out***).
  - Definen las salidas del shader.
  - Cada shader tiene además una serie de salidas predefinidas que no hay que declarar.
  - Las salidas del Vertex_Shader deben corresponder a las entradas del Fragment_Shader. El proceso de linkado del programa verifica esta condición.
  - EL Fragment_Shader debe tener una salida de tipo vec4 que corresponde al color del pixel.

Las declaraciones pueden ser:

- Declaraciones de variables uniformes (***uniform***):
  - Corresponden a variables que tienen un valor común en todas las ejecuciones del shader.
  - Por ejemplo, el Vertex_Shader se ejecuta sobre cada vértice de manera que en cada ejecución los valores de las variables de entrada serán diferentes (los valores de los atributos de cada vértice). Sin embargo las variables uniformes tendrán el mismo valor en todas las ejecuciones.
  - Se suelen utilizar, por ejemplo, para almacenar matrices de transformación de coordenadas.

Las declaraciones pueden ser:

- Declaraciones de funciones auxiliares:
  - Son funciones que pueden ser llamadas desde el bloque principal del shader.
  - Se declaran de la misma forma que las funciones normales de C.
- Declaraciones de subrutinas (***subrutine***):
  - Permiten declarar tipos de funciones auxiliares (interfaces).
  - Se utilizan para trabajar con "punteros a funciones".

- El bloque main() describe el código a ejecutar en el shader.
  - Debe asignar los valores de las variables de salida.
  - Puede utilizar los valores de las variables de entrada, de las variables uniformes o de variables locales definidas en el propio bloque.
  - Puede incluir llamadas a las funciones auxiliares declaradas en el shader.
  - Puede incluir llamadas a funciones predefinidas de GLSL.
  - El código es parecido al lenguaje C, aunque maneja tipos de datos optimizados para vectores y matrices y operadores específicos para estos tipos de datos.

- Tipos de datos básicos en GLSL:
  - **void**: tipo de dato de las funciones que no devuelven nada.
  - **bool**: tipo de dato booleano. Puede tomar los valores **true** y **false**.
  - **int**: tipo de dato entero con signo de 32 bits.
  - **uint**: tipo de dato entero sin signo de 32 bits. Los literales de tipo uint se marcan con el sufijo 'u', por ejemplo, "255u".
  - **float**: tipo de datos en coma flotante de 32 bits. Los literales de tipo float no llevan sufijo. Por ejemplo, "0.5" es un valor de tipo float.
  - **double**: tipo de datos en coma flotante de 64 bits. Los literales de tipo double utilizan el sufijo 'LF'. Por ejemplo, "0.5LF" es un valor de tipo double.

- Tipos de datos vectoriales en GLSL:
  - **vec2, vec3, vec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *float*.
  - **dvec2, dvec3, dvec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *double*.
  - **ivec2, ivec3, ivec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *int*.
  - **uvec2, uvec3, uvec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *uint*.
  - **bvec2, bvec3, bvec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *bool*.
- Se puede acceder a los componentes del vector mediante los campos ***x***, ***y***, ***z*** y ***w***. Estos campos también se pueden nombrar como ***r,g,b,a*** o ***s,t,p,q***. Puede accederse a trozos del vector mediante los campos ***xy*** o ***xyz***.

- Tipos de datos matriciales en GLSL:
  - **mat2, mat3, mat4**: tipos de datos que describen matrices cuadradas 2x2, 3x3 y4x4 de tipo *float*.
  - **mat2x2, mat2x3, mat2x4, mat3x2, mat3x3, mat3x4, mat4x2, mat4x3, mat4x4**: tipos de datos que describen matrices de tipo float de distintos tamaños.
  - **dmat2, dmat3, dmat4**: tipos de datos que describen matrices cuadradas 2x2, 3x3 y4x4 de tipo *double*.
  - **dmat2x2, dmat2x3, dmat2x4, dmat3x2, dmat3x3, dmat3x4, dmat4x2, dmat4x3, dmat4x4**: tipos de datos que describen matrices de tipo *double* de distintos tamaños.
- Se puede acceder a los campos de una matriz como si fuera un array. Por ejemplo, si *m* es una variable de tipo *mat4*, *m[0]* es un valor de tipo *vec4* y *m[0][0]* es un valor de tipo *float*.

- Arrays:
  - Se pueden definir arrays sobre cualquier tipo de datos.
  - La declaración es de la forma "*tipo nombre [tamaño]*".
  - Por ejemplo, "*vec4 points[10];*" define un array de 10 vectores.
  - Tambien se puede dejar sin declarar el tamaño. "*vec4 points[];*"
- Estructuras:
  - Se pueden definir estructuras formadas por campos.

```
struct light {
  float intensity;
  vec3 position;
} lightVar;
```

- Tipos de datos opacos:
  - Se refieren a tipos de datos que almacenan objetos que no pueden ser modificados. Sólo se puede acceder a la información que contienen.
  - Las texturas se declaran como tipos de datos opacos denominados sampler. Existen diferentes tipos de datos para los diferentes tipos de texturas: ***sampler1D***, ***sampler2D***, ***sampler3D***, ***samplerCube***, ***sampler1DShadow***, ***sampler2DShadow***.
  - Las imágenes tambien se almacenan como tipos opacos. Se declaran como ***image1D*** o ***image2D***.

- Constructores:
  - Para inicializar un vector se puede utilizar una lista de valores de sus componentes. Por ejemplo, "*vec4 v = { 0.1, 0.2, 0.3, 0.4 }; *"
  - También se puede utilizar un constructor con los valores de sus compoenentes. Por ejemplo, "*vec4 v = vec4(0.1, 0.2, 0.3, 0.4);*".
  - Si se utiliza un único número en el constructor, todos los componentes se asignan a ese valor. Por ejemplo, "*vec4 v = vec4(1.0);*".
  - Los constructores de vectores tambien pueden utilizar como argumento a otros vectores de mayor tamaño. En ese caso se eliminan las compoenentes sobrantes. Por ejemplo, "*vec4 v = vec4(0.1,0.2,0.3,0.4); vec3 w = vec3(v); *". El vector w solo copia los tres primeros componentes.
  - Hay versiones de constructores que toman como argumento vectores mas pequeños y los componentes que falta. Por ejemplo, "*vec3 w=vec3(0.1,0.2,0.3); vec4 v = vec4(w, 0.4);*"

- Constructores:
  - Para inicializar matrices se pueden utilizar listas de listas. En ese caso cada lista corresponde a una columna de la matriz. Por ejemplo, "*mat2 m = {{0.1,0.2},{0.3,0.4}};*".
  - Se puede inicializar una matriz diagonal indicando un único valor en el constructor. Por ejemplo, "*mat3 m = mat3(1.0);*".
  - También se pueden utilizar constructores basados en vectores. Cada vector corresponde al contenido de una columna. Por ejemplo, "*mat3(vec3,vec3,vec3)*".
  - Se pueden utilizar constructores basados en una única lista de componentes. En ese caso se asume que la matriz se rellena por columnas. Por ejemplo, "*mat2 m = mat2(0.1,0.2,0.3,0.4);*" es lo mismo que "*mat2 m = {{0.1,0.2},{0.3,0.4}};*".

- Operadores
  - Los operadores aritméticos incluidos son la suma (+), la resta (-), la multipliación (*), la división (/) y el módulo (%). El módulo solo se aplica a datos *int* y *uint*.
  - Estos operadores se pueden aplicar a dos escalares, con el significado habitual.
  - Si se aplican a un escalar y un vector, el resultado es un vector y el operador se aplica de manera independiente a cada componente. Por ejemplo, "*2*vec3(1.0,2.0,3.0)*".
  - Si se aplican a un escalar y una matriz, el resultado es una matriz y el operador se aplica de manera independiente a cada componente. Por ejemplo, "*mat3(1.0) / 2*".
  - Si se aplican a dos vectores (del mismo tamaño) el operador se aplica componente a componente. Por ejemplo, "*vec3(1.0,2.0,3.0) + vec3(0.1,0.2,0.3)*".
  - Si se aplica entre dos matrices, los operadores suma (+), resta (-) y división (/) se aplican componente a componente. Si embargo, el producto (*) aplicado a matrices o a un vector y una matriz se trata como un producto matricial.
  - Para calcular el producto escalar de dos vectores se utiliza la función *dot()*. Para calcular el producto vectorial se utiliza la función *cross()*.

- El código de los shaders no permite incluir librerías externas. Sin embargo, el estándar incluye un gran número de funciones predefinidas que pueden incluirse en el código. (Ver la descripción de las funciones en el manual)

Angle and Trigonometry Functions					
radians	degrees	sin	cos	tan	asin
acos	atan	sinh	cosh	tanh	asinh
acosh	atanh				

Exponential Functions					
pow	exp	log	exp2	log2	sqrt
inversesqrt					

Common Functions					
abs	sign	floor	trunc	round	roundEven
ceil	fract	mod	modf	min	max
clamp	mix	step	smoothstep	isnan	isinf
floatBitsToInt	intBitsToFloat	fma	frexp	ldexp	

Floating-Point Pack and Unpack Functions			
packUnorm2x16	packSnorm2x16	packUnorm4x8	packSnorm4x8
unpackUnorm2x16	unpackSnorm2x16	unpackUnorm4x8	unpackSnorm4x8
packDouble2x32	unpackDouble2x32	packHalf2x16	unpackHalf2x16

Geometric Functions				
length	distance	dot	cross	normalize
faceforward	reflect	refract	ftransform	

Matrix Functions				
matrixCompMult	outerProduct	transpose	determinant	inverse

Vector Relational Functions				
lessThan	greaterThan	equal	all	any
lessThanEqual	greaterThanEqual	notEqual	not	

Integer Functions				
uaddCarry	umulExtended	bitfieldInsert	bitfieldReverse	findLSB
usubBorrow	imulExtended	bitfieldExtract	bitCount	findMSB

Texture Functions			
textureSize	textureSamples	textureQueryLod	textureQueryLevels
texture	textureProj	textureLod	textureProjLod
textureOffset	textureProjOffset	textureLodOffset	textureProjLodOffset
textureGrad	textureProjGrad	textureGather	texelFetch
textureGradOffset	textureProjGradOffset	textureGatherOffset	texelFetchOffset
texture1D	texture1DProj	texture1DLod	texture1DProjLod
texture2D	texture2DProj	texture2DLod	texture2DProjLod
texture3D	texture3DProj	texture3DLod	texture3DProjLod
textureCube	textureCubeLod		

Atomic Counter Functions		
atomicCounterIncrement	atomicCounterDecrement	atomicCounter

Atomic Memory Functions				
atomicAdd	atomicMin	atomicMax	atomicAnd	atomicOr
atomicXor	atomicExchange	atomicCompSwap		

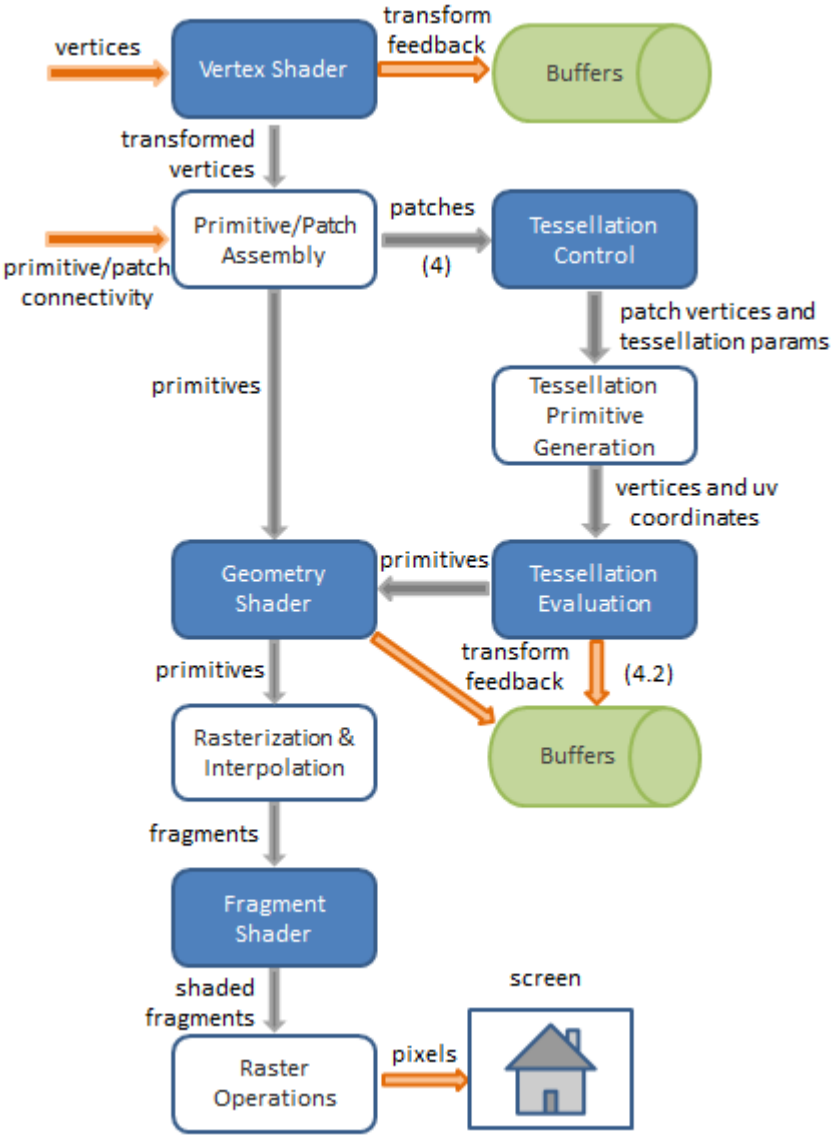
Image Functions			
imageSize	imageSamples	imageLoad	imageStore
imageAtomicAdd	imageAtomicMin	imageAtomicMax	imageAtomicAnd
imageAtomicOr	imageAtomicXor	imageAtomicExchange	imageAtomicCompSwap

Fragment Processing Functions		
dFdx	dFdy	fwidth
dFdxFine	dFdyFine	fwidthFine
dFdxCoarse	dFdyCoarse	fwidthCoarse
interpolateAtCentroid	interpolateAtSample	interpolateAtOffset

Geometry Shader Functions			
EmitStreamVertex	EndStreamPrimitive	EmitVertex	EndPrimitive

Shader Control Functions			
barrier	memoryBarrier	memoryBarrierAtomicCounter	memoryBarrierBuffer
memoryBarrierShared		memoryBarrierImage	groupMemoryBarrier

- Una vez activado un programa en la GPU (*glUseProgram*), el proceso de renderizado requiere la introducción de los datos utilizados en las distintas etapas.
- La ejecución de todo el proceso se lanza al solicitar el dibujo de un conjunto de primitivas (*glDraw…*).
- En ese momento comienza a leerse los datos de los vértices y a ejecutar todos las etapas del pipeline.
- El resultado final es el color de los píxeles afectados por el proceso, que se almacenan en un *ColorBuffer*.



- La introducción de datos en el proceso de renderizado se hace por los siguientes medios:
  - **Atributos de los vértices**: configurados por medio de *Vertex Array Objects* (VAO) y almacenados en la memoria de la GPU por medio de *Vertex Buffer Objects* (VBO).
  - **Variables uniformes**: declaradas en los shaders y asignadas mediante comandos *glUniform…().*
  - **Primitivas a dibujar**: indicadas en los comandos *glDrawArrays()* o *glDrawElements()*.
  - **Texturas**: almacenadas como objetos *sampler* y accesibles en el Fragment_Shader.
  - **Configuración de opciones**: que determinan el comportamiento de las etapas no programables, activadas con *glEnable()* y configuradas con comandos específicos.

- Atributos de los vértices
  - Para generar los valores de entrada del Vertex_shader se utiliza una estructura denominada *Vertex Array Object* (VAO), que define el mapeo entre los atributos y los nombres de las entradas del shader.
  - En primer lugar hay que asignar la posición de las entradas. Estas funciones deben ejecutarse antes de linkar el programa.
  - Por ejemplo, si definimos las variables de entrada *VertexPosition* y *VertexColor* en el Vertex_Shader, podemos asignarles la posición 0 y 1 con los siguientes comandos:

```
glBindAttribLocation(program, 0, "VertexPosition");
glBindAttribLocation(program, 1, "VertexColor");
```

- Vertex Buffer Objects (VBO)
  - A continuación hay que crear los *Vertex Buffer Objects* y almacenar sus datos.
  - Para generar los VBOs se utiliza el siguiente comando:

```
// crea n VBOs y almacena sus identificadores en el array.
void glGenBuffers(GLsizei n, GLuint * vboIDs);
```

  - Se pueden eliminar VBOs con el comando

```
// elimina n VBOs referenciados en los elementos del array.
void glDeleteBuffers(GLsizei n, const GLuint *vboIDs);
```

  - Para saber si un cierto identificador corresponde a un VBO se utiliza

```
// verifica si un identificador corresponde a un VBO
GLboolean glIsBuffer(GLuint vboID);
```

- Vertex Buffer Objects (VBO)
  - Una vez creado el VBO hay que activarlo para poder utilizarlo. Al activarlo se indica el tipo de buffer (*target*). Existen muchos tipos de buffers. Para almacenar los atributos de los vértices se utiliza GL_ARRAY_BUFFER . Para almacenar una lista de índices a vértices se utiliza el tipo GL_ELEMENT_ARRAY_BUFFER.

```
// activa un determinado VBO
void glBindBuffer(GLenum target, GLuint vboID);
```

- Vertex Buffer Objects (VBO)
  - Tras activar el buffer se puede almacenar su contenido.

```
// almacena los datos en el VBO activo
void glBufferData(GLenum target, GLsizeiptr size,
        const GLvoid * data, GLenum usage);
```

  - *target* indica el tipo de buffer a rellenar.
  - *size* indica el tamaño en bytes a almacenar en el buffer.
  - *data* es el puntero a los datos.
  - *usage* indica el uso que se va a dar al buffer para que la GPU decida el lugar más apropiado para almacenar esos datos. Puede ser GL_STATIC_DRAW, GL_DINAMIC_DRAW y GL_STREAM_DRAW.

- Vertex Array Objects (VAO)
  - Una vez creados los VBOs y almacenados los datos de los vértices el siguiente paso es crear los *Vertex Array Objects* (VAO). Se trata de estructuras de datos que definen el mapeo entre los atributos (vinculados a VBOs) y los nombres de las entradas del shader.
  - Para crear un VAO se utiliza el comando

```
// crea n VAOs y almacena sus identificadores en el array.
void glGenVertexArrays(GLsizei n, GLuint *vaoIDs);
```

  - Se puede eliminar un VAO por medio del comando

```
// elimina n VAOs referenciados en los elementos del array.
void glDeleteVertexArrays(GLsizei n, const GLuint *vaoIDs);
```

- Vertex Array Objects (VAO)
  - Para saber si un determinado identificador corresponde a un VAO se usa

```
// verifica si un identificador corresponde a un VAO
GLboolean glIsVertexArray(GLuint vaoID);
```

  - Una vez creado, se puede activar un VAO con el siguiente comando.

```
// activa un determinado VAO
void glBindVertexArray(Gluint vaoID);
```

  - Las llamadas a las primitivas lanzan el proceso de renderizado utilizando los valores de los atributos del VAO activo.

- Vertex Array Objects (VAO)
  - Para definir un atributo perteneciente a un VAO se usa

```
void glVertexAttribPointer(GLuint index,
        GLint size,
        GLenum type,
        GLboolean normalized,
        GLsizei stride,
        const GLvoid * pointer);
```

donde *index* es el índice del atributo en el VAO, *size* es el número de datos por cada atributo (1,2,3 o 4), *type* es el tipo de dato, *normalized* indica si los valores deben normalizarse o no, *stride* es el offset (número de bytes) entre los valores de dos atributos consecutivos y *pointer* es el offset del primer atributo respecto del comienzo del buffer.

  - Los datos se toman desde el buffer activo (glBindBuffer).

- Vertex Array Objects (VAO)
  - La función *glVertexAttribPointer()* asume que los datos se transformarán al tipo GLfloat. Si la opción de normalizar está activa, los datos enteros se transformarán en reales en el rango [-1,1] para valores con signo (GLint, GLshort,…) o en el rango [0,1] para valores sin signo (GLuint, GLushort, …).
  - Para configurar atributos cuyos valores deban tratarse como enteros se utiliza

```
void glVertexAttribIPointer(index,size,type,stride,pointer);
```

  - Para configurar atributos cuyos valores deban tratarse como double se utiliza

```
void glVertexAttribLPointer(index,size,type,stride,pointer);
```

- Vertex Array Objects (VAO)
  - El índice del atributo se puede asignar antes de linkar el programa por medio del comando

```
void glBindAttribLocation(GLuint program,
        GLuint index,
        const GLchar *name);
```

donde *name* es el nombre que se asigna al atributo en el VertexShader.

  - También se puede buscar el índice de un determinado atributo con

```
GLint glGetAttribLocation( GLuint program,
        const GLchar *name);
```

  - Aunque lo más sencillo es asignar directamente el índice dentro del programa del VertexShader mediante el modificador **layout**.

```
    layout(location = 0) in vec3 VertexPosition;
```

- Vertex Array Objects (VAO)
  - Para obtener información acerca de un cierto atributo que está activo un programa, se utiliza

```
void glGetActiveAttrib( GLuint program,
        GLuint index,
        GLsizei bufSize,
        GLsizei *length,
        GLint *size,
        GLenum *type,
        GLchar *name);
```

donde *index* es el índice del atributo por el que se pregunta, *bufsize* es el número máximo de caracteres que se va a copiar del nombre, *length* es la longitud del nombre del atributo, *size* su tamaño, *type* la descripción del tipo de datos y *name* el nombre del atributo.

- Vertex Array Objects (VAO)
  - También se puede saber cuantos atributos se encuentran activos y cual es la longitud máxima de sus nombres con

```
void glGetProgramiv(GLuint program,
                    GL_ACTIVE_ATRIBUTES,
                    const GLint* nAttribs);

void glGetProgramiv(GLuint program,
                    GL_ACTIVE_ATRIBUTE_MAX_LENGTH,
                    const GLint* maxLength);
```

donde *index* es el índice del atributo por el que se pregunta, *bufsize* es el número máximo de caracteres que se va a copiar del nombre, *length* es la longitud del nombre del atributo, *size* su tamaño, *type* la descripción del tipo de datos y *name* el nombre del atributo.

- Ejemplo de definición de los atributos de los vértices

```
// Crea el VBO del atributo VertexPosition
GLuint vertexPositionVBO;
glGenBuffers(1, &vertexPositionVBO);
glBindBuffer(GL_ARRAY_BUFFER, vertexPositionVBO);

// almacena los datos
int numVertex = …;
Glfloat *vertexPositionData = { … };
glBufferData(GL_ARRAY_BUFFER,
             3*numVertex*sizeof(GLfloat),
              vertexPositionData, GL_STATIC_DRAW );

…
```

- Ejemplo de definición de los atributos de los vértices (sigue)

```
…

// Crea el VBO del atributo VertexColor
GLuint vertexColorVBO;
glGenBuffers(1, &vertexColorVBO);
glBindBuffer(GL_ARRAY_BUFFER, vertexColorVBO);

// almacena los datos
Glfloat *vertexColorData = { … };
glBufferData(GL_ARRAY_BUFFER,
             3*numVertex*sizeof(GLfloat),
              vertexColorData, GL_STATIC_DRAW );

…
```

- Ejemplo de definición de los atributos de los vértices

```
// Crea el VAO y lo activa
GLuint vaoHandle;
glGenVertexArrays(1, &vaoHandle);
glBindVertexArray( vaoHandle );

// Activar los atributos
glEnableVertexArrayAttrib(0);  // VertexPosition
glEnableVertexArrayAttrib(1);  // VertexColor

// Indicar el origen de los atributos
glBindBuffer(GL_ARRAY_BUFFER, vertexPositionVBO);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
                      (GLubyte*) NULL);
glBindBuffer(GL_ARRAY_BUFFER, vertexColorVBO);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
                      (GLubyte*) NULL);
```

- Vertex Array Objects
  - Para lanzar un dibujo se utiliza el siguiente código

```
glBindVertexArray(vaoHandle);
glDrawArrays(GL_TRIANGLES, 0, numVertex);
```

  - Para utilizar arrays de índices se ejecutaría *glDrawElements()*.

- Variables uniformes
  - Describen valores que son comunes a todos los vértices (o a todos los fragmentos).
  - Para asignar los valores se utilizan los comandos *glUniform…()*. Existen numerosas versiones que se diferencian en el tipo de dato a asignar.
  - Por ejemplo, las siguientes definiciones se refieren a variables formadas por un valor en coma flotante (*float*) o por dos (*vec2*).

```
void glUniform1f( GLint location,
       GLfloat v0);

void glUniform2f( GLint location,
       GLfloat v0,
       GLfloat v1);
```

- Variables uniformes
  - Las funciones disponibles son las siguientes:

glUniform1f	glUniform2f	glUniform3f	glUniform4f
glUniform1i	glUniform2i	glUniform3i	glUniform4i
glUniform1ui	glUniform2ui	glUniform3ui	glUniform4ui
glUniform1fv	glUniform2fv	glUniform3fv	glUniform4fv
glUniform1iv	glUniform2iv	glUniform3iv	glUniform4iv
glUniform1uiv	glUniform2uiv	glUniform3uiv	glUniform4uiv
glUniformMatrix2fv		glUniformMatrix3fv	
glUniformMatrix4fv		glUniformMatrix2x3fv	
glUniformMatrix3x2fv		glUniformMatrix2x4fv	
glUniformMatrix4x2fv		glUniformMatrix3x4fv	
glUniformMatrix4x3fv			

- Variables uniformes
  - El campo *location* indica la posición de la variable a la que se quiere asignar el valor.
  - Las funciones (*glUniform*v*) y (*glUniformMatrix**) utilizan un segundo campo, *GLuint count*, que describe el número de elementos que contiene el puntero al valor. De esta forma se pueden asignar valores a *n* variables del mismo tipo (por ejemplo, 3 variables de tipo *vec3* o *mat4*).
  - Las funciones (*glUniformMatrix**) utilizan un tercer campo, *GLboolean transpose*, que indica si las matrices deben cargarse en posición normal o traspuestas.

- Variables uniformes
  - Para conocer la posición de una variable uniforme, utilizada en el campo *location*, se usa la función

```
GLint glGetUniformLocation( GLuint program,
        const GLchar *name);
```

donde *name* se refiere al nombre de la variable en el código del shader.

  - Para obtener información sobre una cierta variable uniforme se utiliza

```
void glGetActiveUniform( GLuint program,
        GLuint index,
        GLsizei bufSize,
        GLsizei *length,
        GLint *size,
        GLenum *type,
        GLchar *name);
```

- Variables uniformes
  - También se puede saber cuantas variables uniformes se encuentran activas y cual es la longitud máxima de sus nombres con

```
void glGetProgramiv(GLuint program,
                    GL_ACTIVE_UNIFORMS,
                    const GLint* nUniforms);

void glGetProgramiv(GLuint program,
                    GL_ACTIVE_UNIFORM_MAX_LENGTH,
                    const GLint* maxLength);
```

- Bloques de variables uniformes
  - En las aplicaciones gráficas es bastante corriente trabajar con varios shader-programs e ir intercambiándolos con *glUseProgram()*. En tal caso, cada programa tendrá sus propias posiciones para sus variables uniformes.
  - Cuando las variables uniformes son las mismas en diferentes programas (por ejemplo, las variables dedicadas a describir la luz o las dedicadas a las matrices model-view-projection) resulta más cómodo que las posiciones de las variables uniformes sean las mismas.
  - Para conseguir esto se trabaja con bloques de variables uniformes.
  - Un bloque no es más que una lista de variables agrupadas en la misma estructura. Por ejemplo:

```
uniform BlockSettings {
   vec4 InnerColor;
   vec4 OuterColor;
   float InnerRadius;
   float OuterRadius;
};
```

- Bloques de variables uniformes
  - Para obtener el índice de un bloque de variables uniformes se usa el comando

```
GLuint glGetUniformBlockIndex( GLuint program,
       const GLchar *uniformBlockName);
```

  - Para obtener información respecto a un bloque se utiliza

```
void glGetActiveUniformBlockiv( GLuint program,
       GLuint uniformBlockIndex,
       GLenum pname,
       GLint *params);
```

  - Por ejemplo, el parámetro GL_UNIFORM_BLOCK_DATA_SIZE contiene el tamaño total ocupado por el bloque.

- Bloques de variables uniformes
  - Una ventaja de utilizar bloques es que su contenido se puede almacenar en la memoria de la GPU utilizando *Uniform Buffer Objects* (UBO).
  - Los UBOs se utilizan como cualquier otro buffer (*glGenBuffer*, *glBindBuffer*, *glBufferData*), indicando el tipo GL_UNIFORM_BUFFER.
  - Para enlazar un UBO con un bloque se utiliza el comando

```
void glBindBufferBase(GL_UNIFORM_BUFFER,
                      blockIndex, uboID);
```

- Posición de las variables uniformes
  - Con respecto al contenido del buffer, es importante que cada variable del bloque esté colocada en la posición correcta.
  - Para conocer la posición de cada variable, en primer lugar hay que conocer el índice de cada variable en el programa

```
void glGetUniformIndices( GLuint program,
        GLsizei uniformCount,
        const GLchar **uniformNames,
        GLuint *uniformIndices);
```

donde *uniformCount* es el número de variables por el que se pregunta, *uniformNames* contiene sus nombres y *uniformIndices* devuelve los índices de las variables consultadas.

- Posición de las variables uniformes
  - A partir de los índices se puede preguntar por la posición de cada variable

```
void glGetActiveUniformsiv( GLuint program,
        GLsizei uniformCount,
        const GLuint *uniformIndices,
        GLenum pname,
        GLint *params);
```

donde *uniformCount* es el número de variables por el que se pregunta, *uniformIndices* contiene sus nombres, *pname* es el parámetro por el que se pregunta y *params* devuelve los valores de dicho parámetro. Para preguntar por la posición de cada variable se utiliza el parámetro GL_UNIFORM_OFFSET.

- Primitivas
  - Para lanzar un proceso de renderizado, asumiendo que el programa con los shaders está en uso y que los datos de los vértices y de las variables uniformes ya han sido cargados, se utilizan los comandos de dibujo *glDraw…()*.

```
void glDrawArrays( GLenum mode,
       GLint first,
       GLsizei count);

void glDrawElements(   GLenum mode,
       GLsizei count,
       GLenum type,
       const GLvoid * indices);
```

- Primitivas
  - El comando *glDrawArrays* asigna los vértices a las primitivas de manera consecutiva.
  - El comando *glDrawElements* asigna los vértices a partir de una lista de índices que toma del buffer GL_ELEMENTS_ARRAY_BUFFER activo.
  - Las versiones instanciadas de estos comandos ejecutan las primitivas un cierto número de veces (*primcount*).

```
void glDrawArraysInstanced( GLenum mode,
       GLint first,
       GLsizei count,
       GLsizei primcount);

void glDrawElementsInstanced( GLenum mode,
       GLsizei count,
       GLenum type,
       const void * indices,
       GLsizei primcount);
```

- Primitivas
  - El comando *glDrawElementsBaseVertex* es similar a *glDrawElements*, pero suma *basevertex* a los índices de vértices. Por ejemplo, si en el buffer de índices aparece { 0, 1, 2} y *basevertex* es 4, los vértices que utiliza en la primitiva serían { 4, 5, 6 }.

```
void glDrawElementsBaseVertex( GLenum mode,
       GLsizei count,
       GLenum type,
       GLvoid *indices,
       GLint basevertex);

void glDrawElementsInstancedBaseVertex( GLenum mode,
       GLsizei count,
       GLenum type,
       GLvoid *indices,
       GLsizei primcount,
       GLint basevertex)
```

- Primitivas
  - También existe el comando *glDrawRangeElements* (y las correspondientes versiones *Instanced* y *BaseVertex*) que añade los campos *start* y *end*. Estos campos indican un rango en el que se encuentran todos los índices a utilizar. Esta información puede ser utilizada por la GPU para optimizar el uso de memoria y ejecutar más rápido las primitivas (aunque no siempre es posible).

```
void glDrawRangeElements( GLenum mode,
        GLuint start,
        GLuint end,
        GLsizei count,
        GLenum type,
        const GLvoid * indices);
```

- Primitivas
  - Los diferentes tipos de primitivas soportadas (valores del campo *mode*) son los siguientes.
  - Estos comandos se envían a la etapa de ensamblado de primitivas, que los distribuye hacia el shader de teselado (GL_PATCHES) o hacia el shader de geometría.

GL_POINTS	GL_LINE_STRIP
GL_LINE_LOOP	GL_LINES
GL_LINE_STRIP_ADJACENCY	GL_LINES_ADJACENCY
GL_TRIANGLE_STRIP	GL_TRIANGLE_FAN
GL_TRIANGLES	GL_TRIANGLE_STRIP_ADJACENCY
GL_TRIANGLES_ADJACENCY	GL_PATCHES

- Texturas
  - Los comandos *glGenTextures()*, *glDeleteTextures()*, *glIsTexture()* y *glBindTexture()* permiten crear, destruir, verificar y activar texturas.
  - Los comandos *glTexImage..()*, *glCopyTexImage..()*, *glTexSubImage()* y *glCopyTexSubImage..()* permiten cargar las imágenes de textura.
  - Los comandos *glTexParameter..*() y *glTexEnv..*() permiten configurar la aplicación de texturas.
  - Las texturas las estudiaremos en el Tema 6.

- Configuración de etapas no programables
  - Con la aparición de etapas de renderizado programable, algunas de las opciones clásicas de configuración de OpenGL se han declarado obsoletas y no aparecen en la versión Core Profile del estándar. Por ejemplo, el efecto niebla (GL_FOG) o los patrones de interferencia (GL_LINE_STIPPLE).
  - Las opciones que admite el comando glEnable() son las siguientes

Opción	Funciones de configuración
GL_BLEND	glBlendFunc
GL_CLIP_DISTANCE i	
GL_COLOR_LOGIC_OP	glLogicOp.
GL_CULL_FACE	glCullFace
GL_DEBUG_OUTPUT	
GL_DEBUG_OUTPUT_SYNCHRONOUS	glDebugMessageCallback.

- Configuración de etapas no programables

Opción	Funciones de configuración
GL_DEPTH_CLAMP	glDepthRange.
GL_DEPTH_TEST	glDepthFunc, glDepthRange.
GL_DITHER	
GL_FRAMEBUFFER_SRGB	
GL_LINE_SMOOTH	glLineWidth.
GL_MULTISAMPLE	glSampleCoverage.
GL_POLYGON_OFFSET_FILL	glPolygonOffset.
GL_POLYGON_OFFSET_LINE	glPolygonOffset.
GL_POLYGON_OFFSET_POINT	glPolygonOffset.
GL_POLYGON_SMOOTH	
GL_PRIMITIVE_RESTART	glPrimitiveRestartIndex.

- Configuración de etapas no programables

Opción	Funciones de configuración
GL_PRIMITIVE_RESTART_FIXED_INDEX	
GL_RASTERIZER_DISCARD	
GL_SAMPLE_ALPHA_TO_COVERAGE	
GL_SAMPLE_ALPHA_TO_ONE	
GL_SAMPLE_COVERAGE	glSampleCoverage.
GL_SAMPLE_SHADING	glMinSampleShading.
GL_SAMPLE_MASK	glSampleMaski.
GL_SCISSOR_TEST	glScissor.
GL_STENCIL_TEST	glStencilFunc, glStencilOp.
GL_TEXTURE_CUBE_MAP_SEAMLESS	
GL_PROGRAM_POINT_SIZE	

- OpenGL Mathematics (glm)
  - Es una biblioteca escrita en C++ basada en la especificación de GLSL, que implementa los tipos de datos y las funciones de GLSL para su uso en los programas de OpenGL.
  - Por ejemplo, define los tipos glm::mat3 o glm::vec4 para tratar con matrices o vectores.
  - La biblioteca contiene además muchas funciones para tratar con vectores y matrices, algunas de ellas similares a las utilizadas en las versiones anteriores de OpenGL. Por ejemplo, glm::frustum() o glm::rotate().
  - Para usar la biblioteca tan solo es necesario copiarla en el directorio include del compilador.

- Una subrutina es un mecanismo que permite seleccionar una función (entre un conjunto de funciones posibles) basado en el valor de una variable uniforme.
- Las subrutinas proporcionan una forma de seleccionar implementaciones alternativas en tiempo de ejecución sin necesidad de cambiar el programa GLSL ni utilizar instrucciones if.
- En muchos sentidos es similar a los punteros a funciones de C.
- La variable uniforme sirve como puntero a la función y se utiliza para invocarla.
- Las funciones que implementan una subrutina utilizan distintos nombres, pero debe tener el mismo número y tipo de parámetros y el mismo tipo de retorno.

- Para declarar una subrutina en un shader se utiliza la instrucción ***subroutine*** seguida de la declaración del tipo de función. Por ejemplo,

```
subroutine vec3 shadeModelType( vec4 position, vec3 normal);
```

- Esta declaración define el nombre de la subrutina, los argumentos de la función y el tipo de dato devuelto.
- Para declarar una variable uniforme que contenga un puntero a una subrutina de utiliza la instrucción ***subroutine uniform***.

```
subroutine uniform shadeModelType shadeModel;
```

- Para declarar una función que desarrolla una subrutina se utiliza el modificador ***subroutine(nombre)*** seguida de la declaración de la función.

```
subroutine( shadeModelType )
vec3 phongModel( vec4 position, vec3 norm )
{
    …
}

subroutine( shadeModelType )
vec3 diffuseOnly( vec4 position, vec3 norm )
{
    …
}
```

- Una misma función puede desarrollar varias subrutinas al mismo tiempo. En ese caso aparece una lista de nombres entre paréntesis.

- Para usar una subrutina se utiliza la variable uniforme como si fuera una función.

```
void main()
{
    …
    LightIntensity = shadeModel(eyePosition,eyeNorm);
    …
}
```

- La selección de la subrutina en el programa OpenGL se realiza asignando el valor a la variable uniforme. Para ello en primer lugar hay que obtener la posición de las funciones que desarrollan la subrutina:

```
GLuint phongModelIndex = glGetSubroutineIndex(
          programHandle,
          GL_VERTEX_SHADER,
          "phongModel" );

GLuint diffuseOnlyIndex = glGetSubroutineIndex(
          programHandle,
          GL_VERTEX_SHADER,
          "diffuseOnly");
```

- Por último, una vez conocida la posición de cada función (el "puntero" a la función), se asigna el valor deseado a la variable uniforme.

```
glUniformSubroutinesuiv( GL_VERTEX_SHADER,
                         1,
                        &phongModelIndex);
```

- Esta función asigna una lista de subrutinas. El segundo argumento indica el número de subrutinas a asignar. El tercer argumento es un array a las posiciones de las funciones. En nuestro caso solo se asigna una función así que se puede utilizar la dirección de la variable *phongModelIndex* como si fuera el comienzo del array.

- El orden en el que se asignan las subrutinas a las variables uniformes es el mismo en el que se han definido en el programa. El array debe contener la asignación de funciones en el orden adecuado.
- Cuando el programa incluye varias variables uniformes de tipo subrutina, se puede consultar la posición de cada una con el comando *glGetSubroutineUniformLocation()* y de esa forma ordenar el array de manera correcta.

- El proceso de renderizado genera el contenido de la imagen en una estructura que conocemos como ColorBuffer.
- Si activamos la opción de test de profundidad, además de información sobre el color de cada pixel también se almacena información sobre la coordenada *z* (la profundidad) asociada a cada pixel. Esta información se almacena en una estructura paralela conocidad como DepthBuffer.
- Existe otro buffer, denominado StencilBuffer que se utiliza como buffer auxiliar para almacenar otra información configurable. Este buffer se utiliza para programar algunos efectos (como reflejos, por ejemplo).

- OpenGL permite dirigir la salida del proceso de renderizado hacia otras estructuras que no estén vinculadas directamente con el contenido a mostrar en la pantalla. De esta forma, se puede generar una imagen y almacenarla en lugar de mostrarla. Esta imagen almacenada se puede utilizar posteriormente (por ejemplo, como textura).
- Se denomina Frame Buffer Object (FBO) a la estructura de datos que describe la salida del proceso de renderizado. Este FBO contiene enlaces al ColorBuffer, DepthBuffer y StencilBuffer utilizados en el renderizado.
- OpenGL crea un Frame Buffer Object por defecto que asocia a la pantalla. Este FBO tiene preasignado el identificador 0.

- Para dirigir el proceso de renderizado a memoria (en vez de a la pantalla) hay que crear un nuevo FBO, crear los buffers necesarios, vincularlos al FBO y activar el FBO. De esta forma el renderizado se dirigirá a las estructuras enlazadas al FBO.

```
GLuint fboHandle; // The handle to the FBO

// Generate and bind the framebuffer
glGenFramebuffers(1, &fboHandle);
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);
```

- El identificador de un FBO es un valor de tipo *GLuint*. Para crear un nuevo FBO se utiliza el comando *glGenFramebuffers()*. Para activarlo se utiliza el comando *glBindFramebuffer()*.

- Para crear el ColorBuffer y asociarlo al FBO se utiliza una textura 2D.

```
// Create the texture object
GLuint renderTex;
glGenTextures(1, &renderTex);
glActiveTexture(GL_TEXTURE0); // Use texture unit 0
glBindTexture(GL_TEXTURE_2D, renderTex);

glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,512,512,
             0,GL_RGBA,GL_UNSIGNED_BYTE,NULL);

glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Bind the texture to the FBO
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D, renderTex, 0);
```

- En este caso el contenido de la textura no se lee desde un fichero, sino que se generará en el proceso de renderizado. El comando *glTexImage2D()* se utiliza para configurar las propiedades de la textura (tamaño y formato de almacenamiento) pero el último argumento (puntero al contenido) se deja a nulo.
- El comando *glFramebufferTexture2D()* permite enlazar el ColorBuffer del FBO al buffer de textura que hemos definido.

- Para crear el DepthBuffer y asociarlo al FBO se utiliza un RenderBuffer.

```
// Create the depth buffer
GLuint depthBuf;
glGenRenderbuffers(1, &depthBuf);
glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
glRenderbufferStorage(GL_RENDERBUFFER,
                      GL_DEPTH_COMPONENT,
                      512, 512);

// Bind the depth buffer to the FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                          GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, depthBuf);
```

- El tamaño del DepthBuffer debe coincidir con el del ColorBuffer.

- También se puede crear el DepthBuffer por medio de una textura.

```
// Create the texture object
GLuint depthTex;
glGenTextures(1, &depthTex);
glActiveTexture(GL_TEXTURE0); // Use texture unit 0
glBindTexture(GL_TEXTURE_2D, depthTex);

glTexImage2D(GL_TEXTURE_2D,0,
             GL_DEPTH_COMPONENT,512,512, 0,
           GL_DEPTH_COMPONENT,GL_UNSIGNED_BYTE,NULL);

glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MAG_FILTER, GL_NEAREST);
// Bind the texture to the FBO
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, depthTex, 0);
```

- En este caso el formato de la textura no es GL_RGBA sino GL_DEPTH_COMPONENT.
- El comando *glFramebufferTexture2D()* utiliza el parámetro GL_DEPTH_ATTACHMENT para enlazar el DepthBuffer del FBO al buffer de textura que hemos definido.
- Para terminar, hay que indicar cual va a ser el buffer de dibujo.

```
// Set the target for the fragment shader outputs
GLenum drawBufs[] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, drawBufs);

// Revert to default framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- El último comando vuelve a activar el FBO por defecto, dirigiendo de nuevo la imagen a la pantalla.

- Una vez configurado el FBO alternativo, para activarlo se utiliza el siguiente código.

```
// Bind to texture's FBO
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);

// Viewport for the texture
glViewport(0,0,512,512);

// Render the scene to the FBO
…
```

- Para reactivar el modo pantalla sería

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0,0,width,height);
// Render the scene to the window
…
```