Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Lecturas en Ciencias de la Computación
ISSN 1316-6239

Despliegue Básico en OpenGL Moderno

Esmitt Ramírez

ND 2014-01

Centro de Computación Gráfica de la UCV
CCG-UCV
Caracas, enero 2014

ND 2014-01
Lecturas en Ciencias de la Computación
ISSN 1316-6239

Despliegue Básico en OpenGL Moderno

Esmitt Ramírez J.
Enero 2014

Resumen

OpenGL (**Op**en **G**raphics **L**ibrary) es una librería gráfica ampliamente usada a nivel mundial para el despliegue de gráficos 2D/3D. Esta librería presenta una API multiplataforma y es soportada por diversos lenguajes de programación. Desde su aparición en 1992, es empleada en la creación de programas CAD, visualización científica, despliegue de simulaciones, juegos de video, entre otros.

La documentación asociada a OpenGL es muy variada y extendida. A lo largo de los años se han creado diversas funcionalidades que tratan de explotar al máximo las capacidades del hardware gráfico. Recientemente, hubó cambios considerables en la estructura del funcionamiento de los programas de OpenGL. Estos cambios se deben principalmente a la evolución de las GPUs (*Graphics Processing Unit*) modernas, haciendo ciertas funciones obsoletas y otras necesarias. De esta forma, las funciones actuales de OpenGL aprovechan las capacidades de almacenamiento en memoria de video y de los núcleos de procesamiento de la GPU. En Computación Gráfica, a esta reciente forma de operar con OpenGL se le denomina OpenGL moderno.

En este documento, se realiza una breve introducción a los requerimientos necesarios para desarrollar programas basados en OpenGL moderno (con el uso de shaders). Al mismo tiempo, se presenta la evolución del pipeline gráfico en OpenGL y cada una de sus etapas.

Tabla de Contenido

1 Introducción **3**

2 Evolución del Pipeline Gráfico **3**

3 Pipeline Gráfico **4**
- 3.1 Vertex Shading . . . 4
- 3.2 Tessellation Shading . . . 4
- 3.3 Geometry Shading . . . 5
- 3.4 Ensamblaje de Primitivas . . . 5
- 3.5 Clipping . . . 5
- 3.6 Rasterización . . . 5
- 3.7 Fragment Shading . . . 5
- 3.8 Operaciones por Fragmento . . . 5

4 Configuración de OpenGL **6**

5 Ejemplo de despliegue **7**

6 Conclusiones **10**

1 Introducción

OpenGL (Open Graphics Library) es una API multiplataforma para el despliegue de primitivas gráficas en 2D y 3D. Con OpenGL es posible crear aplicaciones interactivas en computación gráfica para el despliegue de imágenes a color en alta calidad que representa a objetos geométricos, primitivas, escenas 3D, datos multidimensionales, imágenes, entre otros. La API de OpenGL es independiente del sistema operativo y del sistema de ventanas. Así, parte del código se ejecuta independiente de la plataforma de ejecución y otra es controlada por el sistema de ventanas de la plataforma donde se ejecuta (e.g. X Windows, API Win32, etc.).

La versión 1.0 de OpenGL fue liberada en enero de 1992 y desde entonces es ampliamente utilizada en la construcción de aplicaciones CAD, realidad virtual, visualización científica, simulaciones, y videojuegos. A lo largo de los años, la versión original ha sufrido numerosos cambios a la fecha. Este documento se enfoca en las versiones modernas de OpenGL (desde la versión 3.1) la cual está basada en shaders (conocidda como *shader-based*).

La documentación asociada a las primeras versiones de OpenGL u otros pipelines gráficos basados en rasterización que incluyen procesamiento sobre funciones fijas (*fixed-based*), no son cubiertas en este documento. En el desarrollo de aplicaciones bajo OpenGL moderno la funcionalidad fija está en estado de *deprecated*[1]. Así, en las siguientes secciones solo se mencionará el uso de los shaders para el desarrollo de una aplicación simple así como las estructuras de datos asociadas a ella.

2 Evolución del Pipeline Gráfico

La versión inicial de OpenGL (versión 1.0) incluía un pipeline de funcionalidad fija. Esto significa que todas las funciones soportadas por OpenGL están bien definidas y una aplicación solo puede modificar los parámetros de entrada a dichas funciones como los valores de posición, color, entre otros. En un pipeline de funcionalidad fija, el orden de ejecución de las etapas operacionales es fijo. Con ello, no es posible reordenar las etapas para que se ejecuten en un orden distinto. Esta forma de operar fue la base de diversas versiones de OpenGL durante muchos años, y aún está disponible para ser empleado así. Sin embargo, el hardware gráfico moderno introdujo nuevas funcionalidades para crear un cambio de paradigma y aprovechar al máximo las capacidades de las GPUs actuales (Graphics Processing Units), pero manteniendo compatibilidad hacia atrás con las versiones previas. En la Figura 1 se puede observar la evolución años/versiones de OpenGL hasta la fecha.

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

Figura 1: *Evolución de las versiones de OpenGL a través del tiempo. Actualizado a enero 2014.*

En Septiembre de 2004, fue liberada la versión 2.0 de OpenGL con compatibilidad a versiones anteriores. En dicha versión fue añadido de forma oficial la funcionalidad de *programmable shaders* en el pipeline gráfico. Un *shader* es un pequeño programa responsable de la implementación de características inherentes a la aplicación que se encuentra dentro del pipeline gráfico y es ejecutado directamente por la GPU. En esta versión de OpenGL fueron añadidas dos etapas programables:

- vertex shading, que permite a la aplicación manipular toda la geometría 3D provista por un programa en OpenGL .
- fragment shading, que provee la capacidad a la aplicación de sombrear píxeles, es decir, determinar el color final de un píxel.

Hasta la versión 3.0 de OpenGL, fueron añadidas diversas funcionalidades ofreciendo siempre la compatibilidad hacia atrás. Esta versión introduce el mecanismo de remoción de características de OpenGL, denominado como *deprecation model*. De esta forma, el comité encargado de las normas de OpenGL, el OpenGL Architecture Board (ARB) perteneciente a Khronos Group, advierte cuáles funcionalidades serán removidas de OpenGL para futuras versiones. Esta remoción se debe principalmente a la eficiencia de las funciones en OpenGL dado que las extensiones eran creadas sobre otras funciones que no explotaban las capacidades de las GPUs modernas. Así, la construcción de un pipeline más simple y optimizado sería posible.

Al mismo tiempo, se introduce en OpenGL una estructura de datos llamada *context* la cual se emplea para almacenar shaders y otros datos internos de OpenGL. Básicamente se existen dos tipos de contexto: full y forward-compatible. Un contexto del tipo full incluye todas las características de la actual versión de OpenGL, incluyendo las funcionalidades no-aprobadas (*deprecated*). Si se define el contexto como forward-compatible, se habilita solo las características que estarán disponibles en la próxima versión de OpenGL. Esto se hace para ayudar a los desarrolladores a asegurar que sus aplicaciones se ejecutarán de forma correcta en las futuras versiones de OpenGL.

La versión 3.1 de OpenGL introduce las funcionalidades no-aprobadas y rompe con la compatibilidad hacia atrás. Esto hace que también se haga obsoleto el uso del pipeline gráfico del tipo *fixed-based*. En esta versión, todos los datos de la aplicación son *GPU-resident* en estructuras llamadas *buffer objects* ayudando a reducir ciertas limitaciones de varias arquitecturas en relación a la ejecución de programas en la GPU.

[1] En software, el estado *deprecated* se refiere a obsoleto o desfasada al referirse a una API, librería, función ó metodología empleada.

La posterior versión, OpenGL 3.2, añade una nueva etapa de shader en el pipeline gráfico llamada geometry shading que permite la modificación y generación de geometría en OpenGL. Esta nueva etapa se ubica después de la etapa de vertex shading. Además, en esta versión se introduce el concepto de context profiles (*core* y *compatible*) para lograr una selección más amplia de los contextos en OpenGL.

En marzo de 2010, la versión 4.0 añade dos etapas más al pipeline programable: tessellation-control y tessellation-evaluation. Ambas trabajan de forma conjunta y permiten el teselado dinámico en la GPU. La versión 4.3 agrega una forma de aumentar el paralelismo en la GPU dentro de contextos gráficos con shaders llamados *compute shaders*. La última versión de OpenGL, la versión 4.4 (Julio 2013) incluye muchas funcionalidades añadidas sobre el pipeline gráfico de la versión 4.0 pero no nuevas etapas de shading.

3 Pipeline Gráfico

El pipeline gráfico o *rendering pipeline* es una secuencia de etapas de procesamiento que dada una entrada basada en vértices y sus atributos, generan una imagen final para su despliegue. En la figura 2 se muestra un pipeline gráfico simplificado asociado a la versión más reciente de OpenGL. En la figura se puede observar que existen etapas que son provistas por la aplicación, requeridas u obligatorias, opcionales y fijas (y algunas configurables).

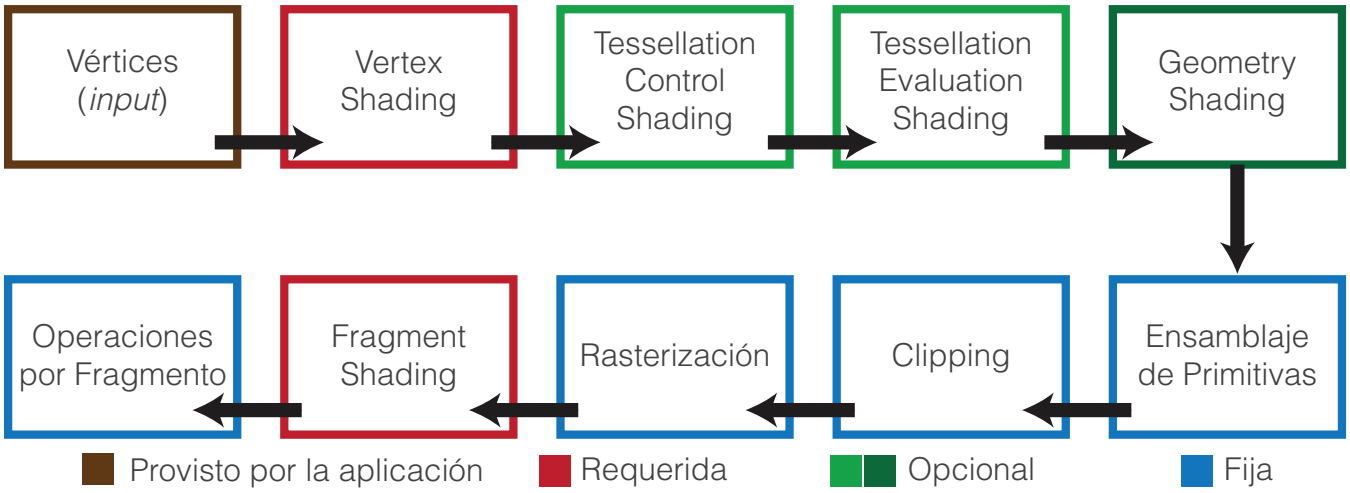

Figura 2: *Una representación simplificada del pipeline gráfico de OpenGL mostrando cada una de sus etapas.*

El pipeline se inicia cuando la aplicación (programa escrito en OpenGL en algún lenguaje de programación) envía datos geométricos como vértices y primitivas a ser procesadas. Estos datos pasan primero por las etapas de vertex shading, tessellation shading (que emplea dos shaders) y geometry shading antes de llegar a la etapa de rasterizado. La etapa de rasterizado generará fragmentos para cualquier primitiva que se encuentre dentro de la región de clipping, y posteriormente se ejecutará un programa de shader (fragment shader) por cada fragmento generado. A continuación se describe brevemente cada una de las etapas involucradas.

3.1 Vertex Shading

Una vez el pipeline gráfico recibe desde la aplicación el conjunto vértices y su conectividad, los vértices son procesados uno a uno. Los vértices que se procesan son aquellos que son invocados por alguna función de despliegue en la aplicación, e.g. la función `glDrawArray`.

El vertex shader puede ser muy simple, ejecutando al menos la copia de los vértices desde esta etapa a la siguiente. A este tipo de shaders se le conoce como *pass-through shaders*. La complejidad de un vertex shader va a depender de la complejidad de las etapas previas a la rasterización, pudiendo calcular operaciones complejas basadas en la posición de los vértices aplicando diversas transformaciones.

Típicamente, una aplicación puede contener múltiples vertex shaders, pero solo uno estará activo en un instante de tiempo.

3.2 Tessellation Shading

Luego de procesar cada vértice en la etapa de vertex shading, la etapa de tessellation shading continúa el proceso del pipeline gráfico. En esta etapa se emplean primitivas llamadas *patchs* que describen la forma de un objeto y permiten el manejo de estructuras simples que serán teseladas. Una teselación se define como un patrón de figuras que cubre una superficie plana tal que no queden orificios entre estas y no se superpongan. Así, la etapa de tessellation shading incrementará el número de primitivas geométricas para proveer una mayor calidad/resolución en los objetos 3D. En la Figura 3 se observa un ejemplo donde una mitad de un modelo geométrico de baja resolución está teselada, y la otra no.

Esta etapa añade dos etapas de shading al pipeline para generar una malla de primitivas geométricas. Para ello, se debe especificar un parche (*patch*) que consiste en una lista ordenada de los vértices. Primero se ejecuta el tessellation control shader (TCS) que especifica la forma

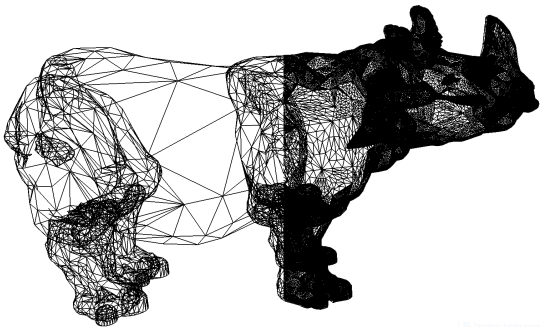

Figura 3: *Ejemplo de un proceso de teselación sobre un objeto 3D. El lado derecho del modelo del rinoceronte tiene aplicado una teselación en sus triángulos, y el lado izquierdo no.*

en que será generada la nueva geometría. Esta etapa es opcional, ya que puede emplear los valores por defecto de la teselación. Después, existe una etapa llamada tessellation primitive generator (TPG) que con el parche como entrada la subdivide basado en los valores del TCS, generando coordenadas de teselación. Es importante destacar que esta etapa no es programable. Luego, se ejecuta el tessellation evaluation shader (TES), donde se crean las nuevas posiciones de la geometría creada basado en las coordenadas de teselación. Finalmente, los datos se envían a ser rasterizados.

3.3 Geometry Shading

Esta etapa se ubica antes de la etapa de rasterización y del fragment shading. Esta recibe como entrada una primitiva completa (i.e. vértices y su información de conectividad) como una colección de vértices, representados como un arreglo unidimensional. Generalmente, esta entrada es provista por el vertex shader. sin embargo, cuando existen los shaders de teselación, la entrada proviene de dicha etapa.

Una característica importante de esta etapa, es su capacidad de cambiar el tipo (a cuatro posibles tipos) y el número de primitivas que son pasadas a través del pipeline de OpenGL.

3.4 Ensamblaje de Primitivas

Las etapas previas de shading operan sobre los vértices dentro del pipeline gráfico, que posteriormente deben ser transformadas en primitivas de acuerdo a la información de interconectividad. La etapa de ensamblaje de primitivas se encarga de organizar estos vértices en las primitivas adecuadas y las prepara para las etapa de clipping y rasterización.

3.5 Clipping

Cuando los vértices se encuentran fuera de la ventana de despliegue final (i.e. viewport) y la primitiva asociada a este vértice es modificada, se descartan solo los píxeles que esten fuera de la ventana. Esta operación se denomina recorte de primitivas o clipping, y es controlado de forma automática por OpenGL.

3.6 Rasterización

Después de la etapa de clipping, las primitivas que son actualizadas y se encuentren dentro del viewport son enviadas al rasterizador para la generación de fragmentos. Un fragmento se puede definir como una estructura que contiene diversos atributos (e.g. posición, color, profundidad, entre otros) de un candidato a píxel final, ver Figura 4. Un fragmento puede ser descartado y nunca ser actualizado en su posición de píxel correspondiente. El procesamiento de los píxeles sucede en las próximas dos etapas: fragment shading y operaciones por fragmento.

3.7 Fragment Shading

El fragment shading es la última etapa donde se tiene control sobre el pipeline gráfico. El fragment shader determinará el color final de un fragmento (que puede ser modificado luego en la siguiente etapa) así como su valor de profundidad. En esta etapa es posible tener acceso a texturas que permiten modificar el color de un fragmento. Además, se permite descartar un fragmento de acuerdo a cierto criterio provisto por la aplicación, a este proceso se le conoce como *fragment discard*.

Los shaders que operan sobre vértices (vertex, tessellation y geometry) determinan en qué posición de la pantalla estará ubicado una primitiva. Un fragment shader emplea esta información para determinar que color tendrá el fragmento.

3.8 Operaciones por Fragmento

El último paso para cada fragmento se ejecuta en esta etapa donde se determina su visibilidad empleando la prueba de profundidad (*depth testing* ó *z-buffering*) y la prueba de plantilla (*stencil testing*).

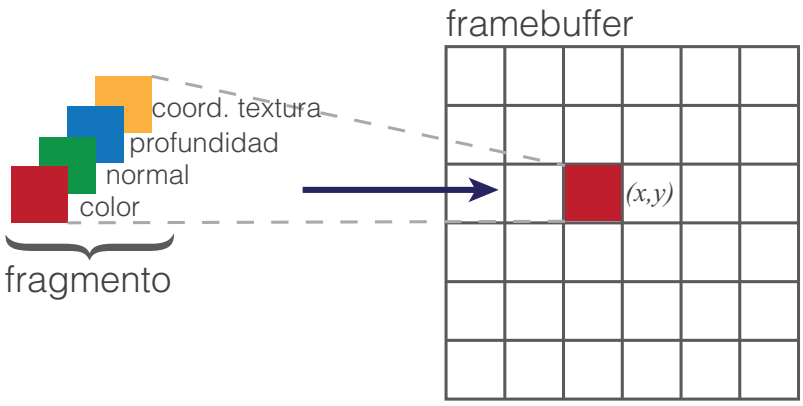

Figura 4: *Representación de un fragmento.*

De esta manera, si un fragmento llega con éxito a esta etapa pasando por las etapas anteriores, su valor puede ser colocado directamente en el framebuffer actualizando el color ó valor de profundidad de un píxel. Si está habilitada la mezcla (*blending*) entonces su color puede mezclarse con el color actual del píxel para generar un nuevo color que será escrito en el framebuffer.

Como se observo en la Figura **??**, existe un camino para obtener el color final de un píxel. Los datos que se ejecutarán en el pipeline provienen de un archivo con información geométrica o de otro tipo, directamente de las estructuras de datos de la aplicación o de forma procedimental.

Cabe destacar que para el render de una primitiva no se requieren de todas las etapas de shading. De hecho, solo son necesarias al menos las etapas de vertex y fragment shading. Las etapas de tessellation y geometry son opcionales. Al mismo tiempo, para el despliegue efectivo de las primitivas, éstas deben almacenarse en estructuras de datos llamadas *buffer objects*. Un buffer object es una estructura que almacena ”pedazos” (*chunks*) de memoria que es administrada por OpenGL directamente. Así, la aplicación debe llenar de datos los *buffer objects* (para más información, referirse a la wiki oficial de OpenGL `http://www.opengl.org/wiki/Buffer_Object`).

Actualmente en OpenGL, sin el uso de shaders para procesar las primitivas geométricas solamente se puede ”limpiar” la ventana de despliegue. La importancia del uso de shaders es vital para el desarrollo de un programa gráfico en OpenGL. El lenguaje definido para ello se denomina GLSL (OpenGL Shading Language), el cual ha evolucionado desde la versión 2.0 con cada versión de OpenGL. Básicamente, se puede definir GLSL como un lenguaje especialmente diseñado para el despliegue de gráficos, siendo muy similiar al lenguaje C, con una pequeña mezla de C++. A continuación se presenta la configuración necesaria en OpenGL para emplear shaders y *buffer objects*.

4 Configuración de OpenGL

Un programa escrito en OpenGL moderno, esencialmente realiza los siguientes pasos:

1. Cargar y crear los programas de shader en GLSL desde un archivo o directamente escrito en la aplicación
2. Asignar los datos de los objetos almacenados en estructuras de datos (e.g. memoria RAM) a la memoria de la GPU. Para realizar este proceso se crean *buffer objects* y se cargan los datos en dichos búferes.
3. Indicar a OpenGL la forma de interpretar los datos que se encuentran en *buffer objects* y de qué forma se asociará con las variables presentes en los shaders. Esta conexión se denomina *shader plumbing*.
4. Desplegar los objetos.

En este punto, se asume que se cuenta con el sistema de ventanas donde OpenGL se ejecutará. Por ejemplo, empleando una librería de binding que comunique a OpenGL con el sistema nativo de ventanas de un sistema operativo (por ejemplo freeglut `http://freeglut.sourceforge.net`, GLFW `http://www.glfw.org`, o cualquier otro). Igualmente, se asume el uso de GLEW `http://glew.sourceforge.net` para el manejo de ciertas funciones de OpenGL de forma más sencilla.

En OpenGL, y en muchas otras librerías gráficas, los objetos de una escena se componen por primitivas geométricas basando en vértices. Un vértice en OpenGL moderno es una colección de valores asociados con una ubicación espacial en $\mathbf{R}^3$. Esta colección de datos puede incluir color, coordenadas de textura, información para el cálculo de reflexión o refracción, entre otros. Una ubicación espacial puede ser definida en 2, 3 ó 4 dimensiones pero almacenadas empleando coordenadas homogéneas en 4 dimensiones.

El pipeline gráfico está diseñado en una estructura cliente-servidor. El lado del cliente es el código que reside en memoria de la CPU y es ejecutado con el programa de aplicación o empleando el driver (de una tarjeta gráfica no dedicada) en la memoria principal del sistema. Este driver, ensamblará los comandos de rendering y los datos que enviará al servidor OpenGL para su ejecución (e.g. hardware y memoria de la tarjeta gráfica). Tanto el cliente como el servidor funcionan de forma asíncrona, siendo piezas independientes de software o hardware, o ambas. Para alcanzar un alto rendimiento, es ideal emplear ambos lados tanto como sea posible. El cliente está continuamente ensamblando bloques de datos y comandos en buffers que son enviados al servidor para su ejecución. Así, el servidor ejecuta estos buffers, mientras que al mismo tiempo el cliente se prepara para enviar el próximo bloque de datos o información de rendering.

Los vértices deben ser organizados en objetos del lado servidor llamados *vertex buffer objects* (conocidos como VBOs), que contienen toda la información asociada a los vértices para todas las primitivas que se quieran desplegar. Los VBOs pueden almacenar la información de un vértice en diversas maneras como un arreglo de estructuras (AoS) o estructuras de arreglos (SoA). Los datos en un VBO tienen que ser contigua en memoria, pero no necesariamente "ajustada", es decir, los datos de un mismo tipo pueden estar separados por un número de bytes constante entre datos del mismo tipo. Por ejemplo, un VBO puede almacenar una posición de vértice (x, y, z), su color (r, g, b), y su vector normal (n_x, n_y, n_z) de todo un objeto 3D quedando como $xyzrgbn_xn_yn_zxyzrgbn_xn_yn_z...$ en memoria del lado servidor.

Dado que numerosos VBOs pueden estar asociados con un objeto 3D, y una escena puede contener diversos objetos, existe una estructura que permite simplificar el manejo de éstos. Los *vertex array objects*, llamados VAO, que son estructuras que manejan una colección de VBOs.

Por otro lado, la forma de componer los vértices y sus atributos en objetos 3D es a través de las primitivas que OpenGL puede desplegar. Estas primitivas son solo de 3 tipos: puntos, líneas y triángulos. A continuación se presenta un ejemplo de programa para el despliegue de un cubo empleando OpenGL basado en triángulos.

5 Ejemplo de despliegue

En está sección se describe un programa de ejemplo para el despliegue de un simple cubo con diferentes colores en cada uno de sus vértices. Por simplicidad, el ejemplo solo está enfocado en las técnicas empleadas en OpenGL más no en su óptimo rendimiento.

El cubo a desplegar, ver Figura 5, consta de 6 caras cuadradas donde cada una de ellas está formada por 2 triángulos. Así, se tiene un total de 12 triángulos para el cubo y como cada triángulo está formado por 3 vértices, se define la variable `iNVertices = 36` para almacenar el número de vértices totales (con repetición). Es importante destacar que una definición óptima sería emplear solamente 8 vértices, ya que triángulos adyancentes comparten vértices.

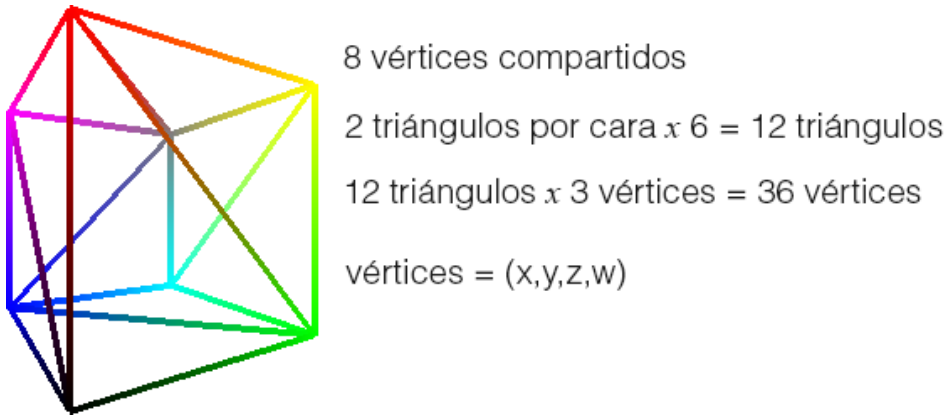

Figura 5: *Cubo de ejemplo con su especificación geométrica.*

Para simplificar la comunicación del programa con el shader en GLSL, se emplea una librería llamada OpenGL Mathematics (`http://glm.g-truc.net/`) que consta de una serie de definiciones de tipo y funciones que tienen la misma semántica que los tipos de dato en GLSL. Por ello, la estructura de almacenamiento de los vértices y de sus colores, se realizará con el uso de esta librería. Al mismo tiempo, las acciones para crear, asociar, asignar atributos/uniformes y habilitar/deshabilitar de un programa en GLSL se realizarán empleando una clase llamada `CGLSLProgram` que encapsula dichas funciones. Para este documento se asume el conocimiento de la sintaxis de GLSL básica, así como su uso de shaders en un programa de OpenGL. En el segmento de Código 1 se presentan las variables a emplear en este ejemplo.

Código 1: *Variables definidas para el despliegue del cubo.*

```
const int iNVertices = 36;
glm::vec4 vec_points[iNVertices];
glm::vec4 vec_colors[iNVertices];
glm::mat4x4 mProjMatrix, mModelViewMatrix;
CGLSLProgram m_program;
GLuint m_iIndexVAO;
```

Para enviarle la información de los datos del cubo a OpenGL, se requiere colocar los datos en estructuras que serán pasadas a los VBOs correspondientes. Estos datos pueden ser leídos desde un archivo o creadas dentro del código. Para cada vértice, se emplean 2 atributos (posición y color) que son definidos como `vec4`, es decir, un arreglo de 4 posiciones (i.e. en coordenadas homogéneas y en el espacio de color RGBA respectivamente). De esta forma, los arreglos `vec_points` y `vec_colors` contendrán está información para formar el cubo como se muestra en la Figura 5.

Una vez con los datos de posición y color en los arreglos correspondientes, se procede a la carga de los shaders. Para este ejemplo, solo se empleará el uso de un programa de vértices y uno de fragmentos. En el Código 2 se muestra la secuencia requerida para dicho proceso.

Código 2: *Carga e inicialización del vertex y fragment shader.*

```
m_program.loadShader("basic.vert", CGLSLProgram::VERTEX);
m_program.loadShader("basic.frag", CGLSLProgram::FRAGMENT);
m_program.create_link();
m_program.enable();
  m_program.addAttribute("vVertex");
  m_program.addAttribute("vColor");
  m_program.addUniform("mProjection");
  m_program.addUniform("mModelView");
m_program.disable();
```

Primeramente, se adquiere el vertex shader desde un archivo llamado `basic.vert`, y el fragment shader desde `basic.frag`. Note que dicha carga se puede realizar a través del uso de una variable o literal del tipo string que contenga el código en GLSL. Luego se crea y enlaza el programa en GLSL con el código en OpenGL, añadiendo variables del tipo atributo (*attribute*) y uniforme (*uniform*). Las funciones `enable/disable` permiten habilitar/deshabilitar el uso de un programa de fragmento dentro del código. Para este ejemplo, el vertex shader es como se muestra en el Código 3, y el fragment shader como aparece en el Código 4.

Código 3: *Vertex shader.*

```
#version 400

uniform mat4 mModelView, mProjection;

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

out vec4 vVertexColor;

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

El vertex shader está formado por la definición de la versión de GLSL a emplear, para este caso la versión 4.0 identificada como 400. Por ejemplo, si se emplea la versión 3.30 de GLSL y de OpenGL, se debe identificar la directiva `#version 330` para que sea interpretada por el pre-procesador del shader. Existe una asociación entre la versión de OpenGL y GLSL desde la versión 3.3, antes de ello se manejaban independientes. Luego se definen las variables `uniform` en el shader, solo las matrices de transformación de coordenadas de objeto a coordenadas de clipping.

A continuación, se definen los atributos o variables asociadas a cada vértice dentro del vertex shader. En OpenGL moderno, los atributos se definen del tipo *in/out*, donde `in` representa cuando provienen de la aplicación o de una etapa anterior de shading, y `out` cuando van hacia los buffers de salida o a una etapa posterior. El calificador (*qualifier*) del tipo layout, indica donde se define una variable (i.e. índice empleado para su almacenamiento y acceso) y donde puede ser accedido por otra instancia. Este calificador puede poseer más información de los recursos a utilizar por un atributo. En este caso, los vértices se encuentran en el slot 0 y su color en el slot 1.

La última definición es de la variable `vVertexColor`, donde se indica que dicha variable será pasada a una etapa posterior (fragment shader), y almacenará el color del vértice que será interpolado para cada fragmento visible. El código dentro de la función `main`, transforma cada vértice al multiplicarlo por las matrices de transformación y copia el color proveniente de la aplicación.

Código 4: *Fragment Shader.*

```
#version 400

in vec4 vVertexColor;

layout(location = 0) out vec4 vFragColor;

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

El fragment shader es un programa sencillo donde se copia la información de color que luego de pasar la etapa de rasterización, ésta interpola los colores dentro de la primitiva, pasando cada valor como un color de salida de un fragmento. Nótese que la variable `vFragcolor` escribirá en el framebuffer el color resultante.

Posterior a la carga e inicialización de los programas de shader de forma correcta, se procede a la creación de los buffers de almacenamiento. Como se mencionó anteriormente, los VAOs encapsulan todos los datos contenidos en los VBOs permitiendo un rápido intercambio entre múltiples objetos. El proceso de inicialización de un VAO es similar a la inicialización de un VBO. Primero, se genera un VAO invocando la función `glGenVertexArrays()`. Luego, se indica que se utilizará un VAO en particular con la instrucción `glBindVertexArray()`. El Código 5 muestra la secuencia adecuada para la creación de un VAO y un VBO donde se almacenará la posición de los vértices y su color.

Código 5: *Inicialización y configuración de un VAO y un VBO para el despliegue.*

```
//VAO
glGenVertexArrays(1, &m_iIndexVAO);
glBindVertexArray(m_iIndexVAO);
//VBO - create and initialize a buffer object
GLuint iIdBuffer;
glGenBuffers(1, &iIdBuffer);
glBindBuffer(GL_ARRAY_BUFFER, iIdBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vec_points) + sizeof(vec_colors), NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vec_points), vec_points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(vec_points), sizeof(vec_colors), vec_colors);
```

En OpenGL, un *vertex buffer object* como todos los objetos que residen en memoria son creados de la misma forma, empleando el mismo conjunto de instrucciones. Así, es posible el patrón de invocaciones realizadas para la creación de un VBO es similar a otras operaciones en OpenGL. En el caso de un VBO, la secuencia para su creación/utilización es como sigue:

1. Se genera un nombre del buffer, asociado a un id (con el uso de `glGenBuffers()`).
2. Seleccionar dicho buffer para lectura o escritura de valores, empleando la instrucción `glBindBuffer()`, del tipo `GL_ARRAY_BUFFER`. Existen diversos tipos de *buffer objects* (e.g. `GL_ATOMIC_COUNTER_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_SHADER_STORAGE_BUFFER`, `GL_TEXTURE_BUFFER`, entre otros) siendo el *array buffer* el más idóneo para almacenar datos geométricos. Para información más detallada, la wiki de OpenGL en `http://www.opengl.org/wiki/Buffer_Object`.
3. Para inicializar de valores un buffer, se debe invocar a `glBufferData()` que copia los datos desde la aplicación a la memoria de la GPU. En caso de requerir actualizar los datos previamente almacenados, se debe realizar la misma operación.
4. Finalmente, cuando se realice el render de los datos almacenados en los buffers, se debe invocar nuevamente a `glBindVertexArray()` para hacerlo (esto se efectuará de la misma manera para todos los VBOs asociados a un VAO).

En el Código 5 se crea un solo buffer para almacenar los arreglos de vértices y de colores, sin transferir los datos aún. Esta procedimiento es equivalente a invocar la función `malloc` en lenguaje C. Luego, la función `glBufferSubData()` permite reemplazar una sub-sección del buffer (una para los datos de los vértices, y otra para los colores).

El paso final para la completa inicialización de los datos en OpenGL moderno, consiste en especificar cuáles atributos de los vértices se emplearán en el pipeline gráfico. Para ello, se debe conectar la información de variables presentes en el vertex shader con la información geométrica de los objetos 3D (*shader plumbing*). Con ello, se asocia cuáles son los buffers que correspondes a las variables de los shaders en un instante de tiempo durante el rendering. Cada atributo que fue habilitado, ver Código 2, debe ser asociado a una variable del tipo `in` dentro del vertex shader. Para ello, se debe obtener la ubicación de cada atributo (función `getLocation`). La secuencia de instrucciones a realizar se muestra en el Código 6.

Código 6: *Asociación de los atributos con los buffers creados (continuación del Código 5).*

```
// Vertex
glEnableVertexAttribArray(m_program.getLocation("vVertex"));
glVertexAttribPointer(m_program.getLocation("vVertex"), 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
// Colors
glEnableVertexAttribArray(m_program.getLocation("vColor"));
glVertexAttribPointer(m_program.getLocation("vColor"), 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(vec_points)));
// unbind the use of buffers
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Para este ejemplo, la ubicación de los atributos es `vVertex` y `vColor` corresponde a los valores de 0 y 1 respectivamente, siendo el siguiente fragmento de código equivalente al Código 6 para el caso del atributo `vVertex`:

```
//vertex
glEnableVertexAttribArray(0)
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
```

Esto se debe a que el vertex shader contiene el calificador layout que indica la ubicación del atributo, ver Código 3. De esta forma, los datos del buffer que corresponden al arreglo `vec_points` se asocian a la variable `vVertex` y el arreglo `vec_colors` a `vColor`.

Solamente resta definir a `BUFFER_OFFSET` que es una macro simple para crear un código más legible, su definición es:

```
#define BUFFER_OFFSET( offset )   ((GLvoid*) (offset))
```

Para este punto, se tienen inicializadas y configuradas las variables involucradas en el rendering. Ahora, se procede a construir una función de dibujado de las primitivas almacenadas. Existen diversas funciones que permiten realizar este proceso. Una de las más simples es `glDrawArrays()`, que permite especificar qué tipo de primitivas gráficas se desplegarán, en cuál vértice del arreglo de vértices habilitado comenzará, y cuántos vértices serán enviados para su despliegue. El código asociado a este despliegue se observa en el Código 7.

Código 7: *Instrucciones dentro de la función de rendering en OpenGL.*

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glClearColor(0.15f, 0.15f, 0.15f, 1.f);          //a grey color

mModelViewMatrix = glm::translate(glm::mat4(), glm::vec3(0,0,-1.5));  //-1.5 or other value behind the near clip plane

m_program.enable();
  glUniformMatrix4fv(m_program.getLocation("mView"), 1, GL_FALSE, glm::value_ptr(mViewMatrix));
  glUniformMatrix4fv(m_program.getLocation("mModel"), 1, GL_FALSE, glm::value_ptr(mModelMatrix));
  glUniformMatrix4fv(m_program.getLocation("mProjection"), 1, GL_FALSE, glm::value_ptr(mProjMatrix));
  glBindVertexArray(m_iIndexVAO);
    glDrawArrays(GL_TRIANGLES, 0, iNVertices);
  glBindVertexArray(0);
m_program.disable();
```

En este código, se construye la matriz de *ModelView* con alguna transformación (traslación en este ejemplo). Del mismo modo, a la matriz de proyección se le asigna generalmente una proyección perspectiva u ortogonal (e.g. dentro de la función de re-dimensionamiento de la viewport). Luego, se habilita el programa de shaders donde dichas matrices son pasadas a los programas en GLSL.

Posteriormente, se hace un `glBindVertexArray` del VAO creado y se procede a invocar la función de dibujado. Dicha función desplegará triángulos desde el índice 0 del buffer construido en el VAO asociado con la variable `m_iIndexVAO`, un total de `iNVertices` vértices involucrados.

Finalmente, se obtiene un simple despliegue de un cubo coloreado en cada vértice. Es importante recordar, que en este ejemplo no se hace uso eficiente de los buffers dado que se duplica la información por cada triángulo. El código completo presentado en este trabajo se encuentra disponible en `https://github.com/esmitt/OGLCube` para su descarga.

6 Conclusiones

Las funciones provistas en OpenGL moderno permiten un manejo total de las operaciones y transformaciones de las primitivas de despliegue. Se hace necesario el conocimiento del pipeline gráfico para realizar la interacción y control de cada una de las etapas programables provistas por OpenGL. Al mismo tiempo, los shaders creados en GLSL permiten una amplia flexibilidad sobre cada uno de lo vértices, primitivas y fragmentos generados.

El uso de buffers permite que el almacenamiento del lado servidor de OpenGL sea eficiente en su ejecución en el hardware gráfico. Dado que es posible realizar una asociación entre los buffers y la manera de operar dentro de un shader, es posible emplear las operaciones escritas para un shader en distintos datos provistos por distintos buffers. Igualmente, es posible aplicar diversos programas de shader a un mismo buffer.

Las posiciones, colores, coordenadas de texturas u otros atributos asociados a un vértice en un mundo 3D, permiten que se puedan efectuar operaciones en conjunto con otros valores dentro de un escenario virtual de forma eficiente. Dichas operaciones generalmente relacionadas con efectos gráficos como iluminación, *motion blur*, sombras, entre otros, u otras operaciones basadas en GPGPU -*General-Purpose computing on Graphics Processing Units*-, abren la opción a la creación de efectos muy particulares que brindan un alto realismo gráfico ó logran explotar de forma adecuada las capacidades gráficas de las GPU actuales.

Bibliografía

ANGEL, E., AND SHREINER, D. 2012. Introduction to Modern OpenGL Programming. In *ACM SIGGRAPH 2012 Courses*, ACM, New York, NY, USA, SIGGRAPH '12, 2:1–2:109.

SELLERS, G., WRIGHT, R. S., AND HAEMEL, N. 2013. *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 6th ed. Addison-Wesley Professional, July.

SHREINER, D., SELLERS, G., KESSENICH, J. M., AND LICEA-KANE, B. M. 2013. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*, 8th ed. Addison-Wesley Professional, March.

WOLFF, D. 2011. *OpenGL 4.0 Shading Language Cookbook*, 1st ed. Packt Publishing, July.