Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Lecturas en Ciencias de la Computación
ISSN 1316-6239

Comparación de Rendimiento de Múltiples Técnicas de Despliegue de Modelos Estáticos de OpenGL 4.5

Miguel Astor, Francisco Moreno y Rhadamés Carmona

RT 2018-01

Centro de Investigación en Comunicación y Redes
Caracas, Marzo 2018

Comparación de Rendimiento de Múltiples Técnicas de Despliegue de Modelos Estáticos de OpenGL 4.5

Miguel A. Astor[1], Francisco Moreno[2] y Rhadamés Carmona[2]
Universidad Central de Venezuela, Facultad de Ciencias, Esc. de Computación
[1]Centro de Investigación en Comunicaciones y Redes CICORE
[2]Centro de Computación Gráfica CCG
Urb. Valle Abajo, San Pedro, Caracas, Venezuela
Email: {miguel.astor, francisco.moreno, rhadames.carmona}@ciens.ucv.ve

***Resumen*—OpenGL es un API para el desarrollo de programas gráficos, ampliamente utilizada en la industria de la Computación Gráfica. El principal objetivo de OpenGL es abstraer el GPU al programador, de forma que este pueda concentrarse en la lógica de la aplicación gráfica en lugar de los detalles del hardware. La tarea más común que se realiza con OpenGL es el despliegue de mallados de triángulos o modelos en 3-D, tarea que en OpenGL 4.5 (la versión más reciente) puede ser realizada con múltiples técnicas. En este trabajo de investigación se realizó un estudio comparativo del rendimiento de estas técnicas aplicadas a la tarea de desplegar modelos estáticos (sin animación). Concretamente, se evaluó el desempeño del uso de *Display Lists* con despliegue en modo inmediato, y el uso de *Vertex Array Objects* con y sin *Instanced Rendering*. Se determinó que de estas técnicas, *Display Lists* es la más eficiente para el despliegue de modelos estáticos a medida que la cantidad de modelos a desplegar es mayor.**

Palabras clave*—*OpenGL, modelos estáticos, comparación de rendimiento, Display List, Vertex Array Object, Instanced Rendering.

I. Introducción

Académicamente, el campo de la Computación gráfica concierne al estudio de las técnicas y algoritmos utilizados para la generación de imágenes sintéticas en tiempo real o no, sea a partir de datos bidimensionales, tridimensionales, o incluso de orden superior [1]. Industrialmente, la Computación Gráfica se centra en el desarrollo y uso de herramientas que aplican las técnicas y algoritmos mencionados dentro de una amplia variedad de tópicos, como son la presentación de información, el cine de animación, el CAD (*Computer Aided Design* - Diseño Asistido por Computadora), la visualización de datos médicos, entre otros [1].

Independientemente de si se trata de un ámbito académico o industrial, la generación de imágenes sintéticas, en particular si se realiza en tiempo real, necesita de alta eficiencia para poder producir resultados en tiempos aceptables a los usuarios. Una forma de lograr esta eficiencia es usando hardware de propósito específico diseñado para realizar a nivel de micro-arquitectura, las operaciones algebraicas y geométricas tan comunes en la Computación Gráfica. Este hardware es conocido comúnmente como GPU (*Graphics Processing Unit* - Unidad de Procesamiento de Gráficos) [2]. Sin embargo, como suele suceder en la industria de la computación, distintos fabricantes de GPUs desarrollan arquitecturas diferentes, lo que dificulta la tarea de los programadores.

Para lograr la portabilidad de las aplicaciones gráficas, se propuso en la década de 1980 la creación de un sistema estándar que permitiera escribir programas gráficos de forma independiente del hardware de salida. Este primer sistema, conocido como GKS (*Graphical Kernel System* - Sistema de Núcleo Gráfico), fue desarrollado por la ISO (*Iternational Standards Organization* - Organización Internacional de Estándares) en 1984 [3]. GKS tenía como objetivo el despliegue de gráficos bidimensionales.

Posteriormente, GKS fue actualizado con capacidades para el despliegue de primitivas y modelos 3D con un conjunto de extensiones conocidas como PHIGS (*Programmer's Hierarchical Interactive Graphics Standard* - Estándar de Gráficos Jerárquicos Interactivos para Programadores). PHIGS derivó en los sistemas PHIGS+ y SPHIGS (*Simple PHIGS* - PHIGS Simple), los cuales agregaban más herramientas para despliegue 3D como NURBS (*Non-Uniform Rational B-Spline - B-Spline* Racional No-Uniforme), al mismo tiempo que simplificaban la interfaz de programación [1].

A finales de la década de 1980 la empresa Silicon Graphics, Inc. (SGI) publicó la biblioteca propietaria IRIS GL (*Integrated Raster Imaging System Graphics Library* - Biblioteca Gráfica del Sistema de Imágenes *Raster* Integradas) o sencillamente GL, diseñada especialmente para ser usada en las estaciones de trabajo que fabricaba la empresa. GL se convirtió en el estándar de facto de la comunidad de la Computación Gráfica, desplazando completamente al sistema PHIGS [3].

En 1992, después de un proceso de extensión y formalización de la interfaz de la biblioteca, SGI decidió publicar IRIS GL como un estándar abierto que fue llamado OpenGL (*Open Graphics Library* - Biblioteca Gráfica Abierta), diseñado para ser independiente del hardware [3]. Actualmente, OpenGL es desarrollado por el OpenGL ARB (OpenGL *Architecture Review Board* - Comité de Revisión de Arquitectura de OpenGL), asociado al consorcio industrial The Khronos™Group, Inc.

OpenGL se caracteriza por ser una herramienta de bajo nivel que permite especificar el procedimiento de despliegue de modelos tridimensionales, utilizando una interfaz de programación completamente independiente del hardware gráfico

subyacente. Se dice que OpenGL es una herramienta de bajo nivel porque no provee abstracciones de ningún tipo para la especificación de datos y programas, enfocándose exclusivamente en la representación en memoria de primitivas geométricas (puntos, lineas, triángulos, etc.), y la especificación de comandos para ser ejecutados por la GPU [3].

OpenGL ha evolucionado considerablemente desde sus inicios, dados sus más de 25 años de historia. De esta forma, las herramientas que provee OpenGL para la representación en el hardware gráfico, así como los comandos utilizados para el despliegue de modelos en OpenGL moderno son considerablemente diferentes a aquellos provistos por OpenGL clásico. Sin embargo, a pesar de esta evolución OpenGL sigue siendo compatible con los mecanismos clásicos, siendo posible escribir aplicaciones que hagan uso de los distintos mecanismos de despliegue, incluso simultáneamente dentro de una misma aplicación [3]–[5].

Dado que OpenGL provee múltiples mecanismos de despliegue de modelos, se hace interesante entonces el conocer cual de estos mecanismos es el más adecuado para determinadas tareas de despliegue. En base a esto, planteamos como objetivo de este trabajo de investigación el realizar un estudio comparativo del desempeño de las distintas técnicas de despliegue disponibles en la versión más reciente de OpenGL (la 4.5), en particular en lo referente a la tarea de desplegar modelos estáticos. Por lo que se pudo determinar, esta comparación es una carencia notoria en la bibliografía, razón por la cual se justifica este trabajo.

El resto de este documento se estructura de la siguiente manera. En la Sección II se describen en detalle los modos de funcionamiento de OpenGL 4.5. Las pruebas realizadas con las distintas técnicas de despliegue se presentan en la Sección III, donde también se analizan e interpretan los resultados obtenidos. Finalmente, en la Sección IV se muestran nuestras conclusiones y se proponen posibles trabajos futuros.

II. Técnicas de Despliegue en OpenGL 4.5

OpenGL es la especificación de una API (*Application Programming Interface* - Interfaz para la Programación de Aplicaciones) abierta para el desarrollo de aplicaciones en el lenguaje C que necesiten desplegar gráficos en 2D y 3D dentro de algún sistema de ventanas. La especificación de OpenGL es desarrollada por el consorcio industrial The Khronos™Group, Inc., el cual abarca múltiples empresas, universidades e individuos involucrados con el estudio y la industria de la Computación Gráfica. Adicional a OpenGL, The Khronos™Group, Inc. publica, entre otras, las especificaciones de las siguientes APIs:

- **OpenGL ES**: Adaptación de OpenGL para sistemas embebidos.
- **Vulkan**: Reemplazo de OpenGL en sistemas de escritorio.
- **OpenCL**: Desarrollo de aplicaciones para GPGPU[1].
- **OpenVX**: Visión por computador.
- **OpenVG**: Despliegue de gráficos vectoriales.
- **OpenXR**: Realidad Virtual.
- **WebGL**: Adaptación de OpenGL para desarrollo Web.

[1]GPGPU (*General Purpose Graphics Processing Unit* - Unidad de Procesamiento de Gráficos para Propósito General).

OpenGL se define como un API que "*consiste en un conjunto de varios cientos de procedimientos y funciones que permiten a un programador el especificar* shaders*, objetos y operaciones involucradas en la producción de imágenes gráficas de alta calidad, específicamente imágenes a color de objetos tridimensionales*" [4]. Estos objetos tridimensionales que despliega OpenGL son especificados como mallados de triángulos, mallados que son procesados por programas especializados llamados *shaders* (sombreadores), encargados de aplicar transformaciones geométricas o de otra índole para luego generar la imagen sintética final.

La versión más reciente de OpenGL es la 4.5, especificada en tres documentos que describen el llamado perfil central o perfil núcleo [4], el perfil de compatibilidad [5] y el lenguaje de programación de *shaders* GLSL [6]. El perfil central de OpenGL define el funcionamiento y los comandos disponibles a los programadores para el control del llamado *shader-based pipeline* usado por las versiones de OpenGL desde la versión 3.1 en adelante [7]. Por su parte, el perfil de compatibilidad define el funcionamiento y los comandos disponibles en el *fixed-function pipeline* usado por versiones de OpenGL anteriores a la 3.1.

De esta forma, OpenGL provee dos modos de funcionamiento dependiendo del perfil que se esté utilizando en la aplicación a desarrollar. Ambos modos de funcionamiento se basan en flujos de procesamiento de datos conocidos como tuberías o *pipelines* gráficos, los cuales definen las etapas por las que deben pasar los datos generados por el programa de aplicación para ser desplegados como una imagen sintética. Como se mencionó anteriormente, OpenGL define dos *pipelines* gráficos: el *shader-based pipeline* y el *fixed-funcion pipeline*.

II-A. Despliegue con el Shader-Based Pipeline

Desde la versión 3.1 de OpenGL se ha favorecido el uso de un *pipeline* gráfico caracterizado por poseer etapas programables que permiten al desarrollador el controlar el funcionamiento de la GPU mediante programas escritos en el lenguaje GLSL [7]. El flujo de datos de este *pipeline* puede observarse en la Figura 1.

El *shader-based pipeline* posee 11 etapas básicas para la generación de imágenes sintéticas, de las cuales 5 son programables y las otras seis son fijas y definidas por la especificación. La primera etapa es el *Vertex Puller* o extractor de vértices la cual, como su nombre indica, se encarga de extraer vértices de las estructuras de datos que utiliza OpenGL para almacenar datos geométricos. Estas estructuras de datos serán descritas más adelante en las Secciones II-A1 y II-A2.

En OpenGL, un vértice corresponde a una esquina de un triángulo y se representa como un punto 3D, el cual puede tener asociado información como su posición, color, vector normal y coordenadas de textura, así como cualquier otra

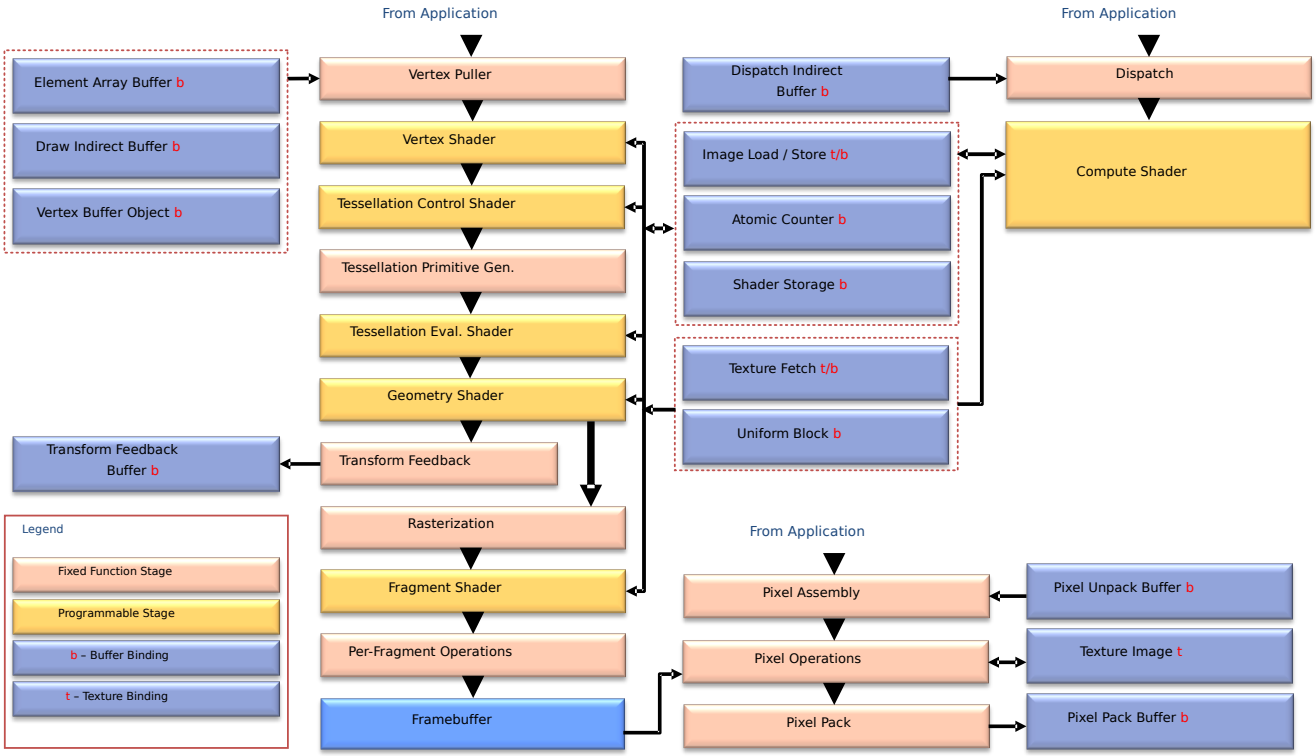

Figura 1: Flujo de datos del *shader-based pipeline* de OpenGL 4.5. Figura recuperada de [4].

información adicional que provea la aplicación. Los vértices extraídos son pasados uno por uno a la siguiente etapa que corresponde al *Vertex Shader*, la cual es controlada por un programa escrito en GLSL para aplicar transformaciones a los vértices. El *Vertex Shader* debe aplicar como mínimo las transformaciones de modelo, vista y proyección al vértice para su posterior despliegue. La definición del *Vertex Shader* es obligatoria por parte del programador [4].

Por cada vértice, la salida del *Vertex Shader* incluye como mínimo el vértice resultante de aplicar las transformaciones mencionadas al vértice original, así como los datos resultantes de aplicar cualquier otra transformación sobre la información adicional incluida con los vértices. Estos resultados son enviados a la siguiente etapa programable, el *Tessellation Control Shader*. Esta etapa recibe agrupaciones de vértices que definen triángulos o cuadriláteros, y define parámetros para la aplicación de un algoritmo de subdivisión de primitivas en la etapa siguiente, correspondiente a la *Tessellation Primitive Generator* (generador de primitivas subdivididas). La definición de un *Tessellation Control Shader* es opcional y en caso de que no se defina uno, la etapa del *Tessellation Primitive Generator* deja pasar cada primitiva recibida a la siguiente etapa sin subdividirlas [4].

Después de la aplicación del *Tessellation Primitive Generator*, se procede a la etapa del *Tessellation Evaluation Shader*, otra etapa programable, la cual puede aplicar transformaciones a las primitivas subdividas generadas por la etapa anterior. Esta etapa también es opcional. Sin embargo, si el programador definió un *Tessellation Control Shader* anteriormente, entonces es obligatorio definir su correspondiente *Tessellation Evaluation Shader* [4]. El resultado de esta etapa es enviado a la etapa del *Geometry Shader*, la última etapa programable que es opcional, encargada de generar las primitivas finales que serán utilizadas para la generación de la imagen final. El *Geometry Shader* puede aplicar cálculos y transformaciones sobre cada primitiva generada anteriormente, así como generar nueva geometría [4].

En caso de estar habilitada, la salida del *Geometry Shader* puede ser enviada a la etapa de *Transform Feedback* (retroalimentación de transformaciones), la cual se encarga de actualizar las estructuras de datos enviadas al *pipeline* al inicio del proceso, de forma que las transformaciones aplicadas puedan ser utilizadas por el programa de control cuando culmine la ejecución del *pipeline* [4].

Independientemente de si se utiliza el mecanismo de *Transform Feedback* o no, la salida del *Geometry Shader* es enviada a la etapa de *rasterization*, la cual realiza la proyección de las primitivas generadas sobre el plano de proyección, aplicando de ser necesarios los algoritmos de *stencil* (máscara), para enmascarado de regiones del plano de proyección, y *Z-Buffer* para ordenamiento en profundidad de las primitivas a desplegar [4].

La etapa de *rasterization* produce fragmentos, los cuales son procesados por la etapa del *Fragment Shader*. El *Fragment*

Shader es una etapa programable obligatoria, la cual se encarga de calcular el color final que debe poseer cada fragmento en base a los datos generados en las etapas anteriores [4]. En esta etapa es donde se suelen aplicar los modelos de iluminación y texturizado [7]. Finalmente, los fragmentos coloreados que genera el *Fragment Shader* son enviados a la etapa de *Per Fragment Operations* (operaciones por fragmento), la cual se encarga de determinar cuales fragmentos serán utilizados para generar la imagen final.

La salida de esta última etapa es una imagen que es almacenada en una estructura de datos llamada *Framebuffer Object*. Este *Framebuffer Object* puede ser mostrado directamente al usuario por medio del sistema de ventanas, o puede ser almacenado en memoria para su uso como una textura, o para su posterior procesamiento con técnicas de visión por computador.

Adicional a las etapas descritas anteriormente, OpenGL define las siguientes series de etapas opcionales adicionales. Por una parte las etapas de *dispatch* (despacho) y *Compute Shader*, la primera fija y la segunda programable, se usan para realizar cómputo mediante el paradigma GPGPU dentro de programas que hacen uso de OpenGL sin necesidad de utilizar APIs adicionales como OpenCL. Estas etapas se utilizan aparte del *pipeline* gráfico [4].

Por otra parte, las etapas de *Pixel Assembly* (ensamblado de píxeles), *Pixel Operations* (operaciones por píxel) y *Pixel Pack* (empaquetado de píxeles) son utilizadas automáticamente por OpenGL para controlar como se realiza el almacenamiento de los píxeles dentro de los *Framebuffer Objects* [4]. Estas etapas pueden ser configuradas por el programador aunque no son programables.

Los datos de entrada necesarios para ejecutar este *pipeline* son provistos a OpenGL en una serie de estructuras de datos definidas por la especificación [4]. Estas estructuras se utilizan para definir tanto los datos geométricos (los vértices que componen los triángulos de todos los mallados a desplegar), así como las texturas y los programas de GLSL que definen las distintas etapas programables del *pipeline*. Estas estructuras de datos se definen en las Secciones siguientes.

1) Vertex Array Objects: En OpenGL 4.5 un *Vertex Array Object* o VAO es una estructura de datos opaca utilizada para almacenar metadatos referentes a un conjunto de vértices [4]. Por si solo un VAO no almacena los vértices ni posibles datos asociados tales como colores, vectores normales o coordenadas de textura, entre otros. Los datos de los vértices en si son almacenados en una estructura de datos asociada al VAO conocida como*Vertex Buffer Object* o VBO. Los VBO se definen como regiones de tamaño fijo de la memoria operativa la GPU que se utilizan para el almacenamiento de datos provenientes de la aplicación [4]. Los VAO se utilizan para definir cuales VBO contienen las variables de entrada (conocidas en GLSL como atributos [6]) que serán enviadas por el *Vertex Puller* al *Vertex Shader*, mientras que por su parte los VBO son usados para almacenar, leer o modificar los datos de los vértices [4]. El Listado 1 muestra el código fuente necesario para la creación y destrucción de un VAO y estructuras asociadas.

Listado 1: Creación y destrucción de un VAO y estructuras asociadas.

```
#define BUFFER_OFFSET(i) ((char *)NULL + (i))

/* Creacion de un VAO. */
GLuint vao;
glGenVertexArrays(1, &vao);

/* Creacion de un VBO. */
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

/* El VBO contiene posiciones y vectores
   normales, cada uno con tres componentes
   de tipo float. */
glBufferData(GL_ARRAY_BUFFER,
             (num_positions * 3) *
             (num_normals * 3) *
             sizeof(float),
             NULL,
             GL_STATIC_DRAW );

/* La funcion glBufferSubData se utiliza
   para agregar datos al VBO. */
glBufferSubData(GL_ARRAY_BUFFER,
                0,
                (num_positions * 3) *
                sizeof(float),
                &positions[0]);
glBufferSubData(GL_ARRAY_BUFFER,
                (num_positions * 3) *
                sizeof(float),
                (num_normals * 3) *
                sizeof(float),
                &normals[0]);

/* Asociacion del VBO al los atributos
   de un shader especifico. */
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

/* La ubicacion en memoria del atributo
   posicion se guarda en position_attr. */
glEnableVertexAttribArray(position_attr);
glVertexAttribPointer(position_attr,
          3,
          GL_FLOAT,
          GL_FALSE,
          0,
          BUFFER_OFFSET(0));

/* La ubicacion en memoria del atributo vector
   normal se guarda en normal_attr. */
glEnableVertexAttribArray(normal_attr);
glVertexAttribPointer(normal_attr,
          3,
          GL_FLOAT,
          GL_FALSE,
          0,
          BUFFER_OFFSET((num_positions * 3) *
                        sizeof(float)));

  /* Destruccion del VBO y el VAO. */
  glDeleteBuffers(1, &vbo);
  glDeleteVertexArrays(1, &vao);
```

Existen dos formas de utilizar un VAO para desplegar geometría en OpenGL 4.5. La primera forma consiste en sencillamente usar el VAO para el despliegue de los datos señalados por este. Sin embargo, si es necesario desplegar la misma geometría múltiples veces, entonces es necesario usar

múltiples llamadas al mismo VAO, una por cada instancia que se desea desplegar.

Para evitar tener que realizar múltiples llamadas seguidas a un mismo VAO, OpenGL define la segunda forma de usar estas estructuras conocida como *Instanced Rendering* (despligue instanciado). Con *Instanced Rendering*, el programa especifíca con una sola llamada a OpenGL cuantas instancias de la geometría indicada por el VAO necesita desplegar, siendo posible distinguir cual instancia se está desplegando en el *Vertex Shader* mediante una variable especial. El Listado 2 muestra ambas formas de despliegue de un VAO.

Listado 2: Despliegue de un VAO.

```
glBindVertexArray(m_iIndexVAO);
/* 10 veces el mismo mallado sin
   Instanced Rendering. */
for (i = 0; i < 10; i++) {
  /* Aqui se aplican los shaders y
     transformaciones geometricas. */
  ...
  /* Despliegue. */
  glDrawArrays(GL_TRIANGLES,
               0,
               num_triangles);
}

/* Con Instanced Rendering. */
glDrawArrays(GL_TRIANGLES,
             0,
             num_triangles,
             10);
```

2) Program Objects: Para la definición de *shaders*, OpenGL 4.5 define una serie de estructuras de datos que apoyan los procesos de compilación, enlazado y uso de los programas de GLSL que pueden definirse para las distintas etapas programables del *shader-based pipeline*. La primera de estas estructuras es el *Shader Object*, usado para cargar y compilar el código fuente de un (y solo un) *shader* específico [4]. El Listado 3 muestra el proceso de creación y compilación de un *shader*, omitiendo por brevedad el código fuente necesario para verificación de errores de compilación del *shader*. El código fuente del Listado 3 funciona para cualquier tipo de *shader*, independientemente de si se trata de un *Vertex*, *Tessellation Control*, *Tessellation Evaluation*, *Geometry* o *Fragment Shader*.

Listado 3: Creación y compilación de un *shader*.

```
GLuint shader;
GLint  status;

/* Creacion del shader object, por ejemplo
   un Vertex Shader. */
shader = glCreateShader(GL_VERTEX_SHADER);

/* Compilacion del shader. */
glCompileShader(shader);

/* La variable status se utiliza para conocer
   el resultado de la compilacion del shader.
   Si su valor es GL_FALSE entonces ocurrio un
   error en el proceso. */
glGetShaderiv(shader, GL_COMPILE_STATUS, &
    status);
```

Una vez se han compilado los códigos fuente de los distintos shaders (como mínimo un *Vertex Shader* y un *Fragment Shader*), estos deben ser enlazados en un *Program Object* para poder ser usados para el despliegue de geometría. El proceso de enlazado se encarga de asignar un espacio de memoria a los *shaders*, así como de conectar las salidas de cada *shader* con las entradas del siguiente en el *pipeline*.

El *Program Object* es una estructura de datos que se utiliza para solicitar la ejecución de un conjunto de shaders, y tiene como responsabilidad el almacenar la dirección en memoria de la GPU de las distintas variables de entrada necesitadas por los shaders. Las variables de entrada se distinguen en dos clases: atributos y uniformes. Los atributos se utilizan para definir propiedades de los vértices que puede utilizar el *Vertex Shader*. Las uniformes se utilizan para pasar variables auxiliares (por ejemplo la matriz de modelo-vista-proyección) a cualquier *shader* del *Program Object*, no solo al *Vertex Shader*. El Listado 4 muestra el proceso de enlazado de un conjunto de shaders dentro de un *Program Object*. Igualmente, el Listado 4 omite las verificaciones de errores.

Listado 4: Enlazado de un *Program Object*.

```
/* Creacion del Program Object. */
GLuint program = glCreateProgram();

/* Adjuntado de los Shader Objects en el
   Program Object. En caso de usar shaders
   adicionales al Vertex y Fragment shader,
   el proceso de adjuntado es igual. */
glAttachShader(program, vert_shader);
glAttachShader(program, frag_shader);

/* Enlazado del Program Object. */
glLinkProgram(program);

/* Una vez se enlaza el Program Object, los
   Shader Objects ya no son necesarios. */
glDeleteShader(vert_shader);
glDeleteShader(frag_shader);
```

II-B. Despliegue con el Fixed-Function Pipeline

El *fixed-function pipeline* de OpenGL corresponde al flujo de datos utilizado desde las primeras versiones de OpenGL hasta la versión 3.0. En las versiones de OpenGL desde la 1.0 hasta la 1.5 este *pipeline* consistía únicamente en etapas fijas definidas por la especificación. Sin embargo, desde la versión 2.0 de OpenGL en adelante se introduce el concepto de los *shaders*, siendo en esta versión donde se permite al programador controlar parte del *pipeline* mediante la definición del *Vertex Shader* y el *Fragment Shader* [7].

El flujo de datos básico de este *pipeline* puede observarse en la Figura 2. Como se puede observar, este flujo de datos es más sencillo que el mostrado en la Figura 1. En particular, las etapas programables referentes a los distintos *shaders* no están presentes en el modelo básico. Los comandos de OpenGL y los datos necesarios son introducidos al *pipeline* por la izquierda de la Figura 2 [8].

Independientemente de la forma en que se envíen los comandos y los datos a OpenGL, estos son procesados por la etapa del *Evaluator*, la cual se encarga de preparar las

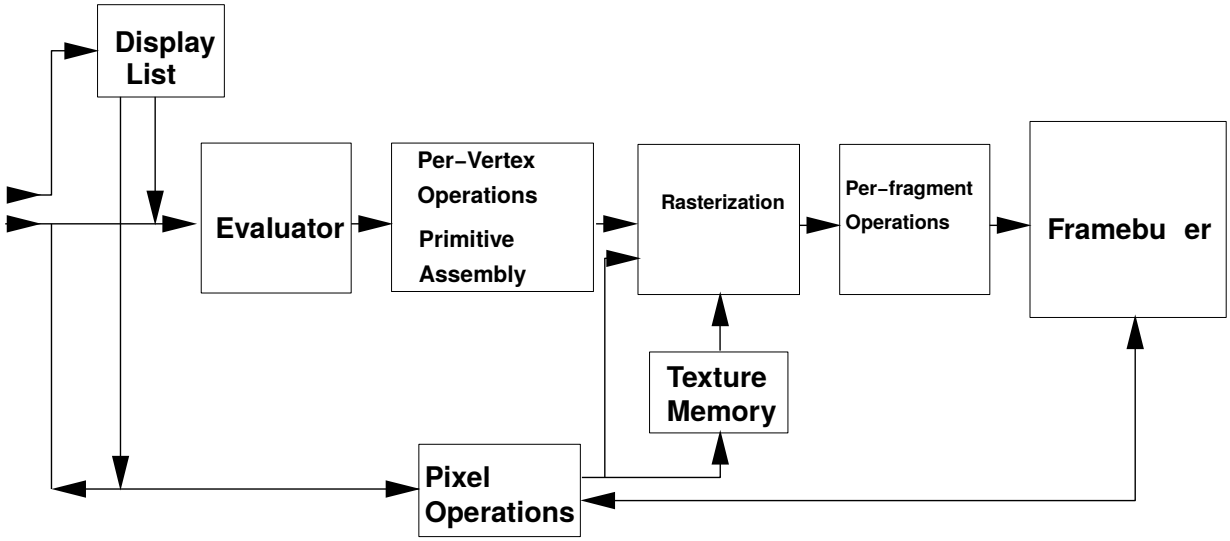

Figura 2: Flujo de datos del *fixed-function pipeline* de OpenGL. Figura recuperada de [8].

listas de vértices a procesar, y establece el estado de ejecución de OpenGL mediante la ejecución de los comandos. Luego, los vértices son enviados a la etapa de *Per-Vertex Operations* y *Primitive Assembly*, las cuales se encargan de transformar los vértices y generar los triángulos a desplegar [8]. Estas dos etapas corresponden a las etapas ubicadas entre el *Vertex Puller* y el *Geometry Shader* en el *pipeline* definido en la Figura 1, con la salvedad de que las etapas de subdivisión de primitivas no existen en el *fixed-function pipeline* [4].

Posteriormente, las primitivas son enviadas a la etapa de *Rasterization* donde se generan los fragmentos a desplegar. Estos fragmentos son enviados a la etapa de *Per-Fragment Operations*, donde se aplican los modelos de iluminación, sombreado y texturizado [8]. Finalmente, la salida de la etapa de *Per-Fragment Operations* produce la imagen final a ser almacenada en el *framebuffer* de OpenGL.

Como se mencionó anteriormente, desde OpenGL 2.0 las etapas de *Per-Vertex Operations* y *Per-Fragment Operations* pueden ser controladas mediante un *Vertex Shader* y un *Fragment Shader* respectivamente [7]. En versiones modernas de OpenGL, el *fixed-function pipeline* es provisto mediante el perfil de compatibilidad, siendo ejecutado con *shaders* que imitan el funcionamiento de versiones anteriores de OpenGL sobre el *pipeline* moderno de forma transparente [5].

Los datos geométricos y los comandos utilizados por el *fixed-function pipeline* pueden enviarse directamente a OpenGL uno a uno mediante el modo inmediato de ejecución, o pueden compilarse dentro de listas de comandos llamadas *Display Lists* (listas de despliegue) [8], las cuales se utilizan para enviar múltiples comandos de OpenGL a la GPU con una sola transferencia de datos.

1) Modo inmediato: El modo inmediato es el modo clásico de enviar datos a OpenGL [7]. Este modo se basa en el paradigma de funciones `glBegin()` y `glEnd()`, las cuales permiten especificar, uno a la vez, los vértices que componen varios tipos de primitivas [8]. El uso del modo inmediato se caracteriza por realizar una transferencia entre la CPU (*Central Processing Unit* - Unidad Central de Procesamiento) y la GPU por cada llamada ubicada dentro de los pares `glBegin()` y `glEnd()`.

A manera de ejemplo, supongamos que queremos enviar una serie de mallados de triángulos a la GPU (una cantidad arbitraria de mallados que llamaremos `num_meshes` para el ejemplo), los cuales están compuestos por cantidades de triángulos arbitrarias. En el lenguaje C, esto puede lograrse con instrucciones similares a las mostradas en el Listado 5, asumiendo que los vértices de cada triángulo están definidos de manera independiente y ordenados por triángulo al que pertenecen.

Listado 5: Despliegue de mallados triángulos en modo inmediato.

```
for (i = 0; i < num_meshes; i++) {
  glBegin(GL_TRIANGLES};
  for (j = 0; j < meshes[i].num_verts; j++) {
    /* Vectores normales. */
    glNormal3f(meshes[i].N[j].x,
               meshes[i].N[j].y,
               meshes[i].N[j].z);

    /* Coordenadas de textura 2D. */
    glTexCoords2f(meshes[i].T[j].s,
                  meshes[i].T[j].t);

    /* Color del vertice. */
    glColor4f(meshes[i].C[j].r,
              meshes[i].C[j].g,
              meshes[i].C[j].b,
              meshes[i].C[j].a,);
```

```
      /* Posicion del vertice. */
      glVertex3f(meshes[i].V[j].x,
                 meshes[i].V[j].y,
                 meshes[i].V[j].z);
    }
    glEnd();
  }
```

El código anterior se basa en las definiciones mostradas en el Listado 6 para los vértices, colores, coordenadas de textura y los mallados respectivamente.

Listado 6: Estructuras de datos simples para mallados de triángulos.

```
typedef struct VECTOR_3D {
  float x, y, z;
} vertex_t;

typedef struct COLOR_RGB {
  float r, g, b, a;
} color_t

typedef struct TEX_COORD_2D {
  float s, t;
} tex_coord_t;

typedef struct TRI_MESH {
  unsigned int num_verts;
  vertex_t *    N; /* Vectores normales. */
  tex_coord_t * T;
  color_t *     C;
  vertex_t *    V; /* Posiciones. */
} mesh_t;

mesh_t * meshes;
```

2) Display Lists: Una *Display List* es una estructura de datos que almacena una lista de datos y comandos de OpenGL para ser enviados a la GPU como un solo lote, en lugar de como comandos independientes [8]. Estos datos y comandos son almacenados en la memoria operativa de la GPU o VRAM (*Video RAM* - RAM de Video) de forma que puedan ser utilizados posteriormente sin necesidad de retransmitirlos continuamente como sucede con el despliegue en modo inmediato. Para lograr sus objetivos, las *Display Lists* deben ser precompiladas antes de proceder al despliegue.

La principal desventaja de las *Display Lists* es que los datos y comandos almacenados en una display list no pueden ser modificados una vez que esta es creada, siendo necesario destruir la *Display List* y volverla a crear para modificar datos. Las funciones usadas en la creación, uso y destrucción de una *Display List* se muestra en el Listado 7.

Listado 7: Creación y uso de una *Display List*.

```
/* Creacion de una nueva Display List. */
GLuint display_list = glGenLists(1);

/* Inicio del proceso de compilacion de
   la Display List. */
glNewList(display_list, GL_COMPILE);
/* Comandos y datos para OpenGL. */
...
glEndList();
/* En este punto la Display List se
   encuentra disponible en la GPU. */

/* Para indicar al GPU que ejecute los
   comandos de la Display List. */
glCallList(display_list);

/* Para destruir la Display List. */
glDeleteLists(display_list, 1);
```

III. Comparación de Rendimiento de Técnicas de Despliegue de Modelos Estáticos

En este trabajo de investigación se realizó una prueba comparativa de rendimiento de las técnicas de despliegue descritas en la Sección II para la tarea de desplegar modelos estáticos. En particular se comparó el desempeño del uso de VAO (con y sin *Instanced Rendering*) y de *Display Lists*, usando el escenario descrito en la Sección III-A. Todas las pruebas fueron realizadas con el entorno descrito en la Sección III-B.

Por modelos estáticos nos referimos a mallados de triángulos sin animación los cuales además son utilizados en modo de solo lectura, es decir, sin utilizar el mecanismo de *Transform Feedback* de OpenGL.

III-A. Diseño de las Pruebas

Para comparar el rendimiento de las técnicas de despliegue con VAO y con *Display Lists*, se realizaron una serie de pruebas con base en los modelos mostrados en la Figura 3. La cantidad de triángulos de cada modelo se muestran en la Tabla I.

Tabla I: Cantidad de triángulos por modelo.

Modelo	Cantidad de Triángulos
Suzane	3872
Goblin	9499
Train	31477

Las pruebas consistieron en el despliegue de múltiples instancias de los modelos mencionados durante un minuto, tiempo en el cual se midieron la cantidad de cuadros desplegados y los tiempos necesarios para el despliegue de los mismos. Se tomaron como medidas de rendimiento la cantidad total de cuadros durante toda la ejecución, así como los tiempos mínimo, máximo y promedio para el despliegue de los cuadros, medidos en milisegundos por cuadro, limitando cada escenario a un máximo de 60 cuadros por segundo aproximadamente. Se probaron cuatro escenarios por modelo, con 42, 168, 1200 y 10000 instancias de cada modelo. Cada escenario fue repetido 10 veces y luego promediado para tratar de mitigar el error estadístico.

El despliegue de los modelos se realizó con el código fuente que muestran los Listados 8, 9 y 10 para el uso de VAO sin *Instanced Rendering*, VAO con *Instanced Rendering* y *Display Lists* respectivamente. En todos estos fragmentos de código las variables `xOffset` y `yOffset` se refieren al desplazamiento aplicado a cada instancia de los modelos para que estas no aparezcan unas sobre las otras. El código fuente completo de la aplicación desarrollada para realizar las pruebas puede obtenerse en [9]. Los shaders utilizados para el despliegue con VAO se listan en el Apendice A, y los

(a) (b) (c)

Figura 3: Modelos estáticos usados en las pruebas: (a) Suzane, (b) *Goblin*, (c) *Train*.

correspondientes shaders usados para la técnica de *Display List* se listan en el Apendice B.

Listado 8: Código de prueba de VAO sin *Instanced Rendering*.

```
xOffset = xInit;
for(int i = 0; i < xInstances; i++) {
  yOffset = yInit;
  for(int j = 0; j < yInstances; j++) {
    /* Se omite el codigo de inicializacion
    de los shaders. */
    ...
    glBindVertexArray(m_iIndexVAO);
    glDrawArrays(GL_TRIANGLES,
    0,
    obj.Triangles.size() /
    obj.dataPerTriangle * 3);
    glBindVertexArray(0);
    yOffset += stride;
  }
  xOffset += stride;
}
```

Listado 9: Código de prueba de VAO con *Instanced Rendering*.

```
/* Se omite el codigo de inicializacion
de los shaders. */
...
glBindVertexArray(m_iIndexVAO);
glDrawArraysInstanced(GL_TRIANGLES,
0,
obj.Triangles.size() /
obj.dataPerTriangle * 3,
xInstances * yInstances);
glBindVertexArray(0);
```

Listado 10: Código de prueba de *Display List*.

```
xOffset = xInit;
for(int i = 0; i < xInstances; i++) {
  yOffset = yInit;
  for(int j = 0; j < yInstances; j++) {
    glLoadIdentity();
    /* Se omiten las transformaciones
    geometricas y el codigo de
    inicializacion de los shaders. */
    ...
    glCallList(m_iIndexList);
    yOffset += stride;
  }
  xOffset += stride;
}
```

III-B. Entorno de Pruebas

Para el desarrollo de las pruebas se usaron las dos PC de escritorio descritas en la Tabla II. Como se indica, todas las pruebas fueron realizadas sobre distribuciones del sistema operativo GNU/Linux en versiones de 64 bits. Los dos tipos de GPU utilizados soportan OpenGL 4.5 [10], [11].

Tabla II: Equipos usados en las pruebas.

	PC 1	**PC 2**
Procesador	Intel Core i7 3.4 GHz	Intel Core i5 3.2 GHz
RAM	8 GB	4 GB
Sistema Operativo	GNU/Linux Elementary OS 0.3	GNU/Linux Ubuntu 14.04
Arquitectura	AMD64	AMD64
GPU	Nvidia GeForce GTX 660	Nvidia GeForce 405m
VRAM	2 GB [10]	512 MB [11]
Driver	Nvidia 352.63	Nvidia 367.57

III-C. Resultados Obtenidos

En esta sección se muestran los resultados obtenidos en cada una de las pruebas realizadas. Los resultados se presentan agrupados por cada PC descrita en la Sección III-B.

1) Resultados en la PC 1: Los resultados obtenidos en la PC 1, basada en una GPU Nvidia GTX 660, se muestran en las Figuras 4a, 4c y 4e para los modelos Suzane, *Goblin* y *Train* respectivamente. Como se puede observar, de las técnicas evaluadas *Display Lists* es la que produjo mejores resultados con todos los modelos.

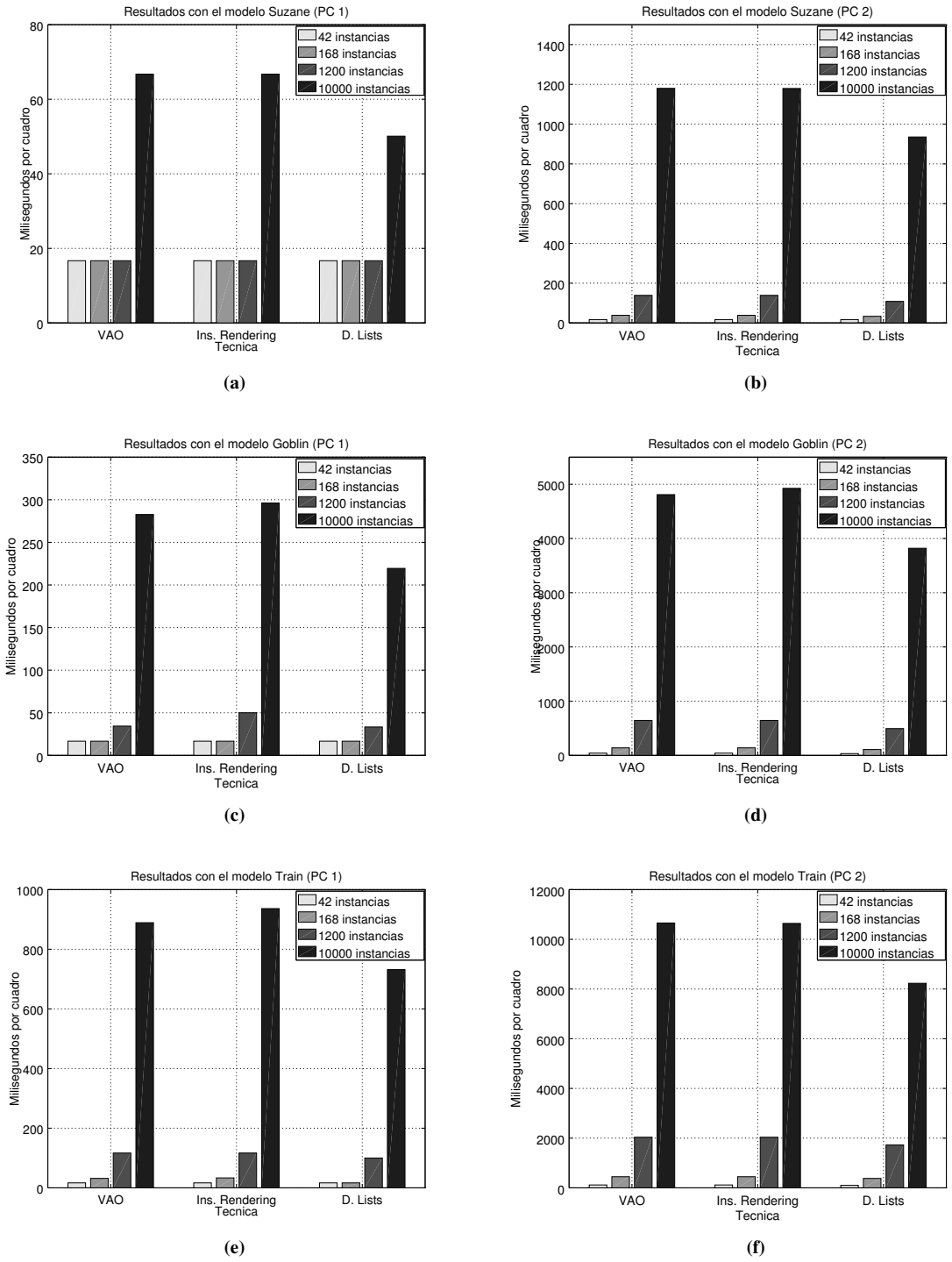

Figura 4: Tiempos promedio para despliegue de un cuadro por cada modelo: (a) Resultados con el modelo Suzane en la PC 1, (b) Resultados con el modelo Suzane en la PC 2, (c) Resultados con el modelo *Goblin* en la PC 1, (d) Resultados con el modelo *Goblin* en la PC 2, (e) Resultados con el modelo *Train* en la PC 1, (f) Resultados con el modelo *Train* en la PC 2.

Como se puede observar en las Figuras, cuando la cantidad de modelos es poca los tiempos necesarios para desplegar un cuadro son muy similares entre todas las técnicas. Sin embargo, a medida que la cantidad de modelos incrementa se puede observar que las *Display Lists* producen tiempos de despliegue aproximadamente un 20 % menores que las otras dos técnicas. Este resultado se comprueba con los tres modelos utilizados.

Así mismo, los tiempos producidos por las técnicas VAO e *Instanced Rendering* son muy similares entre si para todos los casos evaluados. Sin embargo, en las Figuras 4c y 4e se puede apreciar como el uso de *Instanced Rendering* produce tiempos ligeramente mayores que VAO.

2) Resultados en la PC 2: Los resultados obtenidos con la PC 2, basada en la GPU Nvidia 405m, se presentan en las Figuras 4b, 4d y 4f respectivamente para los modelos Suzane, *Goblin* y *Train*. Los resultados obtenidos en esta PC son consistentes con los resultados mostrados anteriormente para la PC 1.

IV. Conclusiones y Trabajos Futuros

En este trabajo de investigación se realizó un estudio comparativo del rendimiento de diferentes técnicas para despliegue de modelos estáticos usando OpenGL 4.5. Las técnicas evaluadas corresponden a los mecanismos modernos de despliegue en OpenGL como lo son los *Vertex Array Objects* (VAO) con y sin *Instanced Rendering* (despliegue instanciado), junto a la técnica clásica de despliegue en OpenGL conocida como *Display Lists* (listas de despliegue) usando el modo inmediato de despliegue.

La comparación fue realizada mediante una serie de pruebas sucesivas de cada técnica de despliegue, usando los tres modelos estáticos mostrados en la Figura 3. Se evaluaron cuatro escenarios de prueba, realizando el despliegue de 42, 168, 1200 y 10000 instancias por modelo durante un minuto con cada técnica. Cada escenario fue repetido 10 veces y luego promediado para mitigar el error estadístico.

Los resultados descritos en la Sección III muestra que de las técnicas evaluadas, el uso de *Display Lists* es la técnica más eficiente para el despliegue de modelos estáticos. Esto a pesar de que *Display List* es la más antigua de las técnicas evaluadas y de estar considerada obsoleta (*deprecated*) en el perfil núcleo de OpenGL 4.5 [4].

Con base en el trabajo realizado, se proponen los siguientes trabajos futuros:

- Verificar si los resultados obtenidos son consistentes al realizar las mismas pruebas con GPUs de AMD e Intel.
- Realizar un estudio comparativo similar con modelos animados.
- Comparar el rendimiento de las técnicas de despliegue de OpenGL moderno con las técnicas de despliegue equivalentes de otros API como Vulkan y DirectX 12.

Agradecimientos

Los autores queremos agradecer al profesor MSc. Esmitt Ramírez Jacobo por sus notas docentes [7] y su código fuente para el manejo de *shaders* [12], que fueron de gran ayuda para el desarrollo de este trabajo.

Referencias

[1] J. D. Foley, A. Van Dam, S. K. Feiner, J. F. Hughes, y R. L. Phillips, *Introducción a la Graficación por Computador*. Addison-Wesley, 1996.

[2] D. A. Patterson y J. L. Hennessy, *Estructura y Diseño de Computadores*, 4th ed. Reverté, 2011.

[3] D. Hearn y M. P. Baker, *Gráficos por Computadora con OpenGL*, 3rd ed. Pearson, 2006.

[4] M. Segal y K. Akeley, *The OpenGL®Graphics System: A Specification (Version 4.5 (Core Profile) - October 24, 2016)*, The Khronos™Group Inc., octubre 2016.

[5] M. Segal y K. Akeley, *The OpenGL®Graphics System: A Specification (Version 4.5 (Compatibility Profile) - October 24, 2016)*, The Khronos™Group Inc., octubre 2016.

[6] J. Kessenich, *The OpenGL®Shading Language*, The Khronos™Group Inc., abril 2016.

[7] E. Ramirez, "Despliegue básico en opengl moderno," Escuela de Computación, Facultad de Ciencias, UCV, Caracas, Venezuela, Rep. Tecn. ND 2014-1, 2014.

[8] M. Segal y K. Akeley, *The OpenGL®Graphics System: A Specification (Version 2.0 (Core Profile) - October 22, 2004)*, The Khronos™Group Inc., octubre 2004.

[9] M. A. Astor y F. Moreno, "Performance comparison of opengl's instanced rendering, vao and display list rendering methods," https://github.com/miky-kr5/OGL4-Instanced-Perf, revisado: 2017-2-06.

[10] N. Corporation, "Geforce gtx 660 — specifications," http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-660/specifications, revisado: 2017-2-06.

[11] N. Corporation, "Geforce 405m — specifications," http://www.geforce.com/hardware/notebook-gpus/geforce-405m/specifications, revisado: 2017-2-06.

[12] E. Ramirez, "A glut code to display a simple 3d cube using opengl 4.0," https://github.com/esmitt/OGLCube, revisado: 2017-2-06.

APÉNDICE A
Shaders USADOS PARA EL DESPLIEGUE CON VAO

Listado 11: Vertex shader.

```
#version 400

precision lowp float;

layout(location = 0) in vec3 vVertex;
layout(location = 1) in vec3 vNormal;

uniform mat4 mView, mProjection, mModel;
uniform vec4 cameraPos, lightPos;
uniform bool bInstanced;
uniform int xInstances, yInstances;
uniform float xInit, yInit, stride;

out vec4 vNormalVs;
out vec4 vViewVector;
out vec4 vLightVector;

void main() {
  vec4 vPos = mModel * vec4(vVertex.xyz, 1.0);
  if(bInstanced)
    vPos = vec4(vPos.x + ((gl_InstanceID % xInstances) * stride) + xInit, vPos.y + ((gl_InstanceID /
        xInstances) * stride) + yInit, vPos.z, vPos.w);
  vNormalVs = mModel * vec4(vNormal, 0.0);
  vViewVector = normalize(cameraPos - vPos);
  vLightVector = normalize(lightPos - vPos);
  gl_Position = mProjection * mView * vPos;
}
```

Listado 12: Fragment Shader.

```
#version 400

layout(location = 0) out vec4 vFragColor;

in vec4 vNormalVs;
in vec4 vViewVector;
in vec4 vLightVector;

const vec4 baseColor = vec4(1.0, 1.0, 1.0, 1.0);
const vec4 ambient = vec4(0.025, 0.025, 0.025, 1.0);
const vec4 bLightColor = vec4(vec3(0.15), 1.0);

void main(void) {
  float NdotL, RdotV;
  vec4 r;
  vec4 diffuse = vec4(0.0);
  vec4 specular = vec4(0.0);
  vec4 n = normalize(vNormalVs);
  vec4 l = normalize(vLightVector);
  vec4 v = normalize(vViewVector);

  NdotL = max(dot(n, l), 0.0);
  if(NdotL > 0.0) {
    diffuse += vec4(NdotL);
    r = normalize(reflect(l, n));
    RdotV = max(dot(r, normalize(v)), 0.0);
    specular += vec4(pow(RdotV, 105.0));
  }

  NdotL = max(dot(n, -l), 0.0);
  if(NdotL > 0.0)
    diffuse += bLightColor * NdotL;

  vFragColor = vec4(clamp(diffuse.rgb + specular.rgb + ambient.rgb, 0.0, 1.0), 1.0);
}
```

APÉNDICE B
Shaders USADOS PARA EL DESPLIEGUE CON *Display Lists*

Listado 13: Vertex shader.

```
#version 120

uniform vec4 cameraPos, lightPos;

varying vec4 vNormal, vView, vLight;

void main(void) {
  vec4 vPos = gl_ModelViewMatrix * gl_Vertex;
  vNormal = vec4(gl_NormalMatrix * gl_Normal, 0.0);
  vView   = normalize(cameraPos - vPos);
  vLight  = normalize(lightPos - vPos);
  gl_Position = ftransform();
}
```

Listado 14: Fragment Shader.

```
#version 120

varying vec4 vNormal, vView, vLight;

const vec4 baseColor = vec4(1.0, 1.0, 1.0, 1.0);
const vec4 ambient = vec4(0.025, 0.025, 0.025, 1.0);
const vec4 bLightColor = vec4(vec3(0.15), 1.0);
const float shiny = 105.0;

void main(void) {
  float NdotL, RdotV;
  vec4 r;
  vec4 diffuse = vec4(0.0);
  vec4 specular = vec4(0.0);
  vec4 n = normalize(vNormal);
  vec4 l = normalize(vLight);
  vec4 v = normalize(vView);

  NdotL = max(dot(n, l), 0.0);
  if(NdotL > 0.0) {
    diffuse += vec4(NdotL);
    r = normalize(reflect(l, n));
    RdotV = max(dot(r, normalize(v)), 0.0);
    specular += vec4(pow(RdotV, shiny));
  }

  NdotL = max(dot(n, -l), 0.0);
  if(NdotL > 0.0)
    diffuse += bLightColor * NdotL;

  gl_FragColor = vec4(clamp(diffuse.rgb + specular.rgb + ambient.rgb, 0.0, 1.0), 1.0);
}
```