UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES
ARAGÓN

“Gráficos en 3D sobre los navegadores web usando la tecnología WebGL”

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

P R E S E N T A N:

ALEXEI EMMANUEL MARTÍNEZ MENDOZA

Y

LUIS RICARDO LEYVA CAMPOS

ASESOR DE TESIS:
M. en C. Jesús Hernández Cabrera

MÉXICO, 2012

AGRADECIMIENTOS

A mi Padre Rodolfo.

Quien siempre me ha dado su apoyo y sus consejos incondicionalmente, a pesar de la distancia en la que nos encontramos.

A mi Madre Marina.

Por ser padre y madre a la vez, ya que sin su guía esta tesis no hubiera sido posible.

A mis amigos Juan Carlos y Karin.

Por haberme dado ánimo, y por demostrarme, que a pesar de las situaciones, todavía seguimos siendo amigos.

A Jesús.

Quien nos brindó la ayuda necesaria para poder realizar este trabajo.

A mis Profesores.

Por haberme enseñado y preparado para los retos que me deparan.

A la UNAM.

Por haberme cobijado y transmitido el sentimiento de pertenencia, por dejarme ser un integrante de esta hermosa institución.

Alexei Emmanuel Martínez Mendoza

Esta tesis está dedicada a todas aquellas personas que contribuyeron en mi formación, me dieron aliento y me brindaron todo su apoyo para que esto pudiera ser posible.

Gracias a mis padres Marbella y Luis por otorgarme las oportunidades, la guía y los valores sin los cuales no habría podido culminar esta importante etapa de mi vida.

Gracias a mis hermanas, las cuales con su ejemplo y apoyo me supieron mostrar el camino para ser una persona de bien.

Gracias a mis profesores y compañeros de la FES Aragón, UNAM, quienes directa o indirectamente me ayudaron en el aprendizaje adquirido durante esta larga y enriquecedora travesía, que significó para mí el haber estudiado esta carrera.

Gracias al profesor Jesús por sus enseñanzas y el respaldo que me ofreció durante el servicio social y hasta la conclusión de este proyecto.

Luis Ricardo Leyva Campos

CONTENIDO

INTRODUCCIÓN GENERAL

El siguiente trabajo de tesis para obtener el título de Ingeniero en Computación, el cual lleva por título “Gráficos en 3D sobre los navegadores web usando la tecnología WebGL”, que como su nombre lo indica, está dedicado al estudio de WebGL, la cual es una tecnología de reciente desarrollo usada para poder mostrar gráficos tridimensionales avanzados en un navegador web, sin necesidad de aplicaciones extras para funcionar.

Para lograr esto, el trabajo se ha dividido en cinco capítulos, desglosados de la siguiente forma.

En el primer capítulo se dará una pequeña introducción de las tecnologías actuales de tercera dimensión, así como un recorrido por la historia de los gráficos tridimensionales.

El segundo capítulo mencionará principalmente, las tecnologías disponibles hoy en día para mostrar gráficos tridimensionales en la web y así, al final, poder hacer una correcta comparación entre ellas.

El tercer capítulo se enfocará en el tema de la tercera dimensión, por lo que se verán conceptos más técnicos relacionados con dicho tema, con el fin de poder comprender de mejor forma la programación en OpenGL, que se estudiará al final de este capítulo.

En el cuarto capítulo se abordará por completo a WebGL, estudiando la forma en que está estructurado y de qué manera se ejecuta en los navegadores web, por lo que podremos ver ejemplos básicos para aprender a usar esta tecnología.

El último capítulo esta dedicado al desarrollo de un ejemplo en el cual se vean empleados los conceptos aprendidos durante el desarrollo de este trabajo, para esto, se mostrará la creación de un recorrido virtual tridimensional capaz de funcionar usando solamente un navegador web.

CAPÍTULO 1

INTRODUCCIÓN A LOS GRÁFICOS EN TERCERA DIMENSIÓN EN LA WEB

Actualmente el uso de las computadoras y de tecnologías de comunicación avanzadas, como el internet, se han vuelto parte esencial de las sociedades modernas. Desde que el internet tiene presencia en las vidas de la gente ha presentado un crecimiento y un desarrollo enorme.

Lo que hoy conocemos como Internet o *World Wide Web* ha ido evolucionando desde hace más de 20 años. Comenzó como un proyecto de interconexión de diferentes nodos, utilizando hipertexto, el cual proporcionaba diferentes contenidos, entre ellos informes o bases de datos las cuales podían enlazarse con otros documentos de hipertexto; estas ideas fueron formalmente estandarizadas con el *Lenguaje de Marcado de Hipertexto* o HTML por sus siglas en inglés, que definió un vocabulario para la escritura de dichos contenidos. Aparte de los contenidos en sí, también los mecanismos para proporcionar hipertexto fueron estandarizados como el *Protocolo de Transferencia de Hipertexto* o HTTP, que define la comunicación entre el contenido que el servidor de hipertexto proporciona y un navegador que mostrará este contenido para el usuario.

En sólo un par de décadas, la web evolucionó y se convirtió en una potente herramienta a la que se le da un sin número de usos en muchas áreas de investigación o de recreación; este crecimiento se encuentra fuertemente aunado a hardware cada vez más potente y computadoras con capacidad de procesamiento mucho mayor.

HTML creció continuamente con nuevas características, como la posibilidad de incluir gráficos y figuras. Las capacidades de diseño mejoraron aún más por un estándar llamado *Hoja de Estilo en Cascada* (CSS), el cual fue el responsable de definir la presentación del contenido con el objetivo de separarlos, ya que era común el uso de una gran cantidad de código HTML para definir la presentación de un documento que no tenía ningún significado al contenido en absoluto; las capacidades de las CSS se siguen expandiendo y se están añadiéndo nuevas propiedades. Con el fin de hacer que el usuario pueda interactuar con el contenido

que se muestra, se añadió una secuencia de comandos a la web en la forma del lenguaje de programación ECMAScript; comúnmente llamado JavaScript[1], aunque no tiene relación con el lenguaje de programación Java. ECMAScript es capaz de interactuar con el contenido de una página HTML y sus propiedades de presentación a través de la interfaz de programación y está definido en el estándar Document Object Model (Modelo de Objetos del Documento, DOM). Estos tres estándares definen el contenido, la presentación y la interacción de las páginas web que están en constante evolución, con nuevas características añadidas y nuevas páginas que toman ventaja de estas técnicas para proporcionar aplicaciones que se aprovechen de toda la funcionalidad y el rendimiento de una computadora. Incluso ya hay suites de correo electrónico y procesadores de texto que se ejecutan completamente en un navegador. Un ejemplo es el proyecto llamado Chromium OS de Google que tiene por objetivo proporcionar un sistema operativo completo, que consista únicamente en un navegador y que sea capaz de ejecutar solamente aplicaciones web sin el apoyo nativo de aplicaciones de escritorio y el cual está pensado principalmente para al mercado de dispositivos móviles.

Una vez que las computadoras fueron capaces de desplegar imágenes y animaciones en sus navegadores web, se comenzó a analizar y a estudiar la posibilidad de mostrar gráficos aún más elaborados y fue entonces que la realidad virtual y la tercera dimensión despertaron un gran interés. Hubo varios intentos por hacer esto posible pero distintas incapacidades y limitaciones (tanto de las conexiones a internet como de hardware), provocaron un aletargamiento en el desarrollo del 3D para la web.

Afortunadamente, hoy en día las conexiones de banda ancha y el uso de tarjetas gráficas potentes están facilitando la penetración e implementación de los gráficos

[1] JavaScript es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

de tercera dimensión en el internet y concretamente una tecnología está logrando esto: El WebGL.

El tema principal de esta tesis es “WebGL” (Web-based Graphics Library) el cual es un estándar relativamente nuevo que aún está en fase de desarrollo y documentación y que propone traer toda una serie de nuevas características en gráficos tridimensionales conducidos a la web. Esto se consiguió mediante la creación de una “envoltura” alrededor del estándar de aplicaciones tridimensionales OpenGL ES[2], un enlace de gráficos 3D más ligero derivado del ampliamente conocido OpenGL[3] destinadas a sistemas integrados. En los siguientes capítulos de este trabajo, se explicarán los acontecimientos que han llevado al crecimiento del estándar WebGL; se explicarán a fondo los fundamentos de los gráficos 3D, además, se definirán conceptos como *rasterización* y *shaders*[4] que constituyen el estándar OpenGL ES. Se explicarán los métodos básicos de posible interacción en la web con el usuario, así como la diferencia entre los principales frameworks[5] o arquitecturas usadas en su funcionamiento, terminando con la aplicación práctica de WebGL.

1.1. QUÉ SON LOS GRÁFICOS TRIDIMENSIONALES

Los gráficos por computadora surgieron a principios de los años cincuenta en el conocido MIT (Instituto Tecnológico de Massachusetts). En él se conectó por primera vez un osciloscopio a una de sus computadoras, de modo que éste podía controlar directamente la posición del haz catódico de la pantalla. Este sistema

[2] OpenGL ES (OpenGL for Embedded Systems) es una variante simplificada de la API gráfica OpenGL diseñada para dispositivos integrados tales como teléfonos móviles, PDAs y consolas de videojuegos.

[3] OpenGL (Open Graphics Library) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

[4] Un shaders es un procedimiento de sombreado e iluminación que permite al artista/programador especificar el renderizado de un vertex o de un pixel.

[5] Los frameworks son soluciones completas que contemplan herramientas de apoyo a la construcción (ambiente de trabajo o desarrollo) y motores de ejecución.

podía realizar dibujos simples, moviendo el haz de manera que fuese trazando las líneas que lo componían; sin embargo, cuando la complejidad del gráfico aumentaba, aparecía un problema de difícil solución: el parpadeo. Éste surgía porque el punto trazado por el haz desaparecía unas décimas de segundo después de que el haz se hubiese movido, lo que obligaba a redibujar constantemente el gráfico (a este proceso se le denominó refresco). Con figuras de pocas líneas, el dibujo podía ser refrescado a una velocidad suficiente como para que no hubiese problemas, pero al aumentar el número de segmentos, la computadora tardaba más en generar cada imagen, de modo que cuando empezaba a redibujar una línea, la original ya se había borrado totalmente. A este problema había que añadirle que la computadora tenía que dedicar la totalidad de su tiempo de ejecución a mantener el refresco de la imagen, lo que impedía la realización de cálculos simultáneos, provocando un ralentizado en la máquina.

Con el paso del tiempo se ideó un tipo de pantalla que mantenía la luminosidad de un punto, incluso después de que el haz dejase de iluminarlo, usando una especie de condensador entre el fósforo de la pantalla y el tubo. Para borrar la pantalla, simplemente se descargaba ese condensador; este sistema tenía la ventaja de que eliminaba totalmente el refresco de la imagen, pues ésta se mantenía aunque el haz dejase de pasar. Sin embargo, tenía el problema de que no se podía borrar una parte de la pantalla, sino que se debía borrar toda y reescribir la parte común; y por otra parte el bajo contraste de la imagen.

El anterior sistema de generación de imágenes es denominado *vectorial*, pues sus imágenes están compuestas por vectores unidos entre sí. Pronto se hizo evidente que este sistema no era necesariamente práctico, así que se empezó a trabajar en uno mejor; el resultado fue el sistema “Raster Scan”, en este sistema el CPU ya no controla el tubo de rayos catódico, sino que existe una circuitería independiente que realiza todo el trabajo; esta vez, el haz no crea la imagen a partir de segmentos o vectores, sino que lo hace a partir de puntos (pixeles). Para ello, el haz recorre toda la pantalla en líneas horizontales, de izquierda a derecha y

de arriba abajo, este movimiento se realiza con el haz a baja intensidad, si se necesita activar un pixel, se aumenta la intensidad en el momento en que pase encima de él; dado que el haz tiene que recorrer siempre toda la pantalla, el parpadeo de la imagen solo depende de la velocidad del rayo, nunca de la complejidad de la imagen. Hemos mencionado que una circuitería externa genera la imagen y la refresca, por lo tanto la CPU queda totalmente liberado de esta tarea, pudiendo mantener una imagen en pantalla mientras realiza otros cálculos. A todo esto se unió la posibilidad de conseguir una amplia gama de tonos y colores, frente a la gran limitación de los sistemas vectoriales en este aspecto; la supremacía de los *Raster Scan* quedó demostrada al comprobar que todas las computadoras actuales lo usan, sin importar su tamaño y potencia (este sistema es más detallado en la sección 1.2).

Con la mejora de las tecnologías tanto de los CPU's como de los circuitos generadores de imágenes, no solo fue posible hacer gráficos simples en pantalla, sino también la posibilidad de crear imágenes mucho más detalladas y complejas. Los gráficos tridimensionales son generados a partir de los puntos o pixeles y éstos a su vez generan figuras geométricas simples como triángulos o cuadrados, los cuales son la base para poder formar imágenes con volumen.

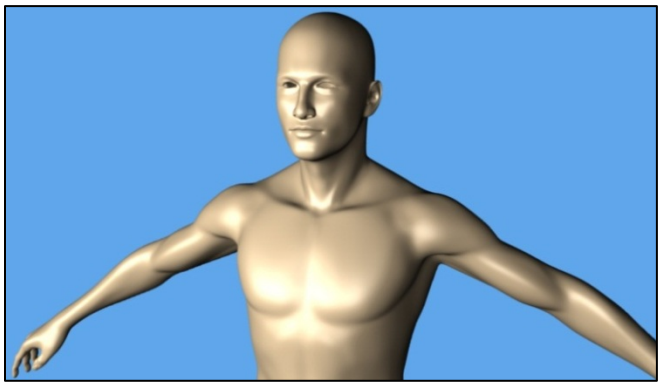


Figura 1.1. Modelo tridimensional.

Los gráficos tridimensionales son generados a partir de cálculos matemáticos para proyectar las figuras geometrías de tal forma que puedan dar la sensación visual en pantalla de anchura, altura y profundidad. Tal y como podemos observar

en la figura 1.1; la cual a pesar de ser una imagen en dos dimensiones al estar impresa en papel, la sensación visual es de una imagen en tres dimensiones, ya que fue creada de tal manera que las figuras geométricas que la conforman sigan un patrón específico logrando dicho efecto.

Una analogía que suele usarse en el mundo del diseño por computadora para describir los gráficos tridimensionales, es la que dice que el 3D en el arte es como la escultura, mientras que el 2D es como la pintura ya que para lograr que un diseño tridimensional quede correctamente, se deben seguir una serie de pasos detallados para poder modelarlo.

1.2. HISTORIA DE LOS GRÁFICOS 3D

Los orígenes de los gráficos en tercera dimensión se encuentran sumamente ligados al concepto de “Diseño Asistido por Computadora” o CAD por sus siglas en inglés, el cual tiene como principal objetivo establecer los principios, técnicas y algoritmos para la generación y manipulación de imágenes mediante una computadora. Dichas imágenes pueden ser de distinta complejidad, desde imágenes en dos dimensiones hasta modelos tridimensionales.

A lo largo de más de cuarenta años de evolución de los componentes electrónicos digitales y dispositivos computacionales, la capacidad de generar imágenes por computadora también se ha incrementado, a tal punto que ya no podemos concebir una computadora, por más sencilla que esta sea, sin alguna capacidad gráfica.

El diseño por computadora es una herramienta eficaz y sencilla que se implementa en diversas áreas, tales como la industria, arte, televisión, gobierno, publicidad, ciencia, ingeniería; con finalidades desde el entretenimiento hasta la capacitación.

El objetivo del presente capítulo es entender cómo es que el Diseño Asistido por Computadora llegó a ser una importante rama de la computación y para esto es necesario hacer un breve recorrido a través del tiempo, presentando su desarrollo histórico hasta llegar a lo que se vive hoy en día en lo referente a imágenes generadas por computadora.

Después de la segunda generación de computadoras (cuando se dejaron de usar los bulbos y transistores), comenzó el nacimiento de la graficación por computadora y como se mencionó anteriormente, fue en la década de los cincuenta cuando se mejoraron los dispositivos de presentación usados por las computadoras.

En el MIT (Instituto Tecnológico de Massachusetts) crearon la primera computadora con un sistema de representación visual, la cual fue llamada "Whirlwind" y contaba con una pantalla de Tubo de Rayos Catódicos (CRT)[6] donde se generaban dibujos sencillos, pero debido a las limitaciones en su hardware[7], se veía afectado el tiempo empleado para realizar cálculos y después presentarlos, así que la pantalla actuaba sólo como un dispositivo de salida y no como una interfaz entre el usuario y la computadora.

A partir de ese momento se inició un desarrollo importante en los monitores de Tubo de Rayos Catódicos, con el fin de obtener un mejor aprovechamiento en la generación de imágenes gráficas de las computadoras de aquella época. Se distinguieron dos tecnologías completamente distintas; la primera que se desarrolló fue el *Direct View Storage Tube* o DVST (Tubo Adaptador de Visión Directa), también conocido como "tubo de almacenamiento" o "tubo de memoria"; en éste, la información gráfica se envía en una sola emisión y se muestra permanentemente en pantalla, siendo la imagen de tipo vectorial, ya que estaba

[6] El Tubo de Rayos Catódicos (CRT del inglés Cathode Ray Tube) es una tecnología que permite visualizar imágenes mediante un haz de luz constante a una pantalla de vidrio recubierta de fósforo y plomo.

[7] El hardware corresponde a todas las partes tangibles de una computadora; entre ellos, sus componentes eléctricos, electrónicos, electromecánicos y mecánicos.

formada por vectores contínuos en línea recta entre dos puntos de la pantalla. Este tipo de presentación se caracterizó por una excelente calidad en sus líneas; se dejo de trabajar con él por numerosos inconvenientes, como fueron, su alto costo de implementación, no permitir imágenes a color y la ineficiencia en el relleno de áreas de dibujo.

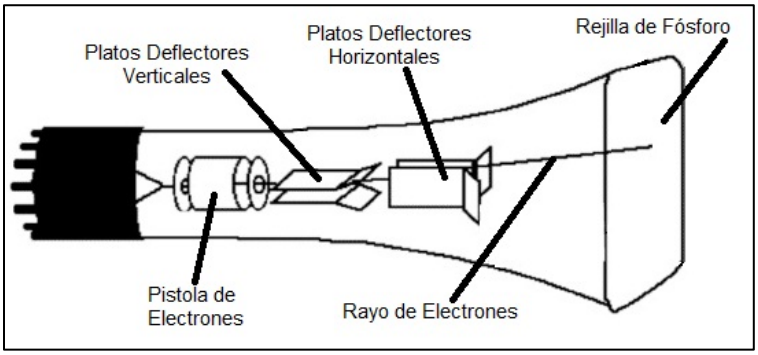


Figura 1.2. Diagrama de un Tubo de Rayos Catódicos.

Para lograr realizar las tareas no permitidas, al trabajar con DVST, se desarrolló una segunda técnica que empleaba los llamados “tubos de refresco” (Refresh Tube). En dichos tubos se generaba una imagen cada determinado periodo de tiempo, que por lo general era sólo una fracción de segundo y este tiempo se encontraba bajo el control del CPU; esto permitió editar partes de un dibujo o realizar secuencias animadas.

Los primeros tubos de refresco aún generaban la imagen usando representaciones vectoriales, por lo que seguían teniendo dificultades para rellenar dibujos, además, como ya se mencionó, presentaban tendencia a parpadear, es decir, producían la sensación visual de discontinuidad en las imágenes. Es por ello que conforme se fue mejorando la tecnología empleada en los tubos de refresco se logró añadir una rejilla de fósforo al final del tubo para formar los puntos o pixeles[8], unido al sistema Raster Scan para evitar parpadeos;

[8] Un píxel o pixel, plural píxeles (acrónimo del inglés picture element, "elemento de imagen") es la menor unidad homogénea en color que forma parte de una imagen digital, ya sea ésta una fotografía, un fotograma de vídeo o un gráfico.

generaron una mejora considerable en la imagen de la pantalla y cuya técnica aún es usada en los monitores y televisiones CRT convencionales.

Empleando este tipo de tubo, la pantalla se divide en miles o millones de pixeles que son o no iluminados en cada barrido de la imagen según la información enviada por el CPU. Con esta técnica, la calidad de la línea depende evidentemente del número de pixeles por unidad de superficie.

Figura 1.3. Union de pixeles que forman una imagen, generada por computadora.

Para inicios de la década de 1960 (cuando los monitores apenas empezaban a formar parte esencial de las computadoras), un alumno del MIT llamado Ivan Edward Sutherland, escribió una tesis que dio el primer paso para el concepto actual de Diseño Asistido por Computadora, dicha tesis fue titulada “A Man-Machine Graphical Communications System” (Un sistema de comunicaciones gráfico humano-máquina) y en ésta se sentaban las bases de los gráficos interactivos por computadora, tal y como los conocemos hoy. Sutherland propuso la idea de utilizar un teclado y un lápiz óptico para seleccionar, situar y dibujar conjuntamente con una imagen presentada en pantalla.

Lo más significativo del trabajo de Sutherland fue el sistema llamado “Sketchpad” el cual usó para la computadora TX-2 que fue el primer programa que permitía la manipulación de objetos gráficos, es decir, describía las relaciones entre las diferentes partes que lo componían, introduciendo lo que hoy conocemos como Programación Orientada a Objetos. Antes de esto, las representaciones

visuales realizadas de un objeto se habían basado en un dibujo y no en el objeto en si mismo. Este sistema causó una gran expectación en las universidades y sus limitaciones estaban presentes más en el hardware empleado que en el concepto como tal. Por todo esto, Sutherland es conocido mundialmente por muchos como el Padre del Diseño Asistido por Computadora.

El año decisivo para los sistemas gráficos fue a finales de la década de 1970. A partir de esas fechas los sistemas gráficos (que hasta entonces habían sido del dominio de los científicos, matemáticos e ingenieros), comenzaron a ser usados en televisión y estudios de animación, produciéndose una gran alza en la venta de estos equipos. A finales de 1979, IBM lanzó su terminal 3279 la cual contaba con la capacidad de mostrar gráficos a color; nueve meses después recibió más de diez mil pedidos, muchos de los cuales iban destinados a personas o corporaciones que usaban por primera vez una computadora para la realización de tareas gráficas.

Este éxito en los sistemas gráficos se puso de manifiesto a través de la creación de asociaciones y la celebración de congresos. La *Association for Computing Machinery* (ACM) creó lo que se denominaría “Grupo de Interés Especial en Gráficos por Computadora y Técnicas Interactivas” (SIGGRAPH por sus siglas en inglés) la cual presenta anualmente conferencias para estudiantes y entusiastas, convirtiéndose así en el foro académico más importante en este tema.

A partir de 1980 se presentó un rápido desarrollo en el entorno de los microprocesadores, consiguiendo un considerable aumento de velocidad y memoria tanto en estaciones de trabajo como en computadoras personales, los precios se volvieron accesibles, incluso para la economía doméstica. Para la década de 1990 los sistemas de automatización de las grandes empresas ya dependían directamente del Diseño por Computadora y se tornaron realmente indispensables para un mercado mucho más competitivo, esto aunado a la comercialización de programas de CAD y de Graficación en general cada vez más

abiertos y sofisticados. Esta evolución hizo posible que la aplicación de técnicas CAD esté limitada solamente por la imaginación del usuario.

Una vez que los CPU y los Procesadores de Gráficos se desarrollaron lo suficiente para crear y soportar gráficos mucho más elaborados, se inició la era de la tercera dimensión y la realidad virtual, todo tipo de animaciones, videojuegos, películas, medios publicitarios, entre otros, han sacado provecho del Diseño por Computadora 3D.

Muchas empresas aprovecharon esta creciente tecnología; las más conocida es sin duda *Silicon Graphics*, la cual dominó el mercado de los gráficos tridimensionales por computadora desde finales de la década de 1980 hasta inicios del 2000; en aquel entonces se especializaba en la fabricación de estaciones de trabajo dedicados al 3D, computadoras de escritorio con todas las capacidades (tanto de hardware como de software) para generar gráficos por computadora sumamente avanzados.

Dichas estaciones de trabajo solían funcionar con un sistema operativo basado en UNIX, además de que *Silicon Graphics* fue la primera en crear un sistema de ficheros tridimensional llamado XFS el cual aún puede ser usado en algunas distribuciones Linux.

Además de lo mencionado, *Silicon Graphics* es la empresa que inicialmente desarrollo *OpenGL*, la especificación libre para crear gráficos 3D y que está muy ligada al tema principal de esta tesis. En un principio dicha API[9] era llamada *IrisGL,* pero debido a problemas de portabilidad y algunas dificultades para programar en él, dificultaron su uso, entonces se decidió mejorarlo y licenciarlo para que los competidores pudieran hacer uso de esta tecnología; este tuvo una amplia aceptación y ayudó para que se convirtiera en un estándar, terminando así con los problemas de portabilidad. Hoy el principal competidor de *OpenGL* es

[9] Interfaz de Programación de Aplicaciones (Application Programming Interface, por sus siglas en inglés) en un conjunto de procedimientos para generar un programa de computadora.

DirectX de Microsoft pero este tiene la desventaja de funcionar sólo en sistemas Windows.

El uso de las librerías de *OpenGL* supuso un gran avance para los gráficos tridimensionales y por otra parte las computadoras fabricadas por *Silicon Graphics* se volvieron muy populares sobre todo en la industria cinematográfica, donde los efectos especiales hechos con computadora comenzaron a jugar un papel importante para la calidad visual de las películas. *Silicon Graphics* era el principal proveedor de estaciones de trabajo para los estudios de cine, un ejemplo de esto es la película “Jurassic Park”, estrenada en 1993, la cual provocó asombro debido a que algunos de los dinosaurios mostrados en la pantalla estaban completamente generados por computadora. Otro ejemplo más reciente es la trilogía de “The Lord of the Rings” de 2001, en donde los diseñadores emplearon algunas computadoras “Indy” de *Silicon Graphics* (Figura 1.4) para generar ciertos efectos especiales de la película.

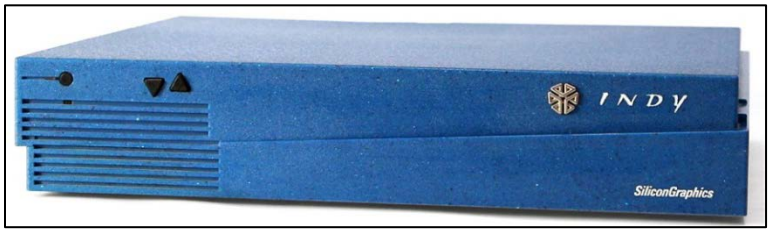


Figura 1.4. La Silicon Graphics “Indy”, introducida en 1993.

Para inicios de la década del 2000, comenzó la decadencia de la compañía *Silicon Graphics* debido al crecimiento y abaratamiento de las computadoras personales y al hecho de que los usuarios ahora podían simplemente comprar una tarjeta gráfica e instalarla en su computadora; por lo que compañías que fabrican chips de video como ATI y Nvidia, y la aparición de programas para manipular gráficos tridimensionales como “Autodesk Maya”, “3ds Max” y “Blender” terminaron por destruir el negocio de *Silicon Graphics* de estaciones gráficas completas. Actualmente esta compañía se dedica sólo a la fabricación de servidores y supercomputadoras.

Además de *Silicon Graphics*, otras empresas desempeñaron un papel importante en la historia de los gráficos tridimensionales. Una de ellas es el estudio de animación *Pixar,* cuya principal aportación fue el desarrollo del programa “RenderMan” para la creación y edición de gráficos tridimensionales, este es muy usado en la industria del entretenimiento y cuyo logro más impactante fue la creación de la película “Toy Story” en el año 1995; el primer largometraje de animación enteramente hecho por computadora. Con todo esto se generó un gran entusiasmo en los desarrolladores interesados por la tercera dimensión.

Como nos hemos dado cuenta, se ha recorrido un largo y difícil camino para que los gráficos de tercera dimensión generados por computadora sean lo que hoy conocemos, desde los simples gráficos vectoriales desplegados en un Tubo de Rayos Catódicos, hasta las complejas imágenes formadas por millones de pixeles; los gráficos tridimensionales se encuentran en constante evolución y en conquista de más plataformas y dispositivos, como es el caso de la internet con el WebGL.

1.3. ANTECEDENTES A LAS APLICACIONES DE TERCERA DIMENSIÓN EN LA WEB

Como antecedente se encuentran varios intentos por conseguir integrar la realidad virtual y la tercera dimensión en los navegadores web, los cuales tuvieron su mayor apogeo e investigación en los años noventa cuando la *World Wide Web* se encontraba en pleno auge. Debido a cierta apatía y poco desarrollo por parte de los fabricantes (Netscape y Microsoft, en aquella época), falta de potencia de hardware en las computadoras y de escaso ancho de banda en las conexiones, todos los intentos fracasaron.

Uno de los primeros y quizás el más conocido intento fue el VRML (Virtual Reality Modeling Language) o Lenguaje para Modelado de Realidad Virtual. Era un formato de archivo que permitía almacenar modelos y texturas en 3D, pensado

para su uso en la Web, de forma que se pudieran asociar objetos de una escena a hipervínculos, creando así cierta interactividad en un modelo 3D. Su estructura estaba regulada por el entonces conocido como “VRML Consortium” y reconocido más tarde como estándar ISO.

La primera versión de VRML apareció a mediados de los años noventa, teniendo su primera especificación en 1995; aunque fue hasta 1997 cuando se presentó la versión VRML97, que es la que alcanzaría su relativa popularidad. Ésta tuvo bastante éxito en el campo educativo y de investigación, aunque no llegó a utilizarse por el público en general.

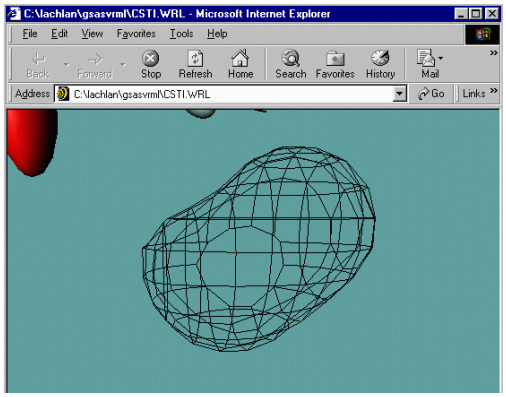


Figura 1.5. VRML en Internet Explorer.

En la época de apogeo de VRML, la mayoría de los usuarios contaban solamente con una conexión básica por módem, lo cual provocaba que fuese necesaria una carga de varios minutos para poder visualizar, en muchos de los casos, un modelo de calidad bastante reducida. Además de esto, *Silicon Graphics* (una de las empresas importantes detrás de VRML), detuvo en 1998 todo nuevo desarrollo sobre esta tecnología. Estos inconvenientes provocaron que VRML prácticamente cayera en el olvido recién comenzado el nuevo siglo**.**

A pesar de esto, a inicios de la década del 2000, se intentó de nuevo con otro formato: el X3D, el cual corrige ciertas carencias del VRML original, como lo es la integración con HTML. Sin embargo, aunque X3D es un formato abierto y estandarizado desde el principio y que contó con el apoyo del W3C (World Wide

Web Consortium), no tuvo un claro futuro ya que padecía el mismo problema que impidió la popularización de VRML: la necesidad de un plugin[10] para poder interpretarlo. Por lo tanto, y debido a la arquitectura de plugins de los navegadores web, esto lo limitaba a ser solo un rectángulo dentro de una página web, como ocurre con el popular *Flash Player*. Sin importar esto, la más reciente revisión del lenguaje HTML (HTML5) incluye en sus especificaciones a X3D pudiéndole sacar provecho y logrando una integración total con la web.

Intentando superar las limitaciones de sus predecesores, en 2009 se empezó el desarrollo de una nueva especificación conocida como “WebGL” el cual cuenta con el apoyo de los principales desarrolladores de navegadores web y como se mencionó anteriormente, tiene por objetivo introducir definitivamente la tercera dimensión a la internet mediante navegadores y aplicaciones completamente preparados para desplegar contenidos tridimensionales usando las librerías y especificaciones de OpenGL ejecutadas directamente en el navegador web. En el siguiente capítulo se hablará un poco sobre las tecnologías actuales que están intentado llevar el 3D a la internet.

[10] Un Plugin es un complemento de una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica.

CAPÍTULO 2

LLEVAR EL 3D A LA WEB

Las tecnologías básicas de la web han estado en constante evolución, añadiendo más características para ser usado por los desarrolladores web. Sin embargo, las tecnologías web básicas siguen sin soporte para objetos multimedia, como el vídeo o el audio, los cuales se basan en plugins propietarios del navegador como *Flash Player*; los plugins de Flash proporcionan soporte para contenido de audio y vídeo, así como vectores gráficos altamente interactivos. Pero el plugin tiene algunas desventajas, tales como fallas o vulnerabilidades a la seguridad, además con los plugins de código cerrado las soluciones dependen en gran medida de la empresa responsable de ellos y sólo es soportado en un conjunto limitado de plataformas.

La constante innovación en la web se basa en una mezcla de competencia y cooperación entre los proveedores de navegadores, cooperando en el desarrollo de estándares abiertos que pueden ser implementados por cualquiera de forma gratuita y que compitan tratando de ofrecer la mejor aplicación de dichos estándares, dando periódicamente actualizaciones a su producto logrando una mejor realización de las implementaciones. Precisamente con HTML5 se pretende cambiar esta dependencia de plugins para mostrar contenido multimedia, pudiendo introducir este contenido directamente en el código de la página.

Definitivamente, estamos viviendo en el tiempo correcto para que los gráficos tridimensionales se empiecen a ver de forma cotidiana en los contenidos de internet, el hardware con el que se cuenta ya no es tan limitado como el que se tenía hace una década en los tiempos del VRML, las conexiones de banda ancha son tan rápidas que se pueden descargar archivos de gran volumen en pocos minutos y tanto los procesadores como los chips de video son capaces de procesar rápidamente una gran cantidad de datos. Todo esto despierta entusiasmo por el futuro de esta tecnología y hace pensar que vale la pena invertir en los proyectos relacionados a la tercera dimensión en la web.

2.1. TECNOLOGÍAS EXISTENTES

El desarrollo de aplicaciones tridimensionales para el internet se está volviendo cada vez más común y los navegadores web de última generación están completamente preparados para soportar los recientes estándares empleados; existen tres tecnologías, las cuales tienen un mayor peso en lo referente al tema: Stage 3D, WebGL y HTML5.

Stage 3D se está desarrollando de forma paralela con Adobe Flash, para ser parte o una característica extra de éste, por lo que cuenta con la desventaja antes mencionada de necesitar de un plugin para funcionar.

En cuanto a WebGL y HTML5, podríamos decir que WebGL es el hermano mayor de HTML5, ya que este último hace uso solamente del lenguaje X3D que está incluido dentro de sus especificaciones; mientras que WebGL además de usar el propio HTML5, también emplea las librerías de OpenGL ES, por lo que es capaz de mostrar gráficos mucho más detallados al lograr un mejor aprovechamiento de la aceleración de la tarjeta gráfica.

2.1.1. SOLUCIONES DE PROPIETARIOS

Una solución que puede traer los gráficos 3D para la web es mediante el uso de dichos plugins o extensiones. Esta solución no es muy factible debido a la necesidad de los reproductores de dichos plugins que requieren ser transferidos y compilados para cada una de la plataformas de destino. Esto hace que sea muy difícil traer el contenido 3D a una amplia gama de plataformas, como lo son los dispositivos móviles.

Un ejemplo notable del enfoque hacia el plugin es el visto en el juego “Quake Live 2”, el cual tiene un modo “multijugador” para los navegadores, usando un plugin que está especializado para los navegadores Internet Explorador, Firefox y

Safari y que se ejecuta en sistemas operativos Windows, Mac y Linux. Por otra parte, el plugin también gestiona el contenido del juego en el sistema local de archivos, por lo tanto, puede lograr rápidos tiempos de carga sin la necesidad de cargar toda la escena desde la web cada vez que se inicia.

Una forma diferente de aprovechar el 3D es emularlo usando funcionalidades 2D. De esta manera los objetos 3D son renderizados por la CPU usando Canvas2D o Adobe Flash. Por ejemplo, Papervision 3D es un motor[11] de tercera dimensión (anterior a Stage 3D), que utiliza el lenguaje Flash (ActionScript) de Renderizado Vectorial para dibujar escenas en 3D, logrando un rendimiento de alrededor de 2000 a 5000 polígonos. Incluso una tarjeta gráfica de diez años de antigüedad, como la Geforce256 (lanzado a finales de 1999) puede lograr un rendimiento de 15 millones de polígonos por segundo.

Como se puede ver en estas mencionadas tecnologías, emulando 3D usando 2D y que el CPU haga la mayoría del trabajo, no es funcional para grandes escenas tridimensionales. Utilizar las capacidades de una GPU (Unidad de Procesamiento Gráfico) dedicada a este trabajo es la clave para lograr gráficos más detallados sobre la web.

2.1.1.1. Adobe Flash

Adobe Flash es una serie de aplicaciones usadas para crear contenido multimedia mediante gráficos vectoriales enfocado principalmente a las páginas web, dicho contenido de audio y video es creado por el programa de diseño Adobe Flash en un archivo con extensión SWF para posteriormente poder ser leído o visualizado en una computadora cliente por medio del plugin de Adobe Flash Player.

[11] El término motor hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de un gráfico.

La historia de Adobe Flash comienza en 1993, cuando un arquitecto llamado Jonathan Gay programó un editor de imágenes al que llamo “Superpaint”, después de esto trabajó en algunas empresas donde conoció gente interesada en los gráficos por computadora, creando así su propia empresa: *FutureWay*, en la que con ayuda de sus colaboradores diseñaron el programa “SmartSketch” con el cual se podían crear dibujos vectoriales y con base en este se derivó un plugin para *Netscape* al que bautizaron como “Future Splash Animator”. Fue hasta el año de 1996 cuando el internet empezó a tener gran auge, cuando la empresa *Macromedia* compró la empresa de Gay y Future Splash Animator pasó a ser “Macromedia Flash 1.0”. Esta tecnología permitió a los desarrolladores de páginas web crear entornos completamente gráficos en las páginas y todo tipo de contenidos interactivos en sus diseños. Fue hasta el año 2006 cuando la empresa *Adobe* compró a Macromedia para tener los derechos de Flash; esto parece irónico ya que Adobe había rechazado el programa de Jonathan Gay “Future Splash Animator” cuando se los ofreció, por considerar que no tendría futuro.

Adobe Flash trabaja de la misma forma que un estudio de animación, manipulando los gráficos vectoriales mediante fotogramas y manejando el código por medio del lenguaje de programación ActionScript. Ha crecido tanto como el mismo internet, es casi imposible navegar por la web sin encontrarse con algún tipo de anuncio, botón o hasta páginas enteras hechas en Flash.

- Stage 3D

Cada vez que navegamos por la web, nos encontramos con la tecnología Flash en variadas aplicaciones, esto se debe a que desde hace tiempo, se han empleado en los sitios web para enriquecer el contenido, elaborando animaciones que resaltan los diseños, es decir, páginas web totalmente multimedia.

Hasta hace un tiempo, sólo se veía a esta tecnología como un conjunto de imágenes que permitían elaborar un contenido interactivo pero en formato 2D. Esto era una desventaja evidente frente a las tecnologías que le hacían competencia directamente. Así fue como empezó a trabajarse en el proyecto "*Stage 3D"*, el cual forma parte del mismo Flash Player, pero agregando la capacidad de mostrar gráficos tridimensionales.

Figura 2.1. Videojuego programado con Stage 3D corriendo en Firefox.

Los intentos por traer el 3D a Flash Player se remontan al año 2005 cuando se creó el primer motor 3D para Flash llamado "Sandy", fue escrito con el lenguaje ActionScript 2.0 que era el lenguaje más reciente para Flash de aquella época. Debido a la naturaleza interpretativa de ActionScript y a la capacidad un tanto limitada de renderizado de gráficos en Flash Player, el 3D de salida generado por Sandy era bastante básico y se limitaba a carteles de escala simple. Sin embargo la liberación del código de Sandy representó el punto de origen para el 3D de tiempo real en Flash y las librerías de código abierto se establecieron como un estándar para futuros motores 3D.

Siguiendo los pasos de Sandy, a finales del 2006 se lanzó "Papervision" como otra librería de código abierto para Flash Player, diferenciándose principalmente porque proporciona un método más sencillo para la creación de los contenidos

tridimensionales y que fue escrito en el lenguaje mejorado de ActionScript 3.0. En este punto muchos desarrolladores ya coincidían en que los aspectos más importantes de cualquier formato web 3D no son cuantos polígonos puede manejar, si no su accesibilidad para los desarrolladores y el usuario.

Fue hasta octubre del año 2011 cuando Adobe liberó la versión 11 de Flash Player, la cual incluyó a Stage 3D en respuesta al creciente uso de WebGL. Aunque Stage 3D tiene ciertas ventajas con respecto a sus antecesores (como la aceleración por hardware), presenta también una gran desventaja, ya que al depender directamente de Adobe Flash, es necesario nuevamente bajar el plugin para poder hacer uso de ella y ejecutarlo en la computadora para visualizar los contenidos, con las desventajas ya mencionadas que esto representa; además también el hecho de que algunos dispositivos portátiles que ejecutan sistemas operativos *iOS* y *Windows Phone 7* no son compatibles con Flash Player.

2.1.2. EL ESTÁNDAR HTML5

HTML5 es la quinta revisión del lenguaje usado para la construcción de las páginas de internet (HyperText Markup Language), la cual presenta una serie de mejoras y nuevas características con respecto a la anterior versión, la cual data de 1999 y que en la actualidad requiere de tecnologías externas para trabajar con contenidos variados como el ya mencionado Flash, Quicktime o Java; haciéndola una tecnología un tanto obsoleta. Así, HTML5 incorpora nuevas etiquetas, las cuales determinan las instrucciones que debe interpretar un navegador web (códigos de fuente, tablas, imágenes, etcétera), al mismo tiempo que elimina aquellas en desuso.

Todo esto implica que los agregados o plugins que se utilizan hoy, se volverán innecesarios, así por ejemplo en HTML5 se tiene el elemento de “lienzo” (canvas) que permite dibujar y animar imágenes en la pantalla sin necesidad de

aplicaciones adicionales; en este sentido, el propio lenguaje incluirá las características que actualmente sólo pueden disfrutarse teniendo instalado Flash. Aunque aún está en fase de desarrollo y pruebas, ya está empezando a ser usada por muchos desarrolladores web y prueba de esto es que las grandes empresas del internet han empezado a migrar sus portales a esta tecnología y a mediados de 2011 algunas de estas grandes compañías de internet mantuvieron abiertas durante 24 horas sus páginas principales hechas completamente en este lenguaje a fin de poner a prueba la usabilidad por parte de los clientes. Fue de esperarse que no hubiese problema ya que todos los navegadores recientes tienen soporte para HTML5.

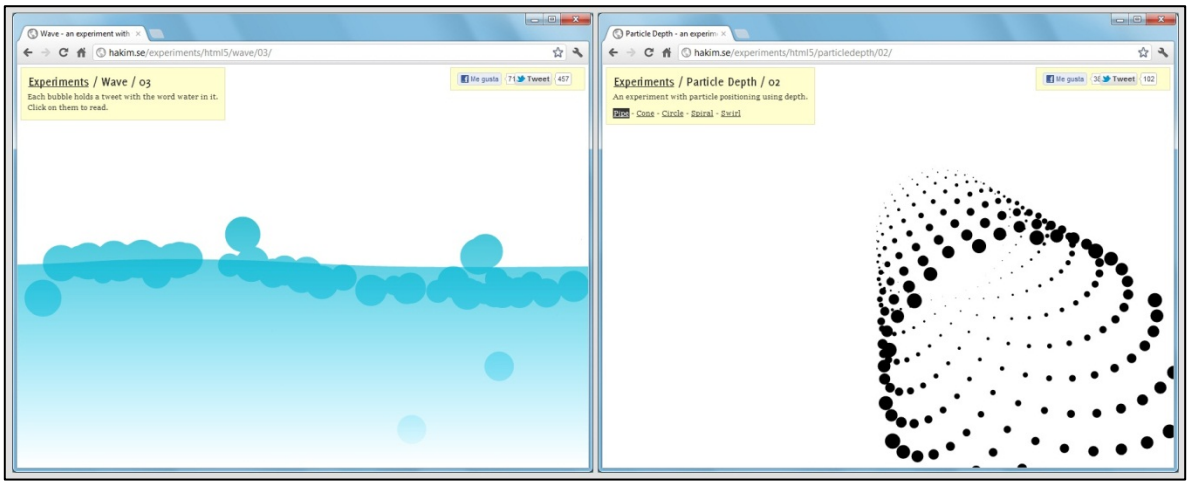


Figura 2.2. Animaciones creadas con HTML5 ejecutándose en Chrome.

Las características mencionadas son sólo la punta del iceberg. HTML5 promete traer numerosas modificaciones a la manera en que se trabaja en Internet. Aunque aún falta un largo camino para que esta revisión se implemente completamente, algunas de sus funciones ya se encuentran entre nosotros.

Los cimientos de HTML5 se empezaron a formar en 2004 cuando representantes del “Semantic Web Community”, del W3C[12] (World Wide Web Consortium) y los mayores desarrolladores de navegadores web se reunieron en

[12] El World Wide Web Consortium, abreviado W3C, es un consorcio internacional que produce recomendaciones para la World Wide Web.

San José California para discutir sobre las nuevas aplicaciones web. En esta reunión no lograron llegar a acuerdos importantes respecto a estándares, pero posteriormente los más grandes desarrolladores de navegadores web (Apple, Fundación Mozilla y Opera Software), formaron una comunidad llamada WHATWG (Web Hypertext Application Technology Working Group) con el fin de resolver problemas emergentes de HTML y empezar a trabajar en la quinta revisión de este lenguaje. Mientras tanto la W3C siguió adelante con el desarrollo de XHTML (eXtensible Hypertext Markup Language) que era el lenguaje pensado para sustituir a HTML. Cinco años después la W3C decidió abandonar XHTML y trabajar de forma conjunta con WHATWG para el desarrollo de HTML5.

El estándar HTML5, ha sido desarrollado para incorporar características avanzadas como multimedia en la web en forma de un estándar abierto para resolver la dependencia de los plugins y los problemas que están causando. HTML5 incluye funciones para la interacción del usuario como arrastrar y soltar, video, audio y un lienzo genérico se puede utilizar para dibujar sobre él. Sin embargo, el estándar solo define el marcado y la secuencia de comandos de parte de estas características. En el caso del soporte de video, permite la inserción de video en un documento HTML y usar ECMAScript para interactuar con él, como una búsqueda. Un elemento importante de HTML5 es el ya mencionado “lienzo” o *canvas* del cual se deriva el llamado *Canvas3D* con el cual es posible generar gráficos tridimensionales simulados sin hacer uso de la aceleración por hardware.

- CANVAS HTML5

Una de las otras nuevas características de HTML5 es el elemento “canvas” que define un lienzo base de dibujo. Canvas fue introducido originalmente por Apple en su navegador Safari y posteriormente se estandarizó como parte de HTML5 por el antes mencionado WHATWG. Múltiples Interfaces de Programación de Aplicaciones (API’s) se pueden utilizar para dibujar en canvas. Estos pueden incluir las API’s estandarizadas o propietarias que proporcionen una extensión del explorador. La interfaz DOM[13] (Modelo de Objetos del Documento) del elemento canvas define una función simple que inicializa un contexto de dibujo en función del argumento dado, permitiendo generar gráficos estáticos o animaciones.

Uno de esos contextos es Canvas2D. Se trata de un API de dibujo 2D que proporciona métodos para dibujar rectángulos, círculos o dibujos de cualquier forma, llenando los objetos con colores diferentes o degradados y aplicar estos efectos en ellos. Esta API se parece mucho a especificaciones de dibujo en plataforma 2D como GDI+ en Windows, Cairo en Linux o Quartz 2D en Mac OS X.

Cuando se usa el elemento canvas en una página web, se crea un área rectangular. Por default este rectángulo tiene un área de 300 pixeles de largo por 150 pixeles de alto, aunque esta especificación se puede cambiar sin problema dependiendo de las necesidades. Una vez que se añade canvas a la página, se puede usar JavaScript para manipular cualquier aspecto de ésta; pudiendo agregar gráficos, líneas y texto; se puede dibujar en ella e incluso agregar animaciones avanzadas. Canvas juega un papel importante cuando se trabaja con gráficos tridimensionales usando HTML5 3D porque con él se define el área de trabajo para dicha animación.

[13] DOM (Document Object Model) es esencialmente una interfaz de programación de aplicaciones (API) que proporciona un conjunto estándar de objetos para representar documentos HTML y XML

2.1.3. WebGL

Desde la introducción de canvas HTML5 y su contexto en 2D, los desarrolladores han estado experimentando con aplicaciones 3D capaces de elaborar complejas escenas tridimensionales como sea posible con API's específicas de 3D. Dos de los prototipos más notables fueron Canvas3D y Opera-3D. Opera-3D fue introducido en el blog de desarrolladores de Opera y ofreció una API de dibujo centrada en torno a los modelos 3D que pueden ser texturizados.

Canvas3D se desarrolló primeramente como una extensión de Mozilla y no tanto para crear una nueva API de dibujo, inicialmente su objetivo es ser una "envoltura" entre ECMAScript y OpenGL ES.

El proceso de estandarización de WebGL ha adoptado el mismo enfoque que el prototipo Canvas3D, utilizando una API que se ajusta estrechamente a la API OpenGL ES 2.0. Es claro que un contexto 3D para el elemento canvas solamente tiene sentido cuando se pueden aprovechar las modernas tarjetas gráficas. Esta es la razón por la que OpenGL ES 2.0 fue elegido como la base para WebGL.

La diferencia más importante con sus antecesores es que se trata de tan sólo una especificación y no de una implementación. Esto significa que el organismo *Khronos Group* (encargado actual del desarrollo de OpenGL), es quien decide cómo funciona y luego los propios fabricantes son quienes realizan la implementación en sus navegadores de forma nativa, sin requerir la instalación de un plugin, evitando el mencionado efecto rectángulo dentro de una página web.

La primera versión de WebGL (1.0) fue liberada el 6 de marzo de 2011 y actualmente está siendo activamente implementado por los navegadores de Apple, Google, Opera y Mozilla. Siendo únicamente Microsoft con su Internet Explorer la única empresa que no se encuentra desarrollando el WebGL en su navegador; teniendo como principal argumento el supuesto hecho de que WebGL tiene graves fallos de seguridad, aunque es sabido por muchos que la principal

razón es porque esta tecnología emplea OpenGL, que es el competidor directo de código abierto para DirectX de Microsoft. Lamentablemente al ser Internet Explorer uno de los navegadores más ampliamente usado a nivel mundial, provocará que aproximadamente el 35% de los usuarios de Internet no tengan acceso a esta tecnología hasta dentro de varios años.

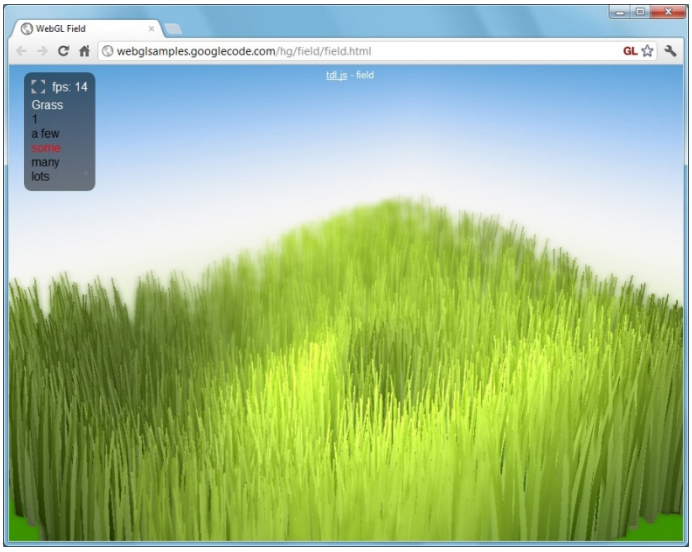


Figura 2.3. Animación 3D en Google Chrome usando WebGL.

En cuanto a la compatibilidad entre distintas plataformas, WebGL está pensado para ser utilizado tanto en navegadores de escritorio como en dispositivos móviles. Para ello parte de la especificación ligera de OpenGL: OpenGL ES 2.0 que ya se implementa con éxito en dispositivos portátiles actuales como el iPhone y teléfonos inteligentes con el sistema operativo Android. Esto en principio amplía la compatibilidad a más plataformas, provoca que desaparezcan dificultades de portabilidad y que los programadores se puedan iniciar en OpenGL más rápido.

2.2. Comparación de las tecnologías web 3D

Como se puede apreciar, Stage 3D, HTML5 y WebGL presentan cualidades muy propias, por lo que el siguiente cuadro comparativo resume las características más sobresalientes de estas tecnologías.

	Stage 3D	HTML5	WebGL
Desarrollador	Adobe Systems	W3C	Khronos Group
Sistema operativo	Windows, Mac OS X y Linux	Múltiple	Múltiple
Navegadores compatibles	Google Chrome, Firefox, Opera, Internet Explorer y Safari	Google Chrome 1.0, Firefox 4.0, Opera 10.5, IE 9 y Safari 5.1	Google Chrome 9.0, Firefox 4.0, Opera 12 y Safari 5.1
Última versión estable	11.0.1.152 (Octubre de 2011)	Revisión 5	1.0 (Marzo de 2011)
Aceleración por hardware	Si	No	Si
Funciona mediante	Plugin	De forma nativa en el navegador	De forma nativa en el navegador
API gráfica	DirectX 9 (Windows) y OpenGL 1.3(Linux y Mac)	No Aplica	OpenGL ES 2.0

CAPÍTULO 3

ESTUDIO DE LAS API'S PARA EL DESARROLLO DE APLICACIONES GRÁFICAS EN 3D

En el capítulo anterior conocimos a los principales exponentes en lo referente a aplicaciones para llevar los contenidos tridimensionales al internet, se escribió acerca de Stage 3D, HTML5 y WebGL; leímos de manera general como trabajan estas tecnologías y cómo están intentando posicionarse como las aplicaciones de referencia para el futuro del 3D en la web.

Ya se ha dado una introducción de lo que son los gráficos tridimensionales y de la forma general en que éstos son representados; por lo que en el presente capítulo se pretende ofrecer una visión más detallada de cómo trabajan dichos objetos tridimensionales, de tal manera que conceptos básicos en 3D, como son, las primitivas, vértices, texturas, proyección, rasterización, etcétera serán expuestos y de qué forma éstos convergen en el proceso de dibujado de los objetos en la pantalla.

Actualmente dos API's (Interfaz de Programación de Aplicaciones) gráficas son las que predominan en la industria: OpenGL y DirectX, éstas son las que proporcionan a los programas las librerías necesarias para implementar funcionalidades específicas mediante una interfaz, con la cual se le facilite al programador acceder a dicha función para explotarla. Por esto es importante hablar de cada uno de ellos, cómo funcionan, las ventajas y desventajas que se puedan presentar en ambos para así posteriormente poder hacer una comparación adecuada.

Antes de comenzar a hablar más detalladamente de los gráficos tridimensionales también es necesario entender a lo que se hace referencia cuando se menciona el término "motor gráfico"; una forma simple de explicarlo es comparándolo con el motor de un automóvil, el cual es el encargado de que el auto se mueva; cuando se gira la llave de ignición, el motor es encendido y posteriormente se pisa el acelerador para que el auto se ponga en movimiento. El conductor no necesita saber con exactitud lo que está pasando en el interior del motor, sólo le interesa manejar el vehículo; este mismo concepto se aplica para un

motor gráfico cuando da la orden de inicializar el adaptador gráfico para recibir órdenes, en el momento que se manda alguna información sobre un modelo 3D el motor debe desplegarlo en pantalla de la forma en que se le ordenó; el paso anterior equivaldría a pisar el acelerador del motor.

Así por ejemplo, se puede tomar el motor gráfico de un videojuego y con él crear otro videojuego o alguna otra aplicación 3D; de forma más técnica, se puede decir que el juego o la aplicación son todos los modelos, los sonidos, las animaciones, los escenarios, etcétera; mientras que el motor son todas las especificaciones tecnológicas que no se ven en dicha aplicación, como es el renderizado[14], la manipulación de los objetos 3D, los métodos de iluminación, el sombreado, los métodos de reflejos, rasterización[15], el manejo de la memoria, texturas y materiales, entre otros aspectos.

Así, internamente un motor gráfico hace uso de una API gráfica para acceder a las funcionalidades que nos proporciona el hardware de video. Ya que WebGL está basado en el API de OpenGL, es necesario explicar el funcionamiento interno de este enlace 3D y la forma en que se dibuja el contenido de la ventana en el navegador, por lo que para poder hablar del funcionamiento de WebGL se necesita repasar primero el funcionamiento de OpenGL y sus principios básicos.

[14] Proceso mediante el cual la computadora genera una imagen 2D con base en un modelo 3D (véase la sección 3.1).

[15] Proceso por el cual las imágenes vectoriales son convertidas a pixeles (véase la sección 3.1).

3.1. GRÁFICOS 3D

En el punto 1.1 se explicó de forma general lo que son los gráficos tridimensionales, pero para seguir con el tema de forma correcta es necesario conocer algunos conceptos técnicos, usados frecuentemente en los temas relativos al 3D.

Los elementos gráficos en 3D dependen de un concepto llamado vértices, que se generan al definir los objetos que se quieran dibujar. Un vértice es un punto único en un espacio tridimensional que se expresa por tres componentes vectoriales. Cada componente de este vector corresponde a un eje en el sistema de coordenadas, el sistema de coordenadas utilizado es un sistema ortogonal de coordenadas.

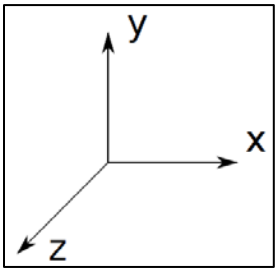


Figura 3.1. Sistema tridimensional de coordenadas.

Primitivas

Los vértices sin embargo, son sólo puntos en el espacio tridimensional y no se pueden dibujar por ellos mismos; pero lo que sí se puede dibujar en los modelos de los gráficos tridimensionales se les llama "primitivas".

Las primitivas son objetos geométricos básicos, creados a través de parámetros; dependiendo del tipo de primitiva se tienen parámetros para configurar. Existen dos grupos de primitivas: comunes y extendidas, las cuales son similares y sólo varían con algunos detalles extras.

Generalmente se tienen parámetros que controlan el tamaño del objeto, otros que regulan algún detalle (como biselado, o redondeado en esquinas) y otros grupos que determinan la segmentación que tiene el objeto, o sea en cuantos pasos está construido, esto influye en el comportamiento al realizar modificaciones.

-Puntos:

A pesar de que un vértice de por sí es un único punto en el espacio, no se considera una primitiva por sí mismo; un punto sin embargo, es una primitiva cuya posición es definida por un solo vértice. Los puntos pueden tener propiedades adicionales, como el color o el tamaño, los cuales controlan la forma en que el punto es dibujado en la pantalla.

-Líneas:

Otra primitiva son las líneas. Dos vértices son usados para definir los puntos de los extremos de un segmento de línea, una línea puede tener una anchura específica y se puede dibujar como una línea de puntos.

-Polígonos:

Cuando una primitiva está compuesta por más de dos vértices se dice que es un polígono. En este caso, los vértices definen las esquinas de una superficie convexa; esto significa que los bordes de una superficie no pueden cruzarse y no se puede cortar.

Cuando se utilizan más de tres vértices para definir una superficie, es claro que la superficie no tiene que ser plana, puede tomar formas arbitrarias; sin embargo en estos casos no se garantiza que la superficie se pueda renderizar correctamente. Para evitar este problema, lo que más se utiliza son los triángulos construidos a partir de tres vértices, ya que éstos siempre son planos. El uso de

cuadrados que constan de cuatro vértices en un arreglo plano también suele ser común, como se observa en la figura 3.2.

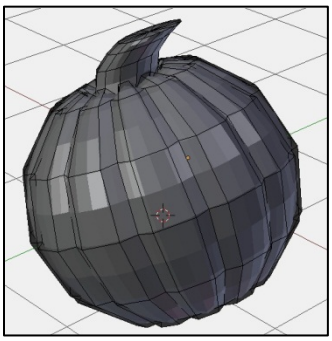


Figura 3.2. Diseño 3D formado por polígonos.

Atributos de los vértices

Además de las coordenadas que definen la posición del vértice en el espacio, también hay otros atributos que pueden ser definidos para la superficie. A continuación se explica lo que es el color y los atributos de coordenadas de textura, que son sólo algunos de los posibles atributos de los vértices.

-Color

A cada superficie se le puede asignar un valor de color, el color está representado por tres o cuatro componentes del vector que contiene los valores de los canales de color básicos: rojo, verde y azul y posiblemente un valor de transparencia si el vértice debe ser transparente.

Cuando se construye una primitiva usando dos o más vértices (como un segmento de línea o un triángulo), los valores de color del vértice pueden ser interpolados en la superficie de la forma resultante.

-Normales

Cada vértice también puede tener lo que se llama una normal, este es un vector de tres componentes y define la dirección en la que estará la cara del vértice. Estas normales se utilizan en combinación con la luz vectorial para iluminar la superficie del objeto definido por el vértice.

Cuando la normal está mirando hacia la luz, la superficie resultante será pintada o coloreada de color blanco o de cualquier otro color, dependiendo del tono que se le haya asignado a la luz; por otro lado, cuando la normal está de espaldas a la luz no se ilumina y por lo tanto ese lado es pintado en color negro. Similar al atributo de color, las normales son interpoladas en toda la superficie.

-Coordenadas de textura

Cuando se trabaja con texturas en un modelo, dichas texturas son asignadas por coordenadas para posteriormente poder ser usadas y ser proporcionadas para cada vértice; las coordenadas definen la posición de la textura en ese vértice. A continuación se da una explicación más detallada de cómo funciona la asignación de texturas.

Texturizado

Como hemos mencionado, cada vértice puede tener un valor de color asignado, estos valores de color se interpolan en toda la superficie que se construye con tres o más vértices. Por ejemplo se puede dibujar una hoja de papel blanco con letras negras en ella y se puede hacer uso de triángulos individuales para modelar cada una de las letras, el uso de este método sería teóricamente posible, pero el problema es que requiere de una gran cantidad de vértices y datos de triángulos; sería mucho más factible dibujar un solo cuadrante conformado por cuatro vértices

y hacer el dibujo de la hoja de papel en ella. Así es exactamente como funciona el mapeo de textura.

La esquina inferior izquierda de la imagen correspondería a la coordenada de textura (0,1); mientras que la esquina superior derecha de la imagen sería la coordenada de textura (1,0). En el ejemplo de la figura 3.3, un sencillo triángulo se muestra con las coordenadas de textura de sus vértices, los cuales definen la parte de la imagen que se muestra en la superficie del triángulo resultante, no es necesario que la dimensión del triángulo corresponda con el tamaño de la textura que se establece en la parte superior de la misma. En tal caso la textura es sesgada e iterada usando al vecino más cercano que encaje, una interpolación puede ser usada por el programador.

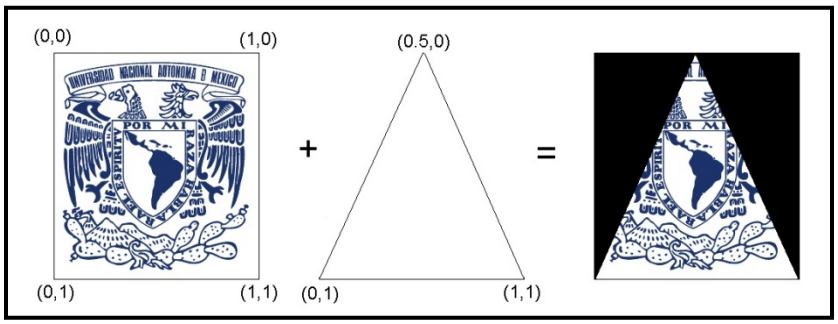


Figura 3.3. Ejemplo de mapeado de textura.

Proyección

Desplegar objetos o escenas tridimensionales en superficies de dos dimensiones como si fuese una hoja de papel o una pantalla de televisión no es realmente un problema, existen diferentes soluciones que permiten alcanzar estos objetivos.

Hay básicamente dos tipos de proyecciones: proyección de perspectiva y proyección paralela, esta última también llamada proyección ortogonal; la diferencia entre estos dos tipos, es la forma en que igualan o calibran el tamaño de los objetos proyectados dependiendo de la distancia desde el punto de vista.

Usando proyección paralela (como se muestra en la figura 3.4B), nos damos cuenta que las proyecciones de los objetos son calibradas del mismo tamaño sin importar la distancia del punto de vista; este tipo de proyección es usado en técnicas de dibujo proporcionadas por los diferentes lados de un objeto, por ejemplo el lado de enfrente o la cara superior. La proyección paralela se logra mediante la construcción de una línea a través del objeto que es normal al punto de vista, el punto en el que se cruza la línea de la ventana es la proyección del objeto original.

Con la proyección de perspectiva (que se puede ver en la figura 3.4A), los objetos del mismo tamaño que se encuentran más lejos a la vista tienen una proyección más pequeña que los objetos que están más cerca de la ventana de vista. Este tipo de proyección es usualmente la elegida cuando se hace uso de aplicaciones 3D interactivas, como los juegos de video; en este modo de proyección la intersección entre la línea y la ventana es la posición de los objetos proyectados.

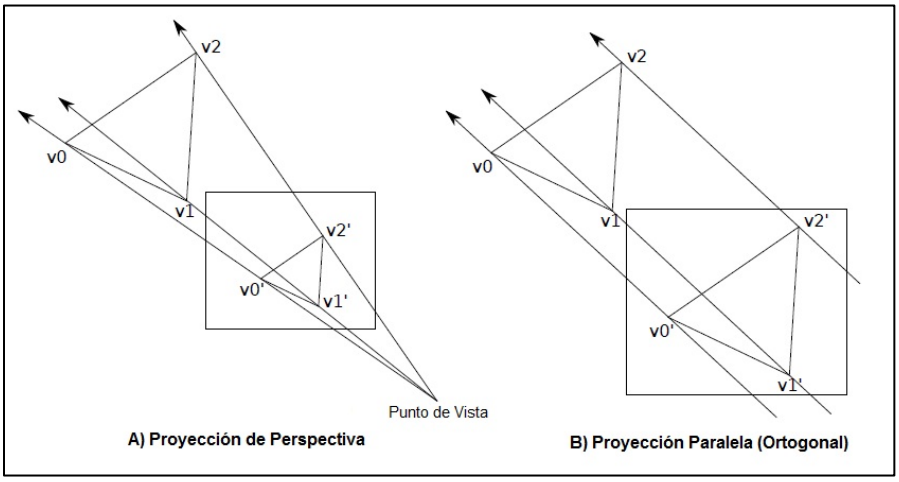


Figura 3.4. Tipos de proyección.

Rasterización

Después de que cada vértice es proyectado hacia el punto de vista, los triángulos definidos por los vértices deben ser convertidos a pixeles reales que

puedan ser visualizados en pantalla, a este proceso se le llama *rasterizado*. En este proceso son evaluadas todas las primitivas del punto de vista y son convertidas en pixeles que pueden ser mostrados al usuario; asi mismo para lograr efectos como superficies semitransparentes, los pasos de rasterización también deben realizarse para las superficies que están detrás del área visible; una función de mezcla puede ser utilizada para interpolarse entre las primitivas del frente y la parte trasera.

Shaders

Se llama Shaders a las técnicas y programas usados para lograr un efecto de sombreado en el diseño tridimensional, los cuales pueden compilarse por separado. Los procesamientos de vértices y pixeles se hacen en los Shader de vértices (Vertex Shader) y de fragmentos (Fragment Shader) respectivamente.

El uso de Shaders en los modelos tridimensionales es relativamente reciente, ya que anteriormente el proceso de sombreado lo realizaba el procesador y no se contaba con la capacidad de compilarlos por separado; fue alrededor del año 2001 cuando las primeras tarjetas gráficas incluyeron la capacidad de modelar el sombreado, dando así la posibilidad de crear efectos especiales más detallados.

Renderización

La renderización es un término usado para referirse al proceso en el que se genera una imagen en dos dimensiones con base en un diseño tridimensional, así, el renderizado es el proceso final que realiza el motor gráfico para mostrar el resultado en la pantalla. En la figura 3.5 se puede observar el mismo diseño de la figura 3.2 pero ya renderizado.

Figura 3.5. Diseño 3D renderizado.

El proceso de renderizado suele ser el que más recursos de memoria y procesador consume en una computadora ya que en él se realizan una gran cantidad de cálculos matemáticos con el fin de "interpretar" todos los procesos antes definidos en el modelo tridimensional. Por ejemplo, en una película de animación en 3D como lo es "Toy Story", el proceso de renderizado puede llevarse varios días en completarse; para hacer este trabajo se emplean los llamados "Render Farm" (Granja de Renderizado) el cual es un conjunto de computadoras repartiendo el trabajo de renderizado, tomando en cuenta que el proceso para crear un solo *frame*[16] requiere aproximadamente seis horas y cada segundo está conformado por 24 frames.

Actualmente estos tiempos de trabajo y de carga del sistema se han reducido considerablemente, gracias a que el hardware gráfico ha evolucionado para ser más flexible y poderoso, siendo capaz de cargar cálculos más complejos de la CPU al procesador gráfico. Incluso los procesadores de video más recientes pueden ser usados para cálculos de proceso general que no tienen relación alguna con los gráficos, como procesos físicos o de criptografía.

En las siguientes secciones se hablará de los principales API's gráficos: OpenGL y DirectX; se mencionará un poco acerca de sus orígenes y la forma en que cada uno de ellos enfoca los principios de los gráficos tridimensionales y los implementa

[16] Término en inglés usado para referirse a un fotograma o cuadro particular dentro de una sucesión de imágenes.

en su lenguaje para programar aplicaciones multimedia. Primero se hablará de DirectX y posteriormente de OpenGL que es el que tiene mayor relevancia para el tema de la presente tesis.

3.1.1. API GRÁFICO DIRECTX

Microsoft DirectX es una colección de Interfaces de Programación de Aplicaciones (API's) integrada a los sistemas operativos Microsoft Windows. Este conjunto de API's proporciona una plataforma de desarrollo de aplicaciones multimedia estándar para Windows, permitiéndoles a los programadores de software acceder al hardware especializado sin tener que escribir código específico para cada tipo de ellos.

La primera versión de DirectX se introdujo en Windows 95 para dar respuesta a los problemas que se presentaban en los videojuegos para el sistema operativo MS-DOS; los cuales al ser diseñados se debían programar para muchas tarjetas gráficas, lo que reducía la compatibilidad y complicaba la instalación. DirectX se desarrolló para garantizar a los programas basados en Windows un gran rendimiento en el acceso a hardware, mediante una capa de abstracción entre el software y los distintos dispositivos, desde entonces DirectX ha evolucionado agregando funciones y mejoras, la última versión estable de DirectX es la 11.0.

Cuando una aplicación o un juego son escritos para DirectX, el programador no tiene que preocuparse exactamente por cuál tarjeta de sonido o cuál adaptador gráfico tiene el usuario final en su computadora, ya que DirectX se encarga de eso y es el responsable de muchas funciones multimedia, incluyendo renderización, reproducción de video, interfaces para dispositivos de entrada, gestión de redes, entre otras cosas.

Esto es una ventaja porque además de brindar soporte para la parte de gráficos y de 3D, también posee todas las herramientas para construir aplicaciones completas de alto nivel, de una manera en la que el hardware no es una limitación,

sino que el programador sólo debe conocer el API y éste es el que se encarga de saber cómo realmente funcionan los distintos tipos de dispositivos.

Lo anterior es logrado mediante varios API's que componen a DirectX, los cuales realizan funciones específicas, estos son principalmente: DirectDraw (Administra la memoria de video), Direct3D (se encarga de los servicios de gráficos tridimensionales), DirectSound (proporciona utilidades de sonido), DirectInput (Permite el acceso a los dispositivos de entrada). El correcto funcionamiento de este conjunto de componentes comúnmente se puede comprobar mediante la herramienta de diagnóstico de DirectX usando el comando *dxdiag* en Windows (figura 3.6).

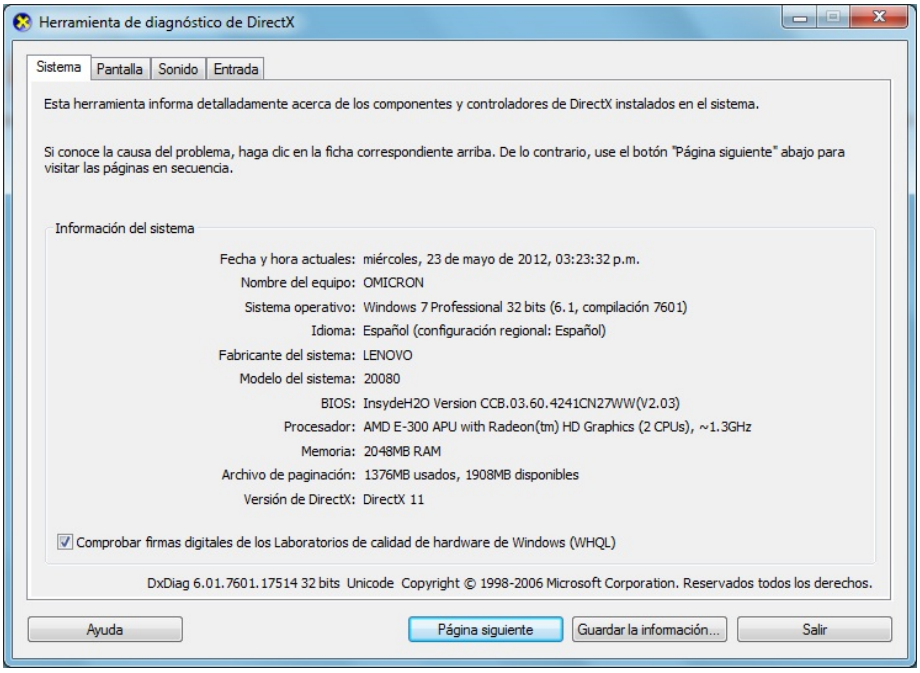


Figura 3.6. Herramienta de diagnóstico de DirectX.

Una gran desventaja de DirectX es que no es portable; como se mencionó, una aplicación programada con DirectX está condenada a trabajar solamente en Windows, lo cual no es nada deseable, a menos que se quiera que el único mercado al que va dirigido la aplicación sea a personas con una computadora con el sistema operativo de Microsoft. A pesar de ser desarrollado sólo para el sistema operativo Windows, existe una empresa llamada *TransGaming Technologies,* la

cual actualmente se encuentra desarrollando una aplicación de pago conocida como "Cedega", esta aplicación funciona como un intermediario con Wine[17] y tiene el propósito de brindar compatibilidad completa a videojuegos escritos para Windows y DirectX en sistemas operativos Linux.

En general DirectX ha estado ganando mucha popularidad en los últimos años entre los programadores de videojuegos, en especial desde que Microsoft comenzó a mejorar la parte de Direct3D haciéndola más sencilla y eficiente, por lo que se ha vuelto una seria competencia para OpenGL entre los estándares gráficos de gran aceptación.

- Estructura de DirectX

Cuando se habla del funcionamiento interno de una API gráfica es bastante recurrente el uso de la palabra "Pipeline" cuya traducción literal del inglés es *tubería*, pero en lo que concierne a los gráficos tridimensionales se refiere al método de renderización que hace uso de la aceleración gráfica de la tarjeta de video para mejorar el resultado final; algunos libros sobre el tema se refieren al Pipeline como "Tubería de Renderizado". Ambas API's gráficas: OpenGL y DirectX, hacen uso de modelos de Pipeline pero varían en la forma de implementarlo.

Para implementar el Pipeline gráfico, DirectX sigue una serie de pasos específicos antes de poder mostrar el diseño final al usuario, estos pasos se encuentra detallada en la figura 3.7 mediante una sencilla máquina de estados.

[17] *Wine* es una implementación usada en sistemas operativos Linux para poder ejecutar algunos programas diseñados para funcionar solamente en Windows.

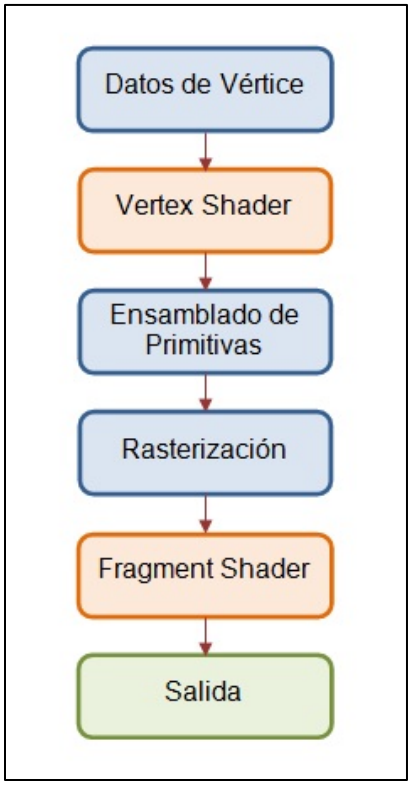


Figura 3.7. Pipeline gráfico en DirectX.

El primer paso del Pipeline de DirectX consiste en obtener los datos de las primitivas, como lo son las líneas y los puntos para posteriormente transformarlos e iluminarlos individualmente, realizando cálculos geométricos, una vez hecho esto se pasa al proceso de rasterizado, en el cual las figuras geométricas obtenidas son trasformadas a pixeles y finalmente se realiza el proceso de efecto y sombreado para los pixeles resultantes. En el apartado 3.2 del presente trabajo se hablará de la estructura general del Pipeline de OpenGL, el cual presenta cierta similitud con el antes mostrado Pipeline de DirectX.

3.1.2. API GRÁFICO OPENGL

Como se mencionó en el capítulo 1, OpenGL es una API de código abierto que fue en un principio introducido por la compañía *Silicon Graphics* e inicialmente fue llamado *IrisGL*, fue hasta el año 1992 cuando OpenGL formalmente se

estandarizadó y empezó a ser administrado por el "OpenGL Architecture Review Board" (ARB), desde entonces ha tenido varias revisiones para incorporarle más funcionalidades y para aprovechar de mejor manera los procesadores gráficos; como los primeros hardwares gráficos eran muy limitados en sus capacidades, todavía la aceleración, la rasterización de primitivas y la interpolación de color requerían de mucho trabajo hecho por el CPU, pero con la introducción de OpenGL se pudo obtener una mejora considerable en el uso de las tarjetas gráficas.

En el año 2006 OpenGL cambió de administración hacia el consorcio conocido como "Khronos Group" el cual está conformado principalmente por compañías como Apple, AMD, Google, Intel, Mozilla, Nvidia, Sony, Oracle entre otras, las cuales son las principales encargadas de su desarrollo. La versión más reciente de OpenGL es la 4.2.

OpenGL es el competidor directo de la API DirectX de Microsoft, la cual como ya mencionamos, sólo es compatible con los productos de esta compañía como lo son los sistemas operativos Windows o la consola de videojuegos Xbox. En cambio, una de las características más destacables de OpenGL es el gran número de plataformas con la que es compatible; a diferencia de su contraparte de Microsoft, OpenGL tiene la ventaja de ser soportado por el *Khronos Group* por lo que se puede desarrollar y ejecutar en una gran variedad de sistemas operativos como Windows, Mac OS X o Linux, en un amplio número de dispositivos portátiles como el iPhone, teléfonos inteligentes y tablets con Android e incluso en consolas de videojuegos como la PlayStation 3.

Otra gran ventaja que ofrece OpenGL es que su arquitectura fue pensada para poder evolucionar conforme avanza el tiempo, de modo que pueda adaptarse con facilidad a las novedades que van surgiendo en la industria y para que los programadores no encuentren problemas a la hora de querer desarrollar un proyecto.

En general, desde su aparición, OpenGL ha sido una aplicación de programación bastante estable y confiable; quizás la razón por la que es más conocido es porque fue la primera en ofrecer la posibilidad de desarrollar aplicaciones 3D; fue precisamente con OpenGL que se hicieron los primeros juegos 3D como "Wolfenstain" y el popular "Doom", pero lo importante es que aún es muy utilizado en aplicaciones y juegos modernos como los de PlayStation 3.

3.2. Estructura de OpenGL

En OpenGL también es usado el término *Pipeline* para definir el proceso de renderizado que tiene que llevar a cabo la tarjeta gráfica. Al igual que en DirectX, el funcionamiento básico de OpenGL consiste en convertir a pixeles las primitivas como líneas o polígonos, este proceso lo hace una Pipeline Gráfica conocida como "Máquina de estados de OpenGL". Para diseñar en OpenGL se requiere que el programador conozca bien los pasos que sigue el Pipeline gráfico.

OpenGL usa dos tipos de Pipeline: de Función Fija (Fixed-Function Pipeline) y Programable (Programmable Pipeline). Algo notable que se incorporó en las últimas versiones de OpenGL fue el cambio de uso de Pipeline de Función Fija a Pipeline Programable, debido a que este último presenta ventajas considerables para el acabado final del diseño, al hacer uso de Shaders en el procesado de las primitivas.

Las diferencias básicas entre estos dos serán explicadas a continuación, junto a una vista a los shaders y como estos conceptos se relacionan con OpenGL. La figura 3.8 da una visión simplificada de las diferencias entre Pipeline de Función Fija y Pipeline Programable, con cada uno de sus componentes explicados en su respectiva sección.

- Pipeline de Función Fija

Las tempranas generaciones de hardware gráfico que soportaban la primera versión de OpenGL fueron denominadas como Pipeline de Función Fija, esto significa que el procesador gráfico sólo es capaz de hacer tareas muy específicas. La carga de aplicaciones 3D de datos de vértices y atributos para el procesador gráfico discutido anteriormente junto con las texturas, proyecciones específicas y la transformación de matrices para ser utilizado con el objeto, es una etapa de procesado en el Pipeline de Función Fija llamada "Transformación e Iluminación" (T&L), la cual realiza la transformación de vértices y la proyección junto con efectos de iluminación simple, además del mapeo de textura. La geometría resultante es rasterizada y mostrada al usuario. El Pipeline de Función Fija estaba diseñado para funcionar en tarjetas de video antiguas con soporte para OpenGL 1.0.

- Pipeline Programable

Con un hardware que tiene más capacidades y flexibilidad, surgió la necesidad de que los programadores pudieran explotar esas nuevas características, OpenGL 2.0 fue diseñado para cumplir con estos requisitos y permitió que los programadores tuvieran mayor capacidad para escribir programas que funcionaran mejor en el procesador gráfico. En lugar de utilizar el Pipeline de Función Fija para la proyección de vértices (donde el programador tenía que escribir un programa que hiciera la multiplicación de matrices y los cálculos de iluminación), esta nueva especificación permitía al programador utilizar otras funciones dentro de estos cálculos; por lo que las funciones trigonométricas pudieron ser usadas para crear superficies onduladas de los datos de vértices que originalmente eran pasadas al procesador gráfico como una superficie.

Además del procesamiento de vértices, otra etapa de procesamiento fue incluida al Pipeline, la cual se ocupa de los pixeles individuales de la pantalla después de que el proceso de rasterización ha concluido, estos programas son llamados "Programas Fragmentados" porque trabajan con los pixeles, los cuales son llamados "fragmentos" en términos de Open GL.

Mientras que en el caso del Pipeline de Función Fija, los efectos de luz se interpolan entre cada vértice de la superficie, en el Pipeline Programable pueden hacer los efectos de iluminación para cada pixel de forma individual, lo cual puede resultar en efectos mucho más realistas. Un efecto sobresaliente es el llamado "Mapeo Normal" (como se explicó anteriormente), donde las normales son usadas para calcular el ángulo de la luz entrante y por lo tanto el brillo de la superficie es iluminada. El Mapeo Normal es la textura que oculta los componentes de un vector normal en sus canales de color rojo, verde y azul y puede así lograr efectos de iluminación más detallados que no serían posibles únicamente con normales de vértice. La figura 3.8 muestra la forma en que ambos Pipelines trabajan.

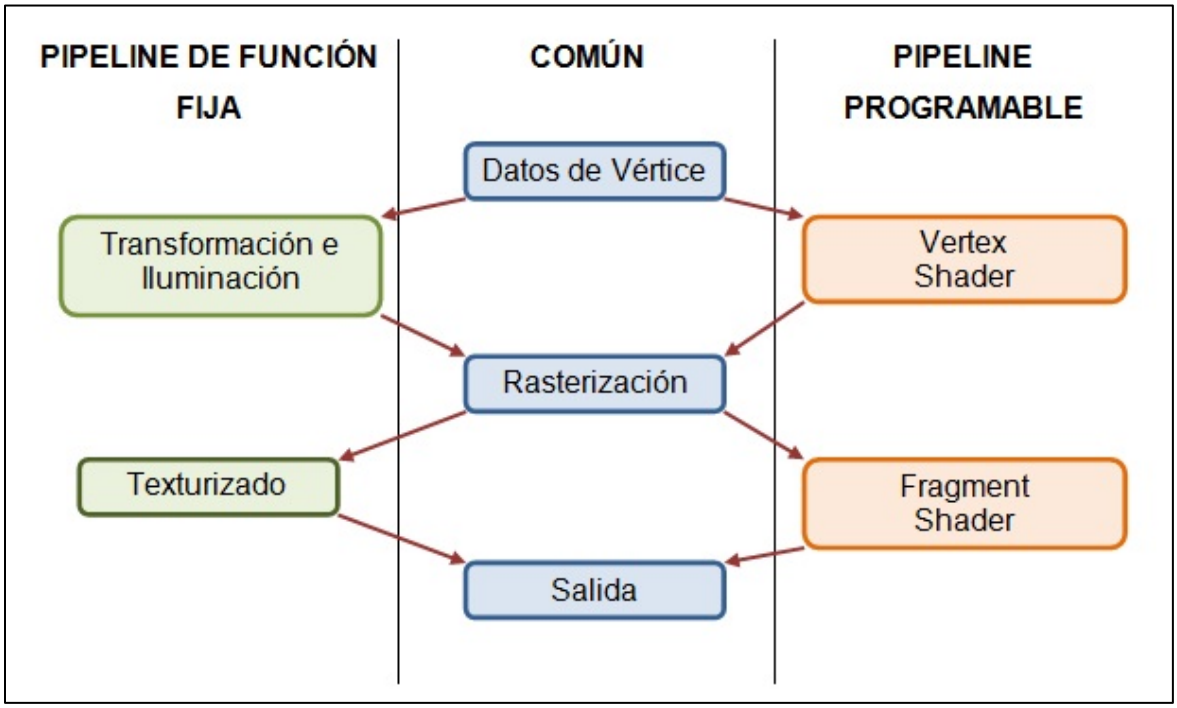


Figura 3.8. Pipeline de Función Fija y Programable en OpenGL.

- Shaders en OpenGL

Las especificaciones más recientes de OpenGL añaden otros tipos de Shaders, como los Shaders geométricos en OpenGL 3.0 que operan en la transformación de geometrías y pueden producir propiedades adicionales, os cuales pueden ser usados por el Shader de Fragmentos en un proceso posterior. OpenGL 4.0 también añadió un Shader de Mosaico que puede crear dinámicamente nuevas primitivas en el procesador gráfico mejorando la calidad de renderizado sin la intervención del CPU.

Aunque el uso de Shaders permite un enfoque más flexible y ágil en el Pipeline, las nuevas especificaciones de OpenGL siguen manteniendo el soporte de la antigua Pipeline de Función Fija por razones de retro compatibilidad.

3.3. Comparación entre OpenGL y DirectX

En general DirectX está diseñado para funcionar solamente bajo los sistemas operativos Windows, además de la versión diseñada para las consolas Xbox. Como se mencionó anteriormente, otras organizaciones externas a Microsoft han realizado intentos para portar DirectX a otras plataformas ajenas a Windows pero sólo se ha logrado de forma parcial, debido a la gran dependencia que tiene esta API con el sistema operativo.

Por otro lado, ya se ha dicho que OpenGL está disponible para un gran número de plataformas tales como Linux, Windows, MacOS y consolas de videojuegos (como las fabricadas por Nintendo y Sony). Como se puede ver, en términos de portabilidad, DirectX es una opción más limitada debido a que está enfocado únicamente a Windows, sin embargo si se planea utilizar sólo este sistema operativo, DirectX sería una mejor opción al ofrecer la máxima compatibilidad con éste.

Funcionalmente DirectX está diseñado para ser una interfaz con el hardware 3D, de tal forma que las características disponibles dependerán de las características del hardware 3D que se tiene. Por otro lado, OpenGL es un sistema que procesa la información y que opcionalmente puede ser acelerado mediante el uso de hardware 3D, pero no es necesariamente requerido.

En términos de facilidad de uso, DirectX le deja a la aplicación en desarrollo la tarea de manejar los recursos de hardware, lo cual causa que se requiera una mayor cantidad de código para ponerse en marcha mientras que OpenGL puede tener implementaciones para la misma tarea en una o pocas llamadas haciendo muchos procedimientos transparentes al usuario aunque con el costo de tener un menor control sobre lo que está pasando en el hardware.

Finalmente, cabe señalar la diferencia entre el sistema de extensión de características que ha adoptado cada una de estas dos API's. Por un lado, OpenGL deja que el fabricante de hardware 3D implemente las características adicionales en sus controladores de hardware; esto permite que las nuevas características estén disponibles con mayor rapidez aunque inicialmente genera una confusión entre desarrolladores ya que es hasta después de un tiempo cuando se estandarizan estas funciones como parte íntegra del API y no sólo como una funcionalidad exclusiva de determinado fabricante. Por otra parte las especificaciones de DirectX son dadas por una sola organización, que en este caso es Microsoft, por lo cual la API se mantiene más consistente, aunque las actualizaciones se lleven a cabo de una forma no muy periódica y algunas veces se pierden de vista algunas características importantes recién implementadas en el hardware de algunos fabricantes. A pesar de que ambas API's presenten ventajas y desventajas, cumplen su tarea principal que es darle al desarrollador las herramientas necesarias para crear todo tipo de diseños tridimensionales de una forma sencilla y accesible. El siguiente cuadro comparativo muestra las principales características de ambas API's gráficas.

	OpenGL	DirectX
Desarrollador original	Silicon Graphics	Microsoft
Desarrollador actual	Khronos Group	Microsoft
Sistema operativo	Múltiple	Windows
Primera publicación	Enero de 1992	Mayo de 1998
Última versión estable	4.2 (8 agosto de 2011)	11 (18 abril de 2011)
Se programa en	C, C++	C++, C#, Visual Basic .NET

3.4. Programación en OpenGL

En esta sección se pretende enseñar al lector las bases de programación de la API de OpenGL versión 2.0, que es una de las versiones más extendidas y soportadas por la mayoría de las tarjetas gráficas. Los ejemplos que se muestran durante este trabajo están escritos usando el lenguaje de programación C, por ser el lenguaje más utilizado actualmente en este tipo de aplicaciones, además de ser el código nativo de OpenGL. También se utiliza la librería *FreeGLUT*[18] la cual será explicada más adelante.

A continuación se mostrará la instalación y la configuración de las librerías necesarias, así como los compiladores correspondientes para los sistemas operativos en que se van a programar los ejemplos.

[18] FreeGLUT es una alternativa de código abierto a GLUT (OpenGL Utility Toolkit) que brinda las bibliotecas de utilidades para programas OpenGL, principalmente proporciona diversas funciones de entrada/salida con el sistema operativo.

● *Instalación en Linux*

Los ejemplos en los que se usa el sistema operativo Linux en este trabajo, se encuentran hechos con la distribución OpenSUSE 11.4, debido a que es una de las distribuciones que cuenta con un mayor soporte y estabilidad en lo referente a drivers, tanto propietarios como de código abierto para la tarjeta gráfica a usarse, dependiendo del chipset gráfico que la computadora tenga; ya sea ATI, Nvidia o Intel. Para empezar debemos seguir los pasos que se muestran a continuación.

- Añadir los repositorios correspondientes:

Abrimos una consola y nos identificamos como root[19]

```
unam@LOCAL:~/Documentos>  su
Contraseña:
LOCAL:/home/unam/Documentos #
```

Una vez identificados como root procedemos a añadir el repositorio correspondiente según nuestra tarjeta gráfica (ATI o Nvidia). En OpenSUSE este procedimiento se puede hacer ya sea mediante el administrador del sistema (YaST) usando la opción "Repositorios de Paquetes" o por consola usando el comando *addrepo*.

- Tarjetas gráficas Nvidia: ftp://download.nvidia.com/opensuse/11.4/

```
zypper addrepo-f ftp://download.nvidia.com/opensuse/11.4/
```

- Tarjetas gráficas ATI: http://linux.ioda.net/mirror/ati/openSUSE_11.4/

```
zypper addrepo-f http://linux.ioda.net/mirror/ati/openSUSE_11.4/
```

Si no se tiene una tarjeta gráfica que use un chip ATI o Nvidia, lo más probable es que se tenga un procesador gráfico Intel y en este caso no es necesario añadir ningún repositorio adicional, debido a que los drives de Intel son de código abierto,

[19] Root es un término usado en Sistemas Operativos de tipo UNIX para referirse a la cuenta de usuario con derechos de administrador, también llamado "Superusuario".

por lo que OpenSUSE instalará automáticamente el controlador que mejor se adapte al dispositivo al actualizar el sistema.

- Instalar el controlador

Ya con el repositorio agregado se instalarán los controladores necesarios dependiendo del modelo de la tarjeta gráfica usada.

Controlador Nvidia GeForce 6 y versiones posteriores:

```
zypper install x11-video-nvidiaG02
```

Para FX5XXX:

```
zypper install x11-video-nvidiaG01
```

Para GeForce 4 o posteriores:

```
zypper install x11-video-nvidia
```

Para ATI:

```
zypper install x11-video-fglrxG02
```

Ahora se requiere instalar el compilador y las librerías necesarias para empezar a programar:

```
zypper install gcc freeglut
```

Por último se reinicia el sistema.

- Compilar Programas

Los programas se compilan en consola con la siguiente orden:

```
gcc -lglut -lGL <archivo fuente> -o <archivo de salida>
```

● *Instalación en Windows*

Para el sistema operativo Windows se eligió el IDE[20] *Code::Blocks* por soportar las últimas versiones de las librerías antes mencionadas, además de ser una aplicación flexible, gratuita y con un amplio soporte en línea. Se puede descargar de la siguiente página:

```
http://www.codeblocks.org/downloads/26
```

Este IDE se puede instalar en Windows 2000, XP, Vista y 7. Al elegir la descarga se debe seleccionar el paquete que incluye al compilador MinGW[21] ya que es necesario para las aplicaciones de OpenGL.

- Configurar el IDE

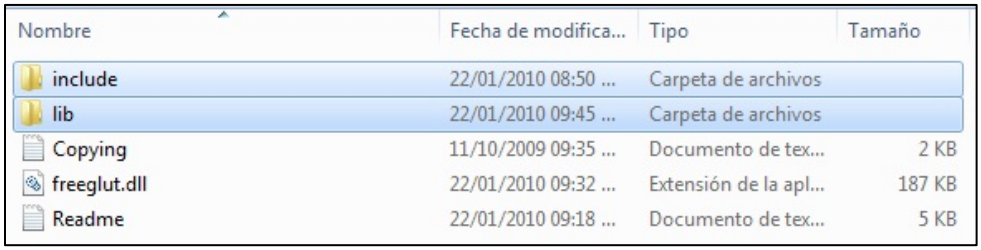
Ya instalado el IDE necesitamos la librería FreeGLUT, la cual es una alternativa a la librería GLUT y representa una mejor opción, debido que a ésta última no se le da soporte desde 1998. FreeGLUT sirve principalmente para el manejo de ventanas, eventos del ratón, menús, entre otras funciones. Puede ser descargado de la siguiente dirección seleccionando el paquete para el compilador MinGW:

```
http://www.transmissionzero.co.uk/software/freeglut-devel/
```

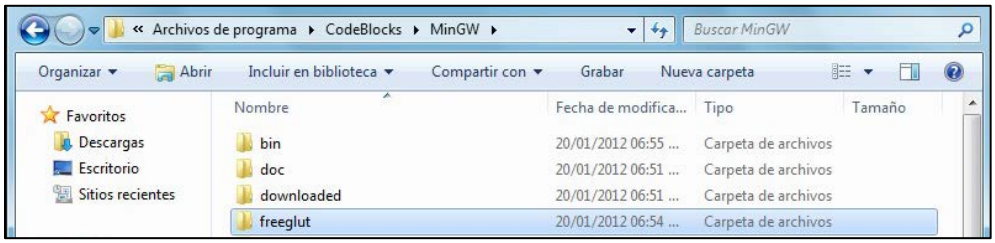
Una vez descargado el paquete de FreeGLUT debemos extraerlo para obtener las librerías.

[20] Siglas en inglés de "Integrated Development Environment" o Entorno de Desarrollo Integrado; es un programa compuesto por un conjunto de herramientas de programación.

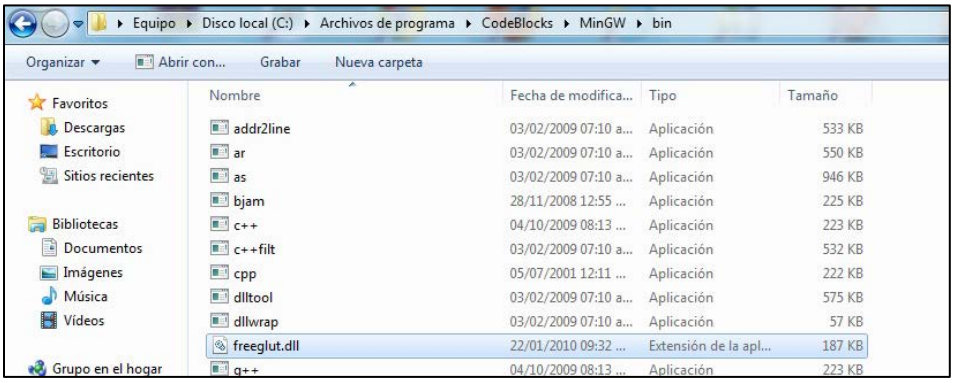
[21] "Minimalist GNU for Windows" es una implementación que permite migrar las capacidades del compilador GCC (GNU Compiler Collection) en Windows.



Ahora hay que copiar las carpetas *include* y *lib* del paquete a la dirección donde está instalado nuestro IDE. Lo más común sería: C:\Program Files\CodeBlocks\MinGW y colocarlos dentro de una carpeta como se muestra en la siguiente captura.



Dentro del paquete descomprimido también viene incluido el archivo DLL[22] *freeglut.dll*. Este archivo lo copiamos en la siguiente ruta: C:\Program Files\CodeBlocks\MinGW\bin

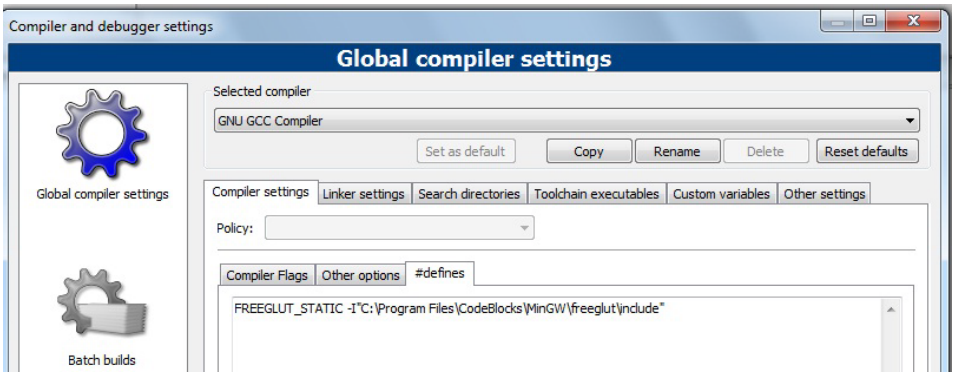


Una vez hecho lo anterior, necesitamos configurar Code::Blocks para que sea capaz de hacer un correcto uso de las librerías al compilar, para esto entramos a

[22] Siglas en inglés de "*Dynamic-Link Library*" o biblioteca de enlace dinámico; es el término con el que se refiere a los archivos con código ejecutable que se cargan bajo demanda de un programa en sistemas operativos Windows.

Settings >> Compiler and debugger… en la ventana abierta nos dirigimos a *Compiler settings >> #defines* y en el recuadro agregamos la siguiente línea:

```
FREEGLUT_STATIC -I"C:\Program Files\CodeBlocks\MinGW\freeglut\include"
```

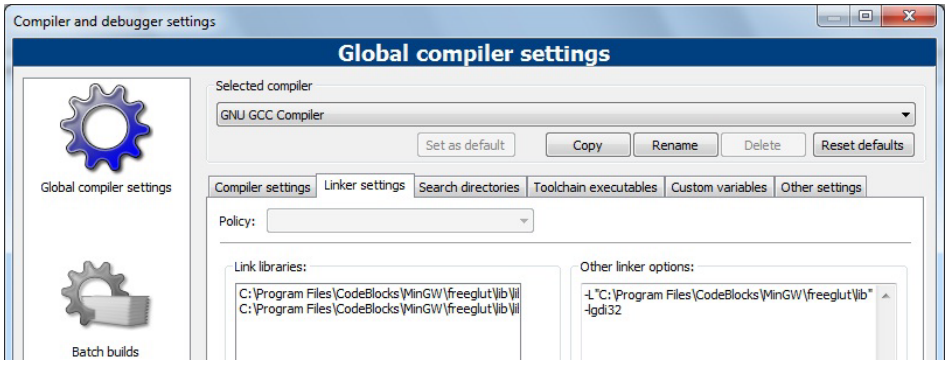


Ahora en esa misma ventana damos clic en la pestaña *Linker settings*, una vez ahí, en el recuadro *Link libraries* agregamos la ruta de nuestra librería:

```
C:\Program Files\CodeBlocks\MinGW\freeglut\lib\libfreeglut.a
C:\Program Files\CodeBlocks\MinGW\freeglut\lib\libfreeglut_static.a
```

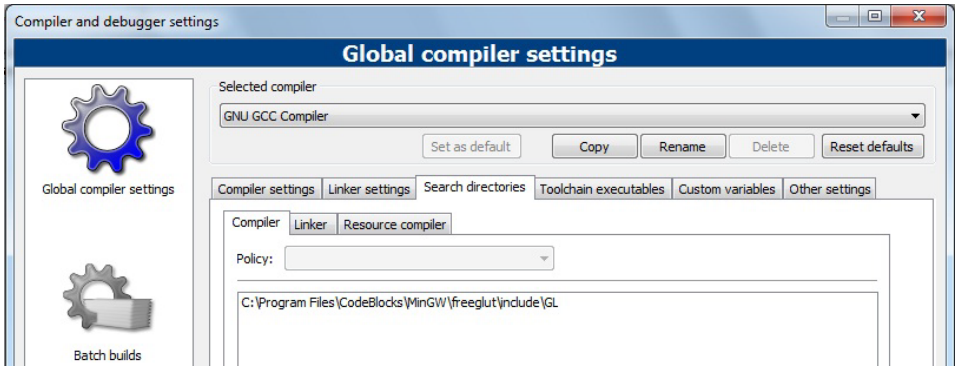
Y en el recuadro *Other linker options* ponemos esta línea para enlazar las bibliotecas:

```
-L"C:\Program Files\CodeBlocks\MinGW\freeglut\lib" -lfreeglut_static -
lfreeglut -lglu32 -lopengl32 -lwinmm -lgdi32
```



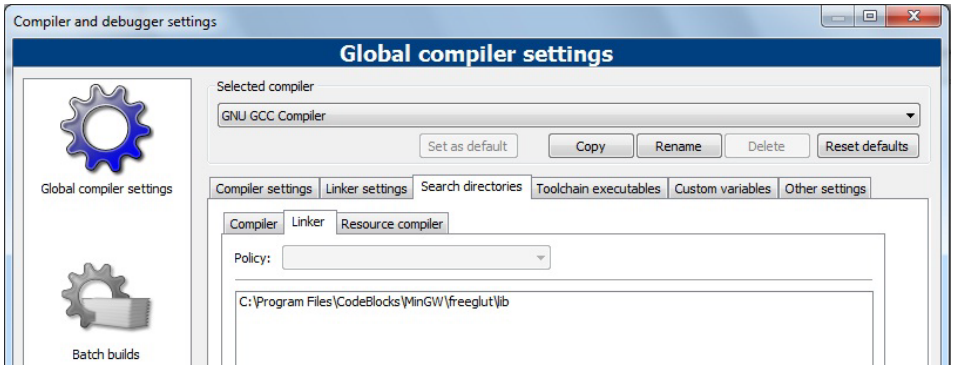
En la pestaña *Search directories* en el recuadro de *compiler* igualmente ponemos la ruta:

```
C:\Program Files\CodeBlocks\MinGW\freeglut\include\GL
```

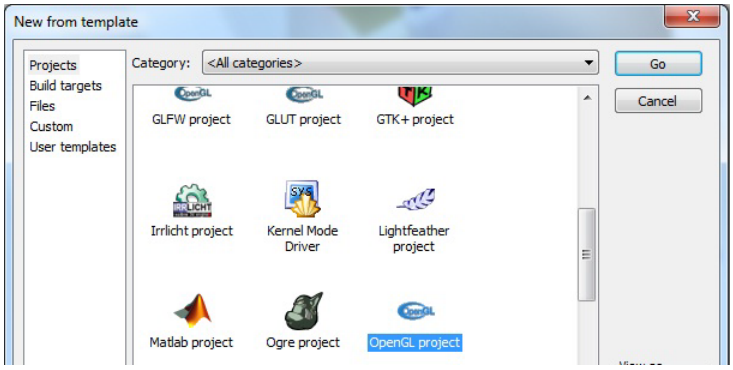


Por último, en el mismo apartado pero en la pestaña *linker* indicamos la dirección:

```
C:\Program Files\CodeBlocks\MinGW\freeglut\lib
```



Ya tenemos Code::Blocks correctamente configurado, ahora sólo hay que crear un nuevo proyecto y elegir la opción *OpenGL Project:*



Con esta guía de instalación y de inicialización deberá ser suficiente para empezar a desarrollar las aplicaciones OpenGL sin ningún inconveniente. Antes de comenzar a programar es importante conocer la sintaxis de la API de OpenGL, por ejemplo todas las funciones empiezan con el prefijo "gl" y las constantes con "GL"; en el manejo de la librería FreeGLUT las funciones empiezan con el prefijo "glut" y las constantes con "GLUT".

En muchas de las funciones, aparece un sufijo compuesto por dos caracteres, una cifra y una letra, como por ejemplo glColor3f() o glVertex3f(). La cifra simboliza el número de parámetros que se le deben pasar a la función y la letra el tipo de estos parámetros.

En OpenGL existen ocho tipos distintos de datos, de una forma muy parecida a los tipos de datos de C o C++. A continuación se muestra la tabla con los tipos:

Sufijo	**Tipo de dato**	**Corresponde en C al tipo:**	**Definición en OpenGL del tipo:**
b	Entero 8 bits	signed char	GLbyte
s	Entero 16 bits	short	GLshort
i	Entero 32 bits	int o long	GLint, GLsizei
f	Punto flotante 32 bits	float	GLfloat, GLclampf
d	Punto flotante 64 bits	double	GLdouble, GLclampd
ub	Entero sin signo 8 bits	unsigned char	GLubyte, GLboolean
us	Entero sin signo 16 bits	unsigned short	GLushort
ui	Entero sin signo 32 bits	unsigned int	GLuint, GLenum, GLbitfield

A continuación se enseñarán los ejemplos de programación, empezando por los *Puntos* que son la unidad básica para construir las primitivas primordiales como las líneas y polígonos, para posteriormente pasar a ejemplos más detallados como lo son los cubos y esferas que forman parte de las primitivas 3D.

Para algunos ejemplos es necesario auxiliarse del encabezado *Libreria.h* que contienen funciones para crear formas tridimensionales, como es el cubo o la

pirámide, manejo de enumeraciones para simular booleanos, así como la carga y el manejo de imágenes en el apartado de texturas. A continuación se muestra su definición:

```
 1  #ifndef _LIBRERIA_1_H
 2  #define _LIBRERIA_1_H
 3
 4    enum boleano {FALSO, VERDADERO};
 5    enum perspectivas{PERSPECTIVA, ORTHO, FRUSTUM};
 6    enum transformaciones{ESCALAR,ROTAR,TRANSLADAR};
 7    enum ejes{X,Y,Z};
 8
 9    typedef struct Image//Estructura para manejar una Imagen BMP
10    {
11          unsigned long tamanoX;
12          unsigned long tamanoY;
13          char *datos;
14    }Image;
15
16    int cargaImagen(char *nombre_archivo, Image *imagen);
17    void cuadradoColor();
18    void cuadrado();
19    void piramide();
20    void centrarVentana();
21    void dibujarEjes();
22
23  #endif
24  #ifndef _LIBRERIA_1_H
```

● Puntos

El *punto* es la unidad básica para construir las primitivas que posteriormente formarán el modelo tridimensional, el siguiente código muestra la forma para dibujar dichos puntos en la pantalla.

```
1   #include <GL/freeglut.h>
2
3   void redibujar(int width, int height)
4   {
5         glViewport(0, 0, width, height);
6         glMatrixMode(GL_PROJECTION);
7         glLoadIdentity();
8         glOrtho(-2, 2, -2, 2, -2, 2);
9         glMatrixMode(GL_MODELVIEW);
10  }/*Fin del método redibujar*/
11
```

```
12  void dibuja()
13  {
14        glClear(GL_COLOR_BUFFER_BIT);
15        glColor3f(1,1,1);
16        glLoadIdentity();
17
18        glPointSize(5.0f);
19        glBegin(GL_POINTS);
20              glVertex3f(0,0,0);
21              glVertex3f(1,0,0);
22        glEnd();
23        glFlush();
24  }/*Fin del método dibuja*/
25
26  void borrar()
27  {
28        glClearColor(0,0,0,0);
29  }
30
31  int main(int argc, char **argv)
32  {
33        glutInit(&argc, argv);
34        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
35        glutInitWindowPosition(50, 50);
36        glutInitWindowSize(200, 200);
37        glutCreateWindow("Puntos");
38        borrar();
39        glutDisplayFunc(dibuja); /*Registra llamada a callback*/
40        glutReshapeFunc(redibujar); /*Registra llamada a callback*/
41        glutMainLoop();
42        return 0;
43  }/*Fin del método main*/
```

El código anterior es una estructura muy común en la programación en OpenGL dividida en 3 funciones básicas que son el borrado, el dibujado y el redibujado, más la función *main* que será la que ejecute el programa.

Empezaremos analizando la función *borrar()* (línea 26), esta función invoca a *glClearColor(0,0,0,0)*; lo que hace es obtener el color con el que se borrará el buffer al hacer un *glClear()*, antes de poder plasmar los objetos que queremos. El color de la función se da por tres parámetros en función de la gama de colores RGB[23] y uno más que define la transparencia.

[23] De las iniciales de los colores en inglés: Red, Green, Blue (Rojo, Verde y Azul). Es un modelo de colores con el que es posible representar cualquier otro color teniendo como base los tres colores primarios.

- Función *dibuja()* (linea 12), contiene los elementos que se van a mostrar en la pantalla, siguiendo una ordenada llamada de funciones.

- Función *glClear(GL_COLOR_BUFFER_BIT)* (línea 14): Esta función limpia la pantalla con el color que se obtuvo de la función *dibuja*(línea 12).

- Función *glColor3f*(1,1,1) (línea 15): Obtiene el color con el que se van a pintar los objetos en la pantalla en función de la gama de colores RGB, en este ejemplo se van a dibujar los puntos de color blanco en un fondo negro.

- Función *glLoadIdentity()* (línea 16): Carga la matriz identidad, para que la multiplicación que se va a realizar en *glOrtho* produzca los resultados esperados.

- Función *glPointSize(5.0f)* (línea 18): Con esta función podemos cambiar el ancho del punto que dibujamos, por default el tamaño del punto es de un pixel.

- Función *glBegin(GL_POINTS)* (línea 19): Es la encargada de dibujar las primitivas; recibe como argumento una constante que le indica el tipo de primitiva a dibujar. Las primitivas que se quieran dibujar deberán estar incluidas entre el bloque *glBegin(Constante)* y *glEnd()*.

En la siguiente tabla se muestran los posibles valores que pueden pasarse como parámetro de la función *glBegin(Constante)* y el correspondiente tipo de primitiva que se generará como resultado:

Constante	Significado
GL_POINTS	Trata a cada vértice como un solo punto.
GL_LINES	Trata a cada par de vértices como un segmento independiente.
GL_LINE_STRIP	Dibuja un grupo de segmentos que empiezan por el primer vértice hasta el último.
GL_LINE_LOOP	Dibuja un grupo de segmentos conectados que empiezan por el primer vértice hasta el último.

GL_TRIANGLES	Trata a cada tres vértices como un triángulo independiente.
GL_TRIANGLE_STRIP	Dibuja triángulos adyacentes
GL_TRIANGLE_FAN	Dibuja triángulos adyacentes con un vértice común
GL_QUADS	Trata cuatro vértices como un polígono de cuatro lados
GL_QUAD_STRIP	Dibuja polígonos de cuatro lados adyacentes.
GL_POLYGON	Dibuja un polígono dependiendo del número de vértices.

El parámetro que se le dio a *glBegin* es *GL_POINTS*, es el que interpreta los vértices contenidos en el bloque *glBeing-glEnd* como puntos.

- Función *glVertex3f(,,)* (linea 20): Esta función contiene el vértice correspondiente en el plano X,Y,Z dependiendo de las coordenadas que se le indiquen. Estas coordenadas deberán estar en el límite cuando creamos la perspectiva ortogonal (linea 8), ya que si le asignamos coordenadas que se pasan del rango, los objetos que se quieran mostrar no se verán. En este ejemplo se dibujarán dos puntos en las coordenadas (0, 0, 0) y (1, 0, 0). Dependiendo del tipo de primitiva que se quiera plasmar, la función *glVertex3f(,,)* guardará un punto solamente, si se quiere dibujar un triángulo se deberá invocar la función tres veces con las coordenadas correspondientes dentro del bloque *glBegin-glEnd*.

- Función *glEnd()* (linea 22): Simplemente cierra el bloque.

- Función *glFlush()* (línea 23): Dependiendo del hardware, controladores, etcétera, que esten instalados, OpenGL guarda los comandos como peticiones en pila para optimizar el rendimiento. La función *glFlush* lanza todas esas peticiones, forzando también que se dibuje todo lo que está en el método dibujo.

- La función *redibujar()* (línea 3) contiene llamadas a funciones cuando pasan ciertos eventos, por ejemplo lo que pasaría si el usuario cambia el tamaño de la ventana, la minimiza o pone encima otra ventana.

Dentro de esta última función se encuentran las siguientes llamadas:

- Funcion *glViewport(0, 0, width, height)* (línea 5): Esta función define una área de dibujo donde puede dibujar OpenGL. Los parámetros son x, y; la esquina superior izquierda del área donde puede dibujar (con referencia a la ventana).
- Función *glMatrixMode(GL_PROJECTION)* (línea 6): Especifica la matriz actual. En OpenGL las operaciones de rotación, traslación, escalado, etcétera. se realizan a través de matrices de transformación (que se verán más adelante). Dependiendo de lo que estemos tratando, hay tres tipos de matriz ("flags"): matriz de proyección (GL_PROJECTION), matriz de modelo (GL_MODELVIEW) y matriz de textura (GL_TEXTURE). Con esta función indicamos a cuál de estas tres le deben afectar las operaciones. Concretamente, GL_PROJECTION afecta a las vistas, perspectivas o proyecciones.
- Función *glLoadIdentity()* (linea 7): Con esta función cargamos el tipo de matriz identidad.

- Función *glOrtho(-2, 2, -2, 2, -2, 2)* (línea 8): *glOrtho()* define una perspectiva ortogonal. Esto quiere decir que lo que veremos será una proyección en uno de los planos definidos por los ejes. Para entenderlo mejor, supongamos que definimos un área de dibujo de 300x300 pixeles, la función *glOrtho* de acuerdo a los parámetros delimitará esa porción de área. Los parámetros son x_minima, x_maxima, y_minima, y_maxima, z_minima, z_maxima. Con estos seis puntos se define una caja que será lo que se proyecte.

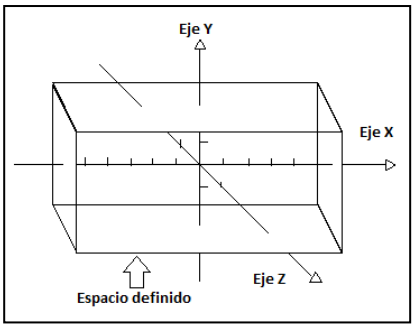


Figura 3.9. Perspectiva ortogonal.

- Función *glMatrixMode(GL_MODELVIEW)* (linea 9): Volvemos a llamar la función *glMatrixMode*, ya que definimos nuestra perspectiva ortogonal pero ahora con el *flag GL_MODELVIEW*, ya que afecta principalmente a las primitivas.

Por último analizamos la función *main()* que contiene las llamadas a las funciones de la biblioteca FreeGLUT así como las dos funciones *callback* que contiene.

Uno de los problemas que se presentó en la programación en OpenGL fue el entorno en que se quería ejecutar, recordemos que OpenGL no tiene nada incluido para el manejo de ventanas, así como la forma de interpretar los eventos que se disparan en cada sistema operativo, ya que cada sistema tiene una forma diferente de implementar sus ventanas y eventos, aquí es donde entra FreeGLUT.

La biblioteca FreeGLUT en sí, no forma parte de OpenGL pero resuelve el problema antes mencionado, utilizando la misma versión de FreeGLUT en todas las plataforma sin perder portabilidad y no hacer una versión diferente para cada sistema, haciéndola imprescindible en el desarrollo de OpenGL.

Figura 3.10. FreeGLUT ayuda a manejar el sistema de ventanas sin importar el SO.

El contenido de la función *main()* es el siguiente:

- Función *glutInit(&argc, argv)* (linea 33): Esta función es la que inicializa FreeGLUT y negocia con el sistema de ventanas para abrir una. Los parámetros deben ser los mismos *argc* y *argv* sin modificar la de *main()*. Es muy importante

que sea la primera función en llamarse, ya que de lo contrario las siguientes llamadas no tendrán efecto.

- Función *glutInitDisplayMode(GLUT_SINGLE* | *GLUT_RGB)* (linea 34): Define el modo en el que se debe dibujar en la ventana. Los parámetros (como gran parte de las funciones que iremos viendo) se definen con *flags* o máscaras de bits. En este caso, *GLUT_SINGLE* indica que se debe usar un solo buffer, *GLUT_RGB* indica el tipo de modelo de color a usar.

- Función *glutInitWindowPosition(50, 50)* (linea 35): Recibe la posición X y Y de la esquina superior izquierda de la nueva ventana.

- Función *glutInitWindowSize(500, 500)* (línea 36): Recibe el ancho y alto de la nueva ventana.

- Función *glutCreateWindow("Puntos")* (línea 37): Esta función es la que propiamente crea la ventana y el parámetro es el nombre de la misma.

- Función *borrar()* (linea 38): Invoca a la función ya descrita.

- Función *glutDisplayFunc(dibuja)* (línea 39): Aquí se registra el primer "callback". En términos de programación una callback es una función que recibe como argumento la dirección o puntero de otra función, en programación orientada a objetos sería el equivalente al polimorfismo; por ejemplo en el lenguaje de programación Java la función *glutDisplayFunc* sería el equivalente a una clase del tipo interfaz ya que es un método abstracto y el parámetro dibuja (la función) sería la clase que implementa la interfaz y contiene la lógica de lo que se quiere dibujar. La siguiente tabla muestra los callbacks o llamadas que se utilizan frecuentemente para el desarrollo de aplicaciones:

Callback
void glutDisplayFunc(void (*func)(void)). Función encargada del despliegue que se quiere mostrar en la ventana.
void glutIdleFunc(void (*func)(void)). Es una función que se llama cuando no se registra un evento, se usa principalmente en el manejo de animaciones resultantes de las transformaciones de OpenGL, como son la rotación, escalado, etcétera.
void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y)). Callback de teclado, recibe el caracter presionado y la posición del ratón.
void glutMotionFunc(void (*func)(int x, int y)). Se llama cuando se mantienen presionados los botones del ratón.
void glutMouseFunc(void (*func)(int button, int state, int x, int y)) Se llama al presionar o soltar un botón del ratón.
void glutReshapeFunc(void (*func)(int width, int height)). Es llamado cuando se cambia el tamaño de la ventana.
void glutSpecialFunc(void (*func)(int key, int x, int y)). Se llama al presionar teclas especiales del teclado por ejemplo F1, teclas de desplazamiento del cursor entre otras.
void glutTimerFunc(unsigned int msecs, void (*func)(int value), value). Se llama al transcurrir un intervalo en milisegundos.

Es muy importante a la hora de registrar un callback con la función que le pasemos como parámetro, que se respete el prototipo de función, esto es definir nuestra función con los mismos parámetros en que está definido nuestro callback.

En el ejemplo, la función pasada como parámetro será llamada cada vez que FreeGLUT determine oportuno que la ventana debe ser redibujada, como al maximizarse, poner otras ventanas por encima o sacarlas.

- Función *glutReshapeFunc(redibujar)* (línea 40): Aquí registramos otro callback, en este caso para saber qué hacer cuando la ventana es reescalada. Esta acción afecta en principio directamente al render, puesto que se está cambiando el tamaño del plano de proyección. También se suele usar para iniciar las matrices de proyección y transformación que usa OpenGL, que se verán más adelante.

- Función *glutMainLoop()* (línea 41): Esta función cede el control del flujo del programa a FreeGLUT, que a partir de estos "eventos", irá llamando a las funciones que han sido pasadas como callbacks.

El programa ya compilado tanto en Windows como en Linux, se puede ver en la siguiente captura:

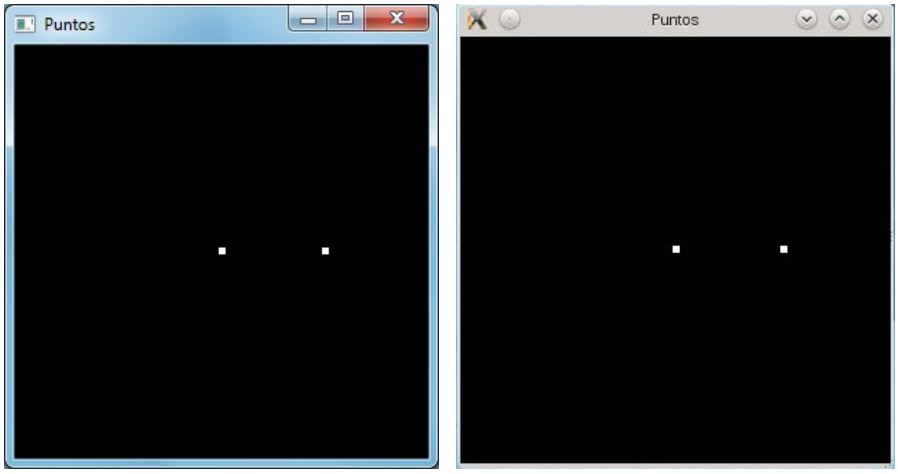


Figura 3.11. Puntos programados en OpenGL.

Para los siguientes ejemplos solamente se van a comentar las nuevas líneas y funciones del programa, ya que la mayoría del código es igual y sólo cambia en ciertas áreas. El código para cada uno de los ejemplos se encuentra en el CD que acompaña a este trabajo dentro de la carpeta "Ejemplos OpenGL". A continuación se mostrarán los ejemplos de las principales primitivas de OpenGL.

PRIMITIVAS

Como ya hemos mencionado, una primitiva es simplemente la interpretación de un conjunto de vértices con el fin de dibujar figuras geométricas de una manera específica en la pantalla. Hay diez primitivas distintas en OpenGL, pero las de mayor relevancia son: Líneas (GL_LINES), triángulos (GL_TRIANGLES) y cuadrados (GL_QUADS). Además de sus variantes GL_LINE_STRIP,

GL_TRIANGLE_STRIP y GL_QUAD_STRIP, que son utilizadas para definir "tiras" de líneas, triángulos y de cuadrados respectivamente. En la carpeta "02-Primitivas" del CD viene el código para un programa que al ser compilado muestra en pantalla las primitivas disponibles (figura 3.12).

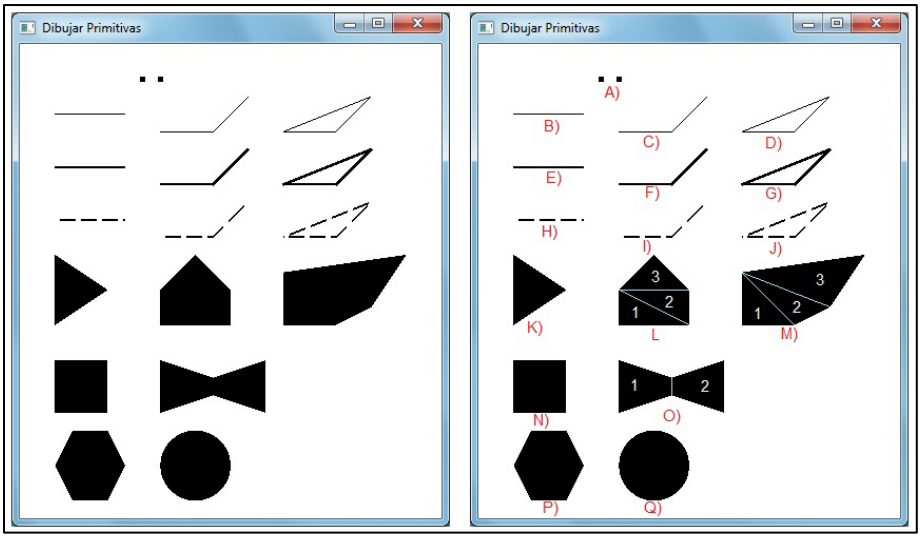


Figura 3.12. Primitivas **A)** Puntos **B)** Línea normal **C)** Línea tira **D)** Línea cuerda **E, F, G)** Línea suavizada **H, I, J)** Línea punteada **K, L, M)** Triángulos **N, O)** Cuadrados **P, Q)** Polígonos.

● Líneas

La *línea* es una de las primitivas primordiales usadas por OpenGL. El programa dibuja en la pantalla distintos tipos de líneas, usando además los atributos de línea punteada y línea suavizada.

Las *líneas* funcionan de forma similar a los puntos, sólo que en las líneas los vértices se cuentan por parejas, denotando un punto inicial y un punto final. Los atributos y nuevas funciones usadas para este programa son los listados a continuación:

- Función glDisable(GL_LINE_STIPPLE)(línea 24): Desactiva la propiedad del punteado.

- Función glDisable(GL_LINE_SMOOTH)(línea 25): Desactiva la propiedad del suavizado.

- Función *glBegin(GL_LINES)* (línea 26): Esta es la función principal que indica el inicio del dibujado de una línea normal.

- Función *glBegin(GL_LINE_STRIP)* (línea 30):Si en lugar de *GL_LINES* utilizamos *GL_LINE_STRIP*, OpenGL ya no trataría los vértices en parejas, si no que el primer vértice y el segundo definirían una línea y el final de ésta definiría otra línea con el siguiente vértice y así sucesivamente, definiendo un segmento continuo.

- Función *glBegin(GL_LINE_LOOP)* (línea 35): Funciona de igual forma que GL_LINE_STRIP, pero además también une el primer vértice con el último, creando siempre una cuerda cerrada.

- *glEnable(GL_LINE_SMOOTH)* (línea 40): Activa el atributo que corresponde a una línea suavizada.

- *glEnable(GL_LINE_STIPPLE)* (línea 57): Activa el atributo de línea punteada.

- Función *glLineStipple(i,0x00FF)* (línea 57): Esta función define el patrón para dibujar las líneas punteadas. El primer parámetro define el factor de escalado del patrón, cuanto más alto, más se alarga. El segundo parámetro, es un fragmento de punteado de línea que se repite, esto se hace a través del sistema binario: donde 0 es no pintar y 1 es pintar. En el ejemplo el parámetro está escrito en hexadecimal; 0xeeee en binario es: 1110 1110 1110 1110. El programa lee el número binario al revés, iniciando por el final como en la figura:

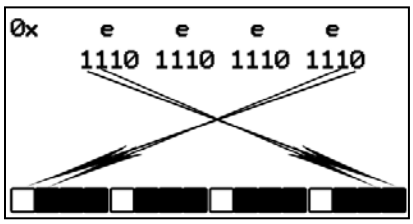


Figura 3.13. Patrón de punteado en la línea.

● Polígonos

Para la creación de objetos sólidos, el uso de puntos y líneas es insuficiente, para esto se necesitan primitivas que sean de superficie cerrada, ilumidadas de uno o varios colores y que en conjunto modelen el objeto deseado. En el área de la representación 3D de los gráficos por computadora, se suelen utilizar polígonos (que a menudo son triángulos) para dar forma a objetos que simulen ser sólidos (ya que en realidad solamente son superficies huecas por dentro).

► Triángulos

El polígono más simple es el triángulo, ya que sólo está formado por tres lados; en esta primitiva, los vértices van de tres en tres. Las funciones disponibles para programarlo en OpenGL son las siguientes:

- Función *glBegin(GL_TRIANGLES)* (Línea 74): Es la función que indica que va a dibujarse un triángulo, indicando la posición de los tres vértices que lo conformarán.

- Función *glBegin(GL_TRIANGLE_STRIP)* (Línea 79): Esta función trabaja con el mismo principio de *LINE_STRIP*, primero damos los vértices de un triángulo y a partir de ahí cada nuevo vértice formará, con los dos anteriores, otro triángulo, creando así triángulos encadenados. De tal forma que con cinco vértices se crearán tres triángulos, como se ve en la figura 3.14.

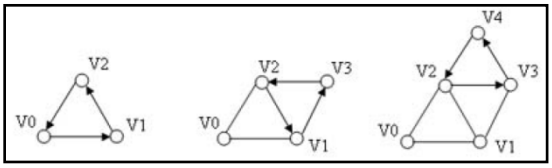


Figura 3.14. Función TRIANGLE_STRIP.

- Función *glBegin(GL_TRIANGLE_FAN)* (Línea 86): Los triángulos creados con la función *TRIANGLE_FAN* son parecidos a los creados con TRIANGLE_STRIP, diferenciándose en que los tres primeros vértices forman un triángulo y partir de

ahí con cada nuevo vértice se genera un nuevo triángulo con el nuevo vértice, el anterior a éste y el primero. Así todos los triángulos comparten un punto central, haciéndolo parecer un abanico o una sombrilla, pero en general se usa para hacer circunferencias.

La razón por la que los diseños avanzados de tercera dimensión se generan mediante el encadenamiento de triángulos es por la sencillez y eficiencia de estos al necesitar sólo tres vértices para formarse.

► Cuadrados

Esta primitiva funciona exactamente igual que *GL_TRIANGLES*, pero usando cuatro vértices en lugar de tres para dibujar cuadrados. También tiene la variación de *GL_QUAD_STRIP*, para dibujar "tiras" de cuadrados.

- Función *glBegin(GL_QUADS)* (Línea 93): Indica que se va a dibujar un cuadrado dando la posición de los cuatro vértices que lo formarán.

- Función *glBegin(GL_QUAD_STRIP)* (Línea 99): Esta función como ya se mencionó, sirve para encadenar cuadrados, así en el ejemplo anterior se indica la posición de seis vértices con los cuales se formarán dos cuadrados.

- Función *glBegin(GL_POLYGON)* (Línea 107): Cuando se desea formar polígonos de más de cuatro lados se debe usar la función *GL_POLYGON*, en el ejemplo se dibujan un hexágono y una circunferencia indicándole la posición de los seis y cincuenta vértices respectivamente que son necesarios para crearlos.

En la figura 3.15 se resumen las diez primitivas que se manejan en OpenGL, se muestra la función para cada una de ellas, así como el dibujo resultante una vez usada dicha función.

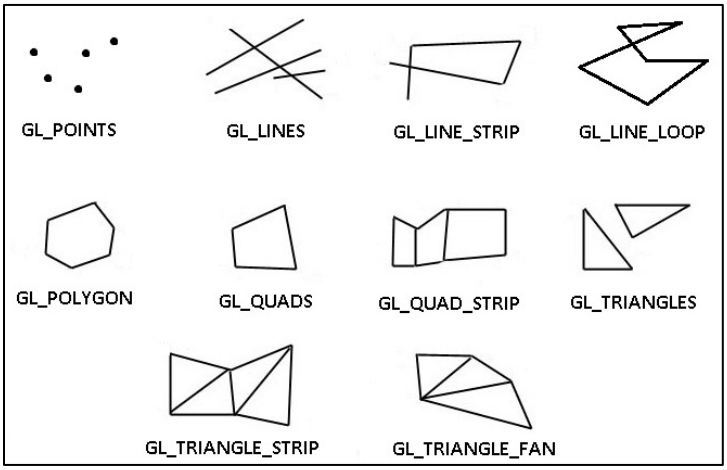


Figura 3.15. Primitivas en OpenGL.

Ahora que ya conocemos todas las primitivas que OpenGL es capaz de manejar y antes de proceder a conocer las variantes tridimensionales de estas primitivas, es necesario mencionar tres conceptos de gran importancia cuando se trabaja con primitivas 3D; estos son Z-Buffer, transformaciones y proyecciones, así como la forma de programarlos.

● Z-Buffer

Cuando tenemos un objeto sólido, o quizá varios objetos, algunos de ellos estarán más próximos a la cámara que otros. Si un objeto está por detrás de otro, evidentemente sólo veremos el de adelante que cubre al de atrás. OpenGL por defecto no toma en cuenta esta posible situación, de forma que simplemente va pintando en pantalla los puntos, líneas y polígonos siguiendo el orden en el que se especifican en el código.

Para solucionar esto, es necesario hacer uso del llamado *z-buffering* o Z-Buffer que en términos de gráficos por computadora hace referencia a la parte de la memoria de video que se encarga de gestionar las coordenadas de profundidad en los gráficos tridimensionales.

OpenGL pone a nuestra disposición el Z-Buffer, al que se le llama "depth buffer" o buffer de profundidad. En él se almacenan las coordenadas de las zetas o distancias desde el punto de vista a cada píxel de los objetos de la escena y a la hora de dibujarlos en pantalla hace una comprobación para identificar que no haya ninguna primitiva que esté por delante, cubriendo a lo que vamos a dibujar en ese lugar.

Para activar esta característica, solo tenemos que hacer una modificación de su variable en la máquina de estados de OpenGL, usando *glEnable*. El ejemplo de la Carpeta "03-Z-Buffer" del CD, muestra el código para usar el buffer de profundidad, el cual se activa presionando la tecla "a" y se desactiva con "d". Al compilar el programa debe verse en pantalla como la figura 3.16; las funciones empleadas para este programa son los siguientes:

- Función *glDepthFunc(GL_LEQUAL)* (Línea 17): En esta línea decimos que el nuevo píxel se dibuje si su coordenada Z es más cercana al punto de visualización, o igual que la que hay actualmente en Z-Buffer.

- Función *glEnable(GL_DEPTH_TEST)* (Línea 19): Con esta función habilitamos la comprobación de la profundidad en el dibujado.

- Función *glClearDepth(1.0)* (Línea 20): En esta línea le decimos al programa que cada vez que se borre el buffer de profundidad se restauren sus posiciones al valor 1.0.

- Función *glClear(GL_COLOR_BUFFER_BIT* | *GL_DEPTH_BUFFER_BIT)* (Línea 21): El uso del "test de profundidad" conlleva a que en la función que renderiza la escena, se necesite limpiar la pantalla antes de dibujar la siguiente escena, esto se logra con estas funciones, las cuales borran el buffer de color y el Z-Buffer respectivamente.

- Función *void teclado(unsigned char key, int x, int y)* (Línea 42): Definimos nuestra función que actuará en el activado y desactivado del buffer Z, por medio de las teclas A y D.

- Función *glDisable(GL_DEPTH_TEST)* (Linea 48): Desactiva la comprobación del Buffer Z.

- Función *glutSetWindowTitle(char *name)*(Línea 49 y 53): Pone un título en la ventana.

- Función *glutPostRedisplay()* (Línea 56): Esta función le indica a FreeGLUT que fuerce un repintado y se mostrarán los cambios que se hicieron al activar o desactivar el buffer.

- Función *glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH)* (Línea 66): Aquí se indica al programa que cuando se cree la ventana, se reserve espacio para el Z-Buffer.

Algunas líneas de código como la 6 *glMatrixMode(GL_PROJECTION)*; se tratarán con mayor detalle en la sección de la Proyección, ya que el Z-Buffer y la proyección tienen una gran relación cuando se trabaja con diseños tridimensionales.

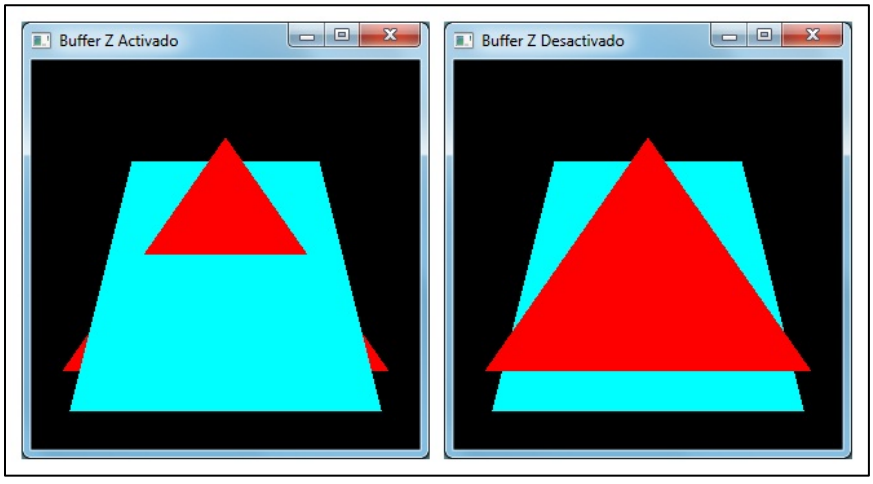


Figura 3.16. Z-Buffer activado y desactivado.

● Transformaciones

Las Transformaciones geométricas son las operaciones que permiten crear una nueva figura a partir de otra previamente dada, haciendo corresponder una serie de puntos coordenados, desde los cuales se pueden aplicar diversas fórmulas para realizar un cambio en la figura. Las transformaciones manejadas en OpenGL son traslación, rotación y escalado.

La traslación es el movimiento en línea recta de un objeto, de una posición a otra, sin rotarla ni voltearla, simplemente "deslizándose"; esto quiere decir que la figura sigue viéndose exactamente igual, sólo que en un lugar diferente. Se aplica una transformación en un objeto para cambiar su posición a lo largo de la trayectoria de una línea recta de una dirección de coordenadas a otra.

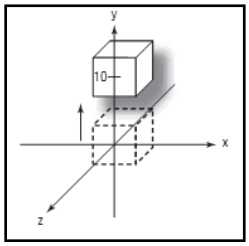


Figura 3.17. La traslación.

Los cálculos que se hacen para calcular la transformación en el sistema de coordenadas tridimensionales, se logran por medio de la multiplicación de dos matrices. La primera matriz conocida como la matriz de modelado es del tipo 4x4 que contendrá las unidades que se quieren trasladar y la segunda contendrá el vértice que queramos trasladar. Las matrices para calcular la traslación se definen de la siguiente forma:

$$\begin{bmatrix} Xt \\ Yt \\ Zt \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

[4x1] = [4x4] * [4x1]

Supongamos que tenemos un vértice con las coordenadas (2, 2, 0) y queremos trasladar una unidad en X, Y, Z, entonces las operaciones a calcular son las siguientes:

$$\begin{bmatrix}3\\3\\1\\1\end{bmatrix}=\begin{bmatrix}1&0&0&1\\0&1&0&1\\0&0&1&1\\0&0&0&1\end{bmatrix}*\begin{bmatrix}2\\2\\0\\1\end{bmatrix}$$

En OpenGL la operación de translación se logra por medio de dos funciones:

Función	Significado
void glTranslated (GLdouble x, GLdoubley, GLdouble z)	Especifica las coordenadas x, y, z en donde se traslade al nuevo punto, usando parámetros del tipo GLdouble.
void glTranslatef (GLfloat x, GLfloat y, GLfloat z)	Especifica las coordenadas x, y, z en donde se traslade al nuevo punto, usando parámetros del tipo GLfloat.

El escalado consiste en alterar el tamaño de un objeto, dependiendo del factor de escalación aplicado, el objeto sufrirá un cambio en su tamaño, siendo mayor o menor en su segmento de longitud.

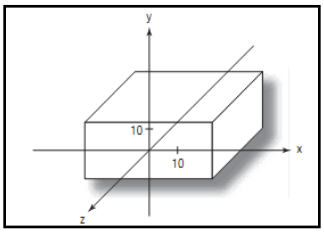


Figura 3.18. El escalado.

El escalado se logra por media del siguiente producto de matrices:

$$\begin{bmatrix}Ex\\Ey\\Ez\\1\end{bmatrix}=\begin{bmatrix}Ex&0&0&0\\0&Ey&0&0\\0&0&EZ&0\\0&0&0&1\end{bmatrix}*\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}$$

[4x1] = [4x4] * [4x1]

Con el ejemplo anterior, nuestro vértice se localiza en las coordenadas (3, 3, 1), si queremos hacer un factor de escala de tres sobre el eje Z, la operación quedaría de esta forma:

$$\begin{bmatrix}3\\3\\3\\1\end{bmatrix}=\begin{bmatrix}1&0&0&1\\0&1&0&1\\0&0&3&1\\0&0&0&1\end{bmatrix}*\begin{bmatrix}3\\3\\1\\1\end{bmatrix}$$

La operación de escalado se logra por medio de dos funciones:

Función	Significado
void glScaled (GLdouble x, GLdouble y, GLdouble z)	Especifica el factor de escala en el eje que se quiera aplicar, usando parámetros del tipo GLdouble.
void glScalef (GLfloat x, GLfloat y, GLfloat z)	Especifica el factor de escala en el eje que se quiera aplicar, usando parámetros del tipo GLfloat.

La rotación se determina en la cantidad de grados en la que ha de girar la figura sobre su propio eje, en un modelo compuesto de varios vértices el tamaño del ángulo debe ser constante sobre todos los vértices que lo componen.

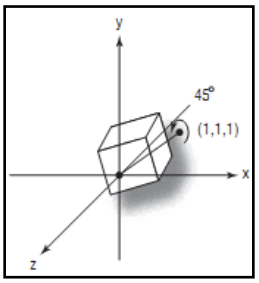


Figura 3.19. La rotación.

A diferencia de las anteriores transformaciones, la rotación se define en tres matrices dependiendo en el eje que se quiera rotar. Las definiciones de las matrices son las siguientes:

$$\begin{bmatrix} X \\ Ry \\ Rz \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\measuredangle) & -sen(\measuredangle) & 0 \\ 0 & sen(\measuredangle) & \cos(\measuredangle) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .*. \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Rotar X

$$\begin{bmatrix} Rx \\ Y \\ Rz \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\measuredangle) & 0 & sen(\measuredangle) & 0 \\ 0 & 1 & 0 & 0 \\ -sen(\measuredangle) & 0 & \cos(\measuredangle) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .*. \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Rotar Y

$$\begin{bmatrix} Rx \\ Ry \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\measuredangle) & -sen(\measuredangle) & 0 & 0 \\ sen(\measuredangle) & \cos(\measuredangle) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .*. \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Rotar Z

La operación de rotación se logra por medio de dos funciones:

Función	Significado
void glRotated (GLdouble angulo, GLdouble x, GLdouble y, GLdouble z)	Produce una rotación de ángulo en grados alrededor del vector xyz, usando parámetros del tipo GLdouble.
void glRotatef (GLfloat angulo, GLfloat x, GLfloat y, GLfloat z)	Produce una rotación de ángulo en grados alrededor del vector xyz, usando parámetros del tipo GLfloat.

Para activar las transformaciones, se debe primero activar la matriz de modelado con las siguientes llamadas:

- *glMatrixMode(GL_MODELVIEW);*
- *glLoadIdentity();*

Estas llamadas, se deben de seguir en este orden, la primera llamada le dice a OpenGL que va usar la matriz de modelado y la siguiente instrucción la carga. Es muy importante saber llamar la función de *glLoadIdentity(),* ya que también sirve para resetear la matriz y si la llamamos después de hacer una transformación no se verán los cambios.

El ejemplo ubicado en la carpeta "04-Transformaciones" del CD muestra las mencionadas transformaciones aplicadas a un cubo; presionando las teclas E (Escalar), R (Rotar) y T (Trasladar) se activan las transformaciones; con las teclas X, Y y Z se selecciona el eje de acción y con las teclas + y – se manipulan los valores.

Las nuevas funciones usadas en este programa son:

- Función *glutSwapBuffers()*(Línea 40): Al utilizar la técnica de doble buffer (línea 161), tenemos que llamar siempre a la función glutSwapBuffers() al final del pintado de cada *frame*, para que vuelque el buffer de escritura en el buffer de visualización; es muy útil en animaciones.

Y por último se implementan tres nuevas funciones personalizadas en el archivo Librería.c y que servirán para los próximos ejemplos, éstas son:

- Función *cuadradoColor()*(Línea 37): Dibuja un cuadrado con colores de dos unidades divididas entre los ejes positivos y negativo de X, Y y Z.

- Función *dibujarEjes()*(Línea 28 y 36): Dibuja los tres ejes, X de color azul, Y de color verde y Z de color rojo.

- Función *centrarVentana()(*Línea 80 y163): Centra la ventana de la aplicación, a través de una serie de cálculos donde se necesitan la anchura y altura, tanto del monitor como el de la ventana de la aplicación, por medio de la función *glutGet*(Constante).

Al compilarlo y ejecutarlo se debe ver como la siguiente figura:

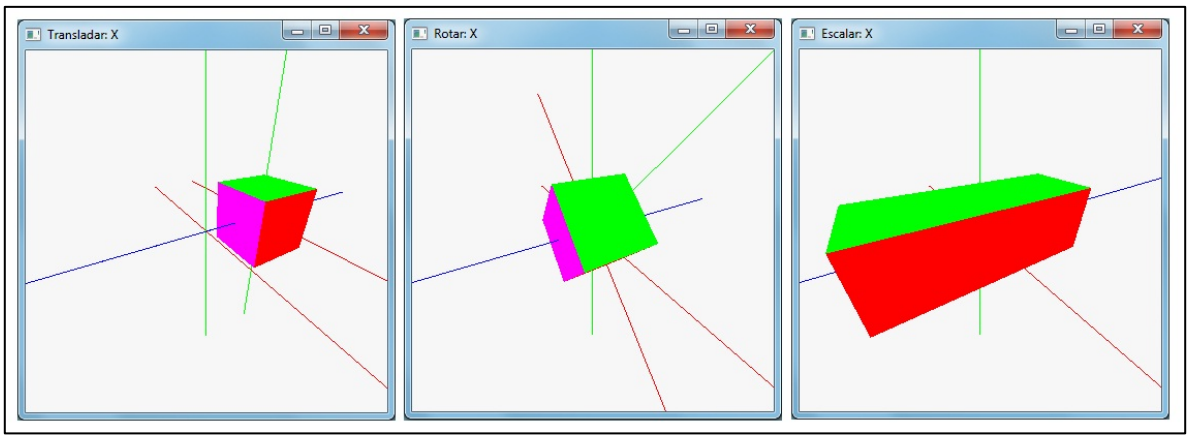


Figura 3.20. Transformaciones aplicadas a una figura.

● Pila de Matrices

Uno de los problemas que se encuentra al realizar escenas complejas con varios modelos, es la aplicación de las transformaciones. Supongamos que tenemos dos objetos y uno de estos queremos que se quede inmóvil y que el otro rote sobre el eje Y, por intuición dibujaríamos primero el modelo inmóvil y después aplicaríamos las transformaciones de traslado y rotación en Y para dibujar el siguiente modelo; sin embargo el problema reside en que cada vez que se llame la función de dibujado para que aplique los nuevos cambios, a la matriz de modelado se le vuelven a aplicar las mismas transformaciones; esto es, se le vuelve a multiplicar las mismas transformaciones de translación y rotación, logrando afectar a los dos modelos.

Para comprender este problema, debemos entender que nuestra matriz de modelado funciona como una pila, por cada transformación que se haga entrará a la pila como la última y saldrá como la primera. Podemos guardar el estado de la pila y recuperarlo por medio de dos funciones:

Función	Significado
void glPushMatrix()	Realiza una copia de la matriz superior y la pone encima de la pila, de tal forma que las dos matrices superiores son iguales
void glPopMatrix()	Elimina la matriz superior, quedando en la parte superior de la pila la matriz que estaba en el momento de llamar a la función glPushMatrix().

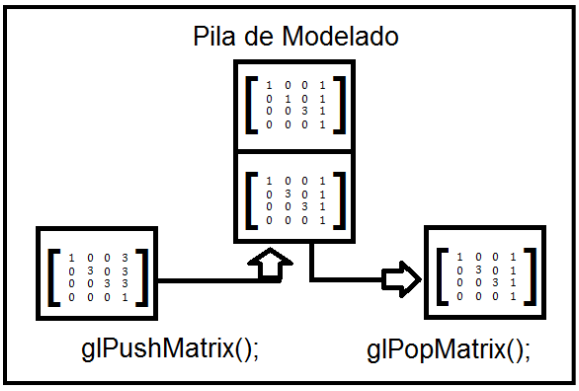


Figura 3.21. Funciones de pila.

En la figura 3.21 se puede observar como con la función *glPushMatrix()* introduce a la pila de modelado una copia con las transformaciones hechas y a partir de esta copia trabajarán los modelos nuevos que deseamos transformar sin alterar a los otros objetos que trabajan sobre la primera matriz de la pila. Al momento de llamar glPopMatrix() se saca la última matriz de transformación, quedando nuestra primera matriz original. La pila de modelado contiene al menos 32 matrices de 4x4. El código de la carpeta "05-Pila Matrices" ejemplifica el uso de las funciones de pila; al compilarse se muestra como en la figura 3.22; para activar y desactivar la pila se debe presionar la tecla "P".

Las nuevas funciones usadas son las siguientes:

- Función *glutGet(GLUT_ELAPSED_TIME)*(Línea 27): Devuelve en milisegundos el tiempo transcurrido desde la llamada de glutinit(línea 80).

- Función *glutPostRedisplay()*(Línea 70 y 75): Esta función fuerza un repintado usando la función dibuja.

- Función *glutIdleFunc(idle)*(Línea 90): Este *callback* llama a la función *idle* cuando no hay eventos y ésta a su vez llama a *gluPostRedisplay()*, haciendo una ilusión de movimiento.

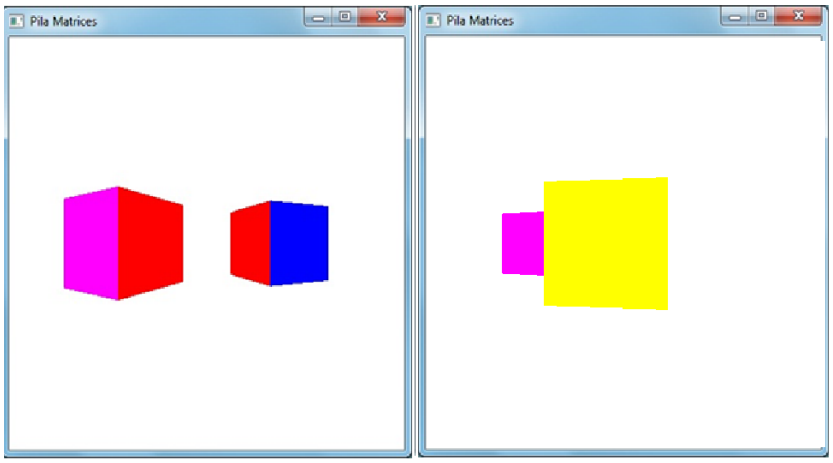


Figura 3.22. Pila activa (izquierda), pila inactiva (derecha).

● La Proyección

El concepto de proyección y sus distintos tipos fueron mencionados en el punto 3.1 de este capítulo, pero de forma general se puede decir que la proyección es la técnica de dibujo usada para representar un objeto tridimensional en dos dimensiones, partiendo de un punto de vista o posicionamiento de una cámara apuntada al objeto.

Para lograr esta ilusión se necesitan de algunos componentes, por ejemplo el ángulo de visión, el punto de vista, sombras e incluso el color. La proyección define el volumen del espacio que va a utilizarse para plasmar los gráficos deseados. OpenGL trabaja con dos tipos de proyección: proyección ortogonal y proyección de perspectiva.

La proyección ortogonal define un volumen de la vista cuya geometría es la de un rectángulo o "caja", la distancia de un objeto a la cámara no influye en el tamaño final del mismo en la imagen. Este tipo de proyecciones son comunes en aplicaciones de diseño asistido por computadora, ya que las medidas sobre la imagen son proporcionales a las reales de acuerdo a un determinado factor de escala. La proyección ortogonal se define de la siguiente forma:

void glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)

Donde los parámetros *left, right, bottom, top, zNear* y *zFar* definen la caja tal y como se muestra en la figura 3.23.

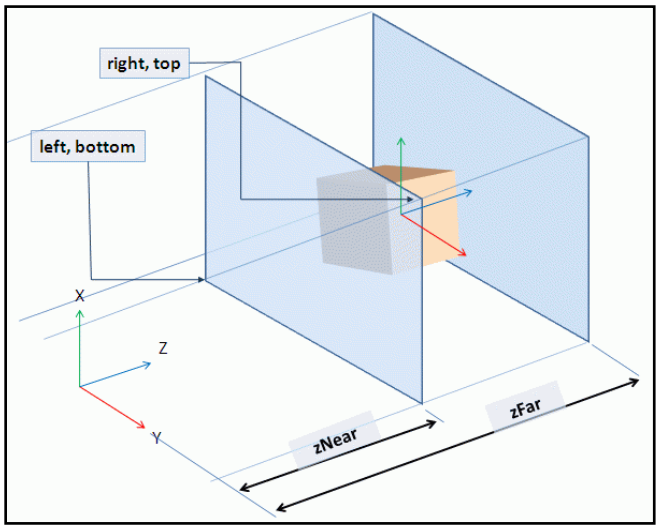


Figura 3.23. "Caja" de visualización en la proyección ortogonal.

La matriz de proyección ortogonal está definida de la siguiente forma:

$$\begin{bmatrix} \frac{2}{derecha - izquierda} & 0 & 0 & -\frac{derecha + izquierda}{derecha - izquierda} \\ 0 & \frac{2}{arriba - abajo} & 0 & \frac{arriba + abajo}{arriba - abajo} \\ 0 & 0 & \frac{-2}{\text{lejos} - \text{cerca}} & -\frac{\text{lejos} + \text{cerca}}{\text{lejos} - \text{cerca}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

En la proyección de perspectiva, el volumen de espacio es una pirámide truncada o *frustum*. Por lo tanto, en esta proyección se produce un efecto tamaño-distancia (los objetos aparecen más pequeños cuanto más alejados están del punto de vista) y es la proyección más usada en animación por computadora o simulación visual, donde se requiere un alto grado de realismo. En OpenGL este tipo de proyección se define en dos funciones, la primera función es de la siguiente forma:

void glFrustum (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)

Donde los parámetros *left, right, bottom, top, near* y *far* definen el *frustum* donde near como far no pueden ser negativos tal y como se muestra en la figura 3.24.

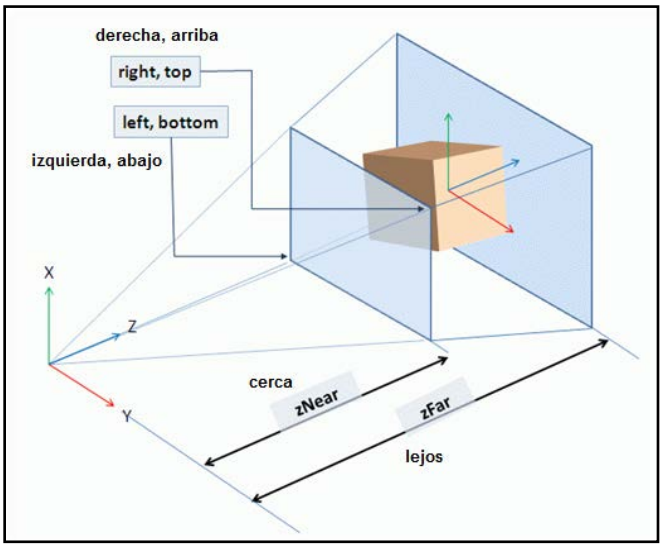


Figura 3.24. Proyección de perspectiva truncada (*frustum).*

El *frustum* no tiene por qué ser simétrico respecto al eje Z, ya que es posible utilizar valores distintos para left y right, o para bottom y top. La matriz de proyección *frustum* está definida de la siguiente forma:

$$\begin{bmatrix} \frac{2 * cerca}{derecha - izquierda} & 0 & \frac{derecha + izquierda}{derecha - izquierda} & 0 \\ 0 & \frac{2 * cerca}{arriba - abajo} & \frac{arriba + abajo}{arriba - abajo} & 0 \\ 0 & 0 & \frac{-\text{lejos} + \text{cerca}}{\text{lejos} - \text{cerca}} & \frac{-2 * lejos * cerca}{lejos - cerca} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

La proyección mediante *glFrustum* puede resultar complicada debido a que la forma de definición no resulta intuitiva; en lugar de esta orden, podemos utilizar una rutina de la librería de utilidades de OpenGL *gluPerspective*. Esta rutina permite especificar el volumen de la vista de forma diferente, utilizando el ángulo de visión sobre el plano XZ (fovy) y el radio de la anchura respecto a la altura (aspect). Mediante estos parámetros es posible determinar el volumen de la vista, como se muestra en la siguiente figura:

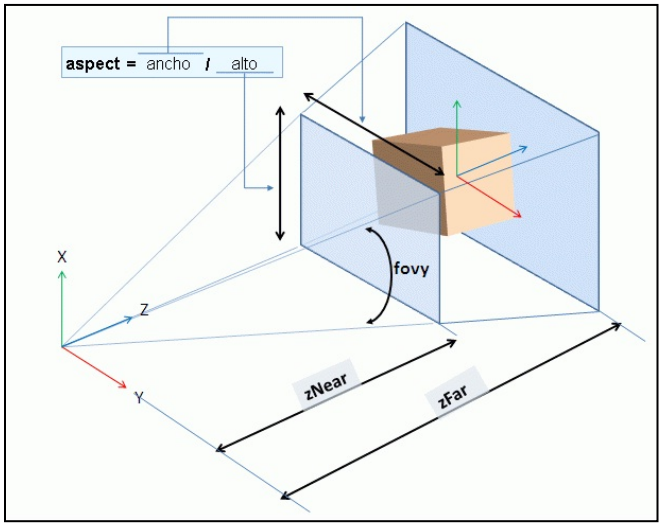


Figura 3.25. Proyección de perspectiva con *gluPerspective.*

Esta forma de definir la proyección resulta más intuitiva. La sintaxis de la rutina *gluPerspective* es la siguiente:

void gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far)

Cuando utilicemos la rutina *gluPerspective* tendremos que tomar precauciones con respecto a los valores de *fovy* y *near*, ya que no pueden tomar valores negativos. La matriz de proyección *gluPerspective* está definida de la siguiente forma:

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{\text{zFar} + \text{zNear}}{\text{zNear} - \text{zFar}} & \frac{2 * zFar * zNear}{\text{zNear} - \text{zFar}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Donde f=cotangente(fovy/2).

Otra forma de definir la manera en que la cámara observa una escena es mediante dos posiciones en el espacio: la del punto de vista y la dirección de interés a mirar, algo que es mucho más natural. Esta forma de definir la vista es la

que utiliza la orden de la librería de utilidades de OpenGL *gluLookAt*. La sintaxis de la orden es la siguiente:

gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);

Donde los parámetros eyex, eyey, eyez indican la posición del punto de vista; centerx, centery, centerz especifican el punto de interés, aunque generalmente se suele referir al punto central de la escena que estamos definiendo y upx, upy, upz la dirección tomada como "arriba" o inclinación de la cámara. Un dato muy importante es que este comando nos permite movernos pero sin mover la escena o el objeto.

Para manejar las proyecciones antes mencionadas, se inicializa la proyección en OpenGL, esto significa que primero debemos de seleccionar la matriz de proyección y cargarla por medio de las siguientes llamadas:

- *glMatrixMode(GL_PROJECTION)*

- *glLoadIdentity()*

- *Tipo de proyección(glfrustum, glortho).*

Sin embargo en la proyección el programador se salta dos procesos que son el *clipping* y el proceso de normalización de coordenadas. Estos procesos se realizan de forma transparente en el pipeline de nuestra tarjeta gráfica.

El proceso de *clipping*, es el recortado de todos los objetos gráficos que se quedaron fuera de la escena o del volumen definidos en los distintos tipos de proyecciones.

El proceso de normalización de coordenadas significa que todos nuestros vértices en el plano tridimensional se convertirán en coordenadas homogéneas las cuales son un instrumento usado para describir un punto en el espacio proyectivo.

Por último se debe definir el *viewport*. El *viewport* es la porción de área definida en la que se mostrará la escena realizada. Esta área se accede de acuerdo a las prestaciones que nuestro dispositivo de salida contempla. Está en manos del programador determinar qué área en pixeles soporta nuestro monitor o cualquier dispositivo que tenga una salida. Por defecto, al crear una ventana se inicializa un *viewport* que ocupa toda el área gráfica disponible; es posible cambiar este valor mediante la rutina OpenGL:

void glViewport (GLint x, GLint y, GLsizei width, GLsizei height);

El radio del *viewport (width / height)* debería ser normalmente equivalente al definido en la proyección utilizada. Existen aplicaciones en las que se divide la ventana en distintos *viewports* para mostrar una vista distinta en cada uno de ellos. Si queremos mantener la ventana de proyección y el *viewport* proporcionales será necesario detectar los eventos de ventana (fundamentalmente el cambio de tamaño) y redefinir la proyección y el *viewport* de forma adecuada. La figura 3.26 resume todo el proceso de proyección.

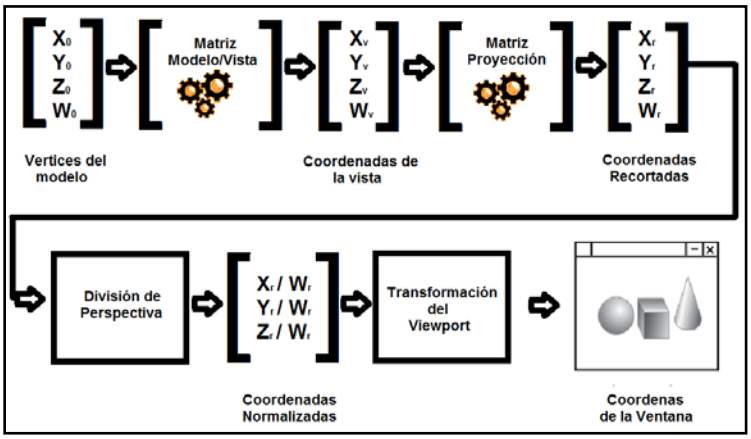


Figura 3.26. Transformación de los vértices en el Pipeline.

Los siguientes dos programas hacen uso de todas las funciones antes descritas, el primer programa contiene los modos de proyección y el segundo hace la simulación de una cámara, utilizando los tipos de proyección. Para ambos programas, una vez compilados y ejecutados se muestran los distintos tipos de proyección y cámara al presionar las teclas F (frustum), O (ortogonal) y P (perspectiva), como se aprecia en la figura 3.27. El código para el primer programa se encuentra en la carpeta "06-Proyeccion" del CD.

Las nuevas funciones utilizadas son las siguientes:

- Función *glutFullScreen()*(Línea 99): Esta función hace una petición para que la ventana actual de la aplicación ocupe todo el fondo.

- Función *pirámide()*(Línea 44): Esta función dibuja una pirámide definida en al archivo Librería.c.

- Función *cuadrado()*(Línea 35): Esta función dibuja un cubo con las mismas proporciones que la función *cuadradoColor()*, salvo que ésta no define un color específico.

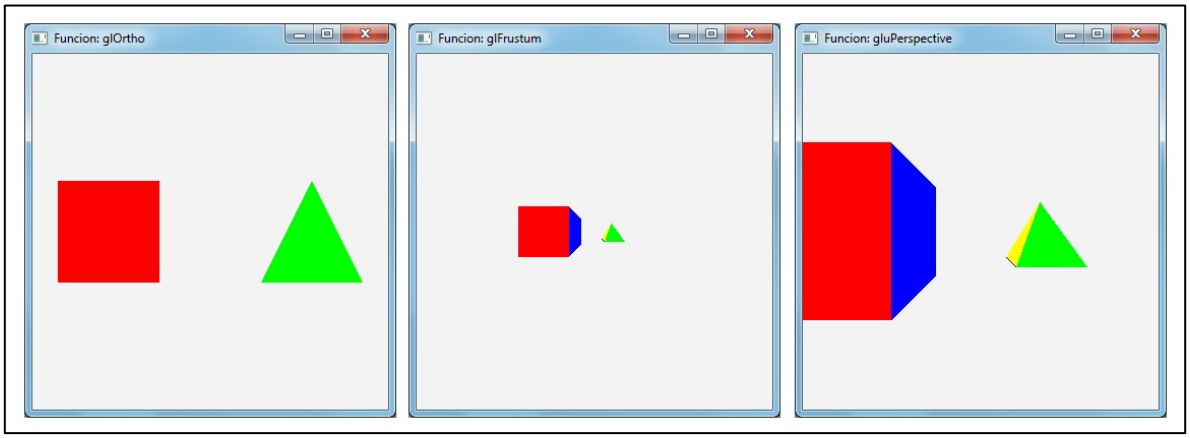


Figura 3.27. Tipos de proyección.

El programa de la carpeta "07-Camara" contiene las bases para crear efectos de cámara, utilizando funciones trigonométricas para calcular los nuevos puntos para colocarla. La figura 3.28 lo muestra una vez compilado y ejecutado.

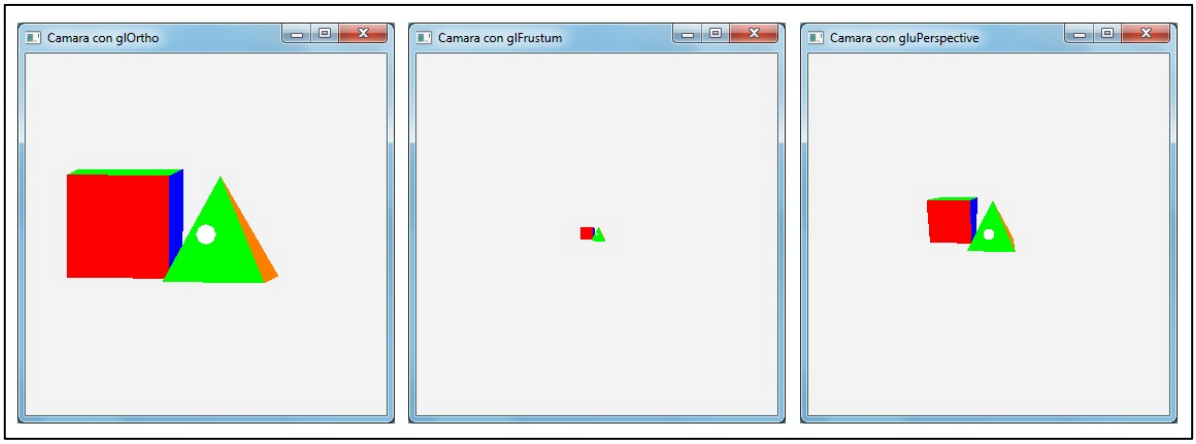


Figura 3.28. Funciones de cámara.

● Primitivas tridimensionales

Ya vimos que a partir de las primitivas, se pueden realizar figuras más complejas, sin embargo, para realizarlas hay que definirlas vértice por vértice, haciendo la tarea demasiado engorrosa. Este problema se puede resolver usando las formas geométricas ya predefinas que contiene la librería FreeGLUT. La siguiente tabla muestra dichas formas 3D:

Primitiva 3D	Descripción
glutSolidCone (GLdouble base, GLdouble height, GLint slices, GLint stacks)	Dibuja un cono sólido.
glutSolidCube (GLdouble width)	Dibuja un cubo sólido centrado en el origen.
glutSolidCylinder (GLdouble radius, GLdouble height, GLint slices, GLint stacks)	Dibuja un cilindro sólido.
glutSolidSphere(GLdouble radius, GLint slices, GLint stacks)	Dibuja una esfera sólida.
glutSolidTeapot(GLdouble size)	Dibuja una tetera sólida.
glutSolidTetrahedron(void)	Dibuja un tetraedro sólido.
glutSolidTorus(GLdouble dInnerRadius, GLdouble dOuterRadius, GLint nSides, GLint nRings)	Dibuja un *torus* (anillo) sólido.
glutWireCone (GLdouble base, GLdouble height, GLint slices, GLint stacks)	Dibuja un cono de alambre.
glutWireCube (GLdouble width)	Dibuja un cubo de alambre centrado en el origen.
glutWireCylinder (GLdouble radius, GLdouble height, GLint slices, GLint stacks)	Dibuja un cono de alambre.
glutWireSphere(GLdouble radius, GLint slices, GLint stacks)	Dibuja una esfera de alambre.

glutWireTeapot(GLdouble size)	Dibuja una tetera de alambre.
glutWireTetrahedron(void)	Dibuja un tetraedro de alambre
glutWireTorus(GLdouble dInnerRadius, GLdouble dOuterRadius, GLint nSides, GLint nRings)	Dibuja un *torus* (anillo) de alambre.

El programa contenido en la carpeta "08-Primitivas 3D" del CD, usa las funciones de la tabla para dibujar las figuras tridimensionales predefinidas. En la figura 3.29 puede verse el programa ya ejecutado; para cambiar entre figuras se deben usar las teclas "derecha" e "izquierda" del teclado.

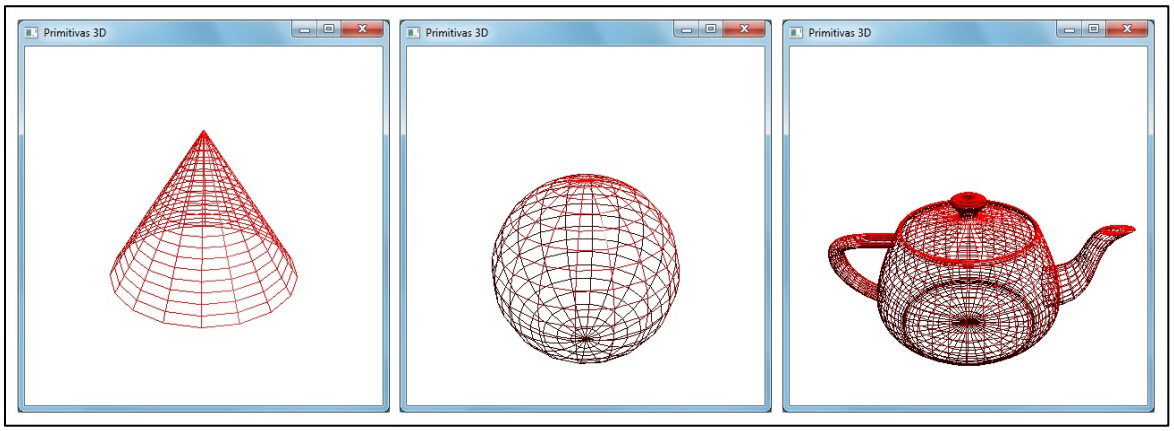


Figura 3.29. Figuras 3D predefinidas en FreeGLUT.

● Iluminación

La iluminación es una técnica que nos permite crear escenas más realistas al simular efectos de luz sobre un objeto. La iluminación de OpenGL se basa en luces y materiales; una luz es una fuente de iluminación para la escena, emite un haz de luz de un color determinado, dividido en las tres componentes de color RGB; el material determina la cantidad de luz que absorbe y refleja un objeto determinado.

Una fuente puede emitir diferentes tipos de luz, que son complementarias y pueden darse a la vez para un cierto tipo de luz. Los tipos de luz son los siguientes:

- "Emitted" (emitida): es la luz emitida por un objeto y no se ve afectada por ningún otro tipo de luz. Por ejemplo, un fuego emite una determinada luz, independientemente del color de las luces que estén apuntando a él.

- "Diffuse" (difusa): es la luz que incide sobre un objeto y proviene de un determinado punto. La intensidad con la que se refleja en la superficie del objeto puede depender del ángulo de incidencia, dirección, etcétera.

- "Specular" (especular): es la luz que al incidir sobre un objeto, se ve reflejada con un ángulo similar al de incidencia. Se puede decir que es la luz que producen los brillos.

- "Ambient" (ambiental): Se considera como los restos de luz que aparecen debido a la refracción residual de una luz que ya se ha reflejado sobre muchos objetos. Es la iluminación global de una escena.

Un material define para un determinado objeto, qué componentes refleja de cada tipo de luz. Por ejemplo, un plástico rojo emitirá toda la componente roja de las luces ambiental y difusa, pero generalmente emitirá brillos de color blanco bajo una luz blanca, por lo que su componente especular será de color blanco; de este modo la componente de emisión será negra puesto que un plástico no emite luz.

La iluminación en OpenGL se calcula mediante vértices; es decir, por cada vértice se calcula su color a partir del material activo, las luces activas y cómo estas luces inciden sobre el vértice.

En la carpeta "09-Iluminacion" del CD se incluye un ejemplo que usa diferentes tipos de luz para iluminar un objeto, que al ser compilado se muestra como en la figura 3.30. Presionando la tecla "a" se activa la luz ambiental, con la tecla "d" la luz difusa, con "e" la luz especular y con las teclas "x", "y", "z" la posición de la luz.

Las funciones nuevas usadas para este programa son las siguientes:

- Función *glEnable(GL_LIGHTING)* (Línea 23): Activa la función de iluminación, una vez activada, debemos establecer las propiedades de cada luz en la escena, y activarlas.

- Función *glEnable(GL_LIGHT0)* (Línea 24): Activa la luz, donde 0 representa el número de luz de un total de ocho que se pueden definir.

- Función *glColor3d(GLdouble red, GLdouble green, GLdouble blue)*(Línea 34): Define un color sólido, con el que se van a pintar los objetos en la escena.

- Función *glLightfv(GL_LIGHT0, GL_[AMBIENT|DIFFUSE|SPECULAR],…)* (Línea 42): Define el color ambiental, difuso y especular de la luz.

- Función *glLightfv(GL_LIGHT0, GL_POSITION, luzPosicion)* (Línea 45): Aquí se define la posición de la luz y su dirección será determinada por los ejes X, Y y Z.

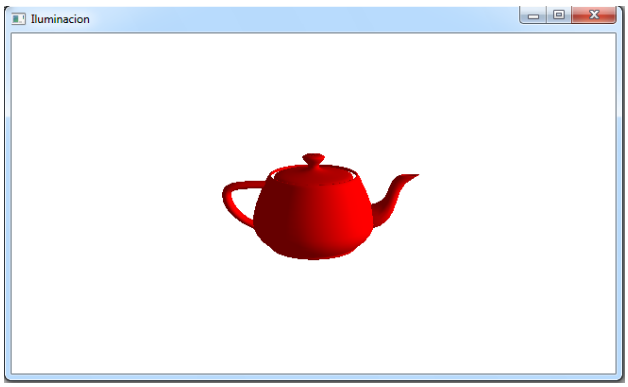


Figura 3.30. Efectos de luz sobre un objeto.

● Texturas

Las texturas son imágenes que aplicadas a un modelo permiten obtener un detallado más realista en nuestras escenas. Por ejemplo, si queremos dibujar una pared de ladrillo, tenemos dos opciones: dibujar cada ladrillo, definiendo su material con colores y dibujar el cemento entre los ladrillos a base de polígonos o dibujar un único cuadrado con la extensión de la pared y hacer que su material sea por ejemplo la foto de una pared de ladrillo.

Para saber qué partes de la textura se dibujan en el polígono hay que mapear la imagen, esto es, asociar los vértices de la imagen con los del objeto (bidimensional o tridimensional), por ejemplo, si queremos mapear nuestra imagen con un triángulo, utilizamos lo que se denomina coordenadas de textura (véase capítulo 3.1), estas coordenadas están representadas por números reales de 0 a 1.

Para aplicar una textura tenemos que seguir una serie de pasos muy definidos:

1. Cargar la imagen desde algún lugar de almacenamiento, lo que implica leerla desde algún tipo de formato (png, bmp, jpg, gif, etcétera) y ponerla en memoria (array de datos). Para este ejemplo cargaremos una imagen del tipo BMP, ya que es un formato sin compresión y no tendríamos que implementar el algoritmo de descomprensión para otro formato (png, jpg, gif).

2. Configurar las características de la textura, es decir, en la forma que OpenGL debe interpretarla y mostrarla.

3. Asociar la textura con las primitivas que se van a mapear.

El código que se muestra en la carpeta "10-Texturas" emplea las funciones de textura en un cubo, al compilarlo debería verse como en la figura 3.32.

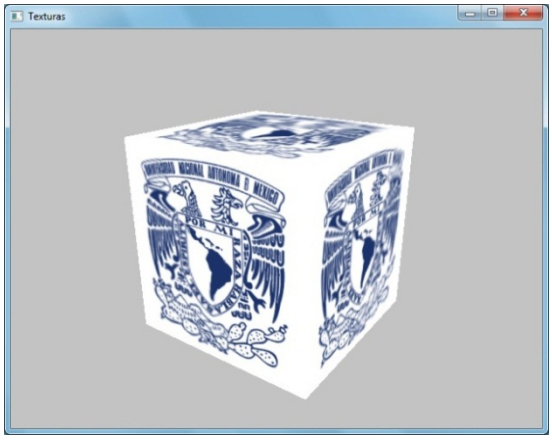


Figura 3.32. Textura aplicada a un cubo.

El primer paso consiste en cargar la imagen en memoria, el programa se auxilia de la función *cargarImagen (*definida en la Libreria.h), esta función por medio de *fseek* y *fread* sitúa el cursor para leer el encabezado del archivo que se usará para crear la textura correspondiente y de acuerdo a sus valores determinará si el archivo es válido para ser cargado. El archivo BMP (como cualquier otro), contiene información en sus encabezados, esta información está representada en bytes y de acuerdo a ésta, nuestra función procederá a la lectura del archivo para posteriormente cargarla en memoria. La siguiente tabla contiene la información necesaria para leer correctamente los encabezados de un archivo en formato BMP:

Cabecera (bytes)	Información
0, 1	Tipo de fichero "BM"
2, 3, 4, 5	Tamaño del archivo
6, 7	Reservado
8, 9	Reservado
10, 11, 12, 13	Inicio de los datos de la imagen
14, 15, 16, 17	Tamaño de la cabecera del *bitmap*
18, 19, 20, 21	Anchura (pixeles)
22, 23, 24, 25	Altura (pixeles)
26, 27	Número de planos
28, 29	Tamaño de cada punto
30, 31, 32, 33	Compresión (0=no comprimido)
34, 35, 36, 37	Tamaño de la imagen
38, 39, 40, 41	Resolución horizontal
42, 43, 44, 45	Resolución vertical
46, 47, 48, 49	Tamaño de la tabla de color
50, 51, 52, 53	Contador de colores importantes

El segundo paso es configurar la textura, para lograrlo OpenGL proporciona las siguientes funciones:

- Función *glGenTextures(GLsizei n, GLuint *textures)*(Línea 28): Nos permite asignarle un identificador a la textura cuando está cargada en memoria, donde 'n' especifica el número de texturas que se van a nombrar y ***texturas*** es el puntero a un *array* donde se va almacenar los nombres; naturalmente son llamadas por números, donde el número está reservado y no se puede usar como identificador.

- Función *glBindTexture(enum target, int texture)*(Linea 29): Esta función asocia en cual textura se van a realizar las cambios que se hagan en ella. Donde *target* es el tipo de textura que vamos a utilizar, que en este caso es una textura 2d y *texture* es el identificador o nombre de la textura.

- Función *glTexParameteri(GLenum target, GLenum pname, GLfloat param)*(Línea 30): Esta función nos permite aplicar los filtros necesarios cuando el objeto al que vamos a mapear es muy grande o muy pequeño, obteniendo una textura muy pixelada o sobrecargada, de acuerdo al tipo de filtro aplicado se verá el rendimiento del programa.

- Función *glTexImage2D(GLenum tipoTextura, GLint nivelMipMap, Glint formatoInterno , GLsizei ancho, GLsizei alto ,GLint borde, GLenum formato , GLenum tipo ,const GLvoid *pixels)* (Línea 31): Nos permite pasar los datos de la imagen cargada en memoria a la textura; no se debe confundir la imagen con la textura ya que la textura además de contener los bits de la imagen también contiene las características de cómo se va a plasmar en el objeto que va a ser mapeado. Los parámetros para esta función son:

- *tipoTextura*: Define la textura que estamos tratando.

- *NivelMipMap*: Indica el nivel de *MipMapping* que deseemos. De momento pondremos '0'.

- *FormatoInterno*: Es el número de componentes en la textura. Es decir, si estamos trabajando en formato RGB, el número de componentes será 3.

- *Ancho, alto*: El ancho y alto de la textura. Deben ser potencias de 2.

- *Borde*: La anchura del borde, puede ser 0.

- *Formato*: Indica el formato en que esta almacenada la textura en memoria.

- *Tipo*: Indica el tipo de variable en la que tenemos almacenada la textura, si la hemos almacenado en un *unsigned char*, usaremos *GL_UNSIGNED_BYTE*.

- *Pixels*: El puntero a la región de memoria donde esté almacenada la imagen.

Por último se mapea el objeto por medio de las siguientes funciones:

- Función *glTexCoord2f(0.0f, 0.0f)*: Define las coordenadas de mapeo de textura, una por cada vértice que asociemos con el objeto (Figura 3.31).

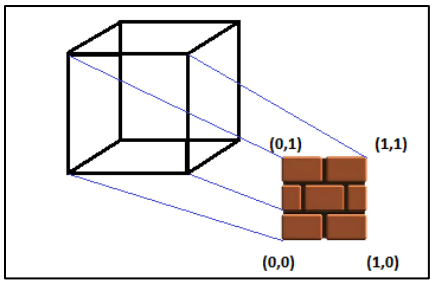


Figura 3.31. Cubo compuesto por 24 vértices mapeado por cada lado que lo compone.

- Función *glVertex3f(-1.0f, -1.0f, 1.0f)*: Dibuja en el espacio tridimensional el vértice que componen el objeto, en este caso un cubo. Por cada cara terminada se dibuja ya con la textura.

Con esto concluimos el capítulo 3; hemos repasado las características más importantes de los gráficos tridimensionales, pasando por la comparación entre los principales API's de diseño 3D: DirectX y OpenGL, para terminar con la programación de las funciones primordiales de OpenGL como lo son las primitivas, la proyección, la iluminación y las texturas. Tomando como base lo repasado anteriormente, para el siguiente capítulo se abordará por completo el tema principal de este trabajo: WebGL. .

CAPÍTULO 4

DESARROLLO DE APLICACIONES 3D USANDO LA TECNOLOGÍA WEBGL

En el capítulo dos se habló de forma general de las principales características de WebGL; por lo que en este capítulo se estudiará a profundidad esta tecnología, tomando como base los conceptos generales y de programación de OpenGL vistos en el capítulo anterior; como WebGL está basado en la versión simplificada de OpenGL (OpenGL ES), muchas de las funciones son muy similares a las empleadas en OpenGL.

Como se sabe WebGL es una API estándar para la creación de gráficos tridimensionales en un navegador web sin la necesidad de plugins para funcionar; el hecho que se trate de un estándar es muy importante, ya que brinda un conjunto de especificaciones técnicas que se encuentran en constante evolución y mejoras para construir sitios web. Con ello se facilita el mantenimiento, la usabilidad, la portabilidad y la calidad de los trabajos.

Al ser un estándar se le asegura una larga vida al proyecto; ya que garantiza por lo menos una pequeña estructura de mantenimiento, también se asegura que la mayoría de las personas puedan visitar un sitio web sin importar que navegador se esté utilizando, además claro, por el hecho de estar basado en OpenGL, otro estándar de gráficos tridimensionales ampliamente aceptado.

La habilidad para poner contenido 3D acelerado por hardware tan solo usando un navegador moderno, abre una infinidad de posibilidades de nuevas aplicaciones basadas en web que anteriormente solo podían verse en otras plataformas; desde juegos, demostraciones interactivas, recorridos virtuales e incluso aplicaciones científicas y educativas como la aplicación *Zygote Body* (anteriormente llamada *Google Body*), la cual, como puede verse en la figura 4.1, permite explorar el cuerpo humano de forma interactiva.

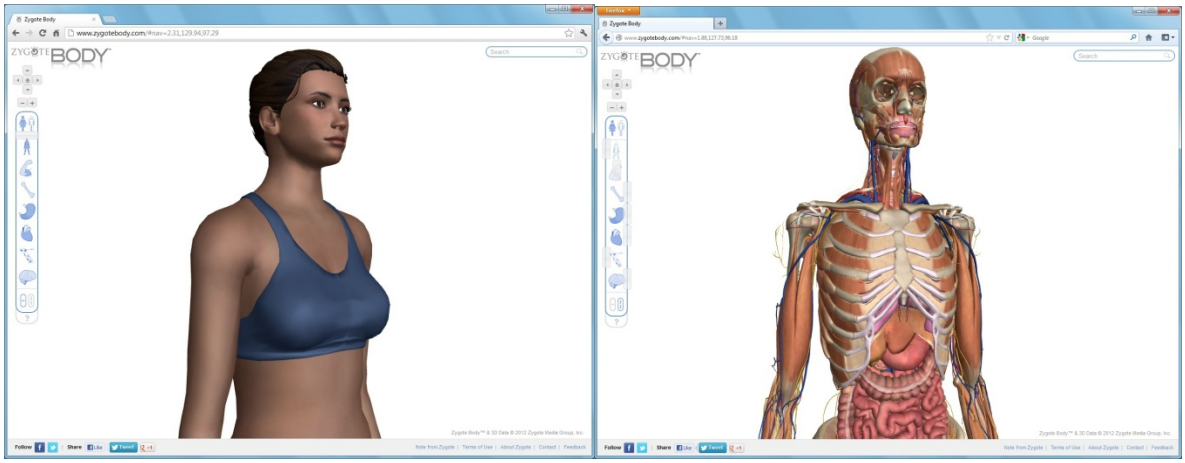


Figura 4.1. Modelo 3D del cuerpo humano en WebGL, ejecutándose en Chrome y Firefox.

Para los usuarios, esto significa una web más interactiva e interesante visualmente al tener acceso a una gama más amplia de aplicaciones y experiencias sobre cualquier dispositivo que soporte WebGL, en lugar de limitarse solo a ciertas plataformas.

A partir de aquí nos centraremos en el desarrollo de aplicaciones en WebGL, por lo que antes de empezar a usar y programar modelos haciendo uso de esta tecnología, necesitamos conocer algunos aspectos esenciales, como la forma en que está estructurado su funcionamiento interno, además de conocer algunos *Frameworks* desarrollados que facilitan el uso de ésta.

4.1. Estructura de WebGL

WebGL significa *Web-based Graphics Library* y es una API de gráficos 3D de bajo nivel basado en OpenGL ES 2.0 y expuesto a través del elemento *Canvas* de HTML5 con interfaces DOM (Document Object Model). Esta API provee enlaces de ECMAScript (JavaScript) a funciones OpenGL haciendo posible proveer contenido 3D acelerado en hardware a las páginas web, esto posibilita la creación de gráficos 3D que se actualizan en tiempo real, corriendo en el navegador.

A partir de la definición anterior, podemos decir que WebGL, es un “contexto” del elemento *Canvas* que provee una API de gráficos 3D; al decir que WebGL es un “contexto” del elemento *Canvas*, podría no quedar muy claro a que se esta haciendo referencia, por lo que a continuación se explica como es que trabajan juntos.

WebGL trae la API de OpenGL ES 2.0 al elemento *Canvas* de HTML5, el cual nos permite la generación de gráficos de forma dinámica, por lo que hay que indicarle en que contexto se quiere trabajar, ya sea en 2D o 3D (WebGL).

Podemos observar un modelo conceptual de la estructura de WebGL en la figura 4.2; en este modelo se utiliza JavaScript para obtener a través de DOM al elemento *Canvas* de HTML5.

Una vez obtenido el elemento *Canvas*, se define el contexto 3D por medio del cual accedemos a la API de WebGL, por eso se dice técnicamente que es un enlace (binding) a JavaScript para usar la implementación nativa de OpenGL ES 2.0.

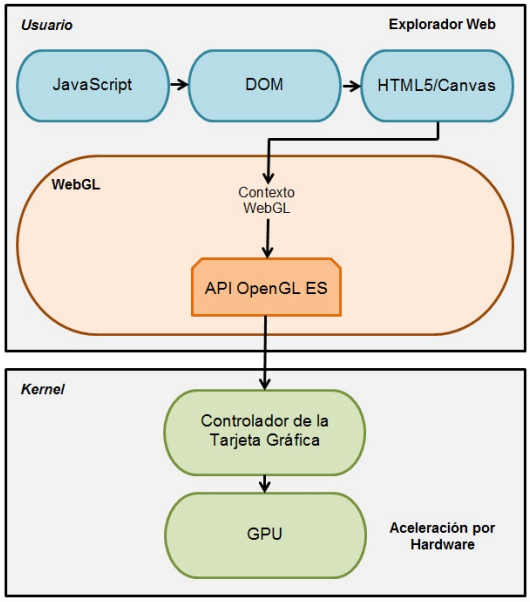


Figura 4.2. Estructura de WebGL.

Por último, OpenGL ES se encarga de comunicarse con el driver de la tarjeta gráfica y así poder realizar la aceleración por hardware en el GPU. Como se puede ver, se separan los eventos que ocurren en el espacio del usuario y del Kernel[24].

La API de WebGL interactúa bien con el resto de las plataformas web, específicamente se proporciona apoyo para la carga de texturas 3D a partir de imágenes y los eventos de entrada del teclado o mouse se manejan mediante DOM.

4.2. Programación en WebGL

Como sabemos, WebGL emplea OpenGL ES que es una interfaz de programación de aplicaciones (API) para gráficos avanzados 3D orientados a dispositivos portátiles como los teléfonos inteligentes, computadoras de bolsillo, consolas de videojuegos, automóviles, entre otros. Se trata de una versión simplificada del API gráfico de OpenGL.

Debido a la amplia adopción de OpenGL como una API 3D, tenía sentido empezar con la versión de OpenGL de escritorio en el desarrollo de un estándar abierto de API 3D para dispositivos portátiles y modificarla para satisfacer las necesidades y las limitaciones de espacio en dichos dispositivos portátiles.

La API de OpenGL es muy grande y compleja y el objetivo de OpenGL ES fue la creación de una API conveniente para solucionar las restricciones de los dispositivos portátiles, para lograr este objetivo, se eliminó cualquier redundancia, así en cualquier caso en que hubo más de una forma de realizar la misma operación, se tomaron las técnicas redundantes y se eliminaron. Las nuevas

[24] Término que hace referencia al núcleo de un Sistema Operativo, el cual es un software que representa la parte más importante del S.O. y se encarga de facilitar el acceso al hardware.

características fueron introducidas para hacer frente a las limitaciones específicas de los dispositivos integrados, por ejemplo, para reducir el consumo de energía y la limitación en el tamaño de la pantalla en algunos dispositivos. En la figura 4.3 se muestra el Pipeline de OpenGL ES, los recuadros *Vertex Shader* y *Fragment Shader* indican los procesos programables en el Pipeline con el lenguaje de sombreado.

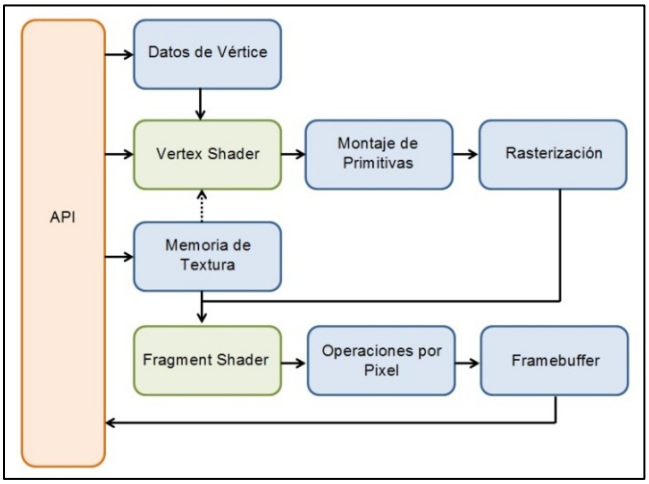


Figura 4.3. Pipeline gráfico de OpenGL ES.

La mayoría de los conceptos que se manejan ya se explicaron brevemente en la sección de “estructura de OpenGL” del capítulo tres, del cual también podemos observar la similitud entre ambos Pipelines, cambiando principalmente en la implementación de los shaders.

El manejo de los shaders se introdujo en el año 2001 con el chip GeForce3 de Nvidia, estos permitían a la GPU hacerse cargo de los cálculos de transformación e iluminación que hasta ahora debía hacerlas la CPU.

A partir de entonces se permitió a los programadores una mayor libertad a la hora de diseñar gráficos en tres dimensiones, ya que fue posible tratar a cada píxel y cada vértice por separado, gracias al cálculo paralelo de la GPU; de esta manera, los efectos especiales y de iluminación pueden crearse mucho más detalladamente, sucediendo lo mismo con la geometría de los objetos.

Al igual que en OpenGL, los shaders tienen también un lenguaje de programación específico para la escritura de estos, dicho lenguaje es conocido como *GLSL*, un lenguaje de alto nivel basado en C desarrollado por el *Khronos Group* y que fue integrado en OpenGL en su versión 1.20.

El lenguaje de Sombreado GLSL es muy similar al C estándar (ya que la sintaxis y las declaraciones de variables son las mismas), el propósito con el que se creó este lenguaje es el de optimizar las operaciones de cálculo de punto flotante que se realizan en la GPU, para esto, GLSL dispone de varios tipos de datos, así como sus calificadores de acceso. La siguiente tabla resume los diferentes tipos de datos que soporta GLSL:

TIPO DE DATO	DESCRIPCIÓN
float	Representa un flotante (Depende el rango de la precisión).
bool	Representa un booleano.
int	Representa un entero (Depende el rango de la precisión).
vec2	Vector de 2 del tipo float {x, y}, {r, g}, {s, t}.
vec3	Vector de 3 del tipo float {x, y, z}, {r, g, b}, {s, t, r}.
vec4	Vector de 4 del tipo float {x, y, z, w}, {r, g, b, a}, {s, t, r, q}.
bvec2	Vector de 2 booleanos {x, y}, {r, g}, {s, t}.
bvec3	Vector de 3 booleanos {x, y, z}, {r, g, b}, {s, t, r}.
bvec4	Vector de 4 booleanos {x, y, z, w}, {r, g, b, a}, {s, t, r, q}.
ivec2	Vector de 2 enteros {x, y}, {r, g}, {s, t}.
ivec3	Vector de 3 enteros {x, y, z}, {r, g, b}, {s, t, r}.
ivec4	Vector de 4 enteros {x, y, z, w}, {r, g, b, a}, {s, t, r, q}.
mat2	Arreglo de 4 del tipo float. Representa a una matriz de 2x2.
mat3	Arreglo de 9 del tipo float. Representa a una matriz de 3x3.
mat4	Arreglo de 16 del tipo float. Representa a una matriz de 4x4.
sampler2D	Tipo especial para acceder a una textura 2D.
samplerCube	Tipo especial para acceder a una textura de cubos (textura 3D).

En la tabla se puede observar que todos los tipos de datos vectoriales (vec*, bvec* e ivec*) contienen las secuencias {x, y, z, w}, {r, g, b, a}, {s, t, r, q}, esas secuencias son descriptores de acceso para los elementos del vector. Estos

elementos se pueden acceder utilizando un *punto* ".". Por ejemplo, xyz pueden representar a los tres primeros elementos de un vec4, pero no se puede utilizar xyz en un vec2, porque en un vec2 está fuera de límites. También puede cambiar el orden para lograr los resultados; por ejemplo, yzx de un vec4 donde está consultando los elementos segundo, tercero y primero, respectivamente. El siguiente ejemplo ayuda a entender los conceptos:

```
1  vec4 miVec4 = vec4(0.0, 1.0, 2.0, 3.0);
2  vec3 miVec3;
3  vec2 miVec2;
4
5  miVec3 = miVec4.xyz;        // miVec3 = {0.0, 1.0, 2.0};
6  miVec3 = miVec4.zzx;        // miVec3 = {2.0, 2.0, 0.0};
7  miVec2 = miVec4.bg;         // miVec2 = {2.0, 1.0};
8  miVec4.xw = miVec2;         // miVec4 = {2.0, 1.0, 2.0, 1.0};
9  miVec4[1] = 5.0;            // miVec4 = {2.0, 5.0, 2.0, 1.0};
```

En la tabla anterior también se puede apreciar que en algunas variables el valor de su precisión varía, esto se debe a que GLSL implementa un “Calificador de precisión”. Dependiendo de las capacidades de hardware de nuestro dispositivo, podemos definir el rango de nuestros datos, los calificadores de precisión que proporciona GLSL son los siguientes:

Precisión	Rango de punto flotante	Rango de números enteros
lowp	-2,0 A 2,0	-256 A 256
mediump	-16,384.0 A 16,384.0	-1024 A 1024
highp	-4,611,686,018,427,387,904.0 A 4,611,686,018,427,387,904.0	-65.536 A 65.536

Recordemos que estas variables son de uso exclusivo para la GPU, para que WebGL pueda acceder a ellas, GLSL implementa el uso de los calificadores de acceso, estos calificadores nos permiten acceder a las variables anteriores. El lenguaje GLSL contempla cuatro calificadores que se muestran a continuación:

CALIFICADORES DE ACCESO	
in	Para cambiar la información frecuente de la aplicación del *Vertex Shader* y leer los valores interpolados en un *Fragment Shader*.
uniform	Para cambiar la información poco frecuente de la aplicación a cualquier *Vertex Shader* o *Fragment Shader*.
out	Para pasar información interpolada del *Vertex Shader* al *Fragment Shader*, así como para la salida del *Fragment Shader*.
const	Para declaraciones no editables en tiempo de compilación variable, como en C.

Conociendo los tipos de variables que se manejan, así como sus calificadores de acceso, podemos definir a los Shaders como programas escritos en el lenguaje GLSL, que se pueden añadir al pipeline de procesamiento y permite añadir nuevas funcionalidades.

Los *Vertex Shader* se ejecutan en módulos especializados llamados "Procesadores de Vértices", estas unidades operan con los vértices que van llegando, así como sus datos asociados, de manera general podemos decir que los Vertex Shader calculan las operaciones de lo que se quiere dibujar. Para realizar todos estos cálculos, los procesadores de vértices contienen variables incorporadas, que nos servirán para el cálculo de las coordenadas de texturas, transformación de vértices, transformación y normalización de normales y aplicación del color de materiales; la siguiente tabla contiene las variables del procesador de vértices.

VARIABLE INCORPORADA	PRESICIÓN	TIPO DE DATO
gl_Position	highp	vec4
gl_FrontFacing	No aplica.	bool
gl_PointSize	mediump	float

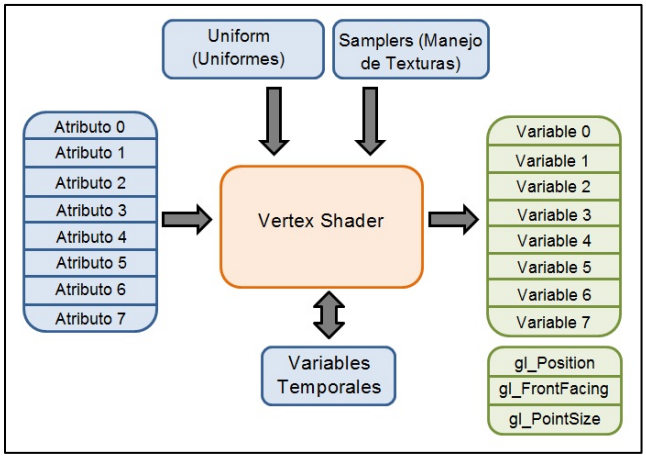


Figura 4.4. Procesador de vértices.

Por otra parte, los *Fragment Shader* se ejecutan en el procesador de fragmentos, su función principal es definir cómo se van a dibujar los modelos; los *Fragment Shaders* calculan el color, la iluminación, el zoom y la interpolación de colores. Los procesadores de fragmentos también tienen sus propias variables incorporadas las cuales se muestran a continuación:

VARIABLE INCORPORADA	PRESICIÓN	TIPO DE DATO
gl_FragColor	mediump	vec4
gl_FrontFacing	No aplica	bool
gl_FragCoord	mediump	vec4
gl_PointCoord	mediump	vec2

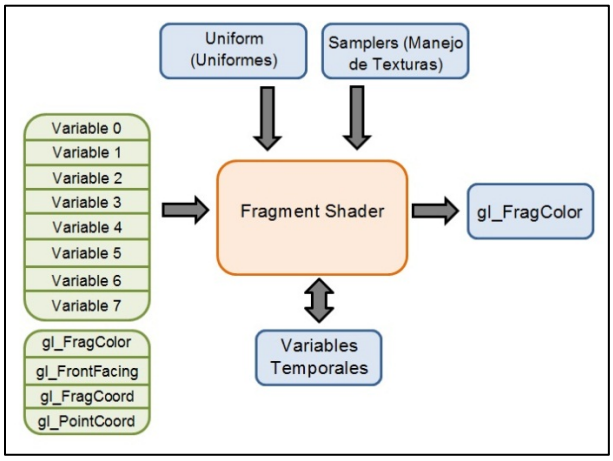


Figura 4.5. Procesador de fragmentos.

Para que los Shaders trabajen conjuntamente, GLSL implementó un contenedor llamado “Programa”, este programa se encarga de controlar los shaders y es posible crear tantos programas como se desee. La siguiente imagen resume el proceso de los Shaders, desde su creación hasta su ejecución.

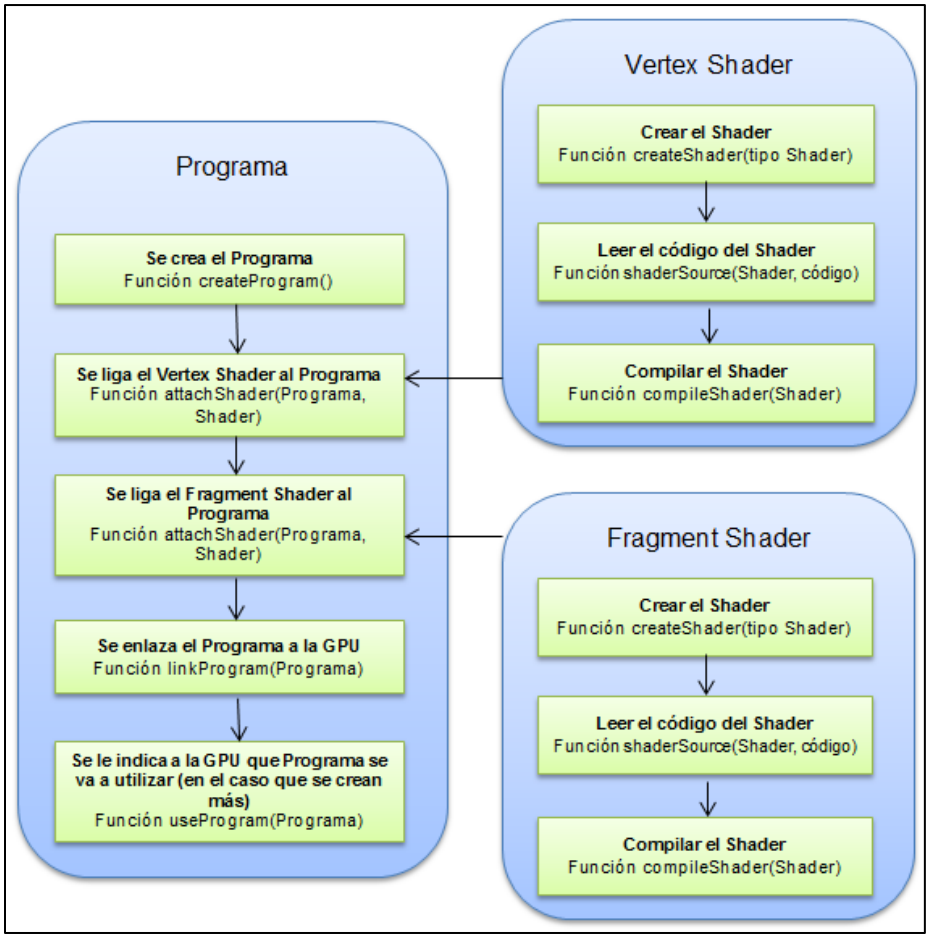


Figura 4.6. Proceso de generación de los shaders.

Para aplicar todos los conceptos antes mencionados, se dará una introducción a la programación en WebGL, por lo que debemos elegir un navegador en el que los ejemplos en WebGL funcionen adecuadamente. Como sabemos *Internet Explorer 9* solo soporta el estándar HTML5 pero no así a WebGL, por lo que éste queda completamente descartado. El navegador *Opera* soporta WebGL sólo en su versión 12 (que actualmente se encuentra en desarrollo), por lo que aún no es muy estable. *Safari* por su parte si tiene implementado WebGL desde la versión

5.1, pero viene desactivado por *default*, así que es necesario hacer un par de ajustes antes de poder usarlo, además de que su implementación de WebGL solo esta disponible para los sistemas operativos Mac OS X "Snow Leopard" y versiones superiores.

Dejando a dos navegadores como la mejor opción para ejecutar aplicaciones WebGL, estos son *Google Chrome* y *Mozilla Firefox*, ambos navegadores son una buena opción para trabajar, aunque es Google Chrome el que hace un mejor uso de los recursos de hardware, además de contar con una implementación más trabajada tanto de WebGL como de HTML5. La siguiente tabla comparativa muestra las características más importantes de los navegadores web.

	 FIREFOX	 CHROME	 OPERA	 SAFARI	 I.EXPLORER
Desarrollador	Mozilla	Google	Opera ASA	Apple	Microsoft
Sistema operativo	Múltiple	Múltiple	Múltiple	Mac OS y Windows	Windows
Motor de renderizado	Gecko	WebKit	Presto	WebKit	Trident
Última versión estable[25]	10.0.2	17.0.963	11.61	5.1.3	9.0.5
Soporte para HTML5	Si	Si	Si	Si	Si
Soporte para WebGL	Versión 4.0 o superior	Versión 9.0 o superior	Versión 12 (desarrollo)	Vesión 5.1 (sólo en Mac)	No

Para empezar a programar en WebGL nos auxiliaremos de las siguientes herramientas, las cuales nos servirán para el desarrollo y depuración de nuestras aplicaciones; recordemos que WebGL utiliza JavaScript, un lenguaje de programación interpretado que al contrario de los lenguajes compilados, los

[25] Últimas versiones estables correspondientes en fecha a Febrero de 2012.

posibles errores se mostrarán en tiempo de ejecución, haciendo la tarea de depuración muy difícil sin una herramienta especializada.

La primera herramienta es "*Firebug"*, una extensión para el navegador Firefox que nos servirá para monitorear y depurar el código fuente, CSS, HTML y JavaScript de una página web de manera instantánea. Se puede descargar de la siguiente URL:

```
http://www.getfirebug.com/
```

Para los demás navegadores no es necesario descargarla ya que implementan sus propias herramientas de depuración.

La segunda herramienta es "*WebGL Inspector"*, una extensión que por ahora solo está disponible para Google Chrome, ésta nos servirá para depurar y descomponer los elementos de una aplicación en WebGL. Se puede obtener de la siguiente URL:

```
http://benvanik.github.com/WebGL-Inspector/
```

Por último descargaremos la librería *glMatrix,* la cual nos ayudará en el cálculo y manejo de matrices usadas en las proyecciones y transformaciones. Se obtiene de la siguiente URL:

```
https://github.com/toji/gl-matrix
```

► Introducción

La primera aplicación que estudiaremos, simplemente dibuja un cuadrado en la pantalla del navegador y nos servirá de base para los demás ejemplos. Todos los códigos y ejemplos se encuentran dentro de la Carpeta "Ejemplos WebGL" del CD que acompaña a este trabajo. El código necesario se muestra a continuación:

```
 1 !DOCTYPE html>
 2 <html>
 3   <head>
 4     <title>WebGL</title>
 5     <meta  charset=ISO-8859-1">
 6     <script  src="src/funciones.js"></script>
 7     <script  src="src/glMatrix-0.9.5.min.js"></script>
 8     <link rel="stylesheet" type="text/css" href="css/hoja.css">
 9     <!-- Codigo del Fragment Shader -->
10   <script id="shader-fs" type="x-shader/x-fragment">
11     #ifdef GL_ES
12          precision highp float;
13     #endif
14     void main(void)
15     {
16         gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
17     }
18   </script>
19
20   <!-- Codigo del Vertex Shader -->
21   <script id="shader-vs" type="x-shader/x-vertex">
22     attribute vec3 aVertexPosition;
23     uniform mat4 uMVMatriz;
24     uniform mat4 uPMatriz;
25
26     void main(void)
27     {
28         gl_Position= uPMatriz * uMVMatriz *
29                       vec4(aVertexPosition, 1.0);
30     }
31    </script>
32
33   <script type="text/javascript">
34     var webgl;
35     var at = new atributosPipeline(mat4.create(),mat4.create());
36     var cuadro = new objeto3D([ 1.0,  1.0,  0.0,
37                                -1.0,  1.0,  0.0,
38                                -1.0, -1.0,  0.0,
39                                 1.0, -1.0,  0.0]);
40     //Funcion que inicia los shaders
41     function initShaders()
42     {
43         var fragmentShader = getShader(webgl, "shader-fs");
44         var vertexShader = getShader(webgl, "shader-vs");
45
46         at.ShaderProgram = webgl.createProgram();
47         webgl.attachShader(at.ShaderProgram, vertexShader);
48         webgl.attachShader(at.ShaderProgram, fragmentShader);
49         webgl.linkProgram(at.ShaderProgram);
50
51         if (!webgl.getProgramParameter(at.ShaderProgram, webgl.LINK_STATUS))
52                 alert("No pueden iniciarse los shaders");
53
54         webgl.useProgram(at.ShaderProgram);
55
56         at.ShaderProgram.vertexPositionAttribute =
57               webgl.getAttribLocation(at.ShaderProgram, "aVertexPosition");
```

```
 58
 59          webgl.enableVertexAttribArray(
 60                  at.ShaderProgram.vertexPositionAttribute);
 61
 62          at.ShaderProgram.pMatrizUniform =
 63               webgl.getUniformLocation(at.ShaderProgram, "uPMatriz");
 64
 65          at.ShaderProgram.mvMatrizUniform =
 66               webgl.getUniformLocation(at.ShaderProgram, "uMVMatriz");
 67      }
 68      //Funcion que inicia los buffers del objeto en la Tarjeta Grafica
 69      function initBuffers()
 70      {
 71          cuadro.bufferVertices=webgl.createBuffer();
 72          webgl.bindBuffer(webgl.ARRAY_BUFFER, cuadro.bufferVertices);
 73          webgl.bufferData(webgl.ARRAY_BUFFER,
 74                           new Float32Array(cuadro.vertices),
 75                           webgl.STATIC_DRAW);
 76      }
 77      //Funcion que dibuja sobre el contexto WebGL
 78      function dibujarEscena()
 79      {   //definimos el viewport
 80          webgl.viewport(0, 0, webgl.viewportWidth, webgl.viewportHeight);
 81          webgl.clear(webgl.COLOR_BUFFER_BIT | webgl.DEPTH_BUFFER_BIT);
 82          mat4.perspective(45, webgl.viewportWidth / webgl.viewportHeight,
 83                           0.1, 100.0, at.pMatriz);
 84          mat4.identity(at.mvMatriz);
 85          mat4.translate(at.mvMatriz, [0.0, 0.0, -7.0]);
 86          webgl.bindBuffer(webgl.ARRAY_BUFFER, cuadro.bufferVertices);
 87          webgl.vertexAttribPointer(at.ShaderProgram.vertexPositionAttribute,
 88                                    3, webgl.FLOAT, false, 0, 0);
 89          setMatrizUniforms(webgl, at);
 90          webgl.drawArrays(webgl.TRIANGLE_FAN, 0, 4);
 91          webgl.flush();
 92      }
 93
 94      //Funcion que llama el navegador cuando la pagina esta cargada
 95      function webGLStart()
 96      {
 97          var canvas = document.getElementById("canvas");
 98          webgl =  contextoWebGL(canvas);
 99          if(webgl==null)
100              alert("No puede iniciarse WebGL en este navegador");
101          else
102          {
103              initShaders();
104              initBuffers();
105              webgl.clearColor(1.0, 1.0, 1.0, 1.0);
106              webgl.enable(webgl.DEPTH_TEST);
107              dibujarEscena();
108          }
109      }
110    </script>
111    </head>
112    <body onload="webGLStart();">
113        <div >
114            <h1>WebGL</h1>
```

```
115             <h3>Cuadro</h3>
116             <canvas id="canvas"width="500" height="500"></canvas>
117         </div>
118     </body>
119 </html>
```

En resumen, el código anterior muestra los pasos para la generación de gráficos en WebGL, los cuales son:

1. Inicialización de los shader.
2. Reserva de los buffers que se necesitan para los modelos.
3. Dibujado del contenido.

Para el código de nuestro ejemplo, estos pasos se describen en las funciones *initShaders*, *initBuffers* y *dibujarescena*, mas la función *WebGLStart,* que será la encargada de ejecutar las funciones mencionadas. El ejemplo se apoya de tres objetos fundamentales definidos dentro del archivo *funciones.js* (líneas 32-34):

- Objeto *webgl* (línea 32): Este objeto obtendrá una referencia al contexto WebGL, nos servirá para ejecutar todas las funciones necesarias para la creación de los shaders, buffers y dibujar el contenido.

- Objeto *atributosPipeline(matriz, matriz)* (línea 33) : Este objeto contiene los atributos para la creación de contenido 3D, por ejemplo las matrices de modelo-vista, proyección y el programa de shaders, así como la pila de matrices.

- Objeto *objeto3D(vértices)* (línea 34): Ese objeto contiene los vértices de los modelos, así como los buffers que lo componen.

Empezaremos analizando el código desde la función *initShaders* (línea 41); ésta función lee y compila el código de los shaders, auxiliándose de la función *getshader(webgl, Shader)* (definida en *funciones.js*), para luego almacenarlos en las variables correspondientes (líneas 43-44). Podemos observar que el código para el procesador de vértices no está escrito en JavaScript sino en GLSL (líneas 21-31), este código calcula la posición para cada vértice por medio de la

multiplicación de las matrices de modelo-vista, proyección y la ubicación en el plano XYZ.

El código para el procesador de fragmentos (líneas 10-18) nos dice que se va tener una precisión alta y todo lo que se va a dibujar será de color rojo.

A partir de aquí los shaders están compilados y listos para ligarse al programa que los manejará, este proceso se ejecuta siguiendo las llamadas a funciones que se vieron en la figura 4.6.

Con la función *initShaders*, a cada atributo que compone el código del *Vertex Shader* se le agregará una referencia a nuestro objeto *atributosPipeline*, recordemos que esas referencias servirán para comunicarle a la GPU los cambios que vamos a realizar (en el caso de usar transformaciones o cualquier otro cálculo, para este ejemplo no se notará la diferencia).

A partir de este momento la GPU esta lista para recibir los vértices y realizar las operaciones para dibujar. La función *initBuffers* será la encargada de crear los buffers mediante las siguientes llamadas:

- Función *webgl.createBuffer()*(Línea 71): Esta función será la encargada de crear el buffer dentro de la GPU y nos devolverá una referencia, que será asignada a nuestro objeto *cuadro* para utilizarla posteriormente.

- Función *webgl.bindBuffer(Tipo buffer, Objeto buffer)* (Línea 72): Esta función necesita dos parámetros, el primero indica el tipo de buffer que se utilizará para almacenar los datos, existen dos tipos, el ARRAY_BUFFER que nos servirá para guardar los vértices que componen nuestro modelo y el ELEMENT_ARRAY_BUFFER que contiene la secuencia para dibujar los vértices. El segundo parámetro hace referencia al buffer que vamos a utilizar.

- Función *webgl.bufferData(Tipo buffer, Datos del objeto, Modo)* (Línea 73): Esta función se encargará de pasar los datos de nuestro modelo a la GPU, señalando

que los datos deben ser pasados a un objeto del tipo Float32Array. El modo indica la forma en que operan los buffers, existen tres tipos:

- *DYNAMIC_DRAW*: Es para los buffers que se utilizan a menudo y se actualizan con frecuencia
- *STREAM_DRAW*: Para los buffers que se inicializan una vez y rara vez se dibujan.
- *STATIC_DRAW*: Es para los buffers que se inicializan una vez dibujado, se utiliza muy a menudo (Por defecto).

Con los datos almacenados en la GPU, ahora es posible enviar todo el contenido hacia el contexto WebGL de nuestro navegador, la función *dibujarEscena()* configura muchos aspectos que se vieron en el capítulo 3, como el *viewport*, la perspectiva o las transformaciones , centrándonos en las siguientes funciones:

- Función *setMatrizUniforms(webgl, at)* (Línea 89): Esta función (definida en *funciones.js*) se encarga de actualizar las variables que contiene nuestro objeto pipeline a la GPU.

- Función *webgl.drawArrays(tipo Primitiva, índice inicial, índice final)*(Línea 90): Esta función será la encargada de dibujar todo el contenido al buffer que se le haga referencia (línea 86), el índice inicial nos indica que a partir de ese índice del buffer comenzara a dibujar las primitivas, el índice final indica hasta qué elemento va dibujar.

Para terminar, la función *WebGLStart()* (líneas 95-109) es la encargada de ejecutar todo nuestro código, el cual funciona cuando la página está completamente cargada (línea 112). Esta función es la encargada de obtener una referencia al contexto WebGL (línea 98) y almacenarla en una variable (webgl), con dicha variable podemos hacer algunas llamadas, como las que se vieron en el capítulo tres.

Para obtener el contexto nos auxiliaremos de la función *ContextoWebGL(canvas)* (declarada en el archivo *funciones.js*), esta referencia se logra con la función *getContext* del elemento Canvas, pasándole como parámetro una cadena constante: “webgl” o “experimental-webgl”. Esto varía según el navegador usado, pero a fin de evitar errores, en este ejemplo se incluyen ambos parámetros.

```
 1  function contextoWebGL(canvas)
 2  {
 3   var cWebgl;
 4   try
 5   {
 6     cWebgl = canvas.getContext("webgl")||canvas.getContext("experimental-
 7                                                                webgl");
 8     cWebgl.viewportWidth = canvas.width;
 9     cWebgl.viewportHeight = canvas.height;
10     console.log("Se consigio el contexto WebGL");
11     return cWebgl;
12   }
13   catch (e)
14   {
15     console.log("No puede iniciarse WebGL en este navegador");
16     return null;
17   }
18  }
```

Una vez que la página web se haya cargado, se debe mostrar en pantalla como la figura 4.7.

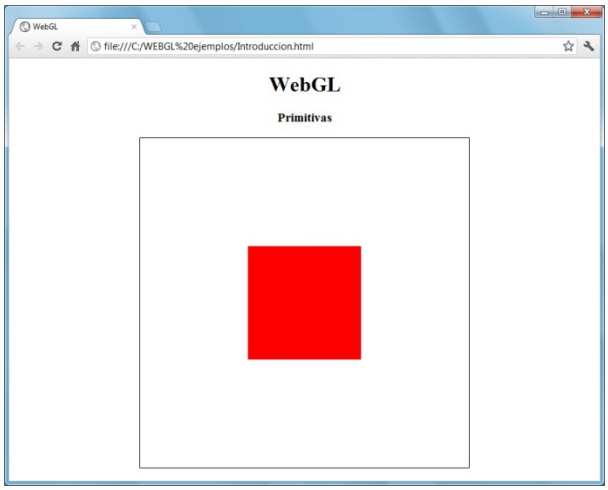


Figura 4.7. Cuadrado en WebGL.

Para los siguientes ejemplos solamente se comentarán las observaciones que se consideren importantes.

►Primitivas

Como se mencionó en el capítulo 3, OpenGL soporta diez primitivas que nos servirán para crear modelos más complejos y también sabemos WebGL utiliza OpenGL ES (la versión simplificado de OpenGL), por lo tanto el número de primitivas que soporta disminuye a siete, donde se incluyen básicamente los puntos, líneas, triángulos, así como sus derivados. Este ejemplo se puede encontrar en la carpeta “Ejemplos WebGL” del CD, con el nombre de “01-WebGL-Primitivas”.

Como se puede ver en el código del *Vertex Shader*, se utiliza la variable incorporada gl_PointSize (Línea 27), con un valor de 9.0, para poder ver los puntos, recordemos que por default tiene el tamaño de un pixel, haciendo muy difícil de ver.

En el método dibujarEscena() (Linea 66), Hace las transformaciones y dibuja las primitivas.

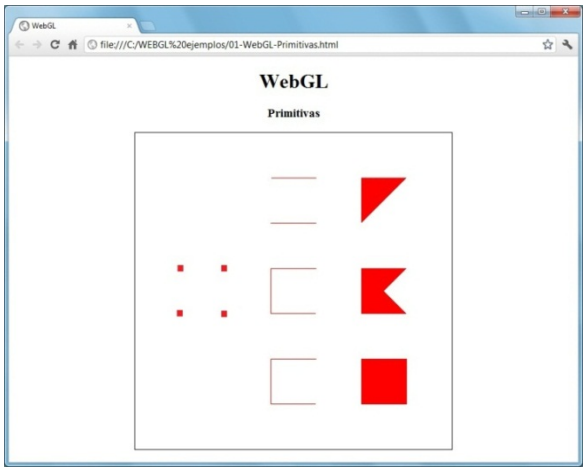


Figura 4.8. Primitivas soportadas por WebGL.

►Color

Este ejemplo corresponde al archivo “02-WebGL-Color” del CD. La mayor parte del código es bastante parecida al del primer ejemplo, agregando pequeñas modificaciones que se explicaran a continuación.

Para el código del *Fragment Shader*, se observa que el color se calcula a partir de la variable *gl_FragColor = vColor* (Linea 29), al contrario de los ejemplos anteriores en los que el color ya estaba definido. También se puede observar que al código del *Vertex Shader* se le agregaron otras variables: *attribute vec4 aVertexColor* (Línea 35) y *varying vec4 vColor* (Línea 38), éste último se declara igual tanto en el *Vertex Shader* como en el *Fragment Shader*, esto se debe a que el *Vertex Shader* una vez que calcula la geometría del objeto, debe agregar una variable para el cálculo del color (vColor), que será calculada por el *Fragment Shader*.

Como se mencionó con anterioridad, la función *initShaders()* (Línea 56) además de iniciar los shaders, también obtiene las referencias necesarias hacia las variables del *Vertex Shader*, para nuestro programa la variable *vertexColorAttribute* obtendrá una referencia hacia el atributo *aVertexColor* (línea 72), el cual es el encargado de recibir la información de los cambios de color en los shaders.

Por último, reservamos los buffers (líneas 84-89), que almacenarán la combinación de colores obtenidos del nuevo elemento *range* (líneas 157, 161,165) que incorpora HTML5 a través de funciones DOM.

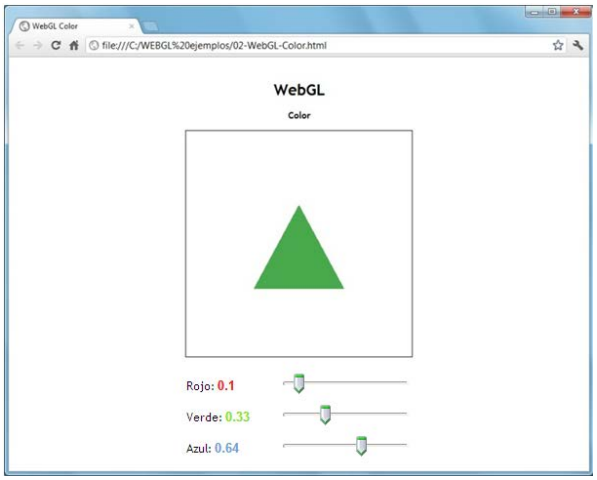


Figura 4.9. Color en WebGL.

►Transformaciones

Hasta este momento solo se han visto modelos bidimensionales, por lo que para la creación de modelos tridimensionales, se necesitan acomodar los vértices y de acuerdo al tipo de primitiva que necesitemos se construirá la figura. Para la creación de un cubo (que servirá para nuestro ejemplo), además de los vértices, se necesitarán los *índices* para armar el cubo (recordando que el tipo de primitiva más compleja que soporta WebGL es el triángulo), estos índices contienen la lógica para acomodar los triángulos necesarios que componen a cada una de las caras del cubo.

En el ejemplo “03-WebGL-Transformaciones” del CD, se puede observar los elementos que componen nuestro cubo, así como los colores y los índices mencionados (líneas 57, 93, 97). Todos estos datos siguen las mismos pasos: reservar los buffers, así como obtener sus respectivas referencias para acceder a éstos, dentro del código se resalta el modo en que se dibujan por medio de la función *webgl.drawElements(primitiva, número de elementos, tipo de variables, posición)* (Línea 170), ésta realiza el renderizado a través del *array* de los índices

que se le mandaron. Para finalizar se hacen las transformaciones, que se encarga de realizar la librería glMatrix (Lineas 158-171).

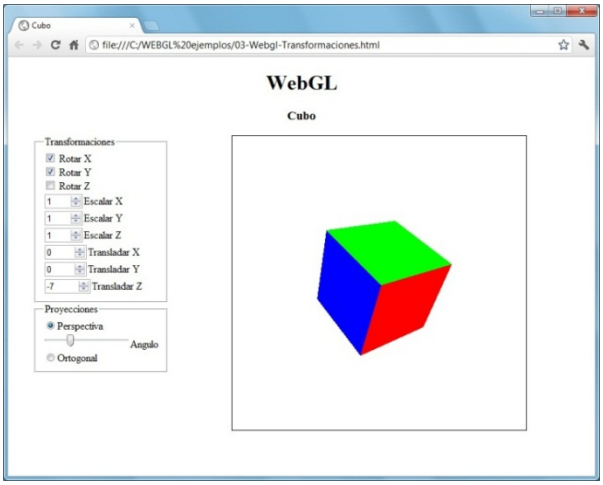


Figura 4.10. Transformaciones aplicadas a un cubo en WebGL.

Estos primeros ejemplos emplean solo WebGL para ser programados, por lo que entre más complejos son los modelos, resulta aún más difícil programarlos, porque se necesita un mayor número de llamadas al contexto de WebGL, por lo que es necesario emplear algún *framework* que facilite la programación. A continuación se explicará de mejor forma lo que son los frameworks, para después poder hacer una comparación adecuada entre ellos.

4.3. Frameworks

Como se pudo apreciar, programar aplicaciones WebGL de bajo nivel, resulta ser una tarea laboriosa, ya que entre más complejos resultan nuestros modelos, se va generando una gran cantidad de datos difíciles de manipular, haciendo que esta tecnología no explote su verdadero potencial.

Precisamente por que WebGL posee una API de muy bajo nivel, podría provocar el rechazo por parte de algunos programadores por la dificultad para programar las aplicaciones, pero gracias a su creciente uso, muchos desarrolladores han

trabajado duro para proveer una biblioteca de alto nivel a través de la creación de varios frameworks. Es por esto que a continuación se procederá a explicar lo que son los framework y la importancia que estos tienen para WebGL.

El término framework es usado para definir una estructura de software que facilita la creación de otro programa mediante una serie de bibliotecas y lenguaje de programación que ayudan al desarrollo de un proyecto; estos frameworks son sólidos y garantizan que funcionan en cualquier sistema, haciendo la programación más sencilla. Los frameworks permiten que los desarrolladores web no tengan que lidiar con las complejidades de WebGL y las dificultades que representan las matemáticas 3D en un diseño (como lo son las coordenadas tridimensionales).

Algunos frameworks disponibles para el desarrollo de WebGL son: C3DL, CopperLicht, CubicVR, GLGE, Jax, O3D, Oak3D, PhiloGL, SceneJS, SpiderGL, TDL, Three.js y X3DOM.

De estos frameworks podemos encontrar algunas diferencias notables, algunos de ellos incluyen un mayor número de características implementadas que otros, lo que los vuelve más accesibles a la hora de programar, en este trabajo nos centraremos en dos de ellos: GLGE y Three.js, debido a que proveen prestaciones más avanzadas para programar, además de ser los que cuentan con un mayor soporte en línea. Sus principales características serán comparadas a continuación.

● GLGE

El framework GLGE fue creado por el diseñador Paul Brunt a finales del año 2009, como respuesta a la simplificación para programar aplicaciones WebGL.

Figura 4.11. GLGE permite crear escenas más complejas.

GLGE logra separar el contenido 3D (escenas, objetos 3d, luces, texturas, cámaras) de forma transparente por medio de la edición de un archivo del tipo .xml, incluido en el mismo HTML o importándolo como un archivo externo, donde los elementos que componen la escena se acceden mediante funciones DOM. El siguiente código muestra un pequeño ejemplo utilizando este framework.

```
 1  <!DOCTYPE HTML>
 2  <html>
 3     <head>
 4        <title>Primitivas con GLGE</title>
 5          <meta http-equiv="content-type" content="text/html;
 6          charset=ISO-8859-1">
 7          <script type="text/javascript" src="src/glge-compiled.js">
 8          </script>
 9     </head>
10     <body style="font-family: Trebuchet MS,Helvetica;">
11       <div id="container" style="text-align:center">
12          <h1>GLGE</h1>
13          <canvas id="canvas" style="border-width: 1px;
14             border-style: solid;border-color: black;"
15             width="900" height="500">
16          </canvas>
17          <p>Cubo Usando el framework GLGE</p>
18       </div>
19       <script type="text/javascript">
```

```
20          //Variable que contendrá al elemento canvas
21          var canvas = document.getElementById( 'canvas' )
22          //Variable que realizara el render en la escena
23          var renderer = new GLGE.Renderer( canvas );
24          //Variable que contendrá el documento XML
25          var XMLdoc = new GLGE.Document();
26          //Funcion que ejecutara la escena con todos los objetos que
27          //contenga
28          XMLdoc.onLoad = function()
29          {
30              var escena = XMLdoc.getElement( "mainscene" );
31              renderer.setScene( escena );
32              renderer.render();
33              setInterval(function()
34              {
35                renderer.render();
36              }, 15);
37          }//Carga el documento XML
38          XMLdoc.load("Cubo.xml");
39
40        </script>
41      </body>
42  </html>
```

El programa empieza en la línea 7 donde se importa la librería *glge*, ya importada empezaremos a escribir el *script* correspondiente que se encuentra entre las líneas 19-40.

Lo primero será crear tres variables fundamentales para la construcción de nuestra aplicación (líneas 21-25), la primera variable será la que contendrá una referencia al elemento *canvas*, las otras dos variables son objetos de la biblioteca GLGE, del cual se detalla su funcionamiento a continuación:

- *GLGE.Renderer(canvas):* Este objeto como su nombre lo dice, será el encargado de renderizar nuestra escena, necesitando como argumento una referencia al elemento canvas para poder plasmar todo el contenido de nuestra escena.

- *GLGE.Document():* Este objeto es el encargado de leer nuestro documento XML ubicado en el directorio de la página, su función es cargar el documento (línea 38), este objeto tiene una función llamada *onLoad* (línea 28), donde se escribirá el código, recordando a la función *main* en otros lenguajes de programación.

En dicha sobreescritura podemos observar como obtenemos la escena con todos los elementos que la componen por el método DOM.

- *XMLdoc.getElement("mainescena")* (línea 30). A nuestro objeto *Renderer* le asignamos una escena para realizar el proceso de renderizado (líneas 31-32) y por ultimo le indicamos a la aplicación el tiempo transcurrido para que se actualice la escena en caso de utilizar animaciones (líneas 33-36).

Ahora vamos a analizar el documento XML (Cubo.xml) que contiene la definición de nuestro modelo.

```
 1 <?xml version="1.0"?>
 2 <!-- Cubo.xml -->
 3 <glge>
 4   <mesh id="cfrontal">
 5      <positions>
 6      -1.0, -1.0,  1.0,
 7       1.0, -1.0,  1.0,
 8      -1.0,  1.0,  1.0,
 9       1.0,  1.0,  1.0
10      </positions>
11   </mesh>
12   <mesh id="ctrasera">
13      <positions>
14         -1.0, -1.0, -1.0,
15         -1.0,  1.0, -1.0,
16          1.0, -1.0, -1.0,
17          1.0,  1.0, -1.0
18      </positions>
19   </mesh>
20   <mesh id="carriba">
21      <positions>
22         -1.0,  1.0, -1.0,
23         -1.0,  1.0,  1.0,
24          1.0,  1.0, -1.0,
25          1.0,  1.0,  1.0
26      </positions>
27   </mesh>
28   <mesh id="cabajo">
29      <positions>
30         -1.0, -1.0, -1.0,
31          1.0, -1.0, -1.0,
32         -1.0, -1.0,  1.0,
33          1.0, -1.0,  1.0
34      </positions>
35   </mesh>
36   <mesh id="cderecha">
37      <positions>
```

```
38          1.0, -1.0, -1.0,
39          1.0,  1.0, -1.0,
40          1.0, -1.0,  1.0,
41          1.0,  1.0,  1.0
42       </positions>
43    </mesh>
44    <mesh id="cizquierda">
45       <positions>
46          -1.0, -1.0, -1.0,
47          -1.0, -1.0,  1.0,
48          -1.0,  1.0, -1.0,
49          -1.0,  1.0,  1.0
50       </positions>
51    </mesh>
52
53    <material id="rojo" color="#FF0000" shadeless="TRUE"/>
54    <material id="amarillo" color="#FFFF00" shadeless="TRUE"/>
55    <material id="azul" color="#0000FF" shadeless="TRUE"/>
56    <material id="verde" color="#00FF00" shadeless="TRUE"/>
57    <material id="violeta" color="#FF00FF" shadeless="TRUE"/>
58    <material id="naranja" color="#FF8000" shadeless="TRUE"/>
59
60    <animation_vector id="spin" frames="240">
61      <animation_curve channel="RotY">
62            <linear_point x="1.0" y="0.0" />
63            <linear_point x="240.0" y="6.282" />
64      </animation_curve>
65    </animation_vector>
66
67    <scene id="mainscene" camera="#mainCamera">
68     <group id="cubo" animation="#spin">
69       <object id="1" draw_type="DRAW_TRIANGLESTRIP"
70               mesh="#cfrontal" material="#rojo"/>
71       <object id="2" draw_type="DRAW_TRIANGLESTRIP"
72               mesh="#ctrasera" material="#amarillo"/>
73       <object id="3" draw_type="DRAW_TRIANGLESTRIP"
74               mesh="#carriba" material="#verde"/>
75       <object id="4" draw_type="DRAW_TRIANGLESTRIP"
76               mesh="#cabajo" material="#naranja"/>
77       <object id="5" draw_type="DRAW_TRIANGLESTRIP"
78               mesh="#cderecha" material="#violeta"/>
79       <object id="6" draw_type="DRAW_TRIANGLESTRIP"
80               mesh="#cizquierda" material="#azul"/>
81     </group>
82     <camera id="mainCamera" loc_z="10" loc_y="3"  rot_x="-0.3"  />
83    </scene>
84  </glge>
85
```

Los elementos que constituyen nuestro modelo se señalan mediante etiquetas y todos sus atributos se definen entre comillas, la definición de nuestro modelo empieza entre las etiquetas *<glge> </glge>* (líneas 3-84).

GLGE trabaja con el concepto de mallas que contienen todos los vértices que componen la parte de un elemento, dichas mallas se representan por medio de las etiquetas *<mesh> </mesh>.* Cada malla contendrá las posiciones de los vértices y sus coordenadas UV (en caso de utilizar texturas), para este ejemplo declaramos seis mallas (líneas 4-51) que representan cada lado del cubo (podríamos hacerlo en una malla, pero para dar color a cada lado fue necesario declararlas por separado), dentro de la etiqueta *<mesh>* se encuentra la etiqueta *<positions>*, que contiene los vértices del modelo.

La definición del color se logra por medio de la etiqueta *<material/>* (líneas 53-58) que contendrá tanto su identificador, definición del color (en formato hexadecimal), así como la opción de sombreado desactivado (shadeless).

La animación se logra por medio de las etiquetas *<animation_vector> </animation_vector>* (líneas 60-65) dicha etiqueta contendrá los *frames* (cuadros) necesarios para lograr la animación, en nuestro ejemplo le decimos que en 240 *frames* dará una vuelta completa sobre el eje Y, declarada entre las etiquetas *<animation_curve> </animation_curve>* y *<linear_point/>* que contendrá los puntos de bases para lograr dicho efecto.

Y por último construimos nuestra escena que se componen de todos los elementos declarados anteriormente, por medio de las etiquetas *<scene></scene>*, para hacer referencia a los elementos ya creados se logra anteponiendo el símbolo # seguido del identificador del elemento. Para terminar construiremos nuestro cubo en la etiqueta *<group></group>* uniendo todas las caras y agregándole la animación, así como agregar una cámara con una posición dada.

Afortunadamente GLGE ofrece un plugin para el programa de diseño *Blender*, que ayuda a exportar nuestros modelos a XML, aunque también esta la posibilidad de exportarlo al tipo *COLLADA*, que es un formato ampliamente aceptado en todos los programas de diseño 3D y que GLGE es capaz de traducirlo a WEBGL.

Como se puede apreciar, se necesitó de pocas líneas de programación y se omitió la programación de los shaders, los buffers, así como actualizar y manejar las matrices de modelo-vista y la perspectiva.

Como características adicionales de GLGE podemos encontrar: animación por fotogramas clave, mapeado de texturas, animación por el método *Skeletal animation*[26], soporte para archivos de distribución de contenido 3D (COLLADA, MD2), mapeado por paralaje, efectos de niebla, partículas, luces, renderizado de texto, reflexiones y refracciones, personalización de los shaders, ofrece widgets personalizables, asi como un plugin para el programa de diseño en·3D *Blender*.

● **THREE.js**

Este framework fue creado por Ricardo Cabello y rápidamente comenzó a ganar popularidad gracias a su diseño bastante elaborado, además de que logró integrar las bases de trabajo de programas de diseño 3D como Autodesk 3ds Max y Blender. Con esto, los diseñadores y programadores se familiarizaron muy bien con él y no tuvieron problemas al momento de desarrollar aplicaciones en WebGL.

Figura 4.12. Diseños 3D usando Three.js.

[26] Skeletal animation es un método de animación por computadora usado para la simulación de animales vertebrados o movimientos musculares

Su éxito es tal que no solo trabaja con WebGL, sino también con el elemento *Canvas*, haciendo que las aplicaciones puedan ser trasladadas a éste, en caso de que nuestro hardware no tenga los recursos gráficos necesarios (como es el caso de algunos chips de video Intel), además de ser extensible, esto significa que puede hacer uso de las bibliotecas de otros desarrolladores, convirtiéndola en una herramienta muy completa. Su estructura se puede ver en la siguiente figura:

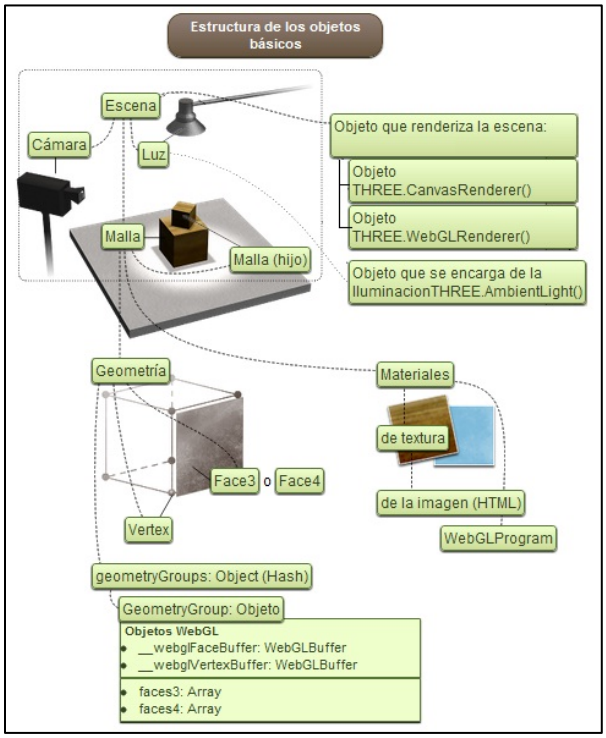


Figura 4.13. Funcionamiento de Three.js.

El siguiente ejemplo muestra el uso de este Framework:

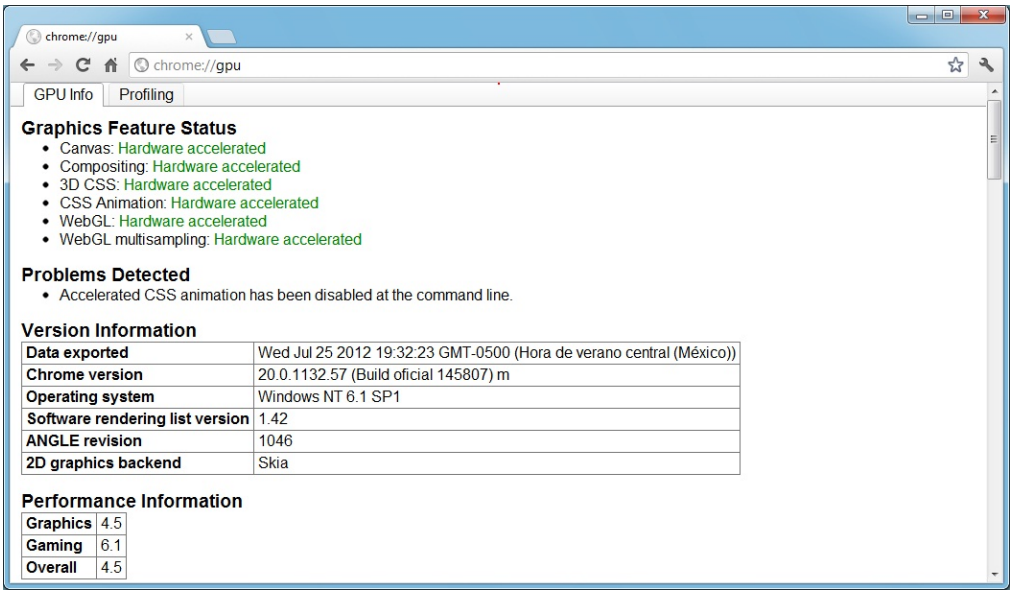
```
 1  <!doctype html>
 2  <html>
 3    <head>
 4      <title>Cubo en Three.js</title>
 5      <meta charset="charset=ISO-8859-1">
 6      <link rel="stylesheet" type="text/css" href="css/hoja.css">
 7      <script src="src/Three.js"></script>
 8      <script src="src/Detector.js"></script>
 9    </head>
10    <body>
11    <div >
12        <h1>Three.js</h1>
13        <p>Cubo Usando el framework Three.js</p>
14    </div>
```

```
15   <div id="webgl"></div>
16   <script>
17
18      if ( ! Detector.webgl )
19               Detector.addGetWebGLMessage();
20      var contenedor;
21      var camara, escena, render;
22      var cubo;
23
24      init();
25      animate();
26
27      function init()
28      {
29
30        escena = new THREE.Scene();
31
32        camara = new THREE.PerspectiveCamera(45,900/500,0.1,1000);
33        camara.position.y = 75;
34        camara.position.z = 10;
35        escena.add(camara);
36
37        var mat = [];
38        mat.push(new THREE.MeshBasicMaterial({color:0xFF00FF}));//violeta
39        mat.push(new THREE.MeshBasicMaterial({color:0x0000FF}));//azul
40        mat.push(new THREE.MeshBasicMaterial({color:0x00FF00}));//verde
41        mat.push(new THREE.MeshBasicMaterial({color:0xFF8000}));//naranja
42        mat.push(new THREE.MeshBasicMaterial({color:0xFF0000}));//rojo
43        mat.push(new THREE.MeshBasicMaterial({color:0xFFFF00}));//amarillo
44
45        cubo = new THREE.Mesh( new THREE.CubeGeometry(50,50,50,10,10,10,mat),
46                               new THREE.MeshFaceMaterial() );
47        cubo.position.y = 10;
48        cubo.overdraw = true;
49        escena.add( cubo );
50
51        render = new THREE.WebGLRenderer({antialias: true});
52        render.setSize(900,500);//asigna el tamaño del del Contexto
53        document.getElementById('webgl').appendChild(render.domElement);
54      }
55
56      function animate()
57      {
58         requestAnimationFrame(animate);
59         renderizar();
60      }
61
62      function renderizar()
63      {
64        var timer = new Date().getTime() * 0.0010;
65        camara.position.x = Math.cos(timer) * 200;
66        camara.position.z = Math.sin(timer) * 200;
67        camara.lookAt(escena.position);
68        render.render(escena, camara);
69      }
70
71   </script>
```

```
72    </body>
73  </html>
74  <!doctype html>
```

Podemos ver en el ejemplo, que three.js esta muy bien elaborado, con los conceptos ya aprendidos, el código resulta fácil e intuitivo al leer y del cual podemos destacar que usando este framework, es posible reducir aún más las líneas de código.

Si por alguna razón se llegaran a presentar problemas cuando se intentan corren los ejemplos o sí se pueden visualizar, pero de forma incorrecta, es posible que la computadora que se está usando no sea totalmente compatible con WebGL. Para averiguar si una computadora cuenta con las capacidades necesarias y que es compatible, en el navegador Chrome existe un pequeño comando con el cual lo podemos averiguar, solo es necesario escribir en la barra de direcciones: *chrome://gpu* y debería abrir la siguiente ventana:



Si en la sección de “Graphics Feature Status” se muestra algún valor en rojo, significa que nuestra tarjeta gráfica no es compatible con la aceleración por hardware con WebGL.

4.3.1. Comparativa entre Frameworks

Como sabemos, los Frameworks descritos anteriormente son un conjunto de librerías JavaScript, pero varían en sus características y la forma en que se trabaja con ellos. La tabla comparativa que se muestra a continuación, resume las características principales de los dos Frameworks a considerar.

	GLGE	THREE.JS
Página web	http://www.glge.org/	http://threejs.org/io/s/
Servidor web[27]	Apache, Node.js	Apache
Formatos que soporta	COLLADA, MD2, JSON	COLLADA, JSON
Soporte para programas de diseño 3D	Si (Blender)	Si (Blender, Autodesk 3ds Max)
Personalización de shaders	Si	Si
Opciones de renderizado	WebGL	Canvas y WebGL
Documentación en línea	Solo revisando los ejemplos	Solo revisando los ejemplos
Tamaño de librería	668.5 KB	385 KB

Tomando como base la anterior comparación, podemos concluir que el Framework Three.js es más completo, sin embargo al momento de analizarlo con el *WebGL inspector*, nos pudimos percatar que emplea más recursos de hardware, haciendo que el tiempo de renderizado de las escenas sea mayor. Por otra parte, GLGE esta más optimizado para el uso de shaders, haciendo que el tiempo de renderizado sea más corto, lo que lo convierte en el ideal para dispositivos portátiles como el iPhone y los teléfonos con sistemas operativos Android.

Por todo lo visto anteriormente, GLGE representa la mejor opción para desarrollar el caso práctico de este trabajo, el cual es expuesto en el siguiente capítulo: “Desarrollo de un caso práctico aplicando la tecnología WebGL”.

[27] El Servidor web es necesario cuando se cargan modelos COLLADA y MD2

CAPÍTULO 5

DESARROLLO DE UN CASO PRÁCTICO APLICANDO LA TECNOLOGÍA WEBGL

El presente capítulo está dedicado al desarrollo de un caso práctico en el que se vean aplicados los conocimientos y conceptos estudiados durante todo el desarrollo de este trabajo. Para este propósito, se presentará la creación de un diseño tridimensional del centro tecnológico de la Facultad de Estudios Superiores Aragón, que será mostrarlo en un navegador web a modo de un recorrido virtual por el edificio, usando por supuesto la tecnología WebGL para este fin.

Figura 5.1. Centro tecnológico Aragón.

El centro tecnológico es un edifico dedicado a la investigación y desarrollo de las carreras profesionales del área científico técnico que son impartidas en la FES Aragón. El edificio fue inaugurado en 1996 y actualmente cuenta con los laboratorios de diseño y manufactura, comportamiento de materiales, cómputo, diagnóstico energético, cómputo y la educación, estudios ambientales, ingeniería ambiental, instrumentación y medición, mecánica aplicada, rehabilitación urbana, seguridad informática e incubadora de empresas; además de contar con un auditorio y salas de videoconferencias.

Para crear el diseño tridimensional del edificio, nos apoyaremos en el programa de diseño 3D “Blender” y una vez terminado dicho modelo se procederá a exportarlo a WebGL con la ayuda del *framework* visto en el capítulo anterior (GLGE) y así poder visualizarlo en un navegador web.

Planeación

Para crear el diseño en tercera dimensión se decidió emplear Blender, el cual es un programa usado para el modelado y animación de gráficos en tercera dimensión. Blender es un programa de código abierto bastante intuitivo y potente, en los últimos años ha ganado mucha popularidad debido principalmente a que ofrece características y funciones avanzadas para el diseño 3D que están a la altura de programas de pago como 3ds Max o Maya.

Blender fue creado en el año 1995 por un estudio de animación holandés llamado “NeoGeo”, pero fue hasta el año 2002 que su código fue liberado y pasó a ser administrado por la *Blender Foundation*. Blender es una herramienta que ha sido usada en proyectos cinematográficos como Spider-Man 2 y Plumíferos, además de cortometrajes como Sintel, Elephants Dream y Big Buck Bunny.

Figura 5.2. “Sintel”, un cortometraje creado con Blender.

Para el diseño 3D del centro tecnológico, usaremos la versión 2.62 de Blender, la cual fue liberada el 16 de febrero de 2012 y que puede ser descargada de la siguiente dirección:

```
http://www.blender.org/download/get-blender/
```

Al ser un programa multiplataforma, está disponible tanto para Windows, Linux, Mac OS X y FreeBSD.

Una vez descargado e instalado podemos empezar a trabajar en él, por *default*, al abrir una nueva ventana se nos muestra un cubo, sobre el cual se puede empezar a diseñar, aunque también es posible usar las diferentes figuras 3D prediseñadas que pueden ser modificadas para darles la forma que se desee. La interfaz gráfica de Blender resulta muy intuitiva y solo es cuestión de habituarse a ella para empezar a trabajar fluidamente.

Figura 5.3. Interfaz gráfica de Blender.

Debido a que Blender es un software libre, la documentación y los manuales de uso completos, solo se encuentran disponibles a la venta en su página oficial, aunque también se pueden encontrar tutoriales gratuitos en la red, como “BlenderWiki”, el cual a pesar de encontrarse casi totalmente en ingles, puede resultar de gran ayuda para comenzar a trabajar con este programa de diseño 3D.

Para construir el modelo tridimensional del centro tecnológico, también nos apoyamos en el plano usado durante la construcción del edificio (Figura 5.4), de esta manera fue posible saber la forma y las medidas reales en metros que debíamos usar para nuestro diseño en Blender. Las medidas que no estaban incluidas en el plano, como la anchura de las puertas y la altura de los pasamanos, se tomaron usando un flexometro.

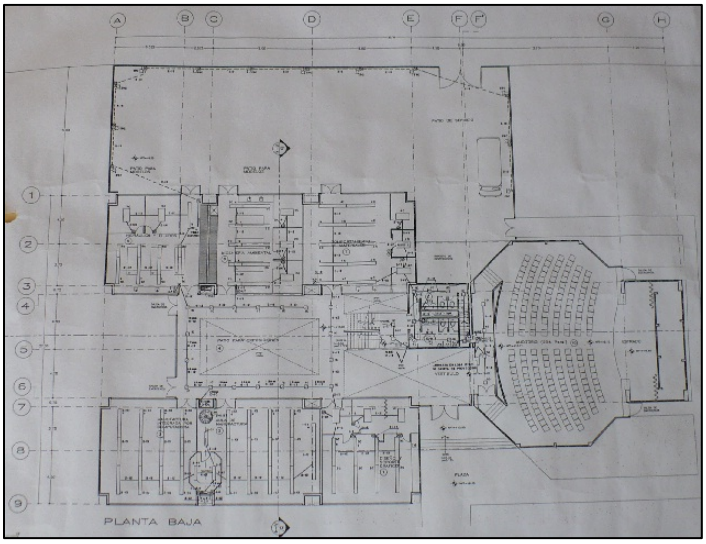


Figura 5.4. Plano de construcción del centro tecnológico Aragón.

Además del plano, también nos apoyamos en fotografías tanto del exterior como del interior del edificio (figura 5.5), a fin de crear un diseño lo más parecido al que se puede encontrar en el edificio real, además también se tomaron fotografías de los muros y puertas para usarlas como texturas en el modelo final.

Figura 5.5. Fotografías exteriores e interiores del centro tecnológico Aragón.

Diseño del centro tecnológico

Una vez recabado todo este material de referencia, fue posible proceder a crear el diseño tridimensional en Blender, comenzando por el exterior del edificio, donde se definió la posición y tamaño de los muros obtenidos del plano y otros detalles vistos en las fotografías, como las ventanas.

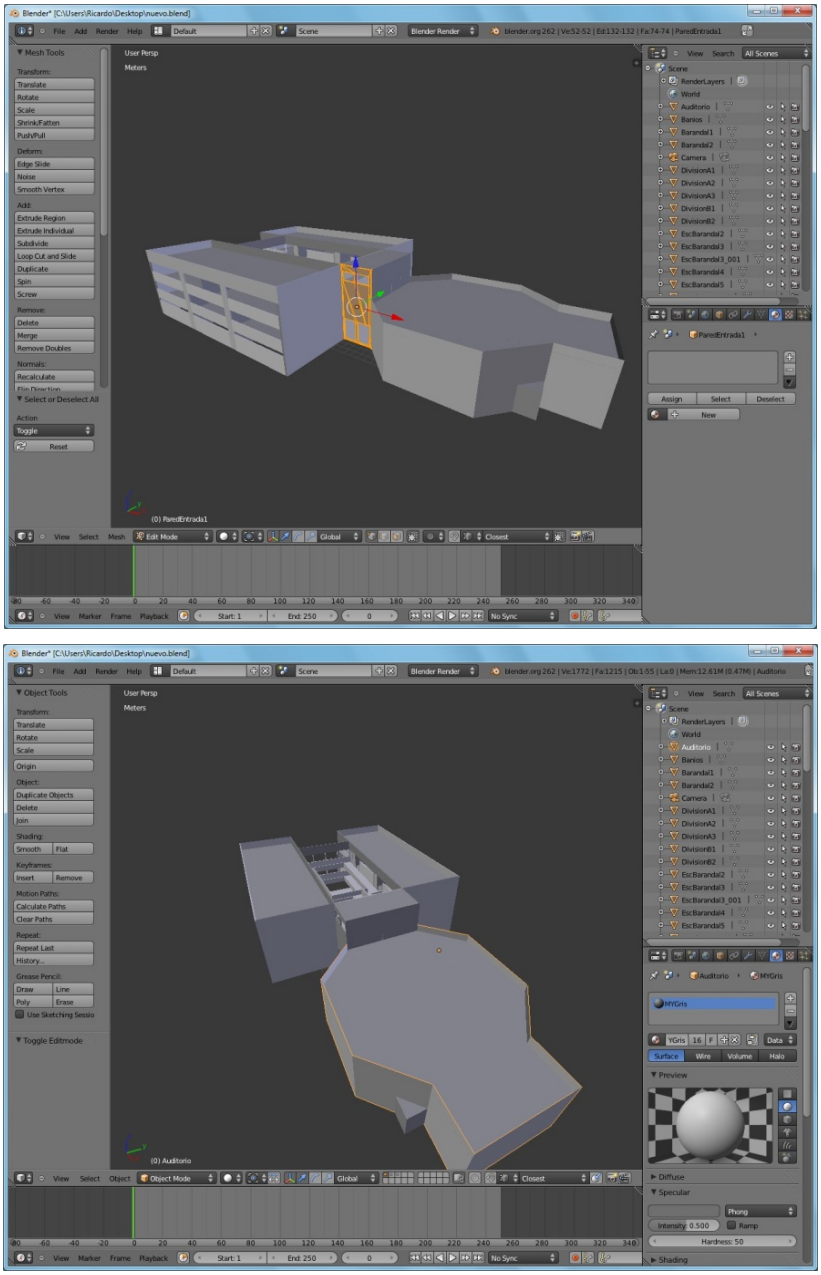


Figura 5.6. Diseño exterior del centro tecnológico en Blender.

Ya teniendo modelado el exterior del edificio, se comenzó a crear el interior de éste, principalmente se definió la ubicación de los laboratorios, las escaleras, pasillos, pasamanos y puertas.

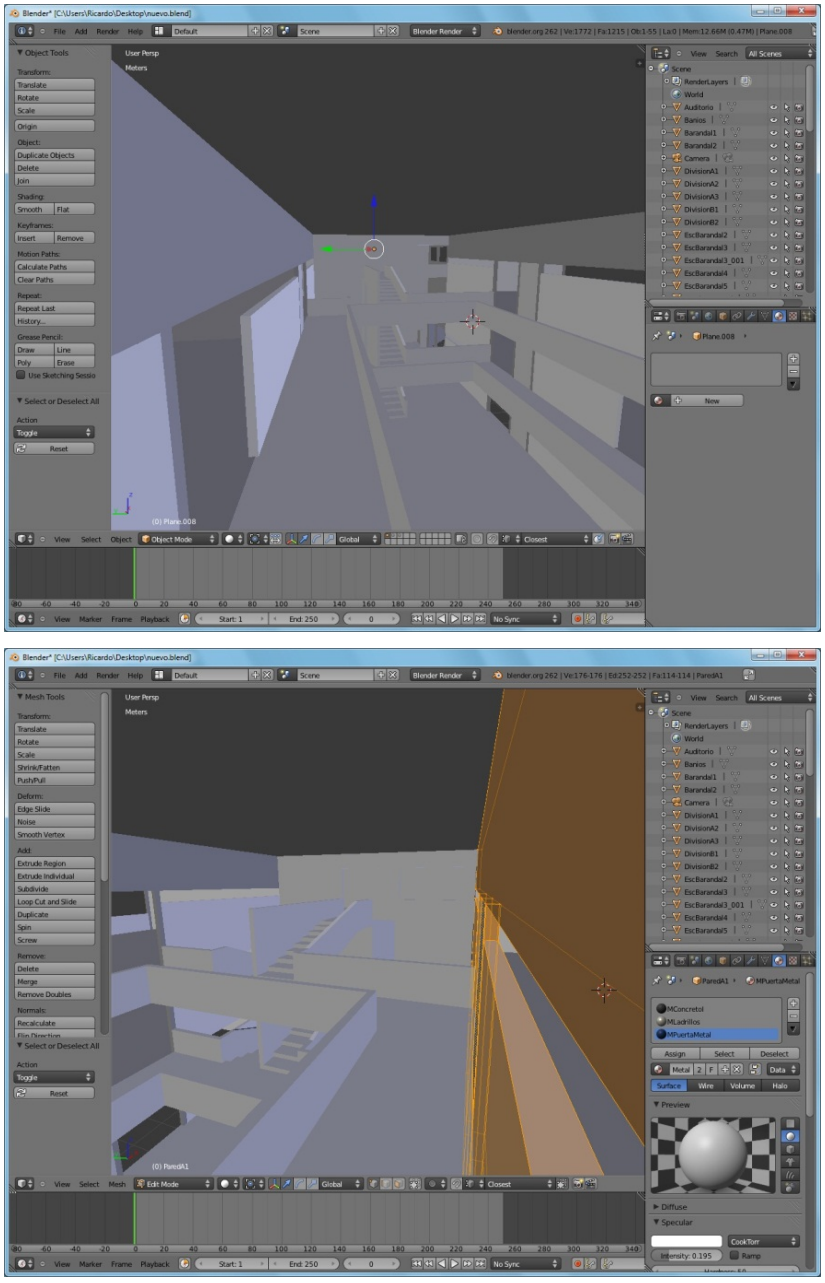


Figura 5.7. Diseño interior del centro tecnológico en Blender.

Al tener el diseño concluido, proseguimos a cargar los materiales y texturas, como ya se mencionó, para las texturas se usaron fotografías de los muros, pisos y puertas. Para algunas otras áreas (como el cristal de las ventanas), se emplearon materiales creados con el mismo Blender.

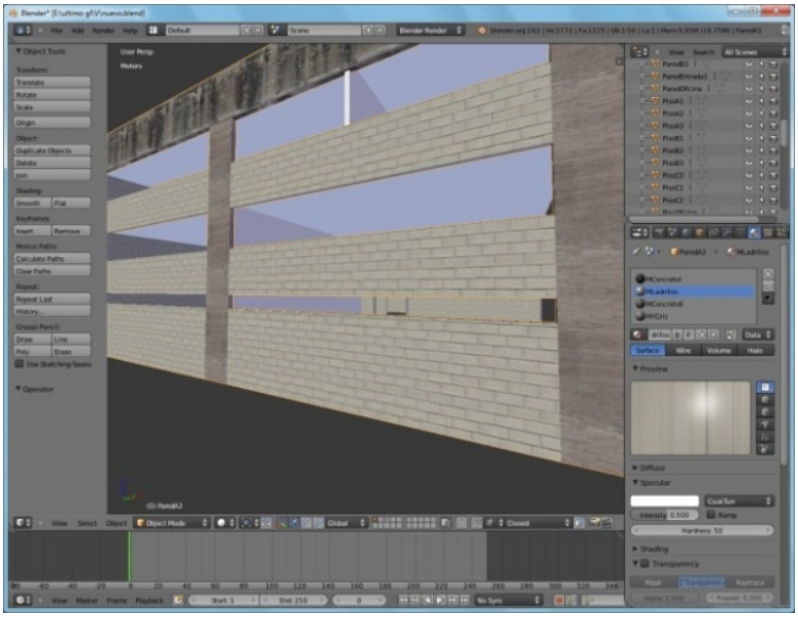


Figura 5.8. Texturas aplicadas en los muros del edificio.

Exportando el modelo de Blender a collada (.DAE)

Ya concluido el trabajo de modelado en Blender, es necesario exportar el diseño al formato collada (DAE), que es un formato estándar basado en XML y que se utiliza para transportar información entre los diferentes programas de diseño, así como de plataformas entre los diferentes dispositivos que existen en el mercado (consolas de videojuegos, dispositivos móviles, etc).

Dentro de las ventajas de exportar nuestro modelo al formato *collada,* se encuentran principalmente el de englobar toda la información de nuestro modelo (por ejemplo los vértices que lo componen, colores, luces, coordenadas del mapeado de texturas, etcétera); se recordará que en el capítulo cuatro toda esta información se declara por separado y resulta muy difícil de manejar para el programador.

Para este caso práctico, programar toda esa información sin el uso de algún framework que utiliza el manejo de algún formato de intercambio para leer aplicaciones 3d (Collada, MD2, WAD, JSON), resultaría una tarea sumamente laboriosa.

Configurando y programando en GLGE

Como recordamos, en el capítulo anterior se vio una pequeña introducción de cómo funciona el framework GLGE, donde se necesitaba su archivo de configuración de todos los elementos que conformarán la escena, ya sea incrustado en el mismo archivo html o definido en un archivo XML, para el recorrido virtual algunos objetos de nuestra escena fueron definidos independientemente de nuestro modelo del centro tecnológico (modeloCentro.dae), con las funciones que servirán como puntos esenciales de interacción con el usuario (puertas de los laboratorios, puertas de vidrio y texto).

Una vez exportado el modelo, se analizará el archivo Centro.xml que contendrá los elementos necesarios para que GLGE lo pueda interpretar. Por cuestión de espacio se omitieron algunos datos con respecto al archivo original, así como información redundante representada con el símbolo "*". A continuación se muestra el archivo correspondiente:

```
 1  <?xml version="1.0"?>
 2  <glge>
 3  <!—Mallas objetos -->
 4  <mesh id=" ObjVidrio">****</mesh>
 5  <mesh id="Puertas">****</mesh>
 6  <mesh id="PAudi">****</mesh>
 7  <!-- Materiales -->
 8  <material id="MVidrio" color="rgb(250,250,250)" specular="0.5"
    shininess="50" emit="0" alpha="0.3" ></material>
 9  <material id="dorado" specular="1.0" color="#ff3"></material>
10  <material id="MPuerta" color="rgb(204,204,204)" specular="0.1" >
11     <texture id="TPuerta" src="img/puerta.png" />
12     <material_layer texture="#TPuerta" mapinput="UV1" mapto="M_COLOR" />
13  </material>
14  <material id="MPuertaMadera" color="rgb(204,196,108)" specular="0.097561"
      shininess="50" emit="0" >
15     <texture id="TPMadera" src="img/puertaMadera.jpg" />
16     <material_layer texture="#TPMadera" mapinput="UV1" mapto="M_COLOR" />
17  </material>
18  <!-- Camara -->
19  <camera id="maincamera" loc_y="-145" loc_x="10" loc_z="2"rot_order="ROT_XZY"
      xtype="C_ORTHO" rot_z="0" rot_x="1.31" rot_y="-0.10993" />
20  <!-- Escena -->
21  <scene id="mainscene" camera="#maincamera" ambient_color="#666"
    fog_type="FOG_NONE">
22  <object id="ventana2" mesh="# ObjVidrio" rot_order="ROT_ZYX"
    material="#MVidrio" ztransparent="TRUE" loc_x="-65.025048" loc_y="0"
    loc_z="13.725" scale_x="0.0075" scale_y="15.449994" scale_z="13.5"  />
```

```
23 <object id="entrada" mesh="# ObjVidrio" rot_order="ROT_ZYX"
   material="#MVidrio" ztransparent="TRUE" loc_x="8.961" loc_y="-15.225"
   loc_z="4.725" scale_x="6.525" scale_y="0.075" scale_z="4.5"  />
24 <object id="p0" mesh="# ObjVidrio" rot_order="ROT_ZYX" material="#MVidrio"
   ztransparent="TRUE" loc_x="2.025" loc_y="1.80" loc_z="25.725003"
   scale_x="0.075" scale_y="2.1" scale_z="3.45" />
25 <object id="p1" mesh="#Puertas" rot_order="ROT_ZYX" material="#MPuerta"
   loc_x="-20.7" loc_y="15.3" loc_z="5.504286" scale_x="3" scale_y="3.225"
   scale_z="3" rot_x="1.570797" />
26 ****
27 ****
28 ****
29 ****
30 ****
31 ****
32 ****
33
34 <collada id="aragon" document="modeloCentro.dae"  rot_x="1.57" scale="3"/>
35 <light id="mainlight" rot_order="ROT_ZYX" color="rgb(255,251,233)" loc_x="-
   134.300055" loc_y="-56.486001" loc_z="63" rot_x="-0.998821" rot_y="0.243257"
   rot_z="-4.355912"  attenuation_constant="0.2" attenuation_linear="0.000001"
   attenuation_quadratic="0.0001" type="L_DIR" />
36 <text id="centro" loc_z="12" loc_x="23" loc_y="-25"  text="    CENTRO  "
   size="100" font="arial" color="yellow" rot_x="1.570797" rot_y="2.3561955"/>
37 ****
38 ****
39 ****
40 ****
41 </scene>
42 </glge>
```

Del archivo se puede observar (líneas 4-6) la creación de las mallas necesarias de los elementos que interactuaran con el usuario, como son las puertas del auditorio, laboratorios del centro tecnológico y la puerta de entrada.

En las líneas 8-17, se declaran cuatro tipos de materiales para algunos elementos de la escena, como lo son el vidrio, un color que resalte de los elementos (que nos servirá para interactuar con el usuario, el cual es llamado *dorado*) y dos más que contendrán las texturas de las puertas del auditorio y los laboratorios. Se podrá apreciar que algunos atributos en la declaración de los materiales son nuevos y resulta fácil intuir que contienen información como los tipos de luz que emite, así como el uso de transparencias o si utiliza texturas.

La línea 29 se declara la cámara, que maneja una proyección del tipo perspectiva, con sus respectivas coordenadas, además del ángulo de rotación en el que está enfocando, dicho ángulo esta expresado en radianes.

Las líneas 25-34 declaran las puertas que interactuan con el usuario, en la línea 25 se puede observar como se empiezan a hacer referencias a las mallas y a los materiales anteriormente creados por medio del símbolo “#” (por cuestión de espacio en el código se omitieron las demás puertas).

En la línea 34 se declara el uso de nuestro modelo (modeloCentro.dae), por medio de la etiqueta <collada>, con una inclinación de π/2 radianes equivalente a 90 grados sobre el eje X y con factor de escala de 3.

La línea 35 define una luz, con la combinación de sus atributos se emulará la luz de un sol y por último en las líneas 36-40 se crearan los textos con los que interactua el usuario. Una vez analizado el archivo Centro.xml se analizará la parte programable del proyecto que se encuentra en el archivo Index.html.

Para llevar a cabo el recorrido virtual, se plantearon varios aspectos a tener en cuenta que se listan a continuación:

- Colisión entre objetos. Como su nombre lo dice, esta parte de la programación nos servirá para ver cómo interactúan dos o más objetos cuando chocan, para este caso la cámara no deberá atravesar los muros y las puertas del edificio, así como también los límites de la escena.

- Ascensión por pendientes inclinadas. Para poder subir por las escaleras y saltar, fue necesario establecer un límite de altura de la cámara, para este caso todos los elementos que tengan una altura inferior a la cámara podrán ser escalados, mientras no se cumpla esta condición no podrán ser escalados y por lo tanto no podrán ser atravesados. Para lograr reproducir el salto, la altura de la cámara se dará en pequeños incrementos y decrementos para lograr dicho efecto (figura 5.9).

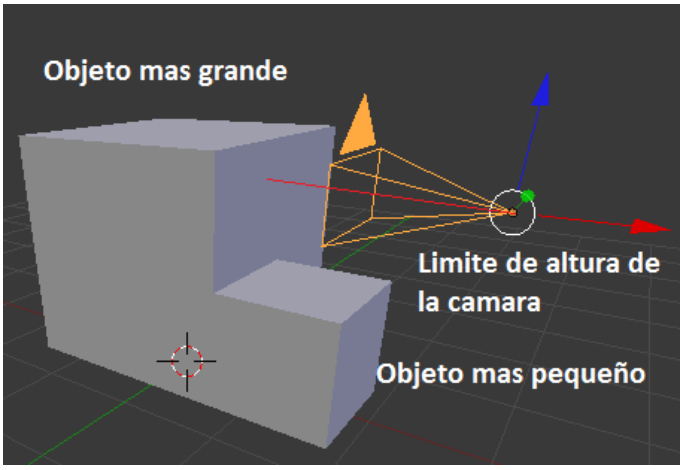


Figura 5.9. Ascensión por pendientes inclinadas.

A continuación se muestra el código necesario para realizar el recorrido virtual con los eventos (teclado y ratón) y los modelos ya definidos.

```
43
44
45  <script type="text/javascript">
46  var doc = new GLGE.Document();
47  doc.onLoad=function()
48  {
49      var Render=new GLGE.Renderer(document.getElementById('canvas'));
50      var dorado=doc.getElement("dorado");
51      escena=new GLGE.Scene();
52      escena=doc.getElement("paseoVirtual");
53      Render.setScene(escena);
54
55      var Raton=new GLGE.MouseInput(document.getElementById('canvas'));
56      var teclas=new GLGE.KeyInput();
57      var mouseovercanvas, objaux, objeto;
58      var entrar=true;
59
60      function raton()
61      {
62        if(mouseovercanvas)
63        {
64          var ratonPos=Raton.getMousePosition();
65          ratonPos.x=ratonPos.x-document.getElementById("canvas").offsetLeft;
66          ratonPos.y=ratonPos.y-document.getElementById("canvas").offsetTop;
67          var camara=escena.camera;
68          camararot=camara.getRotation();
69          inc=(ratonPos.y-(document.getElementById('canvas').offsetHeight/2))/500;
70
71          var trans=GLGE.mulMat4Vec4(camara.getRotMatrix(),[0,0,-1,1]);
72          var mag=Math.pow(Math.pow(trans[0],2)+Math.pow(trans[1],2),0.5);
73          trans[0]=trans[0]/mag;
74          trans[1]=trans[1]/mag;
75          camara.setRotX(1.56-trans[1]*inc);
76          camara.setRotZ(-trans[0]*inc);
```

```
 77         var width=document.getElementById('canvas').offsetWidth;
 78         if(ratonPos.x<width*0.3){//Izquierda
 79           var turno=Math.pow((ratonPos.x-
    width*0.3)/(width*0.3),2)*0.005*(tiempoActual-transcurrido);
 80           camara.setRotY(camararot.y+turno);
 81         }
 82         if(ratonPos.x>width*0.7){//Derecha
 83           var turno=Math.pow((ratonPos.x-
    width*0.7)/(width*0.3),2)*0.005*(tiempoActual-transcurrido);
 84           camara.setRotY(camararot.y-turno);
 85         }
 86         if(ratonPos.x && ratonPos.y)
 87         {
 88           objeto=escena.pick(ratonPos.x,ratonPos.y).object;
 89           if(objeto && (objeto!=objaux)){
 90               switch(objeto.getId())
 91               {
 92                 case "entrada": case "p0": case "p1": case "p2": case "p3":
 93                 case "p4": case "p5": case "p6": case "p7": case "p8":
 94                 case "p9": objeto.oldmaterial=objeto.getMaterial();
 95                   objeto.setMaterial(dorado);}
 96           if(objaux && objaux.getId()!=null)
 97           { switch(objaux.getId())
 98             { case "entrada": case "p0": case "p1":case "p2":case "p3":
 99               case "p4":case "p5":case "p6":case "p7":case "p8":
100               case "p9":objaux.setMaterial(objaux.oldmaterial);
101             }}  objaux=objeto;}
102       if(Raton.isButtonDown(GLGE.MI_LEFT))
103         {
104           Raton.element.buttonState[GLGE.MI_LEFT]=false;
105           switch(objeto.getId())
106           { case "entrada":
107               if(entrar==true){
108                 camara.setLocX(12);
109                 camara.setLocY(-15);
110                 camara.setRot(1.5907537478599434, 1.264062741662344, -
    0.09725331353001312);
111                 entrar = false;}
112               else
113               { camara.setLoc(12,-23.088883008171788,1.8931698069852942);
114
    camara.setRot(1.6219801508128817,9.408716019440131,0.001568727258781407);
115                 entrar=true;}
116             break;
117           case "p1":$( "#dialogo-materiales" ).dialog("open");break;
118           case "p2":$( "#dialogo-ambiental" ).dialog("open");break;
119           case "p3":$( "#dialogo-manufactura" ).dialog("open");break;
120           case "p4":$( "#dialogo-seguridad" ).dialog("open");break;
121           case "p5":$( "#dialogo-medicion" ).dialog("open");break;
122           case "p6":$( "#dialogo-urbana" ).dialog("open");break;
123           case "p7":$( "#dialogo-computo" ).dialog("open");break;
124           case "p8": case "p9":$( "#dialogo-video" ).dialog("open");break;
125           }}}}}
126
127   function teclado(){
128     var camara=escena.camera;
129     var camarapos=camara.getPosition();
130     var camararot=camara.getRotation();
131     var mat=camara.getRotMatrix();
132     var trans=GLGE.mulMat4Vec4(mat,[0,0,-1,1]);
```

```
133     var mag=Math.pow(Math.pow(trans[0],2)+Math.pow(trans[1],2),0.5);
134     trans[0]=trans[0]/mag;
135     trans[1]=trans[1]/mag;
136     var yinc=0;
137     var xinc=0;
138
139     if(teclas.isKeyPressed(GLGE.KI_UP_ARROW))
    {yinc=yinc+parseFloat(trans[1]);xinc=xinc+parseFloat(trans[0]);}
140     if(teclas.isKeyPressed(GLGE.KI_DOWN_ARROW)) {yinc=yinc-
    parseFloat(trans[1]);xinc=xinc-parseFloat(trans[0]);}
141     if(teclas.isKeyPressed(GLGE.KI_LEFT_ARROW))
    {yinc=yinc+parseFloat(trans[0]);xinc=xinc-parseFloat(trans[1]);}
142     if(teclas.isKeyPressed(GLGE.KI_RIGHT_ARROW)) {yinc=yinc-
    parseFloat(trans[0]);xinc=xinc+parseFloat(trans[1]);}
143     if(teclas.isKeyPressed(GLGE.KI_SPACE)) {if(salto==0) salto=35;}
144
145     if(xinc!=0 || yinc!=0){
146       var origen=[camarapos.x,camarapos.y,camarapos.z];
147
148       dist2=escena.ray(origen,[-xinc,0,0]);
149       dist3=escena.ray(origen,[0,-yinc,0]);
150       if(dist2.distance<5) xinc=0;
151       if(dist3.distance<5) yinc=0;
152       //Movimiento Camara
153       if(xinc!=0 || yinc!=0){
154         camara.setLocY(camarapos.y+yinc*0.030*(tiempoActual-
    transcurrido));camara.setLocX(camarapos.x+xinc*0.030*(tiempoActual-transcurrido));
155         if(salto==0) salto=0.001;}
156     }}
157
158   var transcurrido=0;
159   var salto=-1;
160   var altura=-98;
161   empieza=parseInt(new Date().getTime());
162   transcurrido=empieza;
163   var tiempoActual;
164   function render()
165   {
166     tiempoActual=parseInt(new Date().getTime());
167     raton();
168     teclado();
169
170     if(salto!=0 || tiempoActual%10==0)
171     {
172       salto+=altura*(tiempoActual-transcurrido)/1000;
173       var camarapos=escena.camera.getPosition();
174       var origen=[camarapos.x,camarapos.y,camarapos.z];
175       var vector=escena.ray(origen,[0,0,1]);
176       var dist=vector.distance;
177       inc=salto*(tiempoActual-transcurrido)/1000;
178       if(dist+inc<1.8){inc=1.8-dist;salto=0;}
179       if(inc<0.01&& inc>-0.01) inc=0;
180       if(!vector.object){inc=0;}
181       else if(vector.object.parent.animation){
182         if(dist<10) salto=11;
183       }
184       escena.camera.setLocZ(camarapos.z+inc);
185     }
186
187     Render.render();
```

```
188       transcurrido=tiempoActual;
189     }
190     setInterval(render,1);
191     var inc=0.5;
192     document.getElementById("canvas").onmouseover=function(e){mouseovercanvas=true;}
193     document.getElementById("canvas").onmouseout=function(e){mouseovercanvas=false;}
194   }
195   doc.load("Centro.xml");
196   </script>
197
```

Comparando el código anterior con el ejemplo de GLGE del capítulo anterior, se puede observar que se utilizaron nuevas clases que componen GLGE, por ejemplo la clase GLGE.Scene (línea 51) que nos servirá para leer y obtener elementos que se definieron en el archivo Centro.xml, asi como las clases GLGE.MouseInput (línea 55) y GLGE.KeyInput (línea 56) que nos servirán para detectar los eventos y ejecutar el código necesario del recorrido. Este nuevo código se simplifica en tres funciones importantes que se explican a continuación.

- Función raton() (Lineas 60-125). Esta función será la encargada de cumplir tres propósitos del recorrido. El primer propósito será el de realizar las operaciones necesarias para rotar la cámara sobre los ejes correspondientes (líneas 71-85), la ejecución de esta parte del código se realizará cuando el usuario mueva el cursor del ratón en el canvas definido. El segundo propósito (líneas 86-101) será el de apuntar los objetos definidos en el archivo Centro.xml dentro de la escena, a medida que el usuario señale las puertas, inmediatamente se cambiara el tipo de material del objeto por uno de color dorado, asi como también al desapuntar los elementos señalados volverán a su estado anterior. El último propósito (líneas 102-125) servirá para atrapar el evento “click” del botón izquierdo del ratón y mostrar la información necesaria de cada de uno de los laboratorios definidos.

La información mostrada se auxilia de la biblioteca *jQuery UI* que sirve principalmente para la creación de efectos visuales en páginas web, esta biblioteca se puede descargar de la siguiente dirección:

```
http://www.jqueryui.com
```

- Funcion teclado() (Líneas 127-156). Esta función será la encargada de ejecutar 2 funciones primordiales, la primera será la de calcular las operaciones necesarias para trasladar la cámara (lineas128-143) a través de los eventos que se registrarán (líneas 139-143). El segundo propósito será el de analizar la colisión de la cámara (líneas 145-146) con todos los elementos que conforman la escena.

- Funcion render() (Lineas 164-189). Esta función será la encargada de llamar a las funciones anteriores (líneas 167-168) así como realizar los cálculos para subir por las escaleras. La función render se llamará indefinidamente por medio de la función setInterval(funcion,milisegundos) (línea 190, definida en el lenguaje JavaScript) y que nos servirá para actualizar todas las posiciones y rotaciones de la cámara, así como la actualización de la vista de nuestra escena.

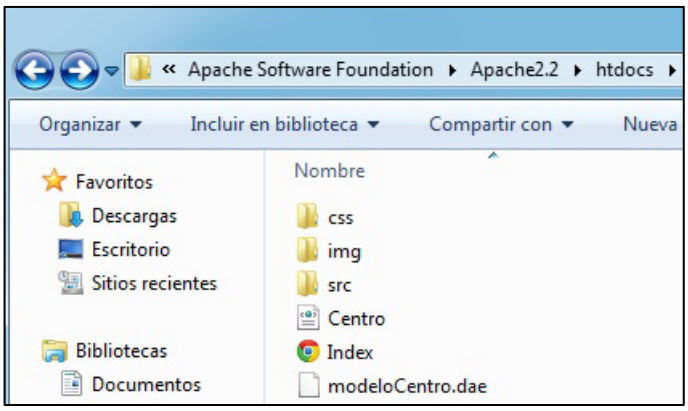
Para poder funcionar, GLGE requiere de un servidor web, el cual es un programa que procesa una aplicación del lado del servidor, para después generar una respuesta del lado del cliente. Para este propósito usaremos el servidor Apache, la última versión disponible para Windows es la 2.2.22 y podemos descargar de la siguiente ruta:

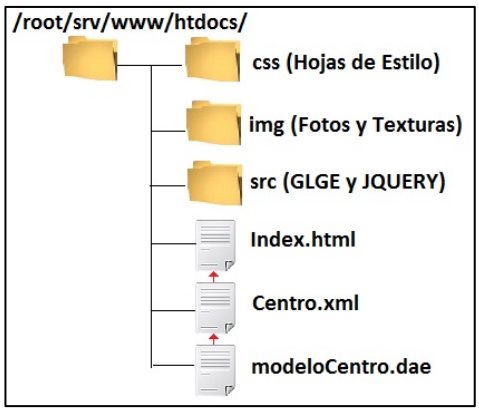
```
http://httpd.apache.org/download.cgi#apache24
```

Para instalarlo en Linux, debemos abrir una consola, identificarnos como root y escribir el siguiente comando:

```
zipper install apache2
```

Una vez descargado e instalarlo, hay que proceder a copiar los archivos que creamos anteriormente o copiarlos de la carpeta “3-Caso Practico” que se encuentra dentro del CD que acompaña a este trabajo y colocarlos en la carpeta *htdocs* de Apache que está ubicada en la ruta: *C:\Program Files\Apache Software Foundation\Apache2.2\htdocs* y en Linux en la ruta: */root/srv/www/htdocs*, desde la cual apache podrá cargar el modelo en el navegador (como lo muestra la siguiente captura).





Ahora solo hay que abrir un navegador compatible con WebGL y en la barra de direcciones escribir: *localhost*, así se podrá tener acceso a los archivos de Apache y el navegador comenzará a correr el paseo virtual del centro tecnológico.

Para esta ocasión, un aspecto importante a considerar es la tarjeta gráfica que se va a usar para ejecutar el paseo virtual. Como pudimos ver en al capítulo anterior, el vertex shader y el fragment shader juegan un papel muy importante en la programación y el funcionamiento de WebGL, por lo que contar con una GPU con dichas características es de suma importancia.

A diferencia de los ejemplos del capítulo anterior que eran muy sencillos y podían funcionar fácilmente en cualquier computadora, inclusive en algunas muy básicas con tarjetas gráficas Intel, esta vez si es necesario contar con un chip de video que cuente con una buena arquitectura de shaders, ya que el paseo virtual carga una gran cantidad de datos de sombreado y texturas.

Para poder averiguar si nuestro GPU cuenta con esta característica, podemos usar un programa llamado “GPU-z”, el cual muestra todas las especificaciones técnicas de la tarjeta de video, lo encontramos en la siguiente dirección:

```
http://www.techpowerup.com/gpuz/
```

Al ejecutar el programa podemos ver en el recuadro “*Shaders”* la cantidad de unidades de cálculo de shaders presentes en la Tarjeta de Video (Figura 5.10), en

los GPU más antiguos se muestran las unidades de cálculo por separado (Pixel y Vertex), mientras que en las tarjetas mas modernas se muestra el valor seguido de la palabra “Unified”, el cual hace referencia al *Unified Shading Architecture*, que es una arquitectura compuesta por un conjunto de unidades de cálculo capaces de procesar cualquier tipo de shader, ya sea pixel o vertex. Por ende entre más unidades de cálculo de shaders tenga la tarjeta gráfica, tendremos menos problemas en el funcionamiento del paseo virtual (como el ralentizado).

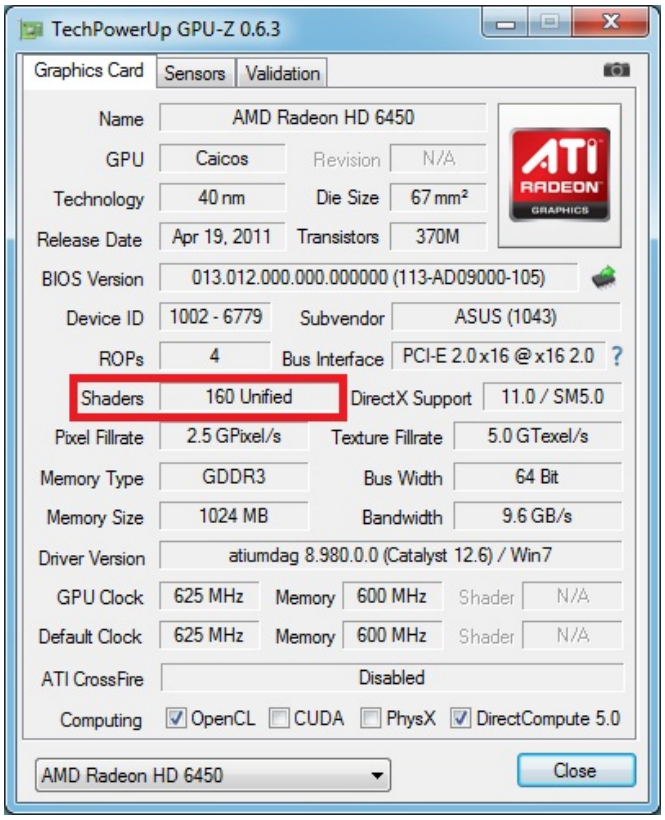


Figura 5.10. GPU-z mostrando las especificaciones de una tarjeta gráfica.

Aunque es necesario mencionar la importancia de la tarjeta gráfica y su soporte de shaders, también es cierto que la gran mayoría de las computadoras modernas cumple con esta característica, por lo que no debería representar mucho problema. Al hacer este trabajo, se usaron principalmente tres computadoras para realizar las pruebas del pasea virtual, cuyas especificaciones se listan a continuación.

Equipo 1	- Procesador Intel Core 2 Duo E8400 3.0Ghz - Tarjeta madre ASRock G31M-VS2 - 2 GB de memoria DDR2 - Disco duro de 160GB SATA II - Tarjeta gráfica NVidia GeForce 210
Equipo 2	- Procesador Intel Pentium G620 2.6Ghz - Tarjeta madre Gigabyte GA-H61M-S1 - 4 GB de memoria DDR3 - Disco duro de 250GB SATA II - Tarjeta gráfica AMD Radeon HD 6450
Equipo 3	- Procesador AMD E-300 1.3Ghz - Tarjeta madre Lenovo G475 - 2 GB de memoria DDR3 - Disco duro de 320GB SATA II - Tarjeta gráfica AMD Radeon HD 6310M

Ejecutando el paseo virtual

Ahora que ya fue considerado lo anterior, solo resta abrir el navegador y escribir: *localhost* en la barra de direcciones, al hacerlo se empezará a cargar el modelo, las texturas, las luces y al concluir ya podemos movernos por el paseo virtual, para hacerlo debemos usar las teclas ↑, ↓, ←, →; para mover la vista o cámara se usa el mouse y para brincar se usa la barra espaciadora.

Figura 5.11. Paseo virtual por el exterior del centro tecnológico.

Al movernos libremente por el diseño, podrá notarse que al pasar el puntero por la puerta principal ésta cambia a un color dorado, indicando que podemos interactuar con ella al hacer clic y al hacerlo se nos envía automáticamente al interior del edificio como se puede apreciar en la figura 5.12.

Figura 5.12. Paseo virtual por en interior del centro tecnológico.

Una vez en el interior de centro tecnológico, podemos notar que al igual que en la entrada principal, las puertas del interior cambian de color al pasar el puntero sobre alguna de ellas, indicándonos que hay información disponible sobre ese laboratorio y al hacer clic en una de estas puertas, se abre una nueva venta en la cual se muestra el nombre del laboratorio, una fotografía y las actividades que se realizan en él (Figura 5.13).

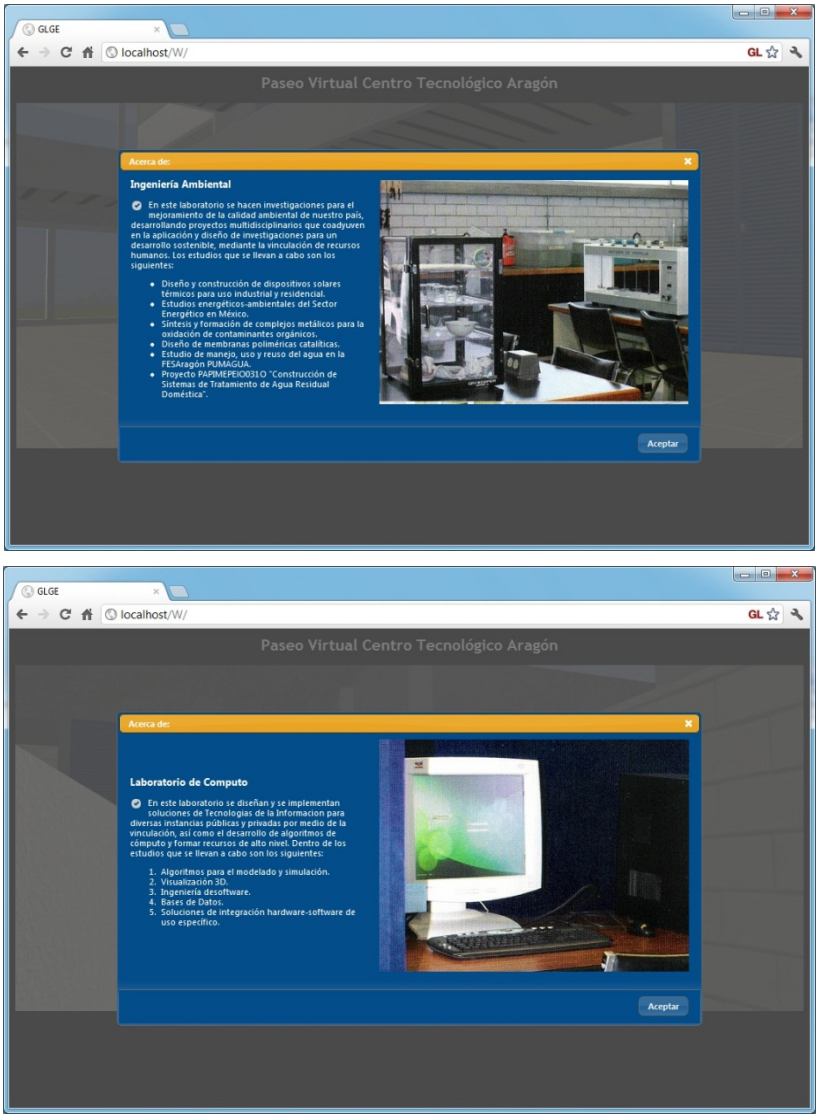


Figura 5.13. Ventanas mostrando la descripción de los laboratorios.

Con el presente capítulo concluimos este trabajo, en el cual pudimos aprender muchos conceptos relacionados con los gráficos por computadora y por supuesto fue posible conocer a profundidad la tecnología WebGL con el desarrollo del caso práctico, el cual cumplió con el objetivo principal de sacar provecho a WebGL, que a pesar de ser un tecnología relativamente nueva, promete mucho en cuanto a un internet más interactivo.

CONCLUSIONES

Después de haber finalizado el desarrollo de este trabajo de tesis, podemos concluir que WebGL es un estándar que está creciendo rápidamente y cuyo uso se ha ampliado considerablemente; es una tecnología 3D muy prometedora, ya que se encuentra apoyado por grandes empresas como Google, Apple, Mozilla y otros grandes proveedores de hardware, lo que indica un gran interes hacia su evolución por parte de los desarrolladores web.

Con el crecimiento de internet, el navegador web es sin duda una de las herramientas mas utilizada a nivel mundial, tanto ha crecido esta herramienta que hoy se tiene la posibilidad de manejar gráficos en 3D y no solo eso, si no que tambien grandes corporaciones han llevado su software a una nueva plataforma llamada “Cloud computing” o computación en la nube y en ella se puede acceder una infinidad de servicios, desde editores de texto online, administradores de bases de datos y hasta alojamiento de proyectos de desarrollo de software haciendo uso sólo del navegador.

Las posibilidades que brinda WebGL a los usuarios es una revolución total, ya que aplicaciones 3D que antes sólo eran posibles mediante su instalación en cada computadora, ahora son posibles en la web, con lo cual se puede ampliar considerablemente el usos de aplicaciones interactivas en tercera dimensión ya que todas las personas cuentan con un navegador web instalado en sus computadoras.

Definitivamente al tener la oportunidad de redactar este trabajo de tesis, fue posible aprender mucho sobre todo lo relacionado a los gráficos por computadora, ya que no era posible hablar de WebGL sin tener que considerar la historia de los gráficos tridimensionales o estudiar conceptos técnicos relacionados con el tema, sin los cuales no hubiera sido posible entender temas más avanzados, como la programación en OpenGL vista en el capítulo tres; todo esto al unirse fue esencial para comprender mejor la forma en que WebGL funciona.

Unos de los principales problemas con los que nos encontramos al desarrollar este trabajo, fue la poca información disponible sobre WebGL, debido principalmente al poco tiempo que tiene de haber sido liberada la primera especificación de esta tecnología; además de que la poca información que se puede encontrar sobre este tema se encontraba en otros idiomas y la documentación oficial es casi inexistente, así que en ocasiones resultaba mejor buscar sitios de internet en donde se mostraran ejemplos y en base a ellos poder estudiar su código fuente y saber como es que fueron hechos.

A pesar de los problemas con los que nos encontramos, para el capítulo final se puso en práctica todo lo aprendido anteriormente mediante el desarrollo de una aplicación de ejemplo que consta de un paseo virtual del centro tecnológico Aragón; por supuesto aunque ya se contaba con el conocimiento obtenido en los capítulos anteriores, se presentaron algunas dificultades, como la compatibilidad de tarjetas gráficas o problemas con algunos navegadores web, con los cuales al momento de trabajar para intentar resolverlos, se tuvo la oportunidad de aprender nuevos conceptos relacionados con el tema, por lo que a final de cuentas se logró completar el proyecto de acuerdo a lo que se había planeado.

Por todo lo anterior podemos afirmar que los objetivos planteados al inicio de este trabajo de tesis se cumplieron de forma satisfactoria.

BIBLIOGRAFÍA

● Hearn, Donald. Gráficos por computadora con OpenGL. Prentice Hall. México Tercera Edición, 2005.

● Munshi, Aaftab. OpenGL ES 2.0 Programming Guide. Addison-Wesley. United States First Edition, 2008.

● Wolff, David. OpenGL 4.0 Shading Lenguage Cookbook. Packt Publishing. United Kingdom First Edition, 2011.

● Williams, James L. Learning HTML5 Game Programming. Addison-Wesley. United States First Edition, 2011.

● Wright, Richard S. OpenGL SuperBible. Addison-Wesley. United States Fifth Edition, 2011.

● Pilgrim, Mark. HTML5 Up and Running. Google Press. United States First Edition, 2010.

● Lubbers, Peter. Pro HTML5 Programming. Apress. United States First Edition, 2010.

● Rost, Randi J. OpenGL Shading Lenguage. Addison-Wesley. United States Third Edition, 2010.

REFERENCIAS DE INTERNET

●WebGL: GPU acceleration for the open web (Consultado en noviembre 2011): http://www.slideshare.net/pjcozzi/webgl-gpu-acceleration-for-the-open-web

●3D in Flash (Consultado en octubre 2011): http://www.go2file.com/790/3d-in-flash.aspx

●What is Web 3D? (Consultado en septiembre 2011): http://www.domusinc.com/blog/2011/04/web-goes-3d-does-advertising-too-the-webgl-silverlight-and-molehill-wars/

●WebGL-A New Dimension for Browser Exploitation (Consultado en febrero 2012): http://www.contextis.com/resources/blog/webgl/

●WebGL samples (Consultado en septiembre 2011):
http://code.google.com/p/webglsamples/

●WebGL examples (Consultado en septiembre 2011):
http://www.ibiblio.org/e-notes/webgl/webgl.htm

●Presentación en WebGL (Consultado en diciembre 2011):
http://fhtr.org/webgl_presentation.html#19

●University Campus Koblenz in WebGL (Consultado en enero 2012):
http://erkunden.uni-koblenz.de/icw/en

●Blender Wiki (Concultado en marzo 2012):
http://wiki.blender.org/index.php/Doc:ES/2.6/Manual

●Modelando en Blender (Consultado en enero 2012):
http://www.foro3d.com/f111/modelando-calabaza-con-blender-69751.html

●GSLS 1.2 Tutorial (Consultado en enero 2012):
http://www.lighthouse3d.com/tutorials/glsl-tutorial/pipeline-overview/

●Modelar arquitectura Blender 2.55b (Consultado en marzo 2012):
http://foroarqsl.creatuforo.com/gua-breve-gua-para-modelar-arquitectura-en-blender-255b-tema36.html

●Quake 3 WebGL Demo (Consultado en febrero 2012):
http://media.tojicode.com/q3bsp/

●Three.JS Walking Map (Consultado en abril 2012):
http://ushiroad.com/3j/

●All about OpenGL ES 2.x (Consultado en diciembre 2011):
http://db-in.com/blog/2011/02/all-about-opengl-es-2-x-part-23/

●Getting Started With Three.js (Consultado en abril 2012):
http://www.aerotwist.com/tutorials/getting-started-with-three-js/

●COLLADA & WebGL (Consultado en marzo 2012):
http://www.slideshare.net/remi_arnaud/collada-webgl

●WebGL/GLSL Sandbox (Consultado en abril 2012):
http://ricardocabello.com/blog/post/714

●WebGL: What Flavor Is Your Engine? (Consultado en marzo 2012):
http://ffwd.typepad.com/blog/2011/04/webgl-what-flavor-is-your-engine.html

●COLLADA (Consultado en febrero 2012):
http://en.wikipedia.org/wiki/COLLADA

●Creando un videjuego en WebGL: ZombieBus FIC (Consultado en febrero 2012):
http://sabia.tic.udc.es/gc/trabajos%202011-12/ZombiBus/index.html

●El proceso de visualización en OpenGL (Consultado en noviembre 2011):
http://graficos.uji.es/grafica/Tutorial/Proyecciones.htm

●Iluminación y Sombreado (Consultado en noviembre 2011):
http://sabia.tic.udc.es/gc/teoria/iluminacion/Index.html

●Texturas en 3D (Consultado en noviembre 2011):
http://sabia.tic.udc.es/gc/teoria/texturas/Indice.htm

●WebGL – Benchmark (Consultado en enero 2012):
http://martin.cyor.eu/benchmark/test.html

●Tutoriales OpenGL - Andromeda Studios (Consultado e noviembre 2011):
http://usuarios.multimania.es/andromeda_studios/paginas/tutoriales/aptutgl01.htm

●Introduction to 3D and shaders (Consultado en diciembre 2011):
http://www.slideshare.net/victorporof/introduction-to-3d-and-shaders-4922024

●Documento sobre DirectX (Consultado en septiembre 2011):
http://www.jeuazarru.com/docs/DirectX.pdf

●Presentación sobre DirectX (Consultado septiembre 2011):
http://www.slideshare.net/caiw1023/directx

●DirectX y OpenGL (Consultado en octubre 2011):
http://www.slideshare.net/guest5506a9/direct-xy-open-gl-presentation?src=related_normal&rel=4993963

●Presentación sobre COLLADA & WebGL (Consultado en marzo 2012):
http://www.slideshare.net/remi_arnaud/collada-webgl