Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Construcción de mundos virtuales y la navegación a través de ellos

Tesis que presenta

Guillermo Augusto Sánchez Sánchez

para obtener el Grado de

Maestro en Ciencias

en la Especialidad de

Computación

Director de la Tesis

Dr. Luis Gerardo de la Fraga

México, D.F. **Septiembre 2014**

Resumen

En este trabajo se construyó una solución para el problema de la construcción y navegación de mundos virtuales, mapeando las funciones principales que un usuario ocuparía para dichas tareas a un dispositivo de seis grados de libertad, el Wiimote. Explotando su cámara infrarroja, botones y giroscopios se creó un proceso natural para la construcción y navegación de un mundo virtual.

La inserción de los objetos tridimensionales se realiza desde un repositorio de modelos que cuenta con objetos primitivos, tales como cubos, esferas, conos y toroides, al igual que algunos objetos compuestos como árboles, permitiendo al usuario crear un mundo virtual sin necesidad de tener conocimientos sobre modelado de contenidos tridimensionales.

Al tratarse de una cantidad considerable de vértices simultáneos los que conforman cada mundo virtual, se optimizó la visualización de tal forma que sólo se realice el procesamiento necesario para los objetos que se encuentran dentro del campo de visión, permitiendo de esta forma, tener en memoria grandes mundos virtuales sin que se vea afectado el rendimiento de la visualización.

La manera de interactuar con el software es mediante una interfaz gráfica de usuario hecha en QT y OpenGL que responde a los eventos del Wiimote, de este modo un usuario novato puede adecuarse rápidamente al uso del software y en pocos minutos tener resultados que pueden ser guardados en disco para una posterior recuperación y edición.

Agradecimientos

Al Centro de Investigación y Estudios Avanzados del I.P.N.
por permitirme y enseñarme a formar parte de algo.

Al Consejo Nacional de Ciencia y Tecnología
por la aportación económica que me proporcionó y ayudó durante mis estudios.

Al Dr. Luis Gerardo de la Fraga
por el invaluable conocimiento que día con día me ofreció.

A todos en mi vida
por compartir su tiempo conmigo.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Existen diversas formas de definir un mundo virtual, cada una de ellas en función de la disciplina o materia que lo está delimitando pero tomaremos la definición general que se presenta en [1] que indica lo siguiente: "Un mundo virtual es un entorno artificial creado dentro de una computadora, el cual imita algunos aspectos del mundo real". Esta definición general envuelve de forma concisa lo que es un mundo virtual aunque también se puede definir como: "Los mundos virtuales usan la simulación para crear un espacio 3D completamente inmersivo, en el cual los usuarios pueden interactuar y colaborar en tiempo real"[2], aunque hay que mencionar que esta definición está planteada desde el punto de vista de sistemas multiusuario y videojuegos. También es importante notar que en esta definición nunca se menciona que un mundo virtual debe estar basado en algunos aspectos del mundo real.

Parafraseando a estas dos definiciones, tenemos que un mundo virtual es un entorno artificial mostrado en tiempo real por una computadora, en el cual al menos un usuario puede interactuar y el mundo virtual puede estar basado o no en la realidad.

En épocas tempranas de la era de la computación los mundos virtuales eran inexistentes debido al limitado poder de cómputo de la época. Alguna salida visual era inexistente y las computadoras solían abarcar pisos enteros de cables, interruptores, bulbos y cintas magnéticas.

Fue en el año de 1969 cuando la ACM creó un Grupo Especial de Interés en Gráficos (SIGGRAPH por sus siglas en inglés) que organizaba conferencias, creaba estándares de gráficos y publicaciones relativas a los gráficos por computadora, pero no fue hasta 1973 que se llevó a cabo su primer conferencia anual.

Con el avance de la computación y la invención de los microprocesadores, al menos en un principio, la situación no cambio drásticamente. En la década de 1970 solamente un minúsculo porcentaje de las empresas se dedicaba a la investigación y uso de los gráficos por computadora, sin mencionar que los mundos virtuales aún no eran desarrollados ni explotados [3]. Los frutos de dichas investigaciones se explotarían

hasta la siguiente década de 1980. Por mencionar un ejemplo, en 1982 la industria ofreció uno de los más memorables, aunque aún primitivos mundos virtuales en pantalla, creado para la película TRON de Walt Disney donde el fondo de pantalla verde es reemplazado por un mundo virtual creado en la computadora. Otro de ellos es el corto animado nombrado *The Adventures of André and Wally B* de Lucasfilm Computer Graphics Project lanzado en 1984, donde se muestra un mundo virtual basado en cilindros, planos y cónicas simulando un bosque.

Aunque estos dos ejemplos no fuesen en tiempo real ni permitieran algún tipo de interacción dejan en claro que la capacidad de crear mundos virtuales durante la década de 1980 sólo estaba al alcance de las grandes compañías de animación debido a sus costos altos.

Esto cambió en el siguiente par de décadas. Durante la decada de 1990 y 2000 el abaratamiento del hardware y el desarrollo de nuevo software permitió a los dueños de computadoras personales crear sus propios mundos virtuales, el software 3D Studio (en 1990) fue de los primeros paquetes que permitían la creación de contenidos 3D en una computadora de escritorio siendo precedido por Blender 3D (en 1995) y Maya (en 1998).

Siendo estrictos, estos paquetes antiguamente permitían la construcción de mundos virtuales pero no su navegación como tal, pues para ello requieren de un procesamiento previo (render en inglés) para su visualización correcta.

Posiblemente, si volteamos a ver la rama de los videojuegos por computadora, podamos observar algo más cercano a la interacción con mundos virtuales en tiempo real. Sin embargo, para permitir el despliegue del mundo en pantalla de forma coherente, de al menos 24 cuadros por segundo, se tienen que sacrificar algunos aspectos de calidad, entre los cuales hay: limitar la cantidad de polígonos por segundo en pantalla, acotar la cantidad y tamaño de las texturas, reestringir la cantidad de elementos en la pantalla a la vez, mostrar sombras de mala calidad, evitar superficies reflejantes, evitar visualizar fluidos, etc.

Hoy en día se cuenta con procesadores gráficos dedicados que implementan operaciones gráficas a nivel de hardware, permitiendo así un rendimiento bastante excepcional que nos permite presentar un mundo virtual con grandes cantidades de polígonos y una gran cantidad de texturas por cada cuadro mostrado en pantalla.

1.1. Planteamiento del problema

Actualmente, los mundos virtuales son el corazón de varias piezas de software que tienen bastantes usuarios. Por mencionar algunos, tenemos a los recorridos virtuales

de edificios o museos, videojuegos, software educativo, simuladores para entrenamiento militar, películas de animación, entre otros [4] [1].

Toda la industria que gira en torno a los mundos virtuales genera una gran cantidad de dinero. Según [5], simplemente en Europa en el año 2010, se generaron $1,019 millones de dólares en ventas de bienes virtuales. En este momento, es necesario definir lo que es un bien virtual. Estos son servicios o productos virtuales para plataformas en línea o videojuegos, tales como saldo, suscripciones o, los que son relativos a esta tesis, inmuebles o productos virtuales. En otras palabras, se compra con dinero real objetos virtuales que usa un personaje o avatar. Los objetos adquiridos son cosas tales como muebles, edificios, ropa, vehículos, plantas, etc.

Pero la construcción de aplicaciones y de entornos 3D ha sido tradicionalmente una tarea tediosa, tardada y difícil requiriendo un alto nivel de experiencia de parte del programador o diseñador [6].

El software que se encuentra actualmente en el mercado deja la tarea de construcción del mundo virtual plenamente al artista de contenidos 3D, tales como Blender, Maya, Ogre, etc. Además de estar atado al proceso de ingeniería del software basado en el desarrollo en cascada donde el artista 3D espera los bosquejos del mundo virtual (etapa de preproducción) para después plasmarlos en el software de diseño 3D plasmando el mundo virtual (etapa de producción).

Este proceso es lento y poco creativo para el artista 3D además de que las herramientas mencionadas no cuentan con instrumentos que faciliten el trabajo en las partes más tediosas como son la creación de texturas, posicionamiento de elementos en el mundo, creación de árboles y arbustos, o diseño de una ciudad grande que contenga edificios similares.

Añadiendo que únicamente los artistas 3D son los capaces de poder modificar el mundo virtual una vez ya creado dejando de lado la posibilidad de que algún miembro del equipo que no sea un artista pueda plasmar sus ideas en el proceso creativo, a menos que sea mediante el artista dando cabida a un filtrado de la idea o a una malinterpretación.

Por otro lado, algunos problemas planteados en [7] tratan sobre la dificultad que tienen los desarrolladores en las primeras fases de los proyectos. Éstos señalan que los entornos de trabajo tienen una curva de aprendizaje bastante prolongada. Las razones son diversas, entre las cuales existen: la dificultad en conseguir el funcionamiento de los dispositivos 3D, los costos altos de los paquetes 3D y de los dispositivos 3D, la documentación no es clara o no es accesible, la interfaz de usuario 3D no es clara o la información no permite un trabajo eficiente.

Cabe mencionar los problemas que el mismo artículo reporta pero ahora en fases intermedias del desarrollo. Entre las que se mencionan están: el rendimiento de la aplicación y del dispositivo 3D son pobres, cada característica 3D añadida al proyecto aumenta la complejidad, limitando el rendimiento y haciendo el desarrollo difícil, probar la aplicación con el ratón y el teclado retrasa el desarrollo del proyecto. Estos problemas señalados están fuertemente relacionados al trabajo de tesis.

Otro de los problemas que se abarcará en la tesis es la visualización, El problema radica en la forma en que OpenGL presenta los gráficos en pantalla. De forma tradicional, OpenGL procesa todos los vértices que existen en el mundo virtual, aun si éstos no están en la escena que se mostrará en la pantalla. Esto último demerita el rendimiento de las aplicaciones cuando existe una cantidad grande de vértices, y evita la construcción de mundos virtuales con un número relativamente mayúsculo de objetos.

1.2. Estado del arte

En la actualidad se han desarrollado varios dispositivos para la navegación de los mundos virtuales para lidiar los problemas que esto representa. Entre estos dispositivos se pueden mencionar los presentados en [8], el Globefish y el Groovepad, o también el hardware desarrollado por las compañías de software Nintendo, Sony y Microsoft, es decir, el Wiimote, el PlayStation Move y el Kinect, respectivamente. Cada uno de estos dispositivos tiene ventajas y desventajas atacando cierto subproblema en específico, entre los cuales existen, sociales, intuitivos, contextuales o de retroalimentación.

En la figura 1.1 se puede apreciar un Wiimote de marca Steren. Aunque sea una marca sin licencia cedida por Nintendo para producir los Wiimote, la marca Steren produjo un control completamente compatible con la consola de videojuegos Wii, tiene las mismas capacidades que el control fabricado por Nintendo y se percibe como un control de la marca original.

Existen dos tipos de Wiimote realizados por Nintendo. Uno de ellos es el primer modelo que salió a la venta en el 2006, el cual cuenta con giroscopios para determinar la inclinación del control (pitch, roll y yaw). Además, tiene la capacidad de detectar la aceleración del desplazamiento a lo largo de los tres ejes mediante un acelerómetro, tiene un altavoz integrado, cuatro leds de color azul en la parte baja, una cámara infrarroja monocromática en la parte superior, un puerto de expansión para otros dispositivos y doce botones en total. Se comunica por medio del Bluetooth, y su conexión y programación es relativamente sencilla.

El segundo modelo de Wiimote que existe en el mercado es el conocido como *Wii MotionPlus Inside*. Salió a la venta a finales del 2009, y éste cuenta con todas

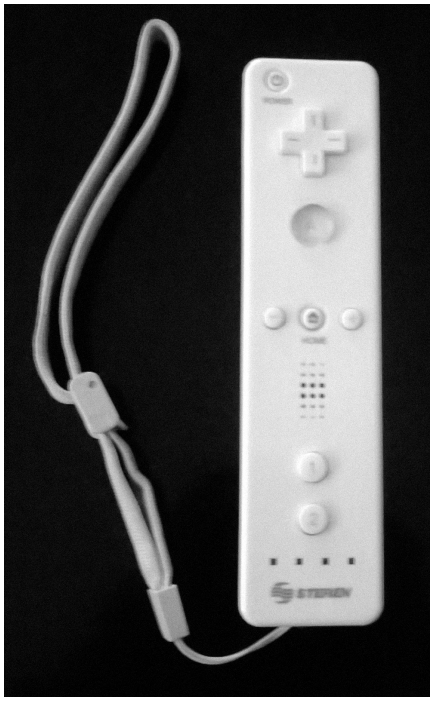


Figura 1.1: Un Wiimote de marca Steren con las mismas capacidades y funciones que un control de la marca Nintendo.

las capacidades del anterior pero añade un segundo acelerómetro para detectar las aceleraciones de los giros en el espacio (aceleración del pitch, roll y el yaw). De igual manera, se comunica por medio del Bluetooth pero ahora, como medida de seguridad, cada vez que se trata de enlazar con algún dispositivo que no sea una consola Wii, el control se apaga o evita el enlace. Su conexión y programación son algo más complejos que el primer Wiimote ya que no se encuentra tan bien documentado como el anterior.

Por otra parte, la compañía Nintendo para compensar a los consumidores que ya contaban con los Wiimote originales, lanzó al mercado un accesorio que se conectaba en el puerto de expansión. Este accesorio, llamado *Wii MotionPlus*, únicamente dota de las nuevas capacidades a los controles originales por un costo menor a un control Wii MotionPlus Inside.

En la actualidad no se encuentra alguna biblioteca que sea compatible con el sistema Mac OS X más reciente, las que se pueden obtener son bibliotecas que usan funciones descontinuadas del entorno de programación del Mac OS X dejándolas obsoletas.

En la figura 1.2 se enfoca la parte que estos dispositivos tienen distinta y sirve para diferencia uno de otro. En la parte izquierda se muestra un Wiimote original. En la parte del medio está un Wiimote original de marca Steren con un Wii MotionPlus

Figura 1.2: En la parte izquierda se muestra un Wiimote original. En la parte del medio está un Wiimote original marca Steren con un Wii MotionPlus debajo de él, mientras que en la parte derecha se encuentra un control Wii MotionPlus Inside.

debajo de él, mientras que en la parte derecha se encuentra un control Wii MotionPlus Inside.

En cuanto a funcionalidad, durante el **Symposium on 3D User Interfaces** que organiza la IEEE se lleva a cabo un concurso anual en marzo con un problema distinto a resolver en cada ocasión. En la octava edición de este simposio (2013) el reto a superar consistía en proponer una forma de construcción de un mundo virtual que mejore la experiencia del usuario usando la tecnología actual y se proporcionó una liga a un cortometraje de ciencia ficción que fungiera como inspiración [9]. La propuesta tenía que contar por lo menos con las capacidades de seleccionar modelos, manipularlos (traslación, rotación y escalamiento), adición de texturas a los modelos y modelos prefabricados [10].

Los ganadores del primer lugar fueron J. Wang, O. Leach y R. Lindeman por una interfaz de usuario que permitía todo lo anterior facilitando el trabajo de construcción, tal como se reporta en [11], usando el Wiimote para la interacción con el entorno y una tableta o teléfono inteligente para la selección de texturas y modelos. El software que usaron como base fue Untiy 3D (es un motor de desarrollo multiplataforma completamente integrado que proporciona la capacidad de crear juegos y otros contenidos 3D interactivos) [12].

Para navegar, desarrollaron una técnica llamada *CloudWlaker* la cual consiste en una serie de sensores que encuentran la posición del usuario relativa al centro de un círculo. Mientras más alejado esté del centro del círculo, más rápido será el desplazamiento en el mundo virtual. La dirección del desplazamiento está dada por la dirección en que apunta el Wiimote.

1.3. Objetivos de este trabajo

El sistema propuesto está conformado por una interfaz de usuario 3D, por una biblioteca de modelos y texturas, el uso del control Wiimote como dispositivo 3D y la optimización de la visualización que en conjunto solvente los problemas establecidos.

El objetivo principal del proyecto es crear un software que sea capaz de facilitar el trabajo creativo de la construcción de mundos virtuales que use el control del Nintendo Wii, el Wiimote, como dispositivo de entrada y una interfaz de usuario que sea amigable.

Los objetivos particulares se pueden envolver en cuatro grupos, los cuales son: biblioteca de modelos y texturas, interfaz gráfica de usuario, comunicación con el Wiimote y optimización de la visualización.

A continuación se describirá brevemente cada uno de estos elementos:

- Comunicación con el Wiimote. Se usará este dispositivo debido a que proporciona seis grados de libertad, doce botones, un altavoz, giroscopio para determinar la inclinación del control y es capaz de vibrar para ofrecer retroalimentación al usuario.

  Por el momento no existe alguna biblioteca libre para el sistema operativo Mac OS 10.9.3. Las que se encuentran son de pago o usan funcionalidades descontinuadas del marco de trabajo de Mac OS. Con este dispositivo de entrada el usuario será capaz de interactuar con la interfaz gráfica de usuario.

- Interfaz gráfica de usuario. Esta es la entidad más compleja del sistema planteado ya que será la encargada de fungir como intermediario entre el usuario y el repositorio de modelos y texturas mediante el Wiimote, gestionando los eventos que reciba del Wiimote, llevando a cabo las acciones pertinentes para que el usuario pueda crear e interactuar con el mundo virtual para posteriormente guardarlo y cargarlo.

  Su diseño se realizará en OpenGL y QT.

- Repositorio de modelos y texturas. Mientras más recursos se manejen en cualquier sistema se tiende a tener una forma de trabajo lenta y desorganizada, esta

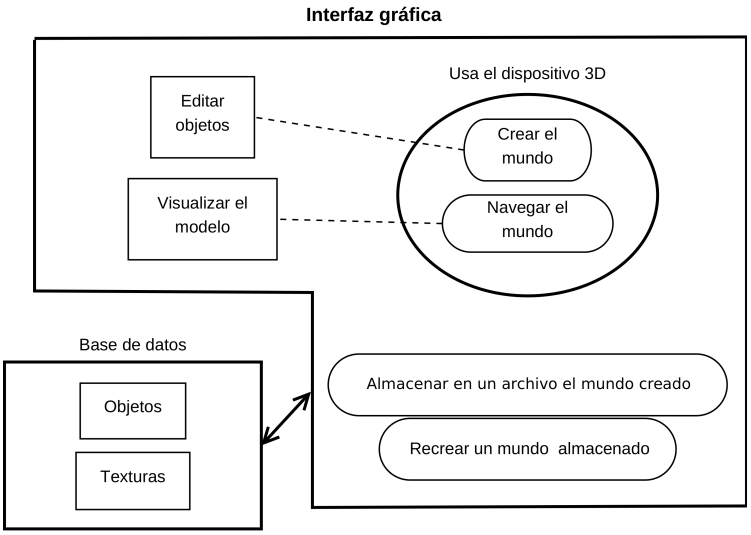


Figura 1.3: Diagrama que muestra el funcionamiento a un alto nivel del sistema propuesto. En la esquina inferior izquierda se encuentra la base de datos de texturas y modelos. El otro bloque representa la interfaz gráfica de usuario y sus elementos.

entidad mantendrá de forma coherente los recursos con los que se construirán los mundos virtuales.

- Optimización de la visualización. Finalmente está la optimización de visualización, debido a que OpenGL procesa todos los vértices que se encuentran en la escena es posible mejorar este concepto evitando que sean procesados los vértices que no estén enfrente de la cámara.

En la figura 1.3 se pueden observar estos cuatro elementos, su funcionalidad básica, y la interacción entre ellos. En la esquina inferior izquierda se encuentra la base de datos de texturas y modelos, el otro bloque representa la interfaz gráfica de usuario y sus elementos.

En la misma figura se puede observar cómo es que la interfaz gráfica realiza peticiones a la base de datos de modelos y texturas, cuando se requiera de éstos, para agregarlos al mundo virtual. Dicha interfaz gráfica será la encargada de grabar el mundo virtual creado en un archivo para su posterior reproducción y edición.

Los eventos producidos por el control de Wiimote serán administrados por la interfaz gráfica de usuario permitiendo al usuario crear y navegar a través del mundo virtual.

A continuación se explica en forma de lista las funciones que se pretenden alcanzar con el sistema propuesto.

- Desarrollar un entorno de trabajo que permita al usuario crear mundos virtuales de forma rápida, sencilla e intuitiva ofreciéndole un conjunto de instrumentos que le permitirá guardar y cargar sus sesiones de trabajo así como también cargar a su biblioteca de objetos cualquier modelo creado en el formato estándar obj.

- Estudiar y crear una estructura de datos para presentar en pantalla únicamente los objetos que serán visualizados y no todos los que conforman el mundo virtual teniendo ganancia en el rendimiento.

- Permitir al usuario navegar en el mundo virtual creado usando el dispositivo de hardware Wiimote que está diseñado para este fin.

- El usuario podrá importar modelos en formato obj [13] hacia la biblioteca de modelos.

- El usuario podrá importar imágenes bmp hacia la biblioteca de texturas.

- Escribir una serie de rutinas para la creación de texturas procedurales y agregarlas a la biblioteca de texturas.

- Se podrá seleccionar desde la biblioteca de modelos algún objeto para ponerlo en la escena.

- Se podrá seleccionar texturizar cualquier modelo puesto en escena.

- Los objetos puestos en escena podrán editarse, es decir, rotar, dimensionar o trasladar así como poderlos duplicar.

- Desarrollar una biblioteca para la comunicación con el Wiimote y la explotación de sus sensores.

- Implementar una forma de navegación sencilla de interacción explotando las capacidades del Wiimote y su distintas formas de usarlo.

- Construir una barra de leds infrarrojos emisores que el Wiimote usa para la calibración de cámara infrarroja monocromática.

Con estas funciones implementadas se espera tener un software que solucione o amortigüe los problemas planteados en la creación y navegación de los mundos virtuales.

1.4. Organización de la tesis

En el capítulo 2 se explica el tipo de ficheros que se usan para describir los modelos y sus componentes, el porqué del uso e implementación de una base de datos para el almacenamiento y recuperación de modelos y texturas así como su uso dentro del prototipo.

Dentro el capítulo 3 se ofrece un resumen sobre las capacidades que tiene el Wiimote, como es llevada a cabo la comunicación con el dispositivo, así como se muestra el mapeado de las funciones de navegación y construcción del mundo virtual en los botones y gestos del mando.

A lo largo del capítulo 4 se mostrará las capacidades de la interfaz gráfica de usuario, el proceso de construcción de un mundo virtual, la interacción del usuario mediante el Wiimote mientras se navega en un mundo virtual. Terminando el capítulo con la forma de texturizar objetos, transformarlos y duplicarlos.

El capítulo 5 explica la técnica empleada para lograr una optimización de la visualización del mundo virtual de tal forma que únicamente se realice el procesamiento de los objetos que se muestren en pantalla y omita aquellos que no estén presentes.

Por último, en el capítulo 6 están presentes las conclusiones del proyecto así como el posible trabajo a futuro y mejoras que se le pueden realizar al software.

Capítulo 2

Repositorio de modelos y texturas

Como ya se mencionó anteriormente, el objetivo de los mundos virtuales es imitar en algunos aspectos al mundo real; en primera instancia tenemos los objetos que nos rodean, estos tienen un volumen determinado, una posición única en el espacio y colores que los caracterizan. Estos objetos objetos son imitados dentro de un mundo virtual por medio de los **modelos**, de esta forma se puede representar la forma de los objetos. Los colores que tienen los objetos son imitados por medio de las **texturas**, estas son aplicadas a los modelos dando una sensación de mayor realismo al observador.

En el presente capítulo se explica que es un modelo, cómo está conformado lógicamente y su forma de almacenamiento en un repositorio, al igual que se explica que es una textura y cómo es que se aplica ésta en un modelo.

2.1. Formato .obj de Wavefront

En el presente existen diversos formatos para la descripción de modelos tridimensionales dentro de una computadora. Por mencionar algunos de los existentes, .3ds es un formato propietario creado en 1990 y ocupado por la paquetería 3D Studio Max de la compañía Autodesk Media; dos de sus características más importantes es que puede almacenar animaciones y cursivas dentro del mismo archivo. Otro formato que está ganando terreno, debido a que es usado por un editor de contenidos 3D de código abierto y multiplataforma, es .blend usado por Blender 3D; creado en 1995, este formato no sólo almacena la descripción de modelos 3D sino también luces, materiales y posición de la cámara, entre otras características [13].

Así como los antes mencionados, existe otro puñado de formatos con distintas características; algunos han sido estándar para la industria, e.g., .dae creado por Sony Entertainment en el 2004, .bvh creado por Biovision en el 2008 y usado para la captura de movimiento, el formato .u3d creado por 3D Industry Forum en el 2005 o .obj creado por Wavefront Technologies en 1980 para su paquete de animación Advanced

Visualizer. Este último formato llegó a ser un estándar para la industria por ser abierto y por la sencillez que representa los vértices, caras, normales y texturas UV; ha sido adoptado por otros proveedores de aplicaciones de gráficos 3D ya que cualquiera puede realizar un interprete para este formato, debido a su amplia documentación disponible.

Debido a que es el formato mayormente extendido, todos los editores de contenidos 3D cuentan con un exportador o importador, a su amplía documentación, y sobre todo, por ser un estándar y su facilidad de uso, se decidió usar el formato .obj de Wavefront para la realización de este proyecto. A continuación, se dan algunos detalles de dicho formato.

2.1.1. Descripción del formato .obj

Es un formato para describir y almacenar polígonos mediante caracteres ASCII; dentro de él se pueden describir vértices, caras, normales de vértices, texturas de vértices, líneas, curvas, así como crear grupos de los elementos anteriores. Sin embargo, lo respectivo a este proyecto usa únicamente una porción de estos elementos.

El formato .obj está basado en identificadores o etiquetas; existe un identificador para cada elemento que se quiere describir y debe de haber un único identificador por línea, seguido por las propiedades que especifican los valores de ésta. El número de propiedades depende del identificador en cuestión.

Los identificadores que se usaron en este proyecto se aprecian en la Tabla 2.1 junto con su significado y el uso que se le da a cada una.

Identificador	Significado	Uso
o	Objeto	Para crear un grupo de elementos bajo un nombre.
v	Vértice	Para especificar un vértice.
vt	Vértice de Textura	Para la textura de un vértice.
vn	Normal al vértice	Para especificar la normal de un vértice.
f	Cara	Para especificar una cara.

Tabla 2.1: Identificadores de .obj usadas en el proyecto.

Para comenzar a explicar cómo está organizado un archivo .obj y cómo se usan los identificadores, será de ayuda el listado 2.1 en el cual se especifica un cubo conformado por doce triángulos.

```
# Esto es un cubo hecho de 12 triangulos.
o Cubo

v -0.500000 -0.500000 0.500000    # Vertice 1
v -0.500000 -0.500000 -0.500000   # Vertice 2
v 0.500000 -0.500000 -0.500000    # Vertice 3
v 0.500000 -0.500000 0.500000     # Vertice 4
v -0.500000 0.500000 0.500000     # Vertice 5
v -0.500000 0.500000 -0.500000    # Vertice 6
v 0.500000 0.500000 -0.500000     # Vertice 7
v 0.500000 0.500000 0.500000      # Vertice 8

vt 0.250000 0.500000  # Vertice de textura 1
vt 0.000000 0.500000  # Vertice de textura 2
vt 0.000000 0.250000  # Vertice de textura 3
vt 0.500000 0.500000  # Vertice de textura 4
vt 0.500000 0.750000  # Vertice de textura 5
vt 0.250000 0.750000  # Vertice de textura 6
vt 0.500000 0.250000  # Vertice de textura 7
vt 0.750000 0.250000  # Vertice de textura 8
vt 0.750000 0.500000  # Vertice de textura 9
vt 0.250000 0.250000  # Vertice de textura 10
vt 0.250000 0.000000  # Vertice de textura 11
vt 0.500000 0.000000  # Vertice de textura 12
vt 1.000000 0.500000  # Vertice de textura 13
vt 1.000000 0.250000  # Vertice de textura 14

vn -1.000000 0.000000 0.000000    # Normal 1
vn 0.000000 0.000000 -1.000000    # Normal 3
vn 1.000000 0.000000 0.000000     # Normal 4
vn 0.000000 0.000000 1.000000     # Normal 5
vn 0.000000 -1.000000 0.000000    # Normal 6
vn 0.000000 1.000000 0.000000     # Normal 7

f 6/1/1 2/2/1 1/3/1      # Cara 1
f 7/4/2 3/5/2 2/6/2      # Cara 2
f 8/7/3 4/8/3 3/9/3      # Cara 3
f 5/10/4 1/11/4 4/12/4   # Cara 4
f 2/13/5 3/9/5 4/8/5     # Cara 5
f 7/4/6 6/1/6 5/10/6     # Cara 6
f 5/10/1 6/1/1 1/3/1     # Cara 7
f 6/1/2 7/4/2 2/6/2      # Cara 8
f 7/4/3 8/7/3 3/9/3      # Cara 9
f 8/7/4 5/10/4 4/12/4    # Cara 10
f 1/14/5 2/13/5 4/8/5    # Cara 11
f 8/7/6 7/4/6 5/10/6     # Cara 12
```

Listado 2.1: Un cubo de doce triángulos con sus coordenadas de textura en el formato .obj.

En la primera línea se encuentra generalmente un comentario señalando algo importante o alguna nota para el lector, aunque esto es opcional.

En la línea 2 está el primer identificador 'o', el cual indica que todo lo que está a continuación forma parte del objeto llamado 'Cubo'. El nombre no puede contener espacios y debe estar únicamente conformado por caracteres ASCII.

De la línea 4 a la 11 se encuentran los 8 vértices que conforman al cubo en el espacio. El identificador 'v' es seguido por las tres coordenadas, *x, y, z* donde está posicionado el vértice en el espacio. El orden en que se especifican los vértices en esta lista es arbitrario.

Desde la línea 13 a la 26 se encuentra la lista de los vértices de textura, usando el identificador 'vt'; esta lista no guarda relación alguna con la lista anterior y de igual forma el orden en que se escriben los vertices es arbitrario. Este identificador es opcional al momento de describir algún modelo y usa dos valores *u, v* que indican las coordenadas sobre un plano. Su aplicación será detallada en la sección 2.1.3.

De las líneas 28 hasta la 33, se especifica los valores de las normales que usarán los vértices, con el identificador 'vn'; esta lista no tiene relación alguna con las listas anteriores. Este identificador ocupa tres valores *x, y, z* que determinan la dirección en que apunta la normal.

Finalmente, en la línea 35 hasta la 46 se encuentra la especificación de las caras; esta lista hace uso de las tres listas anteriores. El identificador 'f' tiene el siguiente formato, `f v/vt/vn ...`, donde `v`, `vt` y `vn` se reemplazan por un **índice** de vértice, de vértice de textura y de una normal respectivamente; esta terna representa uno de los vértice que conforman a la cara. Aunque se pueden especificar caras de un solo vértice, únicamente tienen sentido aquellas que están conformadas por tres o más vértices.

Tomando la primera cara como ejemplo, `f 6/1/1 2/2/1 1/3/1`, la primer terna, `6/1/1`, indica que se usa las coordenadas del vértice 6, el vértice de textura 1 y la normal 1, i.e., el vértice tiene las coordenadas espaciales -0.5, 0.5, -0.5, las coordenadas de textura 0.25, 0.5, y la normal -1.0, 0.0, 0.0. La segunda y tercera terna se interpretan de la misma forma para dar como resultado un triángulo posicionado en el espacio.

Debido a que un lector de archivos .obj debe analizar y cargar en memoria todos los valores de las listas antes de poder mostrar el modelo, se puede deducir que el orden de estas no tiene que ser rigurosamente el presentado aquí, sin embargo es recomendable seguir esta secuencia ya que es la más lógica.

El resultado de este archivo .obj se muestra en la figura 2.1 donde se aprecian siete de los ocho vértices que conforman al cubo y seis de las doce caras.

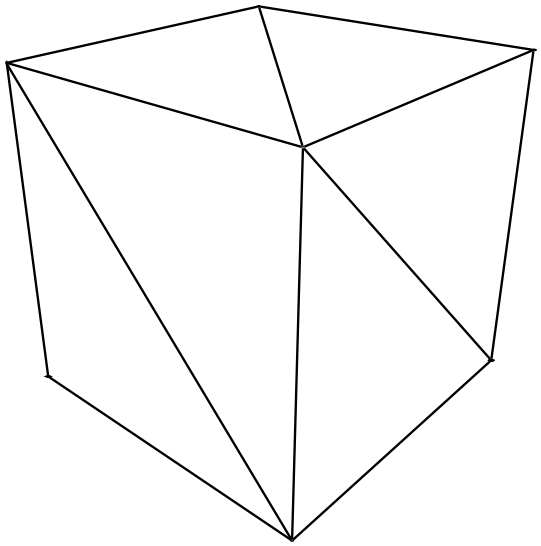


Figura 2.1: El cubo mostandro seis de sus doce triángulos.

Esta lista de vértices, normales, vértices de texturas y caras son cargadas en memoria por un analizador de archivos .obj para que los datos puedan ser usados por OpenGL. Este proceso y la métodología seguida se plasman en el Capítulo 4.

2.1.2. Texturas

El mapeado de texturas o simplemente el **texturizado** es el proceso de **aplicar** una imagen sobre uno o más polígonos. De esta forma, las caras de un modelo tendrán un color imitando la vista que tienen los objetos del mundo real. Las imágenes usadas en este proceso son llamadas **imágenes de textura** o simplemente **texturas**.

Las texturas deben cumplir con ciertas características para que puedan ser interpretadas y cargadas en memoria para posteriormente ser usadas en OpenGL. Es requisito para OpenGL que todas las texturas tengan dimensiones de múltiplos de 2, i.e., 2, 4, 8, 16, 32, ... Estas medidas son en píxeles y el límite de las dimensiones está definido por el hardware de la computadora, específicamente por la tarjeta gráfica y la RAM; actualmente es común ver tarjetas gráficas que soporten texturas de hasta 4096 píxeles por lado, aunque una textura de 256 píxeles por lado, o menor, es recomendable para objetos pequeños con pocos detalles, una textura de 512 píxeles para objetos medianos y reservar las texturas de 1024 píxeles o mayores para objetos grandes con muchos detalles.

El formato para las texturas puede ser cualquiera, png, jpeg, bmp, gif, etc, pero es responsabilidad del programador crear un intermediario entre estos formatos y OpenGL, ya que éste únicamente interpreta arreglos de bytes. El procedimiento

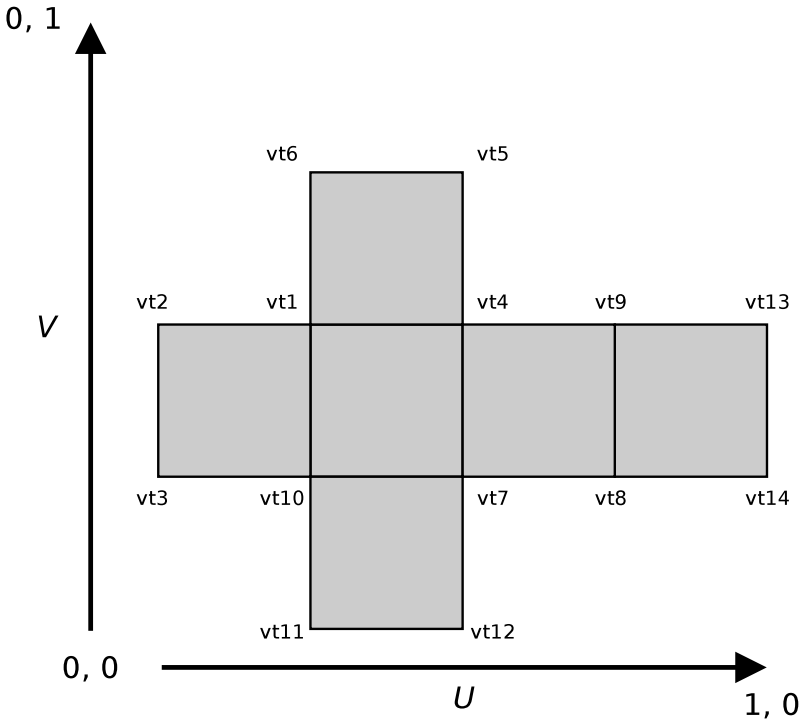


Figura 2.2: Posicionamiento de los vértices de textura en el plano.

seguido se detalla en el Capítulo 4, en la página 39. Aquí se explicará qué relación tiene el identificador 'vt' del formato .obj con las texturas.

2.1.3. Vértices de textura

Retomando de nueva cuenta el listado 2.1, en la página 12, de la línea 13 a la 26 donde se encuentra la lista de vértices de textura, el primero de ellos indica las coordenadas `u, v`: 0.25 0.5; estas coordenadas representan una posición dentro de un plano, específicamente, dentro del plano de la imagen. Para OpenGL todas las texturas, sin importar sus dimensiones reales, miden una unidad de ancho y una unidad de alto teniendo su origen en la esquina inferior izquierda de la imagen.

En la figura 2.2 se muestra un esquema de la textura para el cubo presentado en la sección anterior. Para ejemplificar se señalaron los vértices de textura en la imagen según sus índices del archivo .obj, así como el origen de las coordenadas `u, v`.

Los índices de los vértices de textura almacenados en el archivo .obj servirán para indicar a OpenGL que píxel de la imagen corresponde a qué vértice del modelo, mediante la relación existente en el identificador 'f'. Para tener una idea más clara de esto tomaremos la textura presentada en la figura 2.3, del lado izquierdo, la cual tiene las caras de diferentes colores y en distinto tono a lo largo de la cara.

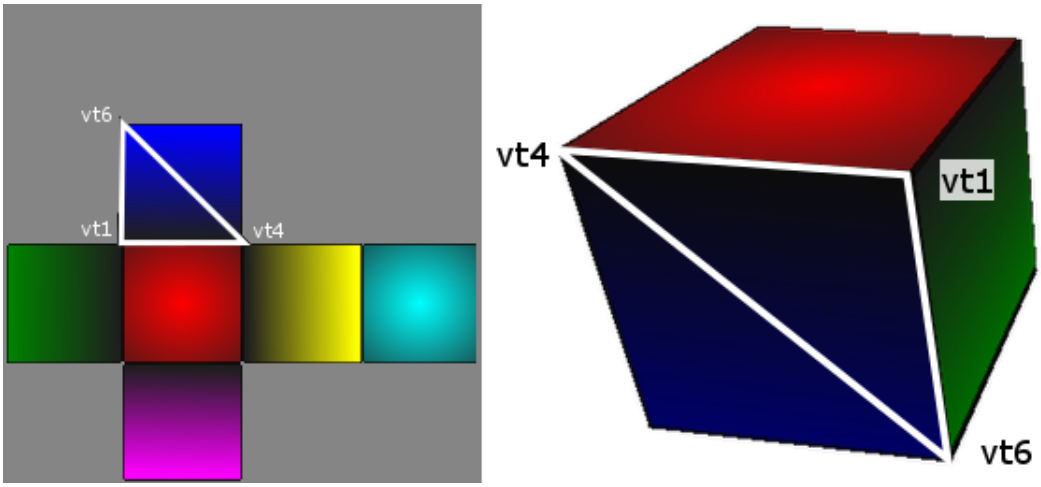


Figura 2.3: El triángulo formado por los vértices de textura vt1, vt4 y vt6 es mapeado en una de las caras del cubo.

En la línea 42 de listado 2.1 se encuentra la cara 8 del cubo, la cual usa los índices de vértices de textura `vt1`, `vt4` y `vt6`; si observamos la textura de la figura 2.3 notamos que contiene en su mayoría la parte más oscura del color azul. Ese segmento de color azul será mapeado en uno de los triángulos que conforman el cubo. Esto se observa más claro en la figura 2.3 del lado derecho.

Las texturas usadas en este proyecto son en formato bmp de 24 bits, i.e., texturas sin comprensión, con tres canales RGB y sin canal alfa.

2.2. Repositorio de modelos y texturas

El proceso de crear un mundo virtual o realizar el prototipo de alguno puede tomar varias horas hombre pero existen procesos en la rutina de trabajo de los de los cuales se puede sacar ventaja, como es la creación y distribución de los modelos y texturas hacia el resto del equipo de trabajo.

En el trabajo presente se propuso e implementó un repositorio de modelos y texturas donde se concentren todos aquellos creados por los artistas y ocupados por los mundos virtuales ya editados.

De esta forma no es necesario pasar las texturas o modelos por métodos convencionales como memorias USB o discos a todos los individuos inmiscuidos en el proceso creativo, si es un número considerable de personas el proceso de copiado entre todas ellas es lento, tedioso y con cabida a la corrupción de datos. Dependiendo de la can-

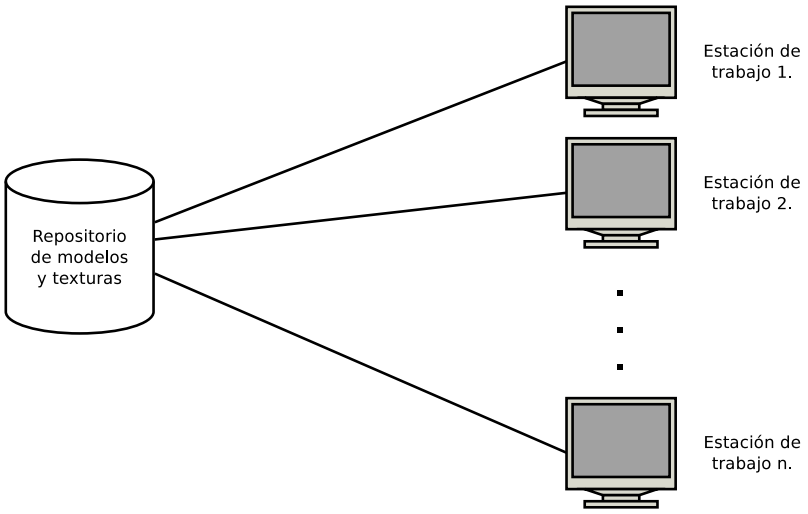


Figura 2.4: Repositorio de texturas y modelos brindandole los recursos a varias estaciones de trabajo.

tidad de texturas y modelos que necesita un mundo virtual pueden ser únicamente unos cuantos cientos de megabytes o podrían ser varios gigabytes de información y duplicar esta cantidad de información en cada estación de trabajo puede no ser lo más óptimo.

Ahora se cuenta con un repositorio de modelos y texturas centralizado donde cada ocasión que un individuo agregue una textura o un modelo al repositorio todos contarán con este recursos de manera inmediata, además de tener la ventaja que los miembros del equipo que no se encuentren en el mismo espacio físico también podrán contar dichas actualizaciones.

En la figura 2.4 se puede observar el concepto de como un mismo repositorio de modelos y texturas puede brindarle los recursos a varias estaciones de trabajo al mismo tiempo.

2.2.1. Conexión de PostgreSQL con C

Para este propósito se creó una base de datos con dos tablas, una para las texturas y otra para los modelos, conservado cierta información acerca del recurso en cuestión. Que si bien ambas tablas son prácticamente iguales en cuanto a sus atributos, no son iguales ni lógicamente ni en su forma de uso, es por eso que existen como dos entidades distintas. Las tablas con sus atributos están presentes en la figura 2.5. El atributo *id* es un número único con el cual identificar a los modelos o texturas, según sea el caso. El nombre es una cadena de texto la cual el usuario pueda identificar al recurso, puede ser repetido. El *formato* indica que tipo de modelo se trata, i.e., obj, 3ds, etc., o

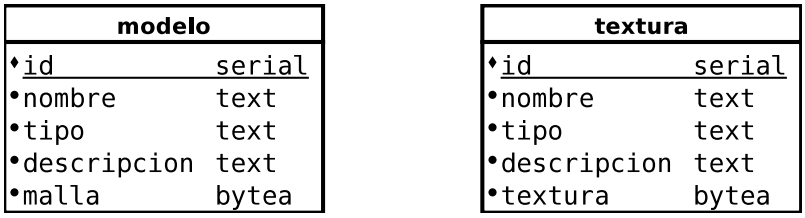


Figura 2.5: Las dos entidades en la base de datos.

en el caso de las texturas es puede ser png, jpg, etc., esto para futuras actualizaciones o parches del software que soporten más formatos. El *tipo* indica la clase de modelo o textura, por ejemplo, árbol, edificio, mueble, piso, etc. La *descripción* es un breve comentario sobre la malla o textura, cualquier información que al usuario le parezca útil. La *malla* en el caso del modelo o el atributo *imagen* en el caso de la textura es un atributo del tipo byte para almacenar el contenido propiamente del modelo o la textura formalmente.

En la implementación se ocupó PostgreSQL [14] como gestor de base de datos, éste cuenta con una colección de bibliotecas de desarrollador para el lenguaje C que proporcionan la capacidad de conectarse y desconectarse a una base de datos, realizar consultas y recibir los resultados de dichas consultas. Para esto cualquier programa cliente que pretenda realizar consultas deberá incluir la cabecera `libpq-fe.h` y ligar el programa con la biblioteca libpq.

Ocupando esta biblioteca fue creada un intermediario entre la interfaz gráfica de usuario y el repositorio de modelos con las operaciones básicas, i.e., abrir y cerrar una conexión con la base de datos, insertar, pedir y borrar un modelo del repositorio, e insertar, pedir y borrar una textura del repositorio.

El abrir la conexión con la base de datos se requiere las credenciales de acceso validas, un usuario, puerto y contraseña que estén dadas de altas en PostgreSQL para poder pasarlas como argumento a la función `PQsetdbLogin` cuyo prototipo se encuentra en el apéndice A, en la página 77, esta función siempre regresa una instancia de objeto `PGconn` sin importar si se llevó a cabo la conexión con la base de datos o no, dado que no es garantía que se realice el enlace con la base de datos por distintas causas, e.g., la base de datos a alcanzado su máximo número de conexiones, la memoria reservada para la aplicación se ha agotado y no se pudo crear el objeto `PGconn` o las credenciales de acceso son erróneas, siempre es necesario verificar si la conexión se llevó a cabo, para esta tarea existe la función `PQstatus` que regresa el estado de la conexión, puede ser comparada con las macros `CONNECTION_OK` o `CONNECTION_BAD`.

En caso de llevarse a cabo el enlace exitosamente, el objeto `PGconn` mantendrá en memoria el enlace creado con la base de datos y será el puente por el cual se realizan las consultas. En caso de que el enlace no sea exitoso se puede obtener información

de depuración con la función `PQerrorMessage`.

El método `abreConexión` de la clase `SQLTrans` es la implementación de este paso, se intenta abrir una conexión con la base de datos, en caso de fallar da aviso del error producido. En el listado 2.2 se dispone el código del método `abreConexion` donde en la línea 9 está la función `PQsetdbLogin` con las credenciales que fueron llenadas al momento de llamar al constructor de la clase, posteriormente se verifica el estado de la conexión reportando si existe algún problema.

```
1  /**
2   * @brief Trata de abrir la conexion con el repositorio de modelos y
3   * texturas usando las credenciales especificadas, en caso de no tener
4   * exito da un aviso.
5   **/
6  void SQLTrans::abreConexion()
7  {
8    /* Creamos la conexion a la base de datos con las credenciales. */
9    conexion = PQsetdbLogin( anfitrion, puerto, NULL, NULL, repositorio,
         usuario, contrasena );
10
11   /* Verificamos que se haya logrado la conexion. */
12   if ( PQstatus(conexion) != CONNECTION_OK )
13   {
14     printf("SQLTrans: No se logro la conexion al repositorio de modelos:
           %s\n", PQerrorMessage(conexion) );
15     PQfinish(conexion);
16   }
17   else printf("SQLTrans: Conexion exitosa.\n");
18 }
```

Listado 2.2: Código del método `abreConexion` de la clase `SQLTrans` donde se realiza el enlace a la base de datos

2.2.2. Almacenamiento y recuperación de los datos

Para ingresar registros en la base de datos, ya sea un modelo o una textura, el procedimiento es similar sin embargo al momento de la recuperación e interpretación de los datos siguen caminos distintos ya que es necesario tratar los datos de manera partícular.

Un prerrequisito para almacenar o recuperar un modelo o textura desde el repositorio es que se haya realizado una conexión de forma exitosa ya que el objeto `PGconn` es el puente entre la aplicación y la base de datos. Otro de los requisitos previos es tener en memoria (en uno o varios buffers) todos los datos que se desean ingresar ya que los comandos de inserción no pueden leer secuencialmente los datos desde un archivo alojado en el disco duro.

La función proporcionada por la biblioteca `libpq` para mandar comandos de cualquier índole a la base de datos es `PQexecParams`, esta función entrega el comando y espera por el resultado de la consulta, tiene la capacidad de poder recibir parámetros dando la flexibilidad de crear distintas consultas en una misma línea de código, la función regresa un objeto del tipo `PGresult` que contendrá, en caso de existir, el resultado debido a las consulta realizada. Debido a posibles fallos de red no es garantía que el resultado de la consulta regrese de forma correcta, por lo que es necesario realizar un chequeo de integridad, para esto se ocupó la función `PQresultStatus` que regresa el estado de la consulta. Existen varios estados que podría regresar la consulta aunque en este trabajo sólo se ocuparon dos:

- `PGRES_COMMAND_OK` Significa la conclusión con éxito de la consulta y no se regresó ningún tipo de dato (usado en el caso de los `INSERT`).

- `PGRES_TUPLES_OK` Es la conclusión con éxito de la consulta y se regresó algún tipo de dato (usado en el caso de los `SELECT`).

Finalmente, se debe de limpiar el resultado de la consulta y liberar el espacio en memoria que fue solicitado para almacenar el resultado, de lo contrario se presentarán fugas de memoria, para esto se usa la función `PQclear`. Es responsabilidad del programador tener un, y sólo un llamado a la función `PQclear` por cada consulta realizada a la base de datos para evitar estas fugas de memoria, si se hacen dos llamados a `PQclear` consecutivos o sin que se haya creado un objeto `PGresult` se produciran errores en la aplicación.

Ejemplificando los párrafos anteriores se muestra el listado 2.3 cuyo código pertenece al método `insertaModelo`, omitiendo algunos chequeos de errores con el fin de ahorrar espacio.

```
1  void SQLTrans::insertaModelo( char *nombre, char *formato, char *tipo,
       char *ruta, char *descripcion )
2  {
3    struct stat datos;
4    char *buffer;
5    FILE *fp;
6    int cantidadDeValores = 5;
7    const char *param_vals[cantidadDeValores];
8    int param_lens[cantidadDeValores];
9    int param_fmts[cantidadDeValores];
10
11   // Obtenemos los datos del archivo a insertar.
12   stat(ruta, &datos);
13
```

```
14    // Creamos una zona donde guardaremos los bytes de manera provisional.
15    buffer = (char *) malloc ( datos.st_size );
16
17    // Abrimos el archivo donde describe el modelo para guardarlo de forma
18    // provisional en el buffer.
19    fp = fopen( ruta, "r" );
20    fread( buffer, sizeof(char), datos.st_size, fp );
21    fclose( fp );
22
23    // Llenamos el arreglo con los datos que vamos a insertar.
24    param_vals[0] = nombre;
25    param_vals[1] = formato;
26    param_vals[2] = tipo;
27    param_vals[3] = descripcion;
28    param_vals[4] = buffer;
29
30    // Ahora llenamos el arreglo con la longitud de cada valor.
31    param_lens[0] = strlen(nombre);
32    param_lens[1] = strlen(formato);
33    param_lens[2] = strlen(tipo);
34    param_lens[3] = strlen(descripcion);
35    param_lens[4] = datos.st_size;
36
37    // Aqui se especifica el formato de cada uno, si es binario o no.
38    param_fmts[0] = 0;
39    param_fmts[1] = 0;
40    param_fmts[2] = 0;
41    param_fmts[3] = 0;
42    param_fmts[4] = 1;
43
44    // Finalmente insertamos el buffer dentro del repositorio de modelos.
45    resultado = PQexecParams(conexion, "INSERT INTO modelo ( nombre,
          formato, tipo, descripcion, malla ) VALUES ($1,$2,$3,$4,$5)",
          cantidadDeValores, NULL, param_vals, (const int *) param_lens, (
          const int *) param_fmts, 1);
46
47    if (PQresultStatus(resultado) != PGRES_COMMAND_OK) {
48      fprintf( stderr, "SQLTrans: Fallo INSERT: %s\n", PQerrorMessage(
            conexion) );
49      exit(-1);
50    }
51
52    printf("SQLTrans: Modelo insertado con exito.\n");
53    PQclear(resultado);
54    free(buffer);
55  }
```

Listado 2.3: El listado resumido del método `insertaModelo`

El método del listado 2.3 recibe como parámetros punteros a cadenas de texto que contienen la información respectiva del modelo, su nombre, formato, tipo, descripción y la ruta donde está almacenado el archivo .obj. Estos datos serán usados para llenar

los arreglos que pide `PQexecParams` como parámetros al momento de su invocación. Estos arreglos son los que se declaran en las líneas 7, 8 y 9. Es necesario obtener otra información necesaria antes de llenar los arreglos, esta información es obtenida desde la línea 11 a la línea 21. Primero se obtiene el mediante la función `stat` los datos del archivo .obj aunque lo que es de interés es su tamaño en bytes, ocupando este dato se reserva un bloque de memoria donde pueda ser alojado, por último es abierto el archivo .obj y volcado en dicho bloque de memoria.

Una vez que ya está el buffer listo llenamos los arreglos desde la línea 23 a la 42, este paso únicamente consiste en asignaciones, `param_vals[]` contiene los valores a insertar en el registro de la base de datos, `param_lens[]` contiene la longitud en bytes de esta información mientras que `para_fmts[]` contiene el formato de los valores, si es un cero será en modo texto, si es uno será en formato binario.

La línea más trascendental es la que contiene el envío y ejecución del comando SQL, la línea 45. Aquí la función `PQexecParams` usa el objeto `PGconn`, una cadena de texto que contiene el comando SQL suplantando los cinco valores que se van a ingresar (nombre, formato, tipo, descripción y el binario del modelo) por `$1, $2, ..., $5`, estos símbolos serán reemplazados por los valores del arreglo `param_vals` teniendo una relación el número del símbolo con una localidad del arreglo, el símbolo `$1` corresponde al elemento `param_vals[0]`, `$2` corresponde al elemento param_vals[1], etc.

Ahora es necesario realizar los dos últimos pasos que se discutieron más arriba, la verificación que se haya llevado a cabo la inserción con éxito con la función `PQresultStatus` y la liberación de la memoria con `PQclear`.

En cuanto a la recuperación de los datos el proceso es bastante similar con la diferencia de que ahora existirá un valor de retorno y éste será el objeto `PGresult`, el cual contiene las tuplas pedidas en el comando ejecutado, cada objeto que invoque al método `pideModelo` o `pideTextura` de la clase `SQLTrans` le será un puntero al objeto `PGresult` siendo responsable de tratar los datos respectivos y de limpiar la memoria.

En el listado 2.4 se muestra el método `pideModelo`, tiene como parámetro un puntero de caracteres que contendrá el id de la textura que se desea extraer del repositorio, de igual forma que el método `insertaTextura` se asignan los datos de la cadena a los buffers `param_vals[]` (el valor que será enviado), `param_lens[]` (la longitud del valor que será enviado) y `para_fmts[]` (el formato del valor que será enviado).

Posteriormente está la función `PQexecParams` que mandará el comando reemplazando el símbolo `$1` por la cadena contenida en `param_vals[0]`. Este método regresa el puntero al objeto `PGresult` creado en la consulta y podra ser tratado como lo desee el método que lo haya invocado.

```
1  PGresult * SQLTrans::pideModelo( char *id )
2  {
3
4    int cantidadDeValores = 1;
5    const char *param_vals[cantidadDeValores];
6    int param_lens[cantidadDeValores];
7    int param_fmts[cantidadDeValores];
8
9    // Llenamos el arreglo con los datos que vamos a insertar.
10   param_vals[0] = id;
11
12   // Ahora llenamos el arreglo con la longitud de cada uno de los
         valores.
13   param_lens[0] = strlen(id);
14
15   // Es turno de especificar si cada uno de ellos es binario o no.
16   param_fmts[0] = 0;
17
18   // Finalmente realizamos la consulta a la base de datos obteniendo los
         valores del modelo especificado.
19   resultado = PQexecParams(conexion, "SELECT * FROM modelo where id=$1",
         cantidadDeValores, NULL, param_vals, (const int *) param_lens, (
        const int *) param_fmts, 1);
20
21   if ( PQresultStatus(resultado) != PGRES_TUPLES_OK )
22   {
23     fprintf( stderr, "Error: PQexecParams: %s\n", PQerrorMessage(
          conexion));
24     PQclear( resultado );
25     return NULL;
26   }
27     if( PQntuples(resultado) == 0 )
28     {
29         printf("SQLTrans: No se han conseguido resultados de la consulta
              , el registro no existe.\n");
30         printf( "No. Filas: %d\n", PQntuples(resultado) );
31         printf( "No. campos: %d\n", PQnfields(resultado) );
32     PQclear( resultado );
33     return NULL;
34     }
35
36   return resultado;
37 }
```

Listado 2.4: El listado resumido del método `pideModelo`

El proceso de interpretar los datos correspondientes a una textura o un modelo recuperados desde el repositorio es tarea pertinente a la interfaz gráfica que es la encargada de pedir los datos, se detallará más al respecto en el capítulo 4, en la página 39, el trabajo de la clase SQLTrans es la de ser un intermediario entre dicha

interfaz y el repositorio de modelos y texturas.

Capítulo 3

Comunicación con el wiimote

El dispositivo Wiimote convencional es un mando para la consola de videojuegos Wii creada por Nintendo. Es un dispositivo inalámbrico que se comunica mediante tecnología estándar bluetooth con el anfitrión y toda la documentación disponible sobre sus registros y comandos están disponibles debido a ingeniería inversa, ya que Nintendo no ofrece algún tipo de información sobre el uso técnico y programación del mando.

El Wiimote tiene un diseño poco parecido a un control convencional de videojuegos, es más parecido a un control remoto de una televisión, esto se debe a que los diseñadores se dieron cuenta que mucha gente no tomaría un control de videojuegos únicamente por su forma nata pero la mayoría de las personas no tienen algún problema al tomar un control remoto de televisión y usarlo. La finalidad fue sobrepasar la frontera que significa ese paradigma y tratar de llegar a un público más adulto.

Este capítulo detallará de forma técnica el Wiimote, sus capacidades y la forma de comunicación con la computadora anfitrión.

3.1. Capacidades del wiimote

El Wiimote es un control diseñado para interactuar con videojuegos bidimensionales y tridimensionales mediante sus métodos de entrada y capacidades de retroalimentación. Su método de entrada más relevante son los doce botones, once de ellos colocados en la parte frontal y el último en la parte trasera; en la figura 3.1 se muestran los botones del control. El botón **POWER** está destinado a apagar y encender el control, el resto de ellos tendrá la acción que se desee, aunque algunos botones sugieren una acción en especial, ya sea por su forma, su nombre o por su posición en el control.

Los botones **arriba**, **abajo**, **izquierda** y **derecha** puestos en la **cruceta de dirección** son recomendables para la orientación o acciones que tengan algún sentido

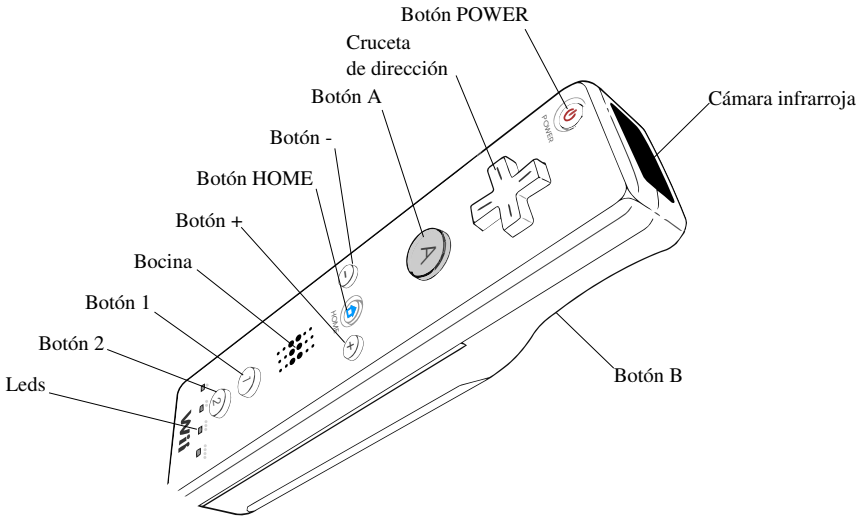
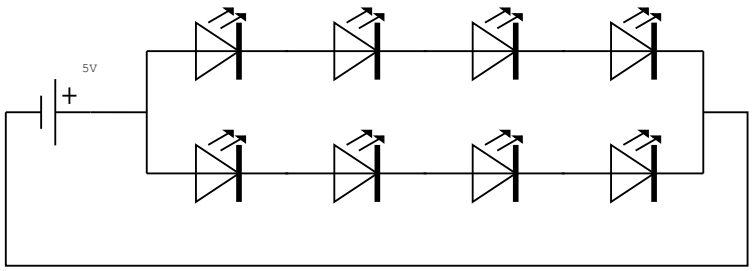


Figura 3.1: Un esquema del Wiimote donde se señalan sus botones, la cámara infrarroja, los leds y su bocina.

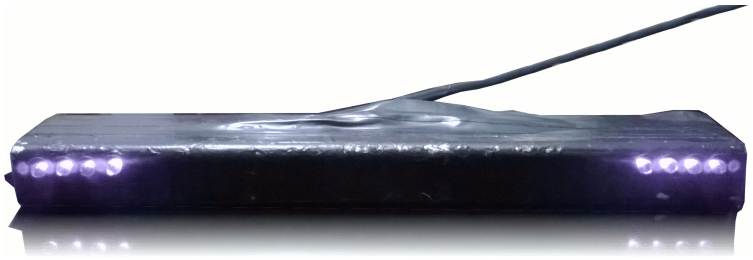
espacial, el botón **A** puesto justo debajo de la cruceta de dirección, siendo el botón más grande y de fácil alcance para el dedo pulgar, está destinado a la acción de uso más común mientras que el botón **B**, puesto en la parte trasera del control, con forma de gatillo y con un fácil alcance por el dedo índice es recomendable para la segunda acción de más uso.

Los botones $+$, $-$, **HOME**, **1** y **2**, se consideran como auxiliares y no son tan fáciles de alcanzar por el dedo pulgar, por eso están destinadas a tareas secundarias. Los botones $+$, $-$ y **HOME** son recomendables para tareas que estén relacionadas con cantidades. Por último, los botones **1** y **2** están destinados para las acciones vinculadas al cambio de opciones. En la figura 3.1 se encuentra la distribución de los botones en el control y otras de sus partes.

Otro de los métodos de entrada es la cámara infrarroja que está en la parte superior del control, dicha cámara es monocromática teniendo un campo de visión de 33 grados horizontalmente y 23 grados verticalmente, contando con una resolución de 1024×768 píxeles. En realidad se trata de una cámara digital convencional pero justo frente de la lente de la cámara se encuentra un filtro óptico que sólo permite pasar longitudes de onda desde los 850 nm a los 950 nm, i.e., sólo permite pasar el espectro infrarrojo cercano. Esta cámara tiene un procesador de imagen incluido que busca por los puntos más luminosos dentro de la imagen reportando su posición, por lo tanto, no es posible acceder a la imagen en bruto, sólo se tiene acceso a ese reporte de posiciones.



(a)



(b)

Figura 3.2: En 3.2(a) se aprecia el diagrama mostrando la conexión de los diodos emisores de luz infrarroja siendo alimentados por los 5 voltios que proporciona el puerto USB. En 3.2(b) se mustra la barra construida y funcionando, el resplandor que se percibe es la luz infrarroja ya que cualquier cámara digital puede captarla.

Los puntos luminosos pueden ser producidos por cualquier fuente de luz infrarroja, sólo deben de estar separados aproximadamente 20 cm, de esta forma el procesador de imagen de la cámara puede reportar la distancia aproximada desde la fuente de luz así como el ángulo de inclinación con respecto a la barra infrarroja, para este fin se construyó una barra que tiene cuatro diodos emisores de luz infrarroja en cada extremo alimentado por los 5 voltios que proporciona el puerto USB, en la figura 3.2 se muestra el diagrama resultante 3.2(a) y la barra construida mientras está funcionando 3.2(b), cualquier cámara digital es capaz de percibir la luz infrarroja.

El Wiinote incluye un acelerómetro, este sensor es capaz de percibir aceleraciones en los tres ejes, `x`, `y` y `z`, éstas son reportadas en fuerza G. La gravedad siempre está influyendo en el acelerómetro dando una lectura en el eje `z` de 1 G cuando el control está completamente horizontal.

El acelerómetro también reporta ángulos que se conocen en matemáticas como ángulos de Tait-Bryan aunque se ha adoptado parte del vocabulario que existe en aeronáutica donde se les conoce en habla inglesa como `pitch`, `yaw` y `roll` a la inclinación de los tres ejes, mientras que en habla española se tradujeron como cabeceo, guiñada y alabeo, respectivamente.

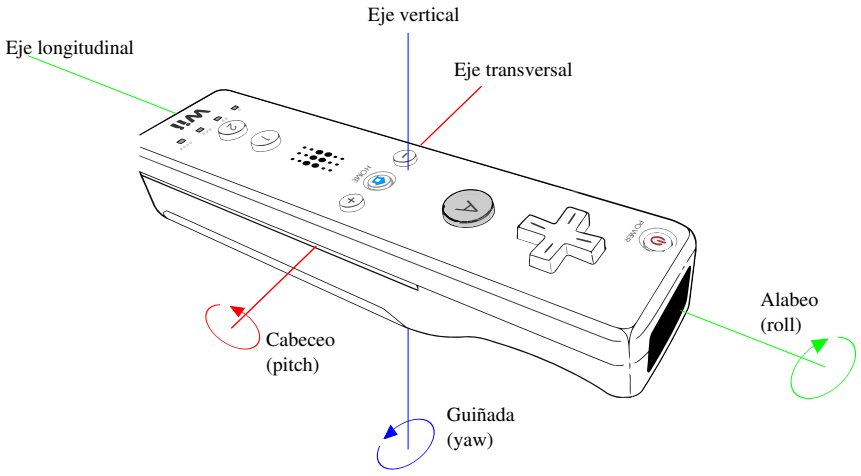


Figura 3.3: Se muestra un Wiimote con sus tres ejes y el nombre del giro correspondiente a cada uno de ellos.

El cabeceo (`pitch`) es la rotación con respecto al eje transversal, la guiñada (`yaw`) es una rotación respecto del eje vertical y el alabeo (`roll`) es una rotación respecto al eje longitudinal, en la figura 3.3 se muestra de forma gráfica lo explicado en este párrafo, se muestra el eje transversal, el longitudinal y el vertical con el nombre que recibe el giro en cada eje.

A partir de la lectura de la gravedad en los ejes se puede determinar la inclinación, e.g. cuando el Wiimote está completamente horizontal sólo existe aceleración en el eje `z` debido a la fuerza de gravedad, si el control es inclinado aunque sea un poco en su eje transversal la fuerza de gravedad afectara al eje `z` y al eje `y`; de esta manera se puede medir la inclinación en dos de sus ejes, sin embargo, al realizar un giro en su eje vertical la gravedad no afecta a ningún eje más que al `z`, por lo tanto, no se puede medir el giro con el acelerómetro, el tercer grado de inclinación es reportado por medio de la inclinación relativa a la barra infrarroja, el procesador de imagen de la cámara es el encargado de proporcionar este ángulo, cuando el control está completamente horizontal viendo directamente a la barra infrarroja se tiene un grado de inclinación cero en los tres ejes.

Esos son todos los métodos de entrada con el que cuenta el Wiimote, el primer método de salida o de retroalimentación que ofrece el mando es el motor incluido que tiene una pequeña masa, cuando el motor gira produce una vibración en el control, este vibrador tiene 255 niveles de intensidad.

Otro método de retroalimentación, aunque no fue usado en este proyecto, es una bocina que es capaz de reproducir sonidos modulados por impulsos codificados (PCM) de 8 bits.

El último método de retroalimentación al usuario son los 4 diodos emisores de luz que se encuentran en la parte inferior del control, estos parpadean mientras el control se encuentra en busca de un anfitrión Bluetooth, al realizar el enlace se pueden mantener encendidos o apagados como lo desee la aplicación que haga uso del control.

Este conjunto de métodos de entrada y salida conforman un mando inalámbrico de seis grados de libertad, barato y de programación relativamente sencilla, como se describirá en las siguientes secciones.

3.2. Biblioteca creada para el Wiimote

Para realizar la sincronización del Wiimote con el sistema operativo se procede igual que cualquier otro dispositivo Bluetooth: abrir el panel Bluetooth de Mac OS para posteriormente presionar el botón de sincronización en el dispositivo, el cual se encuentra en el compartimiento de las baterías. Esto pondrá al Wiimote en modo detectable, el sistema operativo podrá entonces sincronizarse con el Wiimote abriendo los canales de comunicación, sin embargo, se requiere de una aplicación que explote dicho canales de comunicación enviando comandos al control y esperando por la respuesta, o en su defecto, abra unos propios y los use de la misma manera. Por esta razón es necesario una biblioteca que sea capaz enviar los comandos y recibir eventos desde el control, pero todas las bibliotecas que existían al comienzo de este proyecto no eran adecuadas, debido a diversas razones, por mencionar algunas, sólo estaban disponibles para otros sistemas operativos como Windows XP, Linux o Android, no estaba disponible el código fuente u ocupaban funciones y métodos descontinuados de la API de Mac OS, por lo tanto no compilaban o no funcionaban. Para la creación de la biblioteca de comunicación se tomó como base varias ya existentes, estudiando el protocolo de comunicación del Wiimote, la longitud en bytes de los comandos y las respuesta del control.

En esta sección se detallará dicha biblioteca y la forma de comunicación con el control.

3.2.1. Apertura de canales de comunicación

La API de Mac OS proporciona lo necesario para establecer los canales de comunicación por medio de la bibliotea `IOBluetoothDeviceInquiry.h`, como su nombre lo insinúa, tiene métodos para realizar una investigación sobre los dispositivos Bluetooth que se encuentren en modo detectable, algunos métodos deben de ser sobrecargados por el programador para personalizar el comportamiento. En el listado 3.1 se encuentra el método `start` de la biblioteca creada, este método es el encargado de crear la consulta de cuales dispositivos Bluetooth se encuentran en modo detectable, cuanto

tiempo va a durar dicha consulta y el manejo de errores pertinente.

```
1  - (IOReturn) start:(unsigned int) timeout maxWiimotes:(unsigned int)
       wiimotesNum
2  {
3    if ( ![IOBluetoothHostController defaultController] )
4    {
5      printf("No existe algun dispositivo Bluetooth que realice la
           consulta.");
6      return kIOReturnNotAttached;
7    }
8
9    if ( [self isDiscovering] )
10   {
11     printf("Ya existe una consulta en proceso...");
12     return kIOReturnSuccess;
13   }
14
15   [self close];
16   maxWiimotes = wiimotesNum;
17   foundWiimotes = 0;
18
19   inquiry = [IOBluetoothDeviceInquiry inquiryWithDelegate:self];
20
21   if( timeout && timeout < 20 ) [inquiry setInquiryLength:timeout];
22   else [inquiry setInquiryLength:20];
23
24   IOReturn status = [inquiry start];
25   if ( status == kIOReturnSuccess ) [inquiry retain];
26   else
27   {
28     [self close];
29     printf("No es posible buscar dispositivos.");
30   }
31
32   return status;
33 }
```

Listado 3.1: Código del método `start` de la clase `io_mac.h` donde se realiza el enlace con el Wiimote.

Desde la línea 3 a la 15 es el manejo de errores, el primero es para verificar que existe el hardware necesario para realizar las consultas y los enlaces con los dispositivos Bluetooth, el segundo es para comprobar que el hardware está disponible y no está siendo usado por otro proceso realizando otra consulta. En la línea 19 se crea una instancia del objeto `IOBluetoothDeviceInquiry`, en las siguientes líneas, 21 y 22, se establece el tiempo mínimo que debe de durar la consulta en segundos. Finalmente, la consulta se lleva a cabo la consulta en la línea 24 con las condiciones establecidas terminando con el chequeo de errores.

Durante la consulta del objeto `IOBluetoothDeviceInquiry` acontecen ciertos eventos son invocados métodos de la misma biblioteca de forma automática, estos métodos deben de ser sobrecargados para personalizar su comportamiento, el método `deviceInquiryStarted` es invocado cuando la consulta es iniciada, el método es invocado `deviceInquiryDeviceFound` en cada ocasión que la consulta encuentra algún dispositivo en modo detectable pasando como argumento un puntero al dispositivo en cuestión, y el método `deviceInquiryComplete` es llamado cuando la consulta termina. Es en el método `deviceInquiryDeviceFound` donde se obtienen los datos del control comprobando que se trate de un Wiimote, el puntero al dispositivo encontrado contiene el nombre y la dirección MAC y este puntero es necesario para la apertura de los canales de comunicación.

Una vez que ya tenemos un puntero al dispositivo Wiimote almacenado es posible realizar la conexión, tal como se indica en [15] el control exige dos canales de comunicación que siguen el protocolo L2CAP, uno de entrada y uno de salida, el canal de entrada es el número 13 mientras que el canal de salida es el canal 11. La API de Mac OS ofrece la clase `IOBluetoothL2CAPChannel` para manejar estos canales. Mediante el método `openL2CAPChannelWithPSM`, el cual sólo requiere el puntero al dispositivo y el número del canal que se desea abrir, se realiza la apertura de los canales con el control y espera por el apretón de manos o la denegación, la apertura de estos dos canales se ve reflejado en las líneas 14 y 24 del listado 3.2 donde se realiza la solicitud de apertura de los canales y posteriormente el chequeo de errores pertinente.

```
1  - (IOReturn) connectToWiimote:(wiimote*) wm
2  {
3    IOBluetoothDevice* device = [IOBluetoothDevice withDeviceRef:wm->
         device];
4    IOBluetoothL2CAPChannel* outCh = nil;
5    IOBluetoothL2CAPChannel* inCh = nil;
6
7    if( !device )
8    {
9      printf("No existe el dispositivo.");
10     return kIOReturnBadArgument;
11   }
12
13   // El canal de entrada es el numero 11.
14   outCh=[self openL2CAPChannelWithPSM:0x11 device:device delegate:self];
15   if (!outCh)
16   {
17     printf("No se abrio el canal L2CAP de salida (id %i).", wm->unid);
18     [device closeConnection];
19     return kIOReturnNotOpen;
20   }
21   wm->outputCh = [[outCh retain] getL2CAPChannelRef];
22
23   // El canal de entrada es el numero 13.
```

```
24    inCh=[self openL2CAPChannelWithPSM:0x13 device:device delegate:self];
25    if (!inCh)
26    {
27      printf("Unable to open L2CAP input channel (id %i).", wm->unid);
28      [device closeConnection];
29      return kIOReturnNotOpen;
30    }
31    wm->inputCh = [[inCh retain] getL2CAPChannelRef];
32
33    // Guardamos el
34    return kIOReturnSuccess;
35  }
```

Listado 3.2: Código del método `connectToWiimote` de la clase `io_mac.h` donde se abren los canales de comunicación.

Una vez que los canales de comunicación han sido establecidos, se llama a un método delegado automáticamente, concluyendo el apretón de manos entre el Wiimote y el dispositivo Bluetooth de la computadora. Este proceso crea objeto del tipo deviceHandler (controlador de dispositivo) que sirve como interfaz entre el control y el programador, nos podremos referir a él posteriormente para la lectura del estado del control en un momento dado.

3.2.2. Comandos del Wiimote, escritura y lectura de comandos

El nombre de los dos canales de comunicación se asigna desde el punto de vista del anfitrión, i.e. el canal de entrada será un canal donde el anfitrión toma datos desde el control y el canal de salida será un canal donde el anfitrión envíe información al control. Ya que se tienen los canales de comunicación establecidos es necesario la lista de comandos se pueden enviar y que información se puede esperar desde el control, la documentación disponible en [15] nos proporciona dicha lista de comandos con su nombre, su longitud en bytes, código hexadecimal y el saber si es un comando de entrada o de salida, todo ésto último se plasma en la tabla 3.1.

Entrada/Salida	Código	Tamaño	Función
Salida	0x11	1	LEDs
Salida	0x12	2	Modo de reporte de datos
Salida	0x13	1	Activar/Desactivar cámara infrarroja
Salida	0x14	1	Activar/Desactivar bocina
Salida	0x15	1	Solicitud de información de estado
Salida	0x16	21	Escritura de memoria
Salida	0x17	6	Solicitud de lectura de memoria
Salida	0x18	21	Datos de la bocina
Salida	0x19	1	Silenciar bocina
Entrada	0x20	6	Información de estado
Entrada	0x21	21	Envío del contenido de la memoria
Entrada	0x22	4	Acuse de recibo
Entrada	0x30-0x3f	2-21	Reportes de datos

Tabla 3.1: Comandos de entrada y salida del Wiimote.

La tamaño especificado en cada comando son los bytes que serán enviados por el canal de comunicación, además del comando mismo, estos bytes adicionales son los parámetros del comando y son útiles para indicar el comportamiento del comando, e.g. el comando para activar o desactivar la cámara infrarroja (0x13) usa un byte adicional como parámetro, en este caso si el bit 0 está en cero indicará la desactivación, si el bit 2 esta en uno indicará la activación, i.e. si enviamos al control los dos bytes 0x13 0x00 desactivará la cámara infrarroja, en cambio si se enviamos al control 0x13 0x04 se activará.

Ya que la lista de comandos es algo extensa y cada uno tiene una cantidad de parámetros distinto se reportan los comandos y su uso especifico en el Apéndice B, en la página 81.

Es necesario la escritura de los comandos en el Wiimote, la instancia de los canales cuenta con un método para esto, writeSync, el cual recibe como parámetro el buffer a escribir y la longitud del mismo. En el listado 3.3 se muestra el proceso completo de escritura con el manejo de errores necesario aunque la línea significativa es la 35 donde se usa el método writeSync. De esta forma, al mandar comandos al control podemos activar o desactivar la cámara, encender y apagar los diodos emisores de luz o hacer vibrar el control.

```
1 int wiic_io_write(struct wiimote_t* wm, byte* buf, int len)
2 {
3   unsigned int length = (unsigned int)len;
4
5     NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
```

```
6
7   IOBluetoothL2CAPChannel* canalTemp = [IOBluetoothL2CAPChannel
        withL2CAPChannelRef:wm->outputCh];
8
9 IOReturn valRet = [ canalTemp writeSync:buf length:length];
10
11   if ( valRet != kIOReturnSuccess)
12     WIIC_ERROR("Imposible escribir en el canal (id %i).", wm->unid);
13
14     [pool drain];
15
16   return ( valRet == kIOReturnSuccess ? len : 0);
17 }
```

Listado 3.3: Código del método `wiic_io_write` de la clase `io_mac.h` donde se manda el comando al Wiimote.

El saber que botones están presionados, la inclinación del control y el estado de la cámara es cuestión de los reportes que se ofrecen, para obtenerlos de forma continua de parte del Wiimote se ocupa el comando de solicitud de estado (0x12), de igual que manera este comando activa o desactiva el envío de datos dependiendo del parámetro que enviemos, aunque esto sea por medio del canal de entrada abierto anteriormente esto es transparente para el programador y únicamente se debe limitar a procesar los mensajes por medio del controlador de dispositivo extrayendo bytes de forma continua.

Para el fin de extraer la información se escribió un método donde se crea un hilo que extrae los bytes de forma continua y los copia en un buffer dentro de la estructura de datos que contiene los datos del wiimote para su posterior procesado. En el listado 3.4 se muestra el método `wiic_io_read` omitiendo la verificación de errores para ahorrar espacio, las líneas significativas de este método son desde la 15 hasta la 30 donde se crea un puntero que posteriormente apuntará al buffer del controlador de dispositivo, que posteriormente copiará la información en un segmento de memoria dentro de la estructura de datos.

```
1 int wiic_io_read(struct wiimote_t* wm)
2 {
3
4   deviceHandler = (WiiConnect*)(wm->connectionHandler);
5
6     // Comenzamos el ciclo para obtener los datos.
7   [deviceHandler setReading:YES];
8   [deviceHandler setTimeout:NO];
9
10  // Corremos el hilo.
11  [NSThread detachNewThreadSelector:@selector(startTimerThread) toTarget
        :deviceHandler withObject:nil];
12
13  NSRunLoop *theRL = [NSRunLoop currentRunLoop];
```

```
14
15    while([deviceHandler isReading] && ![deviceHandler isTimeout])
16 {
17      NSAutoreleasePool *pool_loop = [[NSAutoreleasePool alloc] init];
18      [theRL runMode:NSDefaultRunLoopMode beforeDate:[NSDate distantFuture
            ]];
19      [pool_loop drain];
20    }
21
22    // Creamos el buffer donde se copiarán los datos.
23    byte* buffer = 0;
24    unsigned int length = 0;
25
26    if(![deviceHandler isDisconnecting])
27 {
28      // Apuntamos al mensaje del manejador de dispositivo.
29      buffer = [deviceHandler getNextMsg];
30      length = [deviceHandler getMsgLength];
31    }
32
33
34    // Lo copiamos a la estructura de datos del wiimote.
35    if( length < sizeof(wm->event_buf) )
36      memcpy(wm->event_buf,buffer,length);
37
38    // Liberamos la memoria.
39    [deviceHandler deleteMsg];
40
41    [pool drain];
42
43      return 1;
44 }
```

Listado 3.4: Código del método `wiic_io_read` de la clase `io_mac.h` donde se copia el buffer en el controlador de dispositivo.

De esta manera ya se podrá averiguar que botones están presionados, la inclinación del control y el reporte del procesador de imagen de la cámara, extrayendo los datos con el comando de entrada 0x33, el cual es el que proporciona dichos datos.

Capítulo 4

La Interfaz Gráfica de Usuario

La interfaz gráfica de un software siempre ha tenido la función de ser un intermediario entre las funciones que proporciona el programa y el usuario, dotándolo de la capacidad de realizar las acciones necesarias para completar la tarea para la cual está hecho el software, esto siempre tratándolo de proporcionar de una forma amigable e intuitiva.

Para mostrar al usuario los mundos virtuales y gestionar los eventos de entrada se implementó un motor de presentación, el cual, tiene distintas etapas que preparan y mandan a la máquina de estados de OpenGL los vértices que serán mostrados en pantalla, siendo lo suficientemente rápido para evitar la percepción de lentitud en el refrescamiento de los gráficos.

Al tratarse de la construcción de mundos virtuales mediante primitivas, en primera instancia se tuvo que identificar las acciones necesarias que un usuario necesitaría para construirlos y establecer la forma en que un usuario interactuaría con el software mediante el Wiimote, resultando en un mapeado de dichas funciones y actividades en el control creando distintos contextos de trabajo, i.e., existe un contexto para navegar, otro para texturizar, otro para trasladar objetos, etc.

Se implementaron y probaron dos tipos de colisiones entre la cámara del mundo virtual y los objetos visualizados, de tal forma que la cámara no pudiese traspasar a través de los objetos, dando una información incorrecta al usuario acerca de su posición, resultando así en una navegación más realista.

La interfaz fue construida con OpenGL para mostrar el mundo virtual y la biblioteca QT fue usada para la creación de la ventana y otras funciones con el sistema operativo. Al termino de esta fase se obtuvo un software con el cual el usuario puede construir mundos virtuales relativamente grandes de forma interactiva.

4.1. El motor de presentación

En inglés se ocupa el termino *render* para referirse al procesado y presentación de modelos 3D en pantalla, aunque se ha extendido el uso y conjugación de este verbo en inglés, un termino adecuado en español podría ser "interpretación" aunque en acorde al contexto en el que se está usando "presentación" también tiene un uso correcto. Esta parte del proyecto es conocida en inglés como *rendering engine* o *graphics engine* sin embargo se usará el termino **motor de presentación** para nombrarlo.

Un motor de presentación es una pieza de software encargada de mantener en memoria, procesar y presentar en pantalla modelos 3D de forma coherente dando al espectador la sensación de fluidez y continuidad en las imágenes observadas, por lo menos, en una tasa de 24 cuadros por segundo. Para presentar una cantidad considerable de polígonos simultáneamente en pantalla siempre se debe de tener en mente la eficiencia. El motor está dividido en varias fases de procesamiento, éstas son cuatro: la fase de **procesamiento de eventos de entrada**, la fase de **actualización de objetos**, la fase de **procesamiento de colisiones** y la fase de **pintado del mundo virtual**.

Estas cuatro fases son continuas, una tras de otra y son accionadas por un temporizador cada 20 milisegundos, la primera fase, el manejo de eventos del Wiimote, realiza un chequeo por los eventos realizados en ese instante de tiempo, i.e., busca que botones están siendo presionados y la posición del puntero de la cámara infrarroja. Dependiendo del contexto (éstos serán revisados en la subsección 4.1.1, en la página 42) es cómo actuará la interfaz, la cámara o algún objetos ante los botones presionados del control.

La segunda fase, la actualización de los objetos, como su nombre lo indica, actualiza los objetos en sus posiciones, dimensiones y rotaciones. Esta fase es dependiente de los botones presionados del control y del contexto en el que se encuentre la interfaz, e.g., si el usuario se encuentra en el contexto de *navegación* al presionar el botón `arriba` del control avanzará; en otro caso, si se presiona el mismo botón, estando en el contexto de *traslación de objeto*, el modelo seleccionado se moverá en el eje x.

La tercera fase, el procesamiento de colisiones, es la que realiza el chequeo de colisiones entre los objetos y la cámara. Esta fase debe de ser necesariamente posterior a la actualización de objetos, ya que si un objeto es trasladado muy cerca de la cámara, o la cámara se acerca mucho a un objeto, aquí es donde se rectifica la posición de la cámara evitando que traspase la frontera de algún objeto.

Un vez que las colisiones son calculadas y la posición de la cámara rectificada en caso de ser necesario, todos los vértices de los modelos son tranformados según las propiedades del objeto (traslación, escalamiento y rotación) y mandados a la máquina

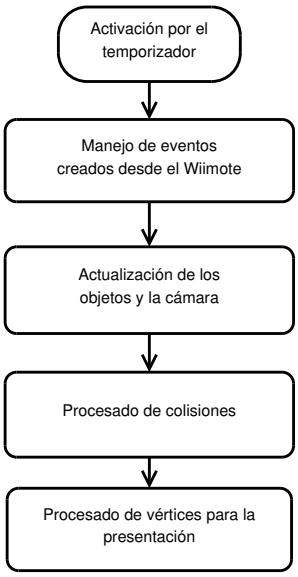


Figura 4.1: Las fases del motor de presentación.

de estados de OpenGL. En la figura 4.1 se muestra un diagrama ejemplificando las cuatro fases accionadas por el temporizador, las flechas indica la secuencia lineal que tienen uno tras otro.

Fue necesario crear una clase que contuviera las propiedades de cada modelo que estuviese en la escena, su posición, su rotación, su tamaño, la textura asignada, sus vértices, etc. Esta clase se fue nombrada `ModeloGL`. Todos los objetos instanciados dentro del mundo virtual están enlazados en una lista doblemente ligada, la cual es recorrida durante la fase de presentación. Para cada objeto dentro de la lista doblemente ligada se analizan sus vértices, se les aplica el escalamiento, la rotación y la posición que tiene asignado en sus propiedades, y por último se texturiza.

El hecho de que dos objetos distintos usen la misma textura, e.g., dos cilindros con la misma textura de metal, es una oportunidad de mejora en cuanto al uso de memoria. En un principio parecía que lo mejor era tener una instancia de textura por cada objeto y que éste guardara la instancia, la implementación era sencilla y rápida pero conforme los mundos virtuales iban creciendo en cuanto a la cantidad de objetos, el programa consumía rápidamente los recursos de la máquina. Las texturas ocupan mucho espacio en memoria.

La solución fue poner también las texturas en una lista doblemente ligada y cada objeto tiene en sus propiedades un puntero que apunta hacia la dirección de memoria de una textura, de esta manera una textura puede ser apuntada por varios modelos a la vez, ahora si existen una cantidad grande de objetos que usen la misma imagen existe en memoria sólo una textura que es apuntada y usada por todos esos modelos.

4.1.1. Mapeado de las funciones del usuario

Para la construcción de mundos virtuales fue requerida la identificación y abstracción de algunas funciones con la cámara así como las actividades concernientes a la transformación de los objetos, una vez que estas funciones son encontradas fue necesario separarlas en subfunciones para así mapearlas en los botones del Wiimote.

Las funciones que ofrece el software son:

- Traslación de la cámara.
- Rotación de la cámara.
- Seleccionar y deseleccionar algún objeto.
- Añadir objetos a la escena.
- Borrar objetos de la escena.
- Clonar objetos de la escena.
- Trasladar objetos de la escena.
- Rotar objetos de la escena.
- Escalamiento de objetos de la escena.
- Texturizado de los objetos.

Estas funciones y sus subfunciones fueron agrupadas en contextos donde el usuario trata con una sola función a la vez, procurando no mezclar tareas y cambiando el comportamiento de los botones dependiendo del contexto en el cual se esté.

En la ventana principal únicamente se muestra una mira en el centro de la ventana, esta mira refleja mediante el cambio de color el contexto en el cual se encuentra la interfaz gráfica, por lo tanto, el tipo de función que desempeñan los botones en dicho momento, por defecto la mira es de color blanco cuando se encuentra en el contexto de navegación, cuando un objeto está en el centro de la mira se considera que está siendo apuntando cambiando al color rojo, si se presiona el botón A el objeto será seleccionado.

Para estas agrupación de funciones se partió desde el concepto que se pueden seleccionar y deseleccionar los objetos existentes en la escena, únicamente a un objeto seleccionado se le pueden aplicar las transformaciones traslación, rotación y escalamiento así como aplicarle una textura; de esta forma se creó el contexto *objeto*

seleccionado que contiene otros cuatro contextos, homónimos a las transformaciones mencionadas, el contexto de *traslación*, de *rotación*, de *escalamiento* y el de *texturizado*.

Contexto de objeto no seleccionado

Cuando no se tiene algún objeto seleccionado es cuando no se está editando el mundo, y entonces la cámara se podrá mover a través del mundo virtual. Este contexto se le nombró como contexto de **objeto no seleccionado** y abarca el contexto de navegación junto con algunas funciones de interacción con los modelos dentro del mundo virtual que sólo requieren ser apuntados por la mira mas no seleccionados, estos son la eliminación de un objeto y la clonación de un objeto.

Por defecto el programa comienza en el contexto de navegación, este contexto requiere un estudio más a fondo para explicar la manipulación de la cámara y los movimientos realizados en el Wiimote que se verá en la subsección 4.2, en la página 45.

La eliminación de objeto es una operación que no se puede deshacer una vez ya realizada, cuando se tiene a algún objeto en la mira mientras se encuentre en el contexto de navegación se puede presionar el botón −, esto hará que la mira se ponga en color cian indicando que el objeto ha sido seleccionado para su eliminación, si se presiona el botón B se cancelará la eliminación, en el caso que se presione el botón − de nueva cuenta procederá con la eliminación y se regresará al contexto de navegación después de esto.

Cuando existen objetos iguales entre si es tedioso realizar la edición del mismo objeto una y otra vez, la clonación de un objeto permite avanzar de forma rápida en este proceso. Cuando se presiona el botón + el último objeto que fue seleccionado es clonado en el mismo lugar y se encontrará seleccionado por defecto, todas las propiedades del objeto original, la posición, rotación, escala y textura son copiadas al objeto clonado.

Cuando se encuentra en el contexto de navegación y se presiona el botón 1 se mostrará el **menú de adición de objeto** donde el usuario se puede auxiliar de la mira móvil para seleccionar alguno de los nueve objetos que se muestran por página, al presionar los botones izquierda o derecha del control se puede cambiar de página navegando de esta forma por todos los modelos que existen en la base de datos. Al seleccionar algún objeto de los mostrados en pantalla se añadirá al mundo virtual en el piso enfrente de la cámara, por defecto el objeto recién añadido se encontrará seleccionado para su edición.

Contexto de objeto seleccionado

Contexto de traslación

El contexto de traslación es con el cual los modelos son cambiados de posición en el mundo virtual, se pueden cambiar los objetos en los tres ejes, si se presiona el botón arriba se desplaza una unidad de manera positiva en el eje x, si se presiona el botón abajo será de manera negativa en mismo eje. Al presionar el botón derecha el desplazamiento será a lo largo del eje z mientras que los botones + y − son para desplazar al objeto a lo largo del eje y. No existe algún límite donde los objetos puedan ser posicionados mas que el límite del campo de visión de la cámara y la posición actual de la misma. El color que toma la mira de la interfaz mientras se encuentra en este contexto es de color amarillo.

Contexto de rotación

Ya que estamos girando al objeto en el espacio en torno de sus tres ejes se usa la nomenclatura para los giros en el espacio de la aeronáutica: cabeceo, guiñada y alabeo. Cuando se presiona los botones arriba o abajo es el cabeceo, con los botones izquierda y derecha es la guiñada mientras que con los botones + y − es el alabeo; es importante señalar que cada vez que se presiona algún botón se gira en cantidades de 30 grados. El color que toma la mira de la interfaz mientras se está en este contexto es de color azul.

Contexto de escalamiento

Un objeto puede ser dimensionado en sus tres ejes por separado, en este contexto al presionar los botones derecha o izquierda se dimensiona a lo largo del eje x del objeto, los botones arriba y abajo son para escalar el objeto en el eje z y los botones + y - son para dimensionarlo en el eje y. Cada vez que algún botón es presionado se modifica el factor de escalamiento en una unidad siendo la menor cantidad posible 1 y no existe un limite superior en el factor de escalamiento.

Contexto de texturizado

Se puede texturizar un objeto cuando este es elegido. Al botón 2 se abre el **menú de texturizado** mostrando el objeto seleccionado con las texturas que se encuentran dentro de la base de datos, su interfaz y navegación igual que el menú de adición de objeto. Dependiendo del modelo que es elegido será el modelo que se muestra en la interfaz, esto con el fin de darle al usuario una retroalimentación de cómo sería el objeto una vez ya texturizado.

Todos los contextos de trabajo que contiene el proyecto se encuentran en el diagrama mostrado en la figura 4.2, el diagrama muestra la navegabilidad entre los distintos

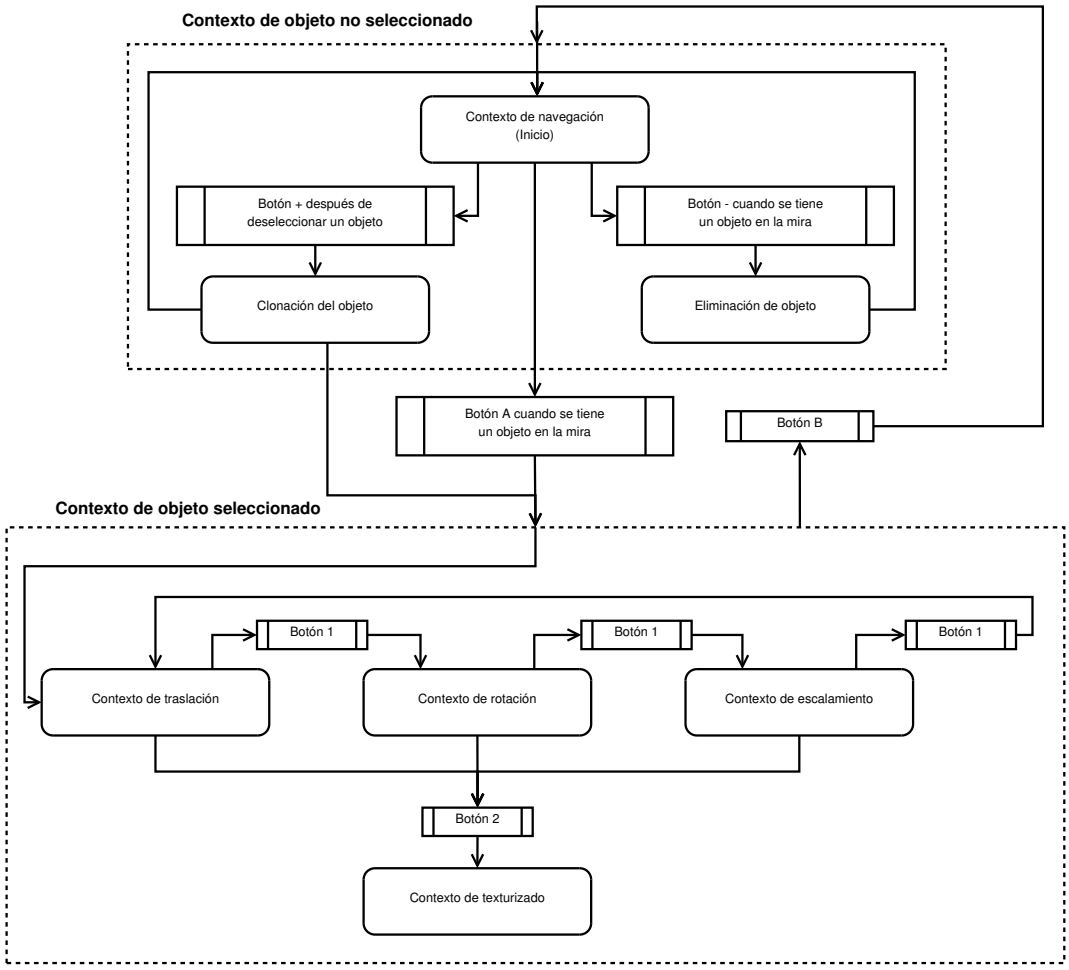


Figura 4.2: Diagrama de navegación entre los contextos de trabajo que tiene la interfaz gráfica. Los cuadros con esquinas redondeadas son los contextos de trabajo en los que puede estár el usuario mientras que las cajas con doble borde son los botones que el usuario debe de presionar para realizar el cambio de contexto.

contextos comenzando con el contexto de navegación, las flechas indican los distintos caminos que pueden tomar los contextos dependiendo de los botones presionados o cuando la acción termina.

4.2. Navegación en el mundo virtual

4.2.1. Manipulación de la cámara

El movimiento de la cámara y su manejo mediante el Wiimote es uno de los puntos más importantes del proyecto, ya que por medio de este movimiento es del que depende la experiencia del usuario con el mundo virtual. Si el movimiento es demasiado

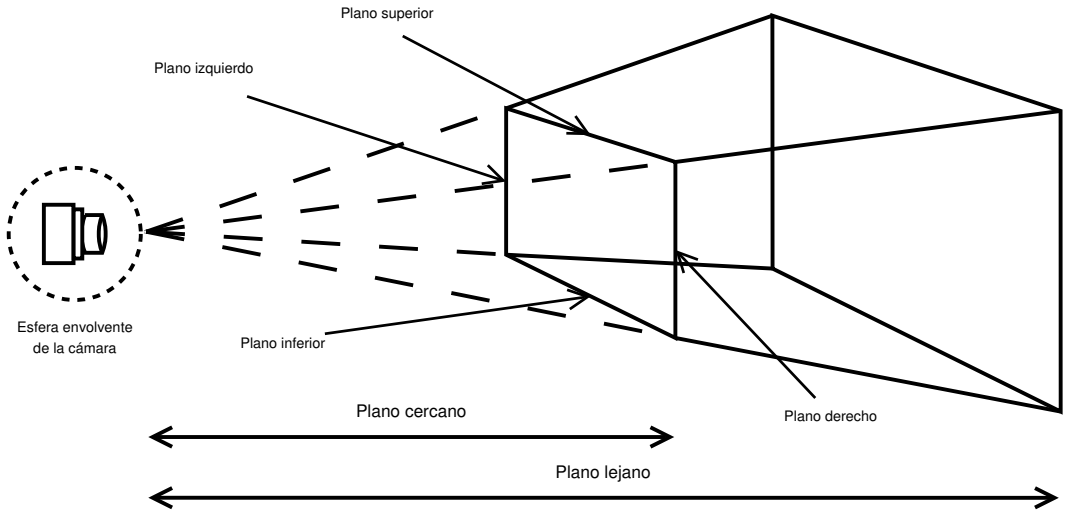


Figura 4.3: El frustum es un volumen definido por seis planos, cualquier vértice que pertenezca a un triángulo o cuadrado dentro de este volumen será puesto en pantalla.

rápido o demasiado lento la experiencia puede ser desagradable, marear o desmotivar al usuario.

Pasa saber el cómo se manipula la cámara dentro del contexto de objeto no seleccionado primero es necesario saber las partes que conforman o describen a la cámara. Las funciones de la cámara en este contexto son las de desplazarse por el mundo virtual en cualquier dirección sobre un plano y de tener la capacidad de voltear a observar en cualquier dirección, por lo tanto se debieron de establecer las propiedades en una clase. La clase `Camara.hpp` tiene definidas el ángulo de observación, una posición en el mundo virtual, dos velocidades de desplazamiento distintas y dos velocidades de rotación además de los planos que definen el frustum mostrado en la figura 4.3.

El frustum es el volumen de visión, este volumen es una pirámide truncada definida por un plano lejano, uno cercano, uno superior, uno inferior, uno lateral derecho y uno lateral izquierdo, todos los vértices existentes en el mundo virtual son procesador por la máquina de estados de OpenGL, pero únicamente los vértices dentro de este volumen son mostrados en pantalla. A partir de esto último se realizó una optimización que será estudiada en el Capítulo 5, en la página 57. El volumen que conforma al frustum y el acomodo de los planos mencionados se muestran en la figura 4.3.

Para obtener la rotación de la cámara se usaron dos ángulos, uno para implementar el cabeceo y otro para implementar la guiñada de la cámara, el alabeo no fue implementado ya que el caminar con la cabeza rotada de esa manera no es una manera natural y puede marear. Estos dos ángulos son modificados mediante los movimientos homólogos del control dando la sensación de coherencia al manejo e la cámara, i.e., si se desea realizar el cabeceo en la cámara se debe de inclinar el control hacia arriba o

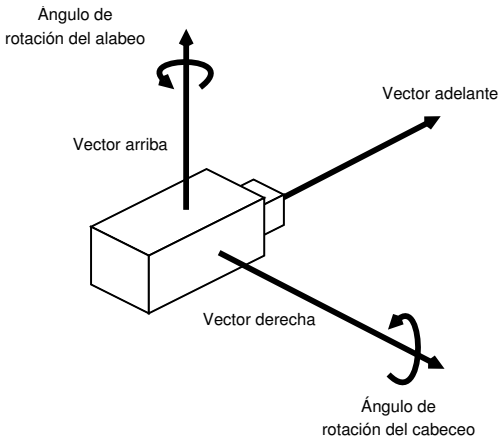


Figura 4.4: La cámara tiene tres vectores que son modificados en función a los ángulos de observación de la cámara. El vector adelante y el vector derecha se descomponen cada vez que son presionados los botones de dirección del Wiimote y los valores de sus componentes son sumados a la posición actual de la cámara.

hacia abajo, si se desea realizar la guiñada en la cámara se debe ladear el control hacia el lado deseado, por cada ciclo de pintado que el control está inclinado se le agrega medio grado al ángulo en cuestión. Mediante la ecuación paramétrica de la esfera y estos dos ángulos se puede determinar hacia que punto en el espacio está observando la cámara.

Para lograr el desplazamiento de la cámara se usaron dos vectores que siempre son actualizados en relación a los ángulos de observación. El primer vector está apuntando hacia el mismo punto de observación de la cámara, i.e., siempre apunta hacia adelante. El segundo vector es perpendicular al primero en el plano xz positivo, i.e., apunta hacia el lado derecho de la cámara; también existe el vector arriba pero este no es modificado por alguna acción del usuario. Cuando el usuario presiona el botón adelante del Wiimote se descompone el vector que está viendo hacia adelante en sus componentes en cada eje, el valor de las componentes x y z es sumada a la posición de la cámara creando de esta forma el desplazamiento en el plano. De forma similar, si se presiona el botón derecha en el control se descompone el vector que apunta hacia un lado de la cámara y el valor de las componentes es sumado a la posición actual de la cámara.

Al tener el punto en el espacio hacia donde observa la cámara actualizada y la posición de la cámara modificada por las acciones del usuario mediante el Wiimote se puede hacer uso de la función `gluLookAt` que pide como parámetros estos datos. En la figura 4.4 se pueden apreciar los ángulos mencionados y los vectores de la cámara.

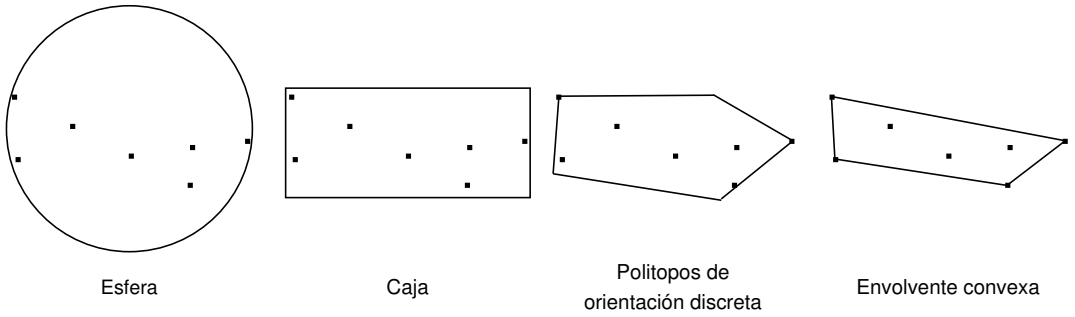


Figura 4.5: Distintos tipos de envolventes, en la izquierda la más eficiente aunque menos exacta, la esfera; pasando gradualmente hasta la más exacta pero menos eficiente, la envolvente convexa.

4.2.2. Colisiones con los objetos

Las colisiones de la cámara con los objetos presentes en el mundo real es un punto clave para el realismo al no permitir que la cámara se encuentre dentro de algún modelo y para probar la eficiencia del motor, ya que éste comprueba la existencia de colisiones de la cámara contra todos los objetos en cada ciclo de pintado. Entonces se debe de tener en mente siempre una implementación eficiente y buscar los algoritmos más rápidos para este propósito sin sacrificar el realismo de la escena.

Un **cuerpo envolvente** o **volumen delimitador** es un cuerpo simple, como una caja o una esfera, que envuelve a todos los vértices de otro cuerpo de naturaleza más compleja con el objetivo de simplificar las pruebas de colisiones. Este cuerpo envolvente debe de ser determinado según el objetivo que se esté buscando, mientras más exactitud se requiera más costoso será el tipo de colisión y determinar el cuerpo envolvente ocupará más memoria y procesos. Entre los cuerpos envolvente más costosos pero exactos se encuentran los politopos de orientación discreta (*DOP* por sus siglas en inglés) y la envolvente convexa (*convex hull*) [16]. Sin embargo, las pruebas de colisiones costosas en memoria no fueron el objetivo de este trabajo, por ello se optó por las opciones más simples y rápidas: las cajas y esferas. En la figura 4.5 se muestran los distintos tipos de cuerpos envolventes, están ordenadas de izquierda a derecha en orden descendente por eficiencia en la prueba de colisión, mientras más cercano esté a la derecha el tipo de envolvente, será menos eficiente pero más exacta.

Existe un paso previo a la prueba de una colisión entre dos objetos, éste es la creación del cuerpo envolvente que consiste en determinar el cuerpo de menor volumen (o un aproximado) que envuelva en su totalidad a los vértices que componen al cuerpo complejo, una vez que ya se cuenta con el cuerpo envolvente se puede proseguir con la prueba de colisión. Este paso se realiza una única vez para cada modelo mientras no exista algún cambio en su geometría, i.e., si se transforma el modelo en su posición, rotación o escala se tiene que calcular de nueva cuenta el cuerpo envolvente. Cabe

mencionar que la cámara siempre está envuelta en una esfera y ésta no cambia su radio en ningún momento.

Por lo antes dicho, cuando un objeto es agregado a la escena, se calcula su cuerpo envolvente y cada vez que en el contexto del objeto seleccionado se transforma al modelo, se calcula de nueva cuenta la envolvente. Ya que el cálculo de colisiones entre esferas es el más rápido, primero se realizó esta implementación. El cálculo de una esfera envolvente aproximada se obtiene de forma rápida mediante el algoritmo diseñado por Ritter [17] donde la esfera resultante es bastante cercana a la óptima y su implementación es sencilla [16].

El método de Ritter consiste en los siguientes pasos:

1. Encontrar en el modelo el par de vértices más alejados entre si, la primer esfera envolvente tendrá como centro el punto medio de estos dos vértices y como radio la mitad de la distancia existente entre los dos vértices.

2. Posteriormente se procesan de nueva cuenta todos los vértices del modelo buscando por algún vértice que se encuentre fuera de esta primer esfera envolvente, en caso de encontrar alguno se recalcula la esfera de tal forma que envuelva al vértice exterior y a la esfera anterior, esto quiere decir que el nuevo diámetro de la nueva esfera será la longitud de la línea descrita desde el vértice exterior pasando por el centro de la esfera anterior hasta que la línea toque la superficie opuesta de la esfera, mientras que el nuevo centro es el punto medio de esta línea descrita.

3. Al finalizar se tendrá una esfera que envuelva a todos los vértices. Este procedimiento se puede apreciar de forma gráfica en la figura 4.6, donde comienza con una colección de puntos arbitrarios, se encuentran los puntos más lejanos y posteriormente se procede a envolver a los puntos que quedaron fuera del círculo inicial, creando un nuevo círculo con un nuevo centro.

De esta forma se obtiene la esfera envolvente de cada objeto en el mundo virtual, en este caso todas las colisiones eran esferas contra esferas. Para calcular una colisión de este tipo es necesario comparar la distancia existente entre el centro de ambas esferas, si esa distancia es igual o menor a la suma de sus radios significa que existe una colisión entre esas dos esferas, ya que se trata de la esfera envolvente de un modelo y la esfera envolvente de la cámara esto es señal de que no se podrá avanzar más.

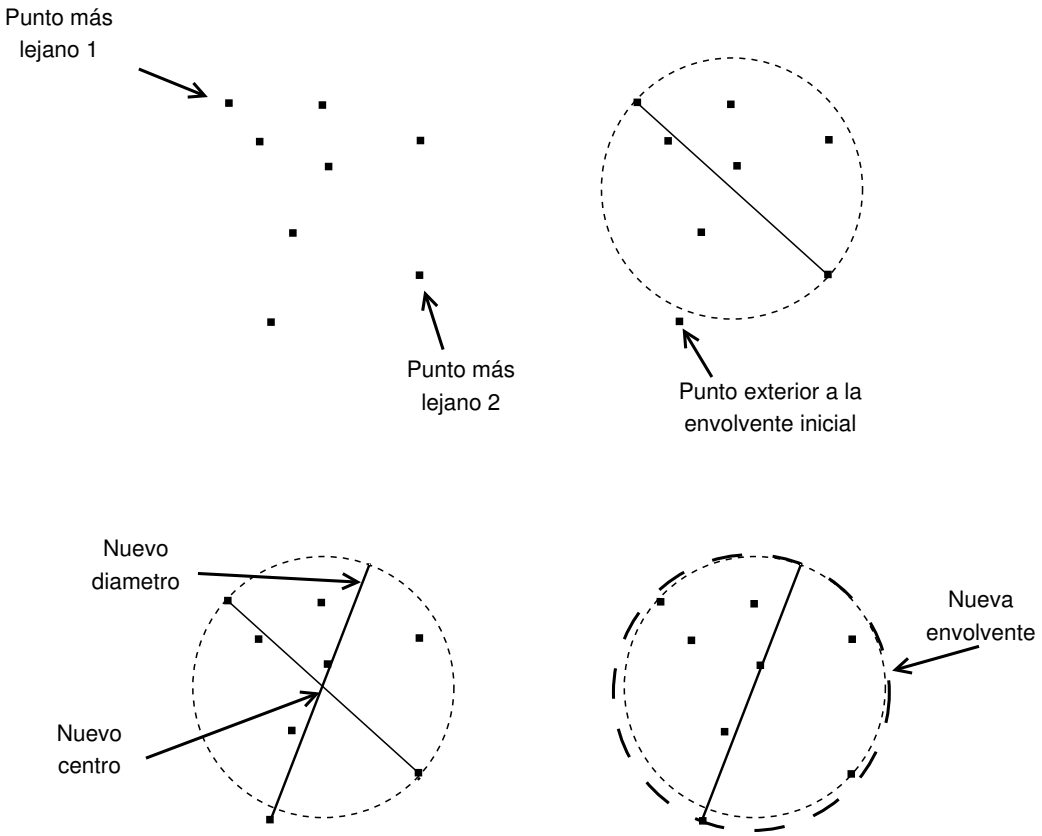


Figura 4.6: Método de Ritter para encontrar la esfera que contiene un conjunto de puntos. En primer lugar tenemos una colección de puntos arbitrarios, posteriormente se encuentran los dos puntos más lejanos de esta colección, este será el diámetro de la primer envolvente. Al encontrar un punto exterior a la envolvente se recalcula la envolvente, su nuevo diámetro será la longitud de la línea descrita desde el punto exterior pasando por el centro de la vieja envolvente hasta que la línea toque la superficie opuesta. Finalmente, el nuevo centro es el punto medio de la línea descrita.

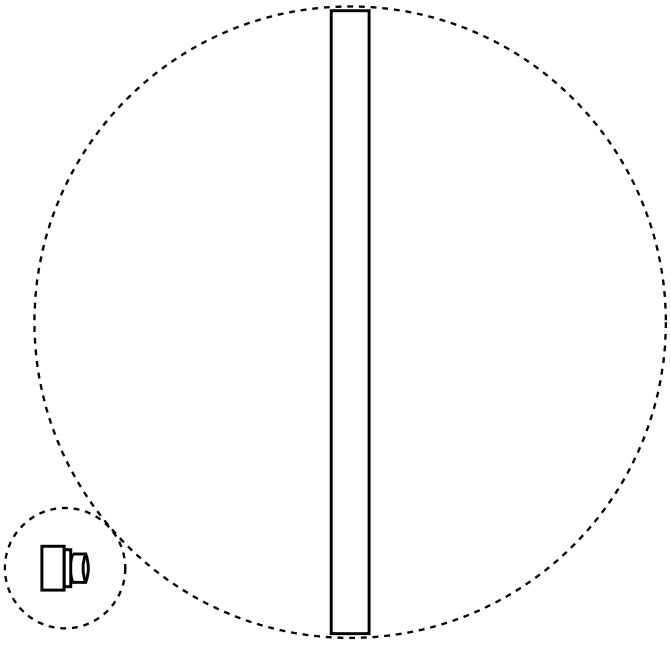


Figura 4.7: Cuando un objeto es muy grande la esfera envolvente desperdicia mucho volumen alrededor del objeto y no permite a la cámara acercarse o pasar por espacios donde si debería pasar.

Este tipo de colisión era buena cuando se trataban de objetos relativamente pequeños, sin embargo cuando se trataba de objetos grandes la esfera envolvente abarcaba mucho volumen alrededor del objeto impidiendo que la cámara se acercará al modelo, en la figura 4.7 se puede apreciar un caso donde la esfera envolvente no servía para esta implementación, por lo tanto se optó por el segundo tipo de colisiones, las colisiones de esferas contra cajas.

Encontrar la caja envolvente es más sencillo que encontrar la esfera envolvente: se requiere encontrar los valores mínimos y máximos en x, y, y z de los vértices del modelo, arrojando seis cifras siendo éstas las únicas necesarias para poder describir una caja. El conjunto de los mínimos es un vértices de la caja mientras que el conjunto de los máximos es otro vértice de la caja.

La prueba de colisión entre caja y esfera consiste en encontrar el punto más cercano sobre la superficie de la caja (o dentro de su volumen) al centro de la esfera. Para realizar esto existen 4 casos distintos a considerar: el primero de ellos es cuando el centro de la esfera se encuentra dentro de la caja, por lo tanto el punto de la caja más cercano al centro de la esfera es mismo centro en si, i.e., el punto de la caja más cercano al centro de la esfera tiene las mismas coordenadas que el centro.

El segundo caso a considerar es cuando el centro de la esfera se encuentra dentro

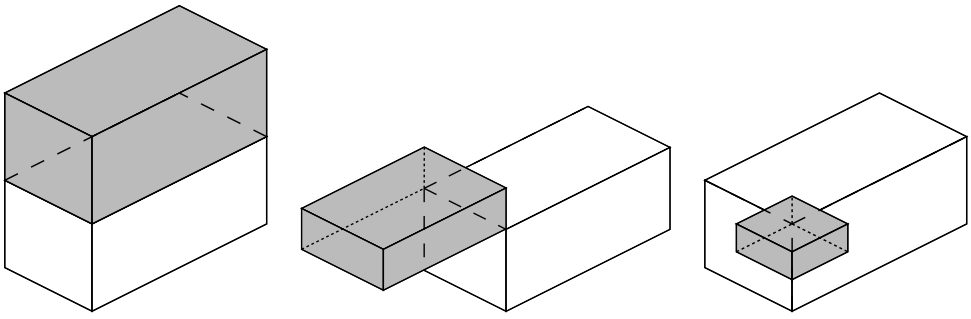


Figura 4.8: Se aprecian las regiones de Voronoi correspondientes a una cara (a), a una arista (b) y a un vértice (c) de una caja.

de la **región de Voronoi**[1] de alguna de las caras de la caja, siendo este el caso el punto más cercano de la caja se encuentra realizando una proyección ortogonal del centro en la cara de la caja. En la figura 4.8 se encuentran en gris las regiones de Voronoi correspondientes a una cara, a una arista y a un vértice.

El tercer caso es cuando el centro de la esfera se encuentra en la región de Voronoi correspondiente a alguna de las aristas de la caja, al ser este caso, el punto más cercano de la caja al centro se encuentra al realizar una proyección ortogonal del centro a la arista de la caja.

El cuarto y último caso es cuando el centro se encuentra dentro de la región de Voronoi perteneciente a algún vértice de la caja, si es este caso el punto más cercano de la caja al centro es el vértice en cuestión.

Una vez que ya se ha determinado el punto más cercano de la caja al centro de la esfera la comprobación de colisión consiste en determinar la distancia existente entre estos dos puntos, si dicha distancia es menor al radio de la esfera significa que la colisión entre estos dos objetos está presente, como se puede inducir, si el centro de la esfera se encuentra dentro de la caja siempre existirá colisión, por lo tanto no es necesario algún tipo de comprobación.

Usando colisiones caja-esfera la cámara no pasará a través de los objetos dando una sensación más realista de coherencia con los objetos presentes en el mundo virtual.

Al tener la colisión de la esfera de la cámara contra la caja envolvente de los objetos se evita el problema del desperdicio de espacio excesivo y la cámara se puede acercar más a los objetos, esto se puede apreciar en la figura 4.9, donde la cámara

[1]Región de Voronoi: dado un conjunto de puntos S en el espacio, la región de Voronoi de un punto P en S está definida por el conjunto de puntos más cercanos a P que a cualquier otro punto en S.

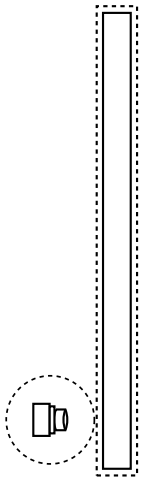


Figura 4.9: Sin importar el tamaño del objeto, la envolvente del tipo caja abarca menos volumen permitiendo a la cámara acercarse más al objeto.

está más cerca del objeto que en la figura 4.7 de la página 51.

4.3. Almacenamiento y recuperación del mundo virtual

Al tener un mundo ya editado es necesario poder guardarlo de alguna forma para su futura recuperación y de nueva cuenta editarlo. Para este fin se usa el teclado y el ratón ya que es necesario navegar entre las carpetas del sistema de archivos y asignarle un nombre al archivo que va a contener al mundo virtual.

Para crear un archivo nuevo, o sobreescribir uno ya existente, en el sistema de archivos se usó clases de la biblioteca `QFileDialog` de QT, que abren un dialogo donde se puede navegar por el sistema de archivos buscando un archivo existente o crear uno nuevo regresando la dirección absoluta de éste. Al tener el nombre y la dirección absoluta puede abrirse y escribir en él de la forma tradicional con funciones de entrada y salida estándar.

Lo primero que se guarda en el archivo contenedor del mundo virtual son las propiedades de la cámara, esto se debe a que al momento de recuperar algún otro mundo virtual la cámara puede quedar "atrapada" dentro de un cuerpo o de algún perímetro hecho de objetos que no permita la movilidad de la cámara. Al guardar la última posición que tuvo en el mundo virtual se garantiza que al recuperarlo la cámara se encontrará en un punto donde se puede navegar y no estará atascada entre objetos.

Un ejemplo de como las propiedades de la cámara son guardadas en el archivo se muestra en el listado 4.1, la primer línea está etiquetada como `posCamara` seguida

por tres números de punto flotante indicando la posición en x, y, y z respectivamente. La segunda línea está etiqueta como `angCamara` y es seguida por dos números que indican la inclinación en grados del cabeceo y la guiñada de la cámara, respectivamente.

```
posCamara -228.955688 23.000000 457.778809
angCamara 28.000000 87.500000
```

Listado 4.1: Ejemplo de cómo son guardadas las propiedades de la cámara en el archivo.

Posterior al guardado de la cámara comienza el registro de los objetos que se encuentran en la escena, esto consiste en recorrer la lista de los modelos guardando de igual manera sus propiedades. En el listado 4.2 en primer lugar está la etiqueta `id` que guarda el número identificador único que tiene en el repositorio de modelos, posteriormente se encuentra la etiqueta `posicion` con los tres valores, x, y, y z de sus coordenadas en el espacio. La etiqueta `rotacion` guarda los valores del cabeceo, alabeo y guiñada del objeto en grados mientras que la etiqueta `escala` se refiere al factor de dimensionado de cada eje del modelo, finalmente la etiqueta `textura` guarda el número único identificador que tiene la textura dentro de la base de datos. Existe un registro similar por cada modelo dentro del mundo virtual, permitiendo así una recuperación completa del mundo virtual.

```
id 1
posicion 1334 8 -1402
rotacion 0 0 0
escala 16 16 52
textura 37
```

Listado 4.2: Ejemplo de cómo son guardadas las propiedades de un modelo en el archivo.

La recuperación de un mundo virtual consiste en primer lugar en eliminar los modelos del mundo virtual y limpiar ambas listas doblemente ligadas, la lista de texturas y la lista de modelos.

Para poder leer el archivo que contiene la descripción de un mundo virtual se escribió un analizador sencillo de etiquetas: para cada etiqueta que encuentra actúa conforme al tipo, e.g., si se encuentra con la etiqueta `textura` el analizador buscará por un único valor del tipo entero sin signo, al leerlo realizará una consulta en la lista doblemente ligada en busca de dicha textura, en caso de ya estar dentro de la lista enlazará al modelo con la textura ya creada, en caso de que no se encuentre se hará una consulta a la base de datos para cargarla en memoria y enlazarla. Otro ejemplo es cuando se encuentra con la etiqueta `escala`, el analizador leerá los tres valores enteros sin signo y llamará a los métodos correspondientes de `ModeloGL` para asignar dichos valores de escala al modelo en cuestión.

Todo este proceso se repite por cada etiqueta `id` que encuentre el analizador del archivo hasta ser encontrado el final del archivo resultando en dos nuevas listas ligadas.

El resultado de esta parte se puede apreciar en el apéndice C en la página 85 donde se muestran capturas de pantalla de la interfaz, de los distintos contextos y de la creación de un mundo virtual.

Capítulo 5

Optimización de la Visualización

Al crear un motor de presentación en tiempo real se debe de tener en mente la eficiencia, ya sea a bajo nivel en el código como a alto nivel en la arquitectura, procurando un balance entre el uso de memoria y el uso del procesador.

Encontrar oportunidades para conseguir una mejora en el rendimiento del sistema es necesario para que cada cuadro mostrado tome un menor tiempo en procesarse, de esta forma, se pueden construir y presentar mundos virtuales con más objetos y detalles.

En este proyecto se implementaron y compararon varias técnicas de graficación para comprobar si existe o no una mejora en el rendimiento en el empleo de dichas técnicas; se probó el nivel de detalle en las texturas utilizadas (*mipmapping* o mapas MIP), el descartar objetos mediante el chequeo de la colisión del frustum (*frustum culling*) contra esferas ó cajas, y la eliminación de objetos diminutos que aportan poco o nada a la escena mediante dos técnicas distintas, el cambio del plano lejano del frustum y el mapeado del objeto en las coordenadas de la ventana, i.e., eliminar el objeto según su tamaño relativo en el plano de proyección (ó en la pantalla).

5.1. Nivel de detalle en texturas

El **nivel de detalle** en las texturas implica el tener en memoria una colección de imágenes redimensionadas pertenecientes a una misma textura, cada imagen es una versión reducida en tamaño de su antecesor, esto con el objetivo de usar alguna de estas versiones de menor tamaño para texturizar a un objeto dependiendo de la distancia a la que se encuentre de la cámara o del tamaño relativo que presenta en la escena, mientras más lejano se encuentre de la cámara se usará una textura de menor calidad. A esta técnica se le conoce como mapas MIP (mipmapping en inglés), el acrónimo MIP proviene del latín *multum in parvo* que significa "mucho en poco", el contenido de esta sección se obtuvo de [18].

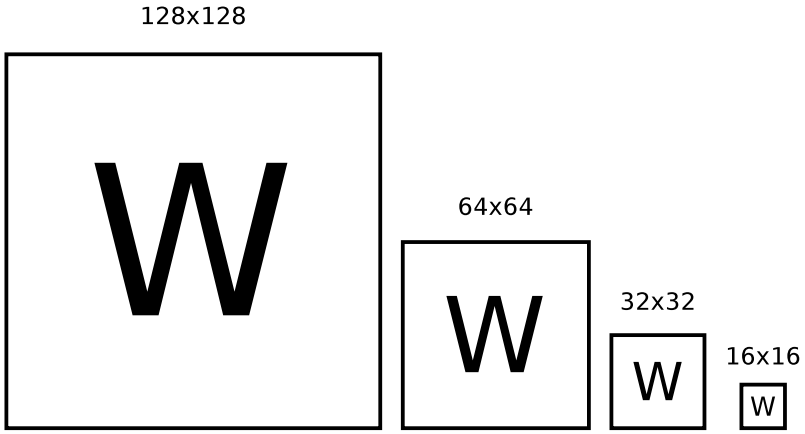


Figura 5.1: Los mapas MIP consisten en la textura original y una serie de texturas que son una versión reducida de la anterior.

Esta técnica toma ventaja del foco de atención del observador, la mayoría del tiempo el observador está prestando atención a los objetos que se encuentran en el primer plano, a los objeto más grandes, mientras que a los objetos que se encuentran en un plano secundario no son observados directamente pero ayudan al observador a dar contexto a una escena, sin embargo no es necesario presentar todos los detalles de los objetos secundarios o se puede presentar una versión más simple de éstos. Un ejemplo de un mapa MIP se presenta en la figura 5.1 donde se encuentra el ejemplo de una textura que originalmente es de 128×128 píxeles junto con cuatro versiones reducidas en tamaño, la segunda es un cuarto del tamaño la primera, la tercera es un un cuarto del tamaño de la segunda y así sucesivamente.

El uso de mapas MIP tiene la desventaja de que se usará más espacio en memoria para almacenar las texturas de menor detalle, este espacio extra se deduce realizando la suma $\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \frac{1}{1024} + \ldots$, esto converge en $\frac{1}{3}$, i.e., usar mapas MIP usa, a lo más, un tercio de memoria adicional que sólo usar la textura original.

La biblioteca GLU ofrece una función para generar los mapas MIP y usarlos de forma automática. La forma convencional de cargar una textura por medio de la función **glTexImage2D** que ofrece OpenGL, carga en memoria la textura que se le indique y esta versión es la única usada para texturizar los modelos, por otro lado, la función **gluBuild2DMipmaps** carga en memoria la textura además de crear tres versiones reducidas de dicha textura; aunque el número de versiones puede ser modificado se usaron sólo tres ya que se evita sacrificar memoria para las versiones pequeñas.

En la figura 5.2 se muestra del lado izquierdo un segmento del mundo creado donde las cajas son texturizadas usando una única versión de la textura, las texturas no pierden detalle si el objeto está lejano a la cámara, mientras que del lado derecho

Figura 5.2: Los mapas MIP son texturas creadas previamente, mientras más lejano se encuentre algún objeto de la cámara se usará una versión de la textura de menor detalle. En el lado izquierdo se encuentra una escena texturizada con una única versión de las texturas, no importa si los objetos se encuentran lejos de la cámara, la imagen no perderá detalles. Del lado derecho se encuentra la misma escena pero los objetos son texturizados usando mapas MIP donde los objetos pierden detalle si están distantes de la cámara. Abajo de cada escena se encuentran los mismos elementos para su comparación.

se encuentra la misma escena pero ahora las cajas son texturizadas usando distintas versiones de una textura, las texturas no se ven tan definidas si el objeto se encuentra lejano a la cámara.

Para medir y comparar las distintas implementaciones se creó un mundo virtual de prueba que consta de 35,937 cubos (estos son 287,496 vértices o 431,244 polígonos), cada cubo es texturizado con una imagen de la base de datos al azar. Se tomó el tiempo en milisegundos de una muestra de 500 cuadros eliminando los 50 tiempos menores y los 50 tiempos mayores para finalmente obtener el promedio de los 400 tiempos restantes. Las ejecuciones se realizaron en una Mac Mini modelo Mid 2011 con un procesador Intel Core i5 a 2.5 GHz, 4 GB de RAM y una tarjeta de video AMD Radeon HD 6630M con 256 MB de VRAM, con un sistema operativo Mac OS X Lion 10.7.5. En la figura 5.3 se encuentra una imagen del mundo virtual de prueba creado para las mediciones de los tiempos.

En el caso de esta comparación se obtuvieron dos tiempos prácticamente iguales, en el caso de usar una única textura el tiempo promedio de presentación fue de 206.13059 milisegundos mientras que el tiempo promedio de usar mapas MIP fue de 204.45167 milisegundos por cada cuadro, estos tiempos son independientes de la cantidad de objetos que son visualizados. Efectivamente, existe una mejora en los tiempos de presentación, aunque sólo se obtuvo 1.67891 milisegundos de ganancia

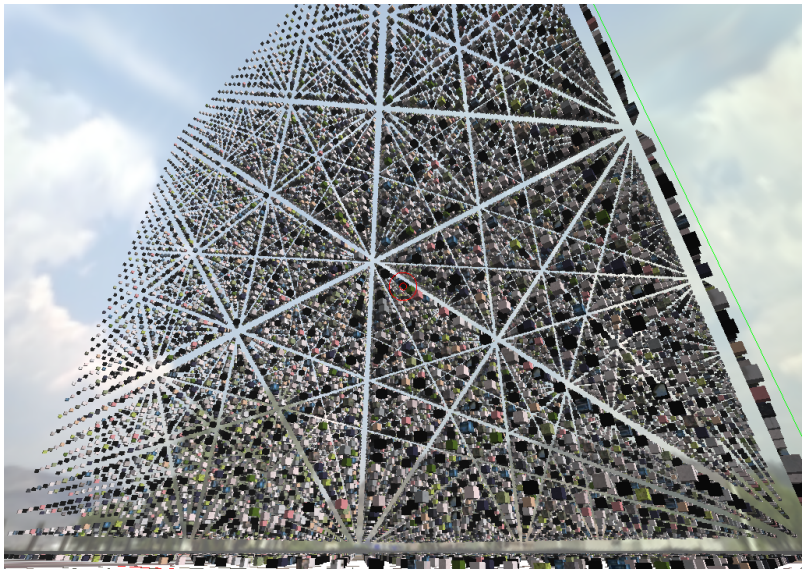


Figura 5.3: Se creó un mundo virtual de prueba para la medición de los tiempos de presentación, consta de 35,937 cubos texturizados al azar, esta cantidad es el resultado de 33×33×33 cubos resultando en un cubo de gran tamaño. Tal cantidad de cubos garantizan que se podrá explorar las capacidades de las optimizaciones implementadas.

en el rendimiento, al tratarse de presentación de gráficas en tiempo real se optó por mantener la implementación que usa los mapas MIP para combinarla con las futuras optimizaciones y conservar esa ganancia aunque sea mínima.

5.2. Descartar objetos con el frustum

Los objetos que conforman un mundo virtual no suelen estar todos en pantalla al mismo tiempo, sin embargo OpenGL realiza el cálculo pertinente para cada vértice sin importar si el objetivo será presentado en pantalla o no. Se puede mejorar este proceso considerando que los objetos que están fuera del volumen del frustum de visualización no deben procesarse con la máquina de estados de OpenGL.

Para realizar la elimación de los objetos fuera del frustum, es necesario obtener las posiciones de los planos que lo conforman, así si la cámaraa se desplaza y gira en el mundo virtual la posición de los planos cambian , por lo tanto, es necesario obtener la matriz que contiene los parámetros de la cámara (modelview) y la matriz de proyección ya que al multiplicar ambas matrices se obtiene el espacio de recorte (clipping) el cual tiene implícito los valores de los planos [19][20]. Esto se realiza después de poner la cámara en su lugar y orientación pero antes de mandar a dibujar los objetos, únicamente es una vez por cada cuadro.

La parte correspondiente de este proceso se encuentra en `camara.cpp` en el método `extraeFrustum`, en el listado 5.1 se muestra la parte correspondiente a la obtención de ambas matrices y a su multiplicación para tener como resultado el área de recorte (*clipping*).

```
/* Obtiene la matriz de proyección actual desde OpenGL. */
glGetDoublev( GL_PROJECTION_MATRIX, proj );

/* Obtiene la matriz de vista del modelo desde OpenGL. */
glGetDoublev( GL_MODELVIEW_MATRIX, modl );

/* Multiplica ambas matrices. */
clip[ 0] = modl[ 0] * proj[ 0] + modl[ 1] * proj[ 4] + modl[ 2] * proj[
    8] + modl[ 3] * proj[12];
clip[ 1] = modl[ 0] * proj[ 1] + modl[ 1] * proj[ 5] + modl[ 2] * proj[
    9] + modl[ 3] * proj[13];
clip[ 2] = modl[ 0] * proj[ 2] + modl[ 1] * proj[ 6] + modl[ 2] * proj
    [10] + modl[ 3] * proj[14];
clip[ 3] = modl[ 0] * proj[ 3] + modl[ 1] * proj[ 7] + modl[ 2] * proj
    [11] + modl[ 3] * proj[15];

clip[ 4] = modl[ 4] * proj[ 0] + modl[ 5] * proj[ 4] + modl[ 6] * proj[
    8] + modl[ 7] * proj[12];
clip[ 5] = modl[ 4] * proj[ 1] + modl[ 5] * proj[ 5] + modl[ 6] * proj[
    9] + modl[ 7] * proj[13];
clip[ 6] = modl[ 4] * proj[ 2] + modl[ 5] * proj[ 6] + modl[ 6] * proj
    [10] + modl[ 7] * proj[14];
clip[ 7] = modl[ 4] * proj[ 3] + modl[ 5] * proj[ 7] + modl[ 6] * proj
    [11] + modl[ 7] * proj[15];

clip[ 8] = modl[ 8] * proj[ 0] + modl[ 9] * proj[ 4] + modl[10] * proj[
    8] + modl[11] * proj[12];
clip[ 9] = modl[ 8] * proj[ 1] + modl[ 9] * proj[ 5] + modl[10] * proj[
    9] + modl[11] * proj[13];
clip[10] = modl[ 8] * proj[ 2] + modl[ 9] * proj[ 6] + modl[10] * proj
    [10] + modl[11] * proj[14];
clip[11] = modl[ 8] * proj[ 3] + modl[ 9] * proj[ 7] + modl[10] * proj
    [11] + modl[11] * proj[15];

clip[12] = modl[12] * proj[ 0] + modl[13] * proj[ 4] + modl[14] * proj[
    8] + modl[15] * proj[12];
clip[13] = modl[12] * proj[ 1] + modl[13] * proj[ 5] + modl[14] * proj[
    9] + modl[15] * proj[13];
clip[14] = modl[12] * proj[ 2] + modl[13] * proj[ 6] + modl[14] * proj
    [10] + modl[15] * proj[14];
clip[15] = modl[12] * proj[ 3] + modl[13] * proj[ 7] + modl[14] * proj
    [11] + modl[15] * proj[15];
```

Listado 5.1: La obtención y multiplicación de la matriz de proyección y la matriz de parámetros de la cámara.

Posteriormente se pueden extraer los valores de cada plano, esta extracción de parámetros de los planos arroja los números correspondientes a la ecuación general de un plano que tiene la forma: $Ax + By + Cz + D = 0$, en el listado 5.2 se muestra el código correspondiente a esta parte.

```
/* Extrae los numeros para el plano derecho. */
frustum[0][0] = clip[ 3] - clip[ 0];
frustum[0][1] = clip[ 7] - clip[ 4];
frustum[0][2] = clip[11] - clip[ 8];
frustum[0][3] = clip[15] - clip[12];

/* Extrae los numeros para el plano izquierdo. */
frustum[1][0] = clip[ 3] + clip[ 0];
frustum[1][1] = clip[ 7] + clip[ 4];
frustum[1][2] = clip[11] + clip[ 8];
frustum[1][3] = clip[15] + clip[12];

/* Extrae los numeros para el plano del inferior. */
frustum[2][0] = clip[ 3] + clip[ 1];
frustum[2][1] = clip[ 7] + clip[ 5];
frustum[2][2] = clip[11] + clip[ 9];
frustum[2][3] = clip[15] + clip[13];

/* Extrae los numeros para el plano superior. */
frustum[3][0] = clip[ 3] - clip[ 1];
frustum[3][1] = clip[ 7] - clip[ 5];
frustum[3][2] = clip[11] - clip[ 9];
frustum[3][3] = clip[15] - clip[13];

/* Extrae los numeros para el plano lejano. */
frustum[4][0] = clip[ 3] - clip[ 2];
frustum[4][1] = clip[ 7] - clip[ 6];
frustum[4][2] = clip[11] - clip[10];
frustum[4][3] = clip[15] - clip[14];

/* Extrae los numeros para el plano cercano. */
frustum[5][0] = clip[ 3] + clip[ 2];
frustum[5][1] = clip[ 7] + clip[ 6];
frustum[5][2] = clip[11] + clip[10];
frustum[5][3] = clip[15] + clip[14];
```

Listado 5.2: Obtención de los planos de recorte que delimitan al volumen del frustum.

Una vez que tenemos los seis planos podemos determinar si un vértice es visible o no, esto será cuando el vértice se encuentre enfrente de los seis planos del frustum al mismo tiempo, se determina calculando la distancia existente entre el vértice y el plano, si la distancia es positiva se encontrará enfrente de dicho plano, si la distancia es negativa estará detrás del plano. Para determinar la distancia existente entre el vértice y el plano se suman las componentes de ambos elementos elementos, $Ax + By + Cz + D$, si la comparación de las seis ecuaciones de los planos resulta

positiva significa que el vértice está dentro del frustum.

Realizar esta comparación para cada vértice de algún modelo no es práctico, es por eso que se volvieron a ocupar los volúmenes envolventes. En el caso de la esfera el criterio es el mismo que un sólo vértice, siendo el punto a comparar el centro de la esfera, si la distancia entre el centro de la esfera es positivo y, menor o igual, que el radio significa que la esfera se encuentra enfrente del plano en cuestión, si esto se cumple para los seis planos del frustum la esfera, o una parte de ella, está dentro del frustum de visualización. En el momento que se cumplan estas condiciones es cuando se mandan los vértices del objeto a la máquina de estados de OpenGL.

La verificación mediante cajas envolventes usa las misma idea básica de la verificación de un punto: se comprueba si alguno de los ocho vértices de la caja se encuentra dentro del volumen frustum, si alguno de los ocho vértices de la caja está dentro del frustum significa que el modelo que es contenido por la caja atraviesa el volumen frustum y debe de ser pintado.

Como es de suponerse esta comprobación del punto dentro de frustum también funcionará para otros tipos de cuerpos envolventes como los politopos de orientación discreta y las envolventes convexas comprobando cada uno de sus vértices contra los planos del frustum.

Desde aquí se puede conjeturar sobre los tiempos que arrojará cada una de estas implementaciones, la comparación contra esferas requiere menos cálculos que la comprobación contra las cajas, sin embargo, los tiempos son muy similares en la práctica. Los tiempos de presentación de cada cuadro van en función de los objetos que son descartados por el frustum, i.e., mientras más objetos estén fuera del volumen frustum menor tiempo tomará en presentar el cuadro, por ello se realizó la medición de los tiempos cuando se están observando el 10 %, 20 %, 30 %, 40 %, 50 %, . . . , 100 % de los cubos del mundo virtual de prueba, los resultados de ambas implementaciones son reportadas en la gráfica de la figura 5.4, en dicha gráfica se muestra sólo el segmento correspondiente a los 42 milisegundos, es decir, el mayor tiempo en que un cuadro se debe de procesar para ofrecer los 24 cuadros por segundo como mínimo, por lo tanto, los tiempos que ofrece la versión que sólo cuenta con los mapas MIP no aparece en la gráfica ya que sus tiempos son mayores a 200 milisegundos.

Como se puede apreciar de la figura 5.4 sí existe una optimización al emplear esta técnica de descartamiento, esto se produce debido a que no hay necesidad de mandar los vértices, sus normales y las coordenadas de textura a la máquina de estados, ahorrando la llamada a tres funciones, `glVertex3f`, `glNormal3f`, `glTexCoord2d`, respectivamente.

Como se puede apreciar en la figura 5.4 la implementación que usa esferas para

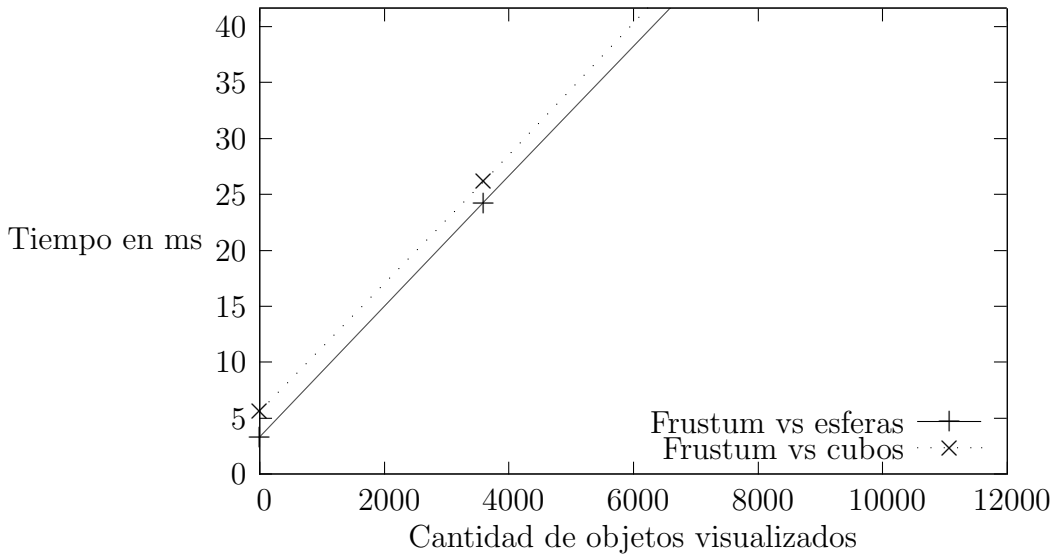
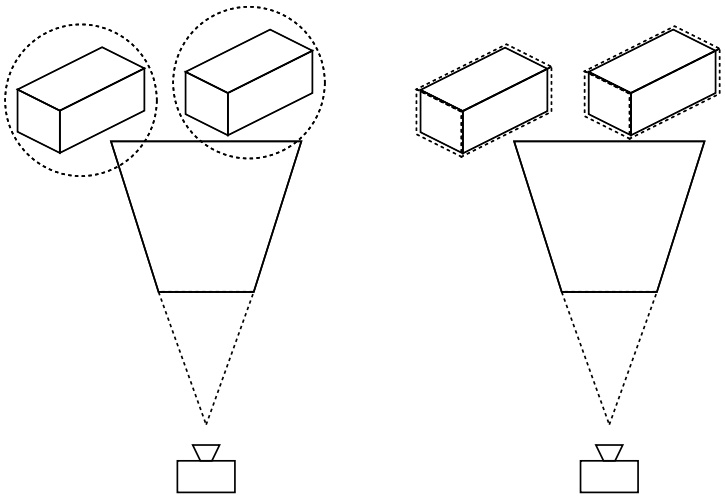


Figura 5.4: Los tiempos que arroja la optimización de descartamiento mediante frustum comparando contra esferas y contra cajas visualizando una cantidad cada vez mayor de objetos a la vez.

descartar modelos fuera del frustum es más eficiente que cuando se usan cajas, sin embargo, dicha versión en las pruebas contaba con la misma característica que el problema descrito en la subsección 4.2.2 en la página 51, la esfera desperdicia demasiado volumen en algunos casos, ese volumen desperdiciado puede estar dentro del frustum creando un falso positivo y mandando a pintar objetos que realmente no están dentro del frustum, dicho problema se puede apreciar de mejor forma en la figura 5.5. Aunque la envolvente del tipo caja no está exento de crear falsos positivos ya que también desperdicia volumen en objetos que no son hexaedros, el desperdicio es, en cierta medida, menor; ya que envuelve a la mayoría de los objetos (a excepción de los objetos completamente esféricos) sin desperdiciar tanto volumen, por esto es que se decidió elegir la comparación contra las cajas para continuar con el proyecto.

5.3. Eliminación de objetos diminutos

El tamaño aparente de los objetos en la pantalla depende tres factores: su tamaño real, la distancia a la que se encuentre de la cámara y el tamaño de la ventana en donde es presentado el mundo virtual. En el caso de dos objetos que sean de las misma dimensiones, si uno está mas alejado de la cámara que el otro, el primero tendrá un tamaño aparentemente menor que el segundo, además de que el segundo estará en un plano más cercano a la cámara y llamará más la atención del observador, los objetos en planos secundarios sirven para dar contexto a una escena, sin embargo, existen objetos que por su lejanía de la cámara y tamaño se muestran diminutos en la escena.



(a) (b)

Figura 5.5: En algunos casos la esfera envolvente desperdicia mucho espacio resultando en falsos positivos cuando se compara contra el frustum (a), en cambio, la caja envolvente desperdicia menos espacio evitando mandar a dibujar objetos que realmente no son necesarios (b).

Este tipos de objetos suelen pasar desapercibidos por su tamaño relativo o por no estar en primer plano, un ejemplo de lo dicho está en la figura 5.6 donde se encuentran dos arboles del mismo tamaño pero en la proyección tienen distinto tamaño relativo. Esta optimización también toma ventaja del foco de atención del observador: quita elementos de la escena que aportan poco o nada al contexto, es decir, los objetos que el ojo humano (mejor dicho, el cerebro) desecha por su tamaño. Posiblemente se pueda tomar como analogía el tipo de codificación mp3 que usa un algoritmo con perdida de información consiguiendo un tamaño menor de archivo quitando frecuencias de audio que el oído humano no percibe.

La primera implementación de esta propuesta fue mediante el mapeado del objeto en las coordenadas de la ventana. Para determinar el tamaño relativo de un objeto es necesario saber cuales serán sus coordenadas finales en el plano 2D de la pantalla, es decir, realizar la proyección de las coordenadas del objeto en las coordenadas reales de la ventana donde se muestra el mundo virtual, pero proyectar todos los vértices que tiene un objeto hacia las coordenadas de la ventana puede que no resulte en una ganancia del rendimiento, por ello se decidió auxiliarse de nueva cuenta en los cajas envolventes ya que de esta forma solo hay que proyectar los 8 vértices de la caja.

Mapear estos puntos del espacio al plano de proyección de la cámara, y posteriormente, a las coordenadas de la ventana se necesita multiplicar las coordenadas del punto por la matriz que contiene los parámetros de la cámara y la matriz de

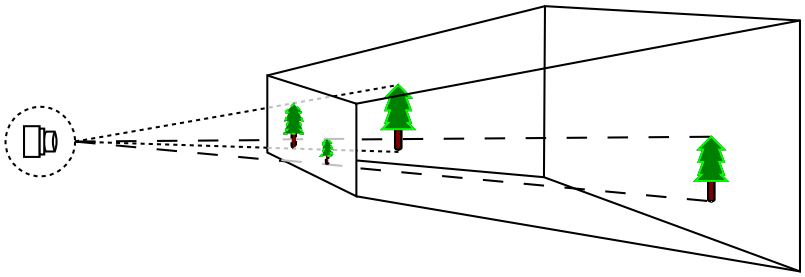


Figura 5.6: Cuando dos objetos del mismo tamaño están a distinta distancia de la cámara aparentan tamaños relativos distintos, dependiendo del tamaño del objeto, su distancia y el tamaño de la ventana puede que el objeto tenga un tamaño relativo muy pequeño y sea imperceptible o descartable.

proyección, es decir, $v' = PMv$ donde P es la matriz de proyección, M es la matriz con los parámetros de la cámara (posición en el espacio y rotación, mientras que v son las coordenadas del punto en el espacio, expresado en forma de matrices esto es:

$$v' = \begin{bmatrix} P_{1,1} & P_{1,2} & P_{1,3} & P_{1,4} \\ P_{2,1} & P_{2,2} & P_{2,3} & P_{2,4} \\ P_{3,1} & P_{3,2} & P_{3,3} & P_{3,4} \\ P_{4,1} & P_{4,2} & P_{4,3} & P_{4,4} \end{bmatrix} \begin{bmatrix} V_{1,1} & V_{1,2} & V_{1,3} & V_{1,4} \\ V_{2,1} & V_{2,2} & V_{2,3} & V_{2,4} \\ V_{3,1} & V_{3,2} & V_{3,3} & V_{3,4} \\ V_{4,1} & V_{4,2} & V_{4,3} & V_{4,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

El punto v' está en coordenadas del plano de proyección, para mapearlo a las coordenadas de la ventana W es necesario multiplicar el punto v' por los valores del puerto de visión (viewport), en la siguiente expresión W es el vector que contiene los valores del puerto de visión y el resultado son las coordenadas de la ventana del punto en cuestión:

$$\begin{bmatrix} v''_{1,1} \\ v''_{2,1} \\ v''_{3,1} \end{bmatrix} = \begin{bmatrix} W_0 + W_2 \times v'_0 \\ W_1 + W_3 \times v'_1 \\ v'_2 \end{bmatrix}$$

Este proceso es facilitado por la función `gluProject` de `GLU`, a la cual se pasa como parámetro un puntero a las matriz de proyección y a la matriz de los parámetros de la cámara, un puntero al vector de valores del puerto de visión, los valores del punto que se desea mapear y finalmente un puntero a valores `double` donde se almacenará el resultado, el prototipo de de la función `gluProject` es:

```
1 GLint gluProject( GLdouble      objX,
2   GLdouble      objY,
3   GLdouble      objZ,
4   const GLdouble *      model,
5   const GLdouble *      proj,
```

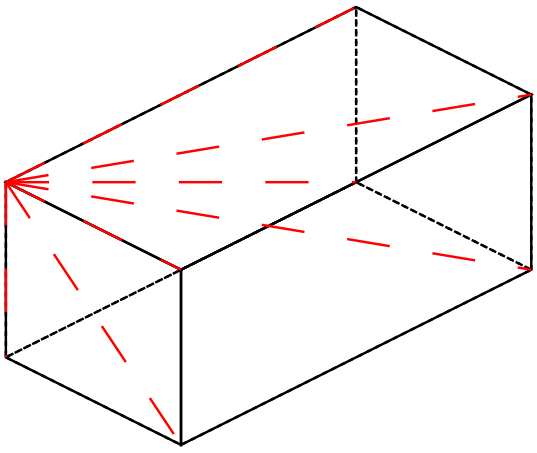


Figura 5.7: De una caja envolvente mapeada a coordenadas de la ventana se obtiene la longitud de las diagonales existente entre los vértices proyectados, si la diagonal mayor mide más de 7 pixeles el objeto es mandado a la máquina de estados de OpenGL. En la figura se muestran las siete posibles diagonales que puede tener un mismo vértice.

```
6    const GLint *    view,
7    GLdouble*    winX,
8    GLdouble*    winY,
9    GLdouble*    winZ);
```

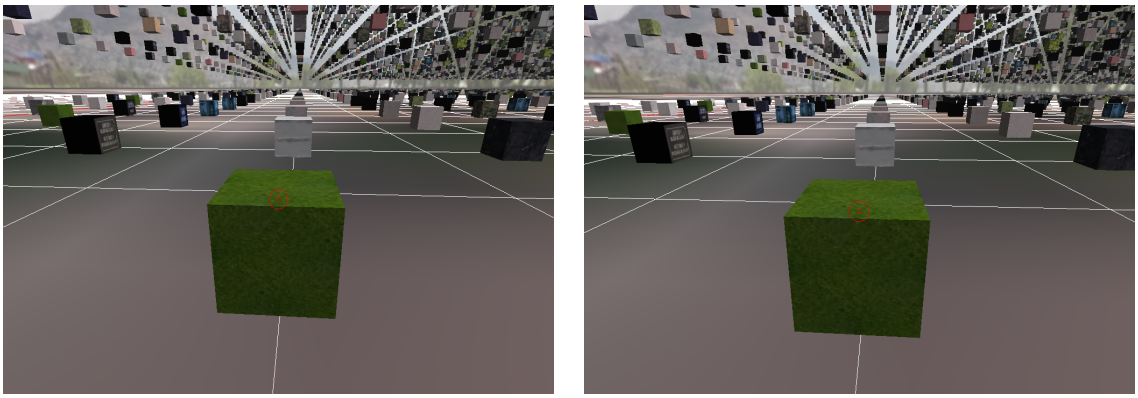
donde:

objX, objY y objZ especifican las coordenadas del punto.
model, especifica la actual matriz con los parámetros de la cámara.
proj, especifica la actual matriz de proyección
view, especifica los valores del puerto de visión.
winX, winY y winZ tendrán las coordenadas de la ventana donde se mapeará el punto.

Esta función es aplicada a los 8 vértices de la caja envolvente y son guardados para el proceso posterior.

Una vez que ya tenemos la posición de los ocho vértices mapeados en las coordenadas de la ventana determinamos la distancia existente entre un punto y los otros 7 puntos mapeados guardando únicamente la distancia mayor, las unidades de esta distancia son pixeles. Repetimos este proceso para todos los puntos mapeados siempre guardando la distancia mayor. Una caja envolvente con sus posibles diagonales se muestra en la figura 5.7.

Si la distancia obtenida, que representa a la diagonal de mayor longitud, es menor



(a) (b)

Figura 5.8: Los objetos diminutos pueden aportar poco o nada a una escena debido a su tamaño. En (a) está una escena con todos los objetos posibles, en (b) está la misma escena quitando los objetos diminutos (ver el fondo de la escena), con esto se reducen los tiempos de presentación de cada cuadro.

a 7 pixeles se considera al objeto como **diminuto** siendo descartable, por lo tanto, no es mandado a la máquina de estados de OpenGL. Un ejemplo del mundo virtual sin quitar los objetos descartables y un ejemplo de la misma escena quitando los objetos si descartables se puede apreciar en la figura 5.8.

Como se podrá observar en la figura 5.8, ambas imágenes parecen idénticas pero si se realiza una inspección más detalla al fondo de la figura 5.8(b) se apreciará que algunos cubos del fondo han desaparecido. La ausencia de estos objetos no es apreciada en la primera observación ya que que la vista se centra en el objeto más grande que se encuentra en primer plano.

Con esta implementación se ofrecen tiempos de presentación mejores a los anteriores, los tiempos reportados en la figura 5.9 son de la optimización de eliminación de objetos diminutos y la optimización de los mapas MIP. Con esto se dota al proyecto de la capacidad de ofrecer más polígonos por escena si el mundo virtual lo necesitara. realizando una rápida interpretación de los datos de la figura 5.9, al comienzo de la creación de un mundo virtual los tiempos de la eliminación de objetos diminutos son superiores al descartamiento mediante frustum, pero conforme se agregan objetos a la escena los tiempos de la eliminación de objetos diminutos superarán en rendimiento al descartamiento por frustum.

Se implementó otra optimización que busca eliminar únicamente los objetos lejanos diminutos (la anterior elimina los objetos diminutos independientemente de su posición), para esta optimización se requiere un preprocesamiento de los objetos, ya que se busca eliminar a los objetos más pequeños que sean lejanos de la cámara, el

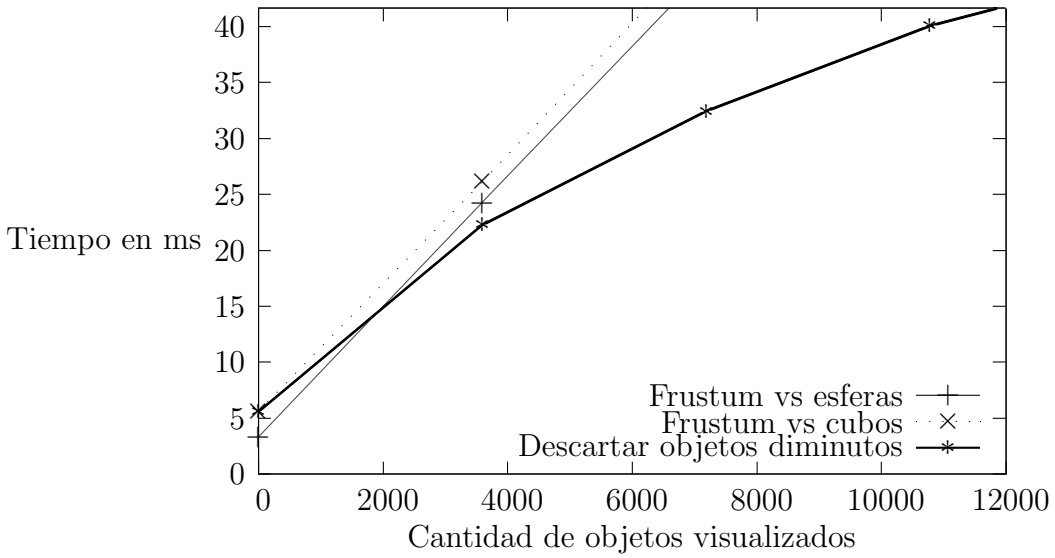


Figura 5.9: Los tiempos que arroja la optimización de descartamiento de objetos diminutos y la optimización de los mapas MIP.

preprocesamiento consiste en guardar en una nueva lista ligada ordenada los objetos según sus distancias a la cámara, i.e., el más cercano estará al comienzo de la lista mientras que el más lejano estará al final de la lista, este paso se realiza al momento de verificar si algún objeto se encuentra dentro del frustum, en caso de que si lo esté se guarda en la lista ligada ordenada, los objetos dentro de esta lista son los objetos candidatos a ser pintados.

Una vez que todos los objetos que se encuentran dentro del frustum han sido puestos dentro de la lista ordenada se comienza a recorrer de forma inversa dicha lista, esto es porque se espera que un objeto lejano a la cámara sea diminuto, comprobando si un objeto mide menos de 7 pixeles en su diagonal de mayor longitud, en caso de que así sea la nueva posición del plano lejano del frustum será la posición de dicho modelo, en caso contrario, termina el ciclo y se mandan a pintar el resto de los objetos en la lista. De esta manera se asegura que el plano lejano del frustum está por detrás de todos los objetos mayores a 7 pixeles y éstos serán mandados a pintar.

Hasta este momento se han pintado todos los objetos de la escena pero la cúpula celeste es el objeto mayor en la escena y es el más lejano de todos, con el cambio del plano lejano al mandarla a pintar no estaría dentro del frustum y el paso de recorte (*cliping*) de OpenGL no lo mostraría en pantalla, por lo tanto es necesario volver a cambiar el plano lejano del frustum a su posición original para que sea pintada, ademas este paso sirve como base para el siguiente ciclo de pintado, los planos del frustum deben de tomar su posición original debido a la optimización de descartamiento mediante el frustum.

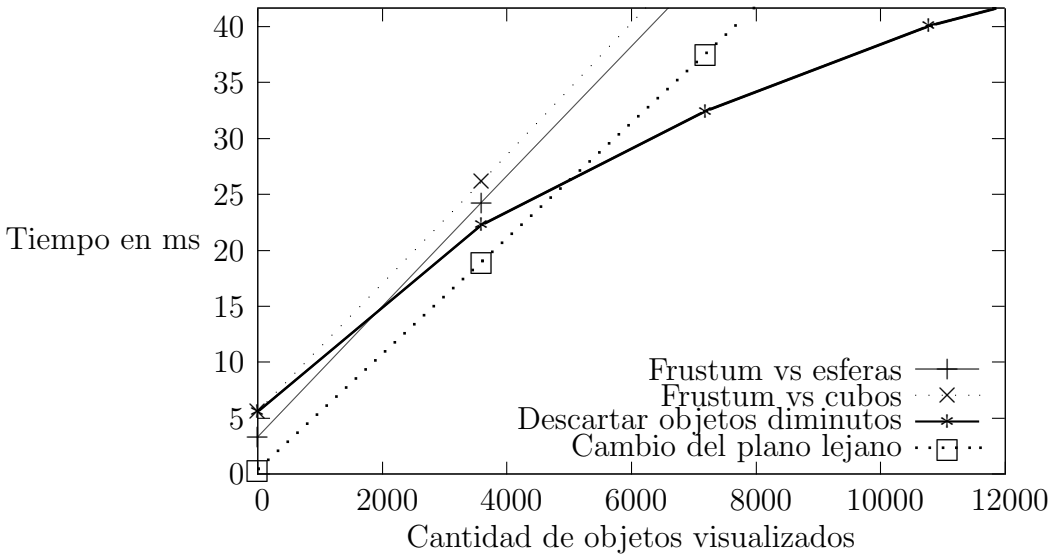
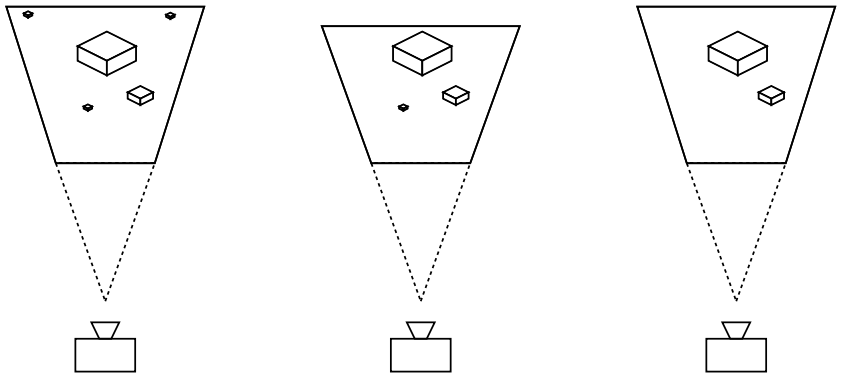


Figura 5.10: Los tiempos de ejecución que arrojan las técnicas de cambio del plano lejano del frustum, el descartamiento de objetos diminutos y el descartamiento mediante frustum contra cajas y esferas.

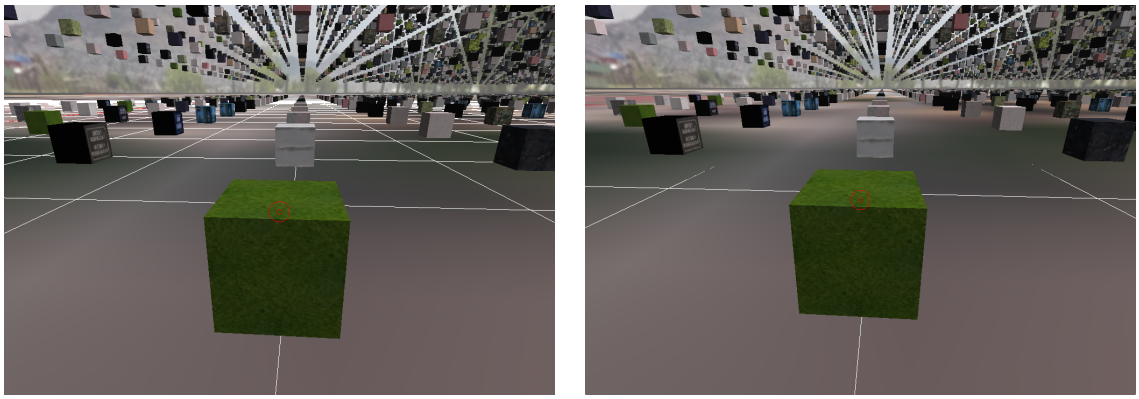
Los tiempos de presentación en el mundo virtual virtual de prueba se reportan en la figura 5.10, donde se puede apreciar que la técnica de cambio lejano del frustum ofrece ventaja de rendimiento cuando los mundos virtuales son relativamente pequeños, sin embargo, conforme van creciendo en cantidad de objetos es la optimización de descartamiento de objetos diminutos la que toma ventaja, esto es debido a que la nueva posición del plano lejano del frustum tiene como limite la posición del objeto **no** diminuto más lejano de la cámara permitiendo que algunos objetos diminutos estén dentro del frustum y sean pintados, por otro lado, la optimización de descartamiento de objetos diminutos elimina dichos objetos sin importar su posición dentro del frustum, un ejemplo más claro de esto se muestra en la figura 5.11, en la figura 5.11(a) se encuentra una representación de la escena original, en la figura 5.11(b) se muestra la eliminación de los objetos al cambiar el plano lejano del frustum y en la figura 5.11(c) se muestra la eliminación de todos los objetos diminutos.

Para una comparación del resultado se muestra en la figura 5.12 la misma escena de la figura 5.8 de la página 68, del lado izquierdo está la imagen correspondiente al descartamiento de objetos diminutos mientras que en la derecha se encuentra la imagen correspondiente al cambio del plano lejano del frustum.



(a) (b) (c)

Figura 5.11: Las distintas optimizaciones planteadas tienen ventajas y desventajas. En (a) se muestra la escena original mostrando todos sus objetos, tanto diminutos como los que no lo son. En (b) se encuentra una representación de la misma escena, los objetos diminutos que están dentro del frustum serán pintados ya que el cambio del plano lejano se limita al objeto no diminuto más lejano de la cámara. En (c) se quitan todos los objetos diminutos sin importar su posición conforme la cámara.



(a) (b)

Figura 5.12: Los resultados de ambas técnicas son similares en cuanto a presentación, sin embargo, los tiempos de presentación son distintos, en (a) se encuentra la imagen correspondiente al descartamiento de objetos diminutos. En (b) se encuentra la imagen correspondiente al cambio del plano lejano del frustum.

Capítulo 6

Conclusiones

En la realización de este trabajo se implementó un software para la construcción y navegación de mundos virtuales para el sistema operativo Mac OS que consiste de las siguientes partes:

- Un repositorio de modelos y texturas ocupando PostgreSQL como servidor de base de datos y C para la conexión al repositorio. Este repositorio centraliza los modelos y texturas creados evitando la duplicidad de información en distintas computadoras permitiendo a varios clientes acceder a los modelos al mismo tiempo.

- Una barra emisora de luz infrarroja que se conecta por medio de USB y una biblioteca hecha en C/Obj-C usando la API de Mac OS, para la comunicación con el Wiimote que fungió como dispositivo de entrada al cual se le mapearon las funciones necesarias para la navegación y construcción de un mundo virtual. El uso de este dispositivo de seis grados de libertad proporciona una forma amigable de navegación del mundo virtual y una intuitiva edición de los objetos proporcionando la interacción del usuario con el software cuando se atienden los eventos que crea el control.

- Una interfaz gráfica construida con QT y OpenGL, esta interfaz gráfica permite al usuario observar los objetos que están dentro del mundo virtual así como la adición, elección, eliminación, clonación, texturizado y transformación de dicho objetos para su construcción, así como también, da la capacidad de guardar los mundos virtuales para una posterior recuperación y edición. Esta interfaz incluye colisiones entre la cámara y objetos impidiendo que la cámara traspase los objetos, evitando así, dar una retroalimentación errónea al usuario. La interfaz gráfica permite desplazarse a lo largo y ancho del plano xz dando la sensación de 'caminar' en el mundo virtual, mediante el uso del Wiimote se puede

rotar el ángulo de visión para 'voltear' a ver cualquier punto en el mundo virtual.

- Se implementaron y compararon cinco técnicas para la optimización de los tiempos de dibujado de los objetos que están en el mundo virtual consiguiendo tiempos de procesado bastante eficientes, estas técnicas fueron: (1) el uso de nivel de detalle en las texturas (mapas MIP), el descartar objetos mediante el chequeo de la colisión del frustum en dos versiones distintas: (2) colisión contra cajas y (3) colisión contra esferas, y por último, dos formas distintas de eliminar los objetos diminutos que aportan poco o nada a la escena, (4) eliminar el objeto según su tamaño relativo en el plano de proyección y (5) eliminar el objeto mediante el cambio de posición del plano lejano del frustum. La técnica más eficiente, que permite visualizar hasta 12,000 objetos (48,000 polígonos) en cada marco a 25 marcos por segundo, fue la (4), la técnica de eliminar objetos según su tamaño relativo en el plano de proyección.

  Todos estos algoritmos presentan un grado de eficiencia bueno comparándolo contra la versión que no tiene algún tipo de optimización, siendo el más complejo de implementar el del cambio del plano lejano del frustum y el más sencillo de implementar el descartamiento mediante frustum usando las esferas envolventes.

Esto resultó en un software para la construcción de mundos virtuales relativamente grandes que bien puede servir para el prototipado de videojuegos, maquetas virtuales, edificios, etc.

Para probar el uso del prototipo se construyó un Cinvestav Zacatenco virtual, siguiendo un proceso iterativo e incremental, el proceso fue el siguiente: estudiar el mundo real que se va a construir en el virtual, para esto se auxilió de mapas, toma de medidas, toma de fotografías de distintos ángulos y toma de videos; si el área es muy grande se puede dividir en segmentos de construcción que fue como se realizó esta prueba. Posteriormente se toman fotografías de las superficies de los elementos de una sección de las que se incluirán en el mundo virtual, tales como: pisos, letreros, paredes de edificios, estructuras, etc., para después editar las fotografías recortándolas, redimensionandolas y guardándolas en formato bmp de 24 bits; se agregan las texturas al repositorio mediante la interfaz gráfica. El siguiente paso es la construcción de la sección del mundo virtual. Apoyándose en los mapas, videos e imágenes se construye con el Wiimote añadiendo y editando los modelos preincluidos en el repositorio de modelos, guardando el mundo virtual cada vez que se realiza un avance significativo. Este proceso se repite para cada sección del mundo virtual recuperando y editando la copia del mundo virtual guardada previamente. El tiempo en el que se concluyó el Cinvestav Zacatenco virtual fue aproximadamente de un mes, resultando en un mundo virtual con cerca de 1,000 objetos añadidos, transformados y texturizados.

6.1. Trabajo a futuro

Los contextos de trabajo y la forma de navegación pueden ser adecuada a otros dispositivos que tengas seis grados de libertad, ya que las funciones y contextos del proyecto son independientes de la entrada se pueden crear bibliotecas para algún otro tipo de dispositivo de entrada, o inclusive, el Kinect de Microsoft para crear el mundo virtual mediante gestos más naturales.

Los arboles octales (*octree*) es una técnica que divide el mundo virtual en ocho volúmenes (cajas) del mismo tamaño, estos son volúmenes son llamados nodos. Si alguno de estos nodos intersercta con algún plano del frustum el nodo será divido en ocho nuevos subnodos, el proceso se puede repetir hasta la granularidad deseada. De esta manera sólo los objetos que se encuentran dentro de los nodos que intersectan y están dentro del frustum serán candidatos a ser pintados, en otras palabras, un nodo de un árbol octal se puede considerar como un super volumen envolvente de objetos. Implementar, probar y trabajar esta técnica, en contra y en conjunto del descartamiento mediante frustum, puede converger en una nueva optimización.

Encontrar alguna estructura de datos, arboles octales o árboles binarios, para ordenar los objetos dentro de la escena y probar las técnicas propuestas en este trabajo solo en los objetos que se encuentran en los bordes del frustum.

Respecto al realismo, los mundos virtuales tienen como objetivo ser una representación del mundo real, por esto siempre existirán detalles que se pueden mejorar o técnicas que se pueden implementar para obtener un mayor grado de realismo, por ejemplo, un sistema para el preprocesado de mapeado de sombras (shadow mapping) para obtener sombras proyectadas en los objetos según la posición de una fuente de luz. Agregar la capacidad para el mapeado topológico (bump mapping) que proporciona relieve a la superficie de los objetos sin modificar su geometría.

Apéndice A

Prototipos de las funciones usadas

`PQsetdbLogin` - Realiza una nueva conexión a un servidor de bases de datos.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

`pghost` - Nombre del host a conectarse. En las máquinas sin sockets de dominio UNIX, el valor predeterminado es el localhost.

`pgport` - Número del puerto para conectarse al servidor, o el nombre del socket para conexiones en dominios Unix.

`pgoption` - Opciones de linea de comando que será enviadas al servidor.

`pgtty` - Especifica a donde será enviada la salida de depuración del servidor.

`dbName` - El nombre de la base de datos. Por defecto es la misma que el nombre del usuario.

`login` - Nombre de usuario PostgreSQL para la conexión. Por defecto es el mismo que el nombre del sistema operativo donde se está corriendo la aplicación.

`pwd` - La contraseña que se usará en caso de ser necesaria.

Ejemplo:

```
PGconn *conexion = PQsetdbLogin( "localhost",
```

```
                "5432",
                NULL,
                NULL,
                "repositorio",
                "postgres",
                "postgres");
```

`PQstatus` - Verifica el estado de conexión de un objeto PGconn.

```
ConnStatusType PQstatus(const PGconn *conn);
```

`conn` - Es un objeto PGconn obtenido mediante la función PQsetdbLogin.

Ejemplo:

```
/* Verificamos que se haya logrado la conexion. */
if ( PQstatus(conexion) != CONNECTION_OK )
{
  printf( "No se logro el enlace, %s\n", PQerrorMessage(conexion) );
  PQfinish( conexion );
}
```

`PQfinish` - Cierra la conexión realizada en el objeto PGconn pasado como argumento y libera la memoria de dicho objeto.

```
void PQfinish(PGconn *conn)
```

`PQerrorMessage` - Regresa el error más recientemente producido por una operación en la conexión.

```
char *PQerrorMessage(PGconn* conn)
```

`PqexecParams` - Manda un comando al servidor y espera por la respuesta, con la capacidad de poder mandar parámetros.

```
PGresult *PQexecParams(PGconn *conexion,
                       const char *comando,
                       int nParams,
                       const Oid *tipoParam,
```

```
                        const char * const *valoresParam,
                        const int *longitudParam,
                        const int *formatoParam,
                        int formatoResultado);
```

`conn` - El objeto de conexión por donde se enviará el comando.

`Command` - Una cadena donde se encuentra el comando SQL que se ejecutará, Si lo parámetros son usados, serán referidos en la cadena como $1, $2, $3, etc.

`nParams` - EL número de parámetros que serán enviados, este es el tamaño de los arreglos paramTypes[], paramValues[], paramLengths[], y paramFormats[]. Los arreglos pueden apuntar a NULL cuando el tamaño es cero.

`ParamTypes[]` - Especifica los tipos de datos que se asignará a los símbolos de los parámetros. Si paramTypes es NULL, o cualquier elemento particular de la matriz es cero, el servidor deduce el tipo de dato para el símbolo de parámetro.

`ParamValues[]` - Guarda el valor de los parámetros. Un valor NULL en el arreglo significa que el parñametro en cuestión no existe.

`paramLengths[]` - Especifica la longitud del parámetro.

`paramFormats[]` - Especifica el formato del parámetro, un cero significa que el parámetro correspondiente es de tipo texto y un uno significa que es de tipo binario.

Al almacenar los datos se requiere saber que tipo de representación usa el servidor, big endian o little endian ya que esto influye al momento de recuperar los datos.

`resultFormat` - Especifica la forma de obtener resultados, un cero significa que será en forma de texto, un uno significa que será en forma binaria.

Ejemplo:

```
PQresult *resultado = PQexecParams(conexion,
            "INSERT INTO nombreTabla ( nombre ) VALUES ($1)",
            1,
            NULL,
            param_vals,
            param_lens,
            param_fmts,
            0);
```

`PQresultStatus` - Regresa el estado de un resultado creado a partir de una consulta.

```
1 ExecStatusType PQresultStatus(const PGresult *res);
```

Puede regresar algúno de los siguientes valores.
`PGRES_EMPTY_QUERY` La cadena enviada al servidor estaba vacía.

`PGRES_COMMAND_OK` Consulta concluida con éxito y no hubo datos de regreso.

`PGRES_TUPLES_OK` Consulta concluida con éxito y sí hubo datos de regreso.

`PGRES_COPY_OUT` Copiado externo desde el servidor, la transferencia de datos ha iniciado.

`PGRES_COPY_IN` Copiado interno en el servidor, la transferencia de datos ha iniciado.

`PGRES_BAD_RESPONSE` La respuesta del servidor no fue comprendida.

`PGRES_NONFATAL_ERROR` Un error no fatal, una advertencia o un aviso ocurrió.

`PGRES_FATAL_ERROR` Un error fatal ocurrió.

`PQclear` - Libera el espacio en memoria asociado a un objeto PGresult, cada resultado debe de tener un PQclear, de lo contrario resultará en perdidas de memoria.

```
1 void PQclear(PQresult *res);
```

`PQntuples` Regresa el número de columnas (tuplas) del resultado de la consulta.

```
1 int PQntuples(const PGresult *res);
```

`PQnfields` Regresa el número de columnas (campos) de cada columna del resultado de la consulta.

```
1 int PQnfields(const PGresult *res);
```

Apéndice B

Comandos de entrada y salida del Wiimote

Los comando del Wiimote tienen distintos parámetros que son enviados como un paquete de bytes al control o recibidos desde éste. En este apéndice se incluye una explicación a fondo de lo que hace cada comando y cómo se usan.

B.1. Comandos de salida

`0x11` - Los diodos emisores de luz.

Cuando el control está en modo detectable, los cuatro LEDs que se encuentran en la parte frontal parpadean de forma automática pero cuando se realiza el enlace son controlables de forma independiente por el anfitrión, y se puede configurar para mostrar cualquier patrón.

El comando `0x11` usa un byte como parámetro donde los 4 bits más altos son los que controlarán que LED estará encendido, mientras que los 4 bits más bajos no tienen uso alguno como parámetro de este comando.

Ejemplo, si se envía 0x11 0x50 (1010 en binario) se prenderán el primer y el tercer LED.

`0x12` - Reporte de datos

El Wiimote tiene distintos formatos para el reporte de datos y estos se pueden especificar por medio de este comando de salida. Usa dos bytes como parámetros, el primer byte se especificará si el envío de reportes será de forma continua o si sólo será un único reporte, el bit 2 de dicho bit deberá de estar en uno si es que se desea

que sea de forma continua. El segundo byte especifica el formato con el que serán reportados los datos, este parámetro puede tomar el valor desde 0x33 hasta 0x3f, que son los formatos admitidos y son explicados más adelante en la sección de comandos de entrada.

Ejemplo, `0x12 0x04 0x33`, al estar encendido en bit 2 del primer parámetro indica que el envío de reportes será de forma continua y será en el formato que `0x33` especifica, reportando únicamente el estado de los botones.

`0x13` - Activación o desactivación de la cámara infrarroja.

Este comando activa o desactiva el uso cámara infrarroja dependiendo del comando que le sea pasado, usa un byte como parámetro donde el bit 2 es el que indicará la activación, si el bit está en uno activará la cámara, si está en cero la desactivará.

Ejemplo, si se envía `0x13 0x04`, en cambio si envía `0x13 0x00` la desactivará.

`0x15` - Solicitud de estado de información

Este comando solicita un único envío de información de estado por el canal de entrada abierto durante el apretón de manos, usa un byte como parámetro donde el bit 2 es el que indicará la solicitud del estado.

Ejemplo, si se envía `0x15 0x04` realizará la petición.

`0x16` - Escritura de memoria.
`0x17` - Solicitud de lectura de memoria.
`0x21` - Envío del contenido de la memoria.

El mando de Wii incluye una EEPROM integrada, parte de la cual es accesible para el usuario para almacenar avances del juego, opciones de juegos y otras características. Estos comandos no son usados en este proyecto.

`0x14` - Activación o desactivación de la bocina.
`0x18` - Envío de datos a la bocina.
`0x19` - Silenciado de la bocina.

La bocina que tiene el Wiimote puede ser activada o desactiva, enviarle datos para que reproduzca algún sonido o silenciarla. Estos comandos no fueron usados en este proyecto.

B.2. Comandos de entrada

`0x20` - Información de estado.

Este es el informe de estado que se envía de forma discreta si así es requerido por medio del comando `0x15`, su prototipo es `0x20 BB BB LF 00 00 VV` donde:

BB BB son bytes que incluyen información de los botones presionados.
L Son 4 bits usados para reportar el estado de los LEDs.
F Son 4 bits usados como banderas para reportar el estado de la cámara, de la bocina, si existe alguna extensión conectada y si la batería está pronta a terminarse.
VV es un byte usado para reportar el nivel de energía de la batería.

`0x22` - Acuse de recibo.

Esta comando de entrada es enviado al anfitrión para informar de un error relacionado con un comando de salida, o del resultado de una comando de salida. Se envía cuando el bit 1 del primer byte de cualquier informe de salida está puesto en 1.

Este acuse de recibo siempre incluye información acerca del estado de los botones del control.

Un prototipo de entrada es `0x22 BB BB RR EE`, donde:
BB BB son 2 bytes que incluyen información sobre el estado de los botones del control.
RR es el comando de salida del cual se está reportando el acuse de recibo.
EE es el código del error o el resultado del comando. `0x00` para éxito, `0x03` para el error y `0x04` para desconocido.

`0x30-0x3f` - Reporte de datos.

Estos son una familia de comandos que dependiendo de la configuración por medio del comando de salida `0x12` puede reportar el estado de los botones o del acelerómetro o el estado de la cámara infrarroja.

`0x30` - Estado de los botones.

Su prototipo es `0x30 BB BB`, donde:
BB BB son dos bytes que reportan el estado de los botones del control.

`0x31` - Estado de los botones y del acelerómetro. Su prototipo es `0x31 BB BB AA AA AA`, donde:
BB BB son dos bytes que reportan el estado de los botones del control.
AA AA AA son tres bytes que reportan el estado del acelerómetro.

`0x32` - Botones del control y botones de la extensión en 8 bytes.

Su prototipo es `0x32 BB BB EE EE EE EE EE EE EE EE`

donde:

BB BB son dos bytes que reportan el estado de los botones del control.
Los bytes EE son 8 bytes que reportan el estado de los botones de alguna extensión sí está conectada.

`0x33` - Botones del control, acelerómetro y cámara infrarroja.

Su prototipo es `0x33 BB BB AA AA AA II II II II II II II II II II II II`, donde:

BB BB son dos bytes que reportan el estado de los botones del control.
AA AA AA son tres bytes que reportan el estado del acelerómetro.
Los bytes II son 12 bytes que reportan el estado de la cámara infrarroja.

Del comando `0x34` al `0x3f` son comandos para reportar el estado de la extensiones conectadas al control, se omite su prototipo ya que no se ocuparon en este proyecto.

Apéndice C

Capturas de pantalla mostrando los resultados

En este apéndice se muestran algunas capturas de pantalla del software terminado presentando algunas de sus capacidades al construir un mundo virtual completo, en este caso se construyó una representación del Cinvestav Zacatenco.

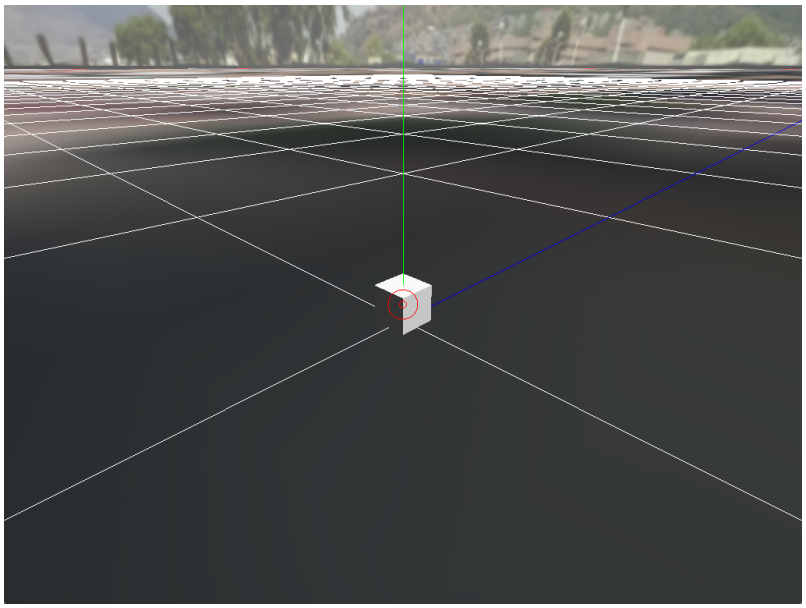


Figura C.1: El inicio de un mundo virtual consiste en un único cubo en el escenario.

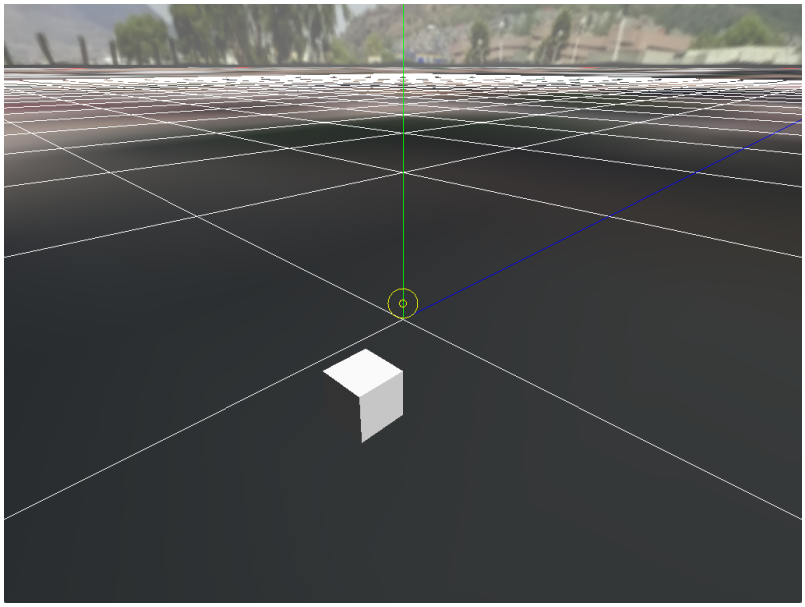


Figura C.2: Al estar en el contexto de traslación el objeto puede ser desplazado en el espacio usando los botones de dirección del Wiimote.

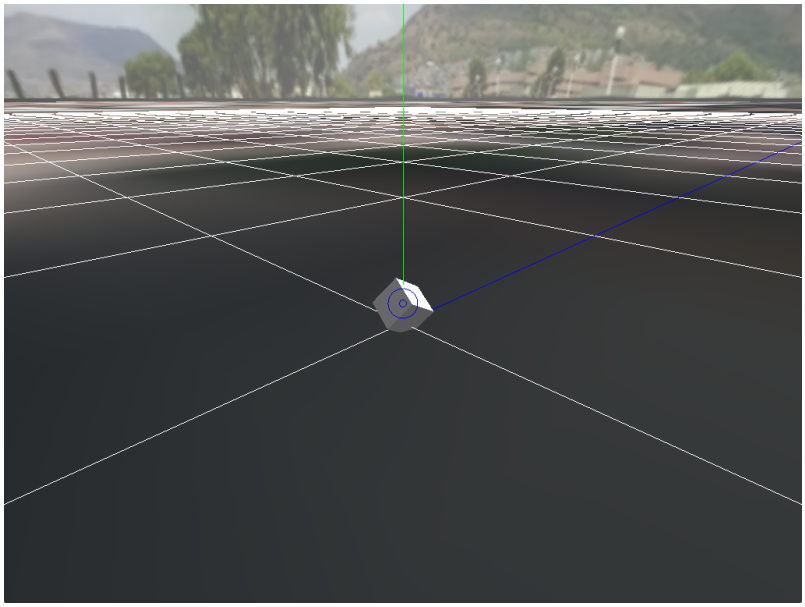


Figura C.3: Al estar en el contexto de rotación el objeto puede ser rotado en el espacio usando los botones de dirección del Wiimote.

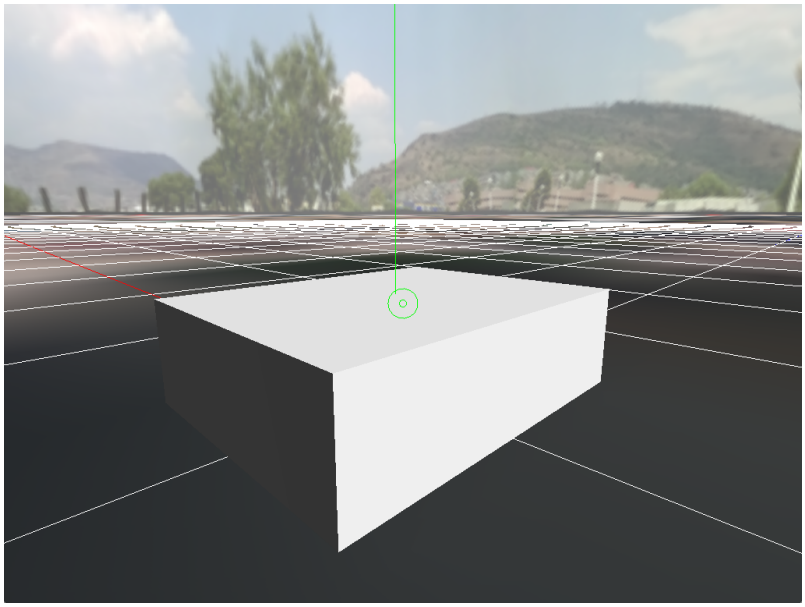


Figura C.4: Al estar en el contexto de escalamiento el objeto puede ser dimensionado en el espacio usando los botones de dirección del Wiimote.

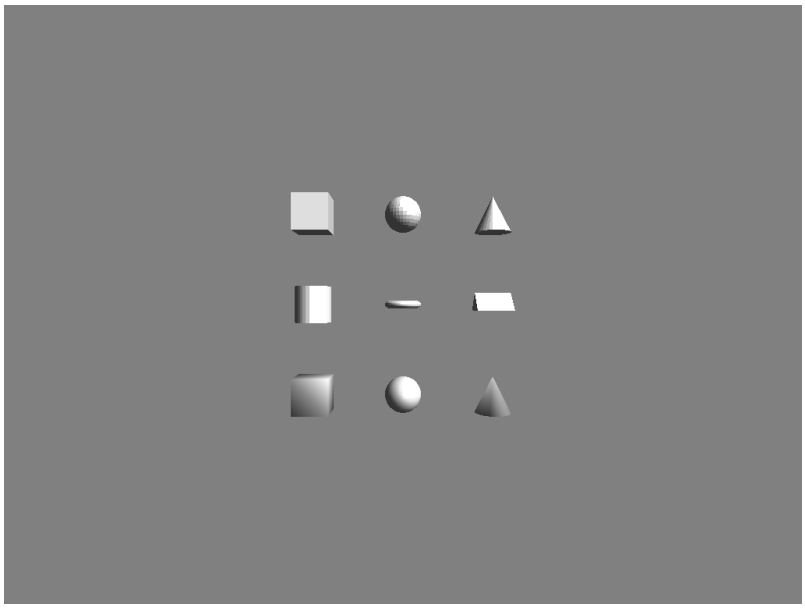
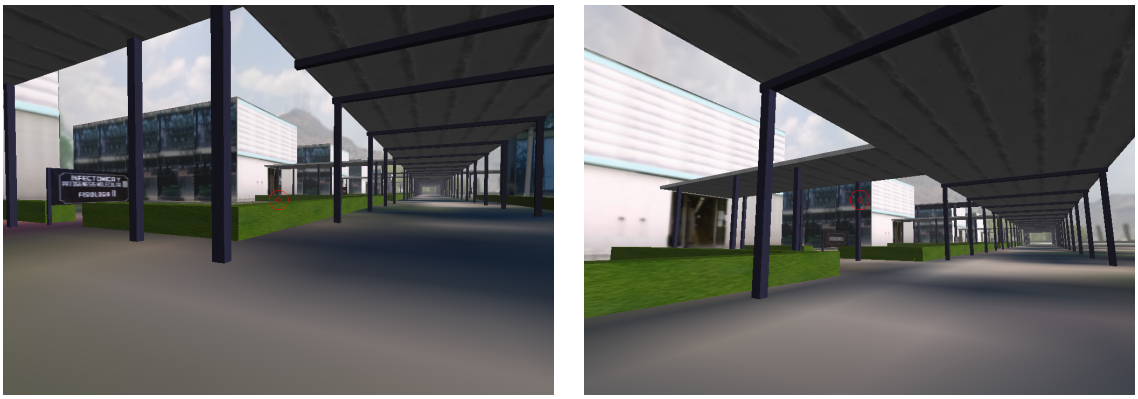


Figura C.5: Se puede añadir distinas formas al mundo virtual para crear distintos tipos de objetos compuestos y complejos.

(a) (b)

Figura C.6: Al texturizar un objeto el menú muestra las texturas sobre el objeto en cuestión que fue seleccionado. En (a) está el menú cómo se muestra al texturizar un cubo mientras que en (b) está el menú al texturizar una esfera.



(a) (b)

Figura C.7: La clonación de objetos es una función bastante útil al crear los mundos virtuales, éstos constan de elementos que se repiten varias veces en una misma escena ahorrando tiempo en la producción.

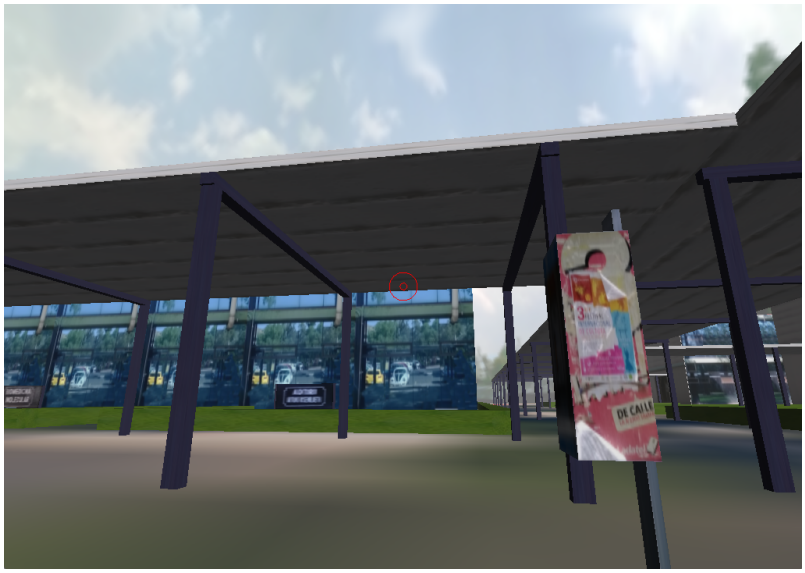


Figura C.8: Se pueden crear escenas bastantes fieles a su símil real, el prototipado de maquetas virtuales es una opción viable para el uso de este proyecto.

Figura C.9: La cúpula celeste celeste agregada al mundo virtual simula un cielo con nubes, añade contexto a la escena y un poco de realismo.

Bibliografía

[1] K. Pulo and M.E. Houle. Evaluation of virtual world systems. In *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pages 98–107, 2001.

[2] Huaiyu Liu, M. Bowman, R. Adams, J. Hurliman, and D. Lake. Scaling virtual worlds: Simulation requirements and challenges. In *Simulation Conference (WSC), Proceedings of the 2010 Winter*, pages 778–790, Dec 2010.

[3] D. Hearn and M. P. Baker. *Gráficas por Computadora*. Prentice Hall Hispanoamericana, 1988.

[4] F. Steinicke, H. Benko, A. Krüger, D. Keefe, J. de la Riviére, K. Anderson, J. Häkkilä, L. Arhippainen, and M. Pakanen. The 3rd dimension of chi (3dchi): Touching and designing 3d user interfaces. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, pages 2695–2698, New York, NY, USA, 2012. ACM.

[5] Superdataresearch european market virtual goods 2010, 2010. http://www.superdataresearch.com/content/uploads/2010/10/EuropeanMarketVirtualGoods101110.jpg.

[6] P. S. Strauss and R. Carey. An object-oriented 3d graphics toolkit. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 341–349, New York, NY, USA, 1992. ACM.

[7] T.M. Takala, P. Rauhamaa, and T. Takala. Survey of 3dui applications and development challenges. In *3D User Interfaces (3DUI), 2012 IEEE Symposium on*, pages 89–96, March 2012.

[8] D.A Bowman, S. Coquillart, B. Froehlich, M. Hirose, Y. Kitamura, K. Kiyokawa, and W. Stuerzlinger. 3d user interfaces: New directions and perspectives. *Computer Graphics and Applications, IEEE*, 28(6):20–36, Nov 2008.

[9] B. branit. world builder. 2009., 2013. http://www.youtube.com/watch?v=VzFpg271sm8.

[10] IEEE 8th symposium on 3d user interfaces. 3dui contest, 2013. http://3dui.org/2013/cfp-3dui-contest.html.

[11] Jia Wang, Owen Leach, and Robert W. Lindeman. Diy world builder: An immersive level-editing system. In *3D User Interfaces (3DUI), 2013 IEEE Symposium on*, pages 195–196, March 2013.

[12] What is unity and what can i do with it?, 2013. http://unity3d.com/pages/create-games.

[13] James D. Murray and William vanRyper. *Encyclopedia of graphics file formats*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1994.

[14] Postgresql, 2014. http://www.postgresql.org/.

[15] Wiibrew, 2014. http://wiibrew.org/wiki/Wiimote.

[16] Christer Ericson. *Real-time collision detection*. Morgan Kaufmann series in interactive 3D technology. Elsevier, Amsterdam, Boston, Paris, 2005.

[17] Jack Ritter. Graphics gems. chapter An Efficient Bounding Sphere, pages 301–303. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[18] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, July 1983.

[19] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 4.3*. Addison-Wesley Professional, 8th edition, 2013.

[20] Ahn SH. Opengl transformation, 2014. http://www.songho.ca/opengl/gl_transform.html.