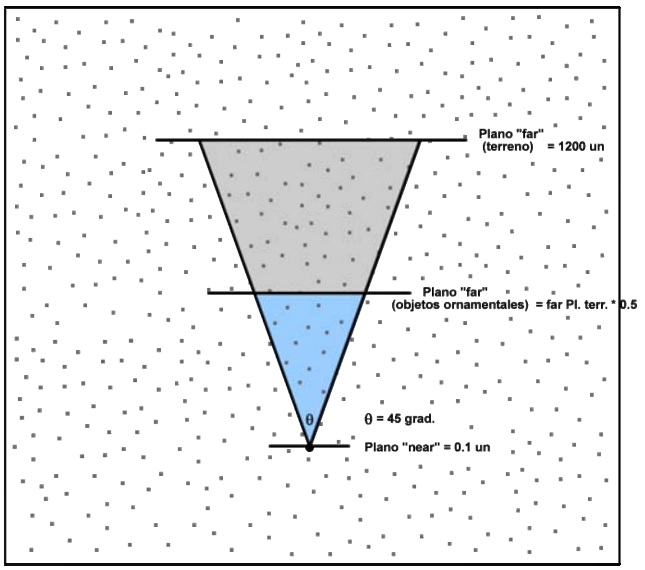
Capítulo 5. Pruebas y conclusiones.

5.1. Pruebas.

Se realizaron tres pruebas de rendimiento en cuatro equipos de cómputo distintos a una resolución de pantalla de 1024x768 y utilizando tres escenas distintas. La primera escena está formada por un terreno cuyas dimensiones sobre el plano cartesiano XZ son de 2000x2000 unidades y consta de 127,608 triángulos. La segunda prueba está formada por el mismo terreno pero además se le han añadido algunos modelos 3D ornamentales que representen la vegetación sobre el terreno con lo cual obtenemos una escena que consta de 451,186 triángulos. La tercera escena es similar a la anterior solo que con una mayor densidad de vegetación obteniendo así una escena que consta de 910,556 triángulos. Los octantes utilizados para almacenar la información de los triángulos de la escena se definieron a 4 niveles de profundidad en el octree, y los octantes utilizados para realizar la detección de colisiones a 8 niveles de profundidad, tanto para el octree utilizado para almacenar la información del terreno como para el octree utilizado para almacenar la información de los objetos ornamentales. El *view frustum* se configuró con los siguientes valores:



Cada prueba consistió en recorrer un circuito trazado mediante esferas rojas durante un periodo de un minuto después de lo cual se obtuvo un archivo con la frecuencia de cuadros en cada segundo transcurrido. A continuación se muestra una captura de pantalla de la ejecución de cada una de estas tres escenas:

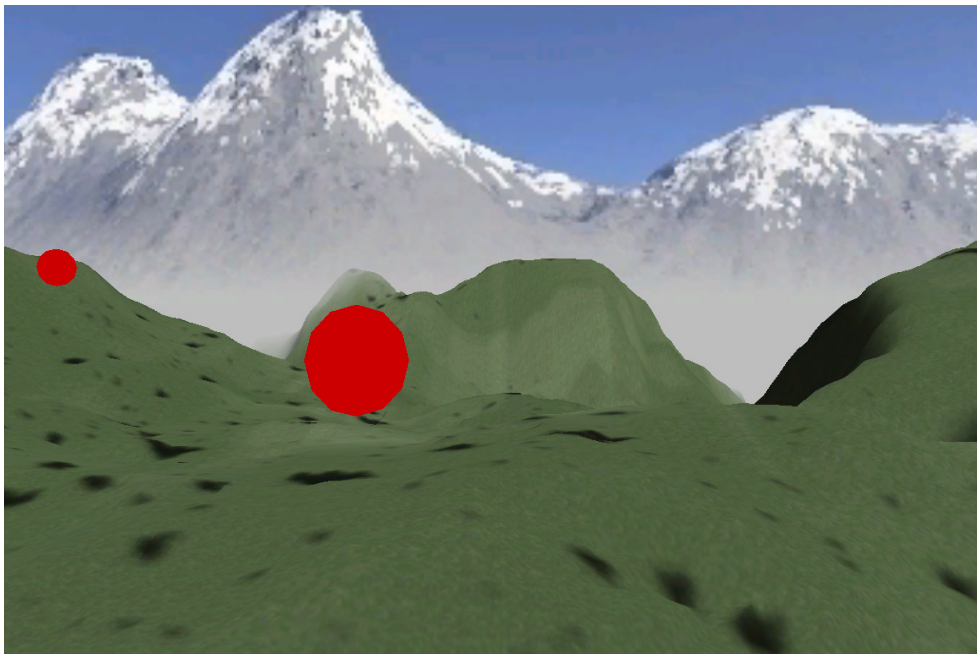


Fig. 5.2.1- Escena de la prueba 1.

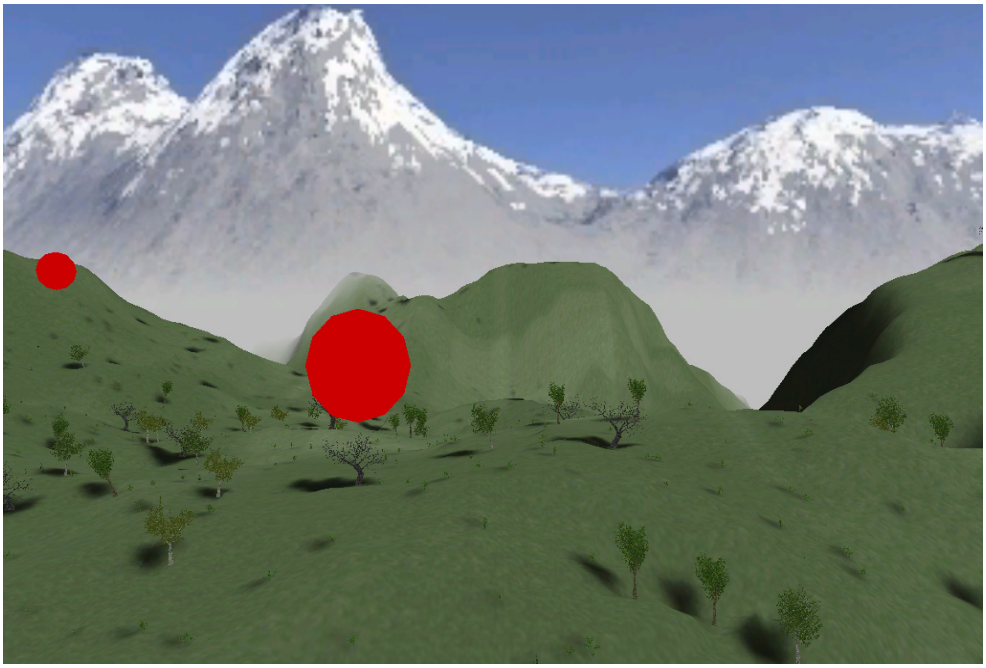


Fig. 5.2.2.- Escena de la prueba 2.

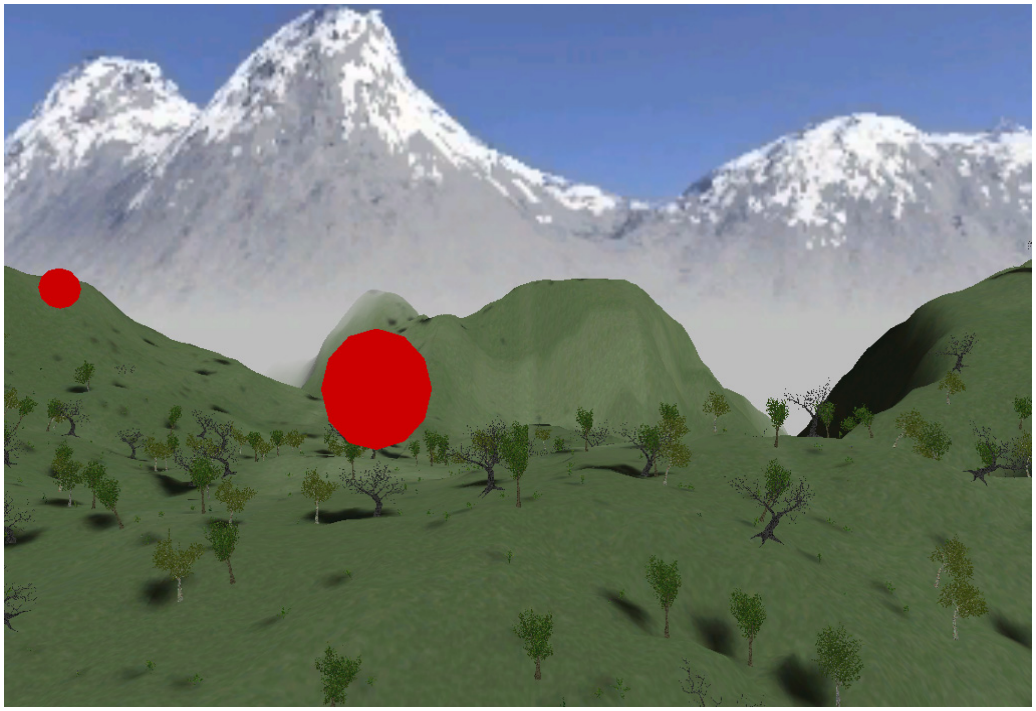


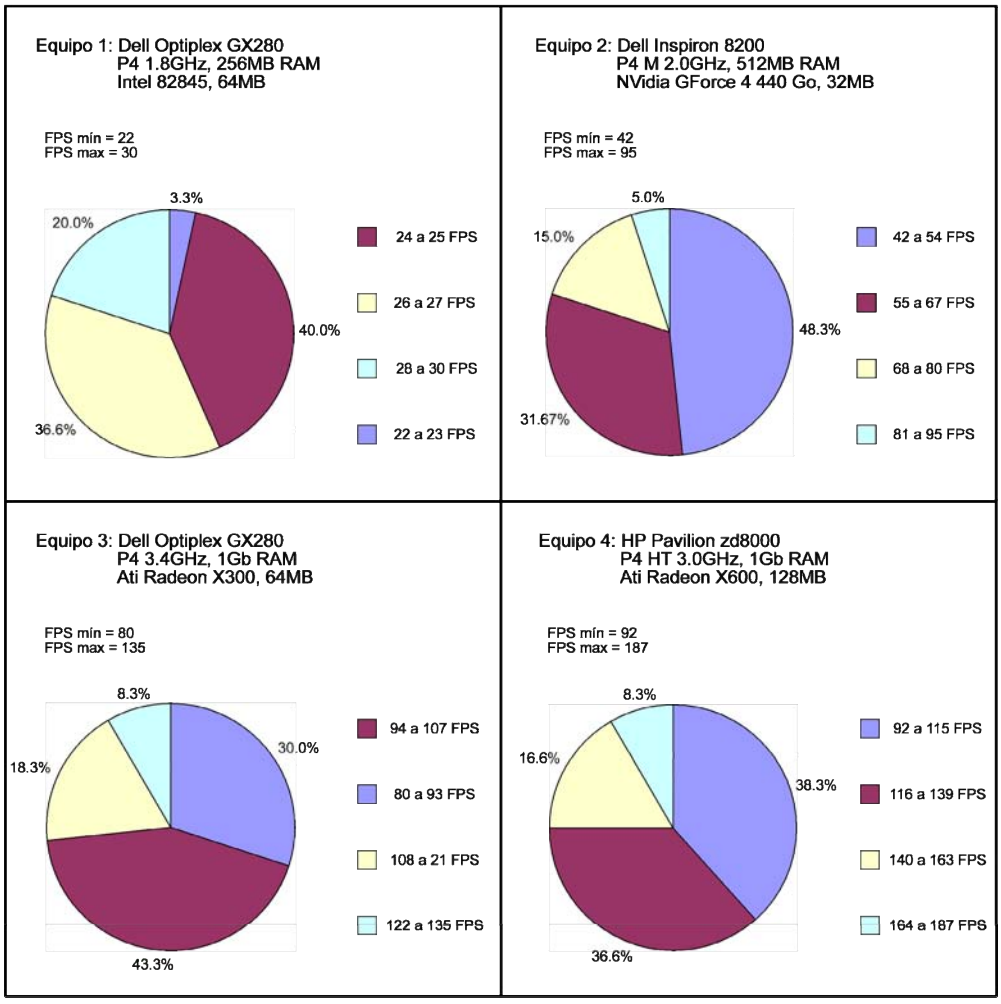
Fig. 5.3.- Escena de la prueba 3.

Los equipos presentados en las pruebas están ordenados de forma ascendente según sus características de hardware en donde el equipo 1 es el que posee las características de hardware que proporcionan el rendimiento más bajo de entre los cuatro equipos, y el equipo 4 es el que posee las características de hardware que proporcionan el rendimiento más alto. Los dos primeros equipos se considera que tienen características inadecuadas(equipo 1) y bajas (equipo 2) para aplicaciones 3D según los estándares de hardware actuales. Los equipos 3 y 4 se considera que tienen características medias-altas según los estándares de hardware actuales. Por otro lado, se ha calificado el rendimiento obtenido según el siguiente cuadro:

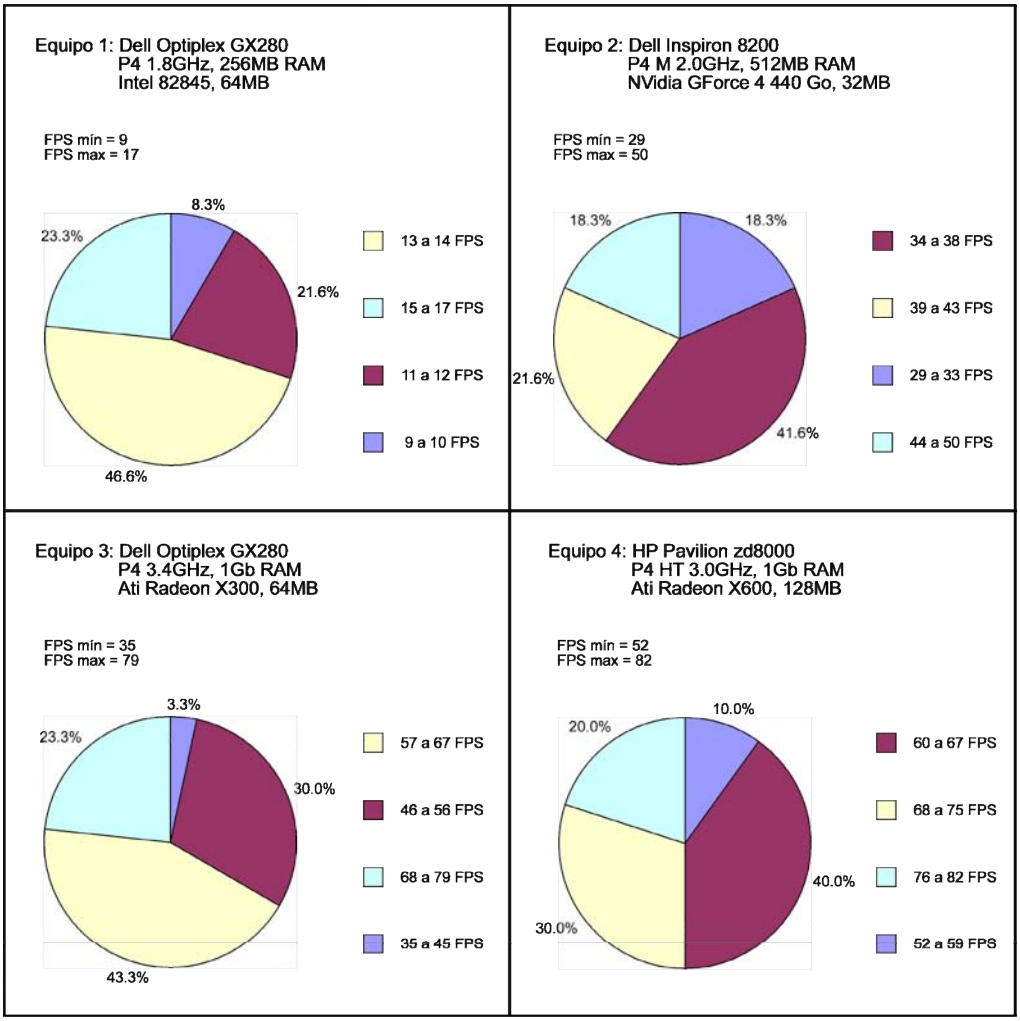
Frecuencia de cuadros por segúndo la mayor parte del tiempo	**Calificación para el rendimiento**
< 15 fps	*pobre*
15 a 30 fps	*bajo*
30 a 45 fps	*aceptable*
45 a 60 fps	*bueno*
> 60	*excelente*

A continuaciones muestran los resultados de las pruebas:

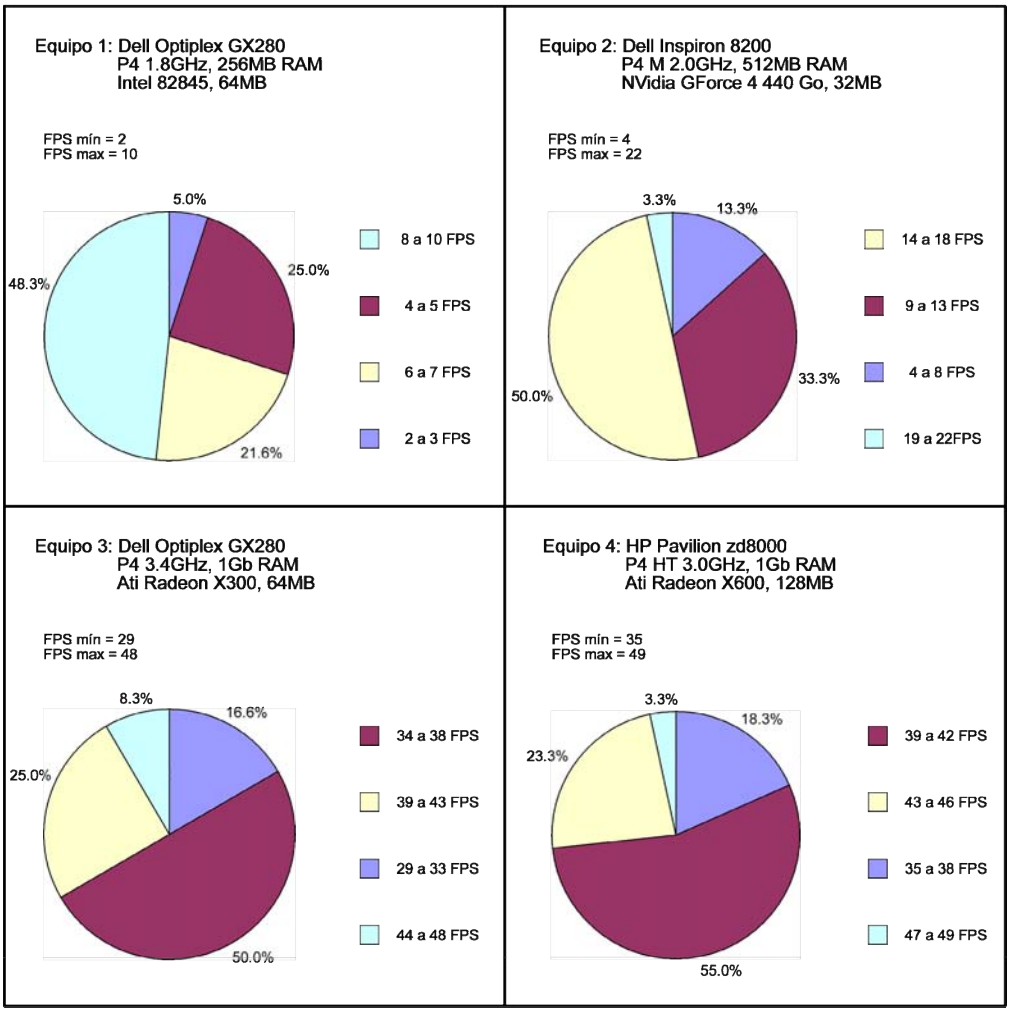
Prueba 1 (127,608 triángulos en total) : Se a considerado como una escena de grandes dimensiones con una baja densidad de triángulos.



Prueba 2 (451,186 triángulos en total) : Se a considerado como una escena de grandes dimensiones con una densidad media de triángulos



Prueba 3 (910,556 triángulos en total) : Se a considerado como una escena de grandes dimensiones con una alta densidad de triángulos.



En la primera prueba, en los equipos con prestaciones inadecuadas para aplicaciones 3D (equipo 1), se pudo observar en todo momento un “bajo” rendimiento. En equipos con bajas prestaciones (equipo 2), se obtuvo la mayor parte del tiempo un “buen” rendimiento llegando en ocasiones a “excelente”. En los equipos con características medias-altas (equipos 3 y 4), siempre se obtuvo un rendimiento “excelente”.

En la segunda prueba, en equipos inadecuados para aplicaciones 3D pudimos observar un rendimiento prácticamente “pobre” en todo momento. Sin embargo en equipos con bajas prestaciones se obtuvo un rendimiento “aceptable” la mayor parte del tiempo, llegando en ocasiones a obtener un “buen” rendimiento. En equipos con características medias-altas se obtuvo la mayor parte del tiempo un “excelente” rendimiento, disminuyendo en ocasiones a “bueno”.

En la tercer y última prueba, en equipos inadecuados para aplicaciones 3D el rendimiento bajo hasta el punto en que fue imposible utilizar adecuadamente la aplicación debido a la falta de poder de procesamiento. En equipos con prestaciones bajas se obtuvo un rendimiento “bajo” a “pobre” prácticamente la mayor parte de tiempo. En los equipos con prestaciones medias-altas se obtuvo prácticamente en todo momento un rendimiento “aceptable”.

5.2. Problemas encontrados.

A lo largo de este proyecto de tesis se suscitaron algunos problemas para los cuales y debido a su naturaleza se requirió de varios días para hallar una solución a estos. Los problemas que serán descritos a continuación representaron un retraso importante para el desarrollo del proyecto.

Problemas relacionados con el ambiente de desarrollo.

- El ambiente de desarrollo *Codeblocks* junto con el compilador *GCC fueron utilizados* para el desarrollo del software. A diferencia de Visual Studio, Codeblocks no es un ambiente considerado como un estándar para el desarrollo de aplicaciones por lo cual es poca la documentación que se puede encontrar y constantemente se tuvo que recurrir a foros de discusión y a la prueba y error para utilizarse correctamente en conjunto con el compilador y las librerías instaladas.

Problemas relacionados con las librerías utilizadas: SDL, Glew.

- Generalmente el desarrollador de una librería externa al API estándar de C++ proporciona una distribución específica según el sistema operativo a utilizar. Cuando se trata de la plataforma Windows lo común es que se proporcione los archivos fuentes junto con un archivo de proyecto del ambiente de desarrollo, o bien las librerías precompiladas pero casi siempre enfocadas a ser usadas bajo el compilador y ambiente de desarrollo oficiales de Microsoft y teniendo esto en mente generalmente se proporcionan como un archivo ejecutable de fácil instalación. En este proyecto de tesis no se utilizaron las herramientas oficiales a ser utilizadas bajo Windows y es por esta razón que no se pudieron utilizar directamente las librerías distribuidas específicamente para este sistema operativo. Finalmente se aprendió que para hacer uso de las librerías, éstas se deben recompilar de forma similar a como se realiza bajo el sistema operativo Linux el cual también hace uso del compilador *GCC*.

- Como se ha sido mencionado a lo largo de este documento, las librerías SDL han sido utilizadas para controlar los eventos del teclado y mouse así como también para llevar a cabo la gestión de ventanas del software. Es por esto último que mediante SDL se debe configurar de manera apropiada una ventana para poder ser utilizada posteriormente por OpenGL. Dicha configuración representó otro problema importante en el desarrollo de software ya que, como se comprobó en la práctica, incluso el orden de los comandos correspondientes y

la inclusión u omisión de algún atributo dentro de ellos puede restringir el uso de alguna característica importante como en el caso de la activación de OpenGL, el uso adecuado de los “buffers” de video, la aceleración mediante el hardware 3D y activar la sincronización vertical con el monitor.

- Se invirtió varios días en buscar el modo de activar las extensiones de OpenGL (específicamente para poder usar vertex buffers). Se descubrió que SDL no cuenta con el apoyo suficiente para poder activarlas ya que aún requiere del uso de funciones específicas del sistema operativo utilizado. Finalmente se recurrió a la librería multiplataforma *Glew* diseñada para activar de manera sencilla la gran mayoría de las extensiones de OpenGL tanto las estándar como aquellas propuestas recientemente por fabricantes de hardware 3D.

Problemas relacionados con OpenGL.

- Es necesario habilitar el conjunto de características de OpenGL a ser utilizadas. Al igual que en el caso de SDL, el orden y la inclusión u omisión de los comandos y atributos correspondientes para la habilitación de ciertas características de OpenGL resulta de vital importancia para su correcto funcionamiento. Esto llego a consumir varios días de investigación y pruebas.

Problemas relacionados con C++.

- Inicialmente se utilizaron objetos para representar a los elementos 3D y sus atributos dentro de la escena. Se invirtió tiempo modificando código para utilizar estructuras el lugar de objetos demostrando un incremento en la velocidad de procesamiento al utilizar estructuras.

- Inicialmente al utilizar los contenedores de la librería estándar STL de C++ se produjo un mal funcionamiento de estos. Posteriormente se descubrió que estos contenedores solo pueden manejar correctamente tipos de datos simples y no

tipos complejos como estructuras. Finalmente solución consistió en almacenar solo las referencias a las estructuras que se deseaban almacenar (punteros).

- En las últimas etapas de desarrollo, eventualmente la ejecución del software terminaba inesperadamente. Después de días de análisis y búsqueda se descubrió que el contenido de un puntero que nunca fue inicializado o nunca le fue asignado un valor, contiene información inconsistente que afecta el funcionamiento de cualquier comando que requiera acceder a ella.

Problemas relacionados con el diseño e implementación de algoritmos.

- Un problema que se tiene con los algoritmos recurrentes radica en que cuando ocurre un problema es difícil detectarlo y aislarlo debido a que no solo hay que detectar en que parte del código ocurre si no también el momento en que ocurre y la razón de ocurrir en ese lugar y momento. Es por esta razón que en el desarrollo del código referente a la construcción de octrees se invirtieron varios días en localizar y corregir algunos errores que finalmente en la mayoría de los casos se descubrió que eran pequeños detalles pasados por alto.

- Inicialmente existieron problemas para la correcta implementación de los procedimientos matemáticos utilizados. La incorrecta interpretación de éstos llevo a realizar varias modificaciones y reescritura de código. Para evitar un retraso mayor se optó por verificar visualmente estos procedimientos mediante la herramienta de software *Geometer SketchPad* con la cual fue posible ver el resultado geométricamente y de forma interactiva antes de escribir el código correspondiente.

- Se invirtieron varios días desarrollando un algoritmo para agrupar los triángulos de los objetos 3D en "*triangle fans*" (abanicos). La ventaja de utilizar "*triangle fans*" radica en que se requiere procesar una menor cantidad de vértices para dibujar un conjunto de triángulos. La cantidad de vértices utilizados en cada "*triangle fan*" es variable debido a lo cual se requiere una llamada a OpenGL para dibujar cada "*triangle fan*" sin importar el modo de dibujo del que se trate.

Estos es factible cuando se utiliza *modo inmediato* en el cual se redujeron los tiempos de despliegue a un %70 aproximadamente. En el caso de los modos de dibujo más avanzados, la única forma de poder dibujar todos los "*triangle fans"* con una sola llamada sería si todos ellos tuviesen el mismo número de vértices pero debido a que el número de vértices de cada "*triangle fan"* es variable se tiene que realizar una llamada por cada uno de ellos con lo cual en lugar de una mejora en el rendimiento se obtiene una severa disminución de este aunado al hecho de que por cada llamada se procesa solo una muy pequeña cantidad de triángulos (se ha mencionado que los modos de dibujo avanzados trabajan correctamente al procesar por lo menos cientos de triángulos por llamada). La razón principal por la cual no se implemento como una opción más fue debido a que ocurrieron problemas con las coordenadas de texturas al implementar este algoritmo. En un foro de discusión se comento que en ocasiones OpenGL tiene problemas con las texturas cuando en un mismo "*triangle fan"* se utiliza una misma coordenada de textura más de una vez. Por otro lado, al utilizar este algoritmo el tiempo inicial de carga del software aumentaba notablemente.

Problemas relacionados con hardware y otras herramientas de software.

- En las primeras etapas un problema importante consistió en el desarrollo de un procedimiento para extraer la información correspondiente a un objeto 3D almacenado en un archivo. Sin esta capacidad era imposible comprobar el correcto funcionamiento de los algoritmos a desarrollar. Uno de los alcances del proyecto consiste en el uso de un formato 3D de amplia difusión por lo cual inicialmente se optó por utilizar una librería con la cual era posible extraer la información del formato 3DS (3D Studio) así como también se desarrollo código correspondiente para el formato MS3D (MilkShape 3D). Estos dos formatos son de tipo binario y su principal desventaja radica en que no son capaces de almacenar modelos con más de 65536 triángulos o vértices. Finalmente se evito el uso de la librería para importar modelos 3DS debido a que se trataba de código propietario. El uso del formato MS3D se evito en las fases finales de desarrollo y se optó por construir un "parser" para el formato OBJ (Alias wavefront) cuyo contenido se encuentra en forma de texto y no posee la

desventaja antes mencionada aunque el tamaño de los archivos es mayor y se requiere un poco más de tiempo para su procesamiento.

- La búsqueda de un editor de escenarios con las características requeridas para ser usado en conjunto con el software desarrollado requirió de varios días de búsqueda. Los paquetes más populares de modelado 3D son incapaces de manejar eficientemente grandes cantidades de polígonos y la mayoría de los editores de escenarios poseen características enfocadas a su uso específico con unos cuantos paquetes de software. Solo dos editores cumplían con las requerimientos buscados: 1) *Panorama terrain editor*, desarrollado por Jeppe Nielsen; 2) *T.ED*, desarrollado por D-Grafix. Estos dos paquetes son comerciales y aunque en realidad no gozan de gran popularidad son bastante accesibles. Finalmente se decidió que la herramienta T.ED cumplía con todas las expectativas.

- Durante las pruebas de ejecución en distintos equipos de cómputo ocurrió que el software solamente funcionaba en el equipo en donde fue desarrollado. Después de varios días se descubrió que el hardware de video de los demás equipos en que fue probado el software no era capaz de utilizar *vertex buffers* por lo cual se tuvo que incluir código para detectar su disponibilidad en el equipo de cómputo utilizado para evitar su uso asegurando así el correcto funcionamiento del software. Las pruebas se realizaron nuevamente y esta vez con éxito. Solo existieron problemas en equipos antiguos.

- Un problema importante ocurrió cuando se deseaba controlar el número de cuadros por segundo a desplegar. Inicialmente se utilizaron las funciones para medición de tiempo proporcionadas por la librería estándar de C++ y SDL pero la frecuencia con la que eran desplegados los cuadros era aún errático debido a que su precisión es en mili segundos. Se desarrollo posteriormente un mecanismo en lenguaje ensamblador que obtiene inmediatamente los ciclos de procesador transcurridos con lo cual se obtuvo una precisión en nano segundos mejorando considerablemente el proceso de despliegue de los cuadros. Finalmente se obtuvieron los mejores resultados cuando se sincronizó el despliegue de cada cuadro con la frecuencia vertical del monitor.

- Otro problema importante consistió en hallar la forma de distribuir el software de modo que pudiera ser implementado lo más fácilmente posible y sin caer en el paradigma de su uso en una sola plataforma. Finalmente se optó por compilar el software en forma de una librería dinámica y opcionalmente proporcionar el código fuente junto con un archivo de proyecto para el ambiente CodeBlocks para así poder compilar el software en distintas plataformas.

5.3. Trabajo futuro.

Evidentemente es inmensa la cantidad de características que pueden ser desarrolladas para el software. Sin embargo podemos mencionar las implementaciones próximas inmediatas que desean realizarse:

- Formato propio con la información preprocesada de los octrees.
- Utilizar nivel de detalle (LOD) en conjunto con los octrees para acelerar el proceso de despliegue.
- Más modos de cámara: caminar en primera y tercera persona sobre el terreno.
- Estudiar que tan factible sería combinar el uso de octrees con otros métodos como podría ser el uso de portales o árboles BSP.

5.4. Conclusiones.

Se observó que existen varios factores cuya naturaleza dificulta automatizar la configuración óptima para obtener el rendimiento óptimo del software según la las características de la escena utilizada:

- *Far plane del view frustrum*: Mientras mayor sea la distancia a la que se encuentra el *far plane*, se tendrá mejor percepción de la profundidad del mundo 3D y por consiguiente una mejor estética de la escena aunque sin embargo a mayor distancia implica que mayor será la cantidad de elementos a procesar resultando en una disminución en el rendimiento general. Determinar la distancia óptima del *far plane* puede resultar una tarea que dependa más de la opinión humana que de un mecanismo automatizado ya que una máquina puede medir el rendimiento pero no tiene sentido de la estética.

- *Tamaño de los octantes de colisión*: El rendimiento general también es afectado por el tamaño elegido para los octantes de colisión. Mientras más pequeño sea el octante de colisión mayor será la precisión con que se colisiona con los objetos 3D con lo cual se tiene un resultado visual más realista de la colisión. Sin embargo cuando utilizamos octantes de colisión muy pequeños, además de realizar un recorrido más largo del octree, podría resultar de mayor costo procesar la geometría de los octantes de colisión que la geometría del objeto 3D mismo. Por estas razones se consideró la intervención humana como lo más adecuado para determinar el tamaño de los octantes de colisión.

- *Tamaño de los octantes de despliegue*: Es importante desplegar el mayor número de triángulos en una sola llamada a OpenGL para obtener un rendimiento adecuado. Por ejemplo, es mucho más eficiente desplegar 10,000 triángulos en una sola llamada a OpenGL que desplegar los mismos 10,000 triángulos realizando más de una llamada a OpenGL. Por consiguiente, mientras más grandes sean los octantes de despliegue, se desplegará un mayor número de triángulos en una misma llamada. Sin embargo hay que tomar en cuenta que si el octante de despliegue es demasiado grande, podrían procesarse más triángulos que se encuentren fuera del *view frustrum* que aquellos que si son visibles en cuyo caso podría resultar más costoso que utilizar octantes de despliegue más pequeños aunque se realicen mayor cantidad de llamadas a OpenGL. Por esta razón también se le a confiado al criterio humano la determinación del tamaño de estos octantes.

En lo referente a la administración de una escena se concluye que el uso de octrees con este fin resulta una buena opción para escenarios exteriores pero sin embargo para escenarios interiores cómo puede ser el recorrido interior de una casa, pueden existir demasiados octantes cuya geometría no se encuentra a la vista. Para evitar los cálculos innecesarios haría falta utilizar un algoritmo para determinar los octantes cuya información no es visible para evitar así su procesamiento.

En lo referente a la detección de colisiones utilizando octrees, se considera que solo vale la pena utilizarse para modelos que utilicen una gran cantidad de triángulos o bien cuando no se requiera demasiada precisión en las colisiones. Si utilizáramos un modo de cámara en donde se simula que se camina sobre el terreno, evidentemente utilizando octrees no podríamos detectar los cambios de inclinación por lo cual en este caso su utilización con este fin no es adecuado. También se pudo comprobar que al procesar los octantes de colisión de la forma en que fue realizada resulta sumamente costoso. Si alternativamente evaluáramos las esferas que envuelven a los octantes de colisión la diferencia en la precisión de la colisión sería mínima pero en teoría el costo computacional requerido es mucho menor. Sin embargo, las rutinas de intersección desarrolladas constituyen una herramienta valiosa para implementaciones futuras.

En cuanto a los objetivos y alcances planteados inicialmente se considera que han sido cumplidos faltando únicamente probar su portabilidad a otras plataformas.