Universidad Nacional Autónoma de México

Facultad de Ingeniería

"MODELOS TRIDIMENSIONALES DE ALTO DESEMPEÑO"

Tesis que para obtener el título de Ingeniero en Computación presentan:

Rojas González Renato Carlos

Soto Serafin Alejandro

Valencia Castro Luis Sergio

Director de tesis

Ing. Emmanuel Hernández Hernández.

Ciudad Universitaria, Abril de 2006.

Dedicatoria de Renato Carlos Rojas González.

Dedico el presente trabajo con cariño a:

- *Mis padres Máximo Rojas de la Cruz y Sofía González Sandoval.*
- *Todos los integrantes de mi familia presentes y a los ya ausentes.*
- *Todos mis amigos.*

Agradecimientos de Renato Carlos Rojas González.

Quiero agradecer a todas las personas que hicieron posible que este trabajo llegara a su fin.

- *A la Universidad Nacional Autónoma de México.*
- *A la Facultad de Ingeniería.*
- *A mi director de tesis Emmanuel Hernández Hernández.*
- *A mis amigos y compañeros de tesis, Alejandro y Luis Sergio.*

Dedicatoria de Alejandro Soto Serafín:

- *Este trabajo se lo dedico a mis padres Saturnina Serafín Hernández y José Soto Ruiz por apoyarme en todos los sentidos a lo largo de mi formación profesional.*

- *A mi tío Enrique Soto Ruiz por todas las experiencias que me obligaste a vivir pero que valieron mucho la pena para ser la persona que soy.*

- *Y también se lo dedico a todas aquellas persona que quieran adentrase al mundo de las graficas por computadora a través del trabajo que aquí se presenta.*

Agradecimientos Alejandro Soto Serafín:

- *Le agradezco a la Universidad Nacional Autónoma de México por permitirme adquirir las experiencias y conocimientos que me ayudaron a tener la capacidad de realizar este trabajo.*

Les agradezco a todas las personas que intervinieron en el desarrollo de este trabajo:

- *A mis compañeros Renato Carlos y Luis Sergio por su trabajo y empeño para que este trabajo saliera a flote pero más por ser los mejores amigos que he tenido, los cuales han estado conmigo en las buenas y en las malas.*

- *A todos los demás amigos que conocí y que hicieron que la estancia en la UNAM fuera una experiencia muy placentera*

- *Y Emmanuel Hernández Hernández por ser la persona que me sirvió de influencia para dedicarme profesionalmente al área de las graficas por computadora*

Dedicatorias de Luis Sergio Valencia Castro:

El presente trabajo lo dedico a:

- *Mi madre, Reyna Cristina, por ser ese pilar sobre el cual me he apoyado en los momentos de mayor apremio.*
- *Mi hermano, Einar, por hacerme reír y enojar en momentos en que lo necesitaba.*
- *A Mayra, por haber sido más que una amiga, y haber estado a mi lado en los momentos más felices, y aún ahora sigues estando a mi lado.*
- *A mi padre, Sergio, por enseñarme lo que no debo de hacer y ser estricto cuando lo necesitaba.*

Agradecimientos de Luis Sergio Valencia Castro:

No hay palabras para agradecer a las personas que, directa o indirectamente llevaron a que este trabajo pudiera llevarse acabo:

- *Mi familia, por el apoyo que siempre me han dado.*
- *Renato Carlos y su familia, debido a que la primer parte de este trabajo se realizó en su domicilio, y sentimos todas las molestias que pudimos haber ocasionado. Además por ser el más responsable de todos nosotros.*
- *Alejandro, por poner todo su empeño para terminar lo más pronto posible, aún cuando tenía que dedicarle tiempo a su trabajo.*
- *Emmanuel, por habernos introducido al mundo de las gráficas por computadora, sin su guía hubiéramos estado perdidos.*
- *Sala Ixtli, por las facilidades prestadas para la realización de este trabajo.*
- *Universidad Nacional Autónoma de México ya que gracias a esta institución pude formarme como profesional y persona.*

ÍNDICE

CAPÍTULO 1.

Introducción.

1.1. Objetivo.

Proponer un formato de archivo con información de modelos en 3D que permita una carga de información poligonal a memoria principal de manera rápida y eficiente, desde cualquier medio, incluyendo Internet. También se desarrollará un API de carga de modelos, la cual permitirá la creación de aplicaciones de una manera más rápida y estructurada.

1.2. Definición del problema.

En el desarrollo de aplicaciones que hacen uso de gráficos tridimensionales en tiempo real (por ejemplo: sistemas de visualización científica, simulación física, videojuegos, modeladores, etc.) uno de los problemas más grandes es la forma en la cual se guarda la información de un archivo, el cual contiene los elementos que componen la escena, datos que el sistema de dibujo (OpenGL o Direct3D) va a mostrar en pantalla. Ya que si la información del modelo es pequeña, el proceso de carga así como el de dibujo será rápido pero el detalle del modelo será deficiente, en cambio si es muy grande la cantidad de información del modelo, tendrá una calidad superior pero tardará más en pasar del dispositivo de almacenamiento secundario a memoria principal y una vez en ella el dibujo del modelo será también más tardado. Los efectos más claros de este problema son los largos tiempos de carga, que gravemente afectan el desempeño de la visualización, haciendo incluso que la aplicación se detenga a mitad del ciclo de dibujo para cargar nuevos objetos, reduciendo el desempeño y con ello la sensación de realismo en las aplicaciones de visualización.

Pensando en esto, uno de los objetivos de esta tesis es desarrollar un formato de archivo que permita minimizar los tiempos de carga de la información poligonal del modelo, ocupando para ello diversas técnicas, y de esta manera obtener un mayor desempeño inclusive con el uso de modelos extensos, como los que serán usados para el proyecto de visualización de la Planta Hidroeléctrica de El Cajón, el cual es un proyecto multidisciplinario que permitirá visualizar cada una de las etapas de la construcción de esta obra en el Observatorio de Visualización Ixtli, el cual hará uso intensivo de los mecanismos de carga de modelos proporcionados como resultado del presente trabajo.

Se diseñarán e implementarán mecanismos de carga, con el objeto de resolver el problema de un envío eficiente de información a través de un canal de comunicación con capacidad limitada, como son los usados en una red de tipo global, por ejemplo Internet, de manera que el desempeño y la calidad de la visualización se afecten en lo mínimo posible hasta que toda la información del archivo llegue completa a su destino, problema que podría resolverse con

algoritmos de carga de alto rendimiento como el “streaming” comúnmente usado por reproductores de audio en línea.

El objetivo del diseño de estos mecanismos de carga no son solo por el hecho de administrar mejor los recursos, sino además, para usarse transparentemente en un esquema cliente/servidor que permita a diferentes aplicaciones en paralelo solicitar solo los datos que necesitan visualizar en determinado momento.

Aunque el enfoque de las aplicaciones en 3D sea muy diferente, como son la investigación, el desarrollo o el entretenimiento, en la mayoría de ellas se hace uso de información de modelos poligonales almacenados en un archivo, es por eso que otro de los objetivos de este trabajo es hacer que el manejo de estos archivos sea lo más sencillo posible para el desarrollador de estas aplicaciones, esto se logrará por medio del desarrollo de librerías de programación (API, Application Programming Interface) que manipularán el nuevo formato de archivo, este API disminuirá el trabajo a los programadores de aplicaciones en 3D, permitiéndoles mostrar modelos de alta calidad en sus aplicaciones de manera sencilla, de esta manera concentrarse en la programación de la dinámica y flujo de su aplicación y no en cómo programar la carga de modelos, ya que el API programada para este trabajo de tesis es la que se encargará de ello.

El trabajo de esta tesis tratará de solucionar estos problemas, y agrupará estas respuestas en una librería, la cual tendrá por objetivo hacer accesibles estos mecanismos a otros programadores, para así reducir el tiempo de desarrollo e incrementar el rendimiento de sus aplicaciones, que hacen uso de información poligonal para el almacenamiento de sus objetos.

1.3. Método.

El desarrollo de un formato de archivo para almacenar información de modelos tridimensionales de alto rendimiento estará compuesto de varios procesos y etapas para llegar a un resultado satisfactorio. A continuación se describirá el método que se seguirá para llevar a cabo cada uno de ellos.

La primera etapa será la investigación de los distintos formatos de archivo con información de modelos tridimensionales, estudiando sus características únicas así como características en común con los demás formatos, de esta manera conocer las ventajas y desventajas sobre el diseño de cada uno de estos formatos. De este análisis, se determinará la información esencial para la representación de un modelo.

Una vez elegidos los formatos de archivo que contengan mayor información acerca de la geometría del modelo, se trabajará con ellos para el diseño de las

estructuras de datos en memoria principal que alojarán los datos más necesarios para que el aspecto del modelo sea el mejor posible, este diseño en memoria principal está pensado para que el proceso de dibujo sea lo más rápido posible; el mecanismo que coloca los datos del archivo a memoria principal es un “Cargador”, y el mecanismo que toma los datos de memoria y dibuja el modelo en la pantalla es el “Renderer”.

La programación de este primer cargador será la referencia con la cual debemos de evaluar el cargador del nuevo formato de archivo, en estos procesos se utilizará el lenguaje de programación C, ya que es un lenguaje de programación de propósito general, conocido por su eficiencia y portabilidad. Mientras estas cualidades lo hacen una buena elección para cualquier tipo de requerimiento de programación, C ha demostrado mucha utilidad en el desarrollo de sistemas por que facilita el escribir programas rápidos y compactos, que son rápidamente adaptables a otros sistemas. Un programa bien escrito en lenguaje C es tan rápido como programas en lenguaje ensamblador, y además de que el código es fácil de leer y de mantener.

Estas características que ofrece C proporcionan la funcionalidad y soporte para crear las librerías que ayuden a la carga rápida y eficiente de modelos tridimensionales, que incluso otros programadores podrán aprovechar.

Para el desarrollo del “Renderer” que se ocupará para las pruebas, existen dos API’s que brindan interfaces de software para el hardware de gráficos de la computadora, Direct3D y OpenGL, estos dos API’s son los que se tomarán en cuenta, ya que por una parte Direct3D es ampliamente usado para la creación de videojuegos y otras aplicaciones gráficas sobre el sistema operativo Windows, y OpenGL por que su compatibilidad con otros sistemas operativos es mucho mayor. En el proyecto para la visualización en la sala Ixtli, de la construcción de una hidroeléctrica, este trabajo de tesis será el encargado de poner en memoria principal los datos a dibujarse, lo cual no implica que se participe en el proceso de dibujo de los modelos. Sin embargo es necesario, para este trabajo de tesis, el desarrollo de un “Renderer” para saber que el diseño de estructuras de datos es el correcto para lograr un buen desempeño de visualización.

Una vez identificada la información útil del archivo origen, le sigue el proceso de diseño del nuevo formato. En esta etapa se buscará que el formato cuente con características únicas y que su diseño permita un mejor desempeño, con respecto a los demás formatos, en su uso dentro del proyecto de visualización. Además desarrollar el “Cargador” de este nuevo formato que debe solucionar los problemas respecto a tiempo de carga, facilidad de manipulación, así como un envió eficiente y seguro a través de una red, esto último se tratará de resolver mediante el empleo de algoritmos, los cuales consistirán en enviar la información con más alta prioridad, para de esta manera tener un desempeño constante en la visualización, mientras que un proceso sigue recibiendo los datos faltantes, asegurándose que los datos recibidos desde la máquina remota son los correctos mediante el uso de un sistema de detección de errores.

Al mismo tiempo se diseñará un API de programación para la carga y manipulación de modelos con este nuevo formato, es decir, se desarrollará un conjunto de funciones que puestas en una secuencia lógica permitan el manejo de la carga y visualización de modelos. La forma de uso del API obedecerá al paradigma de Programación Orientada a Objetos ya que permite al programador un desarrollo más intuitivo debido al mayor nivel de abstracción que este enfoque propone. El objetivo es generar una librería precompilada para su distribución a desarrolladores de aplicaciones con gráficos en 3D, de esta manera, el API desarrollado, no solo se utilizará para el proyecto de visualización Ixtli, sino que estará disponible para cualquier otro proyecto en donde se requiera una carga eficiente de modelos poligonales.

Para verificar la funcionalidad del API de programación, se desarrollará una aplicación con una interfaz de usuario más amigable para el usuario final de cada una de las funciones del API, de manera que fácilmente pueda verificar todas las mejoras que implementa. Esta interfaz de usuario se realizará con el entorno de desarrollo de Borland C++Builder, ya que hace transparente el uso del API Windows para manejo de ventanas, dando como resultado la creación de interfaces de usuario muy profesionales con muy poco tiempo invertido.

CAPÍTULO 2.

Antecedentes teóricos.

2.1. Gráficas por computadora.

2.1.1 Antecedentes históricos.

La investigación de Graficación por Computadora tiene una historia rica, no obstante breve. Como en cualquier campo, resulta difícil asignar graficación a un dominio específico: por definición es multidisciplinaria. Por ejemplo, muchos investigadores trabajan para hacer de la computadora una mejor herramienta para ayudar en la colección e interpretación de información científica a través de la visualización científica, mientras que otros científicos en el procesamiento de imágenes trabajan para desarrollar técnicas para interpretar imágenes para información y enriquecimiento. Además, la animación por computadora ha cambiado el panorama de la industria del entretenimiento para siempre.

Ahora se toman por cotidianas las experiencias surrealistas que se presentan en los comerciales con gatos que hablan y carros que se transforman. La realidad virtual tiene el potencial para cambiar la noción de la realidad, que a fin de cuentas proporciona un mejor entendimiento de lo que se es y del potencial que se tiene para llegar a ser algo más.

La geometría computacional se esfuerza por hacer las representaciones en la computadora más precisas, robustas y realistas. Desde las matemáticas a la medicina hasta al arte, los gráficos de computadora han avanzado y seguirán avanzando en esos campos. El hombre está continuamente esforzándose para lo que Ivan Sutherland, el padre de la graficación interactiva por computadora, llamaba El Último Despliegue. Uno en el cual el mundo "...se ve real, actúa real, suena real, y se siente real." [3]

El uso adecuado y provechoso de la tecnología ha hecho de la computadora un dispositivo poderoso para producir imágenes en forma rápida y económica. Actualmente en todas las áreas es posible aplicar gráficas por computadora con algún objetivo, por ello se ha generalizado la utilización de gráficas por computadora. De igual modo las gráficas por computadora se utilizan de manera rutinaria en diversas áreas, como en la ciencia, ingeniería, empresas, industria, gobierno, arte, entretenimiento, publicidad, educación, capacitación y presentaciones gráficas.

Las computadoras se han convertido en una herramienta poderosa para producir imágenes, interpretar información o mejorar la calidad de visualización de las mismas en forma rápida y económica. Antes de comenzar a describir la historia de la Graficación por Computadora, es necesario aclarar algo, mucha gente tiende a confundir lo que es la graficación por computadora y el tratamiento digital de imágenes, siendo esta última en la actualidad una actividad que interesa a mucha

gente debido a la aparición de la fotografía digital. Sin embargo se debe aclarar que los métodos que se utilizan en las gráficas por computadora y en procesamiento de imágenes tienen características similares pero no son iguales es decir, las dos áreas realizan, en forma fundamental operaciones distintas. Las herramientas para graficación por computadoras, se utilizan para crear una o más imágenes. Por otro lado, en el procesamiento de imágenes se aplican técnicas para modificar o interpretar imágenes existentes como fotografías y rastreos de televisión. El interés por los métodos de tratamiento o procesamiento digital de imágenes deriva de dos áreas principales:

a) La mejora de la información para la interpretación humana.

b) El procesamiento de los datos de la escena para la percepción autónoma por una máquina.

Los avances en la tecnología de la computación han hecho que las gráficas interactivas por computadora sean una herramienta practica, éstas se utilizan en diversas áreas como la ciencia, la ingeniería, empresas, industria, gobierno, arte, entretenimiento, publicidad, educación, capacitación y presentaciones gráficas.

Como resultado del reconocimiento generalizado de la potencia y la utilidad de las gráficas por computadora hay una gran variedad de hardware y software para gráficos. Debido a lo común que se ha vuelto la capacidad de gráficos tanto para aplicaciones bidimensionales como tridimensionales incluyendo calculadoras manuales. Para aplicaciones que requieren una calidad superior se verán varios sistemas y tecnologías avanzados de hardware.

2.1.2. Librerías de programación para gráficos.

2.1.2.1. OpenGL.

OpenGL (Open Graphics Library) es la especificación para definir el API de programación multilenguaje para múltiples plataformas para la creación de aplicaciones que producen gráficas en computadora en 3D (así como también gráficos en computadora en 2D). La interfaz consiste en cerca de 250 diferentes llamadas a funciones, las cuales pueden ser utilizadas para dibujar escenas muy complejas en tercera dimensión con la utilización de simples primitivas. Además de que cuenta con una gran popularidad en la industria de los videojuegos donde compite con Direct3D (en Microsoft Windows). OpenGL es utilizado en herramientas CAD, de realidad virtual, en programas de visualización científicas, visualización de información y en el desarrollo de juegos de video. [11]

La implementación eficiente de OpenGL (haciendo uso de la aceleración por hardware en mayor o en menor medida) existe para Windows, muchas plataformas Unix, Playstation 3 y Mac OS. Estas implementaciones son

generalmente proporcionadas por los fabricantes de los dispositivos de despliegue (tarjetas de video entre otros), y dependen en gran medida del hardware proporcionado por el fabricante. La librería de código abierto MESA es un API gráfico basado completamente en software, y es compatible en código con OpenGL. Aunque, por razones de licencia, se dice ser meramente un API muy "similar".

La especificación OpenGL es supervisada por el OpenGL Architecture Review Board (ARB), el cual fue formado en 1992. El ARB consiste en un grupo de compañías con un gran interés en la creación de un API consistente y con una gran disponibilidad. De acuerdo con el sitio oficial en Internet de OpenGL, los miembros con voto dentro del ARB a Noviembre del 2004 son 3Dlabs, Apple Computer, ATI Technologies, Dell Inc., Evans & Sutherland, Hewlett-Packard. IBM, Intel, Martos, NVidia, SGI y Sun Microsystems. (Microsoft, una de las compañías fundadoras, abandonó el grupo en Marzo del 2003)

El estándar OpenGL permite a los vendedores individuales o fabricantes proporcionar funciones adicionales mediante *extensiones* mientras nuevas tecnologías son creadas. Una extensión, es entonces, distribuida en dos partes, como una cabecera, la cual contiene los prototipos de las funciones de la extensión, y la otra parte es el driver del dispositivo del fabricante.

Muchas librerías han sido construidas en base a OpenGL, esto con el fin de proporcionar características no encontradas en el mismo OpenGL, ejemplo de dichas librerías son *GLUT*, *GLU* y *GLAUX*. Otras librerías, tales como OpenGL Performer, desarrollada por SGI y disponible para IRIX, Linux, y varias versiones de Microsoft Windows, ayuda a que OpenGL proporcione la creación de aplicaciones para la simulación visual en tiempo real.

Para que OpenGL fortaleciera su característica multilenguaje y multiplataforma, se han realizado varias referencias y adaptaciones para el desarrollo de OpenGL en diferentes lenguajes de programación. Uno de los ejemplos más notables es la librería Java 3D, la cual depende de OpenGL para su aceleración por hardware. Más recientemente, Sun ha mostrado al público, versiones beta de el sistema JOEL, el cual proporciona una adaptación directa de los comandos C OpenGL, los cuales, al contrario de Java 3D, proporciona un soporte a un nivel más bajo.

OpenGL fue diseñado para ser "solo salida", es decir, solo se proporcionan funciones para el dibujo. El centro del API no tiene conceptos de ventanas, audio, impresión, teclado, mouse, u otros dispositivos de entrada. Aún cuando esto parezca restrictivo al principio, esto permite al código que realiza el dibujo ser completamente independiente del sistema operativo en el cual se encuentre, lo que permite el desarrollo multiplataformas. Sin embargo se requiere cierta integración con el sistema de ventanas nativo del sistema operativo, esto para permitir una interacción sencilla con el sistema en el cual trabaja. Esto se realiza mediante los siguientes agregados al API:

a) GLX, X11 para el sistema X Window
b) WGL, para Microsoft Windows.
c) AGL para Apple Macintosh

Adicionalmente las librerías GLUT y SDL permiten una funcionalidad básica para el sistema de ventanas con el uso de OpenGL para un manejo portable. Mac OS X tiene tres APIs para el soporte de OpenGL, las cuales son AGL por Carbon (para compatibilidad en retrospectiva con anteriores sistemas operativos Mac), NSOpenGL por Cocoa (basado en Carbon, pero con mejoras para el sistema Mac OS X) y GGL para el acceso directo.

OpenGL evolucionó (y es muy similar en estilo) de la interfaz 3D de SGI, *IRIS GL*. Una de las restricciones de IRIS GL es que solo proporcionaba acceso a las características soportadas específicamente por el hardware, es decir, si el hardware no soportaba una característica, entonces la aplicación no podía usarla. OpenGL solucionó este problema al proporcionar soporte en software para las características que no eran soportadas por el hardware, permitiendo que las aplicaciones usaran gráficas avanzadas en sistemas relativamente poco poderosos.

2.1.2.2. DirectX.

Direct3D es parte del API DirectX de Microsoft. Direct3D solo puede ser utilizado en variantes del sistema operativo Windows de Microsoft, y en el Xbox, aún cuando en este último se trata en realidad de una versión modificada del API. Direct3D es utilizado para el dibujo (render) de gráficas en tres dimensiones en aplicaciones donde el rendimiento es importante, tales como juegos. Direct3D además permite que las aplicaciones corran a “pantalla completa” (full screen) en lugar de obligar que trabajen dentro de una ventana, aún cuando esta última sigue siendo una opción para la presentación dentro del API. Direct3D utiliza la aceleración por hardware en caso de que ésta esté disponible. [6]

Direct3D es un API de 3D. Es decir, contiene comandos para la presentación de gráficas en 3D, pero además contiene algunos comandos para el dibujo de gráficas en 2D. Microsoft se ha esforzado en actualizar constantemente Direct3D, esto con la finalidad de ofrecer soporte a lo último de la tecnología disponible en las tarjetas de video de 3D.

En la versión DirectX 8.0, Direct3D se incluyó dentro de un paquete denominado DirectX Graphics, el cual se trataba de una combinación de DirectDraw y Direct3D (DirectDraw es la parte del API encargada el dibujo en 2D), pero en realidad era solo Direct3D con algunas características de DirectDraw.

Direct3D no era considerado como un API amigable con el usuario, pero con la aparición de la versión 8.1, muchos problemas fueron resueltos. En esta versión Direct3D (DX8) contenía características muy poderosas para los gráficos en 3D, tales como vertex shaders, píxel shaders, niebla, bump mappping y texture mapping.

En DirectX versión 9.0 se incluyó una nueva versión del lenguaje de alto nivel para shaders (High Level Shader Language), el cual soporta un alto rango dinámico de luces, múltiples objetivos de dibujo y un indexado para el buffer de vértices.

Sin embargo uno de los principales inconvenientes que se sigue encontrando en las versiones actuales, es que Direct3D no ofrece soporte para la emulación en software de píxels (píxel software emulator) lo cual provoca que en ocasiones si cierta aplicación requiere del uso de píxel shaders y la tarjeta gráfica del usuario no soporta esta característica, Direct3D no emulará estas instrucciones por software, lo que ocasiona que aparezca un mensaje de error y la aplicación se finalice.

2.1.2.3. OpenGL vs Direct3D.

A pesar que en la actualidad Direct3D y OpenGL son más perecidos de lo que solían ser, aún presentan algunas diferencias, y esto dio lugar a la “guerra de las APIs”, ya que cada uno tiene sus seguidores. A continuación se mostrará un análisis en diferentes áreas en cada una de las APIs. [11]

a) Facilidad de Uso. Antes de su versión 8, Direct3D era considerado poco intuitivo al momento de usarlo, esto debido a que requería el uso de un gran número de operaciones complejas. En general el modelo de Direct3D era realmente frustrante para muchos programadores, tan es así que se llegó a sugerir que Microsoft dejara de apoyar a Direct3D a favor de OpenGL.

   Sin embargo con la aparición de Direct3D versión 8, muchos cambios se realizaron, y Direct3D se convirtió en una API competente. Ahora Direct3D y OpenGL siguen paradigmas diferentes, mientras que el primero se basa en la tecnología COM (Component Object Model, también conocido como ActiveX), y en si mismo es orientado a objetos (por ejemplo, cada textura es un objeto), OpenGL no sigue tal paradigma, en lugar de ello es conocido como una máquina de estados a la cual se accede mediante simples funciones en lenguaje C. Debido a tales diferencias, el decidir cual es mejor, depende primordialmente de las preferencias del programador. La máquina de estados OpenGL puede ser envuelta por una interfaz de programación orientada a objetos como parte de un motor gráfico.

   Direct3D tiende a estar más ligado con el hardware de lo que está OpenGL.

Una de las maneras más comunes de utilizar OpenGL es con lo que se conoce como "*immediate mode*", el cual le permite al programador agregar vértices (o atributos de los vértices, tales como coordenadas de textura o color) uno por uno al buffer de vértices actual (ejemplo, glVertex3f(1.0f, 2.0f, 3.0f);), esto claro, con la correspondiente pérdida de velocidad. Esta es una manera conveniente de dibujar objetos sencillos, pero en la práctica esto proporciona una manera de dibujar notablemente lenta. Este mismo método fue adoptado en Direct3D 5, y después mejorada en DirectX 8 con la adición de Buffers de Vértices y de Índices, los cuales consistían en buffers de geometrías los cuales pueden ser almacenados de manera local en el hardware para un acceso más rápido por el GPU, con lo cual se prohíbe la creación de una copia de los datos por parte del usuario de la aplicación hacia el hardware. Además de lo anterior, Direct3D permite proporcionar datos de una manera inmediata mediante las instrucciones DrawPrimitiveUP/DrawIndexedPrimitiveUP y del uso de los buffers de Vértices y de Índices, por lo que ahora el programador puede elegir la manera de trabajar dependiendo de lo que requiera hacer.

OpenGL 1.1 introdujo los arreglos de vértices (los cuales su desempeño es mucho mejor que el "immediate mode" para una gran cantidad de geometrías), y más adelante, en una extensión de la versión, agregó "*vertex buffer objects*" (VBOs), los cuales funcionan de manera equivalente a los buffers de Vértices de Direct3D.

Otra diferencia significativa entre estas APIs es la manera en que manejan el dibujo de las texturas, Direct3D proporciona una manera conveniente para ello, la instrucción SetRenderTarget(), en OpenGL, al contrario, el programador tiene que hacer uso del buffer-P o de los buffer de los píxels para obtener el mismo resultado, y esto es considerado como uno de los puntos más irritables del API OpenGL. Esto debido a que si un programador crea su ruta de código de manera diferente a la que el fabricante del driver del hardware estipuló, estos cambios pueden acarrear problemas de compatibilidad, provocando errores al momento de que el software de dibujo se ocupe, haciendo a dicha aplicación inútil.

b) Velocidad. Poco después de que ambos, Direct3D y OpenGL, se convirtieran en librerías gráficas viables, Microsoft y SGI entablaron un estudio acerca de cual de las dos API ofrecía un desempeño superior, mucho de esto se debió a que, durante ese tiempo, ambos, Microsoft Direct3D y varias implementaciones de SGI OpenGL, eran relegadas como implementaciones de software para dibujo en el mercado de usuarios consumidores (esto debido a que el costo, en ese tiempo, del hardware para gráficas era muy alto). Entonces la habilidad de una librería específica para superar el desempeño de la otra, dependía única y exclusivamente de la librería misma y no en un factor externo, tal como el desempeño de las tarjetas de gráficos.

Microsoft, quien ha creado implementaciones de ambas API para su plataforma, ha dicho que Direct3D es mucho más rápido en su desempeño, esto resultado de la comparación de las API que el mismo Microsoft creó. En ese entonces, el reporte fue considerado como verdadero (utilizando ambas implementaciones de Microsoft), y el resultado del pobre desempeño de OpenGL se atribuyó a las rigurosas especificaciones y configuraciones requeridas por OpenGL. Esta percepción cambió con la conferencia de 1996 SIGGRAPH (Special Interest Group on Computer Graphics). En ese momento, SGI retó a Microsoft con sus propias implementaciones de software optimizadas para Windows de OpenGL, llamada CosmoGL, en la cual (mediante el uso de varios demos), se mostraba que en muchos casos el desempeño de OpenGL era igual, o mejor, que el de Direct3D. Para SGI esto demostraba que el pobre desempeño que había mostrado OpenGL no era ha causa de la estructura o diseño de OpenGL, sino debido a la falta de optimizaciones en la implementación de Microsoft.

c) Estructura. El API OpenGL, fue originalmente escrito para las entonces poderosas estaciones de trabajo de SGI e incluían un gran número de comandos que son raramente utilizados (usualmente en casos extremos de dibujo). El API está conformada por cerca de 250 llamadas a funciones, pero solo un subgrupo de cerca de 100 son útiles para un contexto de juegos de video. OpenGL nunca definió un subgrupo útil para estas situaciones, un grupo de comandos que deben de implementarse en el hardware para un mejor desempeño. MiniGL, lanzado por 3Dfx, fue una medida provisional para soportar glQuake, esto fue el punto de partida pero funciones adicionales fueron pronto adoptadas por los juegos, validando la inclusión de muchas significativas capacidades. El problema ha desaparecido conforme tarjetas de video modernas soportan en mayor medida el API en el hardware, y se han vuelto mucho más sofisticadas y ricas en características que la versión de OpenGL 1.1 o cualquier otra estación de trabajo de SGI.

Estructuralmente OpenGL fue diseñado como una API con muchas características, esto a pesar de la falta de soporte para hardware. Sin embargo una de las características más sobresalientes de OpenGL es que puede “actualizarse” mediante la implementación de extensiones. Un ejemplo de ello se presentó cuando la tarjeta gráfica GeForce 256 apareció en 1999, y juegos como Quake 3 Arena, podían utilizar las instrucciones de Transform&Lighting mediante extensiones en OpenGL, mientras que los desarrolladores en Direct3D tuvieron que esperar hasta la siguiente versión del API y volver a escribir el código de los juegos para poder aprovechar las características. Claramente la visión de OpenGL tiene sentido para las cosas esenciales que pueden ser realizadas mediante el software, sin embargo, otras características no tan esenciales pueden ser atrasadas hasta que el hardware que las soporte por completo exista. Sin embargo, ¿quién es el que puede decidir cuales características no son importantes?, y como ya se vio anteriormente, no sólo se debe medir el desempeño, sino tomar en cuenta otros factores, como por ejemplo, si se crean extensiones para mejorar el

desempeño en juegos de video, pero esas mismas extensiones perjudican el desempeño de aplicaciones de productividad. Direct3D en cambio, al ser controlado y no tener extensiones creadas por cada uno de los fabricantes, tiene la ventaja de que el fabricante puede revisar las características con las que se cuenta y no tener el riesgo de caer en la ruta equivocada, ya que hasta que el hardware sea compatible con ciertas características es cuando el API Direct3D será actualizado.

d) Portabilidad. En la mayor parte, Direct3D está atado a una sola plataforma: Microsoft Windows. Solo una porción del API Direct3D (y DirectX en general) ha sido implementado en WINE (proyecto en el cual se trata de permitir que máquinas con Unix/Linux ejecuten programas Windows), y el propio Microsoft proporcionó una versión modificada para la consola Xbox. Pero aún con estas excepciones, Direct3D está tan integrado con Windows que tratar de llevarlo a un sistema operativo No Windows es muy difícil. De hecho el mismo Microsoft una vez prometió distribuir una versión para la plataforma PowerPC en la cual se basa Apple Macintosh, pero después abandonó el proyecto. OpenGL, por otro lado, tiene implementaciones en una gran variedad de plataformas, entre las que se incluyen Windows, sistemas basados en UNIX, Linux, y la anteriormente mencionada Mac OS X. En general, todos los sistemas operativos modernos capaces de realizar presentaciones de gráficos en tercera dimensión, tienen un driver nativo de OpenGL, o utilizan algún software para su implementación.

   El punto es que las más grandes compañías en la creación de tarjetas aceleradores de tercera dimensión (ATI Technologies y nVidia) tienen drivers para OpenGL que son aceptables o excelentes para todas sus tarjetas modernas, tanto para plataformas Windows como No Windows.

2.1.3. Algoritmos para mejor desempeño y visualización.

2.1.3.1. Face Culling.

Este algoritmo es un caso particular del método de eliminación de superficies ocultas en el cual solo se eliminan los polígonos que están orientados hacia el lado opuesto del observador, para ello se compara la orientación de polígonos completos con el punto de vista o centro de la proyección y elimina los polígonos que no pueden ser vistos. Si una escena contiene un solo objeto convexo, esta operación generaliza al método de “eliminación de superficies ocultas” Un algoritmo de eliminación de superficies ocultas es siempre requerido cuando un polígono parcialmente oculta a otro.

En promedio la mitad de los polígonos de un poliedro están volteando hacia atrás y la ventaja de este proceso es que con una prueba muy simple se dejan de

considerar estos polígonos en otros algoritmos de eliminación de superficies ocultas más complicados. [3]

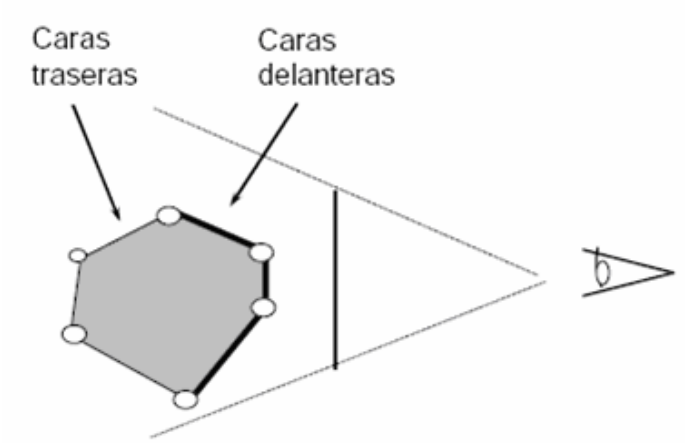


Fig 2.1.3.1.a. Eliminación de caras ocultas al observador.[22]

Lo primero que se necesita hallar es un vector ortogonal o normal al plano que define la cara que se está tratando. Para ello, se seleccionan dos segmentos consecutivos, y se multiplican vectorialmente, tomando sus coordenadas como coordenadas de vectores. Esto da las coordenadas de un vector que cumple la condición de perpendicularidad a los dos iniciales, sin embargo, como el producto vectorial no es conmutativo, por lo que, para que este vector apunte hacia el exterior de la figura, es necesario que los vértices de cada plano estén definidos en sentido antihorario.

Por lo tanto, para definir cada plano, se debe formar la figura de tal modo que el plano que se va a traducir a datos, este orientado hacia el observador.

Una vez que se tienen las coordenadas de estos vectores normales al plano, llega el momento de aplicar la prueba de visibilidad, para ello, es necesario obtener las coordenadas del vector que une el “ojo” del observador con un vértice cualquiera del plano a probar. Una vez hecho esto, se aplica el producto escalar de este vector y el normal a la superficie, y se obtiene un valor dependiente del ángulo que forman, si este valor es uno, entonces el vector apunta hacia el observador, y si es cero, será normal al vector que une el vértice con el “ojo” del observador. De aquí se deduce que, si el valor resultante es positivo, la cara está mirando hacia la parte delantera del objeto, o sea, hacia el observador, y entonces el polígono se debe dibujar.

Si el valor es negativo, la cara está mirando hacia atrás del objeto, así que no se pintará dicha cara. Si el valor es cero, la cara es normal al observador, y puede decidirse pintar o no la cara, ya que solo se vería como una línea.

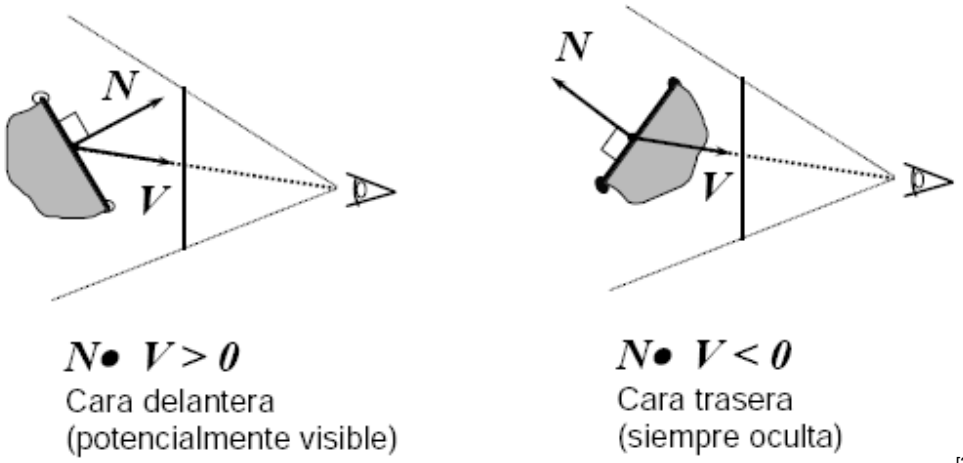


Fig 2.1.3.1.b. Caras de los polígonos, según su posición con respecto al observador. [22]

2.1.3.2. Buffer Z.

La técnica de buffer Z (también conocido como "buffer de profundidad") consiste en almacenar la profundidad de cada píxel cuando es dibujado. Esto por que al momento de realizar la última transformación, de pasar la información de información de un entorno 3D al ambiente bidimensional del dispositivo de visualización (monitor), toda la información de la profundidad se pierde. Solo queda la información de las coordenadas X, Y, que es la información de la posición en pantalla, con lo cual la coordenada Z se pierde.

Con la técnica de buffer Z se está llevando a cabo una clasificación de profundidades separadas para cada píxel en el área de pantalla, con lo cual se determina qué puntos situados sobre determinados polígonos están más cerca del espectador para cada píxel, de manera que solo se muestren dichos puntos.

Este método requiere que el programa configure a parte una serie entera en la cual cada elemento corresponda a un píxel que se visualiza, y cada vez que un punto de la superficie de un polígono se dibuje, la coordenada Z de dicho punto es colocada en el elemento de la serie que corresponde a la posición del píxel en el cual se ha dibujado el punto.

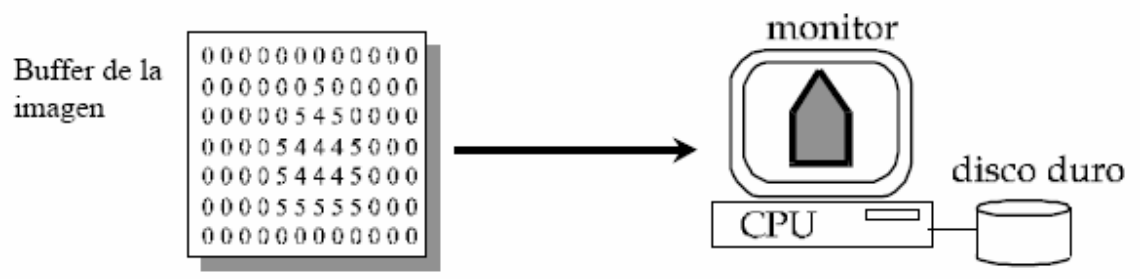


Fig. 2.1.3.2.a. Implementación de algoritmo Buffer Z.[23]

La siguiente vez que un píxel sea dibujado en la misma posición, la coordenada Z del punto representada por el nuevo píxel es comparada con el valor actual en la serie correspondiente a dicha posición. Si la coordenada Z en el buffer es menor que la del nuevo punto, el nuevo píxel no se dibuja, ya que dicho punto estaría más lejos que el punto anterior y formaría parte de una superficie oculta. Si la coordenada Z en el buffer es mayor que la del nuevo punto, entonces el nuevo píxel se dibuja sobre la anterior y la coordenada Z del nuevo píxel se coloca, reemplazando al anterior.

Si está perfectamente implementado, en el algoritmo de Buffer Z no aparecen superficies ocultas transparentadas. Sin embargo, presenta un flanco débil. En realidad se trata de dos: tiempo y memoria (por esta razón, esta función viene implementada en el hardware de las tarjetas gráficas actuales).

Se requieren dos áreas de memoria para implementar este método, el buffer de profundidad y el de renovación. El primero se utiliza para almacenar los valores de Z para cada posición X, Y a medida que se comparan las superficies y el segundo almacena los valores de intensidad de cada posición.

Este método es fácil de implementar, pero si disponemos de un sistema cuya resolución es de 1024 x 768 píxels, se requieren más de un millón de posiciones en el buffer de profundidad.

Otro de los inconvenientes de utilizar este algoritmo es que se debe tener cuidado de que el hardware sea el adecuado para la información que se maneja, Esto se debe a que las tarjetas gráficas llegan a tener buffers Z de 16, 24, o el mejor de los casos, 32 bits. Esto específica la limitante de valores a los cuales se puede asignar profundidad, es decir, en cierto hardware se pueden tener resultados diferentes a los esperados.

2.1.3.3. Frustum.

El Frustum es el campo de visión. Consiste en una especie de pirámide que abarca todo lo que es visible, entendiendo como visible todos aquellos polígonos que se verían en pantalla. Esta pirámide se representa mediante seis planos infinitos, y todo lo que se encuentra dentro de esos seis planos es visible. [11]

El trabajar con el Frustrum requiere que se conozcan los límites del mismo, es decir, los valores de los planos que lo componen, o por lo menos los valores de intersección de los mismos. Ya con estos valores es posible realizar comparaciones con los puntos que forman a los polígonos, con lo cual se determina si dichos puntos son dibujados.

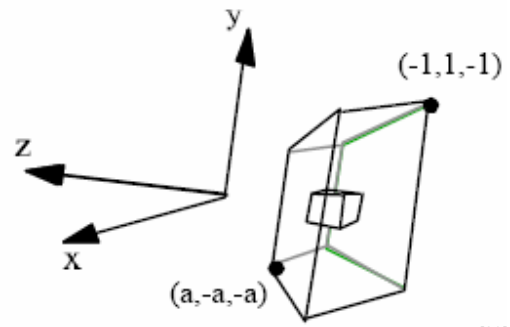


Fig. 2.1.3.3.a. Frustum en el espacio 3D. [20]

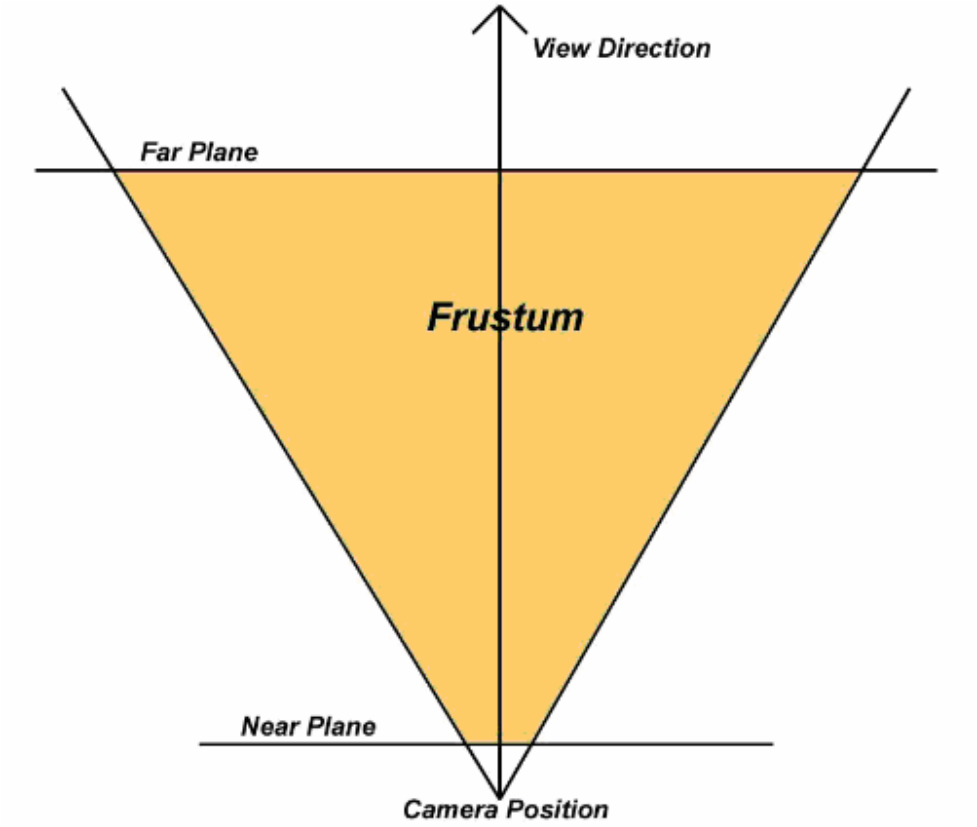


Fig. 2.1.3.3.b. Representación del Frustum en el espacio en 2D.[21]

El primer inconveniente se centra en la obtención de los valores que forman al Frustum, y en seguida aparece la gran cantidad de comparaciones necesarias para llevar a cabo este algoritmo.

Afortunadamente las API's actuales, resuelven esto al permitir que el programador determine el Frustum que desea, así cómo de llevar un registro de los valores del Frustum, en caso de que estos sean modificados en tiempo de ejecución. OpenGL realiza el algoritmo de Frustum de forma automática con solo activar la opción para que esto sea así.

2.2. Programación Orientada a Objetos.

2.2.1. Teoría de objetos.

El término de Programación Orientada a Objetos (POO) indica más una forma de diseño y una metodología de desarrollo de software que un lenguaje de programación, ya que en realidad se puede aplicar el Diseño Orientado a Objetos (En inglés abreviado OOD, Object Oriented Design), a cualquier tipo de lenguaje de programación. El desarrollo de la POO empieza a destacar durante la década de los 80's tomando en cuenta la programación estructurada, a la que engloba, y dotando al programador de nuevos elementos para el análisis y desarrollo de software. [10]

Básicamente la POO permite a los programadores escribir software, de forma que esté organizado en la misma manera que el problema que trata de modelar. Los lenguajes de programación convencionales son poco más que una lista de acciones a realizar sobre un conjunto de datos en una determinada secuencia. Si en algún punto del programa modificamos la estructura de los datos o la acción realizada sobre ellos, el programa cambia.

La POO aporta un nuevo enfoque, convirtiendo la estructura de datos en el centro sobre el que las operaciones actúan. De esta forma, cualquier modificación de la estructura de datos tiene efecto inmediato sobre las acciones a realizar sobre ella, siendo esta una de las diferencias radicales respecto a la programación estructurada.

Se han oído muchos comentarios sobre la incidencia de la POO sobre la programación convencional. Se ha llegado a decir que el cambio introducido por la POO es similar al producido por la aparición del ensamblador sobre el código máquina.

La POO proporciona las siguientes ventajas sobre otros lenguajes de programación:

a) Uniformidad**.** Ya que la representación de los objetos lleva implícita tanto el análisis como el diseño y la codificación de los mismos.

b) Comprensión**.** Tanto los datos que componen los objetos, como los procedimientos que los manipulan, están agrupados en clases, que se corresponden con la estructuras de información que el programa trata.

c) Flexibilidad**.** Al tener relacionados los procedimientos que manipulan los datos con los datos a tratar, cualquier cambio que se realice sobre

ellos quedará reflejado automáticamente en cualquier lugar donde estos datos aparezcan.

d) Estabilidad**.** Dado que permite un tratamiento diferenciado de aquellos objetos que permanecen constantes en el tiempo, sobre aquellos que cambian con frecuencia permite aislar las partes del programa que permanecen inalterables en el tiempo.

e) Reusabilidad**.** La noción de objeto permite que programas que traten las mismas estructuras de información reutilicen las definiciones de objetos empleados en otros programas e incluso los procedimientos que los manipulan. De esta forma, el desarrollo de un programa puede llegar a ser una simple combinación de objetos ya definidos, donde estos están relacionados de una manera particular.

Uno de estos puntos claves a remarcar en esta introducción, es que la programación orientada a objetos no sustituye a ninguna metodología ni lenguaje de programación anterior. Todos los programas que se realizan según la POO, se pueden realizar igualmente mediante programación estructurada. Su uso en la actualidad se justifica porque el desarrollo de todas las nuevas herramientas basadas en un interfaz gráfica de usuario como Windows, OS/2, X-Windows, Mac OS, etc., es mucho más sencilla.

2.2.2. Encapsulación y ocultación de datos.

La capacidad de presentación de información dentro de un objeto se divide en dos partes bien diferenciadas:

a) Interna. La información que necesita el objeto para operar y que es innecesaria para los demás objetos de la aplicación. Estos atributos se denominan privados y tienen como marco de aplicación únicamente a las operaciones asociadas al objeto.

b) Externa**.** La que necesitan el resto de los objetos para interactuar con el objeto que definimos. Estas propiedades se denominan públicas y corresponde a la información que necesitan conocer los restantes objetos de la aplicación respecto del objeto definido para poder operar.

Podemos imaginar la encapsulación como introducir el objeto dentro de una caja negra donde existen dos ranuras denominadas entrada y salida. Si introducimos datos por la entrada automáticamente obtendrá un resultado en la salida. No se necesita conocer ningún detalle del funcionamiento interno de la caja. [10]

El término encapsulación indica la capacidad que tienen los objetos de construir una cápsula a su alrededor, ocultando la información que contienen (aquélla que es necesaria para su funcionamiento interno, pero innecesaria para los demás objetos) a las otras clases que componen la aplicación.

Aunque a primera vista la encapsulación puede parecer superflua, tengamos en cuenta que existen muchas variables utilizadas de forma temporal: contadores y variables que contienen resultados intermedios, etc. De no ser por la encapsulación estas variables ocuparían memoria y podrían interferir en el funcionamiento del resto de los objetos.

Los lenguajes POO incorporan la posibilidad de encapsular también las estructuras de datos que sirven como base a las funciones. Aportan por tanto un nivel superior en cuanto a protección de información.

La encapsulación nos permite el uso de librerías de objetos para el desarrollo de programas. Las librerías son definiciones de objetos de propósito general que se incorporan a los programas. Al ser el objeto parcialmente independiente en su funcionamiento del programa en donde está definido, ya que contiene y define todo lo que necesita para poder funcionar, es fácil utilizarlo en los más variados tipos de aplicaciones. Si aseguramos, depurando las propiedades y las operaciones dentro de la clase que el objeto funcionó bien dentro de una aplicación, con una correcta encapsulación el objeto podrá funcionar en cualquier otra.

Otra de las ventajas de la encapsulación es que, al definir el objeto como una caja negra con entradas y salida asociadas, en cualquier momento podemos cambiar el contenido de las operaciones del objeto, de manera que no afecte al funcionamiento general del programa.

La encapsulación está en el núcleo de dos grandes pilares de la construcción de sistemas; mantenibilidad y reusabilidad.

a) Mantenibilidad. Cualidad que indica que un programa o sistema debe ser fácilmente modificable. Es decir que los cambios en las condiciones externas (como la definición de una nueva variable) implicarán modificaciones pequeñas en el programa / sistema. El concepto de mantenibilidad implica que un programa, al igual que un ser vivo, debe ser capaz de adaptarse a un medio ambiente siempre cambiante.

b) Reusabilidad. Cualidad que nos indica que partes del programa (en este caso objetos) pueden ser reutilizados en la confección de otros programas. Ello implica que los objetos definidos en un programa pueden ser extraídos del mismo e implantados en otro, sin tener que realizar modificaciones importantes en el código del objeto. El objetivo final es que el programador construya una librería de objetos que le

permita realizar programas basándose en la técnica de cortar y pegar. Este extrae (corta) código de otras aplicaciones ya realizadas y las implementa (pega) en la aplicación a realizar donde, tras algunos retoques, la nueva aplicación estará lista para funcionar. Como se podrá observar el concepto de reusabilidad, permite reducir el tiempo de realización, ganando en claridad, mantenibilidad y productividad.

La encapsulación de datos se muestra como una herramienta poderosa que nos permite ganar en tiempo de desarrollo y claridad, con el único costo adicional de definir con precisión las entradas y salida de nuestras operaciones.

En resumen puede decirse que la encapsulación consiste en separar aquellos atributos del objeto que deben ser conocidos, por el resto de aquellos necesarios para su funcionamiento propio. No es propio de los lenguajes orientados a objetos, pero la capacidad de éstos para unir las estructuras de datos a los procedimientos que los modifican lo hacen más potente que los lenguajes llamados “modulares”

2.2.3. Polimorfismo.

El polimorfismo es una nueva característica aportada por la POO. Esta propiedad indica la posibilidad de definir varias operaciones con el mismo nombre, diferenciándolas únicamente en los parámetros de entrada. Dependiendo del objeto que se introduzca como parámetro de entrada, se elegirá automáticamente cual de las operaciones se va a realizar. [10]

Una de las ventajas más importantes, sin entrar en la redefinición de operadores es permitir la realización de las clases que definen un programa de forma totalmente independiente al programa donde se utilizan. Gracias a la encapsulación y el polimorfismo, aunque se utilicen los mismos nombres con las operaciones en dos clases distintas, el programa reconoce a que clase se aplica durante la ejecución.

Como se podrá observar el polimorfismo y la encapsulación de datos están íntimamente ligados y nos permiten un mayor grado de mantenibilidad y reusabilidad que los lenguajes tradicionales. Esta es precisamente una de las causas de la revolución que ha supuesto la introducción de los lenguajes orientados a objetos dentro de la programación.

2.2.4. Herencia.

La herencia es la última de las propiedades relativas a la POO. Consiste en la propagación de los atributos y las operaciones a través de distintas subclases definidas a partir de una clase común. [10]

Introduce, por tanto, una posibilidad de refinamiento sucesivo del concepto de clase. Nos permite definir una clase principal y, a través de sucesivas aproximaciones, cualquier característica de los objetos. A partir de ahora se definirá como subclases todas aquellas clases obtenidas mediante refinamiento de una (o varias) clases principales.

La herencia nos permite crear estructuras jerárquicas de clases, donde es posible la creación de subclases que incluyan nuevas propiedades y atributos. Estas subclases admiten la definición de nuevos atributos, así como crear, modificar o inhabilitar propiedades.

Además, es posible que una subclase herede atributos y propiedades de más de una clase. Este proceso se denomina herencia múltiple. La herencia es, sin duda alguna, una de las propiedades más importantes de la POO, ya que permite, a través de la definición de una clase básica, ir añadiendo propiedades a medida que sean necesarias y, además, en el subconjunto de objetos que sea preciso.

La herencia permite que los objetos puedan compartir datos y comportamientos a través de las diferentes subclases, sin incurrir en redundancia. Más importante que el ahorro de código, es la claridad que aporta al identificar que las distintas operaciones sobre los objetos.

Identidad, clasificación, polimorfismo y herencia caracterizan a los lenguajes orientados a objetos. Cada uno de estos conceptos puede utilizarse aisladamente, incluso aparecen en otras metodologías de programación, pero juntos se complementan en una relación sinérgica. Los beneficios de la programación orientada a objetos son más que los que pueden verse a simple vista. El énfasis en las propiedades esenciales de un objeto, fuerza al desarrollador a pensar cuidadosamente que es un objeto y que es lo que hace con el resultado. Lo que propicia que el sistema sea normalmente más preciso, general y robusto que si pusiéramos el énfasis en los procedimientos y los datos por separado.

2.3. Estructura de Datos.

2.3.1. Estructuras Autorreferenciadas.

En programación, una estructura de datos es una forma de organizar un conjunto de datos elementales (un dato elemental es la mínima información que se tiene en el sistema) con el objeto de facilitar la manipulación de estos datos como un todo y/o individualmente. Una estructura de datos define la organización e interrelacionamiento de estos, y un conjunto de operaciones que se pueden realizar sobre él.

Diferentes tipos de estructuras de datos son adecuadas para diferentes tipos de aplicaciones, y algunas son altamente especializadas para realizar tareas concretas. Por ejemplo, los *Árboles-B*, son particularmente adecuados para la implementación de bases de datos, mientras que las *tablas de ruteo o de rutas* se usan en gran medida en redes de cómputo.

En el diseño de muchos tipos de programas, la elección de la correcta estructura de datos es la primera consideración de diseño, debido a que la experiencia ha demostrado que al construir grandes sistemas la dificultad en la implementación y la calidad y desempeño, dependen en gran medida en la elección adecuada de la mejor estructura de datos. Después de que la estructura de datos se ha seleccionado, los algoritmos que serán usados son relativamente ya obvios. Aunque hay que recalcar que en ocasiones las cosas trabajan de forma inversa, las estructuras de datos se seleccionan debido a que cierta labor esencial tiene algoritmos que trabajan mejor con cierta estructura de datos en particular. En cualquiera de los caso, la correcta elección de la estructura de datos a utilizar es una decisión crucial. [4]

Sobre las estructuras de datos se llevan acabo operaciones, las operaciones básicas son tres:

a) Agregar, adicionar un nuevo valor a la estructura.

b) Borrar, eliminar un valor de la estructura.

c) Buscar, encontrar un determinado valor en la estructura para realizar una operación con este valor.

Las anteriores fueron las operaciones básicas, sin embargo, otras operaciones importantes son las siguientes:

a) Ordenamiento, de los elementos pertenecientes a la estructura.

b) Unión, dadas dos estructuras originar una nueva ordenada y que contenga las dos estructuras originales.

Algunas de las estructuras de datos utilizadas en programación son:

a) Arreglos.
b) Listas.
c) Pilas.
d) Colas.
e) Árboles.
f) Grafos.
g) Tablas Hash.

Para propósitos de este trabajo se analizarán las estructuras mediante el uso de memoria dinámica.

Una de las aplicaciones más interesantes y potentes de la memoria dinámica y los apuntadores son las estructuras dinámicas de datos. Las estructuras básicas disponibles en C y C++ tienen una importante limitación: no pueden cambiar de tamaño durante la ejecución. Los arreglos están compuestos por un determinado número de elementos, número que se decide en la fase de diseño, antes de que el programa ejecutable sea creado.

En muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Por supuesto, se pueden hacer arreglos dinámicos, pero una vez creados, su tamaño también será fijo, y para hacer que crezcan o diminuyan de tamaño, se deberá de reconstruirlos desde el principio.

Las estructuras dinámicas nos permiten crear estructuras de datos que se adapten a las necesidades reales a las que suelen enfrentarse nuestros programas. Pero no sólo eso, también nos permiten crear estructuras de datos muy flexibles, ya sea en cuanto al orden, la estructura interna o las relaciones entre los elementos que las componen.

Las estructuras de datos están compuestas de otras pequeñas estructuras a las cuales se les llama nodos o elementos, que agrupan los datos con los que trabajará nuestro programa y además uno o más apuntadores autoreferenciales, es decir, apuntadores a objetos del mismo tipo nodo.

Una estructura básica de un nodo para crear listas de datos sería:

```
struct nodo {
   int dato;
   struct nodo *otronodo;
};
```

El campo "otronodo" puede apuntar a un objeto del tipo nodo. De este modo, cada nodo puede usarse como un ladrillo para construir listas de datos, y cada uno mantendrá ciertas relaciones con otros nodos.

Para acceder a un nodo de la estructura sólo necesitaremos un apuntador a un nodo.

El nodo anterior se representará así:

Las estructuras dinámicas son una implementación de TDAs o TADs (Tipos Abstractos de Datos). En estos tipos el interés se centra más en la estructura de los datos que en el tipo concreto de información que almacenan.

Dependiendo del número de apuntadores y de las relaciones entre nodos, se pueden distinguir varios tipos de estructuras dinámicas. Se enumeran enseguida sólo de los tipos básicos:

a) Listas abiertas o ligadas. Cada elemento sólo dispone de un apuntador, que apuntará al siguiente elemento de la lista o valdrá NULL si es el último elemento.

b) Pila. Son un tipo especial de lista, conocidas como listas LIFO (Last In, First Out: el último en entrar es el primero en salir). Los elementos se "amontonan" o apilan, de modo que sólo el elemento que está encima de la pila puede ser leído, y sólo pueden añadirse elementos encima de la pila.

c) Colas. Otro tipo de listas, conocidas como listas FIFO (First In, First Out: El primero en entrar es el primero en salir). Los elementos se almacenan en fila, pero sólo pueden añadirse por un extremo y leerse por el otro.

d) Listas circulares. O listas cerradas, son parecidas a las listas ligadas, pero el último elemento apunta al primero. De hecho, en las listas circulares no puede hablarse de "primero" ni de "último". Cualquier nodo puede ser el nodo de entrada y salida.

e) Listas doblemente enlazadas. Cada elemento dispone de dos apuntadores, uno apunta al siguiente elemento y el otro al elemento anterior. Al contrario que las listas abiertas anteriores, estas listas pueden recorrerse en los dos sentidos.

f) Árboles. Cada elemento dispone de dos o más apuntadores, pero las referencias nunca son a elementos anteriores, de modo que la estructura se ramifica y crece igual que un árbol.

g) Árboles binarios. Son árboles donde cada nodo sólo puede apuntar a dos nodos.

h) Árboles binarios de búsqueda (ABB). Son árboles binarios ordenados. Desde cada nodo todos los nodos de una rama serán mayores, según la norma que se haya seguido para ordenar el árbol, y los de la otra rama serán menores.

i) Árboles AVL. Son también árboles de búsqueda, pero su estructura está más optimizada para reducir los tiempos de búsqueda.

j) Árboles B. Son estructuras más complejas, aunque también se trata de árboles de búsqueda, están mucho más optimizados que los anteriores.

k) Tablas HASH. Son estructuras auxiliares para ordenar listas.

l) Grafos. Es el siguiente nivel de complejidad, podemos considerar estas estructuras como árboles no jerarquizados.

m) Diccionarios.

2.3.2 Listas Ligadas y doblemente ligadas.

La forma más simple de estructura dinámica es la *Lista Ligada*. En esta forma los nodos se organizan de modo que cada uno apunta al siguiente, y el último no apunta a nada, es decir, el apuntador del nodo siguiente vale NULL.

En las listas ligadas existe un nodo especial: el primero. Normalmente se dice que la lista es un apuntador a ese primer nodo y se llamará a ese nodo la *cabeza de la lista*. Eso es porque mediante ese único apuntador se puede acceder a toda la lista.

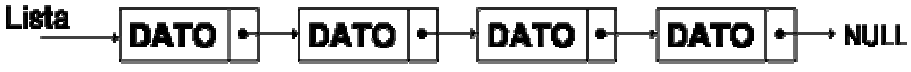
Cuando el apuntador que se usa para acceder a la lista vale NULL, se dice que la lista está vacía.

El nodo típico para construir listas tiene esta forma:

```
struct nodo {
   int dato;
   struct nodo *siguiente;
};
```

En el ejemplo anterior, el elemento de la lista sólo contiene un dato de tipo entero (int), pero en la práctica no hay límite en cuanto a la complejidad de los datos a almacenar.

Así es como se ve una lista ligada:



Es muy importante que el programa nunca pierda el valor del apuntador al primer elemento, ya que si no existe ninguna copia de ese valor, y se pierde, será imposible acceder al nodo y no se podrá liberar el espacio de memoria que ocupa.

Con las listas se tiene un pequeño repertorio de operaciones básicas que se pueden utilizar:

a) Añadir o insertar elementos.
b) Buscar o localizar elementos.
c) Borrar elementos.

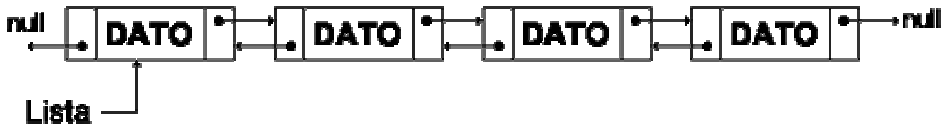
Cada una de estas operaciones tendrá varios casos especiales, por ejemplo, no será lo mismo insertar un nodo en una lista vacía, que al principio de una lista no vacía.

Una lista doblemente ligada es una lista ligada en la que cada uno de sus nodos tiene dos enlaces, uno al nodo siguiente, y el otro al anterior. Las listas doblemente ligadas no requieren de un nodo especial para acceder a ellas, pueden recorrerse en ambos sentidos a partir de cualquier nodo, esto es porque a partir de cualquier nodo, siempre es posible alcanzar cualquier nodo de la lista, hasta que se llega a uno de los extremos.

El nodo típico es el mismo que para construir las listas ligadas, salvo que tienen otro apuntador al nodo anterior:

```
struct nodo {
   int dato;
   struct nodo *siguiente;
   struct nodo *anterior;
};
```

Las listas doblemente ligas se ven de la siguiente manera:



También es posible crear listas doblemente ligadas circulares.

El movimiento a través de listas doblemente enlazadas es más sencillo, y las operaciones de búsqueda, inserción y borrado, también tienen más ventajas.

2.4. Comunicación entre procesos.

2.4.1. Modelo OSI.

La organización internacional de estandarización (The International Organization for Standardization - ISO) desarrollo un modelo para la comunicación de datos en la década de los 70's, llamado el modelo de Sistemas Abiertos de Interconexión (Open Systems Interconnection - OSI). Este modelo es el estándar para todas las comunicaciones de datos, además de que también define los elementos esenciales de otros estándares. [5]

El modelo está basado en siete capas, cada una de ellas tiene su propia y única área de proceso de comunicación de datos. Estas capas están conectadas entre sí una por una. Entre cada par de capas existe una interfaz que es usada por cada una de las capas para trabajar con cada una de las otras.

OSI define las funciones y las interfaces de cada capa del modelo. OSI no define cómo son creadas las capas. Esto hace que sea fácil la creación de nuevas soluciones de comunicación de datos basadas en el modelo OSI. Pero lo más importante es que lo anterior hace la fácil modificación de aquellas soluciones ya existentes. Debido a esto, tenemos la habilidad de elegir de entre muy diferentes opciones como actuar sobre una capa. Un muy buen ejemplo son los protocolos TCP y UDP. Estos dos protocolos pertenecen a la capa de transporte, y se puede elegir cuál de ellos usar para las aplicaciones dependiendo de las necesidades que requieren cubrir.

7	Aplicación
6	Presentación
5	Sesión
4	Transporte
3	Red
2	Enlace
1	Física

Tabla 2.3.1.1. Modelo OSI capas su orden.

Un protocolo de red es el lenguaje que dos o más computadoras usan cuando comparten información una con la otra. Como es lógico, todas las computadoras deben comunicarse con el mismo lenguaje para que la información fluya entre ellas. Si dos computadoras emplean lenguajes diferentes no se entenderán. Por lo que no hay información que sea tratada entre ellas.

Un protocolo define las reglas de comunicación y el formato de los datos. Los protocolos son estándares que trabajan sobre diferentes tipos de

computadoras. Lo cual quiere decir que si dos computadoras pueden ser conectadas de manera física dentro de una red, estas computadoras pueden tratar con información sin importar que sistema operativo están corriendo o de que tipo de computadoras se trata ya que están usando el mismo protocolo.

Las funciones de los protocolos son las siguientes:

a) Encapsulamiento. Se refiere a la adición de cabeceras en cada etapa con el propósito de identificar a los datos. Tiene niveles y no pueden ser saltados.

b) Detección de errores. Se refiere al hecho de saber si la información no sufrió modificaciones.

c) Segmentación y reensamblado. Se refiere a los paquetes de información que deben de ser preparados para el siguiente nivel.

d) Control de conexión. Secuenciado, control de flujo y control de errores.

e) Transmisión punto a punto. La red determina donde debe de dirigirse la información.

f) Ruteo. Se refiere a buscar el mejor camino, es realizado por los router.

2.4.2. Protocolo IP.

El protocolo usado para la comunicación en Internet es un protocolo llamado TCP/IP. Este nombre indica que dicho protocolo está dividido dentro de dos capas. Estas capas son parte del modelo OSI. TCP (Transmisión Control Protocol) es parte de la capa de transporte del modelo OSI, e IP (Internet Protocol) es parte de la capa de red. [5]

El modelo OSI se encuentra dividido en capas debido a que se realizan las mejoras de una manera más fácil, además de que pueden ser sustituidas las capas por otras mejores. Esto se ve reflejado en el hecho de que se pueden cambiar las tarjetas de interfaz de red (network interface card - NIC), controladores, o el tipo de conexión de red, o desarrollar un mejor protocolo.

Para algunas personas resulta insuficiente la tecnología que usa, lo cual permite que haya mejoras a dichas tecnologías, por ejemplo en la actualidad la versión del Protocolo de Internet es IPv4, que fue desarrollado en la década de los 70’s, pero la próxima publicación de la versión de este protocolo será IPv6. Esto en razón de que los números de las direcciones IP libres llegarán a ser insuficientes.

Las direcciones IPv4 están formadas por cuatro bytes separados por un punto, por ejemplo, 192.56.5.1. Debido a que las personas recuerdan de mejor manera nombres que números, a las direcciones IP se les dan nombres, como www.correo.unam.mx. Un DNS (Domain Name System) toma en cuenta el nombre y el número de la dirección IP correspondiente.

El espacio total de direcciones de IPv4 es alrededor de 4 billones de direcciones. Este es un gran número, pero no serán suficientes para las direcciones IP en el futuro.

La solución para este problema es IPv6. Entre otras mejoras, IPv6 provee un nuevo sistema de direcciones de 128 bits que incrementa el espacio de direcciones de manera sustancial. IPv6 fue desarrollado en la década de los 90's, es compatible con IPv4 hasta cierto punto.

2.4.3. Protocolo TCP.

TCP (Transmission Control Protocol) es un protocolo orientado a la conexión. Lo cual quiere decir que cada vez se quiera comunicar con un host remoto, se debe primero establecer una conexión. Una vez que se haya establecido la conexión, no hay que preocuparse acerca del direccionamiento de los mensajes que se envían al lugar correcto. TCP es también un protocolo fiable.

TCP transfiere un flujo de bytes de manera continua a través de Internet. TCP agrupa los bytes en segmentos y decide la manera en que lo hace y los envía del modo que más le convenga. [5]

TCP asigna un número de secuencia a cada byte transmitido. Con éste se asegura que el otro extremo recibe los mensajes que le fueron enviados, y además de que también se ocupa de los paquetes duplicados. El host receptor utiliza los números de secuencia para organizar los segmentos cuando llegan fuera de orden.

Algunas veces cuando un paquete es mandado a la red, este puede perderse durante algún tiempo. Esto puede suceder, por ejemplo, si un router está presentando algunos problemas. El paquete se pega al router así el host receptor nunca envía una confirmación de que el paquete ha alcanzado su destino. El host emisor asume que el paquete está perdido para siempre y lo retransmite después de un cierto tiempo, esto lo hace después de que ha transcurrido un intervalo de tiempo dentro del cual debe recibir la confirmación del host receptor. Este nuevo paquete puede encontrar alguna otra ruta al host receptor mientras el paquete original está aún pegado al router. Si el router empieza a trabajar normalmente otra vez, éste encamina los paquetes al lugar correcto y el host receptor puede recibir el mismo paquete otra vez. De esta manera este paquete se convierte en

un duplicado; TCP lo nota y destruye al paquete, dado que este ha dejado de ser necesario.

TCP se encarga también de que haya flujo de información en ambos sentidos de la conexión, es decir, usa conexiones Full-Duplex. Además usa la conexión punto a punto.

Si el programa de aplicación genera bloques de datos muy largos, el software de protocolo puede dividir cada bloque en partes más pequeñas para su transmisión.

Hay dos límites para el tamaño de los segmentos. Uno es el máximo tamaño de un paquete IP, que es de 64 Kbytes, y el otro la Unidad de Transferencia Máxima (MTU) de cada red.

Este protocolo se describe en el RFC- 793 Transmission Control Protocol.

2.4.4. Protocolo UDP.

Para casos en los cuales no es necesario contar con un control estricto de los datos que son enviados a través de una red la capa de transporte implanta el Protocolo de Datagrama de Usuario.

UDP (User Datagram Protocol) es, al contrario del TCP, un protocolo orientado a la no conexión y por lo tanto no proporciona ningún tipo de control de errores ni de flujo, ya que se supone de antemano que un mensaje aislado no creará problemas de congestión; aunque si utiliza un mecanismo para la detección de errores. [5]

En este caso los paquetes enviados por el protocolo IP reciben el nombre específico de datagramas. Cuando se detecta un error en un datagrama en lugar de ser entregado a la aplicación se descarta. Se dice que es simple, eficiente e ideal para aplicaciones como el TFTP (Trivial File Transfer Protocol) y el DNS (Domain Name System), SNMP (Simple Network Management Protocol), NTP (Network Time Protocol), NFS (Network File System), etc.

No hay conexión establecida entre los dos hosts de comunicación. El host emisor debe definir la dirección objetivo cada vez que envíe algo, por la razón de que los datagramas son encaminados de manera independiente. Cuando un host recibe un mensaje, el host sabe de dónde proviene el mensaje.

UDP no es confiable, porque no asegura que la transmisión es recibida. UDP tampoco se encarga de la duplicidad, pero en este protocolo los duplicados no son tan comunes porque el host emisor no retransmitirá nada si el mensaje no es recibido.

UDP puede ser usado en aplicaciones que necesitan la mejor eficiencia posible pero con muy poca confiabilidad, en otras palabras este protocolo es empleado en aquellos casos donde la entrega rápida de información es más importante que una entrega garantizada o cuando la información puede ser enviada en un solo datagrama.

UDP no admite enumeración de datagramas por lo que la garantía de que un datagrama llegué a su destino es menor a que si se emplea TCP para su envío. Lo cual puede originar que los datagramas lleguen desordenados a su destino.

Su utilización es adecuada cuando se requiere transmitir información en modo multicast, es decir, a muchos destinos; o en modo broadcast, o lo que es lo mismo a todos los destinos; pues no tiene sentido esperar la confirmación de todos los destinos para continuar con la transmisión. También es importante tener en cuenta que si en una transmisión de este tipo los host destinos enviarán confirmación, fácilmente el host emisor se vería colapsado, pues por cada paquete que envía recibiría tantas confirmaciones como host destinos hayan recibido el paquete.

Al igual que TCP, UDP usa al protocolo IP para transportar sus Segmentos.

Este protocolo es descrito en el RFC- 768 User Datagram Protocol.

2.4.5. Sockets.

Los sockets son interfaces lógicos de entrada y salida que permiten la comunicación bidireccional entre procesos que pueden estarse ejecutando en una misma máquina o en máquinas distintas pero que están conectadas en red. Para que dos programas se comuniquen entre si dentro de una red, se requieren por lo menos dos sockets, uno en una máquina y el otro en el sistema remoto. De esta manera se puede suponer que uno de los sockets está actuando como servidor, y se encuentra esperando conexiones por parte de clientes. Y es precisamente el otro socket el que se conectará al socket servidor. [5]

Por ejemplo los navegadores web se comunican con los servidores web mediante sockets.

La comunicación mediante sockets es el mecanismo usado por las aplicaciones en red que se basan en tecnología Internet, es decir, en la familia de protocolos TCP/IP. En las aplicaciones en red es de suma importancia distinguir los recursos locales de los recursos remotos. Por ejemplo, el navegador web prepara un socket local (en la máquina desde la que se desea navegar) para comunicarse con el socket remoto que el servidor web tiene disponible.

Desde el punto de vista de un proceso un socket es un recurso parecido a un archivo que es solicitado localmente al sistema operativo y que de alguna forma establece una puerta hacia el exterior y de esta manera permitir el intercambio de información.

Un socket realiza 7 operaciones básicas:

1. Conectarse a un host remoto.
2. Enviar datos.
3. Recibir datos.
4. Cerrar la Conexión.
5. Acoplarse a un puerto.
6. Esperar conexiones.
7. Aceptar conexiones de host remotos.

En una red una máquina se distingue de manera única de las demás mediante su dirección de nivel de red, es decir, su dirección IP.

Los protocolos del nivel de transporte como lo son el TCP y UDP incorporan un mecanismo de direccionamiento que permite diferenciar los sockets de una cierta máquina. Esto es, un socket que reside en una cierta máquina se distingue de manera única de los demás sockets de la misma máquina mediante su dirección de nivel de transporte, es decir, lo que habitualmente se conoce como puerto. De esto se desprende el hecho de que se hable de puertos TCP y puertos UDP dependiendo del protocolo que se esté empleando.

Cuando se está conectado a un servidor, éste generalmente proporciona gran variedad de servicios, para una aplicación en particular se podría requerir de uno de esos servicios. Cuando un cliente se encuentra conectado a un servidor, el cliente no va a estar verificando cada uno de los servicios que proporciona el servidor para determinar cuál de ellos es el que requiere, ya que esto se ve reflejado en el tiempo que debe de emplear para dicha tarea, lo cual resulta ser ineficiente. Por lo tanto se debe de proporcionar a cada servicio que proporciona el servidor un número que lo identifique, el cual debe ser definido cuando se pretenda realizar la conexión por parte del cliente con el servidor. Es precisamente a estos números a los que se les denomina número de puerto.

En conclusión los números de puerto permiten distinguir los procesos relacionados con la red dentro de una misma máquina.

Por lo tanto un socket se identifica indicando la máquina en la que reside, por medio de una dirección IP, y el puerto al que está asociado.

Existen direcciones IP especiales, que permiten que aquellas máquinas con la capacidad de comunicación TCP/IP respondan, por ejemplo, a la dirección 127.0.0.1 aunque no se encuentre conectada la red. Esto permite que se

establezca comunicación entre procesos locales mediante socktes sin necesidad de que el equipo posea una dirección IP real.

Existen dos tipos de sockets: los sockets TCP y los UDP.

a) Sockets TCP, "stream" u "orientados a la conexión". La comunicación entre los sockets TCP es confiable, dado que no hay pérdida de paquetes; además de ser ordenada, esto es que el protocolo TCP proporciona a los paquetes un número de secuencia; y se da sin duplicaciones de los paquetes. Para que dos sockets puedan comunicarse debe de establecerse un circuito, lo que se le denomina conexión, a través del cual se produzca el intercambio de información, ya sea envío o recepción de datos.

b) Sockets UDP, "datagrama" o "no orientados a la conexión". La comunicación entre sockets UDP no garantiza la llegada de todos los datagramas, ni su orden, ni su unicidad, es decir, la no duplicidad de datagramas. La aplicación que use este tipo de sockets tendrá que encargarse de dar solución a estos problemas. Para la comunicación entre sockets UDP no se establece una conexión, si no que se realiza un intercambio de datagramas, es decir, de paquetes sueltos.

2.4.6. Esquema Cliente /Servidor.

Las comunicaciones basadas en el protocolo TCP/IP se basan básicamente por lo que se llama Esquema Cliente/Servidor.

El término Cliente/Servidor fue usado por primera vez en 1980 para referirse a las computadoras personales conectadas en red. Este esquema comenzó a ser aceptado a finales de la década de los 80's. [5]

De manera general la forma en la que opera este esquema es el siguiente: se tiene una máquina cliente, que requiere algún servicio de una máquina servidor, y éste realiza la función que se solicita. Una computadora puede actuar como cliente y a la vez servidor.

En el esquema Cliente/Servidor, éste se define como una arquitectura distribuida, el cliente envía un mensaje solicitando un determinado servicio a un servidor, en otras palabras hace una petición, y éste envía una o varias respuestas, o lo que es lo mismo provee el servicio.

En un sistema distribuido cada máquina puede cumplir con el papel de servidor para algunas tareas y el papel de cliente para otras. La idea principal es hacer uso de una máquina para que realice diversas tareas, pero que realice aquellas que son más adecuadas a sus características.

De manera general el trabajo que requiere de mayor poder de cálculo es procesado por el servidor y los procesos que tienen que ver con la interacción con el usuario son desempeñados por el cliente.

La esencia de esta arquitectura es separar una gran pieza de software en módulos con el propósito de hacer más fácil su desarrollo y mejorar su mantenimiento.

Esta arquitectura permite distribuir físicamente los procesos y los datos en forma más eficiente, lo que en computación distribuida se ve reflejado de forma directa en el tráfico de la red, reduciéndolo de manera significativa.

Cliente. Es el proceso que permite al usuario formular los requerimientos y pasarlos al servidor, es conocido también como front-end. El cliente de manera general se encarga del manejo de todas las funciones relacionadas con la manipulación y despliegue de datos, por lo que está desarrollado con herramientas que permiten la creación de interfaces de usuario (GUI), además de acceder a los servicios distribuidos en cualquier parte de una red.

Servidor. Es el proceso encargado de atender a múltiples clientes que hacen peticiones de algún recurso administrado por él. Al proceso servidor se le conoce con el término back-end.

Las características básicas de una arquitectura Cliente/Servidor:

a) El proceso cliente proporciona las interfaces entre el usuario y el resto del sistema. El proceso servidor actúa como un motor de software que maneja recursos compartidos.

b) Las tareas del cliente y del servidor tienen diferentes requerimientos en cuanto a recursos de cómputo como velocidad del procesador, memoria, velocidad y capacidades del disco así como dispositivos de entrada y salida.

c) Se establece una relación entre procesos los cuales pueden ser procesados por una sola máquina o por diferentes distribuidas dentro de una red.

d) La relación establecida puede ser de uno a muchos, en la que un servidor puede dar servicio a muchos clientes, regulando su acceso a recursos compartidos.

e) Los clientes corresponden a procesos activos, ya que éstos son los que hacen peticiones de servicios a los servidores. Estos últimos tienen un carácter pasivo ya que esperan las peticiones de los clientes.

f) La relación entre clientes y servidores se establece a través del intercambio de mensajes entre ambos.

g) La plataforma de hardware y del sistema operativo del cliente y del servidor no son siempre la misma. Es esto donde radica una de sus principales ventajas ya que permite conectar clientes y servidores independientemente de sus plataformas.

h) Esta arquitectura permite agregar más clientes, lo que se conoce como escalabilidad horizontal, sin afectar de manera significativa el rendimiento. Permite mejorar las características del servidor o agregar múltiples servidores.

2.5. Detección de errores.

2.5.1. Entropía

La Teoría de la información es una rama de la teoría matemática de la probabilidad y la estadística que estudia la información y todo lo relacionado con ella, como son los canales de comunicación, compresión de datos, criptografía y temas relacionados.

Esta teoría está basada en los trabajos realizados por Claude E. Shannon. Para ser más específicos se inició a partir de un artículo publicado en el *Bell System Technical Journal* en 1948, titulado *Una teoría matemática de la comunicación*. [8] [9]

En este trabajo Shannon demostró que todas las fuentes de información (telégrafo, teléfono, radio, la gente que habla, las cámaras de televisión, etc.) se pueden medir y que los canales de comunicación tienen una unidad de medida similar. Mostró también que la información se puede transmitir sobre un canal si y solamente si la magnitud de la fuente no excede la capacidad de transmisión del canal que la conduce. Además estableció las bases para la corrección de errores, supresión de ruidos y redundancia.

En esta teoría la información es tratada como magnitud física y para caracterizar la información de una secuencia de símbolos se utiliza la entropía. Se parte de la idea de que los canales no son ideales, aunque muchas veces se idealicen las no linealidades, para estudiar distintos métodos para enviar información o la cantidad de información útil que se puede enviar a través de un canal.

En la Teoría de la información la entropía es la magnitud que mide la cantidad de información contenida en una señal, es decir el flujo de datos, en otras palabras lo que nos aporta sobre un dato o hecho concreto.

La medida de la entropía puede aplicarse a información de cualquier naturaleza, y nos permite codificarla adecuadamente, indicándonos los elementos de código necesarios para transmitirla, eliminando toda redundancia.

La entropía nos indica el límite teórico para la compresión de datos.

Shannon define a la entropía en términos de un acontecimiento al azar discreto mediante la siguiente fórmula:

$$H(x) = \sum_{i=1}^{m} p(i)\log_2\left(\frac{1}{p(i)}\right) = -\sum_{i=1}^{m} p(i)\log_2 p(i)$$

En donde H(x) es la entropía, las “p” son las probabilidades de que aparezcan los diferentes códigos y “m” el número total de códigos. Al referirse a un sistema, las “p” se refieren a las probabilidades de que se encuentre en un determinado estado y “m” el número total de posibles estados.

Se utiliza habitualmente el logaritmo en base 2, y entonces la entropía se mide en bits.

2.5.2. Hashing

Hashing es el proceso por el cual se transforma una cadena de caracteres en un valor de un largo preestablecido o clave de la cadena original.

La generación de valores hash es empleado para tener acceso a datos, es decir puede ser usado para indexar o para recuperar elementos pertenecientes a una base de datos esto debido a que es más rápido encontrar el elemento usando una pequeña llave de hash que encontrarlo usando el valor original. También son empleados en el área de seguridad en lo que concierne a los sistemas de cifrado. [7]

El algoritmo hashing es también llamado función hash. Debido a la rapidez para la recuperación de datos, el hashing es usado para cifrar y descifrar firmas digitales (usadas para la autenticación de mensajes enviados y recibidos).

Como firma digital se le denomina a un mensaje resumido mediante una función hash y cifrado con una llave privada.

Las funciones hash usadas en autenticación y firma digital permiten:

No tener que cifrar todo el texto. En los servicios de autenticación y firma digital, ya que el proceso es lento con los algoritmos asimétricos. El resumen sirve para comprobar si la clave privada del emisor es auténtica, no es necesario cifrar todo el texto si no se quiere confidencialidad.

Comprobar automáticamente la autenticidad. Si se cifra todo el texto, al descifrar solo se puede comprobar la autenticidad mirando si el resultado es inteligible; evidentemente este proceso debe realizarse de forma manual. Utilizando un resumen de texto, se puede comprobar si es auténtico comparando el resumen realizado en el receptor con el descifrado.

Comprobar la integridad del texto, ya que si ha sido dañado durante la transmisión o en recepción no coincidirá el resumen del texto recibido con el resultado del proceso de descifrado.

El hash es muy pequeño en comparación con el texto a partir del cual se obtuvo, y es generado por una fórmula o función matemática en la cual es prácticamente improbable que textos diferentes generen el mismo valor hash.

Los valores hash son de gran importancia en los sistemas de seguridad donde son usados para garantizar que los mensajes transmitidos no han sido manipulados o alterados.

El emisor genera el valor hash de un mensaje, cifra el mensaje y envía el hash junto con el mensaje. El receptor descifra el mensaje y el hash, y genera el valor hash del mensaje que ha recibido, una vez obtenido el hash lo compara con el hash que le ha sido enviado. Si los dos valores de hash son los mismos, implica que existe una muy alta probabilidad de que el mensaje ha sido transmitido de forma intacta, es decir, que no fue alterado en su contenido, lo que tiene que ver con el concepto de integridad de los datos.

Las funciones hash son públicas e irreversibles. No cifran, solo comprimen los textos en un bloque de longitud fija. Las funciones hash no son reversibles, es decir, no se puede recuperar el texto desde el resumen.

Es imposible inventar dos mensajes cuya función Hash sea la misma.

Las funciones hash más conocidas son:

a) MD2, abreviatura de Message Digest 2, diseñado para computadoras con procesador de 8 bits.

b) MD4, abreviatura de Message Digest 4, desarrollado por Ron Rivest, uno de los fundadores de RSA Data Security Inc. y padre del sistema asimétrico RSA. Aunque se considera un sistema inseguro, es importante porque ha servido de base para la creación de otras funciones hash. Un sistema de ataque desarrollado por Hans

Dobbertin posibilita el crear mensajes aleatorios con los mismos valores de hash (lo que es conocido como colisiones), por lo que ya no se usa.

c) MD5, abreviatura de Message Digest 5, también obra de Ron Rivest, que se creó para dar seguridad a MD4, y que ha sido ampliamente usado en diversos campos, como autenticador de mensajes en el protocolo SSL y como firmador de mensajes en el programa de correo PGP.

d) SHA-1, Secure Hash Algorithm, desarrollado como parte integrante de Secure Hash Estándar (SHS) y el Digital Signatura Estándar (DSS) por la Agencia de Seguridad Nacional Norteamericana, NSA. Sus creadores afirman que la base de este sistema es similar a la de MD4 de Rivest. La versión actual se considera segura y es muy utilizada en el algoritmo de firma, como en el programa PGP en sus nuevas claves DH/DSS (Diffie-Hellman/Digital Signatura Standard). En la actualidad se está estudiando las versiones de SHA con longitudes de claves de 256, 384 y 512 bits.

e) RIPEMD-160, desarrollado por un grupo de investigadores europeos, entre los que se encuentran Hans Dobbertin y otros investigadores incluidos en el proyecto RIPE (Race Integrity Primitives Evaluation). Su primera versión presentaba las mismas debilidades que MD4, produciendo colisiones, pero las versiones mejoradas actuales son consideradas seguras. Maneja claves muy robustas, normalmente de 160 bits, aunque existen versiones de 128 y se están planteando nuevas de 256 y 320 bits, es muy rápido, no está patentado y su código fuente está abierto.

2.5.3. MD5

El algoritmo de hash más utilizado en estos momentos es el MD5. Es decir, se trata de uno de los más populares algoritmos para la generación de resumen de mensajes.

Este algoritmo es empleado para aplicaciones donde un gran archivo debe ser comprimido de una manera segura entes de ser cifrado con una llave privada bajo un sistema de cifrado de llave pública, tal como lo es el RSA.

En esencia el algoritmo MD5 es una forma para verificar la integridad de los datos que son enviados a través de un medio inseguro.

Este algoritmo fue desarrollado por Ronald Rivest en 1995 y está basado en dos algoritmos anteriores MD2 y MD4. Todos estos algoritmos procesan mensajes

de entrada en bloques de 512 bits, y producen una salida con un número de 128 bits.

MD5 sustituyó a MD4, se dice que este último es de mayor rendimiento, sin embargo no han sido publicados elementos que comprometan su integridad y funcionamiento.

El MD5 no sirve para el cifrado de un mensaje ya que éste es destruido completamente, la información no es recuperable de ninguna manera ya que hay pérdida de información, por lo que si se cuenta con el resultado de este algoritmo no puede ser recuperado el mensaje original del cual fue obtenido. Además no es computacionalmente factible producir dos mensajes teniendo el mismo resumen de mensaje, o producir un mensaje para que sea obtenido un mismo resumen.

En los últimos tiempos el algoritmo MD5 ha mostrado ciertas debilidades, aunque sin implicaciones prácticas reales, por lo que se sigue considerando en la actualidad un algoritmo seguro.

MD5 comienza alargando el mensaje a una longitud congruente en módulo 448 mod 512. Es decir, la longitud del mensaje es 64 bits menos que un entero múltiplo de 512. El alargamiento del mensaje consiste en agregar un bit con valor de “1” seguido por cuantos bits en “0” sean necesarios. Posteriormente la longitud original del mensaje es almacenada en los últimos 64 bits del relleno, es decir, que no se toma en cuenta los bits que se añadieron para obtener la longitud requerida por el algoritmo. Con lo anterior se tiene un mensaje con un número entero de bloques con una longitud de 512 bits.

Adicionalmente se inicializa, con un valor fijo, un buffer de 128 bits. Este buffer puede verse como 4 registros de 32 bits (A,B,C,D) y son inicializados con los siguientes valores hexadecimales:

A=67452301
B=EFCDAB89
C=98BADCFE
D=10325476

Durante varias rondas de procesamiento el algoritmo toma bloques de 512 bits de la entrada y los mezcla con los 128 bits del buffer. Este proceso es repetido hasta que todos los bloques de entrada han sido consumidos. El valor resultante en el buffer es el hash del mensaje.

El algoritmo MD5 implantado en este trabajo fue tomado de "RSA Data Security, Inc. MD5 Message-Digest Algorithm".[15]

CAPÍTULO 3.

Análisis de Formatos de archivos existentes

3.1. Formato Wavefront OBJ.

Este formato de archivos también es conocido como Wavefront Object o como OBJ. [2] [19]

Descripción.

Tipo: Vector 3D.
Colores: Ilimitados.
Compresión: Ninguna.
Tamaño Máximo de Archivo: Ilimitado.
Múltiples Imágenes por Archivo: Si.
Formato Numérico: No Aplica.
Creador: Wavefront.
Plataforma: UNIX.
Aplicaciones que lo soportan: Advanced Visualizer.
Uso: Su principal aplicación es para el almacenamiento e intercambio de información en 3D.
Comentarios: El formato de archivo Wavefront OBJ es un estándar muy utilizado para la representación poligonal de datos en formato ASCII.

Los archivos Wavefront OBJ son utilizados por la aplicación Wavefront Advanced Visualizer para almacenar objetos geométricos compuestos por líneas, polígonos, y formas libres de curvas y superficies. Wavefront es mejor conocido por sus herramientas gráficas para computadoras, entre las que se encuentran para modelar, animación, y herramientas de edición de imagen. Estos programas son para estaciones de trabajo poderosas tales como las fabricadas por Silicon Graphics.

Este tipo de archivos se almacena con extensión ".obj", siguiendo la convesión de UNIX de utilizar letras minúsculas para las extensiones. La última versión de este tipo de archivos es la 3.0.

En los programas de Wavefront 3D, los archivos con objetos geométricos pueden ser almacenados, ya sea en formato ASCII (usando el la extensión ".obj"), o en formato binario (usando la extensión ".MOD"). El formato binario es propietario, y con cuenta con documentación restringida a usuarios que paguen, por lo cual solo se expondrá la información del formato ASCII.

Los archivos OBJ soportan líneas, polígonos, y formas libres tales como curvas y superficies. Líneas y polígonos son descritos en término de sus puntos, mientras que las curvas y superficies se definen mediante puntos de control y otra información dependiendo del tipo de curva. El formato acepta curvas racionales y no racionales, entre estas últimas están las curvas de Bezier, B-spline, Cardinal (Catmull-Ron splines), y ecuaciones de Taylor.

a) Organización del Archivo. Los archivos OBJ no requieren ningún tipo de cabecera en específico, aunque es común que este tipo de archivos empiecen con una línea de comentario de algún tipo. Las líneas de comentario comienzan con el símbolo #, y pueden agregarse líneas en blanco con el fin de ayudar en el formato y la legibilidad del código. Cada línea, que no sea en blanco, comienza con una palabra clave y puede ser seguida en la misma línea con datos para dicha palabra clave. Las líneas son leídas y procesadas hasta llegar al final del archivo, y las líneas pueden ser lógicamente unidas mediante el carácter de continuación de línea (\) al final de la línea.

A continuación se listarán las palabras claves más comunes:

Datos de Vértices:

v	Vértices Geométricos
vt	Vértices de Texturas
vn	Vértices de Normales
vp	Parameter space Vértices

Atributos de formas libre Curvas/Superficies:

Deg	Grados
Amat	Matriz Básica
Step	Avance
Cstype	Curva o tipo de Superficie

Elementos:

P	Puntos
L	Línea
F	Cara
Curv	Curva
Curv2	Curva 2D
Surf	Superficie

b) Detalles del Archivo. Los archivos OBJ qué más se encuentran, son los que contienen solo caras poligonales. Para describir un polígono, el archivo primero describe cada punto con la palabra clave “v”, después describe la cara con la palabra clave “f”. La línea con el comando de la cara, contiene la numeración de los puntos que forman la cara, comenzando desde el índice 1 para la lista de los puntos, en el orden en que aparecen dentro del archivo. Por ejemplo, el siguiente código describe a un triángulo:

```
# Simple Wavefront file
v 0.0 0.0 0.0
v 0.0 1.0 0.0
v 1.0 0.0 0.0
f 1 2 3
```

Es también posible utilizar índices negativos.

Los archivos OBJ no contienen las definiciones de color para las caras, aún así, estos archivos pueden contener la referencia hacia materiales, los cuales se almacenan en un archivo de librería de materiales de manera separada al archivo OBJ. La librería de materiales puede cargarse mediante la palabra clave “mtllib”. La librería de materiales contiene las definiciones para los valores RGB para las componentes difusas, ambientales y especulares del color, así como otras características tales como la transparencia y la refracción.

Para hacer referencia a un material, se ocupa la palabra clave “usemtl”, y todas las caras que le sigan, se le asigna ese material, esto hasta la aparición de una nueva palabra clave “usemtl”.

Uno de los principales inconvenientes de este tipo de archivos es su tamaño. Al tratarse archivos en formato ASCII, su tamaño es mucho mayor, a que si fuera en binario, desafortunadamente su contraparte binaria está muy controlada su documentación, por lo cual su uso es escaso, y se limita a productos del fabricante Wavefront. Y si se agrega que la información de los materiales se encuentra dentro de un archivo diferente al OBJ, tenemos que el tamaño aumenta en mayor medida.

3.2. Formato 3DS.

Este formato de archivo también es conocido como 3DS (modo binario) y ASC (modo texto). [2] [13]

Descripción.

Tipo: Escena.
Colores: Ilimitados.
Compresión: Ninguna.
Tamaño Máximo de Archivo: Ninguno.
Formato Numérico: Múltiple.
Creador: Autodesk.
Plataforma: MS-DOS, Windows.
Aplicaciones que lo soportan: 3D Studio, Calgary TrueSpace, y varios otros programas de modelado en 3D.
Uso: Almacenamiento de información de escenas en 3D para su uso por aplicaciones de render.
Comentarios: Autodesk 3D Studio se ha convertido en un archivo estándar para el intercambio de información de escenas en 3D. Muchos programas de modelado en 3D y aplicaciones de render son capaces de leer y escribir archivos 3DS. Este formato cuenta con una gran estadía en el mercado y además goza de una amplia popularidad en el

mismo. El programa en si mismo tiene implantada una gran cantidad de funciones, y esto se ve reflejado en la composición del archivo.

Autodesk 3D Studio (3DS) es relativamente un programa de gran uso entre la gente dedicada al modelado y rendereado de escenas en 3D. Inicialmente funcionaba bajo la plataforma MS-DOS y usa un manejador de memoria proporcionado por terceros, esto con el fin de optimizar el uso de los recursos disponibles en la computadora o servidor en donde se usa. Su rendimiento se compara favorablemente ante otros formatos en diferentes plataformas, y probablemente por ello es que se ha convertido en una de las soluciones más populares entre la gente que trabaja en escenas 3D con construcciones muy detalladas.

Desafortunadamente, Autodesk ha decidido que la información concerniente a este formato de archivo solo se encuentre disponible mediante la adquisición de su software de desarrollo. Por lo tanto la información que aquí se presenta fue obtenida mediante fuentes públicas, y por lo tanto puede encontrarse incompleta.

Existen dos formatos usados por 3D Studio:

a) Formato binario, al cual se hace referencia mediante el sufijo 3DS.

b) Formato ASCII, el cual tiene el sufijo ASC.

Ambos son usados como formatos de intercambio, sin embargo es más común encontrar el formato binario, esto debido a que la versión ASCII es usada generalmente solo para el almacenamiento de la definición de los objetos que conforman la escena, y además que su estructura es muy parecida a la del formato binario.

a) Organización del Archivo. Los archivos 3DS consisten en etiquetas, llamadas chunk por Autodesk. Cada chunk consisten en un valor de identificación (ID), y el offset que indica el inicio del siguiente chunk. La información asociada con cada chunk sigue después de valor de ID y de la información del offset. Lo anterior se representar con la siguiente estructura:

```
typedef struct _CHUNK
{
        WORD  Chunk_ID;          /* Tag type */
        DWORD Chunk_length;   /* Relative offset in bytes to next chunk */
} CHUNK;
```

Cada chunk puede ser clasificado como Primario, Principal, o Subordinado, y son acomodados jerárquicamente como se muestra a continuación:

Chunk Primario

Chunk Principal #1

Chunk Subordinado #1

Chunk Subordinado #2

.

.

.

Chunk Principal #2

Chunk Subordinado #1

Chunk Subordinado #2

.

.

.

Chunk Principal #3

.

.

.

Los chunk Principales se dice que “pertenecen” al chunk Primario, y los chunk Subordinados asociados a un chunk Principal se dicen que pertenecen a este chunk Principal. Se hace notar esta estructura, ya que muchos de estos chunk están asociados con el comportamiento de 3D Studio al momento de la ejecución. Los chunk se listan aquí ya que la aplicación de render o el programa para la conversión del archivo puede hacer uso de ellos como indicaciones para la creación y el comportamiento de los objetos encontrados dentro del archivo.

Solo existe un chunk Primario por archivo, el cual se encuentra al principio del mismo, y enseguida de la etiqueta del chunk primario, sigue la etiqueta para el primer chunk Principal. El campo para la longitud del chunk (**chunk_length**) de la etiqueta del chunk Principal, es generalmente también el offset para el siguiente chunk Principal.

Después de la etiqueta para cada chunk Principal sigue los datos asociados con el chunk Principal y/o las etiquetas del chunk Subordinado. Por ello, el lector debe entender el formato y longitud de los datos asociados con cada chunk Principal para así obtener la información contenida en los chunk Subordinados pertenecientes al chunk Principal.

3.3. Archivos de Maya (MA y MB).

Estos dos tipos de archivos son generados por el programa Maya, y cada una de las extensiones indica al tipo de salida que se escogió. La extensión MA es para los archivos Maya en ASCII, y la extensión MB denota a los archivos Maya en Binario. [14]

Para cada archivo Maya ASCII (MA) se tiene una línea inicial que indica el nombre del producto. En caso de que el archivo no contenga esta línea, se toma como si el archivo no fuera un archivo Maya ASCII. Los comentarios no se conservan una vez que el archivo ha sido guardado por Maya, esto con excepción de algunas líneas que se muestran con el ejemplo.

```
//Maya ASCII 4.0 scene
//Name: Cube.ma
//Last modified: Wed, Mar 27, 2002 08:57:20 PM
requires maya "4.0";
currentUnit -l centimeter -a degree -t film;
```

En la primera línea se tiene el comentario que indica que el archivo Maya ASCII es válido. La línea dos contiene el nombre de trabajo del archivo, y la línea tres la fecha de modificación del archivo. Como se puede ver, estos son los únicos comentarios que se conservan dentro del archivo una vez que ha sido guardado por Maya. En la cuarta línea se tiene la versión de Maya en que el archivo fue creado, y dependiendo del tipo de datos que se crearon significará que se requiere como mínimo esa versión para el correcto despliegue de los datos del archivo. En la última línea se indican el tipo de unidades que se usarán dentro de las propiedades del proyecto cuando se trabaje con el archivo, solo se guardan dentro del archivo las propiedades o valores que no corresponden a las que vienen por default, si un valor es igual a los de default, entonces no se almacena pero aún así es ocupado cuando se muestra el archivo.

Los atributos o nodos pueden aparecer en cualquier orden, con la única restricción de que si los datos pertenecen a su vez a otro o forma parte de una “relación”, entonces dicha “relación” debe almacenarse en el orden en que los valores son necesitados. Esto significa que un “nodo hijo” siempre aparece después del “nodo padre” y nunca antes, aunque esto no significa que los datos del “nodo hijo” vendrán inmediatamente después de los valores del “nodo padre”, ya que entre ellos pueden encontrarse otros nodos que no tienen relación con ningún otro nodo.

Asimismo los atributos que pertenecen a cada nodo se pueden encontrar en cualquier orden, pero cada dato del nodo sigue en un orden lógico y directamente después que su predecesor. Esto quiere decir que por ejemplo la localización de los vértices, que se encuentra dividido en varias líneas de archivo, primero

contiene una cabecera la cual indica el tamaño total de las entradas necesarias, e inmediatamente después aparecerán las entradas de los datos.

Cada línea de archivo, con excepción de los comentarios, termina con un punto y coma (;), esto quiere decir que aún cuando dentro del archivo aparezca un comando que ocupa varias líneas, en realidad Maya tomará como una sola línea todo lo que se encuentra antes del punto y coma. Ciertos comandos pueden producir una cantidad considerable de valores (como es el caso de los vértices), por lo cual son divididos en diferentes comandos, cada uno separado por punto y coma.

En cada caso que un comando sea dividido, siempre hay un límite para el número de comandos en que será dividido. Esto quiere decir, que si por ejemplo se tiene un arreglo de tres valores por elemento (como es el caso de los vértices), entonces siempre cada uno de estos tres elementos tendrá su línea o comando.

Dentro del archivo se presentará el caso de que muchos de los datos se referirán a otros datos mediante índices. El tamaño de los atributos, si son necesarios en los arreglos, se almacena de tal manera que pueda ser leído antes que los verdaderos datos, esto con el fin de que el tamaño correcto del arreglo pueda ser prealmacenado y subsecuentemente usado con los valores.

Un *nodo* es un objeto dentro de Maya, pero no se debe confundir el término objeto con su definición geométrica. Un *objeto* puede ser cualquier tipo de objeto que es necesario para la escena, pudiendo ser las cámaras, luces, mallas, transformaciones, etc. Existen diferentes tipos de *nodos*, debido a que mediante ellos se representa a un objeto, y cada nodo contiene un identificador que se usó en conjunción con un nombre y un “nodo padre” en caso de que exista éste último. Este “nodo padre” es importante ya que permite que Maya cree jerarquías para los objetos para su representación en pantalla. El *nodo* para un objeto geométrico comienza con una transformación y el nombre.

Cada *nodo* contiene parámetros y propiedades que son válidas para ese nodo en específico. Y cada nodo es descrito completamente hasta la aparición de otro comando *createNode***.** Por lo cual se debe tener cuidado de no mezclar los datos de los nodos, y esto ayuda a la correcta creación y carga de todos los datos en el código que corresponden a un nodo. Una vez que otro *createNode* es encontrado, el nodo anterior se finaliza y no podrán serle añadidos atributos o ser modificado. Cada valor que se le agrega a un nodo es llamado *atributo* (de dicho nodo) y es denotado por el comando *setAttr***.**

Enseguida se presenta un ejemplo más completo de un archivo Maya ASCII:

```
createNode transform -n "pCube1";

    setAttr ".t" -type "double3" 5.96 0.56 4.3 ;
```

```
    setAttr ".r" -type "double3" -12.33 37.79 0 ;
    setAttr ".s" -type "double3" 1.5 0.4 0 ;

createNode mesh -n "pCubeShape1" -p "pCube1";

    setAttr -k off ".v";
    setAttr ".vir" yes;
    setAttr ".vif" yes;
    setAttr ".uvst[0].uvsn" -type "string" "map1";
    setAttr -s 14 ".uvst[0].uvsp[0:13]" -type "float2"
        0 0    1 0    0 1    1 1    0 2    1 2    0 3  1 3   0 4    1 4   2 0    2 1   -1 0   -1 1;
    setAttr ".cuvs" -type "string" "map1";
    setAttr -s 8 ".vt[0:7]"
        -0.5 -0.5  0.5
         0.5 -0.5  0.5
        -0.5  0.5  0.5
         0.5  0.5  0.5
        -0.5  0.5 -0.5
         0.5  0.5 -0.5
        -0.5 -0.5 -0.5
         0.5 -0.5 -0.5;
    setAttr -s 12 ".ed[0:11]"
        0 1 0
        2 3 0
        4 5 0
        6 7 0
        0 2 0
        1 3 0
        2 4 0
        3 5 0
        4 6 0
        5 7 0
        6 0 0
        7 1 0;

    setAttr -s 6 ".fc[0:5]" -type "polyFaces"
        f 4   0     5  -2   -5       mu 0  4    0    1    3   2
        f 4   1     7  -3   -7       mu 0  4    2    3    5   4
        f 4   2     9  -4   -9       mu 0  4    4    5    7   6
        f 4   3    11  -1 -11       mu 0  4    6    7    9   8
        f 4 -12   -10  -8   -6       mu 0  4    1   10   11   3
        f 4  10      4   6    8       mu 0  4   12    0    2  13 ;
```

No hay documentación acerca del diseño de los datos en un archivo de maya binario (mb), sin embargo los programadores de MAYA facilitan el API de programación para manipular su archivo en un nuevo proyecto. La forma en que se carga un archivo a memoria con el API de maya es como sigue:

```
// se inicializa el API de maya antes de usar sus funciones
void main()
{
  MStatus stat = MLibrary::initialize ("cargador de archivos mb" );
  if ( !stat )
    return false;
```

```
  // función para obtener el nombre del archivo
  char* fileName = getFileNameToLoad();

  // se prepara a Maya para que lea los datos de un archivo
  MFileIO::newFile( true );

  // Se cargan los datos desde el archivo
  stat = MFileIO::open( fileName );
  if ( !stat )
    return false;

  // Se remueve cualquier dato temporal que Maya utilizó para cargar el modelo
  // como “la lista de deshacer” que por ahora no se utilizará
  stat = MGlobal::executeCommand( "delete -ch" );
  if ( !stat )
    return false;

  // Se recorren todos los nodos en el DAG y se imprime su nombre
  MItDag dagIter( MItDag::kBreadthFirst, MFn::kInvalid, &stat );
  for ( ; !dagIter.isDone(); dagIter.next())
  {
    MDagPath dagPath;
    stat = dagIter.getPath( dagPath );

    cerr << "Found DAG Node: "
        << dagPath.fullPathName().asChar()
        << endl;
  }

  // Se cierra el API de Maya para liberar recursos
  MLibrary::cleanup();
}
```

El proceso es simple ya que solo se inicializa el API, y se procede a cargar el archivo hacia la estructura DAG (Directed Acyclic Graph), conocido por sus programadores simplemente como “arbol”, la mayor parte de los datos se guardarán en sus nodos y en teoría cualquier nodo puede ser padre o hijo de otro nodo, la información que se encuentra en esta estructura son los vértices, reglas de unión, trasformaciones e información del sistema de esqueletos del modelo.

Otra parte de los datos de carga se alojarán en la estructura llamada DG (Dependency Graph) esta estructura no es como la anterior ya que obedece más a una estructura de red ya que los nodos pueden ir unidos a cualquier otro nodo de esta estructura. Aquí es donde los materiales, texturas e información acerca de la animación del modelo es almacenada.

3.4. LWO.

Los datos que se guardan dentro de este tipo de archivos pueden ser los puntos, polígonos o superficies que describen la geometría o apariencia de un objeto. Entendiéndose como “polígono” cualquier elemento geométrico (caras, curvas o planos, por ejemplo) que pueda ser definido por una lista de puntos que lo conformen, y por “superficie” a una lista de atributos, a veces llamados materiales, que definen las superficie visual de un polígono.

Los objetos del archivo pueden contener múltiples capas, es decir, estar formados por varias partes, y cada una de estas partes puede ser una malla (mesh) o varias mallas (mesh) separadas. Además pueden existir definiciones de una o más superficies sin datos de puntos o polígonos a los cuales se le aplican. Las definiciones a las superficies pueden además incluir referencias a otros archivos (imágenes, por ejemplo), a plug-ins (agregados), y archivos que contengan parámetros que varíen con el tiempo. [13]

Los archivos LWO son archivos binarios compuestos por una serie de bytes en el rango de 0 a 255. Todos los datos son escritos con el ordenamiento de bytes big-endian, Motorola u orden de red, este ordenamiento especifica que el byte más significativo se escribe primero.

Debido a que se sigue este lineamiento, la estructura básica elemental es el *chunk*, el cual se estructura de la siguiente manera:

CHUNK ::= tag[ID4], length[U4], data[...], pad[U1] ?

El *tag[ID4]*, es una secuencia de 4 bytes de valores ASCII 7-bit, generalmente en mayúsculas. Este valor se ocupa para identificar el tipo de dato que le sigue a continuación. Se tiene asignados valores ya preestablecidos de acuerdo al formato LWO.

La longitud del chunk está dada por un valor entero no signado de longitud 4 bytes, este valor se refiere a la longitud del bloque de datos del chunk.

Enseguida después del bloque de datos, aparece un byte, el cual es opcional, el cual sirve para identificar si la longitud es impar, y si se incluye debe ser un valor de 0 (cero).

Los archivos LWO incorporan los llamados “sub-chunks”, los cuales son chunks dentro de otros chunks, estos sub-chunks cuentan con las mismas características de los chunks normales, con la salvedad de que el valor de su longitud es de 2 bytes.

Un archivo LWO comienza con los 4 bytes “FORM” seguido de 4 bytes, los cuales indican la longitud del archivo (no incluyendo estos primeros 8 bytes).

Los chunks pueden encontrarse en cualquier orden, excepto cuando los datos de un chunk dependen de los datos de otro chunk, en este caso el chunk dependiente debe encontrarse después del chunk del cual depende. Sin embargo un problema que se presenta es que las partes de Layout y Modeler de LightWave 3D escriben sus respectivos chunks en diferente orden, por lo cual es importante que los interpretes soporten la *independencia de orden*.

El formato LWO soporta tipos de datos compuestos, es decir, tipos de datos que utilizan los tipos de datos fundamentales, para de esta manera obtener un mejor manejo de datos más complejos. Entre los principales se encuentran los datos del color:

COL4 ::= red[U1], green[U1], blue[U1], pad[U1]

Cada color se especifica por su componente RGB, en donde cada componente de color toma un valor dentro del rango de 0 a 255. La versión LWO2 modificó este parámetro, al pasar de 4 a 12 bytes el tamaño total de este tipo de dato, y son 4 bytes para cada componente de color.

VEC12 ::= X[F4], Y[F4], Z[F4]

Las coordenadas 3D se escriben como si se tratara de un vector XYZ en valores de punto flotantes.

Estos son unos de los principales tipos de datos compuestos, sin embargo existen muchos otros.

En seguida se listan los chunks más importantes con que puede contar un archivo LWO, sin embargo debe recordarse que estos chunks pueden aparecer en cualquier orden.

Point List (Lista de Coordenadas)
*PNTS { point-location[VEC12] * }*

Este chunk contiene la lista de coordenadas X, Y y Z de todos los puntos del objeto. Dado que cada coordenada tiene tres componentes, y cada componente se almacena en un número de 4 bytes de punto flotante, el número de puntos de un objeto puede ser determinado dividiendo la longitud en bytes del bloque PNTS entre 12.

Cada punto dentro del bloque PNTS se numera de acuerdo en el orden en que van apareciendo, comenzando con el valor 0 (cero). Este índice es utilizado por los polígonos para definir sus vértices, por lo que el bloque PNTS debe ir antes que los bloques POLS, CRVS y PCHS dentro del archivo.

Surface List (Lista de Superficies)
*SRFS { surf-name[S0] * }*

Este chunk contiene una lista con los nombres de todas las superficies en un objeto. Cada nombre de superficie aparece como una cadena de caracteres con un valor de NULL al final. Si la longitud de la cadena, incluyendo el valor NULL, es impar, entonces se agrega otro valor NULL a la cadena. Se deben leer los nombres de las superficies hasta que la longitud dada por el chunk se haya recorrido.

En la terminología de LightWave 3D, una “superficie” es el nombre de una serie de atributos, es el equivalente a “materiales” en la terminología de 3DStudio. Estos nombres son numerados de acuerdo al orden en que van apareciendo, se empieza desde 1 (uno), y este índice será utilizado por los polígonos para definir su superficie, por lo que este chunk debe ir antes que los bloques POLS, CRVS y PCHS dentro del archivo.

Face List (Lista de Caras)
POLS { (numvert[U2], vert[U2] # numvert, surf[I2]) }*

Este chunk contiene una lista con todos los polígonos de un objeto. Cada entrada consiste en un entero de 2 bytes [short integer], el cual es el número de vértices que forman al polígono, enseguida aparecen tantos enteros de 2 bytes [short integer] cómo número de vértices se especificaron, estos valores indican los vértices como índices dentro de la lista de coordenadas. El número de vértices en un polígono puede variar desde 1 hasta 200. El orden en que se especifican los vértices de un polígono debe seguir un orden de aparición en su construcción en sentido de las manecillas del reloj.

Como los puntos en el chunk PNTS están referenciados mediante un índice entero de 2 bytes, se tiene que el máximo número de puntos para el formato LWO es de 65,536.

Otra característica especial que tenía el formato LWO en sus primeras versiones, era la existencia de lo que se denominaba polígonos de detalle, los cuales consistían en que se dibujaba un polígono sobre el polígono ya dibujado, pero con la salvedad de que su forma de construcción era en sentido inverso a las manecillas del reloj. Con esto se lograba que los polígonos tuvieran dos caras, esto es importante ya que los polígonos en LWO solo son de una cara. Esta característica fue abandonada y es ignorada cuando se lee esta información.

3.5. Formato MD2.

El formato de archivo MD2 guarda información poligonal de objetos con animación, la manera en que esta información es almacenada en el archivo es guardando las coordenadas de sus vértices de cada cuadro (frame) de animación.

Este archivo fue usado originalmente para guardar la información de los personajes del videojuego Quake 2 programado por ID software, pero debido a la sencillez en su diseño y a los diversos modeladores gratuitos que dan soporte a este formato, se ha vuelto un formato muy utilizado en el desarrollo de aplicaciones en 3D como son demostraciones de capacidades graficas, remasterización de videojuegos antiguos, (Doom, Duke Nukem) por programadores independientes. [19]

Debido a que el archivo fue diseñado para aplicaciones de render en tiempo real y a que la información por frame es bastante, los modelos solo puede tener hasta 2048 vértices siendo este un archivo para modelos de bajo nivel de detalle.

Este archivo no guarda información de los colores en la superficie de los modelos, en lugar de ellos, confía en los efectos que una buena texturización puede dar ya que un modelo md2 bien texturizado tiene una calidad semejante a cualquier modelo con detalle alto.

A pesar que cada frame tiene una cadena de caracteres que lo identifica, no tiene una manera de saber cuáles frames pertenecen a una animación en particular, para ello se hace uso de archivos externos los cuales indican a la aplicación cuáles frames pertenecen a cada animación así como la velocidad a la cual los frames deben ser dibujados.

El modo en que los datos están almacenados en el archivo es binario, y cada una de sus partes son: la geometría del modelo, información de los cuadros de animación, y texturas que son obtenidas de manera indexada por medio de su posición en el archivo, información que está contenida en el header del mismo.

Cada una de las tres coordenadas de los vértices es almacenada en el archivo por medio de un byte, este vector no es la coordenada real del vértice, es un número escalado de manera que pueda ser almacenado en solo 3 bytes, para trasformarlo en un número de punto flotante, este vector debe ser multiplicado por el vector de escala y después sumar el resultado a un vector de traslación, estos vectores deben corresponder al frame de animación que se quiera dibujar.

El archivo guarda las reglas de unión de dos formas, la primera es la tradicional en la cual tiene un arreglo de triángulos con tres índices a los vértices que forma ese triángulo, la segunda manera es agrupando los índices de los vértices en dos grupos, el primero es cuando los vértices forman un conjunto de

triángulos los cuales están unidos por uno de sus lados (triangle strip) y el otro es cuando un vértice es común a todos los demás triángulos (triangle fan), esta forma de guardar las reglas de unión se diseño para acelerar el dibujo del modelo cuando se dibuja con OpenGL ya que las rutinas de dibujo de este API son más rápidas dibujando este tipo de conjuntos de triángulos.

3.6. Formato VRML.

El Lenguaje para Modelado de Realidad Virtual (VRML, por su nombre en inglés Virtual Reality Modeling Language) es un formato de archivo normalizado que tiene como objetivo la representación de gráficos interactivos tridimensionales, diseñado particularmente para su empleo en Internet. Consiste en un formato de archivo de texto en el que se especifican los vértices y las aristas de cada polígono tridimensional, además del color de su superficie. Es posible asociar direcciones web a los componentes gráficos así definidos, de manera que el usuario pueda acceder a una página web o a otro archivo VRML de Internet cada vez que se de clic en el componente gráfico en cuestión. [2] [13] [18]

Animaciones, sonidos, luces y otros aspectos de los mundos virtuales pueden interactuar con el usuario, o pueden activar otros eventos o acciones externos. Y más recientemente, se puede invocar código de programación escrito en Java o JavaScript.

Los archivos VRML son comúnmente conocidos como "mundos" y tienen la estensión .wrl. Aún cuando los mundos VRML son archivos de texto, estos pueden ser comprimidos mediante gzip, para de esta manera poder ser transmitidos por Internet de una manera más rápida. Una gran parte de programas para modelado 3D pueden guardar y crear objetos y escenas en formato VRML.

En conclusión se puede decir que las capacidades VRML, es decir, de crear mundos virtuales, sigue siendo las mismas que desde hace cinco años, mientras que las gráficas en 3D creadas en tiempo real han seguido mejorando. Es por ello que el VRML Consortium cambió su nombre a Web3D Consortium, y se ha comenzado a trabajar en un sucesor de VRML.

El formato VRML fue diseñado para cumplir con los siguientes requerimientos:

a) Independencia en la plataforma en la que se trabaje.
b) Extensibilidad.
c) Capacidad para trabajar con conexiones de bajo ancho de banda.

Desde el principio se decidió que el VRML no fuera una extensión del HTML, ya que este está diseñado para trabajar con texto y no con gráficas.

Además de que VRML requiere estar mejor optimizado para trabajo en una red de lo que está HTML. Sin embargo la razón principal para separar VRML del estándar HTML fue que sería perjudicial para ambos lenguajes el forzarlos a fusionarse y que las aplicaciones existentes para HTML debieran de volver a crearse para cumplir con un nuevo estándar, así se llegó a la conclusión de que VRML triunfaría o fracasaría independientemente de HTML.

3.7. Tabla comparativa de formatos.

Formato	Wavefront Object	3D Studio	Maya	LightWave 3D	Quake 2	VRML
Extensión	OBJ (ASCII) y MOD (Binario)	3DS	MA y MB	LWO	MD2	WRL
Límite de vértices	ilimitado	65,536 por objeto	No documen-tado	65,536 por objeto	2,048	2^{32}
Límite de objetos	Ilimitado	No documen-tado	No documen-tado	No documen-tado	1	2^{32}
Texturas	Si	Si	Si	Si	Si	Si
Animación	No	Si	Si	No	Si	Si
Control de errores	No	No	No	No	No	No
Document ación	ASCII gratuita Binario no existe	Controlada por Autodesk	Controlada por Alias/Wave front	Libre*	Libre*	Libre
Aplicacion es que la soportan	Advanced Visualizer	3D Studio, Calgary TrueSpace , y varios otros programas de modelado en 3D	Maya, y otros programas de modelado en 3D	LightWave 3D y LightWave Modeler.	Videojuego Quake 2	Navegado-res web con plugins
Envío por partes a través de una red	No	No	No	No	No	No
Jerarquía de los objetos	No	Si	Si	No	No	Si
Tipo de lectura	Secuencial	Chunks	Secuencial	Chunks	Indexada	Secuencial

* Requiere suscripción gratuita en página del desarrollador.

CAPÍTULO 4.

Formato GFI

4.1. Descripción del formato GFI.

4.1.1 Consideraciones de diseño.

El formato de archivo GFI (Gráfico de la Facultad de Ingeniería) se encuentra en modo binario y almacena información para la construcción de una escena en tercera dimensión. Contiene la información básica para la construcción de una escena en tercera dimensión, es decir, información acerca de los vértices y reglas de unión, así como información adicional para una mejor presentación visual de la escena, dicha información son las coordenadas de mapeo (si el modelo las requiere) e información de los parámetros de materiales utilizados.

Este formato surgió como una alternativa a los formatos más populares para almacenar información de modelos en tercera dimensión, debido a que presentan ciertas limitaciones y sobre todo a la falta de documentación de los mismos, lo que impide la modificación de los formatos para usos específicos y se genera una dependencia hacia el autor del formato, razón por la cual se ideó crear un formato que cubra esas limitantes y que al mismo tiempo presente un mayor desempeño en la carga dentro de una aplicación, así como en el envío a través de una red, para cubrir estos requerimientos se creó el formato GFI, y al presentar la documentación del formato en forma abierta, se deja la posibilidad de futuras mejoras y adecuaciones para diversos usos y aplicaciones.

En conjunto se desarrolló un API para la carga y visualización de dicho formato, esta API también permite cargar archivos en formato 3DS y su conversión, si así se desea, al formato GFI. El API permite la carga de archivos GFI almacenados de manera local, o cargarlos desde una ubicación remota, esto por que cuenta con una aplicación cliente/servidor lo que le permite comunicarse a otra máquina, la cual puede tener almacenados los archivos GFI y/o 3DS siempre que esta máquina cuente con el programa servidor.

Después de analizar los formatos de archivos más populares se encontró que cada uno de ellos tiene ciertas limitaciones, lo cual los hace inviables para su utilización en ciertas aplicaciones específicas, como por ejemplo si se quiere enviar sólo ciertas partes del modelo a través de una red ninguno de los formatos de archivos 3D existentes facilita este proceso. Al haber analizado las ventajas de cada formato éstas se conjuntan en un formato de archivo, lo que resulta en la obtención de un formato de archivo con mejor desempeño que los formatos existentes.

De los formatos analizados el más utilizado dentro de la industria de gráficos por computadora es el formato 3DS, esto pese a la escasa

documentación que existe, razón por la cual se decidió que sería el principal punto de comparación y de referencia en la construcción del formato GFI.

Al analizar los formatos se encontró dos formas de almacenamiento, las cuales son en forma binaria o de texto. Se decidió utilizar un almacenamiento en forma binaria ya que este tipo de archivos presentan un menor tamaño en relación con su contraparte de texto, esto se comprueba al momento de ser almacenada la misma información, ya que al contar con información de un cierto tipo de dato siempre ocupara un mismo número de bytes lo que no sucede cuando se representa en modo de texto, lo que permite tener un control sobre el tipo de dato que se almacena.

El tener un archivo con un menor tamaño es una ventaja ya que al momento de ser enviados a través de una red se ocupará un menor ancho de banda y será más rápida la transmisión. Otra ventaja es que la lectura de la información se realiza de manera más rápida, lo cual ayuda en el desempeño del formato.

Se encontró que los formatos de archivos utilizan diferentes formas de leer la información. Esto lo realizan a través de chunks, en el caso del formato 3DS; y de forma secuencial, en todos los demás formatos analizados.

Para obtener la información de un archivo que presenta la forma de lectura secuencial se debe de recorrer todo el archivo desde el inicio, y no puede ser seleccionado un objeto en particular o información específica de manera directa, lo que implica que para poder presentar la escena se requiere haber leído el archivo en su totalidad y esto representa una perdida de control al momento de la carga del archivo, por que no es posible seleccionar sólo la información deseada lo cual es una desventaja. Para su envío por red este tipo de archivo no está optimizado para ser cargado si es enviado por partes, ya que no hay forma de saber hasta donde se encuentra la información básica para presentar en escena un elemento del modelo.

Si un archivo está organizado en chunks se pueden conocer los bloques de información pero no en que orden están dichos bloques lo cual implica la lectura de sus identificadores para poder conocer si el bloque es de utilidad, ya que en el caso de 3DS posee datos no documentados ni utilizados para la creación de la escena. Si se encuentra que el bloque no es de utilidad no se toman en cuenta la información contenida dentro de él, además de que es necesario recorrer todos los chunks para saber cual es la información útil.

Vistos los inconvenientes anteriores se decidió que el archivo se basara en una tabla para tener acceso a los elementos que componen al modelo, esto para tener un mejor control acerca de los elementos que se quieran cargar en memoria en cierto momento, es decir, explorar la tabla de manera rápida y que sea cargado un elemento en específico sin tener que recorrer el archivo en su totalidad, contrario a lo que sucede en otros métodos de acceso utilizados por otros formato

de archivo que se basan en chunks o en un acceso secuencial a la información. Esta misma tabla se ocupará cuando el archivo sea enviado por red, esta proporcionará información de la jerarquía de los elementos y de su bloque de información Al ser enviado el archivo por paquetes los requerimientos de ancho de banda disminuirán logrando un mejor aprovechamiento de los recursos de red y con esto lograr trabajar en un ambiente que cuente con recursos de red limitados.

Otra ventaja de esta forma de acceso es que en el caso de que el modelo esté muy grande solo es necesario tener en memoria la tabla de acceso y la información que se utilice, y no el archivo en su totalidad lo cual se traducen en un mejor manejo de los recursos de cómputo.

4.1.2. Técnicas para mejora de desempeño.

Debido a que se realiza una gran cantidad de lectura del archivo, el tiempo de acceso a la información depende del medio en el cual se encuentre almacenado. Se ha demostrado que las lecturas realizadas al disco duro requieren de un mayor tiempo que las realizadas a memoria RAM.

Debido a lo anterior, para obtener la información de forma más eficiente se pueden realizar diferentes acciones para mejorar el desempeño. Una de ellas se presenta cuando los archivos son pequeños en comparación con la cantidad de memoria RAM que posee el sistema, en este caso el archivo puede ser cargado por completo a memoria, de esta manera se puede tener el acceso lo más rápido posible a la información de todos los objetos del modelo, en comparación que si se estuvieran leyendo del disco duro, el cual posee una velocidad inferior a la memoria RAM.

Sin embargo cuando el archivo es muy grande para ser cargado por completo a memoria, éste se puede ir cargando de acuerdo a los objetos que se vayan necesitando. Lo anterior implica varias lecturas al disco duro del sistema y pueden resultar en un bajo rendimiento de la aplicación, para compensar esto, se diseñó la tabla de objetos del archivo en formato GFI, la cual tiene la información más relevante de los objetos, y debido a que tiene un tamaño menor en relación al archivo completo, ésta puede ser guardada en memoria RAM caso contrario a si se quisiera guardar el archivo completo.

Uno de los elementos de la tabla es el offset del objeto el cual indica el inicio de la información del mismo dentro del archivo, lo cual permite posicionarse de la manera más rápida posible en la información de interés, evitando con esto tener que recorrer el archivo en forma secuencial hasta llegar a la posición de los datos requeridos, lo que da como resultado un menor tiempo de lectura de la información.

Para la lectura de los parámetros de los materiales se ocupa también un offset que indica la posición de inicio de la información requerida dentro del archivo, evitado recorrer el archivo de manera secuencial y realizar comparaciones para identificar el material que se busca.

Todos los anteriores planteamientos o técnicas permiten que la aplicación haga un mejor uso de los recursos del sistema, y por ende presente un mejor rendimiento.

4.1.3. Información básica y adicional para la construcción de una escena.

Del estudio de los formatos de archivo se determinó cuál es la información básica para la construcción de un modelo. En todos estos formatos se encontró que almacenan información referente a vértices y caras, pero todos estos tienen como objetivo la construcción de un objeto, ya sea siguiendo reglas de unión o mediante funciones matemáticas. Y a partir de esta información se procedió a establecer la estructura del archivo en formato GFI, siguiendo la idea de reglas de unión para la construcción de los objetos.

El archivo GFI cuenta con la unidad básica denominada objeto, el cual se identifica por un nombre. Un modelo en 3D está formado por al menos un objeto y un máximo de 4,294,967,296 objetos. Para la construcción de un objeto se requiere un conjunto de vértices, necesitando por lo menos tres vértices y un máximo de 65,536 vértices por objeto, y reglas de unión que indiquen de qué manera se unen dichos vértices, todos estos son datos que siempre deben estar presentes en el archivo en formato GFI.

Como se mencionó anteriormente los vértices son elementos necesarios para la construcción del objeto. Dentro del archivo en formato GFI, los vértices están formados por un conjunto de tres números flotantes, los cuales corresponden a las componentes X, Y y Z que definen a cada vértice dentro del área de dibujo.

Las reglas de unión son el otro componente básico para la construcción del objeto ya que determina la manera en que se unen los vértices anteriormente descritos. Las reglas de unión vienen definidas por un conjunto de tres índices, estos índices son números enteros, que indican los vértices y el orden en que éstos son unidos para formar una cara.

Dentro del archivo GFI la información de los vértices así como las reglas de unión se encuentra en el bloque correspondiente a los datos del objeto.

También para poder ser dibujado un modelo se encontró que debe existir al menos un material asociado a las caras, por lo cual se establece un material por

default que es asignado por el API diseñado, esto siempre que no encuentre un material asociado al modelo. Dicho material no es almacenado en el archivo.

La información adicional que presentan los archivos para modelos en 3D es empleada con el propósito de dar una mejor presentación y mejorar el nivel del detalle con lo que aumenta el realismo del modelo. Y esta información no siempre es la misma en cada uno de los formatos, sino que cada formato almacena la que considera como necesaria. Por ejemplo el formato 3DS almacena información de normales, materiales, cámaras, luces, en cambio el formato OBJ no posee información adicional para mejorar el aspecto del modelo. A continuación se presenta la información adicional que es almacenada dentro de un archivo en formato GFI.

Como se mencionó en el apartado correspondiente a la información básica de un modelo, existe un material por default para la presentación del modelo, sin embargo para lograr un mayor nivel de detalle se puede tener almacenada información de diferentes materiales.

Dicha información se encuentra dentro del archivo en un bloque de materiales el cual contiene los parámetros de los diferentes materiales asociados al modelo, el acceso a esta información es mediante una referencia a su posición dentro del archivo.

Cada objeto contiene una tabla de materiales asociados, la cual hace referencia por lo menos a un material y contiene una lista de caras a las cuales se les asigna dicho material, es decir, parámetros que definen la apariencia de cada cara. La información de materiales que contiene el archivo GFI contiene las componentes ambiental, difusa y especular, así como un valor para transparencia u opacidad del objeto.

También, en caso de que alguno de los materiales tenga asociada una textura, se obtendrá la información correspondiente a las coordenadas de texturización, las cuales son empleadas para colocar la imagen sobre la superficie de las caras correspondientes. Es importante considerar que cada material sólo puede tener asignada una textura, sin embargo una textura puede estar asignada a varios materiales. El API, por el momento, solo tiene soporte para texturas en formato TGA.

Otra información adicional que se almacena en los archivo en formato GFI es la jerarquía de los objetos que componen al modelo, esta información es importante ya que con ésta es posible mostrar el modelo sólo con sus características más representativas, esta es una opción cuando no sea necesario cargar el modelo en su totalidad, lo cual ayuda el desempeño de la misma, y con lo cual se maneja un nivel de detalle de la escena. La jerarquía ayudará cuando el archivo sea obtenido mediante las funciones correspondientes del API, ya que serán enviados en base a la jerarquía que presente cada objeto, lo que implica que se mostrarán los objetos con la jerarquía seleccionada.

Aun cuando no se tiene información explicita de las normales a cada cara, las reglas de unión ayudan a la obtención de éstas mediante un algoritmo que proporciona el API.

Se tiene información que ayuda al cálculo del centroide de cada objeto así como su bounding box, que puede ser útil para el cálculo de colisiones y cálculo de distancias para la visualización.

4.2. Layout GFI.

Estructura del archivo con formato GFI.

HEADER	
Número de Identificacion. 8 bytes (char *)	Número de objetos. 4 bytes (unsigned long)
Número de materiales. 4 bytes (unsigned long)	
Tamaño de la tabla de objetos y tabla de materiales. 4 bytes (unsigned long)	

TABLA DE OBJETOS						
Nombre del objeto 1. Tamaño variable en bytes. (char *)	Offset del objeto 1. 4 bytes. (unsigned long)	Tamaño del objeto 1. 4 bytes. (unsigned long)	Jerarquía de objeto 1. 2 bytes. (short)	Cmax Obj 1 Comp X. 4 bytes. (float)	Cmax Obj 1 Comp Y 4 bytes. (float).	Cmax Obj 1 Comp Z. 4 bytes. (float)
				Cmin Obj 1 Comp X 4 bytes. (float)	Cmin Obj 1 Comp Y 4 bytes. (float)	Cmin Obj 1 Comp Z 4 bytes. (float)
Nombre del objeto 2. Tamaño variable en bytes. (char *)	Offset del objeto 2. 4 bytes. (unsigned long)	Tamaño del objeto 2. 4 bytes. (unsigned long)	Jerarquía de objeto 2. 2 bytes. (short)	Cmax Obj 2 Comp X. 4 bytes. (float)	Cmax Obj 2 Comp Y. 4 bytes. (float)	Cmax Obj 2 Comp Z. 4 bytes. (float)
				Cmin Obj 2 Comp X 4 bytes. (float)	Cmin Obj 2 Comp Y 4 bytes. (float)	Cmin Obj 2 Comp Z 4 bytes. (float)
. . .	. . .	. . .	. . .	. . .	. . .	. . .
Nombre del objeto n. Tamaño	Offset del objeto n. 4 bytes.	Tamaño del objeto n. 4 bytes.	Jerarquía de objeto n. 2 bytes.	Cmax Obj n Comp X.	Cmax Obj n Comp Y.	Cmax Obj n Comp Z.

variable en bytes. (char *)	(unsigned long)	(unsigned long)	(short)	4 bytes. (float)	4 bytes. (float)	4 bytes. (float)
				Cmin Obj n Comp X 4 bytes. (float)	Cmin Obj n Comp Y 4 bytes. (float)	Cmin Obj n Comp Z 4 bytes. (float)

*** TABLA DE MATERIALES**		
Nombre del material. Tamaño variable en bytes (char *)		
Comp. ambiental R. 4 byte (float)	Comp. ambiental G. 4 byte (float)	Comp. ambiental B. 4 byte (float)
Comp. difusa R. 4 byte (float)	Comp. difusa G. 4 byte (float)	Comp. difusa B. 4 byte (float)
Comp. especular R. 4 byte (float)	Comp. especular G.4 byte (float)	Comp. especular B. 4 byte (float)
Porcentaje de transparencia 4 bytes (float)		
Nombre de la textura. Puede no haber. Tamaño variable en bytes. (char *)		

**** INFORMACIÓN DE OBJETOS**		
Número de vértices. 2 bytes (unsigned short)	Número de caras. 2 bytes (unsigned short)	
Número de materiales asignados al objeto. 2 bytes (unsigned short)	Número de coordenadas a texturizar. 2 bytes (unsigned short)	
LISTA DE VÉRTICES		
Componente X del vértice 1. 4 bytes (float)	Componente Y del vértice 1. 4 bytes (float)	Componente Z del vértice 1. 4 bytes (float)
Componente X del vértice 2. 4 bytes (float)	Componente Y del vértice 2. 4 bytes (float)	Componente Z del vértice 2. 4 bytes (float)
. . .	. . .	. . .
Componente X del vértice I. 4 bytes (float)	Componente Y del vértice I. 4 bytes (float)	Componente Z del vértice I. 4 bytes (float)
LISTA DE CARAS (REGLAS DE UNIÓN).		
Identificador vértice 1 cara 1. 2 bytes (unsigned short)	Identificador vértice 2 cara 1. 2 bytes (unsigned short)	Identificador vértice 3 cara 1. 2 bytes (unsigned short)
Identificador vértice 1 cara 2. 2 bytes (unsigned short)	Identificador vértice 2 cara 2. 2 bytes (unsigned short)	Identificador vértice 3 cara 2. 2 bytes (unsigned short)
. . .	. . .	. . .
Identificador vértice 1 cara K. 2 bytes (unsigned short)	Identificador vértice 2 cara K. 2 bytes (unsigned short)	Identificador vértice 3 cara K. 2 bytes (unsigned short)
*** LISTA DE COORDENADAS DE MAPEO		
Componente S de la coordenada de mapeo 1. 4 bytes (float)	Componente T de la coordenada de mapeo 1. 4 bytes (float)	
Componente S de la coordenada de mapeo 2. 4 bytes (float)	Componente T de la coordenada de mapeo 2. 4 bytes (float)	

. . .	. . .
Componente S de la coordenada de mapeo J. 4 bytes (float)	Componente T de la coordenada de mapeo J. 4 bytes (float)
****INFORMACIÓN DE LOS MATERIALES ASIGNADOS	
Offset del material en la lista de materiales dentro del archivo. 4 bytes (unisgned long)	
Numero de caras asignadas al material. 2 bytes (unsigned short)	
Identificador de la cara 1. 2 bytes (unsigned short)	
Identificador de la cara 2. 2 bytes (unsigned short)	
. . .	
Identificador de la cara L. 2 bytes (unsigned short)	

* Se repite tantas veces como “Número de materiales” haya.
** Se repite tantas veces como “Número de objetos” haya.
*** Puede no haber coordenadas de mapeo.
**** Se repite tantas veces como “Número de materiales asignados al objeto” haya.

Header.

Número de identificación de archivo. 8 bytes [char*] Se trata de una cadena de caracteres que identifica al archivo correspondiente con el formato GFI. Esta cadena es, “GFIV1.0” y al final se inserta un valor de fin de cadena ‘\0’.

Número de Objetos. 4 bytes [unsigned long] Número de objetos totales en el modelo.

Número de Materiales. 4 bytes [unsigned long] Número de materiales totales que se ocupan en el modelo.

Tamaño de la tabla de objetos y tabla de materiales. 4 bytes [unsigned long] se trata del tamaño en bytes que ocupa las tablas de objetos y materiales.

Tabla de objetos.

Nombre del objeto. Tamaño variable en bytes [char *] Nombre para identificar el objeto.

Offset de la información del objeto. 4 bytes [unsigned long]. Indica el inicio de la información del objeto dentro del archivo.

Tamaño del objeto. 4 bytes [unsigned long] Indica el tamaño en bytes del bloque de información del objeto.

Jerarquía. 2 bytes [short] Indica la jerarquía del objeto dentro del modelo.

Cmax 12 bytes [float, float, float]. Está definido por sus tres componentes en X, Y y Z, y cada una es un valor del tipo float. Representa los valores máximos que forman al objeto. Sirve para el cálculo del bounding box y del centroide.

Cmin 12 bytes [float, float, float]. Está definido por sus tres componentes en X, Y y Z, y cada una es un valor del tipo float. Representa los valores mínimos que forman al objeto. Sirve para el cálculo del bounding box y del centroide.

Tabla de Materiales.

Nombre del Material. Tamaño variable en bytes [char *] Nombre para identificar al material.

Componente Ambiental RGB. Cada componente es de 4 bytes [float] Componente de Color.

Componente Difusa RGB. Cada componente es de 4 bytes [float] Componente de Color.

Componente Especular RGB. Cada componente es de 4 bytes [float] Componente de Color.

Porcentaje de Transparencia. 4 bytes [float] Indicador de transparencia para efectuar el blending.

Nombre de Textura. Tamaño variable en bytes [char *] Nombre de la textura asociada al material.

Bloque Información de Objetos.

Número de vértices. 2 bytes [unsigned short]. Número de vértices con que está formado el objeto.

Número de caras. 2 bytes [unsigned short]. Número de reglas de unión para formar las caras del objeto.

Número de materiales asignados al objeto. 2 bytes [unsigned short]. Número de materiales que serán aplicados a las caras del objeto.

Número de coordenadas de mapeo. 2 bytes [unsigned short]. Número de coordenadas de mapeo usadas para texturizar las caras del objeto.

Lista de Vértices. Dentro de cada bloque de objetos existe una *lista de vértices* con una cantidad de elementos establecida por *Número de vértices,* la información de cada vértice se encuentra representada de la siguiente manera.

Componentes del vértice X, Y, Z. Cada componente es de 4 bytes [float].

Lista de Caras o Reglas de Unión. Dentro de cada bloque de objetos existe una *lista de caras o reglas de unión* con una cantidad de elementos establecida por *Número de caras,* cada cara o regla de unión está formada de la siguiente manera.

Identificadores de los vértices 1, 2, 3, … . Cada identificador es de 2 bytes [unsigned short]. El orden en que se encuentran los identificadores establece cómo es formada la cara.

Lista de Coordenadas de Mapeo. Dentro de cada bloque de objetos puede existir una *lista de coordenadas de mapeo*, dependiendo del valor *de coordenadas de mapeo,* cada coordenada de mapeo está compuesta de la siguiente manera.

Componentes de la coordenada S, T. Cada componente es de 4 bytes [float].

Información de los Materiales Asignados. Dentro de cada bloque de objetos puede existir un bloque de *información de los materiales asignados***,** dependiendo de la cantidad de elementos establecida por *Número de materiales asignados,* en caso de que no haya material asignado la aplicación BlackMageLoader asigna un material por default al objeto. Cada elemento del bloque de los materiales asignados contiene la siguiente información.

Offset del material en la lista de materiales dentro del archivo. 4 bytes [unsigned long]. Información de la posición del inicio de los parámetros del material correspondiente.

Número de caras asignadas al material. 2 bytes [unsigned short]. Establece el número de caras del objeto a las cuales se les aplicarán los parámetros del material.

Lista de los Identificadores. Dentro de cada bloque de información de los materiales asignados hay una *lista de los identificadores de las caras* con una cantidad de elementos establecida por *Número de caras asignadas al material.* Cada identificador se encuentra como sigue.

Identificador de la cara. 2 bytes [unsigned short].

4.3. Estructuras de datos para la carga a memoria principal.

En este capítulo se trata de las estructuras empleadas para guardar la información de los modelos, ya sea que se encuentren en un archivo en formato GFI o 3DS. [1]

Cada modelo cuenta con tres listas ligadas (Fig. 4.3.a) que son las principales para poder ser almacenados en memoria, estas listas son: la lista que representa a la Tabla de Objetos, la lista que representa la Tabla de Materiales y la lista donde se guardan la información correspondiente a los objetos que forman al modelo.

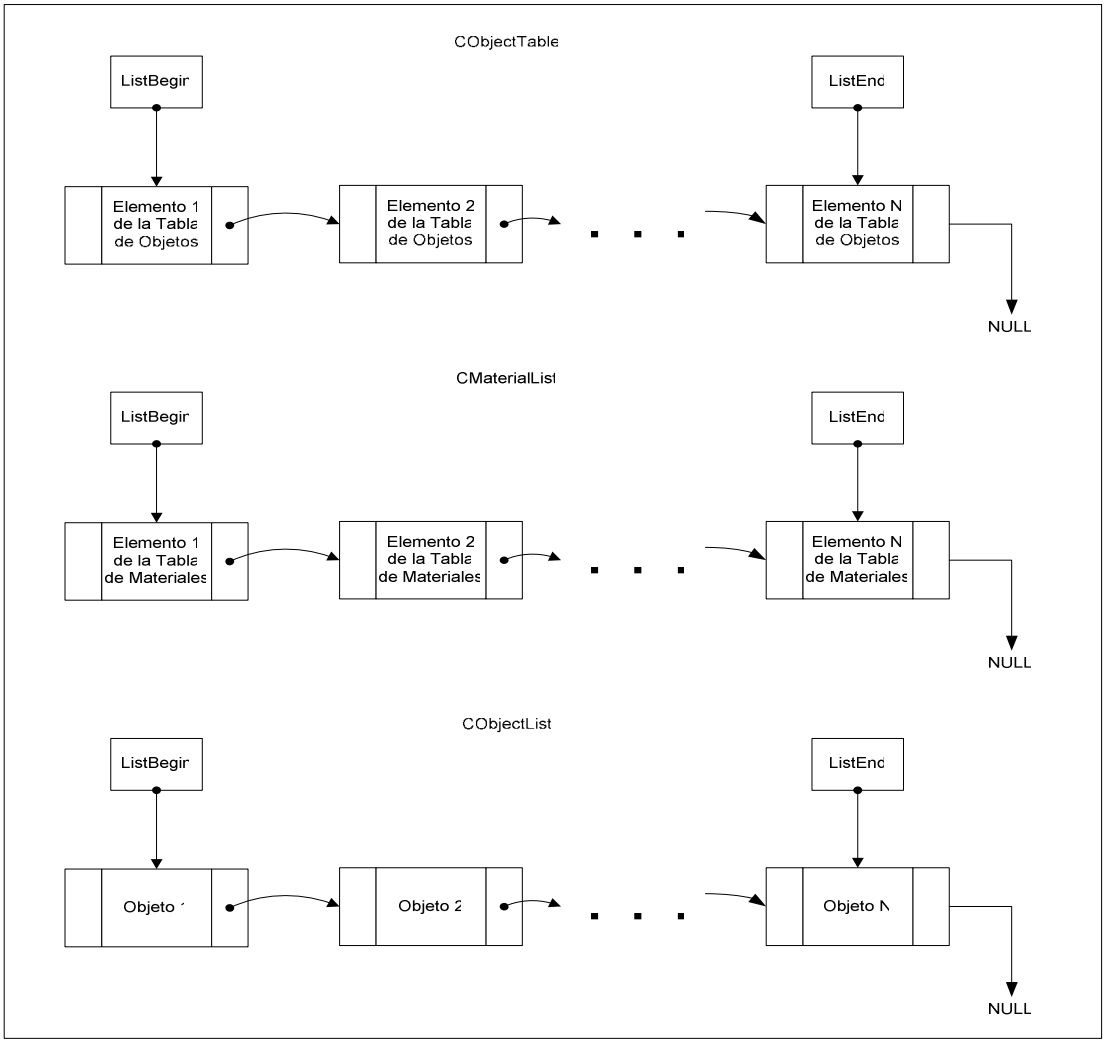


Fig. 4.3.a. Diagrama que muestra las 3 listas principales empleadas para el guardado de un modelo.

Cada elemento dentro de la Tabla de Objetos cuenta con una referencia que indica al objeto correspondiente en la lista de Objetos y que apunta al inicio de los datos del objeto, así mismo se tiene una referencia desde la lista de Objetos hacia la Lista que forma la Tabla de Objetos.

La lista correspondiente a la Tabla de Materiales cuenta con una referencia que indica el elemento de la lista de Imágenes (Fig. 4.3.b) que le corresponde usar como textura. Cabe señalar que cada elemento de la Tabla de Materiales sólo puede tener una referencia a una sola imagen, pero una misma imagen puede estar referida por varios elementos de la Tabla de Materiales.

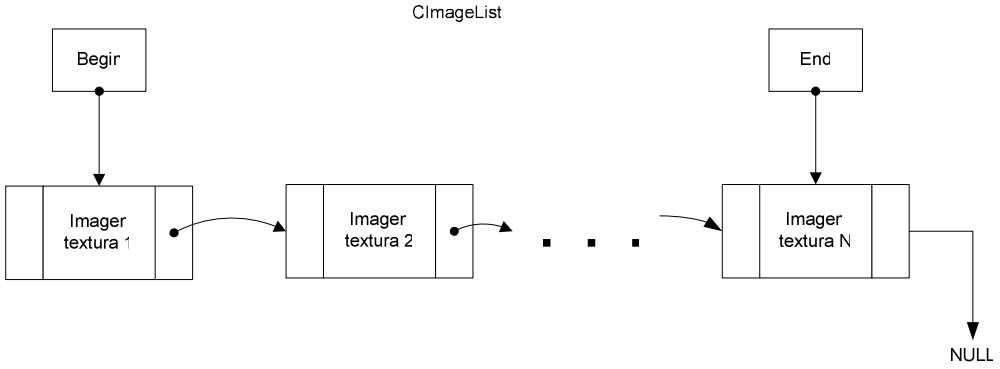


Fig. 4.3.b. Diagrama que ilustra la lista de imágenes las cuales serán empleadas como texturas.

Cada elemento que forma a la lista de Objetos señala a una lista única, es decir, cada elemento cuenta con una lista diferente, dicha lista corresponde a los materiales que son asociados a las caras del objeto. (Fig. 4.3.c)

Cada elemento de esta lista de Materiales Asociados, hace referencia a un solo elemento de la Tabla de Materiales.

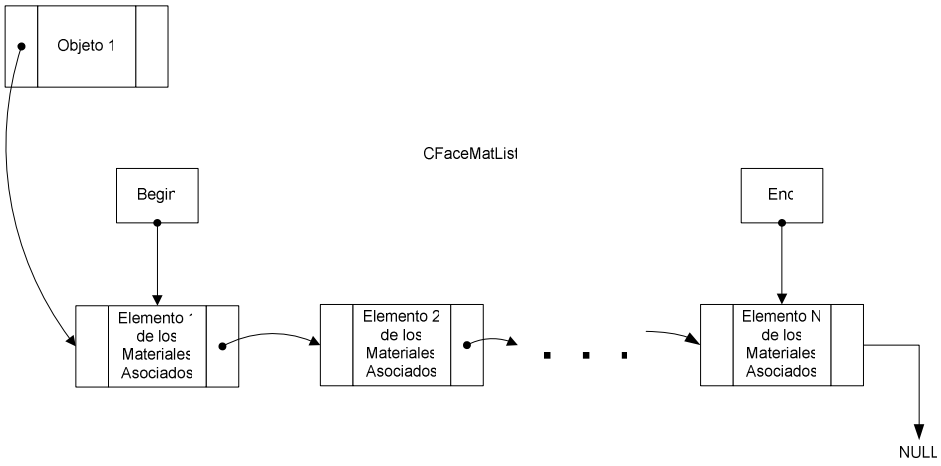


Fig. 4.3.c. Diagrama que ilustra la lista de Materiales Asociados que contiene cada elemento del modelo.

4.4. Carga de archivos en formato GFI de manera local.

4.4.1. Carga de Encabezado y Tablas Índice.

Los archivos en formato GFI están diseñados para ser cargado de varias maneras. Existe cierta información que debe ser proporcionada por el archivo que se desea cargar, esta información es el encabezado y las tablas índice que son de suma importancia, por lo cual es necesario contar con esta información antes de llevar a cabo cualquiera de las diversas formas de carga de modelo tridimensional en formato GFI.

Para llevar a cabo la carga de dicha información lo primero que se realiza es la creación de una instancia de un modelo con el objetivo de generar el espacio en memoria RAM para almacenar la información en las estructuras correspondientes.

Una vez que se ha asignado la memoria dinámica, se procede a realizar la carga del Header del archivo. La secuencia de carga del Header es como se muestra en la siguiente figura (4.4.1.a).

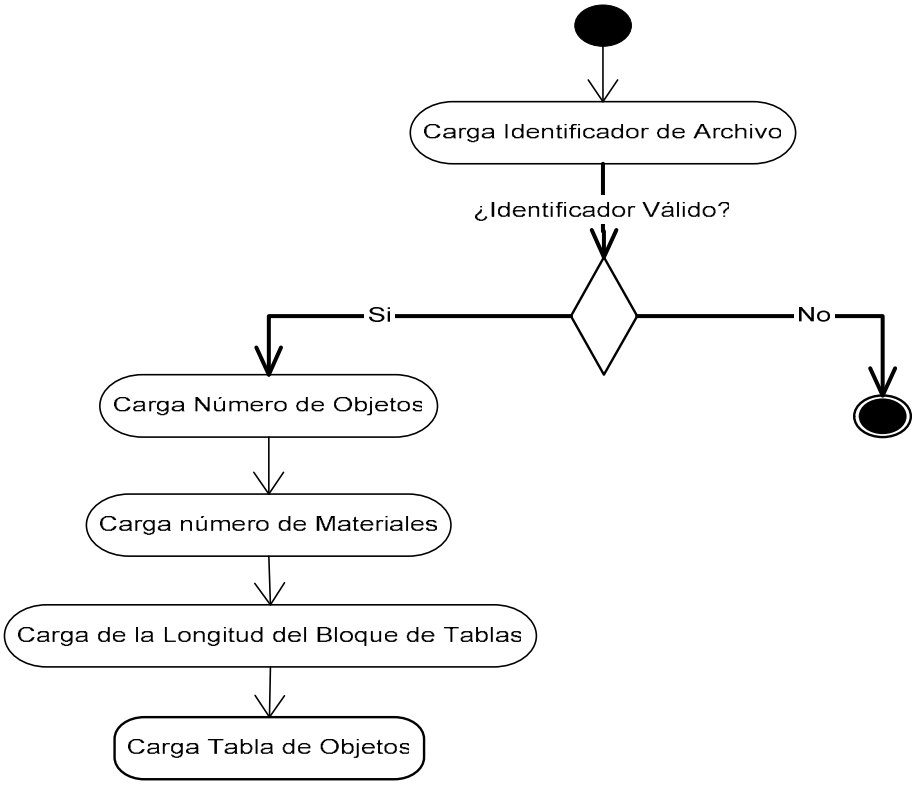


Fig 4.4.1.a. Diagrama UML de lectura de Header.

La lectura del Header tiene varios propósitos uno de ellos es la validación del archivo, es decir, asegurarse de que el archivo que está siendo procesado para ser cargado sea en efecto un archivo en formato GFI válido, el identificador es una cadena terminada en cero de 8 caracteres “GFIV1.0”.

Ya que se ha confirmado que el archivo corresponde a un formato GFI, los siguientes datos como son: el Número de Objetos, el Número de Materiales y el tamaño de las Tablas, compuesto por el tamaño de la Tabla de Objetos más el tamaño de la Tabla de Materiales; se almacenan en sus estructuras correspondientes (ver capitulo 4.3).

Una vez que se ha terminado con la lectura del Header, se procede a la carga de la Tabla de Objetos, tal proceso se lleva a cabo como se muestra en la figura (4.4.1.b).

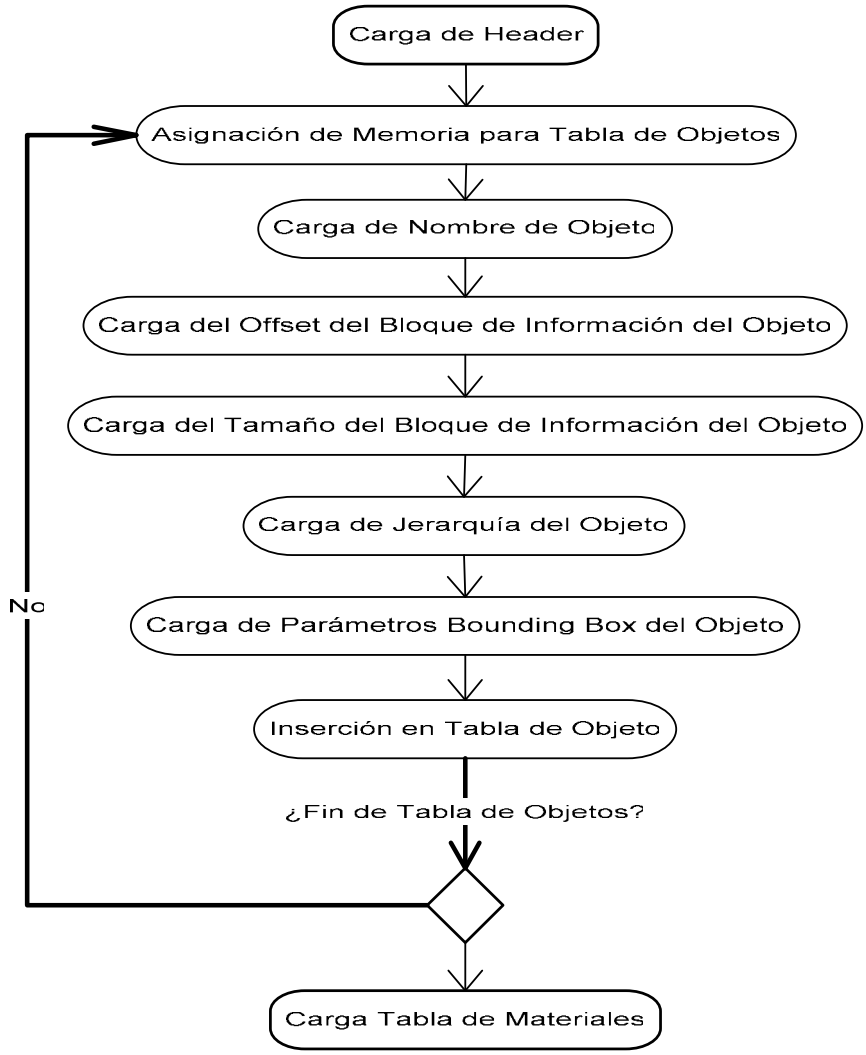


Fig 4.4.1.b. Diagrama UML de lectura de la Tabla de Objetos.

Antes de empezar con la carga de datos correspondientes a la Tabla de Objetos se debe de asignar memoria de manera dinámica para las estructuras que almacenarán dicha información. Esa porción de memoria corresponde a una instancia que representa a un elemento de la Tabla de Objetos.

Todo el proceso de carga se lleva a cabo dentro de un ciclo, el cual está determinado por el número de objetos, dato que se obtuvo de la lectura del Header.

El primer dato a ser cargado es el nombre que identifica a cada elemento de la Tabla de Objetos. Este dato es una cadena, para ser cargado en la Tabla de Objetos se leen los bytes hasta antes de encontrar un byte correspondiente a un valor de cero. La cadena encontrada hasta ese momento será el dato cargado.

Se continúa con la carga del offset del objeto, del dato correspondiente al tamaño en bytes del bloque de datos del objeto y su jerarquía.

Para terminar con la carga de la Tabla de Objetos, se almacenan las coordenadas que indican el valor máximo y el valor mínimo para la construcción del bounding box asociado al objeto.

Una vez que la instancia ha sido llenada con la información, se considera que el elemento ya puede ser insertado a la lista que forma la Tabla de Objetos. Como se indicó, el proceso antes descrito se realiza en un ciclo hasta terminar con la carga de todos los elementos que formarán la Tabla Objetos.

Una vez hecho lo anterior, se procede a realizar la carga de la Tabla de Materiales tal como se muestra en la figura (4.4.1.c).

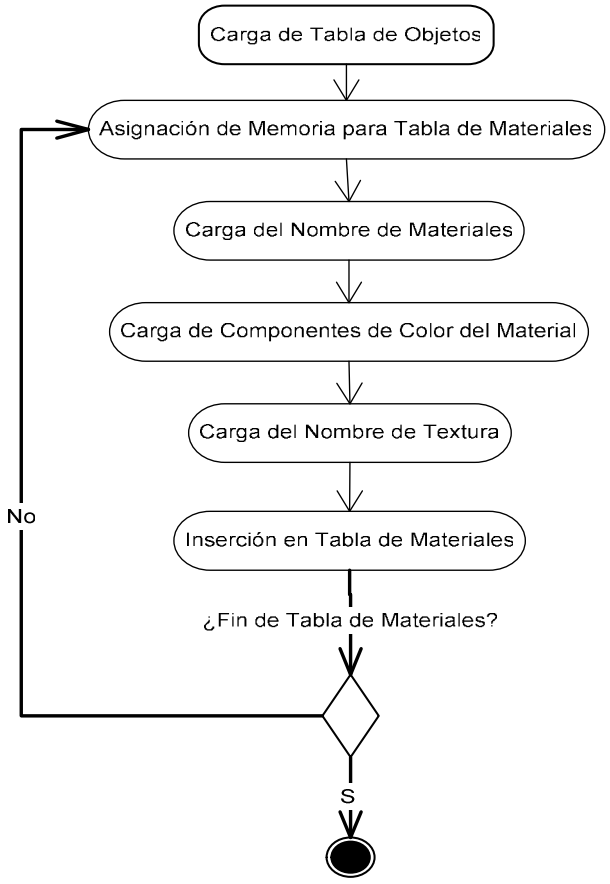


Fig 4.4.1.c. Diagrama UML de lectura de la Tabla de Materiales.

Se asigna memoria de manera dinámica para los elementos que formarán la lista de materiales, es decir, se crea una instancia. Se vuelve a realizar un ciclo de carga para la obtención de los datos del material, dicho ciclo se repetirá de acuerdo al número de materiales, obtenido de la lectura del Header del archivo. De igual manera, como ocurre con los elementos de la Tabla de Objetos, cada elemento de la Tabla de Materiales es insertado a la lista sólo hasta que es llenado con la información correspondiente.

El primer dato cargado en la instancia es el que servirá como identificador del material dentro de la Tabla de Materiales.

El siguiente dato a cargar es el que corresponde al nombre del material y que es un proceso similar al llevado a cabo para la carga del nombre del objeto, es decir, se cargará la cadena encontrada hasta antes del byte correspondiente al valor de cero.

Los siguientes datos cargados son los que proporcionan los parámetros que indican el aspecto que tendrá el material, dichos parámetros son: componente

ambiental, componente difusa, componente especular y un parámetro alfa. Este último valor se toma como el cuarto parámetro para complementar a las componentes mencionadas, dado que dentro del archivo se especifican las componentes con tres parámetros pero para la carga se requiere de cuatro parámetros.

El último dato que se carga para completar la información del elemento perteneciente a la Tabla de Materiales es el nombre de la textura, que puede ser una cadena o un valor nulo.

Y para concluir se inserta el elemento dentro de la lista que representa la Tabla de Objetos. El ciclo termina cuando se encuentran insertados todos los elementos que forman la Tabla de Objetos.

Toda la información ya cargada, como es el Header y las Tablas, requiere de un tamaño muy pequeño, en comparación con la información completa del modelo, para ser almacenada en memoria dentro de sus estructuras correspondientes, lo cual resulta útil para algunas de las diversas manera de carga de un modelo tridimensional en formato GFI.

El proceso anterior se puede resumir en la siguiente figura (4.4.1.d).

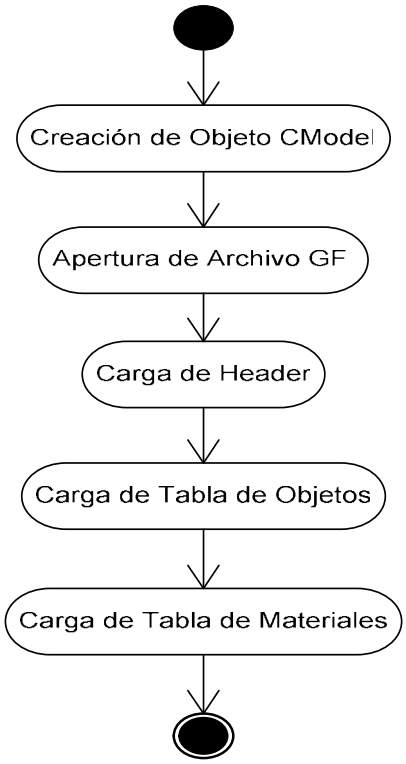


Fig. 4.4.1.d. Diagrama UML para la carga del Header y las Tablas Índices

4.4.2. Proceso de carga de archivo GFI completo.

En este apartado se tratará el caso de la carga completa de los archivos GFI, dado que parte de este proceso ya ha sido realizado con la carga del header y de las tablas índice, ahora se describirán los demás procesos que permiten la carga completa de la información poligonal restante.

Cabe señalar que cuando se realiza la carga de manera completa, el proceso de lectura de la información continúa prácticamente en la posición donde se ha quedado el apuntador a memoria, que corresponde a la copia del archivo GFI. Esto se ilustra de manera general en la figura (4.4.2.a).

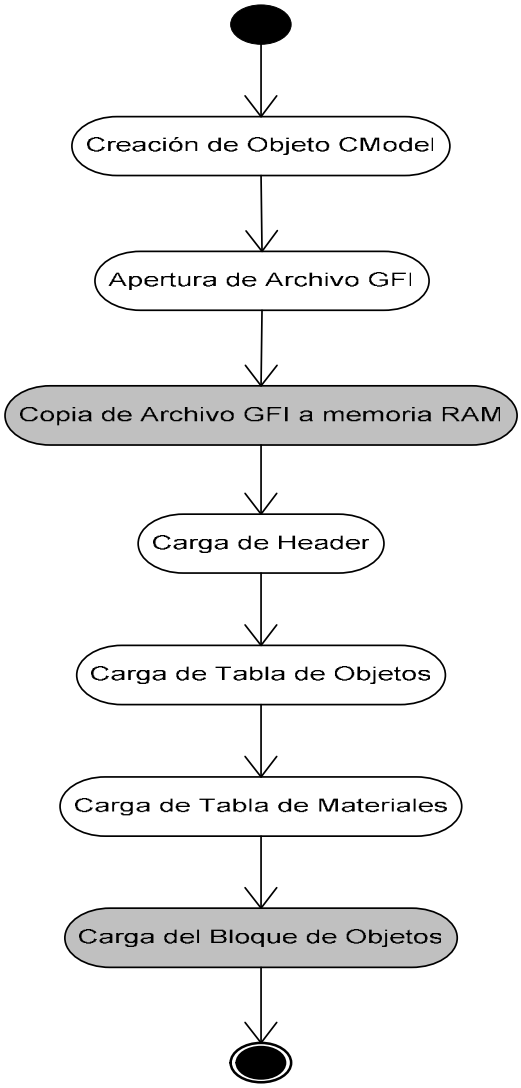


Fig 4.4.2.a. Diagrama UML para la carga completa de archivos GFI.

El Bloque de Objetos es la parte importante del archivo, ya que en ella se encuentra toda la información poligonal y que permite la construcción del modelo.

La lectura del Bloque de Objetos, cuando se realiza la carga del archivo GFI completo, se lleva a cabo de manera secuencial, y es aquí donde se distingue de las otras maneras de carga del archivo GFI, ya que cuando se desea tener la carga del archivo completo, la lectura se realiza dentro de un ciclo, en donde se leen los datos de todos y cada uno de los objetos que componen al modelo.

En este proceso se hace uso de la Tabla de Objetos cargada previamente, de ésta se obtiene el offset que indica la posición en la cual se encuentra el inicio de la información del elemento a cargar. Esto con el fin de garantizar que la información que será cargada corresponde precisamente al elemento de la Tabla de Objetos que se cargará.

El proceso de carga de la información de un solo elemento se muestra en la figura (4.4.2.b).

Se crea una instancia, es decir, se asigna memoria de manera dinámica para alojar toda la información que corresponde al elemento de la Tabla de Objetos al que se hace referencia para que sea cargado.

Se carga la información correspondiente al Número de Vértices, al Número de Caras, al Número de Materiales Asignados y el Número de Coordenadas de Texturización.

Una vez conocido el Número de Vértices se asigna memoria de forma dinámica para crear un arreglo, el cual almacenará los vértices. Para la carga de tales vértices se requiere de un ciclo que está determinado por la cantidad de vértices de que está compuesto el objeto. Los vértices se van cargado con sus tres componentes correspondientes a un vector de posición de tres dimensiones.

Cuando se ha terminado el ciclo para la carga de los vértices se asigna memoria dinámica, para crear un arreglo con una longitud igual al Número de Caras. De igual manera se crea un arreglo con la misma longitud para almacenar los vectores normales a cada una de las caras.

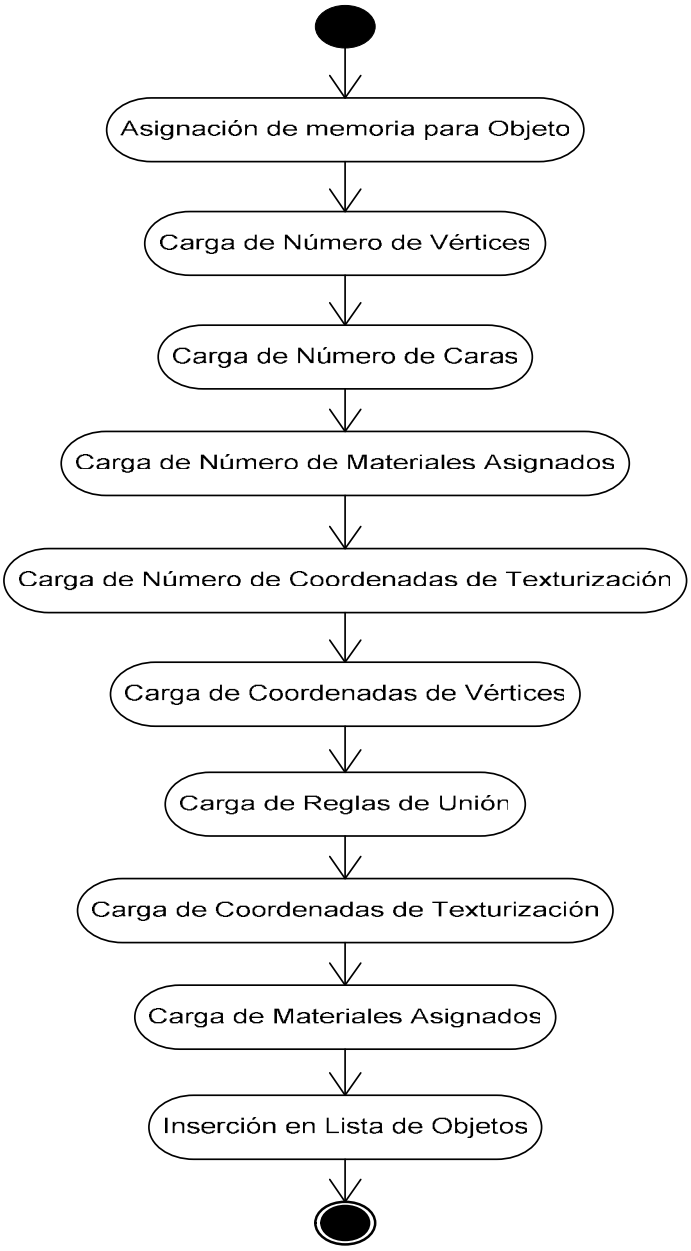


Fig. 4.4.2.b. Diagrama UML de lectura de un Bloque de datos correspondiente a un Objeto.

Para la carga de esta información se requiere leer los tres vértices que definen cada cara. Cada vez que se carga una cara esta viene representada por los identificadores de sus vértices que la forman. Tal identificador corresponde con el índice del arreglo de vértices, que fue llenado con anterioridad. Las componentes de los vértices son empleadas para llevar a cabo el cálculo del vector normal de dicha cara. Este vector normal se va cargando al arreglo de vectores normales con el mismo identificador que la cara para poder asociar.

Lo anterior se repite tantas veces como Número de Caras contenga el objeto.

Una vez completado el proceso de carga de las caras o reglas de unión, se continúa a realizar la carga de las coordenadas de texturización. Para esto primero se realiza una verificación del Número de Coordenadas de Texturización asociadas a este objeto, esto se determina si el Número de Coordenadas de Texturización es mayor a cero.

Si hay Coordenadas de Texturización se crea una instancia, es decir, se asigna memoria de manera dinámica para la creación de un arreglo de dos dimensiones con una longitud igual al Número de Coordenadas de Texturización.

Una vez creado el arreglo, se va guardando en cada elemento del arreglo una Coordenada de Texturización, cada una de ellas está formada por dos componentes. Este proceso se realiza dentro de un ciclo el cual está dado por el Número de Coordenadas de Texturización.

Ya terminada la carga de Coordenadas de Texturización, se procede con la carga de la información referente a los materiales asociados al objeto. Para ello se reserva espacio en memoria, el cual corresponde a la información de un solo material asociado, dicho proceso se ilustra en la siguiente figura (4.4.2.c).

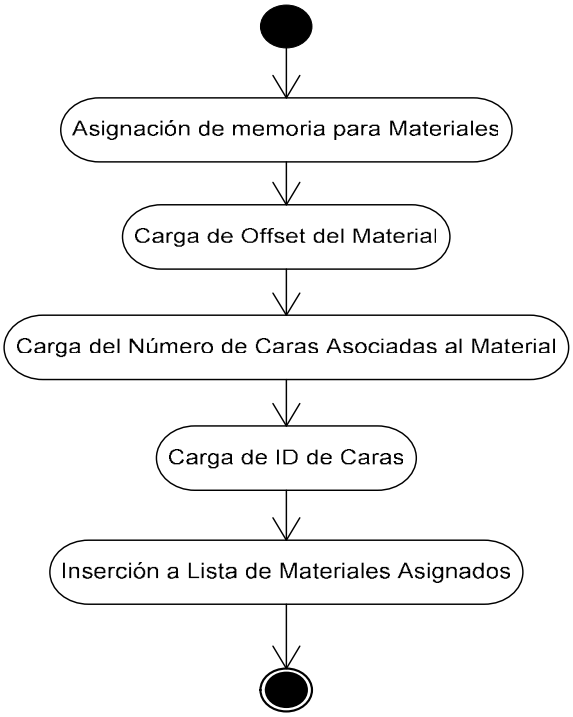


Fig. 4.4.2.c. Diagrama UML de lectura de Materiales Asignados.

Se carga a esta instancia el valor que indica la posición en memoria de los parámetros del material asociado, el Número de Caras a las que se les asocia dicho material.

Conocido el Número de Caras que llevará asociadas dicho material se asigna memoria para crear un arreglo que almacenará los identificadores de dichas caras. En cada elemento del arreglo se guardará un identificador de cara, esta operación se realizará dentro de un ciclo que está determinado por el Número de Caras asociadas al material.

Para poder obtener los parámetros del material asociado es necesario recorrer la Tabla de Materiales, realizando una comparación entre el identificador del material que se desea con los identificadores de los materiales que se encuentran dentro de la Tabla de Materiales. Una vez encontrado el material deseado se guarda la referencia al mismo, para posteriormente poder obtener los parámetros del material. Se asigna memoria para guardar el nombre del material el cual es obtenido de la Tabla de Materiales.

Ya que se ha cargado la información de los materiales asociados se inserta la instancia a la lista de materiales asignados.

El proceso anterior se lleva a cabo tantas veces como Número de Materiales asociados tenga el objeto.

Con todos los datos del bloque de objeto guardados dentro de la instancia del objeto, ésta procede a ser insertada en la lista de objetos.

Todo este proceso se realiza conforme al número de objetos que forman la tabla de objetos.

El código siguiente en C++ carga completamente un archivo GFI a memoria principal y lo dibuja en un “Rendering Context” de OpenGL

```
#include "CModel.h"

CModel modelo;                          //se crea un objeto de la clase CModel
if(!modelo.GFILoad("c:\modelo.gfi"))    //se intenta carga el modelo GFI
      exit(0);                          //si falla, el programa sale
modelo.VertexNormals();                 //se calculan las normales a los vértices
modelo.LoadTextureImages();             //se cargan las imágenes que compondrán
                                        //las texturas
modelo.GLIniTextures();                 //OpenGL crea texturas a partir de las
                                        //imágenes

void RenderGLscene()
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();

      modelo.GLrender(NULL, SHADED);    //se dibuja el modelo sin
                                        //faceculling con el shader por default
}
```

4.4.3. Proceso de Carga de archivo GFI por partes.

4.4.3.1. Carga de archivo GFI por nombre de objeto.

El primer paso para realizar este proceso es la creación de una instancia, es decir, la asignación de memoria requerida para el guardado de las estructuras donde se almacenará la información del modelo. Una vez con la instancia del modelo, se procede a la apertura del archivo GFI para su lectura, si la apertura del archivo fue exitosa, se continúa con el proceso de carga.

La carga de objetos por su nombre se realiza mediante la obtención de la información directamente desde el archivo GFI, esto quiere decir que no se realiza copia a memoria RAM del archivo. La carga de las tablas índice se realiza con el fin de conocer los objetos con que está formado el modelo, y de esta manera poder indicar el objeto que se desea cargar.

La diferencia principal, con respecto al método anterior, radica en la lectura del Bloque de Objetos. Ya que solamente se realizará la lectura de este bloque cuando se indique el nombre del objeto que se desea cargar, y únicamente se obtendrán los datos del objeto indicado. (Fig 4.4.3.1.a)

Se debe indicar el nombre del objeto que se desea cargar. Se procede a realizar una comparación de este nombre con los nombres almacenados en la Tabla de Objetos, se va recorriendo la Tabla de Objetos elemento por elemento, si el nombre que está siendo comparado es diferente al nombre buscado, se continua la comparación con el siguiente elemento de la tabla. Una vez que se ha encontrado el objeto al cual corresponde este nombre, se obtiene el valor del offset almacenado en la Tabla de Objetos para el objeto que resultó seleccionado.

El valor de offset obtenido permite que el apuntador al archivo GFI, con que se está trabajando actualmente, se posicione al inicio de la información del objeto seleccionado. Con lo cual se realiza la lectura de los datos y su almacenamiento a las estructuras como se indica en la figura (4.4.3.1.b). La parte de la carga del bloque de información del objeto se realiza de manera similar a la realiza en la carga del archivo GFI completo, con la salvedad de que ahora la información es leída directamente del archivo GFI, y no de una copia en memoria RAM.

Cada vez que se realice la selección de un objeto, se procede a realizar la obtención de los datos del mismo, siempre y cuando dicho objeto no se encuentre cargado.

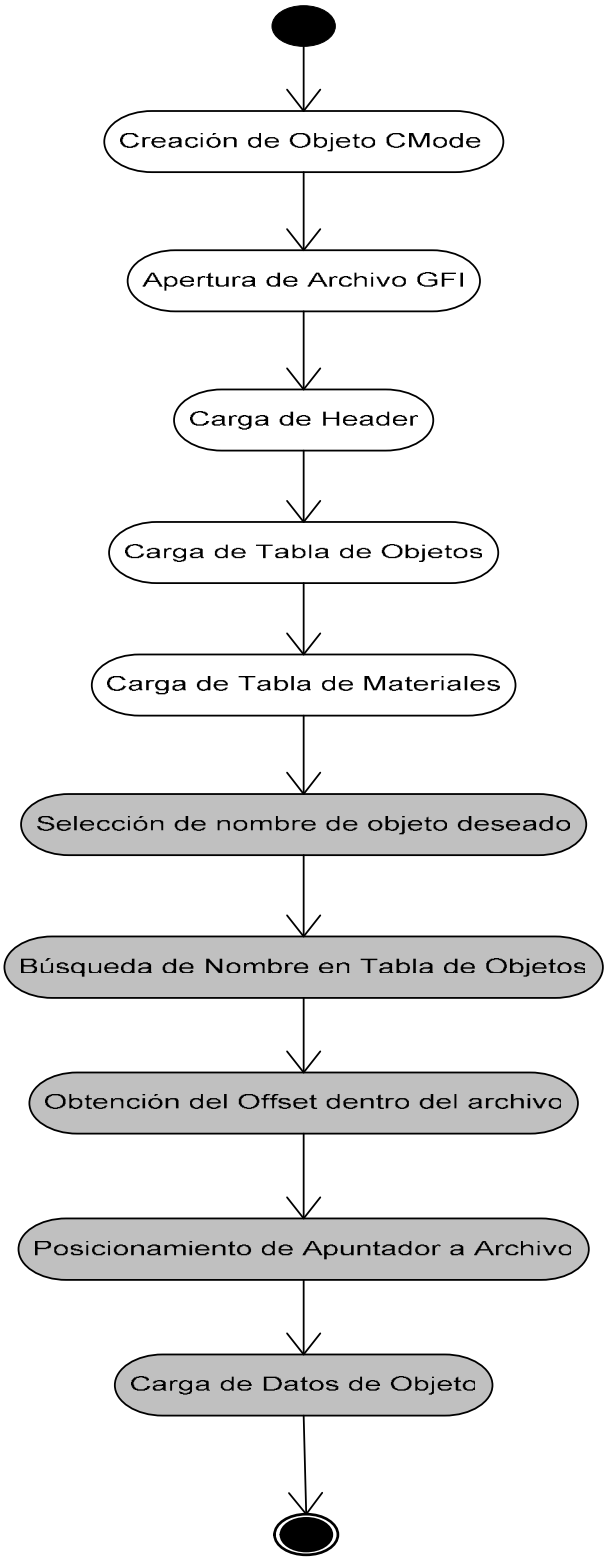


Fig 4.4.3.1.a Diagrama UML de Carga por nombre de objeto.

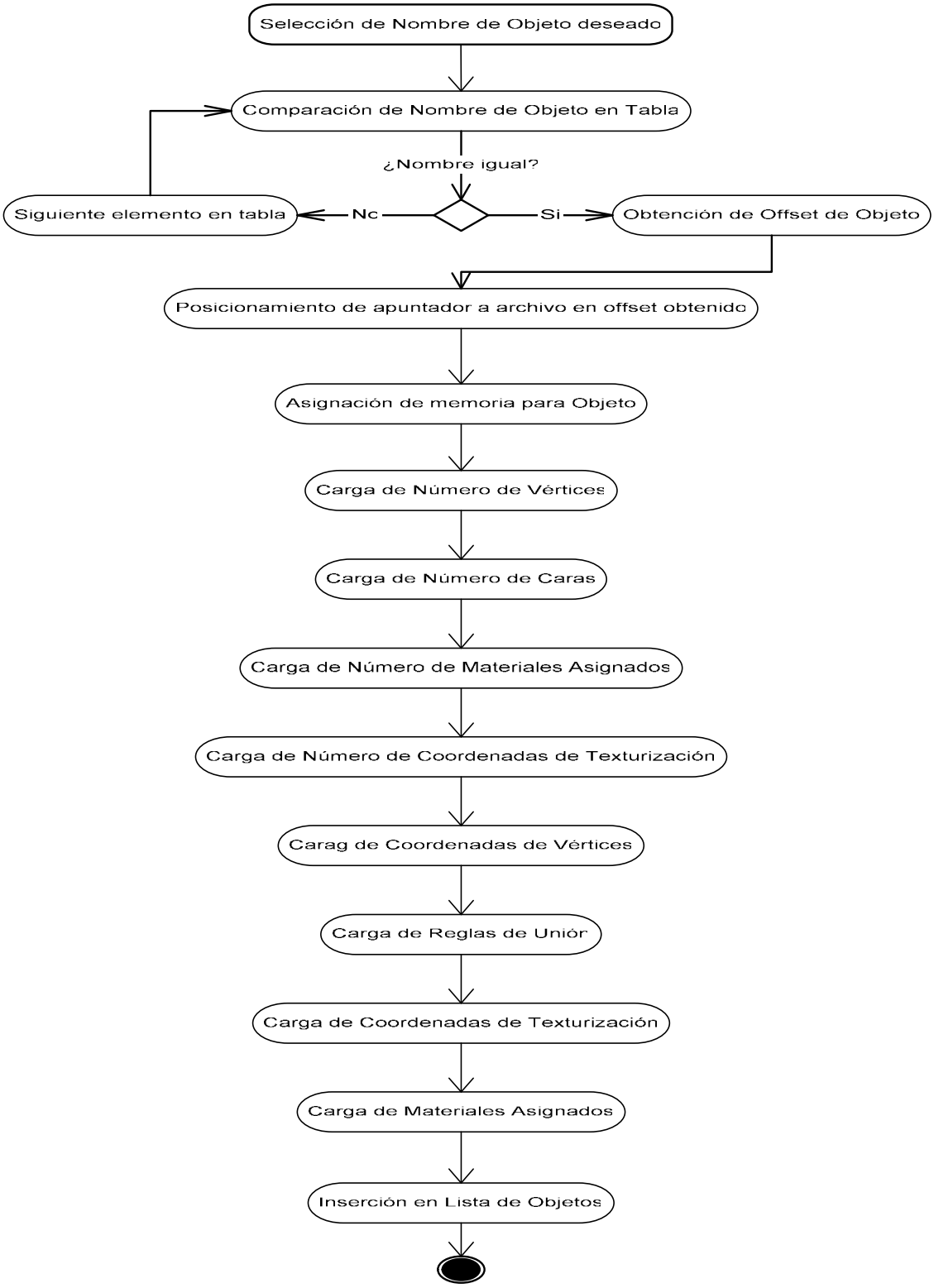


Fig. 4.4.3.1.b. Diagrama UML para lectura de datos de objeto.

El siguiente codigo en C++ muestra un como se usa el API de programación desarrollado para cargar selectivamente algunos de los objetos del modelo guardado en un archivo GFI.

```
#include "CModel.h"

char objname[256];              //arreglo temporal para almacenar el nombre
CModel modelo;                  //se crea un objeto CModel

if(¡modelo.GFIStartSession("c:\modelo.gfi")) //intentamos cargar las tablas
                                             //indice del modelo

       exit(0);

//se itera a través de la tabla índice de objetos

modelo.ObjectTable.current= modelo.ObjectTable.ListBegin();
while(modelo.ObjectTable.current!=NULL)
{
       //se carga el objeto a través de su nombre
       objname= modelo.ObjectTable.current->GetName();
       modelo.GFILoadObject(objname);

       modelo.ObjectTable.current= modelo.ObjectTable.current->NextField();
}



//-------------------------------------------------------------------------------



#include "CModel.h"

CModel modelo;                                  //se crea un objeto CModel

if(¡modelo.GFIStartSession("c:\modelo.gfi")) //intentamos cargar las tablas
                                             //indice del modelo
       exit(0);

//se itera a través de la tabla índice de objetos

modelo.ObjectTable.current= modelo.ObjectTable.ListBegin();
while(modelo.ObjectTable.current!=NULL)
{
       //se carga el modelo por medio de la referencia que contiene la tabla
       //este método es mucho mas rápido
       modelo.GFILoadObject(modelo.ObjectTable.current);

       modelo.ObjectTable.current= modelo.ObjectTable.current->NextField();
}
```

4.4.3.2. Carga de Modelo GFI por jerarquía de objeto.

Esta otra forma de carga de objetos que forman al modelo toma como parámetro a considerar la jerarquía que tiene asignada el objeto. Este tipo de búsqueda es un ejemplo de que se pueden realizar cargas a través de diferentes parámetros contenidos dentro de la Tabla de Objetos, no solo se dependen del nombre que identifica a cada objeto.

La implementación de esta carga se basa en el algoritmo seguido para la Carga de Modelo GFI por nombre de objeto, con algunas modificaciones. Lo que demuestra la característica de reutilización de las funciones de carga.

En esta fase, este tipo de búsqueda solo distingue entre los objetos que tienen jerarquías primarias, que son representadas con el valor (-1), de los objetos con jerarquías secundarias, cualquier otro valor diferente a (-1). Sin embargo, debido a la misma reutilización del código, es posible modificar la función de búsqueda para distinguir entre un mayor número de jerarquías o incluso poder realizar la carga a partir de un árbol de jerarquías.

La figura (4.4.3.2.a) muestra el proceso que se sigue para realizar esta carga, así mismo se pueden apreciar las diferencias con los anteriores tipos de carga.

La lectura del Header, de la Tabla de Objetos, y de la Tabla de Materiales es idéntica a la realizada en los procesos de carga anteriores, y al igual que la carga por objeto la información es leída directamente del archivo. El parámetro para realizar la búsqueda de los elementos deseados es ahora la jerarquía del objeto, y al igual que el parámetro nombre del objeto, este valor se encuentra incluido dentro de la Tabla de Objetos. Por lo que la búsqueda por jerarquía es similar a la búsqueda por nombre de objeto, con la salvedad de que ahora, debe recorrerse todos los elementos de la tabla para identificar los objetos correspondientes a la jerarquía buscada. (Fig. 4.4.3.2.b)

Cada vez que se encuentre que el elemento de la Tabla de Objetos cumpla con el valor de jerarquía deseado, se obtiene el offset del objeto dentro del archivo. La carga de los datos de cada objeto se realiza al momento de identificar la jerarquía, una vez cargados los datos del objeto se vuelve a realizar la comparación con el siguiente elemento de la Tabla de Objetos hasta que se recorra la tabla completa. Otra forma de carga puede ser el guardar los offsets correspondientes a los objetos que cumplan con la jerarquía y realizar la carga de todos los objetos al mismo tiempo.

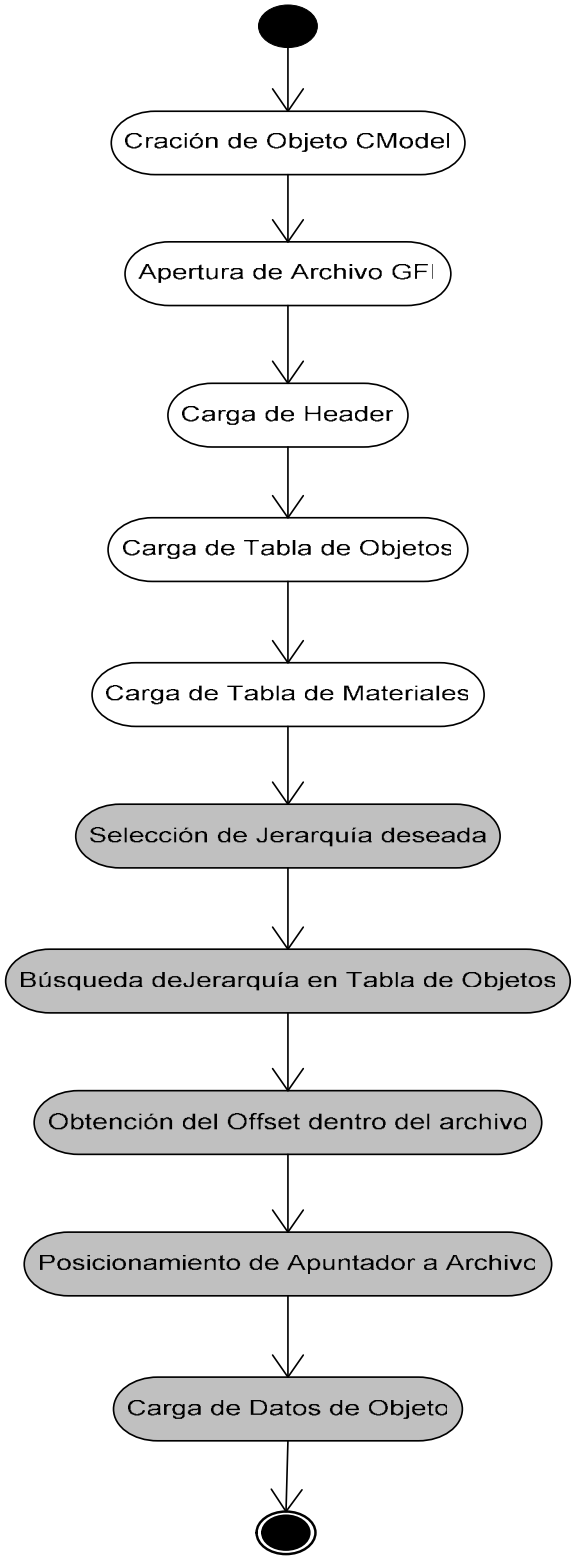


Fig. 4.4.3.2.a. Diagrama UML de Carga por Jerarquía.

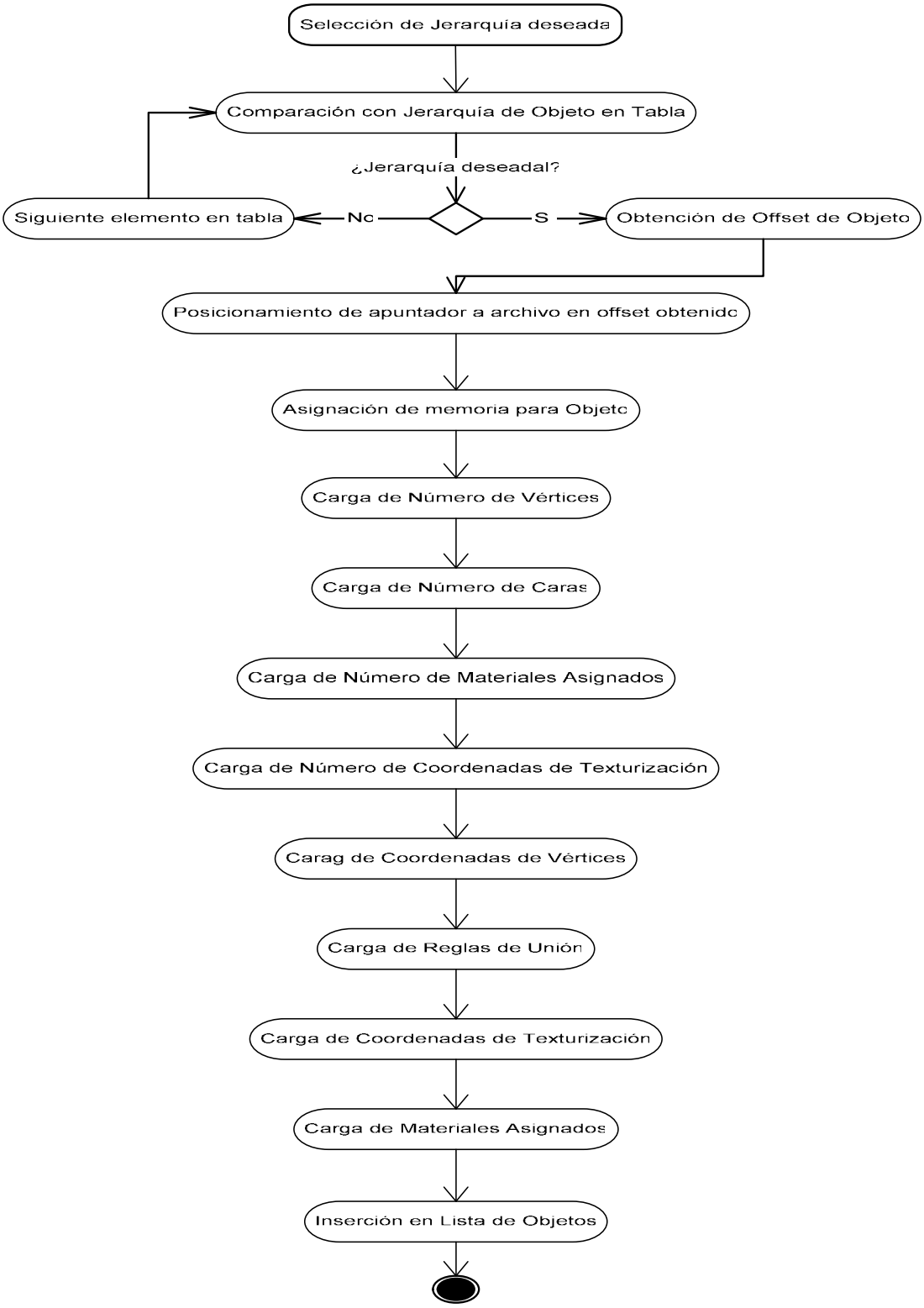


Fig. 4.4.3.2.b. Diagrama UML para lectura de datos de un objeto para carga por Jerarquía.

El siguiente código muestra como obtener el valor de la jerarquía del objeto y usarla para saber si el objeto debe ser cargado

```
#include "CModel.h"

CModel modelo;                                  //se crea un objeto CModel

if(¡modelo.GFIStartSession("c:\modelo.gfi")) //intentamos cargar las tablas
                                                //indice del modelo
       exit(0);

//se itera a través de la tabla índice de objetos

modelo.ObjectTable.current= modelo.ObjectTable.ListBegin();
while(modelo.ObjectTable.current!=NULL)
{
       //se carga el objeto solo si pasa el criterio de jerarquía
       if(modelo.ObjectTable.current->GetHierarchy () >=2 )
              modelo.GFILoadObject(modelo.ObjectTable.current);

       modelo.ObjectTable.current= modelo.ObjectTable.current->NextField();
}
```

4.4.3.3. Modelo GFI por distancia del observador al objeto.

La carga por distancia, está determinada por lo cerca o lejos que se encuentra el observador respecto a los puntos máximos y mínimos del bounding box que corresponde a un determinado objeto. Es decir, que los objetos se van cargando conforme el observador se acerca al objeto. Dicho proceso se ilustra de forma general en la figura (4.4.3.3.a).

De igual manera como ocurre con los métodos de carga anteriores lo primero que se realiza es la creación de una instancia, la cual corresponde al las estructuras y demás instancias que serán creadas para almacenar la información de un modelo.

Una vez asignada la memoria para el modelo se abre el archivo que contiene la información poligonal, y se realiza la lectura del Header, la cual implica la validación del archivo; y la carga de las Tablas, que corresponde a la Tabla de Objetos y a la Tabla de Materiales.

Para la carga de cada uno de los objetos, primero se obtiene de la Tabla de Objetos los puntos correspondientes al bounding box del objeto. Con dichos vectores se realiza el cálculo de la distancia al punto de observación.

Si la distancia calculada cumple con las condiciones de una determinada longitud, entonces se procede con la carga del elemento que cumple con las condiciones de distancia y que no se encuentre cargado. Este proceso empieza obteniendo el offset del objeto dentro del archivo obtenido de la Tabla de Objetos,

con esta información el apuntador del archivo se posiciona al inicio del bloque Objeto y realizar el proceso de carga como ya se ha descrito anteriormente.

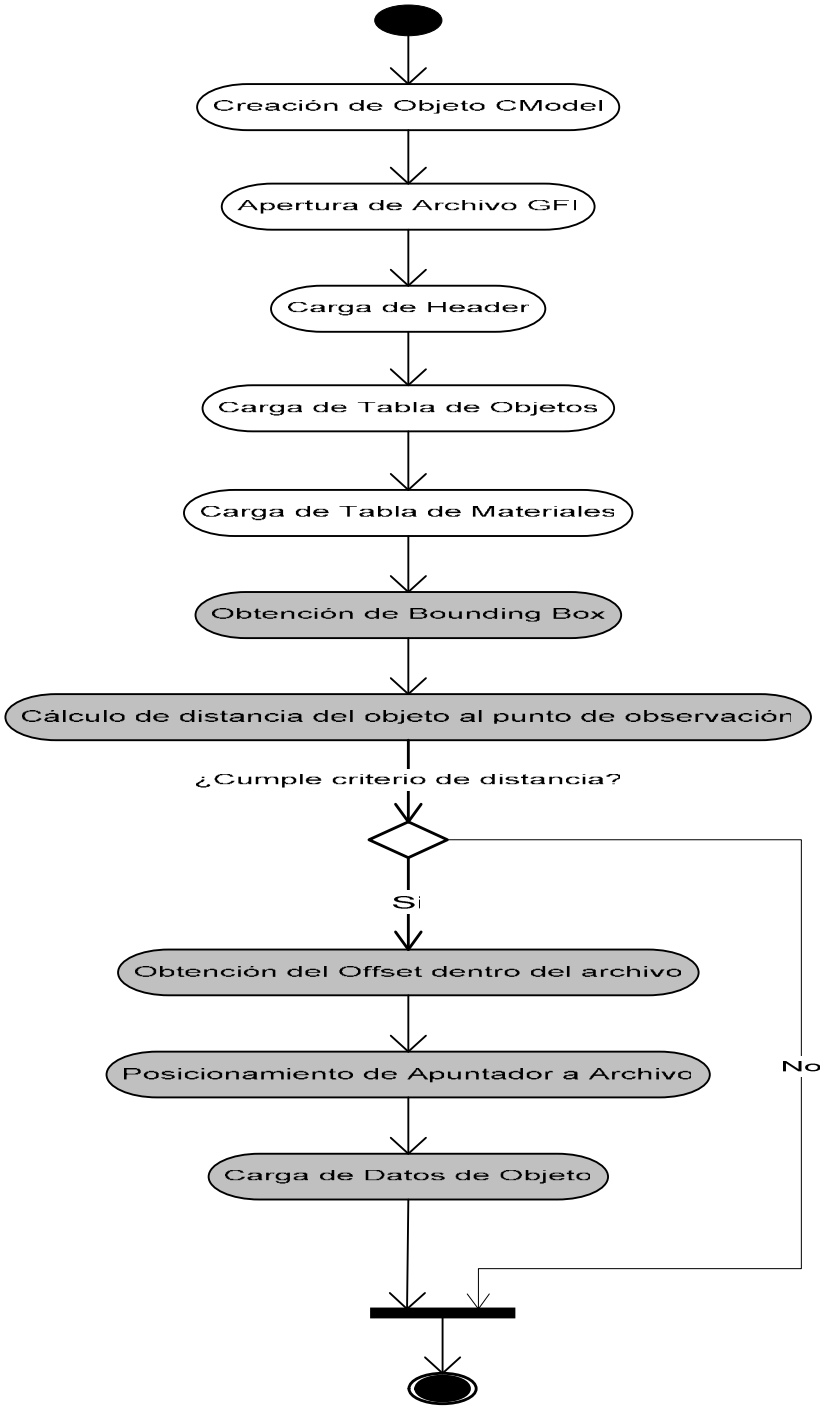


Fig. 4.4.3.3.a. Diagrama UML para carga de objeto por distancia

Una vez cargado el objeto dicho proceso se realiza en un ciclo que está determinado por los elementos de la Tabla de Objetos, es decir, se recorre toda la tabla para obtener la información del bounding box que corresponde a cada objeto y realizar el cálculo de la distancia, de esta manera determinar si se debe cargar o no la información del objeto.

El siguiente código muestra como obtener el valor del bounding box para obtener la distancia del observador al objeto y usarla para saber si el objeto debe ser cargado

```
#include "CModel.h"

CModel modelo;                          //se crea un objeto CModel
BOX bound;                              //variable para el criterio de distancia

if(¡modelo.GFIStartSession("c:\modelo.gfi")) //intentamos cargar las tablas
                                             //indice del modelo
       exit(0);

//se itera a través de la tabla índice de objetos

modelo.ObjectTable.current= modelo.ObjectTable.ListBegin();
while(modelo.ObjectTable.current!=NULL)
{
       //se obtiene el Bounding Box del objeto para sabersus posicion
       bound= modelo.ObjectTable.current->GetBox();

       if(cumple con criterio de distancia)
              modelo.GFILoadObject(modelo.ObjectTable.current);
       else
       {
              //si el bjeto esta ya fuera de rango, se borra de la memoria
              if(modelo.ObjectTable.current->IsLoaded())
                     modelo.ObjectTable.Erase(modelo.ObjectTable.current);
       }

       modelo.ObjectTable.current= modelo.ObjectTable.current->NextField();
}
```

4.5. Carga de archivos en formato GFI de manera remota.

4.5.1. Protocolo de comunicación para archivos GFI.

Las capacidades de carga por partes del archivo GFI antes vistas permiten un uso mas eficiente de los canales de comunicación entre el medio de almacenamiento del archivo y las estructuras en memoria principal del sistema que va dibujar el elemento, estas capacidades lo hacen ideal para su transmisión a través de un canal de comunicación con mucho mas trafico de información, como lo es Internet.

La manera en que se realiza la carga remota no es muy diferente a la manera en que se cargaban los archivos localmente, ya que solo se sustituye el proceso de obtención desde el disco duro local, por un proceso de obtención de la información requerida a través de un puerto de comunicación con algún protocolo de transmisión por red, el protocolo seleccionado fue el protocolo TCP/IP debido a su amplia confiabilidad en el envió de paquetes por una red

El protocolo GFI de comunicación entre el servidor y el cliente es la manera en la cual las funciones de carga de archivos obtendrán los paquetes de datos en donde se encuentra la información que define la geometría del modelo, de manera análoga a como existe el protocolo FTP[17] para transferencia de archivos o el protocolo TELNET para de comunicación entre una terminal y su centro de proceso.

El primero paso del protocolo GFI consiste en establecer la conexión cliente con el servidor, para ello debe haber un programa servidor escuchando peticiones de conexión por un puerto del sistema, de esta manera el cliente puede hacer una petición al servidor, que el caso de tener recursos para atender a ese cliente, crea un nuevo par de sockets, uno para recibir comandos del cliente (conexión de control) y otro para atender las peticiones de archivos (conexión de datos), de manera que se deja libre el puerto original en donde el servidor sigue sondeando el canal por las conexiones que otros clientes quieran realizar.

Esta conexión puede realizarse usando la interfaz de programación de sockets implementada por Berkeley Sofware Distribution (BSD release 4.3) o cualquier otro API que cumpla con esta especificación, por ejemplo Winsock.

La conexión de control manda el primer mensaje con el numero de cliente que el servidor le asigno, el cliente al recibir este identificador da por hecho una conexión correcta con la cual esta listo para mandar comandos de petición de archivos GFI del servidor.

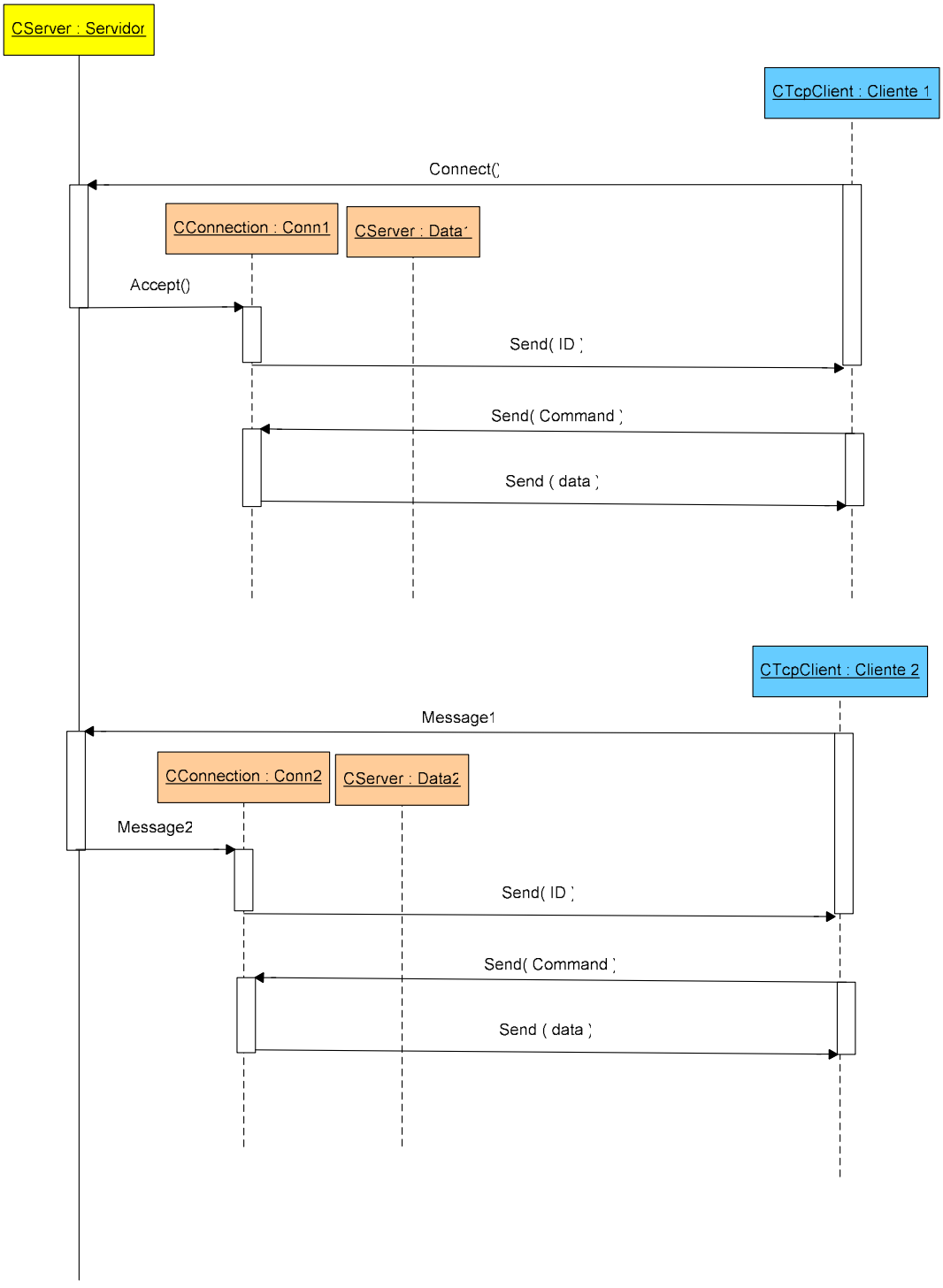


Fig. 4.5.1.a. Diagrama UML de conexión de multiples clientes con el servidor. Cada línea vertical representa un puerto activo en el cliente o en el servidor, y las líneas horizontales representan el envío de información.

Los comandos con los que trabaja este protocolo son cadenas de caracteres de 4 bytes de largo que el cliente manda a través de la conexión de control, comunicándole al servidor qué tipo de bloque de datos enviar al cliente. Los más importantes son:

“load”

Este comando seguido del nombre del archivo le ordena al servidor que transmita una copia de ese archivo al cliente

“ldtb”

Este comando seguido del nombre del archivo, le especifica al servidor que el cliente quiere iniciar una sesión con el archivo GFI especificado, el servidor buscará este archivo y mandará la información correspondiente a las tablas índice del modelo

“objt”

Este comando buscará en el archivo de sesión GFI más reciente, los datos acerca del objeto especificado por su offset y tamaño en el archivo, este offset se obtiene de la tabla obtenida al iniciar sesión con “ldtb”.

“fcls”

Este comando cerrará la sesión con el archivo GFI en el servidor.

“quit”

Este comando termina la conexión del cliente con el servidor.

Los comandos que solicitan datos lo hacen de dos maneras, por archivo o solo parte de él, cuando se solicita el archivo completo (load) el servidor manda el tamaño de éste al cliente, de manera que éste sabe exactamente la cantidad de datos que el servidor le enviará. Cuando solo se solicita cierta parte del archivo (ldtb,objt) generalmente este bloque de datos contiene información acerca de una parte del modelo, el cliente le especifica al servidor el inicio de esos datos así como su longitud, esta información el cliente ya la conoce por que es parte de la información que el servidor le envió cuando se inició sesión con dicho archivo, cuando el servidor recibe el offset y la longitud, envía el bloque de datos del archivo limitados por esa información.

Una vez que el servidor sabe que tipo de bloque de datos desea el cliente, crea una conexión de datos, por la cual se mandarán paquetes de tamaño igual o menor a 1024 bytes hasta que la transmisión del bloque requerido se haya completado, el objetivo de crear esta conexión de datos es dejar libre la conexión de control para que el cliente pueda seguir realizando peticiones al servidor inclusive otras conexiones de datos mientras las primeras terminan.

La Figura 4.5.1.b. muestra el diagrama de cómo el cliente y el servidor interactúan para la trasmisión de un archivo una vez que la conexión entre ellos ha sido llevada a cabo.

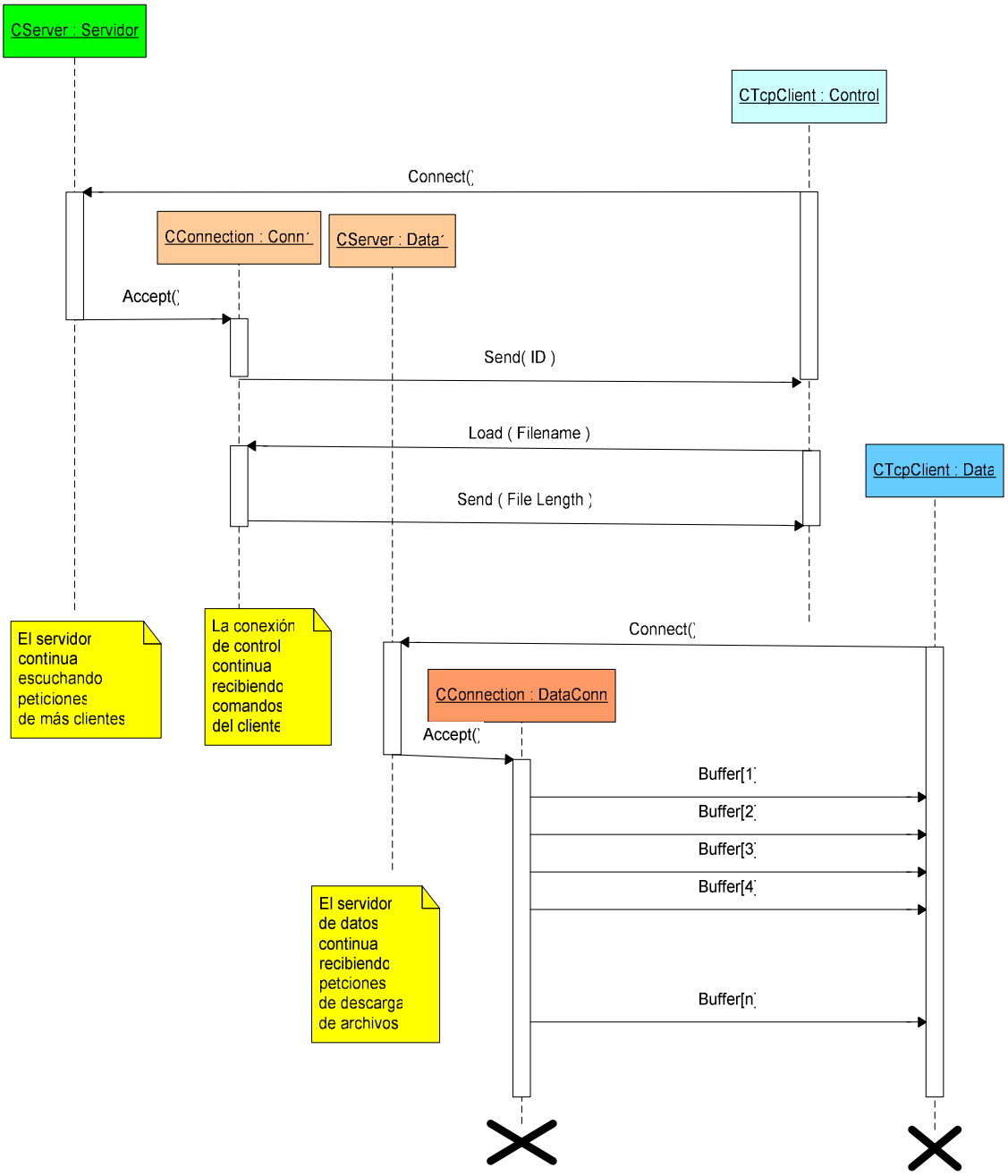


Fig. 4.5.1.b. Diagrama UML de una solicitud de conexión de datos por el cliente.

Debido a la naturaleza insegura del canal de comunicación de una red mundial como Internet, se hace obligatorio que el protocolo GFI cuente con algún

método para verificar la integridad de la información recibida, por esta razón, cada método de envío de bloques de información, es sometido por el servidor a la generación de una huella digital MD5, este método de huella digital electrónica fue seleccionado por que permite identificar si el bloque de información tiene datos corruptos, ya sea por falla en el canal de comunicación o por algún intento premeditado de alterar la información que el servidor de datos envía por algún puerto, esta clave de 16 bytes es enviada cada vez que se trasmite completo el bloque de datos solicitado, el cliente puede elegir revisar la clave recibida con una que el mismo cliente genera a partir de los datos recibidos, si ambas claves son iguales, el bloque de datos llegó en buen estado, de otro modo, el archivo sufrió alguna modificación mientras fue trasmitido.

El cálculo para la obtención de la clave MD5 se hizo siguiendo el algoritmo establecido en el documento "*RFC 1321 MD5 Message-Digest Algorithm, April 1992, by MIT Laboratory for Computer Science and RSA Data Security, Inc.*".

4.5.2. Implementación en el algoritmo de carga.

El funcionamiento de transmisión del archivo o parte de él, consiste básicamente en que el cliente solicite un bloque de datos y después lo reciba en un buffer en memoria, de manera análoga a como funciona la carga de un archivo de manera local, esto con el fin de que los diagramas de bloques entre los dos medios sean lo más parecido posible, es decir, el comportamiento de cada una de las funciones explicadas en el capitulo 4.4 seguirá la misma secuencia lógica excepto en los bloques en donde la obtención del archivo sea de manera remota.

En la figura 4.5.2.a y 4.52.b se muestra que los algoritmos de carga del archivo GFI completo y por partes, no sufren modificación, solo se cambian ciertos bloques funcionales, de esta manera el modo de obtención de los datos ya no es local sino desde servidor de archivos.

En el se aprecia como el bloque de la apertura y carga a memoria RAM se sustituye por un proceso de petición y descarga de datos de el servidor, así como el proceso de posicionamiento del apuntador de archivo en la carga por partes local se sustituye por una solicitud al servidor de una transferencia de datos a partir de ese offset.

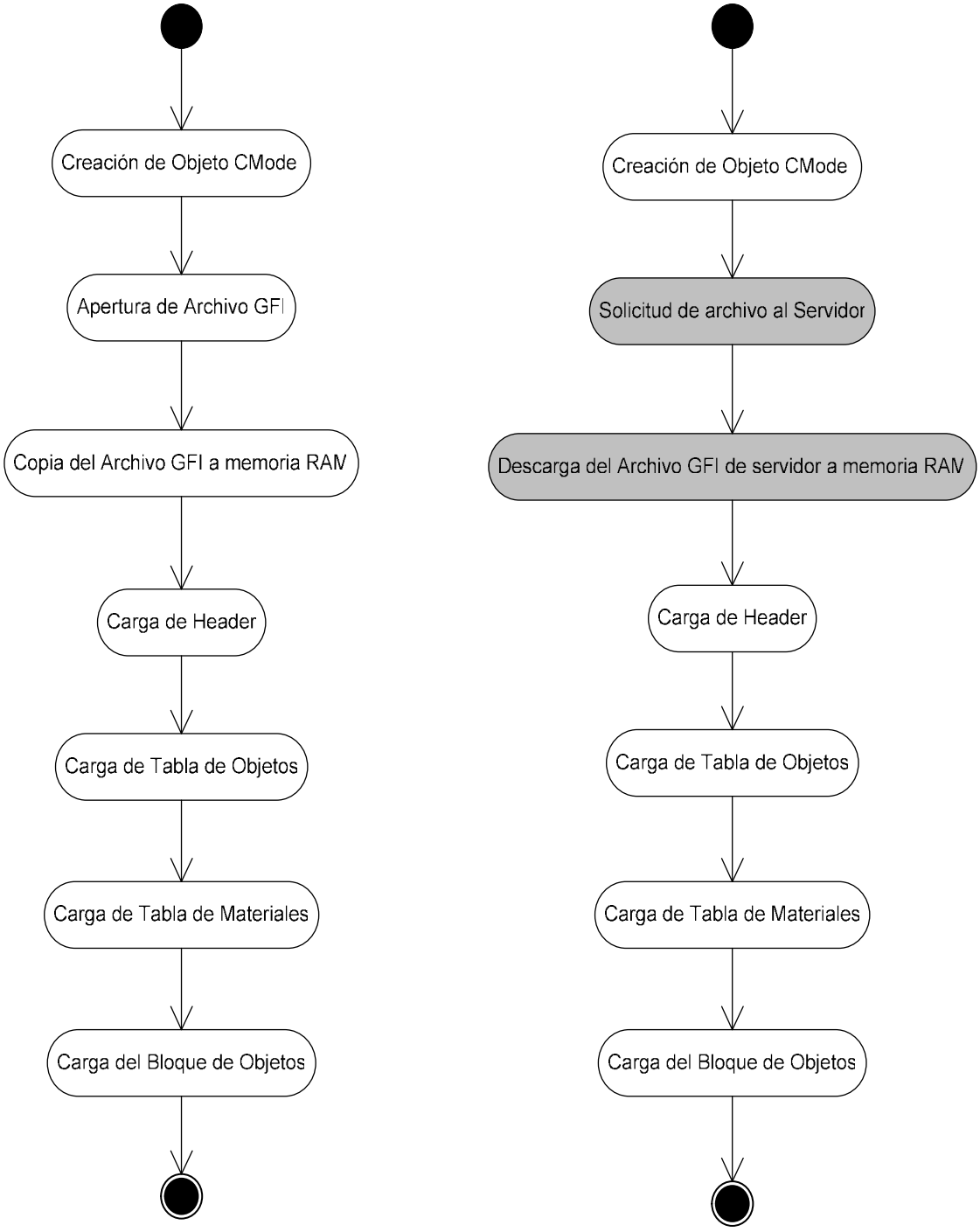


Fig. 4.5.2.a. Comparación entre diagramas UML de carga local y remota respectivamente.

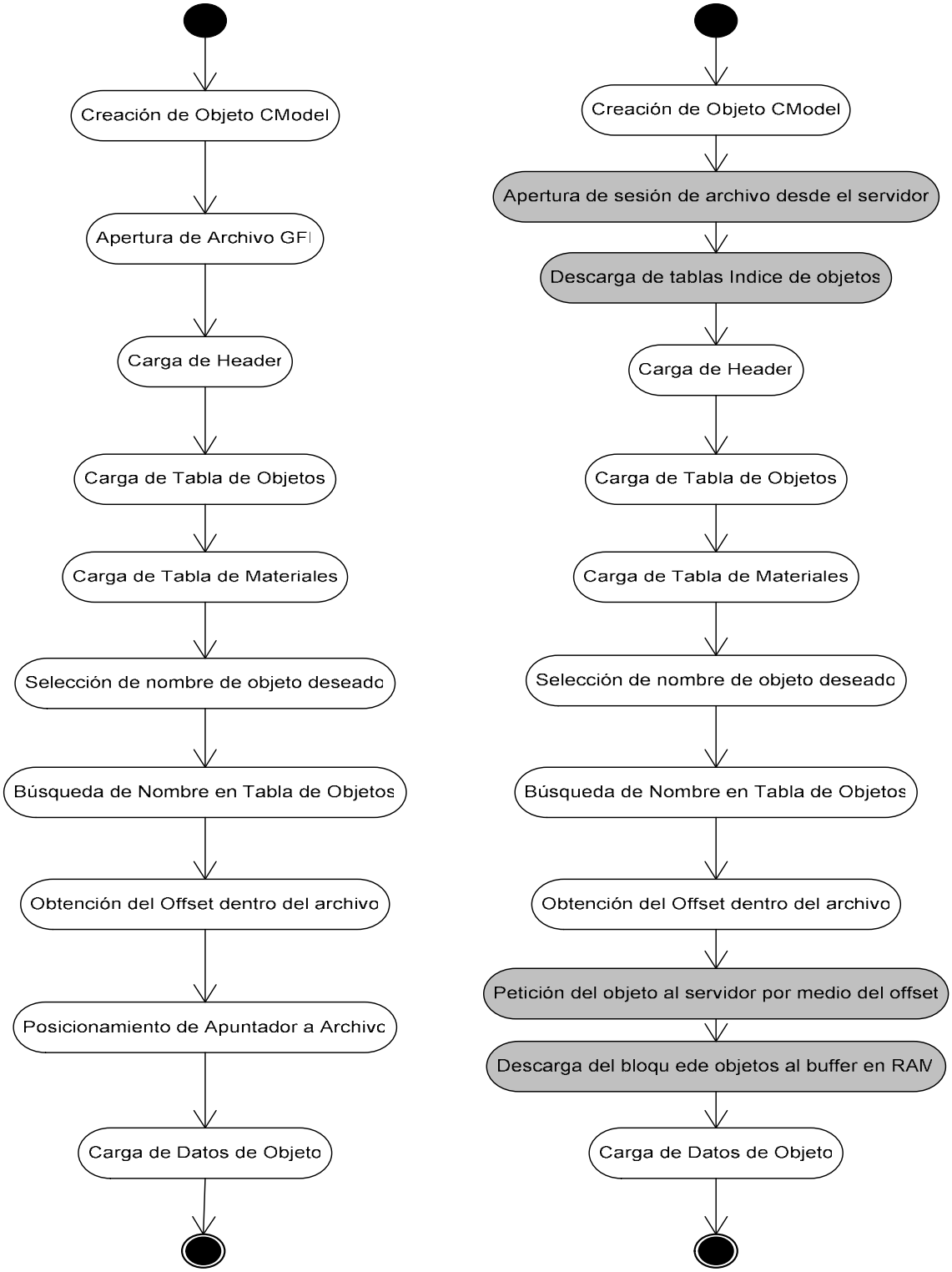


Fig. 4.5.2.b. Comparación entre Diagramas UML de carga local y remota de los objetos del modelo

El siguiente código muestra como se carga un modelo GFI desde un servidor GFI a través de Internet.

```
#include “CModel.h”
#include “CClient.h”

CModel modelo;                  //se crea un objeto para guaradar los datos del modelo
CTcpClient cliente;             //se crea un objeto cliente para conectarse al servidor

If(!cliente.ConnectTo(3490,”132.248.52.156”))   //se hace un intento de coneccion con protocolo
                                                //TCP
        exit(0);


//se intenta descargar el modelo desde el servidor especificado
if(model. GFILoadTCP(“modelo.gfi”, &cliente)== MODEL_LOADED)
{
        //las texturas tambien se debe de obtener desde el servidor
        modelo. LoadTextureImagesTCP(&cliente);
        modelo. VertexNormals();
        modelo. GLIniTextures();
}
else
{
        cliente.Disconnect();   //si el servidor no tiene el modelo el cliente se desconecta
        exit(0);
}
```

4.6. Creación de archivos GFI.

Para la creación de archivos en formato GFI, se tomó como base al formato 3DS, esto por ser uno de los formatos más utilizados y con mayor soporte entre aplicaciones para la creación de gráficas en computadora.

Para comenzar el proceso de guardado del archivo en formato GFI se requiere de la carga a memoria de un archivo en formato 3DS o GFI, el archivo puede obtenerse de manera local o remota. (Fig. 4.6.a).

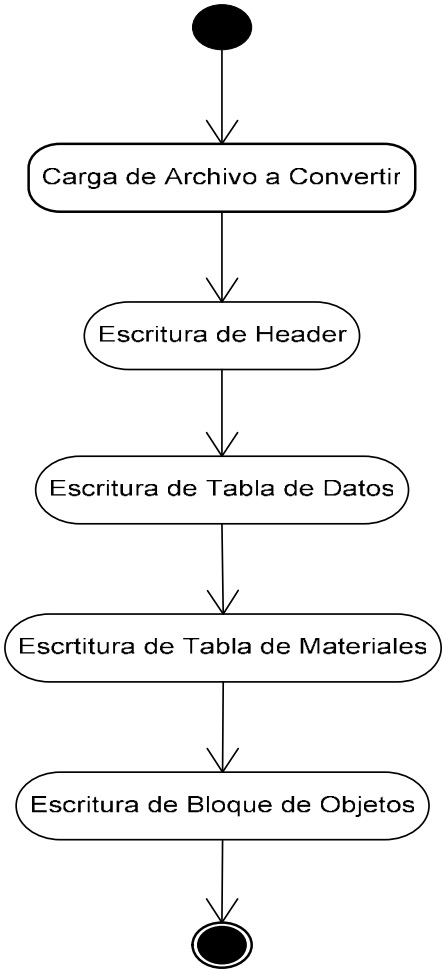


Fig. 4.6.a . Diagrama UML del proceso de Creación de un Archivo GFI.

Una vez que el archivo, en cualquiera de los dos formatos anteriores, ha sido cargado puede procederse a la creación del archivo en formato GFI. Para guardar la información que se encuentra dentro de las estructuras de datos se sigue la estructura que se muestra en el layout con sus correspondientes tipos de datos.

La escritura de los archivo en formato GFI es prácticamente de manera secuencial. Dado que para obtener la información referente a los offsets que indiquen el comienzo de los bloques de los objetos dentro del archivo y que puedan ser leídos en la tabla de objetos, se realiza guardando de manera temporal la posición del archivo al momento de que se está escribiendo el inicio de cada bloque correspondiente a la información del objeto y una vez terminado, mover el apuntador de archivo a la posición dentro de la tabla de objetos y colocar el offset correspondiente a cada objeto dentro de su espacio asignado en la tabla. (Fig. 4.6.b.)

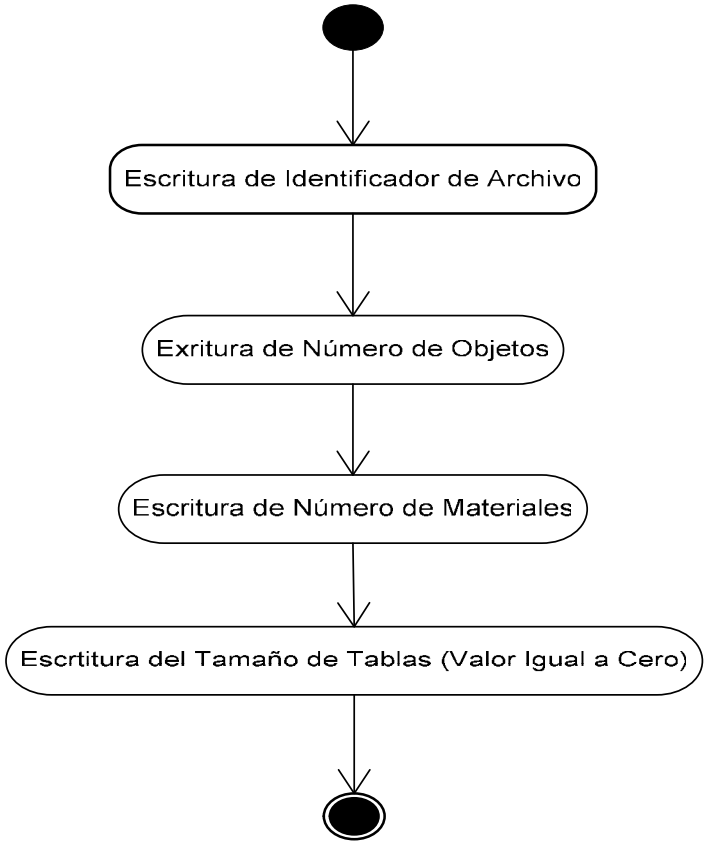


Fig. 4.6.b. Diagrama UML de escritura de Header.

La primera parte del archivo GFI se le conoce como Header, en el que se escribe primero un número que permite verificar si el archivo corresponde a este formato. Dentro del mismo Header, enseguida se escribe el número de objetos que forman al modelo que será almacenado en el archivo, este valor es de una longitud de 4 bytes con lo cual se tiene restringido a un número máximo de 4,294,967,296 objetos el cual es un número lo bastante apropiado para formar un modelo, este mismo número es la limitante para los a archivos 3DS.

Se escribe otro dato, de longitud 4 bytes, el cual es el número de materiales asociados al modelo, al igual que en el número de objetos se tiene un límite de 4,294,967,296. Otro elemento que se escribe dentro del Header es la longitud de la Tabla de Objetos y de la Tabla de Materiales, este dato es conocido hasta que se ha escrito la información de dichas tablas, por lo que hasta el momento es desconocido, y en su lugar se escribe un valor de cero (0) en 4 bytes, con la finalidad de apartar este espacio y más adelante regresar para la escritura de este dato. Esta longitud es necesaria conocerla para así indicar a las rutinas de carga del archivo a través de una red mediante el uso del protocolo TCP, el tamaño de dichos elementos para su encapsulación en paquetes, para su correcto envío, ya

que además estos elementos son los primeros datos que son necesarios enviar para la posterior presentación de los objetos del modelo que forman la escena.

Enseguida de la escritura del bloque de información Header, sigue la escritura de lo que se conoce como Tabla de Objetos, la cual es una forma de indexar los objetos para su rápida localización dentro del archivo tanto para su carga de manera local como por la carga en red. (Fig. 4.6.c.)

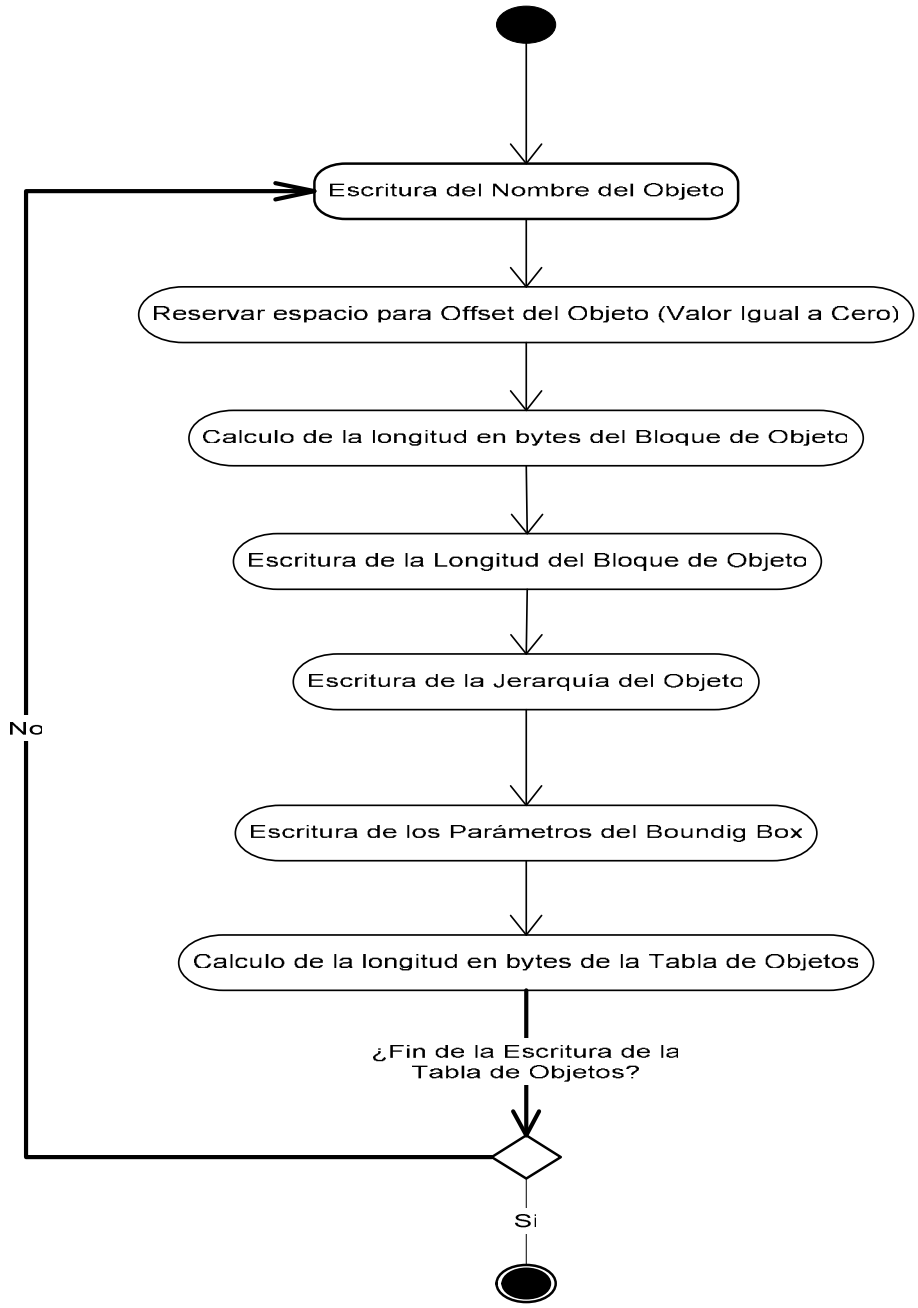


Fig. 4.6.c. Diagrama UML de escritura de Tabla de Objetos.

El primer dato que se escribe dentro de la Tabla de Objetos es el nombre del objeto, al terminar de escribir la cadena que representa a dicho dato, se escribe un byte con un valor de cero (0), esto para delimitar el fin de la cadena y separarla de otros datos. Enseguida se escribe el offset del inicio de la información del objeto, sin embargo en este momento de la escritura del archivo este valor es desconocido, por lo cual en su lugar se escribe un valor de cero (0) representado en 4 bytes, para posteriormente regresar a su escritura cuando el valor ya sea conocido.

Para la escritura del siguiente dato que forma parte de la Tabla de Objetos, se requiere hacer el cálculo de los bytes que ocupa la información del objeto, este cálculo se realiza con los valores de Número de Vértices, Número de Caras, Número de Coordenadas de Texturización y con los bytes que ocupa la información de los materiales asignados. Todos los anteriores datos ya son conocidos por encontrarse en las estructuras. Con el valor obtenido se escribe el dato tamaño del objeto, el cual ayuda a conocer el tamaño del paquete a enviar al proporcionar el tamaño de todo el bloque de información del objeto cuando se realiza la carga del modelo a través de una red.

En esta Tabla de Objetos se escribe también la jerarquía para el algoritmo de carga, la jerarquía es un valor de 2 bytes.

La Tabla de Objetos cuenta con dos puntos que ayudan a la obtención del centroide y además determinan el bounding box para cada objeto. Estos valores son los valores máximos y mínimos de los vértices que forman al objeto, y se obtienen al momento de cargar un archivo en formato 3DS, en el caso de la carga de un archivo en formato GFI, estos valores ya están incluidos por lo que no hay que volver a calcularlos. Se escriben estos valores, el máximo y el mínimo, representados por sus respectivas componentes en los ejes X, Y y Z.

Se escriben estos datos para cada uno de los objetos con los que está formado el modelo, y al grupo de todos estos datos se les conoce como la Tabla de Objetos.

Al siguiente grupo de datos a escribir se le llama Tabla de Materiales, en la cual se escriben los datos de los materiales empleados por cada uno de los objetos (Fig. 4.6.d). Escribiendo primero el nombre que identifica al material, este nombre consiste en una cadena con el nombre del material a la cual se le agrega un bytes con valor cero (0) para delimitar el fin del dato. También se escriben las componentes de color que definen dicho material, las cuales están almacenadas mediante sus parámetros R, G y B, cada uno de ellos de tamaño 4 bytes. Además se escribe un parámetro de transparencia para el material. El último dato a escribir dentro de los datos del material es el nombre de la textura asociada al mismo y este dato puede ser un valor de 0 (cero) para indicar que no existe textura asociada, en caso contrario, se escribe el nombre de la textura delimitada por un cero (0) al final de la cadena.

Una vez concluida la escritura de la Tabla de Materiales ya es posible conocer el valor de la longitud de las tablas, valor que es necesario escribir dentro del Header del archivo dentro de su espacio correspondiente.

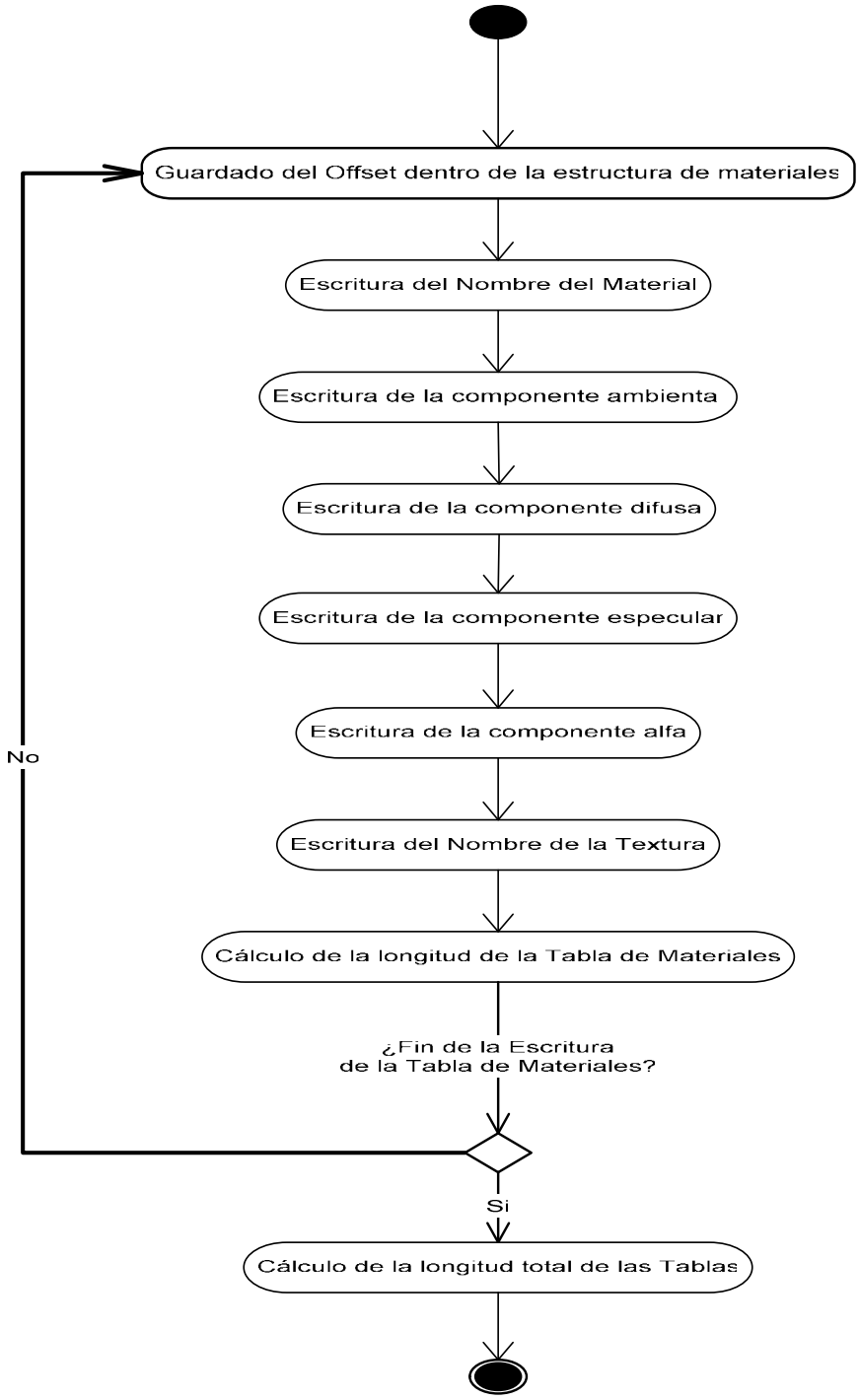


Fig. 4.6.d. Diagrama UML de escritura de Tabla de Materiales.

El Bloque de Datos de los Objetos se encuentra estructurado de tal manera que contenga todos los datos referentes a la información básica para dibujar una escena, como lo son vértices y reglas de unión (Fig. 4.6.e). El primer dato que se escribe es el número de vértices que forman al objeto y que está restringido a 65,536 vértices al ser el campo de longitud 2 bytes. Al anterior dato le sigue la escritura del número de reglas de unión para formar las caras del objeto que se limita a 65,536 reglas de unión por objeto siendo este valor de longitud 2 bytes. Se escribe el número de materiales asignados al objeto en un campo de longitud 2 bytes este número se ocupará en un ciclo para la escritura de los datos de los materiales que corresponden a cada objeto. También se escribe el número de coordenadas de texturización cuya longitud es de 2 bytes y es ocupado para la realización de un ciclo en el cual se escriben los datos de las coordenadas de texturización.

Para cada objeto se escribe un bloque con las coordenadas de los vértices que lo forman, las coordenadas están dadas por sus componentes X, Y y Z que son números flotantes, la escritura de estos valores se realiza dentro de un ciclo, el cual está determinado por el valor que se escribió dentro del campo de número de vértices del objeto.

Se cuenta con un bloque en el cual se escriben las reglas de unión de los vértices para formar las caras del objeto, los datos corresponden a los identificadores de cada uno de los tres vértices que forman la cara. Un bloque con las coordenadas de texturización, en caso de existir, su escritura se llevará a cabo igualmente dentro de un ciclo determinado ahora por el valor escrito en el campo número de coordenadas de texturización, dichas coordenadas están dadas por la escritura de sus componentes S y T con un tamaño de 4 bytes cada una.

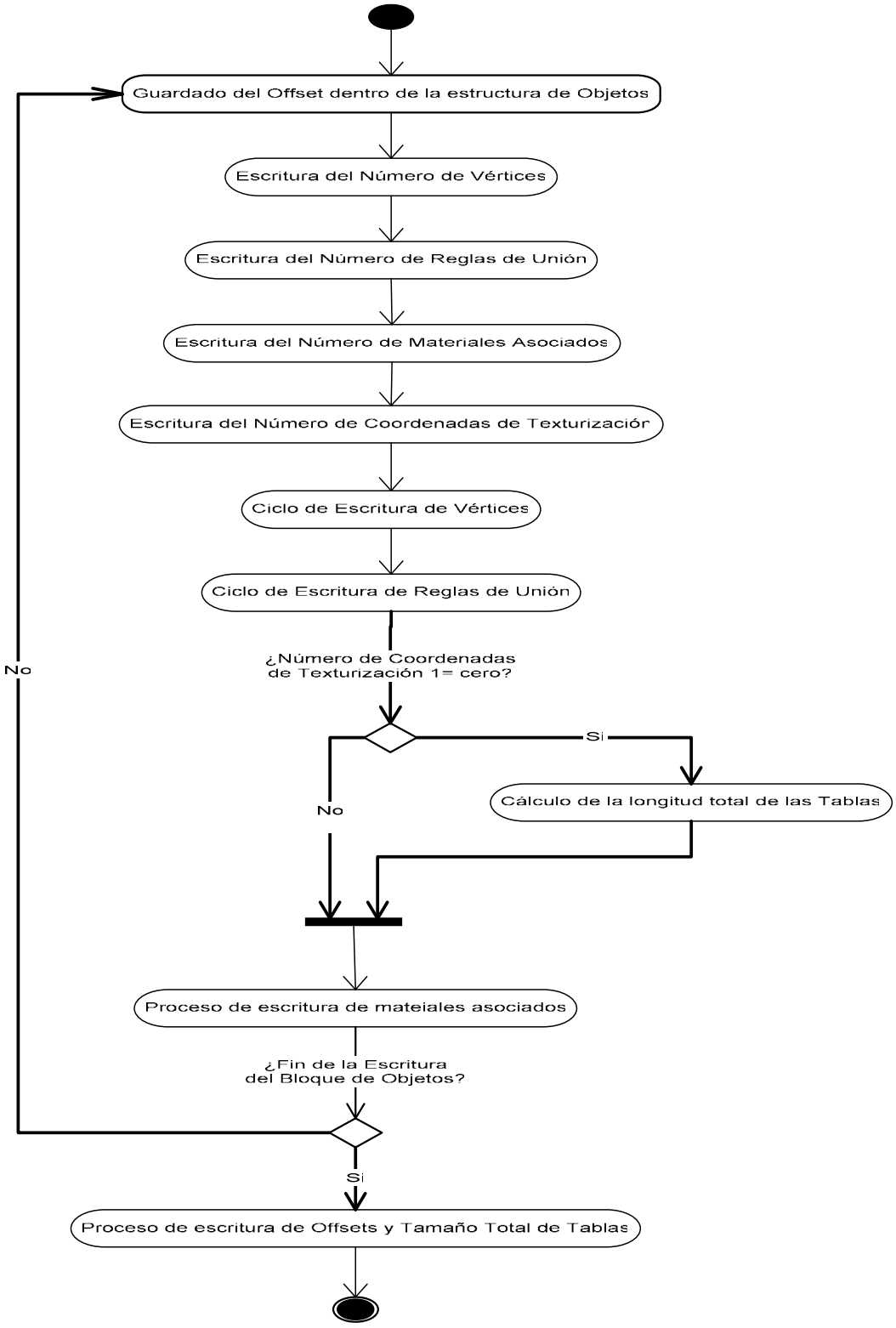


Fig. 4.6.e. Diagrama UML de escritura del Bloque de Objetos.

También se escribe la referencia a los materiales asociados al objeto, número de caras asociadas al material y los identificadores de caras asociadas al material (Fig. 4.6.f). La referencia a los materiales asociados al objeto es el offset o ubicación del inicio de la información del material dentro del archivo con un tamaño de 4 bytes. Se escribe enseguida un campo, de tamaño 2 bytes, con el número de caras asignadas al material con el cual se realizará a continuación un ciclo de escritura de los identificadores de las caras asociadas al material, dichos identificadores son de 2 bytes.

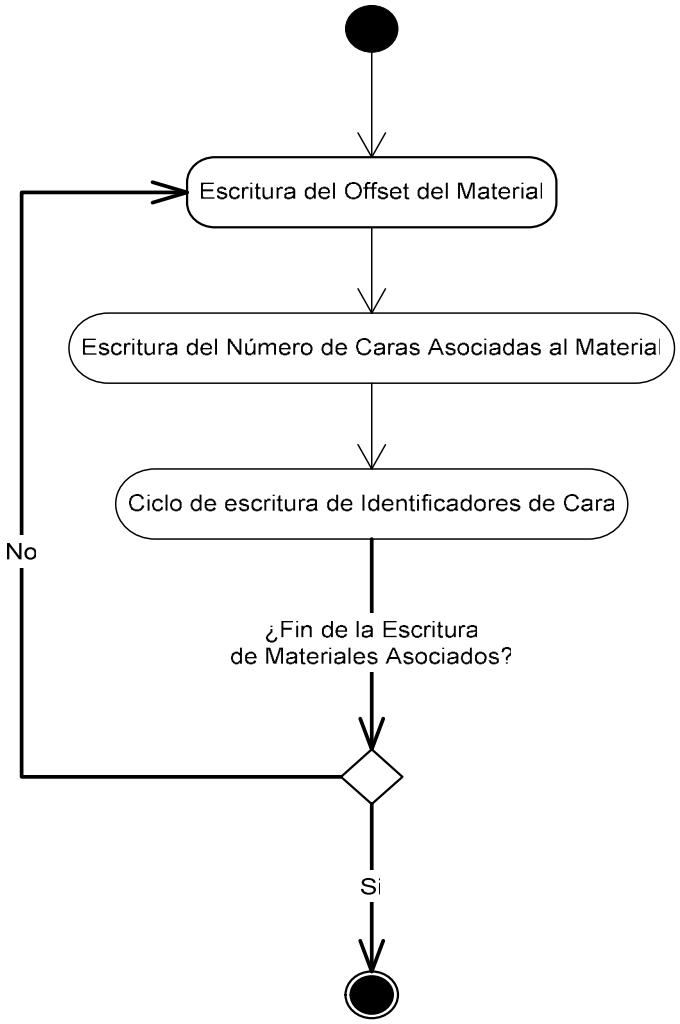


Fig. 4.6.f. Diagrama UML de escritura de Materiales Asociados al objeto.

Este proceso se realiza para la escritura de la información correspondiente a cada uno de los objetos, y cada vez se debe llevar la cuenta de los bytes escritos dentro de cada bloque de objeto, con esto es posible escribir el valor que se tenía reservado en la tabla de objetos para indicar el tamaño del bloque de objeto.

4.7. Ciclo de dibujo de archivos GFI.

4.7.1. Diagrama de flujo.

La manera en que se recomienda recorrer las estructuras para mandar los datos al visualizador (Renderer) es como sigue:

El ciclo de dibujo (Fig. 4.7.1.a) se empieza tomando al primer elemento de la lista de Objetos, una vez seleccionado dicho elemento se procede a la lectura de la lista que contienen los materiales asignados.

Se recomienda dibujar el objeto en el orden que indica la estructura de materiales asignados a él, ya que de esta forma las caras están agrupadas por material, de esta manera se logra que los cambios en los parámetros del material y texturización sean los menos posibles, incrementando el desempeño del proceso de dibujo.

Con el identificador de la cara actual a dibujar, se pueden acceder a los diferentes datos referentes a la cara que pueden ser usados para que el modelo se dibuje con mayor detalle.

Uno de los datos que a los que se puede acceder mediante el identificador de la cara son los valores de los vértices que la componen. Los vértices de cada cara son accesibles para su dibujo, así como para el cálculo del vector de visibilidad de la cara, este vector de visibilidad se utiliza en el algoritmo de Face Culling.

Otro dato que puede ser obtenido a partir del identificador de la cara son las normales a la cara. Los vectores normales a las caras se utilizan para la correcta iluminación del objeto, y debido a su característica de perpendicularidad a la superficie de la cara también se puede ocupar para el algoritmo de Face Culling.

Otros datos de las caras son sus coordenadas de texturización, accesibles también en este nivel. Este dato permite al visualizador colocar la textura actual sobre la cara que le corresponde.

Este proceso se sigue para cada uno de los objetos contenidos dentro de la lista de Objetos.

Para la generación de las texturas necesarias se realiza el recorrido de la lista de imágenes. Desde la lista de imágenes se pueden verificar los parámetros requeridos para que Render pueda crear las texturas. Entre los parámetros que se verifican se encuentran: la profundidad del color, el tamaño de la imagen, y cómo se tomarán las coordenadas de texturización.

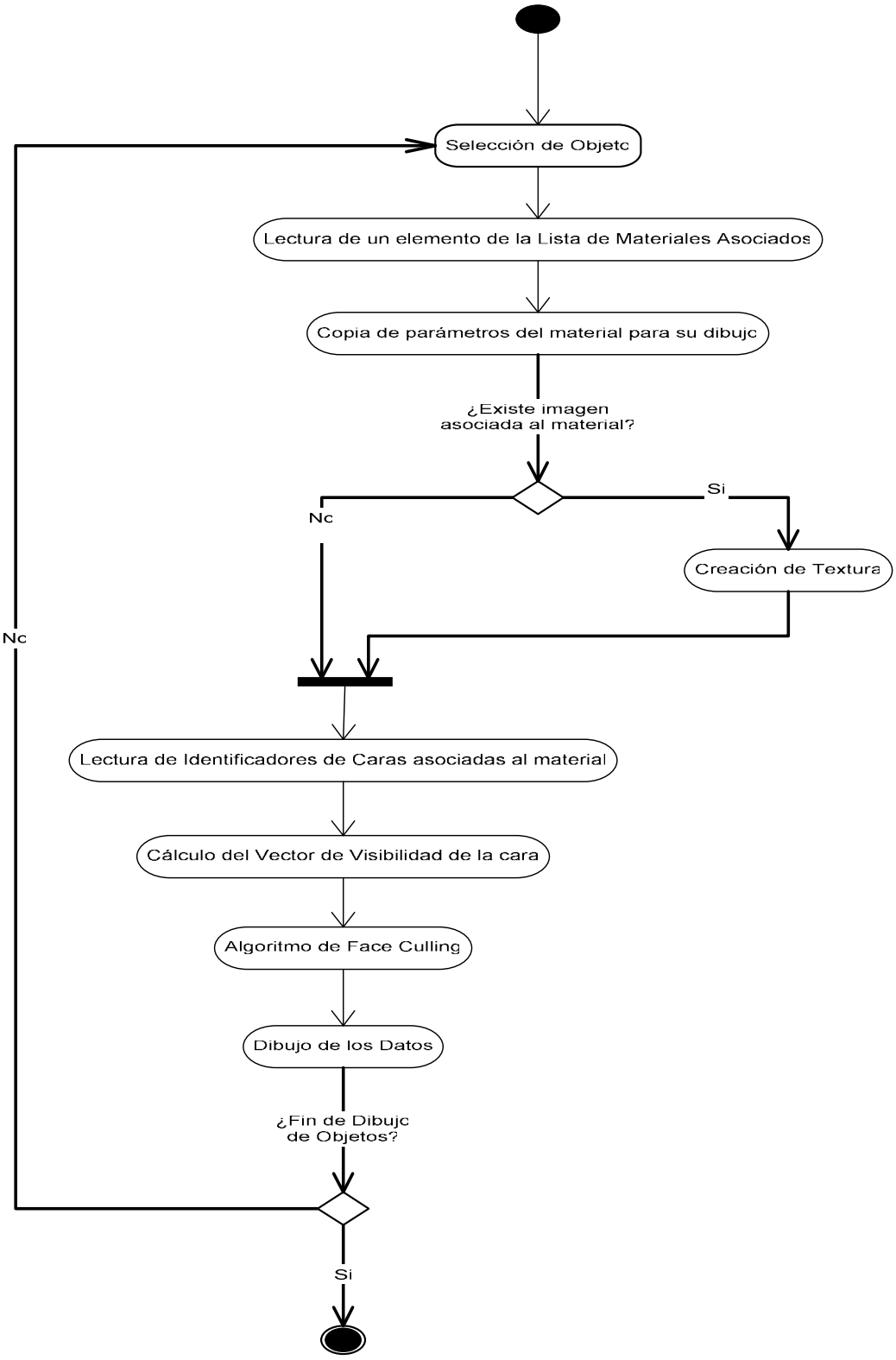


Fig. 4.7.1.a. Diagrama UML del ciclo de dibujo.

4.7.2. Algoritmos para mejor desempeño y visualización.

Uno de los algoritmos que más ayuda para el desempeño al momento de mostrar el dibujo es el Face Culling.

Enseguida se procederá a explicar los diferentes vectores que se ocuparon para el manejo de la cámara y que llevaron a la obtención del vector de visión necesario para la implementación del algoritmo Face Culling. (Fig. 4.7.2.a)

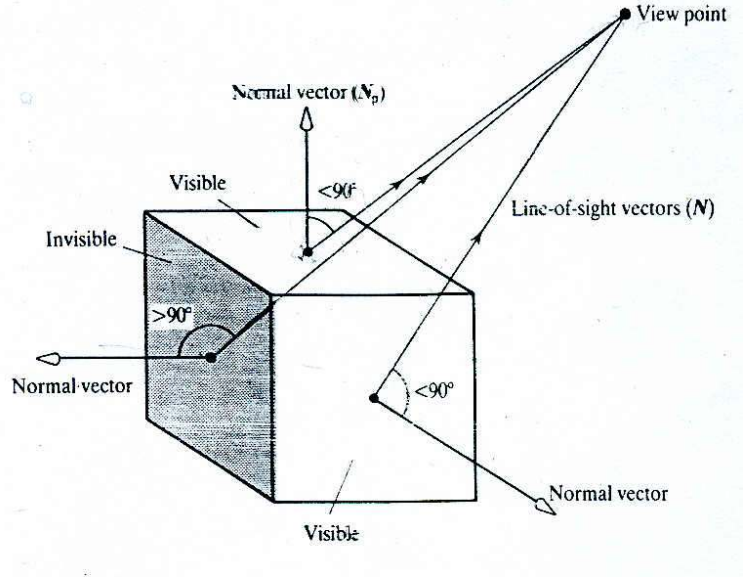


Fig. 4.7.2.a. Vectores necesarios para la aplicación del algoritmo Face Culling. [1]

Se tiene un vector que corresponde al vector de posición de la cámara, el cual se modifica cada vez que el usuario hace uso de las teclas para el movimiento de la cámara, a dicho vector se le denominó **poscam**, y tiene valores para cada una de sus correspondientes componentes. (Fig. 4.7.2.b)

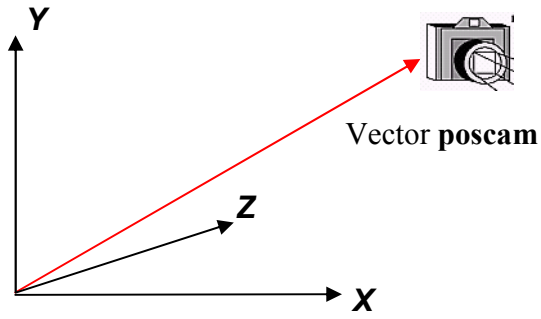


Fig 4.7.2.b. Vector de Posición para la Cámara.

Sin embargo otro movimiento que tiene la cámara es el punto al que apunta, es decir, al punto en el cual centrará su punto de vista. A este vector se le denominó **view**. (Fig. 4.7.2.c)

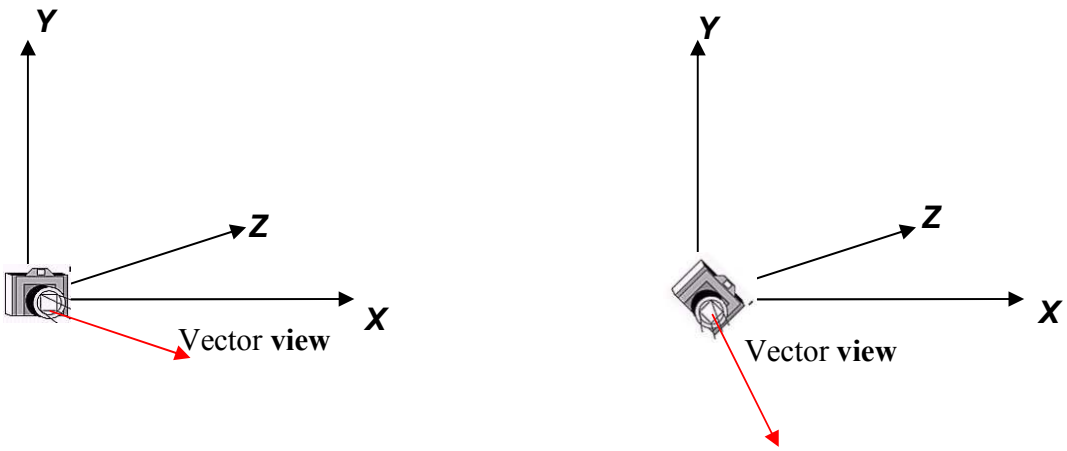


Fig 4.7.2.c. Diferentes Vectores **View** para una misma Posición de la Cámara.

Como puede observarse en las figuras anteriores, la posición de la cámara no varía, sin embargo lo que varía es el vector que indica hacia donde está orientada la cámara, y esto se indica por medio del vector **view**.

Ya con los dos vectores anteriormente descritos se puede tener una mejor indicación hacia donde se encuentra apuntando nuestra cámara, con lo cual se puede obtener el vector que indica la posición actual a la que apunta nuestra cámara. (Fig. 4.7.2.d)

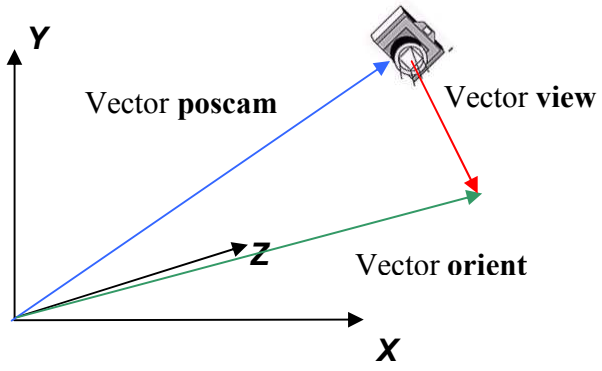


Fig. 4.7.2.d. Obtención del vector **orient**

CAPÍTULO 5.

Pruebas de desempeño y resultados.

5.1. Tabla comparativa del formato GFI con los analizados.

Formato	Wavefront Object	3D Studio	Maya	LightWave 3D	Quake 2	VRML	**Formato Gráfico de la Facultad de Ingeniería**
Extensión	OBJ (ASCII) y MOD (Binario)	3DS	MA y MB	LWO	MD2	WRL	**GFI**
Límite de vértices	ilimitado	65,536 por objeto	No documen-tado	65,536 por objeto	2,048	2^{32}	**65,536 por objeto**
Límite de objetos	Ilimitado	No documen-tado	No documen-tado	No documen-tado	1	2^{32}	**2^{32}**
Texturas	Si	Si	Si	Si	Si	Si	**Si**
Animación	No	Si	Si	No	Si	Si	**No**
Control de errores	No	No	No	No	No	No	**Si**
Documen-tación	ASCII Libre* Binario Controlada por Alias/Wave front	Controlada por Autodesk	Controlada por Alias/Wave front	Libre*	Libre*	Libre	**Libre**
Aplicacio-nes que la soportan	Advanced Visualizer	3D Studio, Calgary TrueSpace y varios otros programas de modelado en 3D	Maya, y otros programas de modelado en 3D	LightWave 3D y LightWave Modeler.	Videojuego Quake 2	Navegado-res web con plugins	**BlackMage Loader**
Envío por partes a través de una red	No	No	No	No	No	No	**Si**
Jerarquía de los objetos	No	Si	Si	No	No	Si	**Si**
Tipo de lectura	Secuencial	Chunks	Secuencial	Chunks	Indexada	Secuencial	**Indexada.**

* Requiere suscripción gratuita en página del desarrollador.

5.2. Diagramas de tiempo de carga.

Los diagramas de tiempo que se muestran en este apartado corresponden a los datos de tiempos de carga y el tamaño en bytes de los modelos usados para la realización de las pruebas.

Los modelos utilizados para realizar las pruebas son los siguientes:

a) Hidroeléctrica. Este modelo fue proporcionado por Ingenieros de la carrera de Ingeniería Civil participantes dentro del proyecto para la Sala Ixtli de la visualización de una planta hidroeléctrica, el caso de estudio fue la hidroeléctrica “El Cajón”. Este archivo cuenta con texturas y una gran cantidad de objetos forman al modelo.

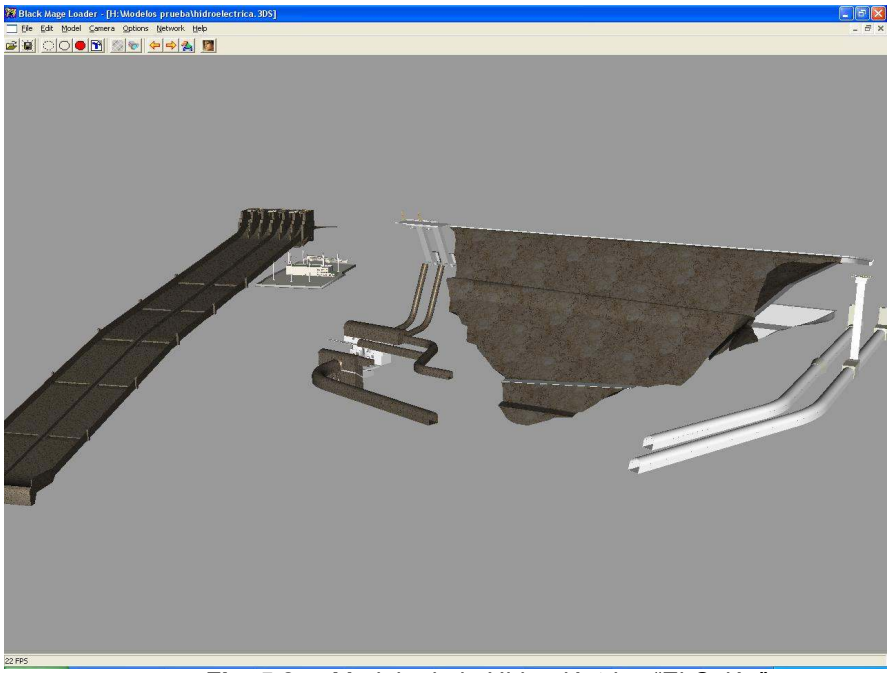


Fig. 5.2.a. Modelo de la Hidroeléctrica “El Cajón”.

b) F1. Este modelo fue seleccionado debido a la gran cantidad de texturas que emplea, casi todos los polígonos se encuentran cubiertos por una textura.

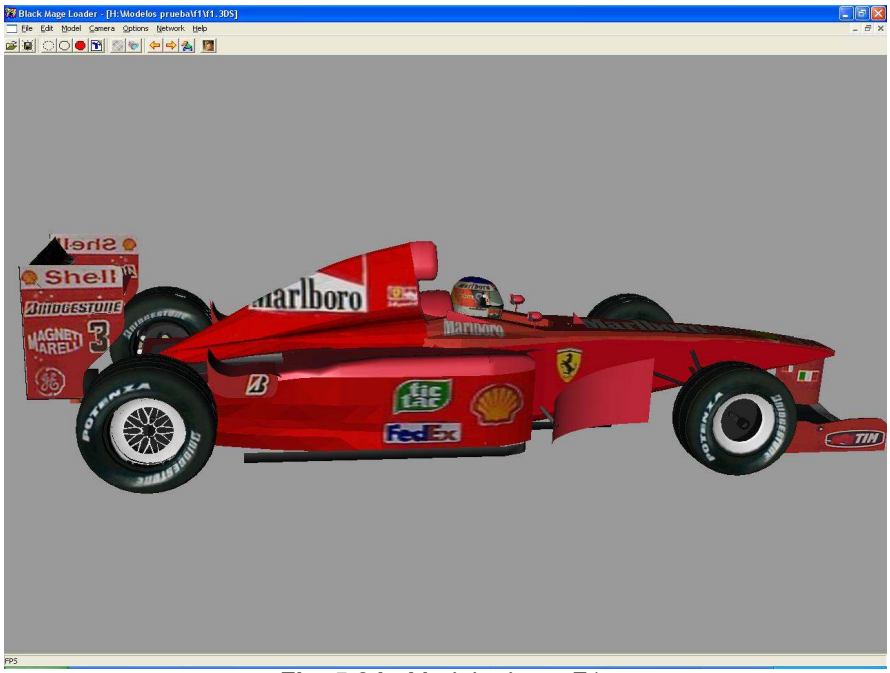


Fig. 5.2.b. Modelo de un F1.

c) Azteca. Este modelo cuenta con una gran cantidad de objetos, así como de materiales, debido a su construcción, se escogió para mostrar el nivel de detalle del modelo.

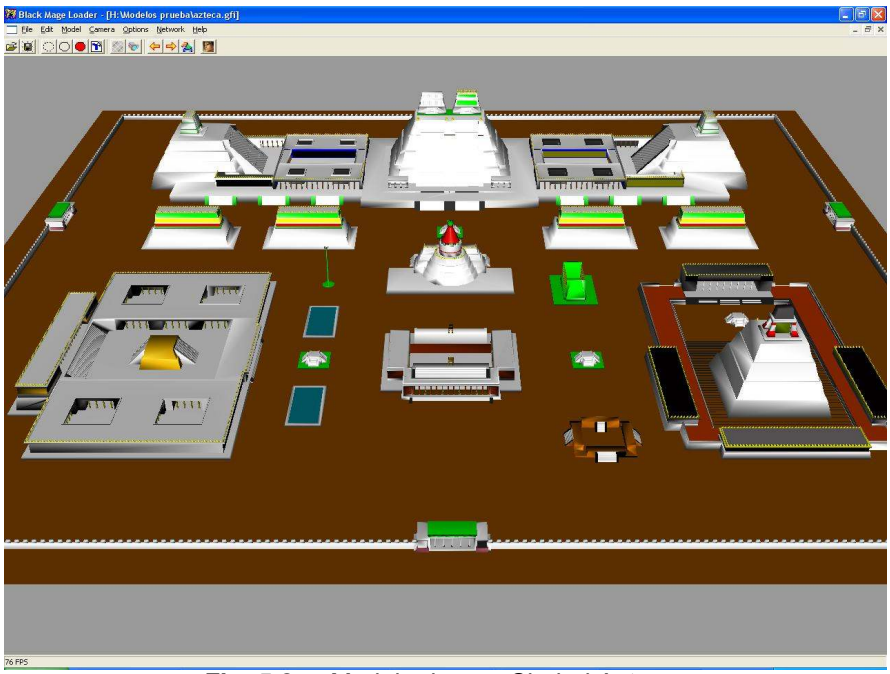


Fig. 5.2.c. Modelo de una Ciudad Azteca.

d) 747. Debido a la simpleza del modelo, y su reducido tamaño, este archivo se escogió para ver el comportamiento del API con archivos pequeños.

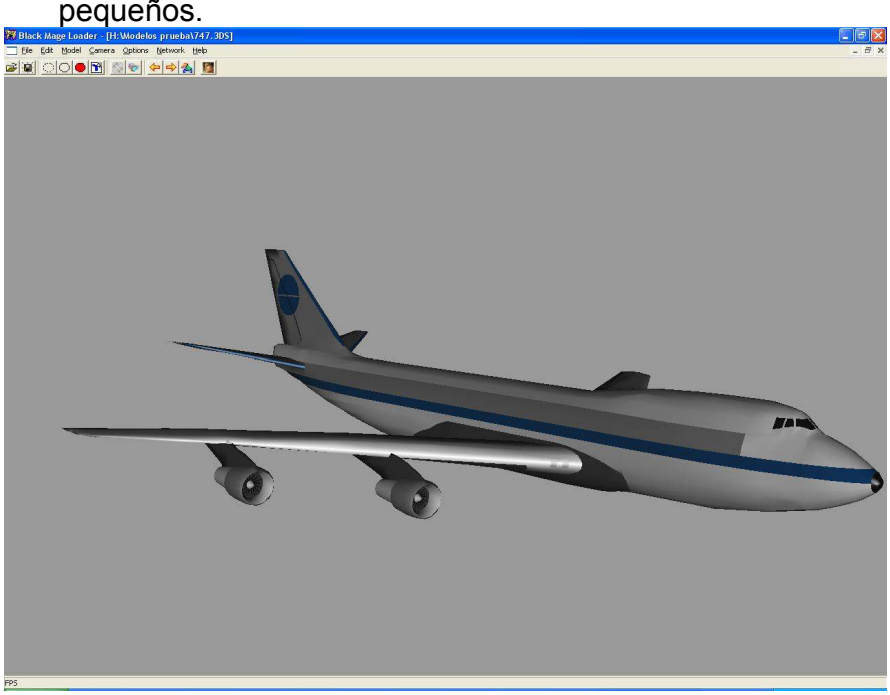


Fig. 5.2.d. Modelo de un Boing 747.

e) Gargoyle. Este archivo también es de tamaño pequeño, sin embargo su construcción es más compleja, por lo cual se escogió para comparar el comportamiento del API bajo estas circunstancias.

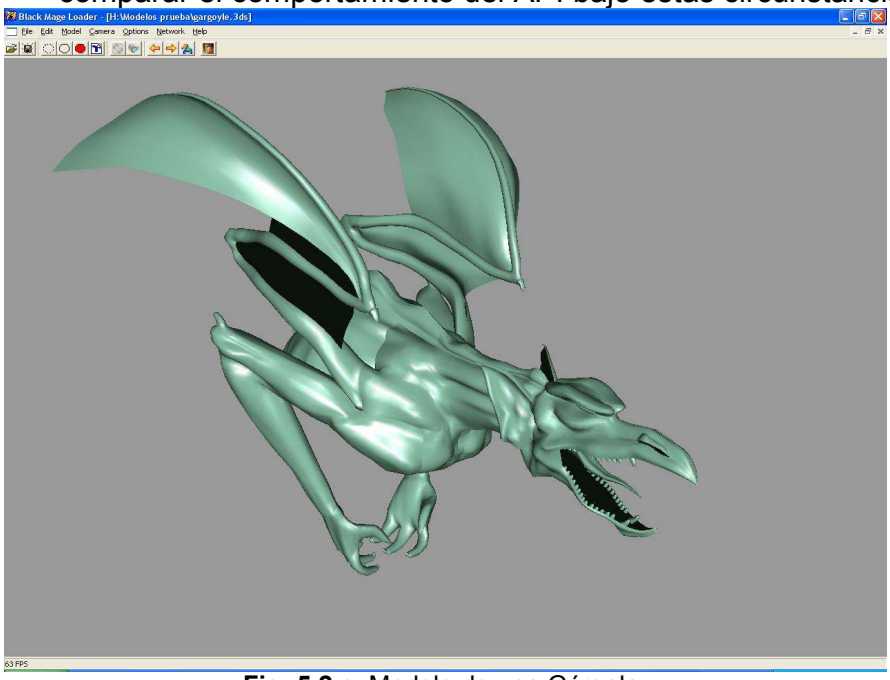


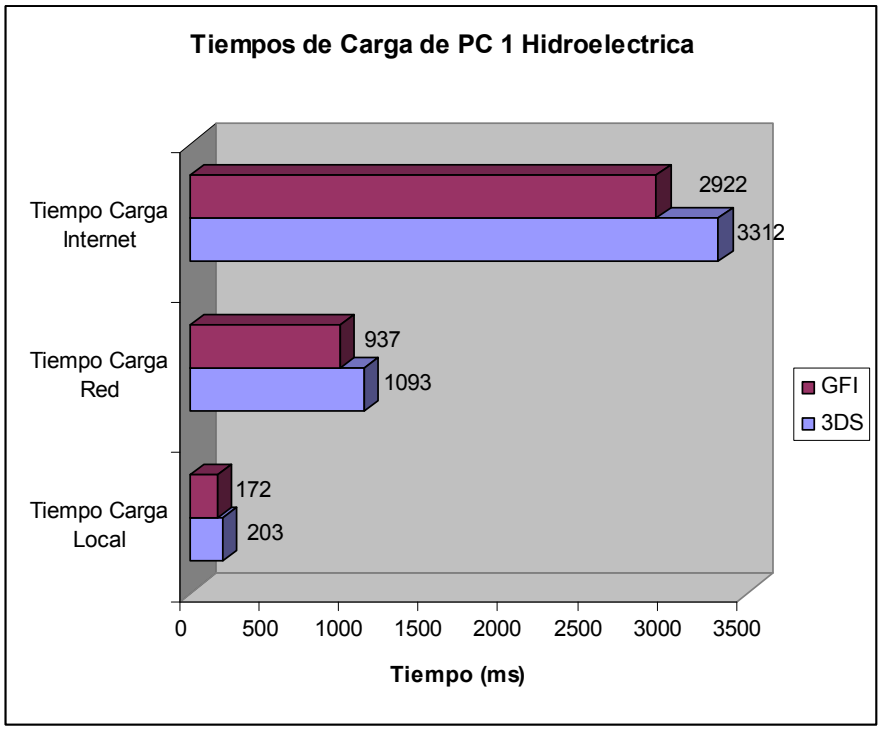
Fig. 5.2.e. Modelo de una Gárgola.

La tabla (Tabla 5.2.a) siguiente son los datos obtenidos como resultado de las pruebas realizadas en una computadora con las siguientes características: Pentium 4, 2.8 GHz, 512 MB RAM, Windows XP Profesional Versión 2002 Service Pack 2, tarjeta de video Radeon X300 SE 256 MB.

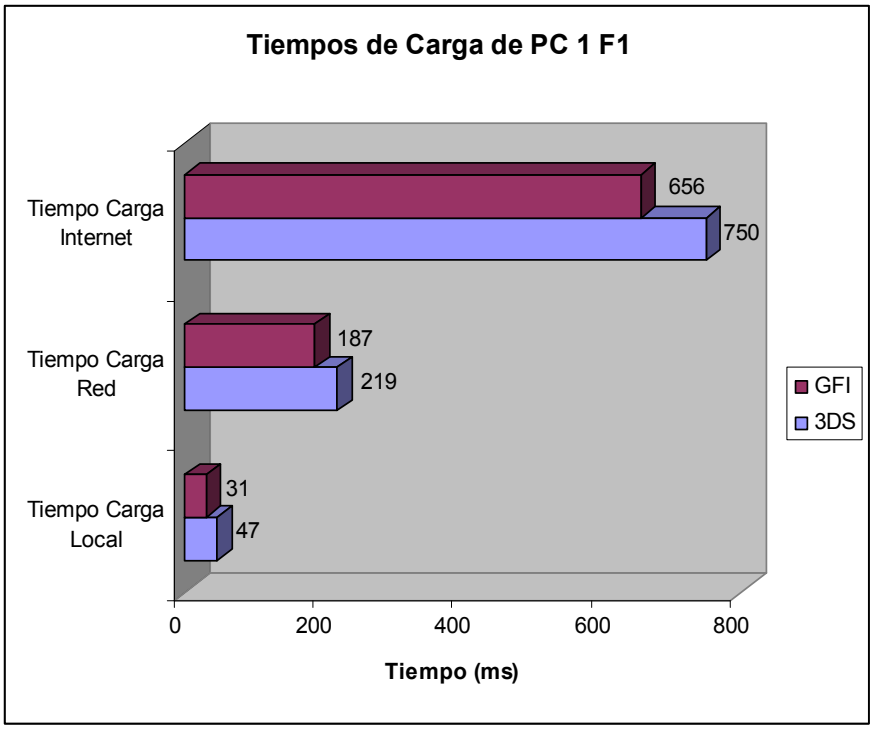
Los tiempos mostrados, son el tiempo promedio de carga obtenidos después de haber realizado quince mediciones por cada modelo.

MODELO		Archivo 3DS	Archivo GFI	Diferencia %
Hidroeléctrica	Tamaño	8,637,038 bytes	7,624,956 bytes	11.7 menor GFI
	Tiempo carga local	203 ms	172 ms	15.2 menor GFI
	Tiempo carga red	1093 ms	937 ms	14.2 menor GFI
	Tiempo carga Internet	3312 ms	2922 ms	11.7 menor GFI
F1	Tamaño	1,785,069 bytes	1,388,923 bytes	22.1 menor GFI
	Tiempo carga local	47 ms	31 ms	34 menor GFI
	Tiempo carga red	219 ms	187 ms	14.6 menor GFI
	Tiempo carga Internet	750 ms	656 ms	12.5 menor GFI
Azteca	Tamaño	2,661,892 bytes	1,383,534 bytes	48 menor GFI
	Tiempo carga local	63 ms	47 ms	25.3 menor GFI
	Tiempo carga red	250 ms	187 ms	25.2 menor GFI
	Tiempo carga Internet	891 ms	656 ms	26.3 menor GFI
747	Tamaño	147,239 bytes	98,892 bytes	32.8 menor GFI
	Tiempo carga local	15 ms	15 ms	0
	Tiempo carga red	16 ms	15 ms	6.2 menor GFI
	Tiempo carga Internet	172 ms	157 ms	8.7 menor GFI
Gargoyle	Tamaño	821,972 bytes	579,453 bytes	29.5 menor GFI
	Tiempo carga local	16 ms	15 ms	6.2 menor GFI
	Tiempo carga red	109 ms	78 ms	28.4 menor GFI
	Tiempo carga Internet	422 ms	328 ms	22.2 menor GFI

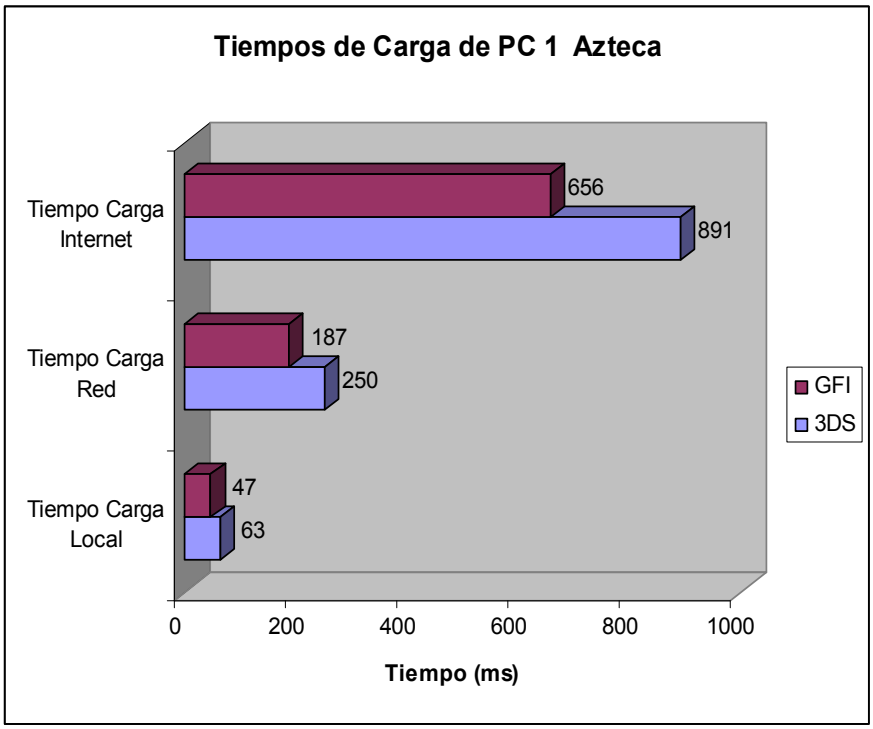
Tabla 5.2.a. Resultados obtenidos



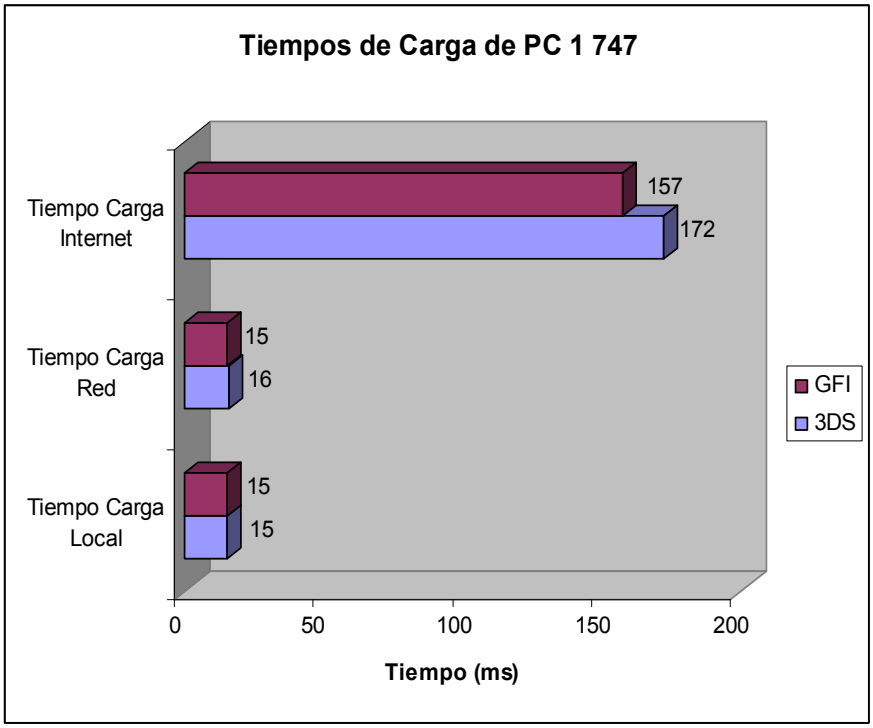
Gráfica 5.2.a. Diagrama de Tiempos de Carga para el modelo Hidroeléctrica.



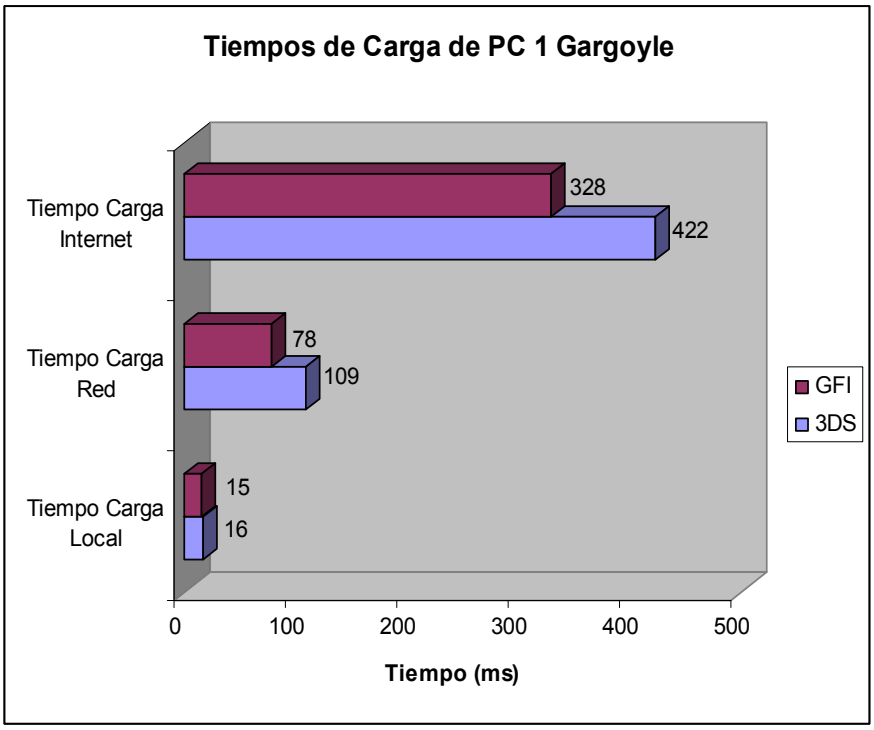
Gráfica 5.2.b. Diagrama de Tiempos de Carga para el modelo F1.



Gráfica 5.2.c. Diagrama de Tiempos de Carga para el modelo Azteca.



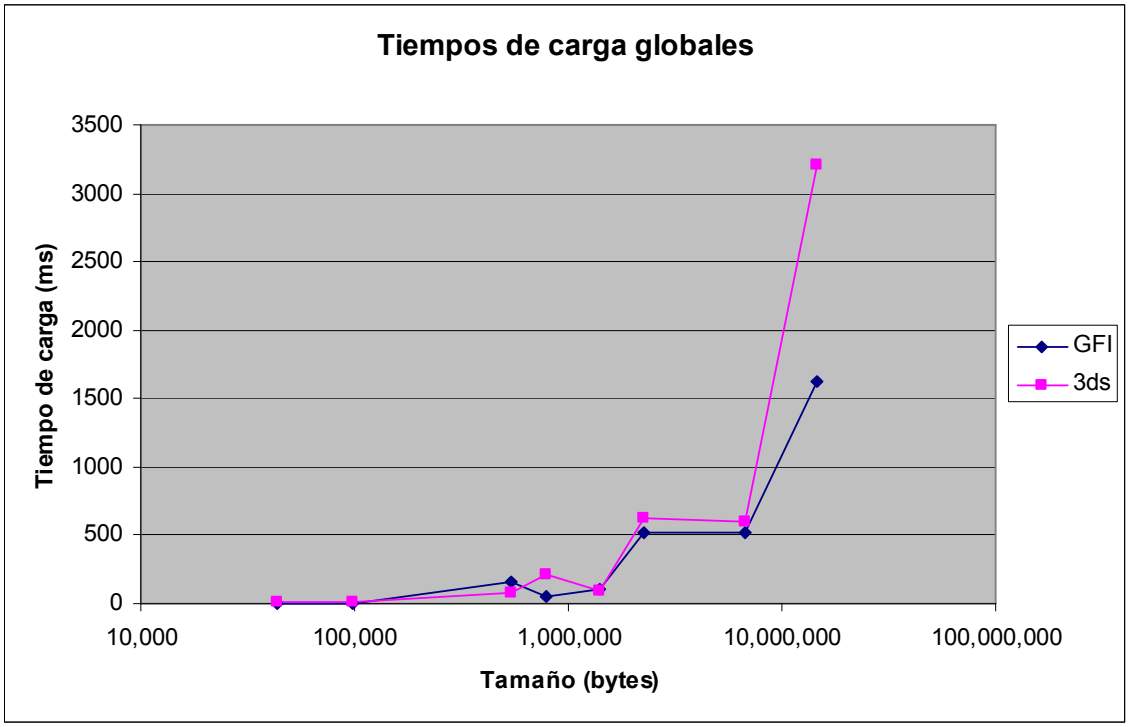
Gráfica 5.2.d. Diagrama de Tiempos de Carga para el modelo 747.



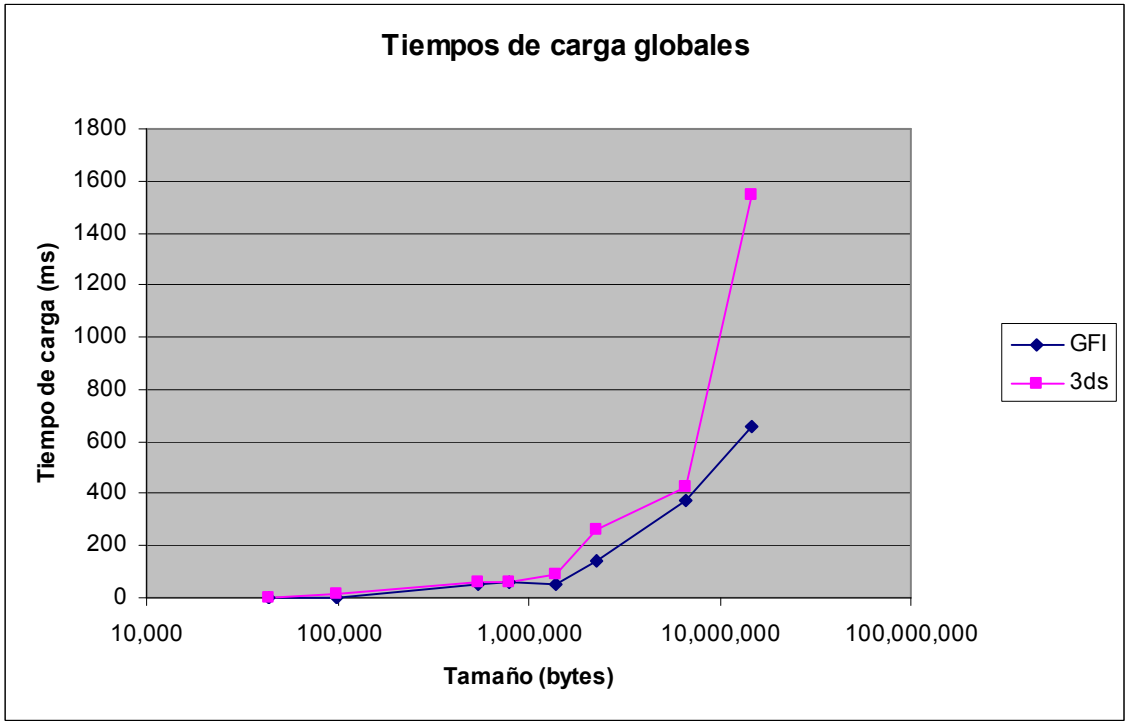
Gráfica 5.2.e. Diagrama de Tiempos de Carga para el modelo Gargoyle.

Esta parte de las pruebas muestra una grafica comparativa entre los tiempos de carga de archivo GFI y 3DS, esta serie de pruebas se hacen con una gama más completa de modelos que van desde unos pocos Kilobytes hasta modelos con densidad de informacion muy grande de hasta miles de Kilobytes.

Debido a este amplio intervalo de valores, las gráficas usan escala logarítmica, de esta manera se aprecia de mejor manera cómo el formato GFI tiende a reducir considerablemente el tiempo de carga conforme el tamaño del archivo se incrementa.



Gráfica 5.2.f. Tiempos de carga de diferente archivo GFI y 3DS en un sistema con un disco duro IDE de 7200 RPM, Procesador Athlon XP a 1.333 GHz, 768 MB en RAM.



Gráfica 5.2.f. Tiempos de carga de diferentes archivo GFI y 3DS en un sistema con disco duro Serial ATA 10,000 RPM, Procesador Athlon 64 a 2.21 GHz y 1 GB de memoria RAM.

Desempeño de Visualización para la Carga del Modelo por Partes.

La Figura 5.2.f. muestra la visualización del modelo de una Ciudad Azteca, cuando se ha cargado completamente, mientras que la Figura 5.2.g. muestra el mismo modelo pero cuando se ha realizado la carga de solo ciertas partes del modelo. Esto se hizo para demostrar las capacidades de ahorro de memoria y mejora en el desempeño de la visualización mediante la utilización de algoritmos especializados para la carga de objetos del modelo más importantes. En este caso, el algoritmo utilizado fue el de la carga de los objetos más cercanos al observador, esto es posible ya que el API de programación desarrollada en este trabajo de tesis permite este tipo de manejo de carga de objetos tridimensionales.

En la prueba de Uso de Memoria (Grafica 5.2.g), la Memoria Total se refiere a la cantidad de memoria ocupada por el modelo que se cargó más la cantidad de memoria ocupada por la aplicación de visualización. Mientras que el valor de Memoria del Modelo, corresponde únicamente al valor de uso de memoria empleado para guardar la información del modelo que se visualiza.

La Gráfica 5.2.h muestra los frames por segundo (FPS) que la aplicación es capaz de dibujar al momento de cargar el modelo de prueba en su totalidad y la carga por partes, en ella se muestra claramente un mayor rendimiento en la carga por partes que la carga completa.

Fig. 5.2.f. Visualización de un modelo cargado de manera completa.

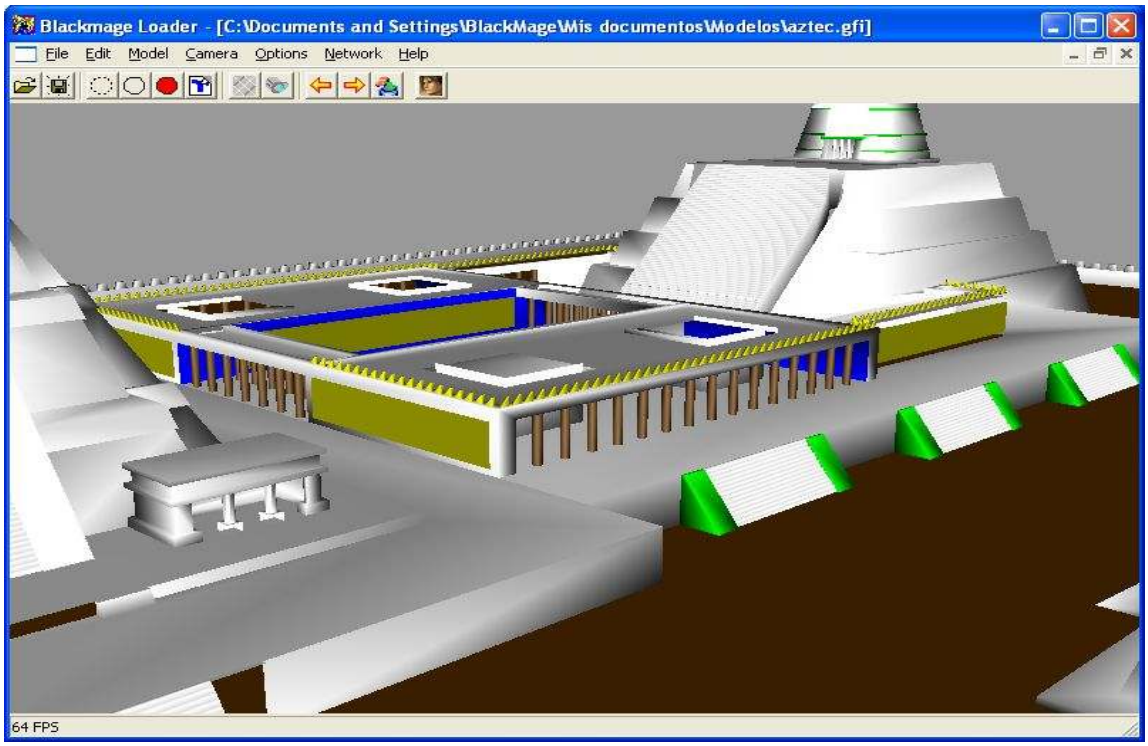
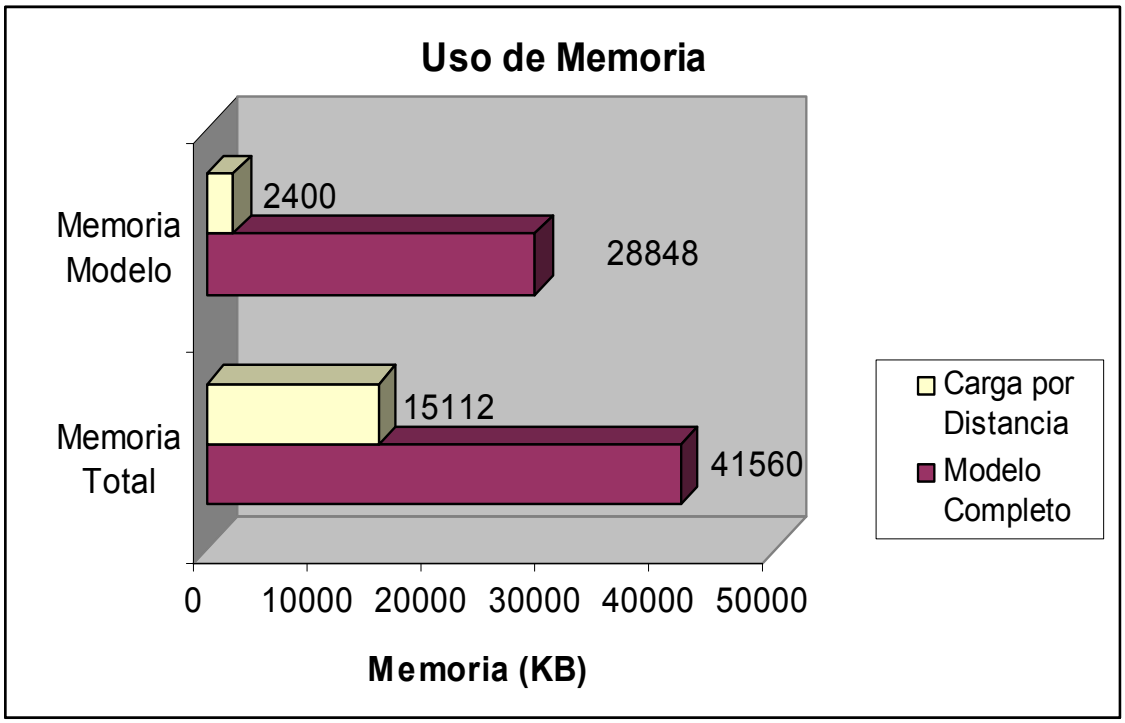
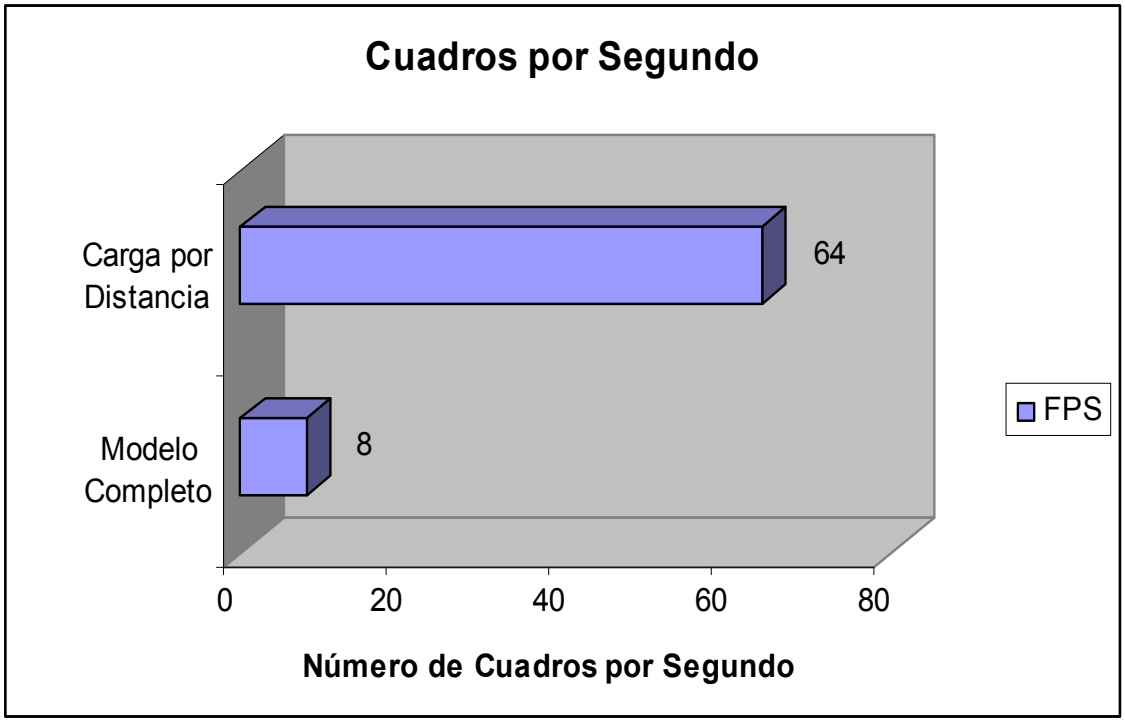


Fig. 5.2.g. Visualización de un modelo cargando los objetos mediante un algoritmo de detección de distancia al observador.



Gráfica 5.2.g. Resultados del uso de memoria del sistema para los dos tipos de carga.



Gráfica 5.2.h. Resultados del desempeño de la visualización, medido en Cuadros por Segundo.

5.3. Resultados.

De las pruebas realizadas se observa que los tiempos de carga de los archivos GFI tienden a ser menores en comparación con los archivos 3DS a medida que el tamaño del modelo aumenta, este porcentaje de tiempo llega a ser hasta de 30% por lo que este formato de archivo es ideal para carga y visualización de modelos tridimensionales muy detallados.

Siendo el Archivo GFI un poco más pequeño que el 3DS, se podría pensar que los tiempos de carga menores serían causa de este hecho, pero esto es engañoso ya que el archivo 3DS vienen separado por chunks o bloques de información, y en esta pruebas solo se lee los chunks con lo información que tiene en común con el formato GFI, de manera que los bytes de lectura por parte del sistema operativo de ambos tipos de archivo son equivalentes.

En la carga por partes los resultados también fueron satisfactorios ya que con modelos extensos que constan de una gran cantidad de objetos, el nuevo formato permitió la carga selectiva de objetos que conforman al modelo. Por ejemplo en la carga por distancia al observador, los tiempos de carga resultaron ser menores permitiendo con esto una visualización sin pausas al momento de la carga de modelos. Además la velocidad del dibujo al cargar el modelo en forma selectiva fue mucho mayor en comparación con la velocidad de dibujo al cargar el modelo completo, esto debido a que solo los objetos más representativos estaban dibujándose. Los objetos que el observador no ve son eliminados permitiendo liberar recursos del sistema.

Además se aseguró que la funcionalidad de carga fuera la misma para cuando se necesita visualizar un modelo que está en un servidor, ya que las características de carga por partes hacen un mejor uso del ancho de banda de la red. Debido a la funcionalidad de la transmisión por Internet, dado sus características inseguras en este tipo de redes, se logró que el protocolo de transmisión soporte detección de errores mediante el algoritmo MD5.

El resultado de la prueba de carga mediante la aplicación cliente-servidor refleja la característica de optimización para su envío por red, ya que la prueba consistió en el envío del archivo completo, para su posterior visualización, obteniéndose un tiempo de carga 23.7% menor de los archivos GFI en comparación con los archivos 3DS.

Este trabajo es el inicio de una serie de herramientas para el desarrollo de aplicaciones 3D que la Facultad de Ingeniería puede usar en sus cursos y proyectos ya que el API de programación permite un rápido desarrollo de aplicaciones de visualización con graficas excelentes y tiempos de desarrollo muy cortos, la secuencia lógica del API es muy sencilla y fácil de usar, incluso puede ser empleado en varias plataformas, ya que la programación fue hecha con

lenguaje C y funciones estándar proporcionadas por el ANSI C, debido a esta facilidad el capitulo estudiantil UNAM de la IGDA (International Game Developers Association) solicitó el uso de este API para el desarrollo de su proyectos que consisten en videojuegos y aplicaciones en tiempo real.

Este mismo trabajo sirve como infraestructura para la continuación del proyecto de visualización de la planta hidroeléctrica “El Cajón” proyecto que la Comisión Federal de Electricidad trabaja en conjunto con la Facultad de Ingeniería y el laboratorio de visualización Ixtli, ya que su versatilidad permitirá agregar las características necesarias al proyecto de visualización que el personal de CFE requiere para la implantación y control de la construcción de proyectos de esta índole.

Debido a que el trabajo de modelado es muy laborioso, se trabajo con los archivo en formato 3DS, para lo cual se investigó la estructura de dicho formato, dando como resultando la documentación más completa y detallada en español acerca de este formato, y que puede ser consultada de manera gratuita.

A pesar de que el objetivo de la tesis es solo la carga eficiente de modelos tridimensionales, en el desarrollo de este trabajo el dibujo en pantalla de los datos contenidos en las estructuras también debe de ser muy eficiente, por lo que uno de los resultados fue una función de dibujo que es muy rápida y eficiente ya que implementa el algoritmo Face Culling y hace uso de funciones de OpenGL para lograr una calidad de visualización muy buena.

5.4. Mejoras al archivo en formato GFI.

Este formato de archivo puede estar sujeto a futuras mejoras, a pesar de que este formato presenta la información completa para dibujar un modelo con gran calidad, sin embargo, debido al avance o aparición de nuevas técnicas, es lógico pensar que para que el formato se adapte a estas nuevas especificaciones se den herramientas necesarias para la adecuación del mismo. Es por ello que se documentó lo más completamente posible las características del formato GFI, para que así esta información sea de ayuda a otra gente interesada en aportar nuevas características a este formato.

Para hacer que una escena parezca lo más real posible, el hardware más reciente presenta muchas características avanzadas tales como multitexturización (que permiten técnicas como el bump mapping), luces, niebla, niebla volumétrica, así como para guardar shaders, que son programas ejecutados por el GPU del chip gráfico para mejorar diferentes efectos de apariencia de los objetos, así como parámetros de animación para los objetos. Estas son características que aún no son soportadas por el archivo en formato GFI.

La información de los modelos en el archivo GFI es puramente poligonal, la cual es una manera de presentar los objetos, sin embargo otra forma para modelar es a partir de información vectorial, como son curvas de bezier, b-splines, las cuales tienen como ventaja una representación de menor tamaño en el archivo. Agregar un bloque con este tipo de información dentro de los archivos en formato GFI hará que se convierta en un formato más versátil y por lo tanto disminuya el tamaño del mismo.

Se puede agregar soporte a diferentes formatos de texturas como pueden ser JPG, TIFF, entre otros.

Conclusiones

A partir de los objetivos planteados se logro la creación de un nuevo formato de archivo, y de sus características se puede concluir lo siguiente.

El formato GFI permite la visualización de la información de manera selectiva, es decir, permite seleccionar solo la información deseada para su representación en el dispositivo de visualización. Esto puede ser realizado por otros formatos de archivos, sin embargo, en estos casos, el programa encargado de la lectura del archivo, requiere recorrer el archivo por completo para así tener conocimiento de la información almacenada, en cambio, el diseño del formato GFI permite obtener esta información, tan solo con explorar las estructuras de datos almacenados en las tablas de objetos y materiales, evitando así recorrer por completo el archivo.

Se cuenta con un archivo que contiene la información de un modelo en 3D con una calidad muy aceptable ya que uno de los resultados más notables es la rutina de dibujo proporcionada por este trabajo que permite dibujar estos modelos con muchas de las opciones con las que son diseñados, por ejemplo los efectos de transparencia así como soporte para una correcta iluminación y texturizado.

Una de las características más importantes de este formato es que permite la transferencia y la carga de la información más significativa desde cualquier medio incluyendo Internet, siendo una alternativa para el formato vrml. Con esto se logró que los recursos de cómputo así como el ancho de banda en una red sean aprovechados de mejor manera, además el hecho de enviar información a través de Internet hizo imperativo un mecanismo de detección de información corrupta, el cual fue implementado para comprobar la confiabilidad de los datos de cada parte del modelo que se está enviando y no solo del archivo completo.

Debido a la estructura definida anteriormente, el formato de archivo puede crecer en los tipos de datos que alberga, ejemplo de ello es la continuación del proyecto Ixtli, en el cual se desea guardar información de recorridos virtuales preestablecidos, y dicha información se define mediante curvas B-Splines, las cuales pueden ser fácilmente incluidas como información adicional al estándar propuesto.

Por todo lo antes mencionado el archivo en formato GFI propuesto cumplió con los objetivos planteados de manera muy satisfactoria.

Actualmente en la Facultad de Ingeniería se imparte la materia de Computación Gráfica, materia en la cual se enseñan los conocimientos previos para el manejo de API de visualización, sin embargo algo que siempre ha sido laborioso y tardado, y que no entra dentro del temario de la materia, es el manejo de modelos tridimensionales, ya que la materia solo se enfoca en trabajar con ellos con las herramientas que el API de visualización proporciona, ya sea OpenGL o Direct3D. Se desarrolló esta API con el propósito de proporcionar una herramienta a la Facultad de Ingeniería para el apoyo a los estudiantes y

profesores de la materia Computación Gráfica, para de esta manera hacer transparente la carga de modelos y con ello enfoquen su atención al manejo del API de visualización. Además la cantidad de líneas de código disminuye bastante ya que solo con la inclusión del archivo de cabecera, con las funciones y la definición de la librería hace que el código sea mucho más entendible ya que el diseño del API tiene una lógica muy intuitiva de manera que es fácil de entender.

En el presente trabajo, se debe recalcar, se implementaron solo algunas de las opciones de carga dentro del API, sin embargo, el mismo API da facilidades para que otros programadores utilicen otros criterios para decidir cuál información es la que debe visualizarse.

Con los resultados de las pruebas de desempeño acerca del diseño de los datos almacenados así como del flujo de programa, se puede concluir que las herramientas desarrolladas son una infraestructura muy sólida para posteriores trabajos relacionados a las gráficas por computadora. El capitulo estudiantil IGDA (International Game Developers Association) de la UNAM de desarrollo de videojuegos, está usando esta librería de manejo de modelos para sus proyectos, ya que además tiene funciones extras que incluyen carga y visualización de modelos con animación en formato md2, logrando con ello centrar su atención en otros aspecto del desarrollo de un videojuego, tales como la programación de un buen modo de juego, sonidos, etc. Otro proyecto que usa este trabajo es el de visualización de la planta hidroeléctrica El Cajón para el Laboratorio de Visualización Ixtli que al momento de terminar esta tesis (abril 2006) continúa en desarrollo, y ahora se está trabajando en detalles como efectos visuales y sistema de navegación automatizada.

Una primera versión de lo que puede hacer este trabajo junto con un modelo proporcionado por los estudiantes de ingeniería civil de la Facultad de Ingeniería de la planta hidroeléctrica en cuestión recibió una calurosa bienvenida por parte los directivos del proyecto de CFE, con lo cual ya están enterados de la calidad del trabajo en cuanto a graficas por computadora realizadas por la Facultad de Ingeniería, trabajo que creemos es de la suficiente calidad para que CFE inicie un convenio de manera más seria. Proyectos como el mencionado anteriormente beneficiarían el desarrollo del área de la graficación por computadora además del campo de conocimiento de la Ingeniería en México.

Por todo lo anterior consideramos que este trabajo de tesis cumplió plenamente con los objetivos planteados al principio de su desarrollo.

Anexo 1.

Documentación del API para la carga de modelos 3D.

Esta es la referencia completa del modo de uso de cada una de las funciones de las clases que en su conjunto permite la carga y manipulación de modelos tridimensionales.

Cabe resaltar que este API de programación esta basado en la funcionalidad que el lenguaje C++ proporciona, por lo que para utilizar este API es necesario un compilador del lenguaje C++, esta clase ha sido probada con los compiladores que Borland y Microsoft distribuyen, es decir, Borland C++ Builder, Visual C++ 7.0. [12]

Clase CModel.

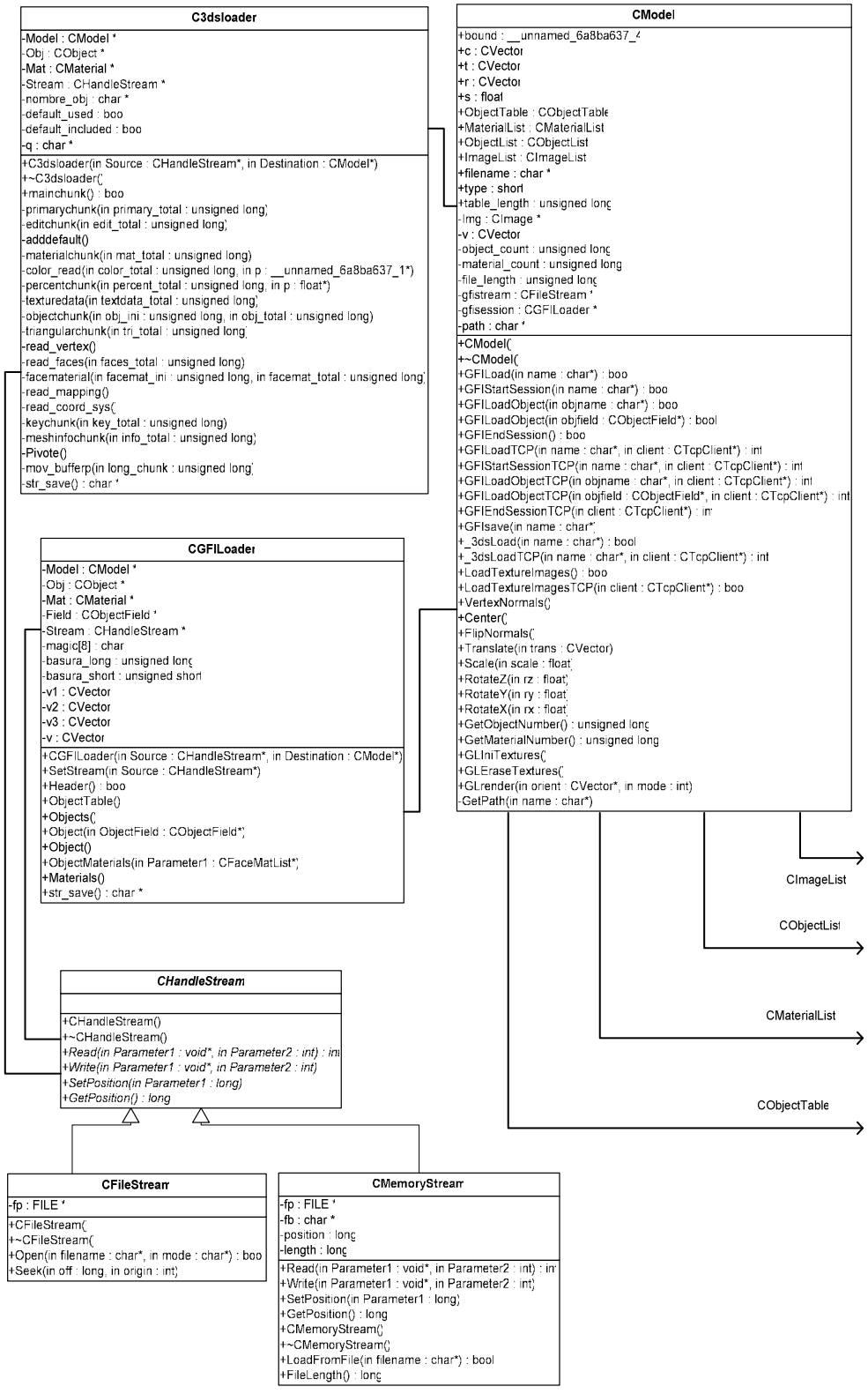
Esta clase es la más importante de todas, ya que por si sola contiene métodos que permiten hacer una rápida implementación para poder cargar modelos tridimensionales GFI y 3DS en casi cualquier proyecto de programación.

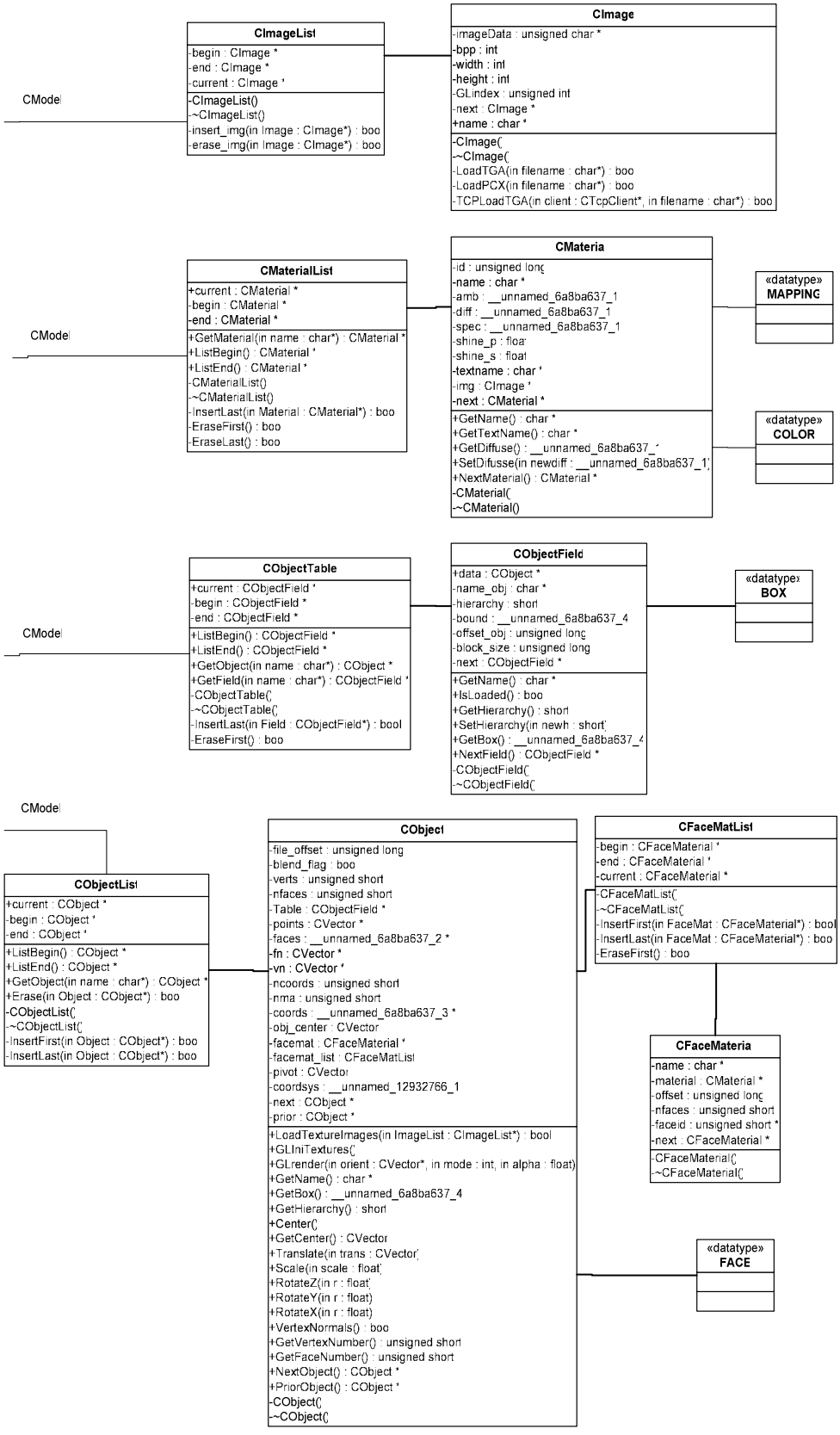
El archivo GFI esta diseñado para ser cargado desde un servidor en una red TCP, es por eso que, junto con las funciones de carga local, se facilitan su equivalente con capacidad de descargar su contenido remotamente, con la única diferencia que se debe de especificar la instancia de la clase CTcpClient por la cual se recibirán los datos desde el servidor.

Una vez que los datos estén completamente en la clase, se pueden hacer diferentes operaciones sobre ellos, como por ejemplo: saber su centroide, las dimensiones de la caja que lo encierra, modificar los parámetros visuales tales como la carga de texturas, la generación de vectores normales a los vértices y colores de los materiales, así como dibujarlo en un contexto de dibujo (rendering context) de OpenGL.

Para ello solo se crea una instancia de la clase CModel, en la cual se guarda el modelo, con esto se tiene acceso a cada una de las funciones diseñadas para su manejo, las cuales se explicarán a continuación.

En las siguientes dos páginas se muestra el diagrama UML de estructuras estáticas, en el cual se muestra cómo se relaciona la Clase CModel con todas aquellas clases que le dan soporte.


C3dsloader
-Model : CModel *
-Obj : CObject *
-Mat : CMaterial *
-Stream : CHandleStream *
-nombre_obj : char *
-default_used : boo
-default_included : boo
-q : char *
+C3dsloader(in Source : CHandleStream*, in Destination : CModel*)
+~C3dsloader()
+mainchunk() : boo
-primarychunk(in primary_total : unsigned long)
-editchunk(in edit_total : unsigned long)
-adddefault()
-materialchunk(in mat_total : unsigned long)
-color_read(in color_total : unsigned long, in p : __unnamed_6a8ba637_1*)
-percentchunk(in percent_total : unsigned long, in p : float*)
-texturedata(in textdata_total : unsigned long)
-objectchunk(in obj_ini : unsigned long, in obj_total : unsigned long)
-triangularchunk(in tri_total : unsigned long)
-read_vertex()
-read_faces(in faces_total : unsigned long)
-facematerial(in facemat_ini : unsigned long, in facemat_total : unsigned long)
-read_mapping()
-read_coord_sys()
-keychunk(in key_total : unsigned long)
-meshinfochunk(in info_total : unsigned long)
-Pivote()
-mov_bufferp(in long_chunk : unsigned long)
-str_save() : char *
CModel
+bound : __unnamed_6a8ba637_4
+c : CVector
+t : CVector
+r : CVector
+s : float
+ObjectTable : CObjectTable
+MaterialList : CMaterialList
+ObjectList : CObjectList
+ImageList : CImageList
+filename : char *
+type : short
+table_length : unsigned long
-Img : CImage *
-v : CVector
-object_count : unsigned long
-material_count : unsigned long
-file_length : unsigned long
-gfistream : CFileStream *
-gfisession : CGFILoader *
-path : char *
+CModel()
+~CModel()
+GFILoad(in name : char*) : boo
+GFIStartSession(in name : char*) : boo
+GFILoadObject(in objname : char*) : boo
+GFILoadObject(in objfield : CObjectField*) : bool
+GFIEndSession() : boo
+GFILoadTCP(in name : char*, in client : CTcpClient*) : int
+GFIStartSessionTCP(in name : char*, in client : CTcpClient*) : int
+GFILoadObjectTCP(in objname : char*, in client : CTcpClient*) : int
+GFILoadObjectTCP(in objfield : CObjectField*, in client : CTcpClient*) : int
+GFIEndSessionTCP(in client : CTcpClient*) : in
+GFIsave(in name : char*)
+_3dsLoad(in name : char*) : bool
+_3dsLoadTCP(in name : char*, in client : CTcpClient*) : int
+LoadTextureImages() : boo
+LoadTextureImagesTCP(in client : CTcpClient*) : boo
+VertexNormals()
+Center()
+FlipNormals()
+Translate(in trans : CVector)
+Scale(in scale : float)
+RotateZ(in rz : float)
+RotateY(in ry : float)
+RotateX(in rx : float)
+GetObjectNumber() : unsigned long
+GetMaterialNumber() : unsigned long
+GLIniTextures()
+GLEraseTextures()
+GLrender(in orient : CVector*, in mode : int)
-GetPath(in name : char*)
CGFILoader
-Model : CModel *
-Obj : CObject *
-Mat : CMaterial *
-Field : CObjectField *
-Stream : CHandleStream *
-magic[8] : char
-basura_long : unsigned long
-basura_short : unsigned short
-v1 : CVector
-v2 : CVector
-v3 : CVector
-v : CVector
+CGFILoader(in Source : CHandleStream*, in Destination : CModel*)
+SetStream(in Source : CHandleStream*)
+Header() : boo
+ObjectTable()
+Objects()
+Object(in ObjectField : CObjectField*)
+Object()
+ObjectMaterials(in Parameter1 : CFaceMatList*)
+Materials()
+str_save() : char *
CImageList
CObjectList
CMaterialList
CObjectTable
CHandleStream
+CHandleStream()
+~CHandleStream()
+Read(in Parameter1 : void*, in Parameter2 : int) : int
+Write(in Parameter1 : void*, in Parameter2 : int)
+SetPosition(in Parameter1 : long)
+GetPosition() : long
CFileStream
-fp : FILE *
+CFileStream()
+~CFileStream()
+Open(in filename : char*, in mode : char*) : boo
+Seek(in off : long, in origin : int)
CMemoryStream
-fp : FILE *
-fb : char *
-position : long
-length : long
+Read(in Parameter1 : void*, in Parameter2 : int) : int
+Write(in Parameter1 : void*, in Parameter2 : int)
+SetPosition(in Parameter1 : long)
+GetPosition() : long
+CMemoryStream()
+~CMemoryStream()
+LoadFromFile(in filename : char*) : bool
+FileLength() : long


CModel
CImageList
-begin : CImage *
-end : CImage *
-current : CImage *
-CImageList()
-~CImageList()
-insert_img(in Image : CImage*) : boo
-erase_img(in Image : CImage*) : boo
CImage
-imageData : unsigned char *
-bpp : int
-width : int
-height : int
-GLindex : unsigned int
-next : CImage *
+name : char *
-CImage()
-~CImage()
-LoadTGA(in filename : char*) : boo
-LoadPCX(in filename : char*) : boo
-TCPLoadTGA(in client : CTcpClient*, in filename : char*) : boo
CModel
CMaterialList
+current : CMaterial *
-begin : CMaterial *
-end : CMaterial *
+GetMaterial(in name : char*) : CMaterial *
+ListBegin() : CMaterial *
+ListEnd() : CMaterial *
-CMaterialList()
-~CMaterialList()
-InsertLast(in Material : CMaterial*) : boo
-EraseFirst() : boo
-EraseLast() : boo
CMateria
-id : unsigned long
-name : char *
-amb : __unnamed_6a8ba637_1
-diff : __unnamed_6a8ba637_1
-spec : __unnamed_6a8ba637_1
-shine_p : float
-shine_s : float
-textname : char *
-img : CImage *
-next : CMaterial *
+GetName() : char *
+GetTextName() : char *
+GetDiffuse() : __unnamed_6a8ba637_
+SetDifusse(in newdiff : __unnamed_6a8ba637_1)
+NextMaterial() : CMaterial *
-CMaterial()
-~CMaterial()
«datatype»
MAPPING
«datatype»
COLOR
CModel
CObjectTable
+current : CObjectField *
-begin : CObjectField *
-end : CObjectField *
+ListBegin() : CObjectField *
+ListEnd() : CObjectField *
+GetObject(in name : char*) : CObject *
+GetField(in name : char*) : CObjectField *
-CObjectTable()
-~CObjectTable()
-InsertLast(in Field : CObjectField*) : bool
-EraseFirst() : boo
CObjectField
+data : CObject *
-name_obj : char *
-hierarchy : short
-bound : __unnamed_6a8ba637_4
-offset_obj : unsigned long
-block_size : unsigned long
-next : CObjectField *
+GetName() : char *
+IsLoaded() : boo
+GetHierarchy() : short
+SetHierarchy(in newh : short)
+GetBox() : __unnamed_6a8ba637_4
+NextField() : CObjectField *
-CObjectField()
-~CObjectField()
«datatype»
BOX
CModel
CObjectList
+current : CObject *
-begin : CObject *
-end : CObject *
+ListBegin() : CObject *
+ListEnd() : CObject *
+GetObject(in name : char*) : CObject *
+Erase(in Object : CObject*) : boo
-CObjectList()
-~CObjectList()
-InsertFirst(in Object : CObject*) : boo
-InsertLast(in Object : CObject*) : boo
CObject
-file_offset : unsigned long
-blend_flag : boo
-verts : unsigned short
-nfaces : unsigned short
-Table : CObjectField *
-points : CVector *
-faces : __unnamed_6a8ba637_2 *
-fn : CVector *
-vn : CVector *
-ncoords : unsigned short
-nma : unsigned short
-coords : __unnamed_6a8ba637_3 *
-obj_center : CVector
-facemat : CFaceMaterial *
-facemat_list : CFaceMatList
-pivot : CVector
-coordsys : __unnamed_12932766_1
-next : CObject *
-prior : CObject *
+LoadTextureImages(in ImageList : CImageList*) : bool
+GLIniTextures()
+GLrender(in orient : CVector*, in mode : int, in alpha : float)
+GetName() : char *
+GetBox() : __unnamed_6a8ba637_4
+GetHierarchy() : short
+Center()
+GetCenter() : CVector
+Translate(in trans : CVector)
+Scale(in scale : float)
+RotateZ(in r : float)
+RotateY(in r : float)
+RotateX(in r : float)
+VertexNormals() : boo
+GetVertexNumber() : unsigned short
+GetFaceNumber() : unsigned short
+NextObject() : CObject *
+PriorObject() : CObject *
-CObject()
-~CObject()
CFaceMatList
-begin : CFaceMaterial *
-end : CFaceMaterial *
-current : CFaceMaterial *
-CFaceMatList()
-~CFaceMatList()
-InsertFirst(in FaceMat : CFaceMaterial*) : bool
-InsertLast(in FaceMat : CFaceMaterial*) : boo
-EraseFirst() : boo
CFaceMateria
-name : char *
-material : CMaterial *
-offset : unsigned long
-nfaces : unsigned short
-faceid : unsigned short *
-next : CFaceMaterial *
-CFaceMaterial()
-~CFaceMaterial()
«datatype»
FACE

GFILoad

Estas funciones cargan los datos de un archivo GFI dentro de la clase CModel, ya que solo estando en la clase pueden ser usados para su dibujo en tiempo real.

bool CModel::GFILoad(char*);

Parámetros

Esta función recibe como parámetro un apuntador a una cadena que contenga el nombre del archivo con formato GFI.

Valor de retorno

Esta función regresa un valor booleano “true” si el archivo se cargó completamente, o regresa “false” si el archivo especificado no tiene el formato “*.gfi” o si no se encuentra dicho archivo.

int CModel::GFILoadTCP(char* , CTcpClient*);

Parámetros

La función requiere un apuntador a cadena con el nombre del archivo ubicado en el servidor, así como un apuntador a una clase CTcpClient previamente conectada a un servidor GFI.

Valor de retorno

Esta función puede regresar una de las siguientes constantes:

MODEL_LOADED	si el archivo fue exitosamente cargado a memoria.
WRONG_FILE	si el archivo no tiene el formato GFI.
NOT_CONNECTED	si el cliente TCP no está conectado antes de hacer la petición.
SOCK_ERROR	si hubo algún problema en el sistema de red.
SERVER_DOWN	si el servidor se apagó antes de completar la petición.
MD5_ERROR	si el archivo sufrió alguna modificación en el canal de comunicación.
FILE_NOT_FOUND	si el servidor no tiene el archivo solicitado.

GFIStartSession

Esta función solo carga a la clase principal, los datos de las tablas de objetos y de materiales desde un archivo con formato GFI, con ellos se puede cargar a la clase solo los objetos que a la aplicación más convenga así como borrar de la clase los objetos que no se necesiten.

bool CModel::GFIStartSession(char*);

Parámetros

Un apuntador a cadena que especifica el nombre del archivo con el cual se estará trabajando y haciendo peticiones de carga de objetos tridimensionales.

Valor de Retorno

Esta función regresa “true” en el caso de cargar completamente las tablas a la clase, y regresa “false” si el archivo especificado no es de tipo “gfi” o si el archivo no se encuentra.

bool CModel::GFIStartSessionTCP(char*, CTcpClient*);

Parámetros

El nombre del archivo con formato GFI desde el cual se cargarán las tablas para la manipulación de la carga del objeto.

Un apuntador a la clase CTcpClient previamente conectada al servidor, desde el cual se cargará la información de las tablas.

Valor de Retorno

Esta función puede regresar una de estas constantes:

TABLE_LOADED	si la información de las tablas ha sido descargada exitosamente.
NOT_CONNECTED	si el cliente no está conectado a un servidor GFI.
WRONG_FILE	si el archivo origen no tiene el formato GFI.
SOCK_ERROR	si hubo algún problema en el sistema de red.
SERVER_DOWN	si el servidor se apagó antes de completar la petición.
MD5_ERROR	si la tabla sufrió alguna modificación en el canal de comunicación.
FILE_NOT_FOUND	si el servidor no tiene el archivo solicitado.

GFILoadObject

Los modelos del formato GFI pueden estar formados por más de un objeto, esta función permite hacer una carga de objetos específicos que se desean tener en memoria en determinado momento.

bool CModel::GFILoadObject(char*);
bool CModel::GFILoadObject(CObjectField*);

Parámetros

Especificando el nombre del objeto, el cual puede ser conocido a través de las funciones de manipulación de la tabla de objetos del modelo (Model → ObjectTable).

De manera directa especificando el campo de la tabla de objetos que tiene su información más significativa.

Valor de Retorno

Esta función regresa el valor "true" si el objeto fue cargado exitosamente, y un valor "false" si el archivo de sesión no está abierto o si no se encuentra el objeto con el nombre especificado.

bool CModel::GFILoadObjectTCP(char*, CTcpClient*);
bool CModel::GFILoadObjectTCP(CObjectField*, CTcpClient*);

Parámetros

Especificando el nombre del modelo, el cual puede ser conocido a través de las funciones de manipulación de la tabla de objetos del modelo (Model → ObjectTable), o de manera directa especificando el campo de la tabla de objetos que tiene su información más significativa.

Y con un apuntador a la clase CTcpClient conectada previamente al servidor por la cual la información será recibida.

Valor de Retorno

Esta función puede regresar uno de estos valores:

OBJECT_LOADED	si la información del objeto ha sido descargada exitosamente.
SESSION_ERROR	si a pesar de estar conectado no se ha iniciado una sesión con un archivo GFI.
WRONG_NAME	si el nombre del objeto no existe en el archivo.
SOCK_ERROR	si hubo algún problema en el sistema de red.
SERVER_DOWN	si el servidor se apagó antes de completar la petición.
MD5_ERROR	si los datos del objeto sufrieron alguna modificación en el canal de comunicación.

GFIEndSession

Esta función cierra la sesión con el archivo, de manera que nuevas llamadas a la función GFILoadObject() no tienen efecto.

bool CModel::GFIEndSession( );
int CModel::GFIEndSessionTCP(CTcpClient*);

Hay que tener cuidado ya que si se vuelve a abrir la sesión con otro archivo diferente, se añadirán estas nuevas referencias de objetos a la tabla, pudiéndose cargar lo nuevos elementos, sin embargo se tendrá un comportamiento inesperado si se desea cargar los modelos del archivo anterior.

Parámetros

Solo para la versión TCP un apuntador a una clase CTcpCLient la cual indica a esta función en que servidor está abierta la sesión con el archivo GFI.

Valor de retorno

Esta función regresa “true” si existe una sesión abierta que cerrar, de otro modo regresa “false”. Para la versión TCP:

SUCCESS	si se cerró correctamente la sesión con el archivo GFI en el servidor.
NOT_CONNECTED	si no se está conectado a un servidor GFI.

GFISave

Esta función guarda el contenido de los datos de la clase CModel a un archivo con formato GFI.

void CModel::GFISave(char*);

Parámetros

Un apuntador a cadena que indica el nombre del nuevo archivo a ser creado, si este nombre ya existe, el archivo se sobrescribirá.

Valor de Retorno

Esta función no regresa ningún valor.

_3dsLoad

Esta función carga a la clase CModel el contenido del archivo de modelos tridimensionales con formato de 3D Studio Max (3DS).

bool CModel::_3dsLoad(char*);

Parámetros

Un apuntador a cadena que contenga el nombre del archivo 3DS a ser cargado a la clase.

Valor de retorno

La función regresa el valor booleano “true” si el archivo fue cargado exitosamente, y un valor “false” si el archivo no se encontró o si no tenía formato 3DS.

int CModel::_3dsLoadTCP(char*, CTcpClient*);

Parámetros

La función requiere el nombre del archivo 3DS en el servidor, así como un apuntador a una clase CTcpClient previamente conectada a un servidor GFI.

Valor de retorno

MODEL_LOADED	si el archivo fue exitosamente cargado a memoria.
WRONG_FILE	si el archivo no tiene el formato 3DS.
NOT_CONNECTED	si el cliente TCP no está conectado antes de hacer la petición.
SOCK_ERROR	si hubo algún problema en el sistema de red.
SERVER_DOWN	si el servidor se apagó antes de completar la petición.
FILE_NOT_FOUND	si el servidor no tiene el archivo solicitado.

LoadTextureImages

Esta función carga a la clase CModel los datos de imágenes necesarias para texturizar el modelo, en este momento las imágenes deben estar en formato TGA, y su longitud y altura en píxeles deben ser iguales y múltiplos de potencias de dos, ejemplo: 8x8,16x16, 32x32, 64x64, etc. [16]

Esta función debe ser llamada después de que el modelo ha sido cargado exitosamente para que tenga efecto.

void CModel::LoadTextureImages( );
void CModel::LoadTextureImagesTCP(CTcpClient*);

Parámetros

La versión para TCP solo requiere una referencia a la clase CTcpClient, la cual proporciona la información a esta función acerca de la dirección del servidor, el cual va a enviar las imágenes.

Valor de retorno

Esta función no regresa ningún valor.

VertexNormals

Esta función calcula los vectores normales a los vértices de todo el modelo, debe ser llamada después de una carga exitosa del modelo para que ésta tenga efecto.

void CModel::VertexNormals( );

Parámetros

Esta función no necesita parámetro alguno.

Valor de retorno

Esta función no regresa ningún valor.

Center

Esta función calcula el centro geométrico del modelo así como los vectores que me limitan la caja mínima que lo encierra (bounding box).

void CModel::Center( );

Parámetros

Esta función no requiere ningún parámetro.

Valor de retorno

Esta función no regresa ningún valor, el valor del centroide puede ser conocido a través de la variable pública “c”, esta variable es de tipo CVector.

FlipNormals

Esta función cambia el sentido de los vectores normales de la superficie del modelo.

void CModel::FlipNormals( );

Parámetros

Esta función no requiere ningún parámetro.

Valor de retorno

Esta función no regresa ningún valor

Translate

Esta función traslada el modelo multiplicando sus componentes por una matriz de traslación, debido a esto la función solo se debe usar para efectos de edición del modelo, ya que el valor de los vértices es completamente modificado por esta matriz, para traslaciones frecuentes usar las funciones del API gráfico correspondiente por ejemplo, glTranslate.

void CModel::Translate(CVector);

Parámetros

Una clase de tipo CVector que indica a la función la posición a la cual el modelo va a ser transladado.

Valor de retorno

Esta función no regresa ningún valor.

Scale

Esta función realiza un escalamiento general al modelo multiplicando sus componente por una matriz de escalamiento, debido a esto esta función solo se debe usar para efectos de edición del modelo, ya que el valor de los vértices es completamente modificado por esta matriz, para escalamientos frecuentes sin alterar las dimensiones del modelo usar por ejemplo glScale.

void CModel::Scale(float);

Parámetros

Un valor flotante que especifica el factor de escala por el cual el modelo será modificado.

Valor de retorno

Esta función no regresa ningún valor.

Rotate

Esta función multiplica los valores de las componentes del modelo por una matriz de rotación, dependiendo el eje en cual el modelo rotará, debido a ello esta función solo se debe usar para efectos de edición del modelo, ya que el valor de los vértices es completamente modificado por dicha matriz, para rotaciones frecuentes sin alterar los valores de las componentes del modelo usar por ejemplo glRotate.

void CModel::RotateZ(float);
void CModel::RotateY(float);
void CModel::RotateX(float);

Parámetros

Un valor flotante que indica el valor del ángulo de rotación en radianes.

Valor de retorno

Esta función no regresa ningún valor.

GetObjectNumber

Esta función permite conocer el número de objetos con los cuales está formado el modelo.

unsigned long CModel::GetObjectNumber( );

Parámetros

Esta función no necesita parámetro alguno.

Valor de retorno

Esta función regresa un valor unsigned long con la cantidad de objetos con los que el modelo está formado.

GetMaterialNumber

Esta función proporciona el número de materiales que el modelo requiere.

unsigned long GetMaterialNumber( );

Parámetros

Esta función no necesita parámetro alguno.

Valor de retorno

Esta función regresa un valor unsigned long con la cantidad de materiales con los que el modelo está asignado.

GLIniTextures

Esta función construye texturas que OpenGL puede usar para su dibujo, a partir de las imágenes cargadas en la ImageList, el identificador en el cual OpenGL las guarda se encuentra en: “CImage.GLindex”.

void CModel::GLIniTextures( );

Parámetros

Esta función no requiere ningún parámetro.

Valor de retorno

Esta función no regresa ningún valor.

GLrender

Una vez cargado el modelo, esta función puede ser llamada en un contexto de dibujo de OpenGL para dibujar el modelo.

void CModel::GLrender(CVector*, int);

Parámetros

Esta función requiere un apuntador a un vector (CVector) que contenga la dirección hacia donde enfoca la cámara, este es necesario si se quiere hacer uso del algoritmo de FaceCulling, que esta función implementa para mejorar el desempeño del dibujo, si no se quiere hacer FaceCulling en el modelo se pasa el valor NULL.

Una constante entera que especifica el tipo de render del modelo:

SHADED	para modelos sólidos con iluminación.
WIRED	para visualizar la malla del modelo.
POINTS	para visualizar los vértices del modelo.

Valor de retorno

Esta función no regresa ningún valor.

Para una mejor visualización estas son las opciones que hay que activar en la maquina de estados de OpenGL.

```
glShadeModel(GL_SMOOTH);
glEnable(GL_NORMALIZE);
glEnable(GL_LIGHTING);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_TEXTURE_2D);
```

Variables Miembro de la clase CModel

CModel::ObjectTable

Cada modelo tiene una Tabla de Objetos, la cual es una clase que manipula elementos de tipo CObjectField, los cuales permiten cargar solo los objetos que interesan según distintos parámetros; por ejemplo: distancia, jerarquía y nombre.

CModel::MaterialList

Cada modelo tiene una Lista de Materiales, la cual es una clase que manipula elementos de tipo CMaterial los cuales guardan los parámetros del aspecto de cada material del modelo, esta Lista de Materiales permite cambiar el color del material que un objeto tiene asignado a su superficie.

CModel::ObjectList

Cada modelo tiene una Lista de Objetos, la cual es una clase que manipula elementos de tipo CObject, de manera que esta lista permite dibujar, trasladar, rotar, escalar y texturizar solo los objetos que interesan, siempre y cuando estos hayan sido cargados por las funciones de carga globales o por medio de las funciones de carga de la clase CObjectTable.

CModel::ImageList

Cada modelo tiene una lista de imágenes, en ella se guardan las imágenes que servirán para texturizar el modelo, internamente la clase CModel se hace cargo de ella, el API de programación la requiere solo a nivel objeto, ya que estos no saben por si solos de cual lista de imágenes obtener sus texturas, así que cada vez que se texturice a nivel objeto, esta lista pasará como parámetro.

CModel::bound

Esta variable es de tipo BOX y contiene dos vectores que indican el tamaño de la caja de volumen mínimo que encierra a ese modelo

CModel::c

Esta variable es de tipo CVector y es un vector que especifica el centro geométrico del modelo

CModel::t CModel::r

Esta variables de tipo CVector realizan traslaciones y rotacion del modelo sin modificar sus datos usando las funciones de OpenGL glTraslate y glRotate, estas se usan solo si se está dibujando el modelo con la funcion GLrender().

CModel::s

Esta constante de tipo float le especifica a la función GLrender la escala con la cual va a dibujar el objeto usando la función glRotate.

Estas últimas tres variables se deben ocupar si es que se quiere usar FaceCulling en el modelo así como las trasformación que OpenGL proporciona, ya que la función GLrender realiza las compensaciones necesarias para que el FaceCulling se realice correctamente.

Clase CObjectField

Esta clase contiene información para la carga de cada uno de los objetos que componen un modelo GFI, información que puede ser conocida sin que el objeto entero este cargado en el sistema, estos datos son su nombre, jerarquía, "bounding box", estos parámetros de identificación de objetos pueden ser usados para tener diferentes tipos de carga, dependiendo las necesidades de la aplicación.

char* CObjectField::GetName( );

Esta función regresa un apuntador a cadena con el nombre del objeto.

bool CObjectField::IsLoaded( );

Esta función regresa un valor booleano que indica si la geometría completa del objeto está cargada en memoria.

short CObjectField::GetHierarchy( );

Esta función regresa un número entero short con el valor de la jerarquía que tiene ese objeto, esta jerarquía puede ser por parte del modelado o puede ser usada para saber cuál modelo tiene prioridad para ser cargado.

void CObjectField::SetHierarchy(short);

Con esta función se asigna otro valor de jerarquía al objeto, dependiendo las necesidades de la aplicación.

BOX CObjectField::GetBox( );

Esta función regresa una estructura de tipo BOX con las medidas de la caja de volumen en la cual está encerrado el objeto, esto le da más resolución a las pruebas de colisiones con el modelo.

CObjectField* CObjectField::NextField();

Con esta función se sabe cuál es el siguiente objeto en la tabla índice.

CObject* CObjectField::data;

Este es un apuntador a los datos sobre la geometría del modelo que puede o no estar en memoria y que proporciona un acceso a las funciones para manipulación del modelo a nivel objeto.

Clase CObject

Esta clase permite manipular cada uno de los objetos que componen un modelo con formato GFI.

bool CObject::LoadTextureImages(CImageList*);

Esta función trata de cargar las imágenes necesarias para texturizar este objeto, como parámetro se necesita un apuntador a la lista de imágenes del modelo completo.

void CObject::GLIniTextures( );

Esta función convierte las imágenes en texturas que OpenGL pueda dibujar para el correcto texturizado del objeto.

void CObject::GLrender(CVector*,int,float);

Esta función dibuja solo un objeto de la totalidad del modelo, al igual que su contraparte de CModel, se puede especificar un vector de posición de la cámara para implementar el face culling así como una constante int para saber el tipo de render, y un flotante para especificar el nivel de opacidad del objeto.

char* CObject::GetName( );

Esta función regresa un apuntador a cadena con el nombre del objeto.

BOX CObject::GetBox( );

Esta función regresa una estructura de tipo BOX con las medidas de la caja de volumen en la cual está encerrado el objeto, esto le da más resolución a las pruebas de colisiones con el modelo.

short CObject::GetHierarchy();

Esta función regresa un número entero short con el valor de la jerarquía que tiene ese objeto, esta jerarquía puede ser por parte del modelado o puede ser usada para saber cuál modelo tiene prioridad para ser cargado.

void CObject::Center();

Esta función calcula el centro geométrico así como los vectores del bounding box, solo se recomienda llamarla una vez por objeto.

CVector CObject::GetCenter();

Esta función regresa un vector de tipo Cvector con el centro geométrico del objeto.

void CObject::Translate(CVector);

Esta función traslada el objeto multiplicando sus componentes por una matriz de traslación, debido a esto la función solo se debe usar para efectos de edición del modelo, ya que el valor de los vértices es completamente modificado por esta matriz, para traslaciones frecuentes usar las funciones del API grafico correspondiente por ejemplo, glTranslate.

void CObject::Scale(float);

Esta función realiza un escalamiento general al objeto multiplicando sus componente por una matriz de escalamiento, debido a esto esta función solo se debe usar para efectos de edición del modelo, ya que el valor de los vértices es completamente modificado por esta matriz, para escalamientos frecuentes sin alterar las dimensiones del modelo usar por ejemplo glScale.

void CObject::RotateZ(float);
void CObject::RotateY(float);
void CObject::RotateX(float);

Esta funciones multiplican los valores de las componentes del modelo por una matriz de rotación, debido a esto esta función solo se debe usar para efectos de edición del modelo, ya que el valor de los vértices es completamente modificado por esta matriz, para rotaciones frecuentes sin alterar los valores de las componentes del modelo usar por ejemplo glRotate.

bool CObject::VertexNormals();

Esta función calcula los vectores normales a los vértices del objeto, debe ser llamada después de una carga exitosa del objeto para que ésta tenga efecto.

unsigned short CObject::GetVertexNumber();

Esta función regresa un entero short con el número de vértices del objeto.

unsigned short CObject::GetFaceNumber();

Esta función regresa un entero short con el número de triángulos del objeto

CObject* CObject::NextObject();
CObject* CObject::PriorObject();

Estas funciones regresan un apuntador al anterior y al siguiente objeto en la lista del modelo, cuando no hay mas objetos el valor regresado es NULL.

Clase CTcpClient

Esta clase permite a la aplicación en donde se implemente, conectarse a un servidor remoto para solicitar descargas de archivos GFI, por medio del protocolo de red TCP.

Su uso es sencillo, simplemente se crea un objeto de tipo CTcpClient, este contiene un método para realizar una conexión remota, especificando el puerto así como la dirección IP del servidor, una vez conectado, las funciones de carga de archivos tridimensionales remota de la clase CModel podrán llevase a cabo satisfactoriamente.

bool ConnectTo(unsigned short, char*);

Esta función se conecta a un servidor de archivo GFI de manera remota, para ello se debe especificar el número del puerto en el servidor GFI que normalmente es el puerto 3490 y un apuntador a cadena con la dirección IP por ejemplo “132.248.52.156”.

Si el servidor aceptó la conexión, el valor de retorno de la función es un valor boolenao “true”, en caso de que el servidor estuviera lleno o no estuviera en servicio, el valor de retorno es “false”

bool Disconnect( );

Esta función termina la conexión con el servidor, su valor de retorno es verdadero si la conexión fue correctamente terminada.

int GetId( );

Esta función regresa un número entero que informa el número de cliente que el servidor asigna al momento de realizar una conexión.

int GetLastLoadingSize( );

Esta función informa el tamaño, en bytes, del último bloque de información solicitado al servidor.

unsigned char* GetLastRecvMD5( );

Esta función regresa un apuntador a cadena, la cual contiene la clave de 16 bytes de la huella digital MD5 calculada por el servidor al momento de enviar el último bloque de información solicitado.

unsigned char* GetLastMD5( );

Esta función regresa un apuntador a cadena, la cual contiene la clave de 16 bytes de la huella digital MD5 calculada por el cliente al momento de recibir el último bloque de información solicitado.

bool GetStatus( );

Esta función informa el estado de la conexión con el servidor, el valor es verdadero si la conexión esta activa y falso si no hay conexión.

void Get(char*);

Esta función descarga un archivo cualquiera y lo guarda en el disco duro del cliente, el parámetro requerido es un apuntador a cadena con el nombre del archivo en el servidor.

Anexo 2

Documentación del formato 3DS.

El archivo 3DS está formado por chunks. Éstos describen que información es la que sigue y lo que ésta hace, y como está representada, su ID y la localización del siguiente bloque. El apuntador al siguiente chunk toma en cuenta el inicio del chunk actual y está dada en bytes. La información binaria en el archivo 3Ds está escrita de una manera especial. Normalmente los bytes menos significativos vienen primero y para el caso de un entero. Por ejemplo: 4A 5C (2 bites en hexadecimal) será 5C el byte más significativo y 4A el menos significativo. En un long esto es: 4A 5C 3B 8F donde 5C4A es la palabra menos significativa y 8F3B es la palabra más significativa. Y para los chunks, los cuales se definen como:

Inicio	*Fin*	*Tamaño*	*Nombre*
0	1	2	Chunk ID
2	5	4	apuntador al siguiente chunk relativo a la posición donde se encuentra el chunk ID, en otras palabras, es la longitud del chunk.

Los chunk tienen una jerarquía, la cual está dada y depende de su identificador o ID. Un archivo 3DS tiene un Primary chunk con ID 4D4Dh. Este es siempre el primer chunk del archivo. Dentro del primary chunk están los main chunks.

Para mostrar una referencia de la jerarquía de los chunks abajo se muestra un diagrama el cual ilustra los diferentes chunk ID's y el lugar que ocupan en el archivo. Cada chunk recibe un nombre por que en el diagrama se listan los chunk ID's y sus correspondientes nombres.

```
MAIN3DS  (0x4D4D)
|
+--EDIT3DS  (0x3D3D)
|  |
|  +--EDIT_MATERIAL (0xAFFF)
|  |  |
|  |  +--MAT_NAME01 (0xA000) (See mli Doc)
|  |
|  +--EDIT_CONFIG1  (0x0100)
|  +--EDIT_CONFIG2  (0x3E3D)
|  +--EDIT_VIEW_P1  (0x7012)
|  |  |
|  |  +--TOP            (0x0001)
|  |  +--BOTTOM         (0x0002)
|  |  +--LEFT           (0x0003)
|  |  +--RIGHT          (0x0004)
|  |  +--FRONT          (0x0005)
|  |  +--BACK           (0x0006)
|  |  +--USER           (0x0007)
|  |  +--CAMERA         (0xFFFF)
|  |  +--LIGHT          (0x0009)
|  |  +--DISABLED       (0x0010)
|  |  +--BOGUS          (0x0011)
|  |
```

```
|  +--EDIT_VIEW_P2  (0x7011)
|  |  |
|  |  +--TOP            (0x0001)
|  |  +--BOTTOM         (0x0002)
|  |  +--LEFT           (0x0003)
|  |  +--RIGHT          (0x0004)
|  |  +--FRONT          (0x0005)
|  |  +--BACK           (0x0006)
|  |  +--USER           (0x0007)
|  |  +--CAMERA         (0xFFFF)
|  |  +--LIGHT          (0x0009)
|  |  +--DISABLED       (0x0010)
|  |  +--BOGUS          (0x0011)
|  |
|  +--EDIT_VIEW_P3  (0x7020)
|  +--EDIT_VIEW1    (0x7001)
|  +--EDIT_BACKGR   (0x1200)
|  +--EDIT_AMBIENT  (0x2100)
|  +--EDIT_OBJECT   (0x4000)
|  |  |
|  |  +--OBJ_TRIMESH   (0x4100)
|  |  |  |
|  |  |  +--TRI_VERTEXL          (0x4110)
|  |  |  +--TRI_VERTEXOPTIONS    (0x4111)
|  |  |  +--TRI_MAPPINGCOORS     (0x4140)
|  |  |  +--TRI_MAPPINGSTANDARD  (0x4170)
|  |  |  +--TRI_FACEL1           (0x4120)
|  |  |  |  |
|  |  |  |  +--TRI_SMOOTH           (0x4150)
|  |  |  |  +--TRI_MATERIAL         (0x4130)
|  |  |  |
|  |  |  +--TRI_LOCAL            (0x4160)
|  |  |  +--TRI_VISIBLE          (0x4165)
|  |  |
|  |  +--OBJ_LIGHT    (0x4600)
|  |  |  |
|  |  |  +--LIT_OFF              (0x4620)
|  |  |  +--LIT_SPOT             (0x4610)
|  |  |  +--LIT_UNKNWN01         (0x465A)
|  |  |
|  |  +--OBJ_CAMERA   (0x4700)
|  |  |  |
|  |  |  +--CAM_UNKNWN01         (0x4710)
|  |  |  +--CAM_UNKNWN02         (0x4720)
|  |  |
|  |  +--OBJ_UNKNWN01 (0x4710)
|  |  +--OBJ_UNKNWN02 (0x4720)
|  |
|  +--EDIT_UNKNW01  (0x1100)
|  +--EDIT_UNKNW02  (0x1201)
|  +--EDIT_UNKNW03  (0x1300)
|  +--EDIT_UNKNW04  (0x1400)
|  +--EDIT_UNKNW05  (0x1420)
|  +--EDIT_UNKNW06  (0x1450)
|  +--EDIT_UNKNW07  (0x1500)
|  +--EDIT_UNKNW08  (0x2200)
|  +--EDIT_UNKNW09  (0x2201)
```

```
|  +--EDIT_UNKNW10  (0x2210)
|  +--EDIT_UNKNW11  (0x2300)
|  +--EDIT_UNKNW12  (0x2302)
|  +--EDIT_UNKNW13  (0x2000)
|  +--EDIT_UNKNW14  (0xAFFF)
|
+--KEYF3DS (0xB000)
   |
   +--KEYF_UNKNWN01 (0xB00A)
   +--.............. (0x7001) (viewport, same as
                               editor)
   +--KEYF_FRAMES   (0xB008)
   +--KEYF_UNKNWN02 (0xB009)
   +--KEYF_OBJDES   (0xB002)
      |
      +--KEYF_OBJHIERARCH  (0xB010)
      +--KEYF_OBJDUMMYNAME (0xB011)
      +--KEYF_OBJUNKNWN01  (0xB013)
      +--KEYF_OBJUNKNWN02  (0xB014)
      +--KEYF_OBJUNKNWN03  (0xB015)
      +--KEYF_OBJPIVOT     (0xB020)
      +--KEYF_OBJUNKNWN04  (0xB021)
      +--KEYF_OBJUNKNWN05  (0xB022)
```

3D Editor Chunks.

Aquí se dará información más detallada acerca de los chunks.

Main chunk.

El main chunk (el chunk primario de 0x4D4D) es realmente el archivo completo. Por ello el tamaño de este chunk es el tamaño del archivo menos la cabecera del main chunk.

Existen dos main chunks más, el 3D-Editor chunk y el Keyframer chunk:

ID	
3D3D	inicio de datos de Editor (este es además el lugar donde se encuentran los objetos)
B000	inicio de datos de Keyframer.

Inmediatamente después del main chunk se encuentra otro chunk. Éste puede ser algún tipo de chunk de entre los permitidos de la siguiente lista de subchunk.

Subchunk de 3D3D.

ID	*Descripción*
0100	parte de la configuración
1100	desconocido
1200	color de fondo

1201	desconocido
1300	desconocido
1400	desconocido
1420	desconocido
1450	desconocido
1500	desconocido
2100	bloque de componente ambiental del color
2200	niebla
2201	niebla
2210	niebla
2300	desconocido
3000	desconocido
3D3E	bloque principal de configuración del Editor
4000	definición de objeto
AFFF	inicio de la lista de materiales.

Subchunks de AFFF (inicio de lista de materiales).

A000	nombre del material.

Este chunk contiene el nombre del material el cual es una cadena ASCIIZ.

Subchunk de 3D3E (configuración del Editor).

ID	*descripción*
7001	inicio del indicador del puerto de vista.
7011	definición del puerto de vista (tipo 2)
7012	definición del puerto de vista (tipo 1)
7020	definición del puerto de vista (tipo 3)

El chunk 3D3E es un chunk raro debido a que contiene una gran cantidad de información redundante (o al menos eso parece). El chunk más importante es 7020. Este chunk describe los 4 puertos de vista que son activados en el editor. La configuración del editor contendrá cinco chunk 7020 y 5 chunks 7011. Sólo los cuatro primeros chunks 7020 serán de importancia para configurar cómo se verán los puertos de vista. La información importante en estos chunks están en los bytes 6 y 7 (tomando como referencia que los 6 primeros bytes son la cabecera del chunk y apuntadores.), estos bytes (unsigned int) contienen la información de qué vista se usa, con los siguientes ID´s:

ID	*Descripción*
0001	Top (Alto)
0002	Bottom (Abajo)
0003	Left (Izquierda)
0004	Right (Derecha)
0005	Front (Frente)

0006	Back (Atrás)
0007	User (Usuario)
FFFF	Camera (Cámara)
0009	Light (Luces)
0010	Disabled (Deshabilitado)

Subchunk de 4000 (Bloque de Descripción de Objetos).

El primer elemento de subchunk 4000 es una cadena ASCIIZ del nombre del objeto. ASCIIZ significa una cadena de caracteres terminados por un cero.

Un objeto puede ser una cámara, una luz o un mesh.

ID	*descripción*
4010	desconocido
4012	sombra
4100	lista de polígonos triangulares (contiene sólo subchunk)
4600	luz
4700	cámara

Mapping (mapeo), estos chunks son opcionales. Se encuentran después de la lista de vértices cuando el objeto es mapeado.

Subchunks del 4100 lista de polígonos triangulares.

ID	*descripción*
4110	lista de vértices
4111	opción de vértices
4120	lista de caras
4130	material de la cara
4140	coordenadas de mapeo
4150	grupo de caras lisas (Face smoothing group)
4160	matriz de translación
4165	objetos visible/invisibles
4170	mapeo estándar

4110 Lista de vértices.

inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	1	2	insigned int	total de vértices en el objeto
2	5	4	float	valor coordenada X
6	9	4	float	valor coordenada Y
10	13	4	float	valor coordenada Z

Los bytes 2 al 13 son repetidos el igual número de veces como cantidad de vértices existan en el objeto.

4111 Opción de vértices.

Primeros 2 bytes: número de vértices.

Después un short int por cada vértice:

Bit 0-7	0
Bit 8-10	x
Bit 11-12	0
Bit 13	vértice seleccionado en opción 3
Bit 14	vértice seleccionado en opción 2
Bit 15	vértice seleccionado en opción 1

Los bits 8 -10 son aleatorios. Cada vez que se guarda la escena, estos valores cambian.

Otros bits (0-7 y 11-12) tienen efectos sobre la visibilidad de los vértices.

El chunk 4111 puede ser ignorado sin ninguna repercusión, ya que el 3DS cargará el archivo de manera correcta.

4120 Lista de caras.

inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	1	2	unsigned int	total de polígonos en el objeto (numpoly)
2	3	2	unsigned int	número de vértice A
4	5	2	unsigned int	número de vértice B
6	7	2	unsigned int	número de vértice C
8	9	2	unsigned int	información de la cara (*)

Se repite ‘numpoly’ veces por cada polígono.

Los primeros tres int son los tres vértices de cada cara. El 0 indica el primer vértice definido dentro de la lista de vértices.

El orden tiene un propósito: dar la dirección para las normales de cada cara.

Para obtener la normal a la cara si se sigue el orden en que se indican los vértices y aplicando la regla de la mano derecha se obtendrá la dirección de la misma.

Si los vértices dados están en el orden A B C:

Aplicando la regla, la dirección de la normal, es hacia el lector.

(*) Este número es un binario el cual se expande a 3 valores. Por ejemplo: 0x0006 se expandirá a 110 binario. El valor deberá ser leído como 1 1 0. Este valor puede encontrarse en los archivos ASCII de 3DS como AB:1 BC:1 AC:0. Lo cual probablemente indica el orden de los vértices. Por ejemplo: AB:1 será una línea normal de A a B. Pero AB:0 será una línea de B a A.

```
bit 0       AC dirección de línea
bit 1       BC dirección de línea
bit 2       AB dirección de línea
bit 3       Mapping (si hay mapeo para esta cara)
bit 4-8   0 (no usado)
bit 9-10  x (???)
bit 11-12 0 (no usado)
bit 13      cara seleccionada en opción 3
bit 14      cara seleccionada en opción 2
bit 15      cara seleccionada en opción 1
```

4130 Face Material Chunk.

Si el material del objeto es el material por default entonces no existirá el chunk 4130. de hecho existe un chunk 4130 por cada material presente en el objeto.

Cada chunk 4130 (face material) empieza con una ASCIIZ de un material, después del carácter nulo es un short int que da el número de caras del objeto que le corresponda este material, después sigue la lista misma de las caras. 0000 significa la primera cara de la lista de caras.

4140 Coordenadas de mapeo.

Primeros 2 bytes: número de vértices.

Por cada vértice hay dos flotantes que corresponden a las coordenadas de mapeo. Esto es, si un punto esta en el centro de la textura este tendrá 0.5 0.5 como coordenadas de mapeo.

4150 Grupo de caras lisas (Face smoothing Group).

Número de caras * 4 bytes

Si se lee como un long int, el enésimo bit indica si la cara pertenece o no a el enésimo grupo liso (smoothing group).

4160 Ejes locales.

Los tres primeros bloques de tres flotantes son la definición de los ejes locales X Y Z del objeto. Y el último bloque de tres flotantes es el centro local del objeto.

4170 Mapeo estándar.

Primeros 2 bytes: tipo de mapeo.

0 = plannar o específico (la información de este chunk es irrelevante)
1 = cilíndrico
2 = esférico

Después vienen 21 valores flotantes que describen el mapeo.

4600 Luz.

inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	3	4	float	posición de la luz eje X
4	7	4	float	posición de la luz eje Y
8	11	4	float	posición de la luz eje Z

Después de esta estructura se verifica otros chunks.

ID	*Descripción*
0010	color RGB
0011	color 24 bit
4610	luz dirigida
4620	luz está activada/desactivada (Boolean)

4610 Luz dirigida.

inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	3	4	float	posición objetivo X
4	7	4	float	posición objetivo X
8	11	4	float	posición objetivo X
12	15	4	float	punto de brillo
16	19	4	float	inclinación.

0010 Color RGB.

inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	3	4	float	Rojo
4	7	4	float	Verde
8	11	4	float	Azul

0011 Color RGB 24 bits.

inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	1	1	byte	Rojo
1	1	1	byte	Verde
2	2	1	byte	Azul

4700 Cámara.

Describe los detalles de la cámara en la escena.

Inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	3	4	float	posición de la cámara X
4	7	4	float	posición de la cámara Y
8	11	4	float	posición de la cámara Z
12	15	4	float	objetivo de la cámara X
16	19	4	float	objetivo de la cámara X
20	23	4	float	objetivo de la cámara X
24	27	4	float	ángulo de rotación
28	31	4	float	lente de la cámara

Keyframer Chunks.

Keyframe chunk.

ID	*descripción*
B00A	desconocido
7001	ver primero descripción de este chunk
B008	frames
B009	desconocido
B002	inicio de descripción del objeto.
B008	información del frame

Estructura básica describiendo información del frame.

Inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	3	4	unsigned long	inicio del frame
4	7	4	unsigned long	fin del frame

B002 Inicio de información del objeto.

Subchunks.

ID	*descripción*
B010	nombre y jerarquía
B011	nombre del objeto dummy
B013	desconocido
B014	desconocido
B015	desconocido
B020	punto pivote de objetos
B021	desconocido
B022	desconocido

B010 Nombre y descriptor de jerarquía.

Inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	?	?	ASCIIZ	nombre del objeto
?	?	2	unsigned int	desconocido
?	?	2	unsigned int	desconocido
?	?	2	unsigned int	jerarquía del objeto

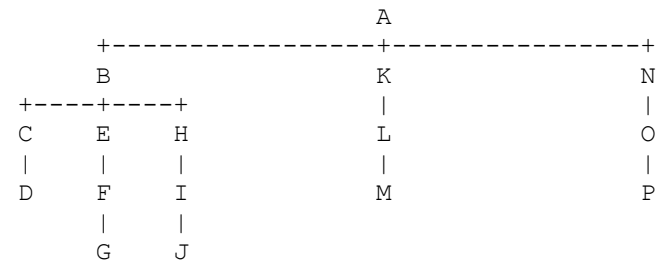
La jerarquía del objeto es un bit complejo pero trabaja de la siguiente manera. Cada Objeto en la escena recibe un número el cual sirve como identificador de su orden dentro del árbol jerárquico. Además cada objeto es ordenado en el archivo 3DS como aparecerá en el árbol.

El *objeto padre* recibe el número -1 (FFFF).

Conforme el archivo se va leyendo se lleva un contador del número de objetos que se han registrado. Mientras el contador aumenta los objetos son hijos de los objetos previos. Pero cuando la cuenta se rompe por un número que es menor que el actual, la jerarquía regresa a ese nivel.

Por ejemplo:

Nombre del objeto	*Jerarquía*
A	-1
B	0
C	1
D	2
E	1
F	4
G	5
H	1
I	7
J	8
K	0
L	10
M	11
N	0
O	13
P	14



Si el nombre del objeto dice $$$Dummy entonces se trata de un objeto Dummy y por lo tanto se debe esperar que aparezcan chunks extras.

B011 Nombre del Objeto Dummy.

Nombre del objeto Dummy, se trata de una cadena ASCIIZ.

B020 Punto Pivote.

El punto pivote del objeto.

inicio	*fin*	*tamaño*	*tipo*	*nombre*
0	3	4	float	desconocido
4	7	4	float	desconocido
8	11	4	float	desconocido
12	16	4	float	desconocido
16	19	4	float	desconocido
20	23	4	float	desconocido
24	27	4	float	Pivote Y
28	32	4	float	Pivote X

Bibliografía.

1. 3D Computer Graphics. Alan Watt. Editorial Addison Wesley. 1993.

2. 3D Graphics File Formats, A Programmer's Reference. Keith. Rule. Editorial Addison Wesley Developers Press. 1996.

3. Advanced Animation and Rendering Techniques, Theory and Practice. Alan Watt, Mark Watt. Editorial Addison Wesley. 1992.

4. Como Programar en C/C++. Deitel Harvey M., Deitel Paul J. Editorial Prentice Hall. 1998. Segunda Edición.

5. Developer's Guide to Multiplayer Games (Wordware Game Developer's Library). Andrew Mulholland, Teijo Hakala. Editorial Wordware Publishing. 2002.

6. Graphics Programming with Direct3D, Techniques and Concepts. Rob Glidden. Editorial Addison Wesley Developers Press. 1997.

7. Introduction to Data Structures and Algorithm Analysis with C^{++}. George J. Pothering, Thomas L. Naps. Editorial West Publishing Company. 1995.

8. Introduction to Information Theory and Data Compression. Darrel Hankerson, Greg A. Harris, Peter D. Johnson Jr. Editorial Chapman & Hall / CRC Press Company. 2003.

9. El Libro de los Cerebros Electrónicos. Walter R. Fuchs. Ediciones Omega. 1975.

10. Mastering Borland C^{++}. Tom Swan. Editorial Prentice Hall Computer Publishing. 1992.

11. OpenGL Programming Guide, The Official Guide to Learn OpenGL, Release 1. Jackie Neider, Tom Davis, Mason Woo. Editorial Addison Wesley Publishing Company. 1993.

12. Teach Yourself Borland C++ Builder in 21 Days. Kent Reisdorph, Ken Henderson. Editorial Sams Publishing. 1997.

Referencias a Páginas web.

13. "The Graphics File Formats Page". 2005.
http://www.dcs.ed.ac.uk/home/mxr/gfx/3d-hi.html

14. "How to Write a Simple Maya Model Exporter". 2005
http://www.gamedev.net/reference/programming/features/mayaexporter/default.asp

15. "MD5 Homepage (Unofficial)". 2005.
http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html

16. "OpenGL Tutorial". Neon Helium Productions (NeHe). 2006.
http://nehe.gamedev.net/

17. "RFC 959 File Transfer Protocol". Internet RFC/STD/BCO Archives.
http://www.faqs.org/rfcs/

18. "VRML97 Functional Specification". Consorcio Web3D. 2005.
http://www.web3d.org

19. "Wotsit's Format, the Programmer's Resource". 2005.
http://www.wotsit.org/

Referencias de Figuras.

20. "Frustum". 2005.
http://www.cadlab.ecn.purdue.edu/cadlab/twin/TWIN_02_PrimRout.html

21. "Frustum Culling". Dion Picco. 2003.
http://www.flipcode.com/articles/article_frustumculling.shtml

22. "Third Year and MSc Interactive Computer Graphics Course. Lecture 05". 2006.
http://www.doc.ic.ac.uk/~dfg/graphics/GraphicsSlides05.pdf

23. "Visible Surface Determination" Brock University. 2004.
http://www.cosc.brocku.ca/Offerings/3P98/course/lectures/hiddenlines/