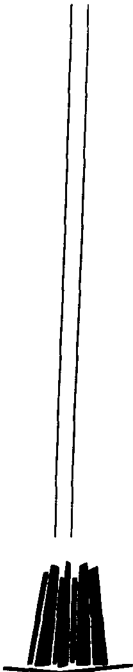
41132
52

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
CAMPUS ARAGÓN



"DISEÑO DEL FORMATO NATIVO PARA EL MODELADOR MATERIAL3D"

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
INGENIERO EN COMPUTACIÓN

P R E S E N T A:

RAMÓN RAMÍREZ GUZMÁN

ASESOR DE TESIS:
M. EN C. MARCELO PÉREZ MEDEL

TESIS CON
FALLA DE ORIGEN

MÉXICO, 2003.

A

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradezco a todos aquellos que colaboraron directa e indirectamente en la elaboración de este trabajo de tesis.

TESIS CON
FALLA DE ORIGEN

RESUMEN.

En el presente trabajo se describen algunas de las partes del modelado de objetos 3D por computadora como la relación entre objetos manejada por medio de las estructuras jerárquicas así como algunas maneras de manejar los objetos que forman al modelo, así mismo se presenta un análisis de las librerías que intervienen directamente con el formato.

También se analizan algunos formatos de modeladores 3D describiendo algunas ventajas y desventajas, y como afecta en eso sus enfoques de trabajo, esto sirve de apoyo al diseño del formato, ya que tiene una base para ver que se puede y que no se debe hacer. Cabe notar que el formato diseñado está optimizado de acuerdo al funcionamiento de Material3D.

Los modeladores 3D se dividieron en 2 tipos, con unos se trabaja directamente sobre los objetos con una interfaz gráfica, a estos se les denomino como modeladores 3D, con los otros denominados como lenguajes de modelado 3D se editan archivos de texto que contienen una serie de instrucciones ordenadas de acuerdo con un conjunto de reglas definidas por el propio lenguaje, estos archivos describen los modelos o escenas que se estén realizando.

Por último se presenta la descripción del formato y la forma en que puede ser utilizado con algunos ejemplos, además de contener un programa con el cual se pueden pasar archivos de 3dstudio al formato de material3D que se diseño.

Índice.

- Resumen 1
- Índice 2
- Introducción 4

1 Capítulo 1. Características de las estructuras 3D 6
  - 1.1 Estructuras jerárquicas 7
    - 1.1.1 Los modelos 8
    - 1.1.2 Jerarquías en los modelos 8
    - 1.1.3 Con transformaciones 9
  - 1.2 NURBS, polígonos y GSC 10
    - 1.2.1 ¿Qué son y para qué sirven? 11
    - 1.2.2 Su uso en modeladores 3D 14

2 Capítulo 2. Análisis de librerías 16
  - 2.1 libxml 17
    - 2.1.1 XML 17
    - 2.1.2 Instrucciones 20
    - 2.1.3 Programa de ejemplo 23
  - 2.2 OpenGL 28
    - 2.2.1 ¿Qué es OpenGL? 29
    - 2.2.2 Vértices y transformaciones 30
    - 2.2.3 Luces 33
    - 2.2.4 Cámara 39

3 Capítulo 3. Formatos de algunos modeladores 3D 42
  - 3.1 3Dstudio 43
    - 3.1.1 Estructura 43
    - 3.1.2 Chunks 47
  - 3.2 Formato obj 48
    - 3.2.1 Objetos 48
    - 3.2.2 Agrupación y conexión 50

4 Capítulo 4. Lenguajes de modelado 3D 51
  - 4.1 POV-RAY 52
    - 4.1.1 Modelado 53
    - 4.1.2 Visión 56
    - 4.1.3 Animación 59
    - 4.1.4 Programación 60
  - 4.2 VRML 62
    - 4.2.1 Nodos 63
    - 4.2.2 Modelado 65
    - 4.2.3 Animación 72

5 Capítulo 5. Diseño del formato para Material3D 75
  - 5.1 Implantaciones del modelador 76
  - 5.2 Formato de Material3D 80
  - 5.3 Instrucciones a futuro 83

6 Capítulo 6. Análisis y discusión de los resultados 87
  - 6.1 Analizador del formato 88
  - 6.2 Ejemplos del formato 93

Conclusiones 106
Literatura citada 107
Referencias 108
Apéndice A: Convertidor de formato 3Dstudio a Material3D 110
Apéndice B: Chunks del formato 3Dstdio 118
Apéndice C: Instrucciones del formato obj 134

INTRODUCCIÓN.

Los archivos que utilizan las aplicaciones de donde leen datos necesarios para poder representar diversos objetos tienen cierta estructura a la cual se le conoce como formato, incluso las imágenes los tienen, de esta manera de pueden diferenciar unas de otras.

Estos tienen cierta estructura que responde a las necesidades de la aplicación en cuanto al número de datos que necesita, el orden de estos y el significado que tienen para el programa.

Ya sabemos a grandes rasgos qué es un formato, pero como en este trabajo se va a diseñar uno para un modelador 3D entonces el enfoque se guía hacia este tipo de formatos, pero ¿qué es un modelador 3D?.

A grandes rasgos, un modelador 3D es un programa que permite crear y modificar estructuras geométricas o mallas en tres dimensiones, que en conjunto forman una escena o un modelo a los cuales se les puede dar animación, esto último lo realizan solamente algunos modeladores 3D. El modelado permite moldear los objetos geométricos que ya tiene predeterminado el modelador y a los cuales se les conoce como primitivas, de esta manera se pueden representar objetos reales como vasos, edificios e incluso se puede modelar una persona ya sea únicamente la cara o el cuerpo entero, además pueden recrearse situaciones que en la vida real es muy difícil o imposible que el ojo humano las pueda captar, o que son muy peligrosas de recrear como un choque entre galaxias o ver a detalle la mezcla entre 2 fluidos. Por lo tanto el modelado cobra mucha importancia al convertirse en una herramienta muy útil tanto para los científicos al representar resultados de sus investigaciones como para ingenieros al poder ver una aproximación del comportamiento de aparatos mecánicos, e inclusive se ha utilizado para el entretenimiento ya sea con las películas o los mismos videojuegos.

Algunos de los modeladores 3D permiten realizar efectos que se ven muy bien en las animaciones pero esto requiere de espacio en el archivo que describe a la escena, mientras que otros modeladores se enfocan a que el modelo esté a las medidas reales o que tenga ciertas características, ya que puede servir para realizar algunos experimentos o investigaciones y como no se necesitan más datos de los que describen al modelo entonces sus archivos deben ocupar poco espacio de almacenamiento, esto dependerá del formato que se utilice. Este enfoque es el que da las características con que cuenta el modelador y que se plasmarán en los modelos como en los ejemplos ya mencionados. Cabe mencionar que ningún modelador es óptimo para todas las actividades en que se han utilizado y aunque varios modeladores tengan el mismo enfoque no significa que utilicen el mismo formato ya que cada uno tiene sus propias herramientas e internamente manejan sus datos de un modo distinto.

Casi todos los modeladores 3D hacen uso del ratón para hacer las transformaciones (rotaciones, translaciones o escalamientos) o deformaciones a las estructuras (o

primitivas) que manejan, además de que la escena o el modelo está siempre presente en la pantalla y los cambios se reflejan inmediatamente.

Pero hay algunos con los que se modela editando archivos directamente con su formato, es como si se estuviera escribiendo un programa, pero en lugar de obtener un archivo ejecutable se obtiene una escena en 3D, la cual se visualiza de distinta manera, dependiendo del modelador. A estos modeladores se les conoce como lenguajes de modelado.

Por sus características únicas estos lenguajes de modelado pueden ser usados de distintas formas y aunque tienen grandes ventajas; como su fácil entendimiento y edición de archivos, también tienen sus desventajas; ya que son menos interactivos y la manipulación es un poco más especializada debido a que los cambios se realizan por medio de los datos de los objetos y no directamente sobre sus superficies. Esto hace que muchos usuarios elijan trabajar con los modeladores 3D y no con estos lenguajes.

Los formatos de los modeladores 3D contienen varias instrucciones que guardan de cierta forma o con cierta estructura los objetos que se están utilizando, así como las transformaciones o efectos que se estén realizando. Algunas de las partes de los formatos no presentan cambios en la escena, pero aún así son importantes como la cabecera, ya que ésta es la que permite a la aplicación (y en ocasiones al sistema operativo) conocer de que tipo de archivo se trata y si pertenece a la aplicación.

No todos los modeladores son comerciales algunos como Material3D[1] que es el modelador para el cual se diseñó el formato son llamados software libre (es decir, su distribución es gratuita), específicamente éste tiene licencia GNU[2] con la que el programa y el código fuente se distribuye sin ningún costo además de que no se tienen restricciones para poder modificar el código o añadirle bloques al programa y que de esta manera el modelador incremente sus herramientas. Este modelador es un proyecto iniciado y desarrollado en la escuela nacional de estudios profesionales campus Aragón[3] por un profesor y algunos alumnos de la carrera de ingeniería en computación.

---

[1] **[1] Análisis y diseño de un sistema de modelado y animación 3D GPL**
[2] **GNU (www.gnu.org)**
[3] **Escuela perteneciente a la UNAM**

CAPÍTULO 1:

CARACTERÍSTICAS DE LAS ESTRUCTURAS 3D

Para iniciar a modelar se debe conocer qué es lo que se quiere representar y que se desea hacer con el modelo, de esta forma se le pueden añadir características para que se le facilite la tarea para la cual se requiere ya sea una animación, una simulación u otro tipo de aplicación.

En general el modelado conlleva a tres pasos; el primero es el modelado formal que es en donde se le da forma a la escena o al modelo incluyendo algunas características que influirán en etapas posteriores (como la adición de un esqueleto), la segunda es la animación en la cual se le da un movimiento predeterminado a cada objeto de la escena, por último se tiene el rendering que toma todos los elementos de la escena y realiza una imagen tomando en cuenta su posición, su color, la cantidad de luz que reflejada o emitida de cada objeto de acuerdo al material asociado a él, etcétera, para plasmarlo en dicha imagen desde un punto de vista dado por una cámara.

En este trabajo solo se revisaron los elementos fundamentales del modelado formal como son la jerarquía entre objetos y algunas formas de representarlos, esto permitió tener un panorama general de los datos que utilizan y la manera en que pueden ser manipulados para poder representarse dentro de un formato.

1.1 Estructuras jerárquicas.

Una jerarquía es una de las formas como se relacionan dos o más entes, en el caso de los modeladores, los objetos (primitivas), las transformaciones y los demás efectos.

La jerarquía define qué parte contiene a qué y qué cosa afecta a los demás. Por ejemplo, podemos definir primero una caja, después trasladarla y ponerle color. Con esto estamos diciendo que la caja contiene una transformación y un atributo de color, pero no que la transformación va a contener al color. Así es la manera de trabajar de POV-RAY como veremos en el capítulo 3, pero hay quienes manejan distinto el orden de los elementos, por ejemplo OpenGL, que primero realiza la transformación y luego define el objeto a dibujar (también se verá más adelante), por lo que la transformación contiene a la caja.

TESIS CON
FALLA DE ORIGEN

Figura 1.1: Arbol del ejemplo de la caja.

Las jerarquías que se generan pueden ser representadas con gráficas dirigidas o con árboles. De este modo se puede ver con mayor claridad cuáles son las relaciones entre

los distintos elementos y cuáles elementos están contenidos en otros. En la figura 1.1 se puede observar el árbol del ejemplo anterior (se nota que se omite poner flechas ya que en un árbol no es necesario debido a que se sabe que su recorrido es de arriba hacia abajo, en cambio en una gráfica dirigida sí son necesarias).

1.1.1 Los modelos.

Un modelo es una representación de algunas o todas las características de una entidad concreta o abstracta, el modelo puede ser físico, como las maquetas o abstracto, como las fórmulas matemáticas. Pero no todos los modelos son fueron interés para el diseño del formato, solo interesaron los que utilizan los modeladores, estos son conocidos como modelos geométricos.

Como los modelos geométricos trabajan con objetos tales como esferas, conos, cilindros, etc., fue conveniente tomar a los polígonos hechos con vértices como un objeto más.

Los objetos no son todo lo que definen esos modelos, también se tienen atributos como el color y transformaciones como la rotación. Estos interactuan entre sí para realizar el modelo, formando las estructuras jerárquicas correspondientes.

TESIS CON
FALLA DE ORIGEN

1.1.2 Jerarquías en los modelos.

Ahora se verá cómo se realizan las jerarquías en los modelos geométricos con un ejemplo simple. Como ya se vio, se necesita definir qué parte del modelo va a ser la raíz, la cual contendrá a las demás en un cierto orden.

figura 1.2: escorpion hecho con cajas (con mucha imaginacion)

Primero debemos ver qué parte es la principal para tomar a ésta como raiz. Se puede tomar a las primitivas ya que a éstas se les aplican las transformaciones y contienen los atributos, hay que notar que se pudo tomar como raíz cualquiera y no forzosamente las primitivas. Por ejemplo, OpenGL aplica la transformación al sistema de coordenadas y después define los objetos, pero en este caso lo se realiza de esta forma.

Teniendo ya un modelo se deben ver las relaciones que hay entre las distintas partes del modelo, ya que una primitiva puede estar ligada con otra con respecto a su

movimiento ya que al mover la principal se debe mover la que tiene ligada pero no necesariamente se deben mover las dos al mover la secundaria.

Para poder entender mejor esto, se tomó el modelo de la figura 1.2 que podría representar un escorpión hecho a base de puras cajas, donde tenemos la cola, las tenazas y el cuerpo. Para comenzar, se observa que tanto la cola como las pinzas están unidas al cuerpo, por lo tanto si se mueve el cuerpo se deberán mover estos, pero si se mueve una pinza o la cola, no necesariamente se tendrán que mover las otras partes, por lo que el cuerpo es tomado como la raíz de la estructura, esto se puede representar como lo indica la figura 1.3, hay que notar que la cola está compuesta por dos cajas y aunque estén pegadas se comportan como objetos distintos; ya que al rotar la parte final no sufre cambios la parte inicial.

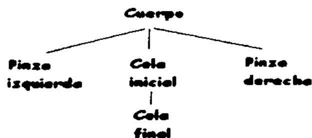


figura 1.3: árbol del escorpión

Las pinzas aunque tengan las mismas características también son objetos independientes uno del otro aunque dependientes del cuerpo.

Como se ha visto, las estructuras jerárquicas ayudan a realizar transformaciones en un modelo con mayor facilidad ya que permiten conocer los objetos que dependen de otros y al aplicar la transformación ver si le afecta a otro objeto y de que forma. Algunos modeladores 3D las utilizan explícitamente al estar agrupando objetos para facilitar la animación del objeto o simplemente para tener un orden y poder localizar o seleccionar primitivas más fácilmente.

1.1.3 Con transformaciones

TESIS CON
FALLA DE ORIGEN

Se ha hablado mucho sobre la facilidad que dan las estructuras jerárquicas para aplicar transformaciones, pero ¿cómo se aplicarían estas? A continuación se ve cómo puede quedar una estructura con éstas, utilizando de nueva cuenta el ejemplo del escorpión pero ahora tomando en cuenta que todas las cajas son de la misma medida y están en el origen, por lo tanto para poder obtener el modelo se aplican transformaciones de tal manera que quede como la figura 1.2.

Primeramente se toma la estructura antes obtenida y sobre ésta se aplican las transformaciones. Para las pinzas se debe escalar, rotar y trasladar, la translación y la

rotación se debe hacer de la misma magnitud pero en sentido contrario, el escalamiento es el mismo.

Ahora para la cola hay que ver que su segmento final es afectado por su segmento inicial, pero solamente le afectan la rotación y la translación por lo que el escalamiento se le aplica aparte y las otras dos se aplican a toda la cola en conjunto. También el segmento final tiene otras transformaciones y como está no afecta a las demás cajas entonces solo se le aplicarán a ella.

Ya por último se ve que el modelo está en su totalidad rotado, por lo que se realiza la rotación a todo el modelo como si se tratara de un mismo objeto, de esta manera se obtiene la estructura de la figura 1.4.

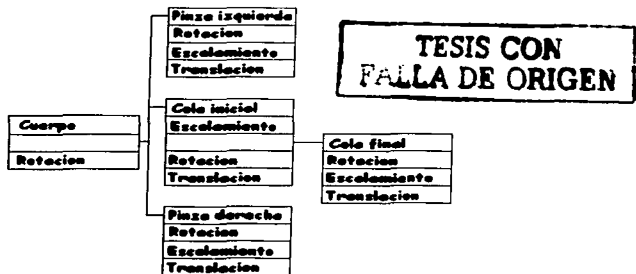


figura 1.4.

Solo falta poner los atributos tales como el color o la iluminación del modelo, su comportamiento es el mismo que con las transformaciones sólo que estas se cambian por el atributo aplicado.

1.2 NURBS, polígonos y GSC.

Los modeladores 3D tienen distintas formas de manejar sus primitivas según las características de este. Las tres formas principales son con NURBS(No Uniform R Bezier Surfaces), con polígonos y con GSC (Geometría Sólida Constructiva), cada una de estas formas tienen una serie de ventajas y desventajas con lo cual unas se pueden

utilizar en unos casos y otras en casos distintos, esta es la razón por la cual un modelador no es útil para todos los problemas.

Estas formas influyen mucho en la intención de trabajo del modelador ya que dan la pauta para ver qué tipo de datos maneja y cómo se maneja.

Estos pueden ser muy parecidos pero cada uno tiene características únicas, aunque algunas si las comparten (sobre todo los NURBS y los polígonos, que se podría pensar que son lo mismo).

TESIS CON
FALLA DE ORIGEN

1.2.1 ¿Qué son y para qué sirven?

Estos, como ya se dijo, son formas de manejar las primitivas de los modeladores, pero esto es a grandes rasgos, por lo que a continuación se explica cada uno por separado.

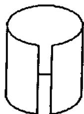


figura 1.5

Los NURBS son primitivas formadas por mallas, estas mallas son transformadas para que tengan la forma de la primitiva deseada, ya que al principio tienen la forma de un plano (aunque esto nunca se ve). Es como se tomara una hoja de algún material maleable y se moldeara de cierta manera para que tome una forma deseada, por ejemplo, para hacer un cilindro se toma el plano y se tiene que juntar dos esquinas de un mismo lado, esto también se debe realizar con las otras dos esquinas pero sin doblar el plano, sino que intentando dejar una forma circular como se muestra en la figura 1.5. Por lo tanto, en los NURBS las superficies curvas no están compuestas por una serie de líneas rectas, por lo cual se tendrá mayor realismo pero se tendrán que realizar unos cuantos procesos para lograr esto.

Los polígonos son una colección de vértices que están conectados en cierto orden por líneas donde el polígono en total estará siempre cerrado, es decir, la última línea une al primer y último vértice. Matemáticamente podemos definir un polígono como sigue: Sean $v_1, v_2, \ldots, v_n$, n puntos en el plano, y $e_{(1,2)}, e_{(2,3)}, \ldots, e_{((n-1),n)}, e_{(n,1)}$ las líneas que unen a esos puntos (también llamados vértices), entonces podemos decir que definen un polígono, donde el número de vértices y de líneas son iguales. Estos vértices están esparcidos en cierto orden, formando la primitiva deseada, pero ya que estos están unidos por líneas, no se puede lograr una superficie totalmente curva, por eso el polígono estará formado por una serie de caras planas como muestra la figura 1.6. Entre más polígonos tenga un modelo más real se verá (como se ve en la figura, si tuviera más líneas se podría ver mejor la curva del cilindro), pero eso implica mas tiempo de procesamiento y mayor espacio en memoria por lo que tendrá que cargar más datos. Por lo tanto se necesita ver qué tanto realismo y velocidad para desplegar el modelo se requiere.

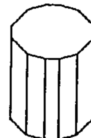


figura 1.6

TESIS CON
FALLA DE ORIGEN

Al estar trabajando con polígonos se tienen algunos problemas que debemos tomar en cuenta. El polígono de la figura 1.6 es simple ya que ninguno de sus componentes atraviesa a otro o toca a más de dos elementos. Sin embargo tenemos polígonos llamados no simples, estos representan un problema a la hora de triangular.

Los polígonos al igual que los NURBS nos sirven como una forma para representar las primitivas de un modelador y así poderlas desplegar en pantalla. Como ya se mencionó, estos dos son muy parecidos aunque tienen sus diferencias, pero se pueden ocupar igual, la diferencia más notable entre estos es que los NURBS pueden representar superficies totalmente curvas gracias a que están definidos por ecuaciones mientras que los polígonos presentan aproximaciones donde entre más vértices se tengan se tendrá una mejor aproximación a una superficie curva, sin embargo siempre estarán formados por caras pero son más fáciles de representar y manipular dentro de un programa. Una de las ventajas de estos es que podemos moldear las primitivas que forman gracias a que nos permiten tomar sus vértices y moverlos como se quiera formando así distintas figuras y en estos recae un mayor interés en este trabajo debido a que es la forma en que Material3D maneja sus primitivas.

Ahora la GSC es una herramienta muy poderosa ya que permite realizar operaciones booleanas como la unión, intersección, complemento y diferencia con las primitivas que tenemos a nuestra disposición. Esta utiliza comúnmente primitivas sólidas, este tipo de primitivas las define el modelador con las ecuaciones de la figura deseada, por ejemplo para definir una esfera se tendrá que utilizar su ecuación $Ax+By+Cz+D=R$, de esta forma se construyen todas las primitivas, aunque también puede ser utilizada con polígonos.

En la unión los objetos involucrados se toman ahora como un solo objeto, por lo que al aplicar una rotación a la unión, todas las primitivas definidas dentro de ésta, se comportan como un mismo ente, por lo que el objeto formado por éstas no se modificará.

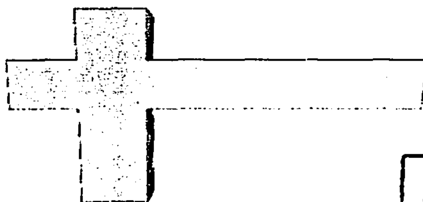


Figura 1.7: Unión

TESIS CON
FALLA DE ORIGEN

La intersección muestra únicamente las partes de los objetos que se tocan, este operador se comporta de igual manera que la unión, la diferencia es la parte que se muestra de las primitivas. En la figura podemos observar que la única parte que se ve es el pequeño cubo definido por el área donde se tocan las dos barras.

TESIS CON
FALLA DE ORIGEN

Figura 1.8: Intersección

El complemento es todo lo contrario a la intersección, ya que éste muestra las partes de los objetos que no se tocan. Al igual que la intersección, este se comporta de la misma forma que la unión.

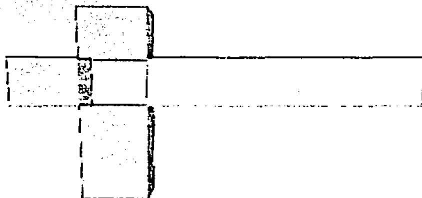


Figura 1.9. *Complemento*

La diferencia es un operador muy poderoso ya que permite quitarle partes a los objetos, obteniendo resultados que con los otros operadores es muy difícil o prácticamente imposibles de obtener. Con dicho operador se tiene un objeto base, el cual es el que se va a mostrar, a éste se le quitarán las partes que toquen con él, o los otros objetos que se estén restando, pero éstos objetos no tendrán ningún efecto entre ellos mismos. También el resultado se comporta como un mismo objeto al igual que en la unión.

Figura 1.10. Diferencia

Estos operadores pueden ser utilizados conjuntamente para crear modelos más complicados, por ejemplo, se puede tener una unión compuesta por una intersección y una diferencia, o una diferencia de un objeto al cual le quitemos otra diferencia. Lo que no se puede es utilizar una operación en ella misma, por ejemplo, no es posible realizar

la unión de un objeto con esa misma unión ya que esta última no está totalmente definida, por esta razón no se puede realizar este tipo de recursividad.

1.2.2. Su uso en modeladores 3D.

Como se ha visto, los NURBS y los polígonos son muy parecidos entre sí, tanto que los dos se utilizan para representar los objetos que se utilizan para realizar los modelos y la escena, cada uno los representa de forma distinta de acuerdo a sus características por lo que dependiendo lo que se requiera modelar se utilizarán unos o los otros.

A diferencia de éstos, la GSC no se utiliza para representar los objetos, si no que ésta es utilizada para aplicar sus operaciones (las cuales ya se han revisado) y de esta forma obtener modelos más complicados.

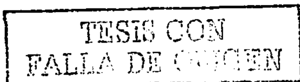
Con lo anterior salta una pregunta ¿pueden utilizarse juntos? Es decir ¿pueden ser utilizados en una misma escena los NURBS y los polígonos para representar los objetos de esta manera utilizar sus ventajas en toda la escena disminuyendo así las desventajas de estos y posteriormente poder combinarlos con las operaciones booleanas que brinda la GSC?.

Para empezar se tienen los NURBS y los polígonos, no es raro ver que los modeladores 3D puedan estar trabajando conjuntamente con estos en una misma escena. El problema aquí es ¿la GSC no puede utilizarse para aplicarles sus operaciones a los NURBS y a los polígonos? La respuesta es en parte negativa, pero también en parte positiva.

En primera, los polígonos son descritos como pequeños segmentos de planos pegados entre sí para formar un objeto más complicado y como la GSC también puede trabajar con segmentos de planos entonces sí puede aplicar sus operaciones a los polígonos sin mayor problema, de hecho una de las formas de trabajar al estar utilizando la GSC es creando mallas conformadas por triángulos, para definir estas mallas se va dando la localización de cada uno de los vértices de estos triángulos, de la misma manera, los polígonos tiene cada una de las coordenadas de sus vértices por lo cual estas mallas poden considerarlas como polígonos.

Ahora, que pasa con los NURBS, al ser estos una malla en la cual se mantiene una curva por medio de algunos métodos matemáticos y no se tienen todos los puntos de esta superficie, faltan varios puntos para poder hacer lo mismo que con los polígonos y ya que estas superficies no se rigen por medio de una ecuación adecuada para este método, no se les podrán aplicar las operaciones definidas por la GSC.

Lo anterior no significa que no puedan ser utilizadas en una misma escena, ya que aunque no se pueda aplicar la GSC a los NURBS sí pueden ser incluidos sin estas


TESIS CON

transformaciones, de hecho el modelador Rhinoceros[1] en su versión de evaluación 1.0 muestra estas tres opciones para poderlas utilizar dentro de una misma escena.

[1] www.rhino3d.com

CAPÍTULO 2

ANÁLISIS DE LIBRERIAS

TESIS CON
FALLA DE ORIGEN

2.1 libxml.

Es un conjunto de bibliotecas para C que es de gran ayuda para manejar los documentos escritos con XML, siendo ésta muy útil para el proyecto, ya que el formato se escribió en este lenguaje. Estas son un poco recientes y las instrucciones que se necesitarán utilizar para la revisión de los archivos con el formato que se diseñó no son muchas disminuyendo un poco la complejidad del parcer[1].

Antes de comenzar a ver las instrucciones de libxml, es necesario saber la idea general que maneja XML ya que realmente este lenguaje es el que se utilizó para el diseño del formato.

2.1.1. XML.

Este es un lenguaje de marcas como su nombre lo indica (lenguaje de marcado extensible), a estas se les conoce como tags y van encerrados entre los signos "<" para iniciar y ">" para finalizar una marca. Este es muy parecido a HTML en su forma de manejarse, solo que XML no acepta que sus archivos tengan partes en su estructura que no sigan con las reglas del lenguaje. Otra diferencia muy marcada entre estos dos lenguajes es el número de instrucciones que tiene, ya que mientras HTML tiene un grupo definido de estas, XML no tiene un límite debido a que es extensible, esto se refiere a que uno mismo puede definir su conjunto de instrucciones, solo se debe cuidar que cumplan con las reglas marcadas para este lenguaje. De hecho, HTML puede ser definido en su totalidad utilizando XML aunque el primero no sea un subconjunto del segundo, sin embargo no quiere decir que XML sea un intento de sustituir a HTML aunque es cierto que últimamente se está utilizando con mayor frecuencia en Internet gracias a sus características que logran hacer los sitios en la red más personalizados y con un mejor control de los usuarios que lo accedan aún cuando su aplicación más común son las bases de datos. Otra de las formas en que es utilizado es en la definición de formatos de distintas aplicaciones (por ejemplo abiword[2]), lógicamente esta es la característica que se utilizo para el formato.

Las instrucciones que se añaden al lenguaje son definidas en un DTD (definición de tipo de documento), de esta forma varios archivos de XML utilizan el mismo DTD para su estructura.

Para iniciar un documento XML se debe poner una marca que lo indique conteniendo la versión que se utilizará, la línea debe estar al inicio del archivo definida como sigue:

---

[1] Es el bloque que se encarga de leer los datos del archivo e interpretarlos para mandarlos al lugar indicado y poder representarlos de una manera correcta dentro del programa
[2] Procesador de texto en Linux (www.abiword.com)

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

Donde versión indica la versión que se utilizará (por ahora la 1.0), encoding es el tipo de código que se utilizará para codificar y decodificar el conjunto de caracteres con el cual esta escrito el documento (puede ser UTF-16, US-ASCII, etc.), y standalone indica si el documento utilizará algún DTD definido por un usuario ("no") o si no utilizará ninguno ("yes"), este último campo se utiliza más que nada para los DTD. No es necesario poner los 2 últimos campos con lo que sus valores por omisión son los que se muestran en el ejemplo.

La DTD puede ser definido en un archivo aparte (que para el caso del formato fue la mejor opción) incluyendo en el documento XML la siguiente línea para poder conocer la ubicación del DTD a utilizar:

<!DOCTYPE tipo SYSTEM "http://.../archivo.dtd">

De esta forma se utiliza el DTD llamado archivo.dtd que se encuentra en la ruta indicada por SYSTEM, DOCTYPE indica el tipo de documento, en este campo se puede ingresar el nombre de la estructura de mayor jerarquía definida en el DTD. Otra forma de utilizar un DTD es incluyéndolo directamente en el archivo XML de la siguiente forma:

```
<?xml version="1.0"?>
<!DOCTYPE tipo [
...
]>
documento xml.
```

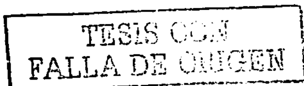
La DTD está definida dentro de "[" y "]" que se encuentran en DOCTYPE, el tipo solo es para darle nombre al DTD, seguido de esto se pone el documento xml.

La forma de definir las marcas de un DTD es la siguiente.

<!ELEMENT nombre_marca contenido>

El nombre_marca es el nombre de referencia de la marca (por ejemplo con HTML el nombre de la marca para cambiar los atributos del texto es "text"), enseguida tiene los datos que manejará o contendrá, aquí varia la forma de declararlos de acuerdo a como se quiera implementar. Una forma es definirlos dentro de la misma marca, la otra es poner estos datos u otras marcas delimitadas entre su marca de inicio y final, para esto el elemento no debe ser vacío (como el salto de línea <BR> en HTML).

En el contenido además de marcas pueden ir datos declarados con #PCDATA el cual indica que se puede ingresar una cadena de caracteres alfanuméricos. Para declarar una marca vacía después de su nombre se pone EMPTY.



Como ejemplo se utiliza un pequeño formato para guardar un dibujo en 2D que contenga elipses, rectángulos y líneas donde se pueda indicar el color además de la posición a donde se quiera mover (originalmente que este en el centro).

```
<?xml version="1.0"?>
<dibujo ancho="100" alto="100">
<color fondo="CYAN"/>
<elipse radiox="3" radioy="4">
<color color="ROJO"/>
</elipse>
<rect ancho="6" alto="4">
<color color="VERDE"/>
<mover>10 28</mover>
</rect>
<linea x1="25" y1="20" x2="30" y2="10">
<mover>50 80</mover>
<color RGB="231244"/>
</linea>
<elipse radiox="5" radioy="5">
<color color="VERDE"/>
<mover>60 40</mover>
</elipse>
</dibujo>
```

Todas estas marcas deben estar definidas en su correspondiente DTD, por ejemplo para el elemento raíz se debe tener algo como lo siguiente:

```
<!ELEMENT dibujo (color?, elipse*, rect*, linea*)>
<!ATTLIST dibujo ancho (#PCDATA) #REQUIRED
        alto (#PCDATA) #REQUIRED>
```

Donde !ELEMENT indica el nombre de la marca y entre paréntesis el nombre de las marcas que puede contener, estos tienen un signo al final de cada uno el cual indica el número de veces que se puede utilizar dependiendo del signo que tenga, estos signos pueden ser:

+: Indica una o más ocurrencias de la marca.
*: Indica cero o más ocurrencias.
?: Indica cero o una ocurrencia.

Debajo de la línea !ELEMENT esta !ATTLIST, ésta contiene la lista de atributos de la marca en cuestión, para el ejemplo son el ancho y el alto con sus respectivos valores que manejan encerrados entre paréntesis, al final de cada uno se tiene una etiqueta la cual puede ser una de las siguientes:

#REQUIRED: Es obligatorio poner el atributo cada vez que se ponga la marca.
#IMPLIED: Se puede poner u omitir el atributo cada vez que se pone la marca.
#FIXED: El valor del atributo es el mismo para todas las marcas de este tipo.

De esta forma se tiene un pequeño ejemplo de un archivo XML, el DTD no es de gran ayuda para este trabajo como se ve a continuación.

2.1.2. Instrucciones.

Este conjunto de bibliotecas contiene muchas instrucciones y estructuras pero para hacer la revisión del formato no es necesario ocupar todas por lo que no serán explicadas todas en este trabajo. Estas bibliotecas trabajan sobre un árbol que estas mismas crean con lo que solamente se debe recorrer para obtener todos los valores necesarios que se encuentran en los distintos niveles.

libxml tiene algunos tipos de datos ya declarados con algunas características especiales que los hace diferir de los tipos del lenguaje C. Los tipos que datos que se utilizarán para el formato son los siguientes:

- xmlChar: Se utiliza para reemplazar el tipo char, es un byte en una cadena codificada con UTF-8.
- xmlDoc: Es una estructura donde se almacena el árbol que se crea al parcear el documento XML.
- xmlDocPtr: Es un apuntador a la estructura xmlDoc.
- xmlNode: Es una estructura que almacena una marca con todos sus atributos y su contenido, ésta tiene un apuntador a la siguiente marca y a las marcas que contiene (marcas hijas en la jerarquía del árbol).
- xmlNodePtr: Es un apuntador a la estructura xmlNode.
- xmlNS: Es una estructura que maneja el espacio de nombre de las marcas.
- xmlNsPtr: Es un apuntador a la estructura xmlNs.

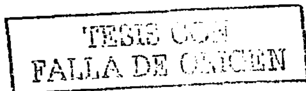
Como se mencionó, existen más tipos de datos pero estos son los básicos y los que se utilizan para la revisión de los archivos con el formato que se diseñó.

Ahora se necesita saber como realizar el parcer del archivo para de esta forma obtener el árbol con el que se trabajó, de esta forma se puede obtener el nodo raíz para comparar si su nombre es igual al nombre de la marca que contiene a todas las demás. Para hacer el parcer se utiliza la siguiente instrucción.

```
xmlDocPtr xmlParceFile(char *archivo);
```

Esta función tiene como parámetro el nombre del archivo en una cadena de caracteres y regresa el puntero de la estructura que contiene el árbol, si la función falla esta regresa un nulo (NULL). Con la siguiente función se obtiene el nodo raíz.

```
xmlNodePtr xmlDocGetRootElement (xmlDocPtr doc);
```



Esta tiene como parámetro el puntero a la estructura xmlDoc que contiene el árbol del archivo a analizar, regresa un puntero a la estructura que contiene el nodo raíz.

Si alguna de las instrucciones anteriores falla o ya no se va a utilizar las estructuras de los nodos y el árbol se deberá liberar la memoria que estos utilizan, para esto se tiene la siguiente instrucción que no tiene valor de regreso y solamente necesita el puntero a la estructura del árbol del archivo analizado.

```
void xmlFreeDoc(xmlDocPtr doc);
```

Se puede usar un espacio de nombre utilizando estas bibliotecas de tal forma que si se lee una marca que no esté dentro de este espacio. Esta podrá ser detectada; esto puede llegar a ser útil para poner una referencia al conjunto de marcas que contiene el archivo; por ejemplo, en el archivo XML que se tiene de ejemplo, si se quiere que las marcas utilizadas sean miembros del conjunto de marcas "Dibujo" entonces estas deberán estar precedidas por el nombre del conjunto.

```
<?xml version="1.0"?>
<Dibujo:Helping xmlns:Dibujo="http://www.gnome.org/some-location">
<Dibujo:dibujo ancho="100" alto="100">
<Dibujo:color fondo="CYAN"/>
...
</Dibujo:dibujo>
```

De esta forma se debe indicar el conjunto al que pertenece cada una de las marcas. Se puede observar que se ha agregado una marca, ésta es la referencia a la localidad donde se encuentra el espacio de nombre válido, esta se obtiene con la siguiente instrucción.

```
xmlNsPtr xmlSearchNsByHref (xmlDocPtr doc,
                            xmlNodePtr nodo,
                            xmlChar *URI);
```

Donde doc es el puntero a la estructura que contiene el árbol del documento a analizar, nodo es el apuntador a la estructura de la marca que contiene la referencia de donde se encuentra el espacio de nombre, este debe ser Helping, y URI es la referencia de dónde está el espacio de nombre.

Una estructura que si es necesario conocer (al menos algunos de sus campos) es xmlNode, ya que en el programa se necesitará utilizar sus elementos para hacer referencias a otros nodos o atributos de él mismo.

```
struct xmlNode {
        xmlNodePtr *parent;  /* Es el puntero al nodo padre */
        xmlNodePtr *next;   /* Es el puntero del siguiente nodo con el mismo
                               padre */
        xmlNodePtr *prev;   /* Es el puntero del nodo anterior con el mismo
                               padre */
```

TESIS CON
FALLA DE ORIGEN

```
        xmlNodeptr *xmlChildrenNode   /* Es el puntero del primer nodo hijo */
        xmlAttrPtr *atributos;  /* Es el puntero a la estructura que contiene los
                                          atributos del nodo */
        xmlChar name;  /* Nombre del nodo o marca */
        xmlNsPtr ns;  /* Apuntador a la estructura xmlNs correspondiente */
}
```

Estos son algunos de los campos de la estructura, ya que contiene otros que no son importantes para el objetivo del trabajo. Los que más se utilizan dentro del programa son "next", "xmlChildrenNode" y "name" ya que se va recorriendo el árbol en orden y no hay necesidad de regresar a otro nodo por la forma en que este estructurado el programa. Para obtener los atributos del nodo no es necesario utilizar el apuntador a la estructura xmlAttr, sino que se pueden obtener utilizando la siguiente instrucción.

```
xmlChar* xmlGetProp (xmlNodePtr puntero, xmlChar *atributo);
```

Esta función recibe 2 parámetros, el primero es un puntero al nodo del cual se quiere el atributo, el segundo es una cadena de caracteres de tipo xmlChar que contiene el nombre del atributo, su valor es regresado como una cadena de tipo xmlChar. Ahora, para obtener el contenido que está entre las marcas de inicio y final del nodo en cuestión se utiliza la siguiente instrucción.

```
xmlChar* xmlNodeListGetString (xmlDocPtr doc, xmlNodePtr nodo, 1);
```

Esta función recibe 3 argumentos, el primero es el puntero al árbol que se realizó del archivo a utilizar, el segundo es un puntero al nodo del cual se requiere el contenido, este valor se regresa como una cadena de tipo xmlChar.

Una instrucción que es muy utilizada, es la de comparación de cadenas, esta es igual a la de C con la diferencia de que en lugar de trabajar con cadenas de tipo char, maneja cadenas de tipo xmlChar.

```
int xmlStrcmp(xmlChar cad1, xmlChar cad2);
```

Si regresa un 0, las cadenas son iguales, en caso contrario las cadenas son distintas. También se tiene una instrucción como la anterior con la diferencia de que se toma una subcadena por cada una de las cadenas del tamaño, indicado por n y estas son las que se comparan, los valores devueltos son los mismos que la instrucción anterior.

```
int xmlStrncmp (xmlChar cad1, xmlChar cad2, int n);
```

De esta forma tenemos otras instrucciones de manejo de cadenas que tiene el lenguaje C, pero con cadenas de tipo xmlChar; por ejemplo, para concatenar cadenas se tiene.

```
xmlChar* xmlStrcat (xmlChar cad1, xmlChar cad2);
```

De la misma manera se tiene la instrucción para concatenar solo un pedazo de las cadenas.

TESIS CON
FALLA DE ORIGEN

```
xmlChar* xmlStrncat (xmlChar cad1, xmlChar cad2, int n);
```

Para obtener la longitud de una cadena.

```
int xmlStrlen (xmlChar cad);
```

O para obtener una subcadena de otra.

```
xmlChar* xmlStrsub (xmlChar cad, int inicio, int final);
```

Los enteros indican los límites de dónde se tomará la subcadena.

Al manejar espacio de nombre se necesita poner el valor por omisión de los nodos en blanco para posteriormente poder detectarlos. Para poner el valor se utiliza la siguiente instrucción.

```
int xmlKeepBlanksDefault (int n);
```

El entero que tiene como parámetro puede ser un 0 o un 1, el entero que regresa también es un 0, lo que significa que no pudo sustituirlo y un 1 si la sustitución fue exitosa. Para detectar estos nodos en blanco se utiliza la siguiente instrucción.

```
int xmlIsBlankNode (xmlNodePtr nodo);
```

El parámetro nodo es el apuntador al nodo en cuestión, esta regresa un 0 si es un nodo blanco y un 1 si es nodo de texto.

Como se mencionó anteriormente, éstas solo son un subconjunto pequeño de todas las instrucciones que contienen éstas.

2.1.3. Programa de ejemplo.

A continuación se tiene el código de un programa que realiza el parcer, la revisión del archivo y la obtención de los datos desde éste, utilizando las bibliotecas que se acaban de analizar. El archivo a revisar tiene que estar escrito conforme al formato que se utilizó para ejemplificar un archivo XML, por lo que ese ejemplo puede ser revisado con este programa.

Dentro del cuerpo del programa se incluyen los comentarios necesarios en el conjunto de instrucciones que lo amerite.

TESIS CON
FALLA DE ORIGEN

```
#include <stdio.h>
#include <stdlib.h>
#include <libxml/parcer.h>
#include <libxml/xmlmemory.h>

/* Se define la estructura que contendrá las propiedades de los
objetos */
typedef struct prop{
     xmlChar color;
     xmlChar mover;
}prop, *propPtr;

/* Esta función obtiene los valores de las marcas que contienen
los objetos (elipse, rect, línea) */
static propPtr
parseProp(xmlDocPtr doc, xmlNodePtr nodo) {
 propPtr ret = NULL;

/* Reserva memoria para la estructura y revisa si esta operación
tubo éxito */
 ret = (propPtr) malloc(sizeof (prop));
 if (ret == NULL) {
  fprintf(stderr,"No hay memoria suficiente\n");
  return (NULL);
 }
 memset (ret, 0, sizeof(prop));

/* Se pasa a los nodos hijos del objeto en cuestión para obtener
sus valores */
 nodo = nodo->xmlChildrenNode;
/* Se pone un ciclo para revisar todas las marcas que contenga
el objeto, cuando ya no tenga hijo nodo valdrá NULL */
 while(nodo ¡= NULL) {
  if (!xmlStrcmp(nodo->name, (const xmlChar *) "color")) {
/* Obtiene el parámetro color de la marca color, si no contiene
este revisa si tiene su parametro RGB */
   ret->color=xmlGetProp(nodo,(const xmlChar *) "color");
   if (ret->color == NULL)
    ret->color=xmlGetProp(nodo,(const xmlChar *) "RGB");
  }
/* Obtiene el contenido de la marca mover */
  if (¡xmlStrcmp(nodo->name, (const xmlChar *) "mover"))
   ret->mover=xmlNodeListGetString(doc,
              nodo->xmlChildrenNode, 1);
/* Pasa al siguiente nodo */
  nodo = nodo->next;
 }
}
```

TESIS CON
FALLA DE ORIGEN

```
/* Se definen las estructuras que contendrán los datos de los
objetos, se define una estructura por cada uno ya que manejan
datos distintos entre si */
typedef struct elipse{
     int radiox;
     int radioy;
     propPtr props[10];
}elipse, *elipsePtr;

typedef struct rect{
     int ancho;
     int alto;
     propPtr props[10];
}rect, *rectPtr;

typedef struct linea{
     int x1;
     int x2;
     int x3;
     int x4;
     propPtr props[10];
}linea, *lineaPtr;

/* La siguiente función obtiene los datos de las marcas que sean
elipses. */
static elipsePtr
parseElipse (xmlDocPtr doc, xmlNodePtr nodo) {
 elipsePtr ret =NULL;

/* Reserva memoria para la estructura elipse */
 ret = (elipsePtr) malloc(sizeof(elipse));
 if(ret == NULL) {
  fprintf(stderr,"Memoria insuficiente\n");
  return (NULL);
 }
 memset(ret, 0, sizeof(elipse));

/* Obtiene los parámetros de la marca elipse y los guarda en la
variable que corresponde dentro de la estructura */
 ret->radiox = xmlGetProp(nodo,(const xmlChar *) "radiox");
 ret->radioy = xmlGetProp(nodo,(const xmlChar *) "radioy");
/* Manda llamar la función que hará el parser del contenido de
esta marca */
 parseProp(doc, nodo);

 return(ret);
}
```

TESIS CON
FALLA DE ORIGEN

```
/* Para el parser de los otros objetos, la función es muy
parecida, lo que cambia son los datos que obtiene y la
estructura que maneja, estas funciones no se presentan en este
ejemplo ya que sería muy repetitivo, pero sí es necesario que se
incluyan en el programa. */

/* La siguiente estructura contiene todos los objetos que se
manejan dentro del dibujo */
typedef struct Dibujo {
     int ancho;
     int alto;
     xmlChar *color;
     int nelipses;   /* Numero de elipses que contiene */
     elipsePtr elipses[200];    /* Datos de las elipses */
     int nrects; /* Numero de rectángulos que contiene */
     rectPtr rects[200]; /* Datos de los rectángulos */
     int nlineas; /* Numero de líneas que contiene */
     lineaPtr lineas[200]; /* Datos de las líneas */
}Dibujo, *DibujoPtr;

/* La siguiente función realizará el parcer del dibujo completo
*/

static DibujoPtr
parseDibujo (char *archivo) {
 xmlDocPtr doc;
 DibujoPtr ret;
/* Se pone una estructura de cada tipo de objeto para almacenar
temporalmente los datos de los objetos obtenidos en las
funciones, estos datos posteriormente se vaciarán en su variable
correspondiente */
 elipsePtr E;
 rectPtr R;
 lineaPtr L;
 xmlNodePtr nodo;

/* Se construye el árbol del documento y se verifican los
errores. El árbol se guarda en doc */
 doc = xmlParseFile (archivo);
 if (doc ==NULL)
  return (NULL);

/* Se obtiene el nodo raiz */
 nodo = xmlDocGetRootElement(doc);
 if (nodo == NULL) {
  fprintf(stderr, "Documento vacio\n");
  xmlFreeDoc(doc);
  return (NULL);
 }
```

TESIS CON
FALLA DE ORIGEN

```
/* Se reserva memoria para el dibujo */
 ret = (DibujoPtr) malloc(sizeof(Dibujo));
 if(ret == NULL) {
  fprintf(stderr,"Memoria insuficiente\n");
  xmlFreeDoc(doc);
  return (NULL);
 }
 memset(ret, 0, sizeof(Dibujo));

/*Obtiene el ancho y el alto del dibujo */
 ret->ancho = xmlGetProp(nodo, (const xmlChar *) "ancho");
 ret->alto = xmlGetProp(nodo, (const xmlChar *) "alto");
 if((ret->ancho == NULL) || (ret->alto == NULL)) {
  fprintf(stderr,"Faltan las medidas del dibujo\n");
  xmlFreeDoc(doc);
  return (NULL);
 }

/* Se pasa al siguiente nivel del árbol para detectar todos los
objetos */
 nodo = nodo->xmlChildrenNode;
 while(nodo ¡= NULL) {
/*Revisa si el nodo actual es color y obtiene su parámetro de
fondo */
  if(¡xmlStrcmp(nodo->name, (const xmlChar *) "color")) {
   ret->color = xmlGetProp(nodo,(const xmlChar *) "fondo");
  }
/*Revisa si el nodo actual es un objeto y cual es, agregandolo y
mandando a su función de parser correspondiente */
  if(¡xmlStrcmp(nodo->name, (const xmlChar *) "elipse")) {
   E = parseElipse(doc, nodo);
   if (E ¡= NULL)
    ret->elipses[ret->nelipses++] = E;
   if (ret->nelipses >= 200)
    break;
  }
  if(¡xmlStrcmp(nodo->name, (const xmlChar *) "rect")) {
   R = parseRect(doc, nodo);
   if (R ¡= NULL)
    ret->rects[ret->nrects++] = R;
   if (ret->nrects >= 200)
    break;
  }
  if(¡xmlStrcmp(nodo->name, (const xmlChar *) "linea")) {
   L = parseLinea(doc, nodo);
   if (L ¡= NULL)
    ret->lineas[ret->nlineas++] = L;
   if (ret->nlineas >= 200)
```

TESIS CON FALLA DE ORIGEN

```
      break;
    }
 /* Pasa al siguiente nodo */
   nodo = nodo->next;
  }
 }

 int main(int argc, char **argv)
 {
  DibujoPtr dib;

  if(argc == 1)
   dib = parseDibujo(argv[1]);
  else
   fprintf(stderr,"Falta el archivo");

  xmlCleanupParcer();

  ...

  return (0);
 }
```

TESIS CON
FALLA DE ORIGEN

Se puede observar que este programa solamente revisa y obtiene los datos de los objetos, lo que después realice el programa con los datos obtenidos ya es responsabilidad del programador y la naturaleza del programa (por ejemplo, en este programa lo que seguiría es dibujar los objetos con los datos obtenidos).

Al final, los datos son almacenados dentro de la estructura dib y es ésta la que se utiliza para manejar los datos, no es estrictamente necesario que los datos se almacenen en estructuras, pero es más cómodo y una mejor forma de almacenarlos.

2.2 OpenGL

Para poder representar los objetos, Material3D utiliza las bibliotecas gráfica de OpenGL[3] las cuales se revisan a continuación.

TESIS CON
FALLA DE ORIGEN

[3] www.opengl.org

2.2.1 ¿Qué es OpenGL?

OpenGL es un conjunto de bibliotecas creadas para el manejo de los gráficos, estas pueden ser utilizadas en conjunto con lenguajes como el lenguaje C o C++ para obtener programas interactivos.

Estas bibliotecas contienen un conjunto de instrucciones básicas pero muy útiles ya que nos ahorran el trabajo de estar realizando varios métodos o cálculos, disminuyendo el código fuente y en ocasiones el tiempo de procesamiento.

Hay que notar que este conjunto de instrucciones solamente maneja gráficos o relaciones entre ellos, por lo que es importante utilizar un lenguaje para poder hacer la inicialización de estas bibliotecas, aparte necesita una interfaz gráfica de usuario (no importando que sea la más simple), por lo cual se utilizan otras bibliotecas para realizar esta, por lo general se utilizan unas llamadas GLUT[4], pero como son muy básicas, Material3D utiliza GTK[5] que están más completas.

Las bibliotecas de OpenGL fueron diseñadas por Silicon Graphics[6] los cuales buscan que estas sean el estándar utilizado para el manejo de los gráficos en 3D para todas las plataformas. Actualmente se está logrando el objetivo, ya que un programa que las utilice funciona en varias plataformas. Para conseguir que un programa con OpenGL se pueda ejecutar en distintas plataformas obteniendo los mismos resultados, las bibliotecas tienen definidos sus propios tipos de datos, aunque estos tienen el mismo comportamiento que los del lenguaje C. La diferencia radica en el número de bits que ocupa para cada tipo de dato. Mientras que con el lenguaje C depende de la plataforma, con las definiciones de OpenGL siempre se utiliza el mismo número de bits para representar ese tipo de dato sin importar la plataforma. Estos tipos de datos se pueden observa en el cuadro 2.1.

Sufijo	Tipo de dato	Típicamente en C	Definición en OpenGL
b	Entero 8-bits	Signed char	GLbyte
s	Entero 16-bits	Short	GLshort
i	Entero 32-bits	Int o long	GLint, GLsizei
f	Flotante 32-bits	Float	Glfloat, GLclampf
d	Flotante 64-bits	Double	GLdouble, GLclampd
ub	Entero sin signo 8-bits	Unsigned char	GLubyte, GLboolean
us	Entero sin signo 64-bits	Unsigned short	GLushort
ui	Entero sin signo 32-bits	Unsigned int o unsigned long	GLuint, GLenum, GLbitfield

Cuadro 2.1. Tipos de datos de OpenGL.

[4] Utilerias de OpenGL (GL Utilities Toolkit)
[5] www.gtk.org
[6] www.sgi.com

TESIS CON FALLA DE ORIGEN

2.2.2 Vértices y transformaciones

OpenGL no tiene definida ninguna primitiva gráfica, este solo trabaja con vértices, los cuales son los que forman los objetos y su salida es hacia el monitor, no hay instrucciones dentro de estas bibliotecas que manden su salida hacia otra parte (por ejemplo un archivo de imagen), para lograr esto se debe utilizar el lenguaje con el cual se esté combinando.

Ya que OpenGL trabaja solo con vértices para definir los objetos, da la oportunidad de realizar una gran variedad de estos y si es posible definir primitivas gráficas propias, solamente se tiene que definir la posición de los vértices de la primitiva y la conexión entre ellos para que OpenGL se encargue de unirlos como se le halla indicado y así formar el polígono, esta es la razón por la cual Material3D maneja los objetos como polígonos teniendo de esta manera una relación directa entre los datos que ingresan desde el formato y las instrucciones que los utilizan sin tener que aplicarles un proceso intermedio.

Hay varias formas de declarar los vértices (y otras instrucciones). Lo que siempre llevan es glVertex seguido del número de argumentos los cuales pueden ser 2 para 2D o 3 para 3D. Por último se pone el tipo de datos que serán los argumentos, estos se pueden ver en el cuadro 2.1 (esto último también se aplica a las otras instrucciones que lo requieran). Los argumentos se ponen enseguida entre paréntesis. Por ejemplo, la siguiente línea describe un vértice en 3D donde los datos son flotantes y el vértice está en el origen.

```
glVertex{1,2,3}{s,i,f,d}(0.0, 0.0, 0.0);
```

Los datos del vértice también pueden ser representados dentro de un arreglo, esto se indica agregando una "v" a la instrucción y entre los paréntesis el arreglo, quedando como sigue:

```
float arr[3] = {0.0, 0.0, 0.0};
...
glVertex{1,2,3}{s,i,f,d}v(arr);
```

Dos instrucciones muy importantes para poder definir un vértice son las siguientes:

```
glBegin(TIPO_DE_CONEXIÓN);
...
glEnd();
```

Dentro de estas, deben estar las instrucciones de los vértices, el número de vértices que deben estar contenidas entre estas instrucciones depende del tipo de conexión que se elija, ya que para conectar los vértices tenemos varias opciones y cada una de estas necesita un número particular de vértices, estas opciones se muestran en la figura2.1.

Por ejemplo, si queremos hacer un cuadrado podemos utilizar el siguiente fragmento de código para realizarlo.

```
void cuadrado(void)
{
 glBegin(LINE_LOOP);
        glVertex2f(0.0, 0.0);
        glVertex2f(1.0, 0.0);
        glVertex2f(1.0, 1.0);
        glVertex2f(0.0, 1.0);
 glEnd();
}
```

TESIS CON
FALLA DE ORIGEN

Note que la definición de los vértices del cuadrado está en orden en dirección contraria a las manecillas del reloj, lo de la dirección se ve más adelante en iluminación, pero es importante resaltar el orden que siguen, ya que si invirtiéramos el orden del segundo y el tercer vértice, resultaría un polígono que no tendrá la forma de un cuadrado ya que el orden en que se definen estos dentro del código define el orden en que van a estar conectados; por lo que esto afecta en gran medida al modelo.

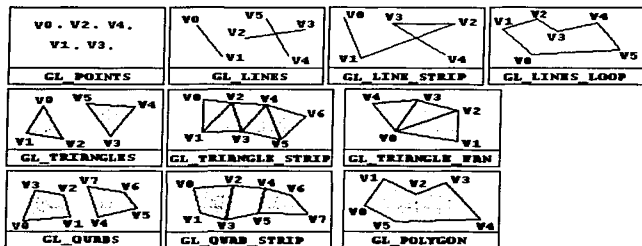


Figura 2.1. Tipos de conexiones de los vertices.

El resultado de poner la definición de los vértices en un orden inadecuado lleva a polígonos los cuales no pueden ser manipulados correctamente con las instrucciones de OpenGL, por lo que se recomienda tratar de no realizar este tipo de polígonos.

Ya conociendo la forma como se definen los polígonos, ahora se necesita saber como aplicarles transformaciones, estas son la rotación, la translación y el escalamiento.

Para poder aplicar estas transformaciones debemos saber como las maneja OpenGL por que de esta manera sabremos como utilizarlas y acomodarlas para lograr el efecto que se busca. OpenGL realiza estos cálculos por medio de las matrices de transformación.

TESIS CON
FALLA DE ORIGEN

Estas matrices las va acomodando en una pila donde la primer matriz que entra es la matriz identidad (la cual se carga con la instrucción glLoadIdentity), esto se debe tener muy presente ya que el orden en que se ponga cada transformación, es el orden en que van a estar acomodadas dentro de la pila y en este caso el orden si altera el resultado, por ejemplo si primero se aplica una rotación a un objeto y posteriormente una translación este estará girando en su propio eje en el lugar a donde ha sido trasladado, en cambio si las transformaciones se aplican en orden inverso, el objeto girará alrededor de su punto original donde el radio de la órbita que sigue está dado por la distancia que ha sido trasladado.

Las instrucciones de transformación se deben poner antes de que se definan los vértices a los cuales se les van a aplicar las transformaciones, es decir, antes de la instrucción glBegin, pero también deben estar después de haber ingresado otra matriz a la pila, esto es después de una instrucción glPushMatrix(). Esta instrucción lo que señala es que se va a agregar una nueva matriz en la pila donde se realicen las nuevas transformaciones, esta nueva matriz tiene los mismos valores que su antecesora, pero los cambios realizados por las nuevas transformaciones solamente se verán reflejados en la matriz que está en el tope de la pila.
Cada vez que se ingresa una matriz a la pila posteriormente se debe remover, ya que si no se quitan las transformaciones que se apliquen, después tomarán en cuenta los valores que guarde la matriz que no se quitó, con esto los resultados variarán dándonos resultados no deseados. Aun cuando ya no se pongan más transformaciones sus matrices debe ser removidas hasta llegar a la matriz identidad (es decir que por cada glPushMatrix() debe existir su correspondiente glPopMatrix()), ya que los entornos de ventanas bajo los que trabaja OpenGL para hacer el manejo de sus elementos, utilizan un ciclo infinito con lo que se llama varias veces a la función de despliegue y si no se sacaron las matrices de la pila hasta llegar a la matriz identidad entonces se empezará con una matriz distinta, la cual tendrá algunas transformaciones ya aplicadas y estas se irán acumulando; por lo que se verán resultados no deseados y en constante movimiento.

Ahora que ya se conoce como se realiza el manejo de las transformaciones dentro de OpenGL se puede ver cuales son sus instrucciones y como se utilizan.

Primero para el escalamiento la instrucción utilizada es:

```
glScale{f,d}( x, y, z);
```

Donde x, y, z representan la proporción en que va a ser escalado el objeto con relación al eje correspondiente; por ejemplo, si se requiere escalar al doble sobre el eje x entonces en su respectivo lugar se debe poner 2.0, hay que notar que si se desea mantener el tamaño original del objeto sobre un eje en particular, solo hay que poner un 1.0 ya que estos números son por los que se va a multiplicar el objeto. Algo importante que se debe tener en cuenta es que no debemos ingresar un valor de 0.0 ya que esto causa que el objeto desaparezca sobre uno de los ejes.

TESIS CON
FALLA DE ORIGEN

Para la traslación la instrucción utilizada es:

```
glTranslate{f,d}( x, y, z);
```

De la misma manera que el escalamiento, en la translación los valores de x, y, z son las unidades que se trasladara al polígono con respecto al eje correspondiente.

La instrucción de la rotación es distinta a las anteriores ya que necesita 4 valores en lugar de 3 como las anteriores, su instrucción es:

```
glRotate{f,d}( ang, x, y, z);
```

El valor de ang son los grados que rotará el polígono y los valores de x, y, z son el número que va a ser multiplicado por el ángulo para ver cuánto finalmente se va a rotar con respecto al eje correspondiente. Es importante notar que OpenGL trabaja los ángulos dados por ang en grados, con esto se conoce qué datos son necesarios para aplicar las transformaciones sirviendo de apoyo para solicitarlos dentro del formato, la forma en que se aplican internamente no es de gran interés para el formato ya que Material3D se encarga de acomodar los datos en las instrucciones de tal manera que concuerden con la escena que describe el archivo.

2.2.3 Luces

TESIS CON
FALLA DE ORIGEN

Antes que nada debemos ver cómo podemos definir el color que va a tener cada polígono, para esto necesitamos un modelo de color con el cual se trabaja para poder poner color a los objetos.

Figura 2.2. Modelo RGB.

El modelo más común utilizado en las computadoras es el RGB, el cual utiliza como colores primarios el rojo (Red), el verde (Green) y el azul (Blue), estos los mezcla para poder obtener los distintos colores ya que dependiendo de la intensidad de cada uno de estos, será el color que forme. Los paquetes de dibujo y los procesadores de imágenes tienen la opción de cambiar el color de sus pinceles o plumas, para ello utilizan interfaces como la que se muestra en la figura 2.2 (tomada de la aplicación "Paint" de Microsoft[7]) o el cubo de colores, cualquiera de los dos muestra la relación que tienen el rojo, el verde y el azul para formar los otros colores.

[7] www.microsoft.com

Este modelo es aditivo, por lo cual cuando los tres colores se encuentran con una intensidad de 0 el color será negro y cuando estén presentes con una intensidad de 1 el color será blanco.

Es importante conocer cual modelo de color tiene OpenGL para saber que datos son necesarios y de que forma los interpreta ya que no solamente existe el modelo RGB, aun así éstas bibliotecas manejan éste modelo añadiendo color a los objetos colocando la siguiente instrucción antes de la definición de los objetos que se quieran con dicho color.

```
glColor3{b, s, i, f, d, ub, us, ui}(R, G, B);
```

Donde R, G y B son las intensidades respectivamente de los colores del modelo. Por lo regular la instrucción se utiliza como glColor3f(R, G, B); donde las intensidades van de 0 a 1. También se puede manejar un modelo ampliado del RGB el cual es el RGBA, este modelo simplemente añade el canal Alpha del color, el cual le da la transparencia al objeto, su instrucción es la misma solo que en lugar del 3, lógicamente estará un 4, este último valor también va de 0 a 1 donde 0 significa que es totalmente sólido (no tiene transparencia) y 1 significa una transparencia total (la superficie no se ve).

Al definir los polígonos solamente podremos ver las líneas que los unen, por lo que se debe calcular cada una de las normales de los polígonos que componen al objeto para que de esta manera se puedan ver las caras del objeto definidas por estos polígonos y no solamente las líneas que los unen.

El cálculo de las normales no es más que encontrar el vector normal a cada una de las caras, por lo que se necesitan 3 de los vértices que la definen, estos deben ser consecutivos, es decir que al menos 2 vértices estén conectados a uno en común, por lo que se descarta la posibilidad de tomar más de una vez a cada vértice.

A continuación se muestra como calcular la normal de una cara tomando los vértices V1, V2 y V3 con sus respectivas coordenadas x, y, z, como vértices consecutivos que definen a este polígono, con éstos se realizan los siguientes cálculos:

```
ax = x1 - x2    bx = x3 - x2
ay = y1 - y2    by = y3 - y2
az = z1 - z2    bz = z3 - z2

x = (ay*bz) - (az*by)
y = (az*bx) - (ax*bz)
z = (ax*by) - (ay*bx)
```

$tam = \sqrt{(x^2 + y^2 + z^2)}$

```
nx = x / tam
ny = y / tam
nz = z / tam
```

TESIS CON
FALLA DE ORIGEN

Donde nx, ny, nz contienen los valores del vector normal, estos deben ser ingresados por medio de la instrucción:

glNormal3{b,s,i,f,d} (nx, ny, nz);

Esta instrucción debe estar situada justo antes de ingresar los valores de los vértices que definen la cara en cuestión, es decir antes de glVertex. Se puede definir una normal por cada vértice que tendrá los mismos valores para los vértices de una misma cara, ya que se calculan tomando en cuenta la orientación de la cara y no los vértices en particular, por lo que es más común definir una sola normal para cada una de las caras del objeto.

Para poder iluminar las caras de un objeto primero se debe definir sus normales, también hay que tomar en cuenta el orden en que se definieron los vértices del polígono para poder obtener una iluminación correcta, estos deben estar definidos en orden antihorario ya que si se definen en sentido de las manecillas del reloj, la cara interior y la cara exterior del polígono estarán invertidas con lo que la iluminación del objeto se verá extraña ya que no es la misma intensidad de luz la que reciben estas caras.

Aunque al momento de diseñar el formato Material3D no contaba con el manejo de luces (se tiene definida una luz por omisión que ilumina toda la escena) no quiere decir que no se piense implantar esta característica, por lo tanto es importante ver a futuro y analizar los datos que se requieren para su manejo, de esta manera se puede ir proponiendo su estructura dentro del formato. OpenGL utiliza la siguiente instrucción para el manejo de las luces:

glLight{i,f}[v](num, opción, valores opción);

Opción	Valor por omisión
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0) o (0.0, 0.0, 0.0, 1.0)
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0) o (0.0, 0.0, 0.0, 1.0)
GL_POSITION	(0.0, 0.0, 1.0, 0.0)
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)
GL_SPOT_EXPONENT	0.0
GL_SPOT_CUTOFF	180.0
GL_CONSTANT_ATTENUATION	1.0
GL_LINEAR_ATTENUATION	0.0
GL_QUADRATIC_ATTENUATION	0.0

Cuadro 2.2. Opciones de iluminación.

TESIS CON FALLA DE ORIGEN

Esta instrucción contiene varias opciones y por lo general se utilizan más de una para definir una sola luz, estas se muestran en el cuadro 2.2. Para saber a qué luz se refiere cada una de las opciones que se activan, se tiene el valor de num el cual indica el

número de luz a la cual se está haciendo referencia, este debe ser un entero y si se utiliza una variable para este campo, entonces deberá ser de tipo Glenum. Como se puede observar con la instrucción, los valores de la opción utilizada pueden estar en un vector o ingresarse directamente, la definición y el uso de estos valores dependerá de la opción para la cual se estén utilizando.

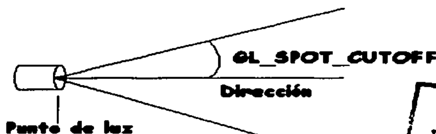
A continuación se explica que efecto tiene cada una de las opciones sobre la luz a la que se aplica.

GL_AMBIENT contribuye a la iluminación del ambiente sobre el escenario, sus valores utilizan el modelo RGBA.

GL_DIFFUSE define una luz que llega de una dirección donde su parte más brillante es a lo largo de la normal esparciéndose en todas direcciones al chocar con los objetos. Al igual que el anterior sus valores son RGBA.

GL_SPECULAR es la luz que rebota en los objetos hacia una cierta dirección, esta es utilizada para calcular el brillo de los objetos. De la misma manera que las anteriores, sus valores están definidos por el modelo RGBA con lo cual en los 3 definen la intensidad que tiene cada atributo de la luz.

GL_POSITION define la posición de la luz con una coordenada homogénea donde los primeros tres valores dan la posición en X, Y y Z respectivamente, el último parámetro llamador W indica si la fuente de luz es infinita (que la luz no ilumina igual en toda la escena) con el valor de 0.0 o es local (que solamente ilumina hasta llegar a cierta distancia) con el valor de 1.0.
GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION y GL_QUADRATIC_ATTENUATION indican que la luz se va atenuando hasta que desaparece, el efecto de atenuación depende de la opción que se esté ocupando.



TESIS CON
FALLA DE ORIGEN

Figura 2.3. Luz de spot

Se tiene otro tipo de luz, la cual es la luz de SPOT y a la cual le pertenecen las opciones restantes. Esta se parece mucho a una lámpara de mano ya que define un punto de emisión con su respectiva dirección, un ángulo el cual es el área que va a iluminar y una atenuación como se puede observar en la figura 2.3. Es importante señalar que para este tipo de luz el valor de W de la opción GL_POSITION debe estar en 1.0.

GL_SPOT_DIRECTION indica la dirección hacia donde va a iluminar la luz, sus parámetros indican la posición en los ejes X, Y y Z respectivamente.

GL_SPOT_EXPONENT es la atenuación de la luz hacia el contorno del cono ya que la luz en el centro es más intensa que en la orillas (como en una lampara). El valor 0.0 indica que la luz es la misma en todo el cono, el valor de 1.0 indica que la intensidad en el contorno del cono va a ser el coseno del ángulo con que se abre el cono.

GL_SPOT_CUTOFF especifica la amplitud del cono, este es el ángulo de cutoff que se puede observar en la figura 2.3. Los valores permitidos para esta opción son de 0.0 a 90.0 de lo contrario se apagarán los efectos de esta luz.

Ya solamente se necesita habilitar las luces, para esto se usa la siguiente instrucción que sirve para habilitar algunos cálculos que necesite dentro del programa y no se encuentren por omisión.

```
glEnable(Opción);
```

La opción es de tipo Glenum y esta indica el módulo que se está habilitando, por ejemplo para que nosotros podamos habilitar la luz, nuestra opción debe ser GL_LIGHTING. Esta instrucción se implementó para poder reducir el número de cálculos a realizar ya que puede haber programas muy simples que no utilicen todos los cálculos que puede realizar OpenGL y como estos cálculos ocupan mucho tiempo de procesamiento hacen más lento al programa y con esta instrucción solamente se ocuparán los módulos que se necesiten.

Al habilitar la luz sus cálculos se tendrán en cuenta en el resto del programa lo cual puede resultar algo ineficiente por que se pueden tener algunas partes que utilicen estos cálculos pero no los necesiten por cualquier razón como que no le afecte la luz a esa parte del código. Para evitar estos cálculos innecesarios tenemos la siguiente instrucción:

```
glDisable(Opción);
```

Esta deshabilita los módulos que se habilitaron con glEnabe, la opción indica sobre que módulo va a trabajar y son exactamente las mismas para las dos instrucciones.

Después de haber habilitado la iluminación se debe habilitar la luz o luces con que se va a trabajar, para esto también utilizamos glEnable para habilitarla y glDisable para deshabilitarla, la opción que contendrán estas instrucciones será GL_LIGHTn donde la n es el número de la luz con que se está trabajando, este debe coincidir con el número utilizado en glLight para que los atributos definidos ahí se apliquen a esa luz.

En la iluminación de la escena aparte de las luces, el material de los objetos toma un papel importante ya que este define como reacciona el objeto con dicha luz, lo cual le da un cierto efecto dando la impresión de qué está echo con una cierta materia ya sea plástico (donde el objeto será opaco y no reflejará mucho la luz) o un metal (donde la luz se refleja más), este efecto ayuda al color para que los objetos se vean más reales.

TESIS CON
FALLA DE ORIGEN

En OpenGL se utiliza la siguiente instrucción para poder agregar y modificar los efectos del material del objeto, hay que tener en cuenta que este efecto se aplica sobre una cara del polígono.

glMaterial{i,f}[v](cara, opción, valores opción);

Donde cara es el lado del polígono al que se le aplicará el material, esta puede tomar los siguientes valores: GL_FRONT (para la cara de enfrente), GL_BACK (para la posterior) y GL_FRONT_AND_BACK (para ambas). Las opciones se pueden ver en la siguiente cuadro junto con sus valores por omisión.

Opción	Valor por omisión
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)
GL_AMBIENT_AND_DIFFUSE	
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)
GL_SHININES	0.0
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)
GL_COLOR_INDEXES	(0, 1, 1)

Cuadro 2.3. Opciones del material.

TESIS CON FALLA DE ORIGEN

A continuación se explica cada una de estas opciones.

GL_AMBIENT es el color ambiental del material, este se localiza donde la luz da indirectamente sobre el objeto.

GL_DIFFUSE da el color difuso del material, esto es donde la luz da directamente sobro el objeto.

GL_AMBIENT_AND_DIFFUSE afecta a los colores difuso y ambiental de la luz reflejada sobre el objeto.

GL_SPECULAR es el color especular del material, este es el que define el brillo del material y se encuentra en el punto donde la luz afecta más al objeto. Todas las opciones anteriores incluyendo esta utilizan el modelo de color RGBA para determinar sus valores.

GL_SHININES este es la intensidad del brillo en el material sobre la parte especular.

GL_EMISSION es el color que va a emitir el material, utiliza el modelo RGBA para definirlo.

GL_COLOR_INDEXES son los índices de color ambiental, difuso y especular respectivamente, este significa si los cálculos de ese tipo para el material se toman en cuenta (1) o se ignoran y no afectan al material (0).

Los materiales, al contrario de las luces no necesita habilitarse, aunque se puede utilizar la siguiente instrucción:

TESIS CON FALLA DE ORIGEN

glColorMaterial(cara, opción);

Donde los parámetros cara y opción los maneja de la misma forma que glMaterial y los valores de la opción se ponen enseguida de la instrucción glColor. Esta es una forma rápida de cambiar las propiedades de los materiales pero necesita ser habilitada con glEnable donde la opción será: GL_COLOR_MATERIAL.

TESIS CON
FALLA DE ORIGEN

2.2.4 Cámara.

La cámara es un elemento imprescindible en la escena ya que es por donde se ven todos los objetos; tiene distintas formas de proyectar los objetos en la pantalla, dependiendo de la transformación que se utilice será la forma que tomen en el monitor. Al igual que con las otras partes analizadas se debe ver la forma en que se maneja la cámara, aunque ésta depende también de la manera como la tiene implementada el modelador.

Hay dos transformaciones que se pueden aplicar con la proyección de la cámara, una es la ortográfica la cual define un área que parece un cubo el cual puede estar alargado, la cámara está en uno de sus lados y todo lo que esté dentro del cubo es la parte de la escena que se podrá ver (como se ve en la figura2.4), con este tipo de proyección se tiene un efecto con lo que los objetos que están lejos de la cámara no disminuyen de tamaño visualizándose del mismo tamaño que si estuviera enfrente de la cámara. Para poder utilizar este tipo de proyección se tiene la siguiente instrucción:

glOrtho(izquierda, derecha, inferior, superior, cerca, lejos);

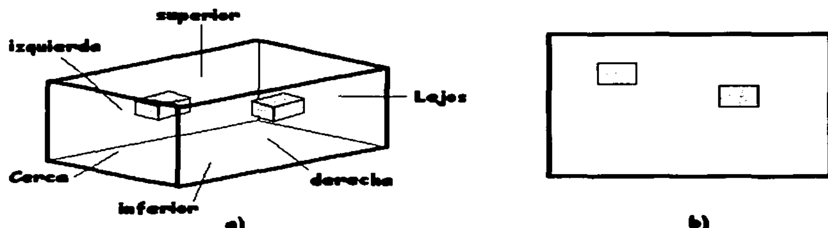


Figura 2.4. a) Diagrama de la vista ortográfica, b) presentación en el monitor de la misma escena.

Como se puede observar, los parámetros que se le pasan a esta instrucción definen los límites del cubo que encierra la escena, todos estos parámetros son de tipo GLdouble.

TESIS CON
FALLA DE ORIGEN

El otro tipo de proyección es la perspectiva, la cual da más el efecto de profundidad ya que los objetos a mayor distancia disminuyen su tamaño. Define un área parecida a una pirámide de base rectangular donde la cámara se encuentra en la punta, ya que desde la cámara se va abriendo la vista hacia el fondo. Para utilizar este tipo de proyección se tienen dos instrucciones, la primera es:

glFrustum(izquierda, derecha, inferior, superior, cerca, lejos);

Los valores de esta instrucción son los mismos que con glOrtho (y de la misma manera son de tipo GLdouble) pero la diferencia radica en que no se define una especie de cubo, sino que primero se define el rectángulo más cercano a la cámara de ahí los cuatro vértices se unen al centro de la cámara definiendo una especie de pirámide que tendrá una profundidad marcada por los valores de cerca y lejos, esto se muestra en la figura 2.5. Los valores de cerca y lejos se toman como la distancia desde la cámara hasta el punto donde queremos situarlos.

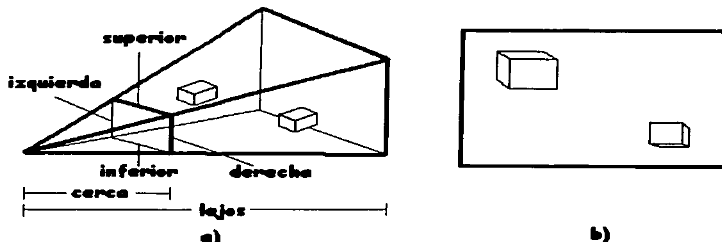


Figura 2.5. a) diagrama de la vista en perspectiva utilizando glFlustrum, b) presentación en el monitor de la misma escena.

La otra instrucción utilizada para este tipo de proyección está definida en las utilerias de OpenGL (glu.h) y es la siguiente:

gluPerspective (fovy, aspecto, cerca, lejos);

Como se ha dicho, esta instrucción también define una especie de pirámide, pero requiere de valores para distinto tipo de parámetros. En fovy se requiere el ángulo para el campo de visión en el eje yz, el aspecto es la proporción del área de visión, esta se calcula dividiendo el ancho entre el alto, cerca y lejos son lo mismo que las instrucciones anteriores.

Ahora solamente falta saber como modelar al observador para que de esta forma se pueda ver la escena desde cualquier ángulo y hacia cualquier parte de esta sin tener que estar aplicando transformaciones a todos los objetos que la componen (ya que aumentaria el tiempo de procesamiento).

TESIS CON
FALLA DE ORIGEN

Para esto se utiliza la siguiente instrucción (también implementada en las utilerias de OpenGL):

```
gluLookAt(ojox, ojoy, ojoz,
          centrox, centroy, centroz,
          arribax, arribay, arribaz);
```

Donde los valores de los ojos definen el lugar donde va a estar el observador, estos valores se definen en coordenadas del mundo. Los valores del centro son el punto hacia donde está mirando el observador y los valores de arriba definen el vector que indica en qué dirección es arriba en la escena, este se toma desde el punto definido como centro hacia el punto que se pone en arriba como se ve en la figura2.6, este no debe ser paralelo a la línea de visión definida por el punto de ojo y el punto de centro.

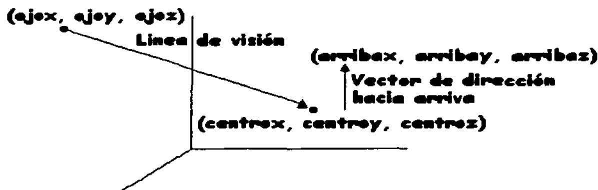


Figura 2.6. Parametros de gluLookAt.

Existen más instrucciones de OpenGL que también son importantes, pero las que aquí se presentan son las básicas y las que ayudan a material3D a visualizar los objetos en su interfaz.

TESIS CON
FALLA DE ORIGEN

CAPÍTULO 3

FORMATOS DE ALGUNOS MODELADORES 3D

TESIS CON
FALLA DE ORIGEN

Todos los modeladores en 3D requieren guardar todos los datos de una escena en un archivo, el cual deberá contener la descripción de todos los objetos que la componen, así como los atributos de todos estos objetos y los efectos aplicados a la escena.

Cada modelador tiene una forma distinta de organizar sus datos para poder guardarlos en un archivo, a esta organización se le llama formato por lo que si conocemos la estructura del formato de un modelador entonces podremos obtener la información que deseemos de la escena y de esta forma poder utilizarla en otra aplicación o podríamos construir un convertidor de formatos para que las escenas que realicemos en un modelador las podamos visualizar en otro.

Esto último es muy importante debido a que un modelador que tenga la opción de importar o exportar escenas con otras aplicaciones (esto se realiza con otros convertidores de formato) tiene un mayor uso (aparte de significar una ventaja) ya que pueden utilizarse sus ventajas sin que tenga que ser ignorado por algunos procesos que no pueda realizar o que no sea tan eficaz al realizar un efecto como otros.

Por estas razones se analizan los formatos de algunos de los modeladores más utilizados, además de ver si podemos tomar como base alguna forma de estructurar los datos de uno de estos modeladores.

3.1. 3Dstdio.

Este es un modelador programado por Autodesk Ltd. el cual es muy utilizado gracias al gran conjunto de efectos y herramientas con las que cuenta. Es un modelador muy potente aunque un poco difícil de utilizar si no se tiene algo de experiencia con él, además de que solo puede ser utilizado bajo ambiente Windows.

Este formato es muy confuso ya que está almacenado en binario, pero sus datos después de leer el archivo en binario son únicamente números los cuales tendrán un significado dependiendo de la localidad en que se encuentren dentro del archivo. Estos números se manejan en hexadecimal debido a que 2 de sus dígitos representan un byte al leer el archivo, por lo que la explicación del formato se realizará tomando en cuenta que solamente se leen números en hexadecimal.

3.1.1. Estructura.

Su formato está estructurado por bloques llamados chunks los cuales están acomodados por niveles aunque en el archivo estén secuencialmente uno tras otro donde el chunk principal es el único que está en el nivel 0 por lo que contiene a los demás chunks.

Los chunks están compuestos por su identificador, su tamaño y los datos que necesite (incluidos los chunks que contenga) como lo muestra el cuadro 3.1, para obtener los números correctamente debemos de ordenarlos ya que están al revés pero por byte. Si tenemos un número de 2 bytes, por ejemplo 3E 12 tenemos que el 12 es la parte alta del número en hexadecimal y el 3E la parte baja con lo que e número queda 12 3E. Para números de 4 bytes debemos hacer algo parecido, por ejemplo con el número 3E 12 45 A7 debemos tomar primero para la parte baja el 3E 12 para hacerle el mismo proceso, con lo que tenemos 12 3E en la parte baja y A7 45 en la parte alta quedando el número como A7 45 12 3E.

Posición	Tamaño (bytes)	Descripción
X	2	Identificador del chunk.
X+2	4	Tamaño = 6 + n + m
X+6	N	Datos.
X+6+n	M	Chunks contenidos.

Cuadro 3.1. Estructura de un Chunk.

En el cuadro, X indica el byte donde inicia el chunk, n es el número de bytes que ocupan sus datos y m el número de bytes que ocupan los chunks que contiene, estos dos últimos datos no son obligatorios. Cada uno de los chunks inicia inmediatamente después de que termina el anterior, pero en el caso de los chunks contenidos dentro de otros, cada uno de los primeros inicia inmediatamente después de que terminan los datos del chunk que los contiene. El único caso especial es el del chunk principal ya que éste inicia en el primer byte del archivo, de esta forma sirve como referencia para iniciar la correcta lectura del archivo.

Tipo	Tamaño (bytes)	Descripción
word	2	Palabra de 2 bytes.
dword	4	Palabra de 4 bytes.
flotat	4	Número de punto flotante.
strz	Variable	Cadena de palabras.
vector	12	3 números flotantes (X, Y, Z).
BOOLEAN	0	Solamente los chunks pueden ser de este tipo, estos son utilizados como banderas.
degree	4	Angulo flotante entre 0 y 360 grados.
rad	4	Angulo flotante entre 0 y 2*Π

Cuadro 3.2. Tipos de datos.

Los datos que contienen los chunks también deben contemplarse en el tamaño del chunk, estos datos pueden ser de distintos tipos por lo que también tendrán distintos tamaños. En el cuadro 3.2 podemos ver los tipos de datos y el tamaño que utilizan.

Se debe tomar en cuenta que no todos los números en hexadecimal pueden ser identificadores de un chunk, por lo que es necesario conocer cuales números sí son identificadores y a cuál chunk se están refiriendo. En el siguiente árbol de chunks podemos observar el nombre de cada uno así como sus identificadores y el nivel donde se encuentran.

TESIS CON
FALLA DE ORIGEN

Chunks de color

- 0x0010 color RGB flotante
- 0x0011 color RGB byte
- 0x0012 corrector de gamma RGB byte
- 0x0013 corrector de gamma RGB flotante

Chunks de porcentaje

- 0x0030 porcentaje entero
- 0x0031 porcentaje flotante

0X4D4D Chunk principal

- 0x0002 versión de 3dstudio utilizada
- 0x3D3D Editor de chunks
  - 0x0100 una unidad
  - 0x1100 mapa de bits de fondo
  - 0x1101 usar mapa de bits de fondo
  - 0x1200 color de fondo
  - 0x1201 usar color de fondo
  - 0x1300 colores de gradiente
  - 0x1301 usar gradiente
  - 0x1400 inclinación del mapa de sombra
  - 0x1420 tamaño del mapa de sombra
  - 0x1450 rango de muestra del mapa de sombra
  - 0x1460 inclinación del trazado de rayos
  - 0x1470 activación del trazado de rayos
  - 0x2100 color ambiental
  - 0x2200 niebla
    - 0x2210 niebla de fondo
  - 0x2201 usar niebla
  - 0x2210 niebla de fondo
  - 0x2300 cola de distancia
    - 0x2310 fondo oscuro
  - 0x2301 usar cola de distancia
  - 0x2302 opciones de niebla por capas
  - 0x2303 usar niebla por capas
  - 0x3D3E versión de la malla
- 0x4000 bloque de objeto
  - 0x4010 ocultar objeto
  - 0x4012 objeto sin modelar
  - 0x4013 material del objeto
  - 0x4015 activación de proceso externo
  - 0x4017 objeto no recibe sombra
  - 0x4100 malla triangular
    - 0x4110 lista de vértices
    - 0x4111 desconocido
    - 0x4120 descripción de caras
      - 0x4130 lista de material de caras
    - 0x4140 lista de coordenadas mapeadas
      - 0x4150 lista de grupo suavizado
    - 0x4160 sistema de coordenadas local
    - 0x4165 color del objeto en editor
    - 0x4181 nombre del proceso externo
    - 0x4182 parámetros del proceso externo
  - 0x4600 luz
    - 0x4610 luz de spot
      - 0x4627 trazado de rayos del spot
      - 0x4630 luz sombreada

TESIS CON
FALLA DE ORIGEN

0x4641 mapa de sombra del spot
0x4650 mostrar cono del spot
0x4651 el spot es rectangular
0x4652 spot lejano
0x4653 mapa del spot
0x4656 giro del spot
0x4658 inclinación de trazado de rayos del spot
0x4620 luz apagada
0x4625 activar la atenuación
0x4659 inicio del rango
0x465A final del rango
0x465B multiplicador
0x4700 cámara
0x7001 ambiente de la ventana
0x7011 descripción de ventana #2
0x7012 descripción de ventana #1
0x7020 ventanas de malla

<u>0xAFFF bloque de material</u>
0xA000 nombre del material
0xA010 color ambiental
0xA020 color difuso
0xA030 color especular
0xA040 porcentaje de brillo
0xA041 porcentaje de brillo intenso
0xA050 porcentaje de transparencia
0xA052 porcentaje de descenso de transparencia
0xA053 porcentaje de reflexión borrosa
0xA081 2 lados
0xA083 añadir transición
0xA084 iluminación propia
0xA085 modelo de alambre activado
0xA087 espesor del alambre
0xA088 mapa de caras
0xA08A en transición
0xA08C ablandado
0xA08E alambre en unidades
0xA100 tipo de render
0xA240 porcentaje descendido sobre la transparencia actual
0xA250 porcentaje de reflexión borrosa actual
0xA252 porcentaje de choque actual
0xA200 mapa de textura 1
0xA33A mapa de textura 2
0xA210 mapa de opacidad
0xA230 mapa de choque
0xA33C mapa de brillo
0xA204 mapa de especularidad
0xA33D mapa de iluminación propia
0xA220 mapa de reflexión
0xA33E mascara para mapa de textura 1
0xA340 mascara para mapa de textura 2
0xA342 mascara para mapa de opacidad
0xA344 mascara para mapa de choque
0xA346 mascara para mapa de brillo
0xA348 mascara para mapa de especularidad
0xA34A mascara para mapa de iluminación propia
0xA34C mascara para mapa de reflexión

TESIS CON
FALLA DE ORIGEN

```
sub-chunks para todos los mapas
        0xA300 nombre del archivo del mapa
        0xA351 parámetros del mapeo
        0xA353 porcentaje de borrosidad
        0xA354 escala V
        0xA356 escala U
        0xA358 compensación en U
        0xA35A compensación en V
        0xA35C ángulo de rotación
        0xA360 tinte RGB de Luma/Alfa 1
        0xA362 tinte RGB de Luma/Alfa 2
        0xA364 tinte RGB rojo
        0xA366 tinte RGB verde
        0xA368 tinte RGB azúl

0xB000 chunk de animación
    0xB001 bloque de información de la luz ambiental
    0xB002 bloque de información de la malla
    0xB003 bloque de información de la cámara
    0xB004 bloque de información del objetivo de la cámara
    0xB005 bloque de información de la luz Omni
    0xB006 bloque de información del objetivo de la luz de Spot
    0xB007 bloque de información de la luz de Spot
    0xB008 Frames (cuadros) (inicio y final)
        0xB010 nombre de objeto parámetros y padre jerárquico
        0xB013 punto de pivote del objeto
        0xB015 ángulo de movimiento del objeto
        0xB020 posición del camino
        0xB021 rotación del camino
        0xB022 escalamiento del camino
        0xB023 FOV del camino
        0xB024 giro del camino
        0xB025 color del camino
        0xB026 movimiento del camino
        0xB027 hotspot del camino
        0xB028 desprendida del camino
        0xB029 camino oculto
        0xB030 posición jerárquica
```

TESIS CON FALLA DE ORIGEN

3.1.2. Chunks.

El árbol presentado ayuda a poder conocer y localizar los distintos chunks descifrados del formato 3ds de una manera rápida, pero es importante conocer que datos manejan y en que orden van, una breve descripción de cada uno de ellos se presenta en el apéndice B.

Se debe tomar en cuenta que los chunks tienen un tamaño mínimo de 6 bytes (2 bytes del identificador y 4 del tamaño), por lo que el chunk si no contiene datos mantendrá ese tamaño lo cual se refleja en dicho apéndice.

Como se puede observar en el árbol de chunks éstos están espaciados de acuerdo al nivel donde pueden ser utilizados, por ejemplo el chunk principal (0x4D4D) es el único que

está en el primer nivel por lo que el resto de los chunks deben ir dentro de él. También se tienen una serie de chunks (los de color y de porcentaje) que pueden ser utilizados en distintos niveles de acuerdo a la necesidad del chunk que los contenga.

La mayoría de los chunks están acomodados en bloques donde los chunks no deben estar definidos fuera del que les corresponde. Estos bloques los definen ciertos chunks los cuales engloban un contexto en especial, por ejemplo el bloque de objeto (0x4000) engloba el conjunto de chunks que definen a los objetos y sus propiedades.

Se puede notar que la serie de chunks sigue una numeración de acuerdo al bloque al que pertenecen, ésta no se traslapa con la de un bloque distinto por lo que cada chunk contiene su propio identificador.

3.2. Formato obj.

Este es un formato creado por Alias Wavefront el cual es utilizado por múltiples modeladores 3D (como Maya, 3Dstdio y Rinho). Este formato (a diferencia del anterior) las instrucciones que maneja para representar los datos son más simples lo que facilita su análisis, puede ser almacenado en ascii (con extensión obj) o en binario (con extensión mod).

El formato tiene un conjunto de palabras reservadas las cuales ayudan a identificar los objetos, sus vértices, sus características, etc., y de esta forma conocer los datos que tiene cada uno.

3.2.1. Objetos.

Para representar objetos se tienen varias formas ya que el formato puede trabajar con polígonos, superficies y curvas de forma libre las cuales pueden ser representadas de distintas maneras con lo que se tiene una amplia variedad de formas que pueden tomar. En el apéndice C se muestra una breve descripción de las instrucciones básicas que maneja el formato.

Para especificar una geometría poligonal los elementos que brinda el formato hacen referencia a vértices declarados en todo el archivo que contiene la escena, la forma de realizar esta referencia hacia los vértices es utilizando su identificador ya que estos se van numerando conforme van apareciendo dentro del archivo donde el primer vértice es el número 1, el segundo es el número 2 y así sucesivamente. La secuencia no se reinicia ya que no importa que los vértices estén separados por otros bloques de elementos o algunos incluso estén dentro de uno de estos bloques, todos los vértices son generales

por lo que llevan la misma secuencia. Hay que destacar que se tiene una secuencia para cada tipo de vértice, es decir que la secuencia de los vértices geométricos es independiente de las otras secuencias (como la de los vectores normales).

También se puede hacer referencia a los vértices con números negativos donde el –1 se refiere al primer vértice que se encuentre arriba de la línea donde se esta haciendo la referencia, el –2 es el segundo vértice que se encuentre arriba de la misma línea y así sucesivamente. Se puede observar que de esta manera los vértices cambian su índice ya que éste depende de la línea donde se realice la referencia.

En los elementos que utilicen varios tipos de vértices para un punto, estos van separados por "/" sin dejar espacios, por ejemplo si "f" utiliza los vértices "v" "vt" y "vn" en ese orden se tendrá.

```
f 1/1/1 2/2/2 ...
```

Se nota que se hace referencia a los vértices por medio de sus índices dados por la secuencia ya mencionada, no es obligatorio que todos los índices utilizados para un mismo punto sean iguales. En algunos elementos se puede prescindir de algunos de los parámetros que pueden utilizar, en estos casos no se pondrá nada en su lugar correspondiente, pero deben estar presentes las diagonales que utiliza el elemento, en el caso que sea el último parámetro podrá prescindirse de ella.

```
f 1//1 2//2 ...
```

Una cosa importante que se debe recordar es que si no se utiliza un tipo de vértice en un punto este mismo vértice no debe ser utilizado en todos los otros puntos, es decir que todos los puntos deben tener los mismos tipos de vértices.

Los elementos que definen curvas o superficies de forma libre pueden tener ciertos parámetros que sirven para obtener más efectos, al utilizarlos el bloque debe ser encerrado entre el cstype y un end como se muestra a continuación.

```
cstype bezier
deg 1 1
surf 0.0 2.0 0.0 2.0 1 2 3 4
parm u 0.00 2.00
parm v 0.00 2.00
trim 0.0 4.0 1
hole 0.0 4.0 2
trim 0.0 4.0 3
hole 0.0 4.0 4
end
```

En este pedazo de código muestra una superficie Bezier con 2 regiones donde cada una contiene un hoyo. Se puede observar el orden que siguen los elementos donde en la primer línea se encuentra el inicio de la superficie con el tipo, en la segunda los grados que utilizará, en la tercera el elemento que define la superficie seguido de los parámetros,

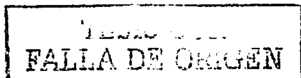
TESIS CON
FALLA DE ORIGEN

hasta el final para cerrar el bloque de esta superficie se encuentra el elemento end. Los parámetros que se pueden utilizar se encuentran descritos en el apéndice C.

3.2.2. Agrupación y conexión.

En los modeladores es común tener un conjunto de objetos que pertenezcan a un caractér o un objeto compuesto por otros, esto se hace definiendo grupos ya que de esta manera podemos aplicarle un cambio a todo un grupo y no tener que realizar ese cambio objeto por objeto. Dentro del agrupamiento puede entrar la conexión de los objetos ya que al declararlos aún estando juntos dentro del archivo no quiere decir que sean parte de un mismo objeto o estén conectados unos con otros.

Este formato también contiene un elemento para realizar la agrupación, el bloque del grupo está definido desde el punto donde se pone el elemento hasta el final del archivo. Un grupo puede contener uno o más grupos donde cada uno puede tener su propio conjunto de elementos incluyendo otros grupos.



CAPÍTULO 4

LENGUAJES DE MODELADO 3D

TESIS CON
FALLA DE ORIGEN

Un lenguaje de modelado en 3D es un conjunto de instrucciones que sirven para describir una escena, éstas las reconoce el programa diseñado para trabajar con su lenguaje en particular y construye la escena definida en un archivo escrito por el usuario el cual contiene algunas de esas instrucciones siguiendo un formato en particular. El lenguaje de modelado más conocido es el VRML, pero existen otros como POV-RAY que es un lenguaje muy poderoso, estos dos lenguajes son los que se describen en este capítulo, aunque no está su descripción completa (ya que contiene bastantes instrucciones) si se encuentran varias de las instrucciones más importantes que debemos conocer para poder modelar con estos lenguajes.

Modelar con un lenguaje de modelado es muy distinto a utilizar un modelador 3D debido a que no se modifican o se mueven los objetos directamente sobre su superficie sino que estos se realizan sobre los datos de los objetos dentro de un archivo, para poder ver la escena se debe contar con el programa indicado que reconozca el formato y muestre o genere una imagen de la escena, el tiempo que arde para generarla depende del lenguaje y del programa que se utilice, además de las características que contenga.

Se puede llegar a pensar que los lenguajes de modelado 3D no son de utilidad al contar con modeladores 3D, pero esto es falso porque al escribir directamente los datos de los objetos en el archivo de la escena su formato debe ser entendible para el usuario además de tener opción de definir un objeto con su conjunto de vértices o ingresando una primitiva indicando únicamente sus características particulares que se quieren de ella, por ejemplo su tamaño o número de divisiones. Esta forma de modelado ayuda a que algunas aplicaciones puedan mostrar sus resultados escribiendo una escena con algunos de estos lenguajes.

Esta última cualidad es algo de lo que se busco para el diseño del formato ya que es de gran utilidad para diversos propósitos que con los formatos de los modeladores 3D es muy difícil de realizar.

Otra de las grandes ventajas de utilizar estos lenguajes de modelado 3D es que con estos mismos podemos programar para poder realizar escenas interactivas o animaciones en las cuales no tengamos que estar haciendo un archivo por cada cuadro que se necesite.

4.1. POV-RAY

Este es un programa que actualmente es gratuito y se puede obtener en Internet, la versión que se utilizó para esta referencia es la 3.1. Como se mencionó este maneja un conjunto de instrucciones donde su compilador (por así llamar al programa que construye la escena con archivos que contengan ese tipo de instrucciones) utiliza el trazado de rayos para dibujar la escena.

Este lenguaje es demasiado extenso ya que maneja una gran cantidad de efectos y características de los objetos. Cuenta con varias primitivas las cuales podemos combinar usando la GSC (descrita en el capítulo 1) para obtener objetos más complicados. Algo importante que debemos tomar en cuenta es que los datos que utilizan las instrucciones son enteros y/o flotantes (los que el usuario ponga).

Este lenguaje es muy parecido al lenguaje C ya que las instrucciones encierran sus características entre llaves ({}) simulando de esta manera funciones, aunque algunos valores dentro de estos bloques son encerrados entre los signos de ">" y "<" pero todos estos siguen cierta jerarquía (como se vio en el primer capítulo). También se le pueden añadir archivos al igual que con el lenguaje C con la instrucción:

```
#include "archivo"
```

POV-RAY tiene unos archivos ya incluidos en el programa (como los ".h" de C), estos tienen extensión ".inc" y contienen algunas definiciones de constantes que se utilizan dentro de una escena. De esta manera podemos definir ciertos bloques que describen en general las escenas que se pueden realizar con esta aplicación:

- Inclusión de archivos: Consiste de los #include y no es necesario que se incluya este bloque dentro de una escena.
- Visión: Contiene la definición de la cámara y las luces, este bloque es imprescindible para poder ver la escena, si se manejan varios archivos al menos uno debe tener la definición de las luces y solo uno debe tener la cámara (ya que solo se puede poner una).
- Definición de los objetos: Es donde se ponen los objetos con sus atributos y los efectos que se aplican a la escena.

No es necesario seguir estos bloques pero puede llegar a ser una buena subdivisión del archivo que contiene la escena.

4.1.1. Modelado.

Dentro del modelado se muestran algunas de las instrucciones para poner objetos en la escena y las instrucciones que permiten utilizar la GSC en este lenguaje. Las instrucciones que a continuación se describen van en el bloque de la definición de los objetos, no se muestran todos los atributos ya que muchos son generales y se explican más adelante.

El análisis de las instrucciones de este lenguaje se inicia con la descripción de sus primitivas donde los datos necesarios indicados para cada una deben ponerse al inicio de su definición, posteriormente se le pueden añadir otros atributos antes de finalizar la definición de dicho objeto.

TESIS CON
FALLA DE ORIGEN

Instrucción: box {<x1, y1, z1><x2, y2, z2>}
Descripción: Define una caja (con sus lados no necesariamente iguales) donde X1, Y1 y Z1 es un vértice de la caja y X2, Y2 y Z2 es el vértice opuesto al primero.
Instrucción: sphere {<X, Y, Z>, R}
Descripción: Define una esfera (completamente redonda) con centro en X, Y, Z y radio R.

Instrucción: cylinder {<X1, Y1, Z1><X2, Y2, Z2>, R}
Descripción: Define un cilindro donde los puntos X1, Y1, Z1 y X2, Y2, Z2 son los centros de los dos círculos que tiene a las orillas y R es el radio de estos.

Instrucción: cone {<X1, Y1, Z1>, R1, <X2, Y2, Z2>, R2}
Descripción: Define un cono donde X1, Y1, Z1 es el centro de uno de los lados, y R1 es su radio, y X2, Y2, Z2 es el centro del otro lado y R2 es su radio.

Instrucción: torus {R1, R2}
Descripción: Define un toroide (una dona), al carecer de valores que lo posicionen en un punto determinado, este siempre se encuentra en el centro de la escena (coordenada 0, 0, 0), si se desea poner en otra posición se tendrá que usar la traslación. R1 es el radio desde el centro de la escena hacia el centro del interior del toroide y R2 es el radio desde el centro del toroide hacia uno de sus extremos.

Instrucción: plane {<NX, NY, NZ>, D} ó plane {eje, D}
Descripción: Define un plano, la instrucción puede ser manejada de 2 formas. La primera es poner el vector normal NX, NY, NZ para saber la inclinación del plano, en la segunda se pone el eje al cual será perpendicular (el eje se pone con su correspondiente literal “x” “y” o “z” en minúscula).

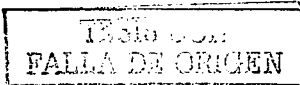
Las siguientes instrucciones son utilizadas para definir polígonos, ya sea utilizando una sola o varias en conjunto.

Instrucción: triangle {<P1> <P2> <P3>}
Descripción: Define un triángulo donde los P son sus coordenadas cada una con sus respectivos valores X, Y, Z, los puntos no deben situarse en el mismo lugar.

Instrucción: smooth_triangle {<P1><N1><P2><N2><P3><N3>}
Descripción: Al igual que la instrucción anterior ésta define un triángulo con la diferencia de que por cada punto se pone su vector normal.

Instrucción: mesh{**triangle**{...} y/o **smooth_triangle**{...}}
Descripción: Define una malla hecha a partir de triángulos, los triángulos son definidos con las instrucciones triange o smooth_triangle, o pueden estar algunos triángulos por una instrucción y los restantes por la otra en la misma malla.

Existen más instrucciones para definir polígonos pero estas son las más comunes y que mejor se adaptan a nuestras necesidades (por su facilidad e uso). Continuaremos con las instrucciones para las transformaciones básicas (traslación, rotación y escalamiento).



Instrucción: translate<X, Y, Z>
Descripción: Traslada el objeto las unidades indicadas en cada uno de los ejes, no hacia el punto que pudiera indicar, es decir, si un objeto se encuentra en 1,2,1 y en la instrucción se indicó 2,3,4 el punto terminará en 3,5,5.

Instrucción: rotate <X, Y, Z>
Descripción: Rota al objeto el ángulo indicado en cada uno de los ejes, el ángulo es dado en grados.

Instrucción: scale <X, Y, Z>
Descripción: Escala al objeto las unidades indicadas en cada eje, si el valor está entre 0 y 1 el objeto reduce su tamaño. Los valores no pueden ser 0 ya que se marca un error ya que el objeto llega a desaparecer, si se quiere mantener la proporción original en uno o más ejes su valor debe ser 1.

Al utilizar las instrucciones de traslación y rotación sobre un mismo objeto se debe cuidar el orden en que se aplican ya que no es lo mismo primero rotar y después trasladar que hacerlo en forma inversa. Estas 3 instrucciones están dentro del bloque de definición del objeto al final de todos sus parámetros.

Las siguientes instrucciones son las operaciones de la GSC que se pueden utilizar en este lenguaje, dentro de cada una están los objetos y atributos que van a interactuar con la operación correspondiente.

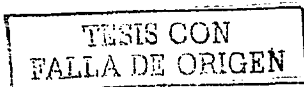
Instrucción: union {objetos y/o atributos}
Descripción: No modifica nada a los objetos, simplemente puede ayudar a aplicar el mismo color así como las transformaciones a todos los objetos definidos en su interior.

Instrucción: difference {objetos y/o atributos}
Descripción: Toma el primer objeto que contiene y elimina las áreas que ocupan los objetos restantes, estos últimos no se muestran en la escena solamente el primer objeto sin las áreas eliminadas. Al igual que la anterior, las transformaciones se pueden aplicar a todos los objetos, lógicamente el color solamente se aplica al primer objeto ya que es el único que puede ser mostrado.

Instrucción: intersection {objetos y/o atributos}
Descripción: Solo muestra las partes de los objetos que se estén tocando, el resto de los objetos lo elimina. También puede aplicarle el color y las transformaciones a todos los objetos.

Instrucción: merge {objetos y/o atributos}
Descripción: Esta solo muestra las partes de los objetos que no son interceptados por otros. De la misma manera puede aplicarle el color y las transformaciones a todos los objetos.

Estas pueden estar anidadas donde el objeto ahora será el resultado de la transformación o pueden ser definidas como un objeto.


TESIS CON
FALLA DE ORIGEN

4.1.2. Visión.

Se refiere a las instrucciones para la cámara, las luces, el color de los objetos con algunos de sus atributos, los materiales y las texturas. Se debe tomar en cuenta que el color (donde quiera que se utilice) usa el modelo RGB o RGBA y sus valores estarán entre 0 y 1, si todos sus valores son 0 el color será negro, al contrario si los valores son 1 el color será blanco, la A indica la transparencia si es 0 será totalmente transparente y si es 1 será sólido.

Instrucción: camera {tipo_de_camara
location <X, Y, Z>
look_at <X, Y, Z>}

Descripción: Pone la cámara en el punto señalado por location y la orienta para que capture la escena viendo hacia el punto dado por look_at, hay que tomar en cuenta que estos dos puntos deben ser distintos. El tipo de cámara se refiere a la proyección que se va a utilizar (este es opcional pero es importante conocerlo), en lugar de tipo_de_camara puede ir una de las siguientes palabras reservadas: perspective, ortographic, fisheye, ultra_wide_angle, omnimax, panoramic y cylinder. Esta contiene más atributos los cuales pueden ser consultados en la ayuda de POV-RAY.

Instrucción light_source {<x, Y, Z>
color <R, G, B>
modificadores}

Descripción: Agrega una fuente de luz a la escena en la posición indicada por (X, Y, Z), su color lo da la combinación de las intensidades proporcionadas en las variables (R, G, B). Los modificadores no son necesarios pero ayudan a agregarle efectos a la luz, se pueden clasificar en bloques, el nombre de estos no se pone cuando se utiliza alguno de sus modificadores, los bloques con sus respectivos modificadores son los siguientes:

Tipo de luz
- **spotlight**.
- **shadowless**.
- **cylinder**.

Opciones de la luz de spot
- **radius** radio
- **falloff** descenso
- **tightness** abertura (la fuente muy abierta o muy cerrada)
- **point_at** <X, Y, Z>

Opciones del área de la luz
- **area_light** <eje1>,<eje2>,tamaño1,tamaño2 (los ejes tienen sus valores X, Y, Z)
- **adaptive** adaptación
- **jitter** fluctuación

Modificadores generales de la luz
- **looks_like** {objeto}

- **fade_distance** distancia
- **fade_power** poder
- **media_atteniation** [Booleano]
- **media_interaction** [Booleano]

Combinando estos modificadores se pueden obtener las distintas luces que requiera una escena, para profundizar más en su uso se puede consultar la ayuda de POV_RAY.

Instrucción: material {**texture**{...}
Interior{...}
transformaciones}
Descripción: Utiliza las instrucciones texture e interior para darle el efecto al objeto por lo que no agrega ningún efecto extra y puede ser reemplazado por estas dos instrucciones. Las transformaciones no son necesarias pero al utilizarlas deben de ir debajo de la instrucción que se quiera modificar.

Instrucción: material_map{"mapa_de_bits"
modo
textura
transformaciones}
Descripción: Esta instrucción se utiliza dentro de texture (esto se vera más adelante), en mapa_de_bits se pone el nombre de una imagen incluyendo su extensión, como se muestra, el archivo va entre comillas. Los tipos de imagen soportados son: gif, tga iff, ppm, pgm, png y sys. Después va el modo en que se mapea la imagen al objeto, estos son:

map_type tipo
once
interpolate tipo

En textura se pone una instrucción texture donde su contenido solo son modificadores para retocar la imagen. Las transformaciones no son necesarias.

Instrucción: texture{opciones}
Descripción: Esta instrucción es muy compleja ya que puede ser utilizada de distintas formas cada una con distintos parámetros y modificadores. Aquí solo se explican las que siguen un patrón y las definidas por omisión en el archivo textures.inc.

Iniciare con las texturas de patrones.

Instrucción: texture{ID_textura_de_patrones
transformaciones}
Descripción: Contiene el identificador de la textura, las transformaciones no son necesarias.

Instrucción: texture{tipo_de_patrón
Modificadores_de_textura}

Descripción: Se le indica el patrón a seguir, los modificadores son opcionales.

Instrucción: texture{**titles** textura
title2 textura
transformaciones}
Descripción: En seguida de titles y tile2 se pone el nombre de una textura, estas pueden ser distintas. Las transformaciones son opcionales.

Instrucción: texture{**material_map**{...}}
Descripción: La textura está dada por material_map.

A continuación veremos como utilizar una de las texturas que POV-RAY tiene predefinidas, estas se pueden encontrar en los archivos glass.inc, metals.inc, stones.inc, woods.inc, texyures.inc y finish.inc.

Instrucción: texture{ID_de_textura
transformaciones}
Descripción: Contiene el identificador de la textura, las transformaciones no son necesarias. También hay texturas predefinidas que se utilizan en los modificadores (como finish{Shiny}).

Las siguientes instrucciones también influyen sobre el material y la textura del objeto ya que estas les dan el acabado al objeto o con estas se les aplican. Estas pueden estar situadas dentro o fuera de la instrucción texture. Con estas también se le puede añadir el color a los objetos.

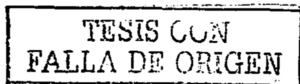
Instrucción: pigment{ID_textura atributos transformaciones}
Descripción: Contiene el identificador de una textura (como checker), sus atributos dependen de la textura y las transformaciones son opcionales.

Instrucción: pigment{**colour** ID_color}
Descripción: Le da al objeto el respectivo color que marca el identificador.

Instrucción: pigment{**color**<R, G, B>}
Descripción: Le da al objeto el color indicado por el modelo RGB.

Instrucción: finish{ID_terminación opciones_de_terminación}
Descripción: El identificador contiene una textura de terminación, este es opcional. Las opciones son las siguientes y pueden ser utilizadas todas juntas o solamente algunas de ellas:

- **ambient** color
- **diffuse** valor
- **brillance** valor
- **phong** valor
- **phong_size** valor
- **specular** valor
- **roughness** valor
- **matellic** valor (el valor es opcional)



- **reflection** color
- **reflection_exponent** valor
- **irid{thickness** valor **turbulence** valor}
- **crand** valor

4.1.3. Animación.

Para realizar la animación con este lenguaje se necesita editar otro archivo a parte del que contiene las instrucciones que definen la escena (el cual tiene extensión ".pov"), este archivo debe tener extensión ".ini" y por lo general tiene el mismo nombre que el archivo ".pov". La estructura del archivo para la animación es la siguiente:

```
Antialias=On

Antialias_Threshold=0.2
Antialias_Depth=3
Test_Abort_Count=100
Input_File_Name=1EE01.pov

Initial_Frame=0
Final_Frame=12
Initial_Clock=0
Final_Clock=12

Cyclic_Animation=on
Pause_when_Done=off
```

Los tres primeros valores son para eliminar el antialiasing que pueda tener la escena, se recomienda dejar los valores que se muestran aunque si pueden ser modificados, esto también se aplica para el cuarto valor. El siguiente valor indica el archivo con el cual se va a realizar la animación.

El parámetro Cyclic_Animation es un booleano que puede tener los valores "on" u "off", si está en "on" y en la animación el primer cuadro es idéntico al último, escalara el valor de la variable clock (que se verá más adelante) para que esto no suceda dejando el mismo número de cuadros indicados, pero el incremento de la variable clock ya no será el mismo. La instrucción Pause_when_Done también es un booleano.

Initial_Frame y Final_Frame indican el número del cuadro inicial y final respectivamente, no es necesario que los cuadros inicien en 0, la única restricción es que el cuadro inicial contenga un número menor al cuadro final, el intervalo entre estos será el número de cuadros que genere el programa.

TESIS CON
FALLA DE ORIGEN

Initial_Clock y Final_Clock indican el valor inicial y final de la variable clock (que es la única variable que se puede declarar en este tipo de archivos), sus valores no tienen que ser iguales a los de las instrucciones para poner el número de cuadros. Los valores que va tomando la variable en cada uno de los cuadros están dados por una interpolación tomando los valores iniciales y finales de los cuadros y de la variable, por ejemplo si tenemos en los cuadros inicial y final los valores 11 y 19 respectivamente y en los del clock 6 y 10 la variable tomará los valores del siguiente cuadro con respecto al cuadro en turno:

Número de cuadro	Valor de clock
11	6
12	6.5
13	7
14	7.5
15	8
16	8.5
17	9
18	9.5
19	10

Cuadro 4.1. Interpolación de los valores de la variable clock.

Ahora como mencione anteriormente la variable clock es la única que se maneja con estos archivos y puede ser utilizada en cualquier parte del código modificando los valores de los atributos, las transformaciones u otras cosas. Como ejemplo podemos tomar el siguiente fragmento de código que trasladaría una esfera sobre el eje x.

```
...
sphere{<0, 0, 0>, 4
        translate <clock, 0, 0>}
...
```

Hay que notar que el valor inicial de la variable podría ser 0 para que concuerde con la posición del centro de la esfera aunque no es estrictamente necesario ya que la esfera es trasladada a donde indique la variable.

4.1.4. Programación.

Con este lenguaje de modelado también se puede programar gracias a que tiene sentencias condicionales (como el if) y de ciclos (como el while), estas sentencias están precedidas siempre por el símbolo "#". También es posible definir variables para poder utilizarlas en las sentencias anteriormente mencionadas. A continuación se explica un pequeño conjunto de este tipo de instrucciones.

TESIS CON
FALLA DE ORIGEN

Instrucción: #include "archivo.inc"
Descripción: Incluye un archivo con definiciones de algunas características (como las texturas) para que no se tenga que poner la definición completa de alguna de estas.

Instrucción: #define variable = valor
Descripción: Esta define una variable con su respectivo valor, la cual podrá ser utilizada en el resto del programa, "valor" puede contener un simple valor numérico, una variable anteriormente declarada o una ecuación con números y/o variables anteriormente declaradas, por ejemplo:

```
#define var1 = 2
#define var2 = var1
#define var1 = var1 + var2
```

Esta instrucción también puede asignar un objeto, una transformación, una operación (como unión donde se tenga la definición de un objeto más complicado), etc., a una variable, esto es muy útil para ahorrar espacio al estar añadiendo varias veces el mismo objeto en distintos sitios o al llamarlo desde otro archivo.

Instrucción:

```
	#if (condición)
	...
	#end
```

Descripción: Funciona igual que en los lenguajes de programación, si la condición se cumple se ejecuta el bloque de instrucciones que contiene, en caso contrario el programa se salta hasta encontrar su bloque #end correspondiente. Esta instrucción también puede utilizar el bloque #else para que en caso de que la condición sea falsa se ejecute el bloque de instrucciones que contiene. Su estructura queda de la siguiente manera:

```
#if (condición)
...
#else
...
#end
```

Instrucción:

```
	#switch (expresión)
	#case (valor)
	...
	#break
	#range (mínimo, máximo)
	...
	#break
	#end
```

Descripción: La instrucción #switch evalúa la expresión (que puede ser una variable), el resultado lo compara con el valor contenido en las instrucciones #case, si es igual ejecuta el bloque de instrucciones que contiene, en caso contrario se pasa a la siguiente instrucción #case hasta pasar por todas estas. Para la instrucción #range es algo similar pero ésta verifica si el resultado de la expresión está dentro del rango indicado por los valores mínimo y máximo que contiene la instrucción. Cada instrucción #case o #range tiene su delimitador #break para saber donde termina su bloque de instrucciones. Dentro de la instrucción #switch se puede utilizar solo instrucciones #case o #range o pueden combinarse, además puede ser utilizada la instrucción #else que indica el bloque a ejecutarse si ningún caso es procesado, ésta no necesita de la instrucción #break ya que la instrucción #end del #switch le sirve de delimitador.

Instrucción:

```
#while (condición)
...
#end
```

Descripción: Define un ciclo, se comporta de la misma manera que en los lenguajes de programación, procesará el bloque de instrucciones que contiene mientras la condición sea verdadera por lo que es importante que el usuario cambie los parámetros de la condición dentro de este bloque ya que se puede llegar a tener un ciclo infinito.

El lenguaje aún contiene otras instrucciones para la programación, pero las que se han analizado son las más representativas, para conocer el uso y funcionamiento del resto de las instrucciones se puede consultar la ayuda del programa.

4.2. VRML.

El VRML (lenguaje de modelado de realidad virtual) tiene un despliegue muy pobre ya que está enfocado a su uso en Internet donde no se pueden manejar tantos datos ya que estos deben viajar lo más rápido posible sobre la red. Este necesita 2 cosas para poder desplegar las escenas que tengan este formato, una es el navegador y la otra es el plugin que permita al navegador poder desplegar los datos de este tipo de archivos.

Es cierto que este lenguaje no posee gráficas poderosas pero el hecho de utilizar gráficas de bajo nivel es a causa de que está más orientado a que los usuarios puedan recorrer (o navegar) libremente por la escena (llamada mundo virtual) a obtener cuadros para posteriormente realizar una animación (como los otros modeladores que se han analizado en este trabajo), estas características enfocan al lenguaje a que sea dibujado en tiempo real.

Aún con sus gráficas algo pobres en calidad, el resultado final puede llegar a ser bueno ya que permite el uso de texturas sobre los objetos y es gracias a éstas que podemos mejorar la calidad de la escena. Este lenguaje cuenta con pocos objetos predefinidos

TESIS CON
FALLA DE ORIGEN

(primitivas), pero al igual que con los otros modeladores, conjugando estas podemos obtener objetos más complicados, el único límite es la imaginación de quien realice la escena.

Es importante saber que la siguiente explicación del lenguaje se hace basándose en su segunda versión por lo que todos los archivos tendrán que utilizar el siguiente encabezado:

#VRML V2.0 utf8

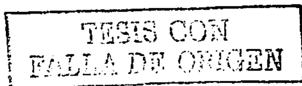
Su extensión es ".wrl", estos son archivos de texto por lo que se pueden editar en cualquier procesador de palabras (incluso el bloc de notas de Microsoft), solo se debe tener cuidado de salvarlo como archivo de texto con su respectiva extensión ya que con cualquier otro formato el navegador no lo reconocerá.

4.2.1. Nodos.

VRML se maneja por una jerarquía de nodos los cuales son comandos que definen formas y propiedades, esta estructuración jerárquica se hace de acuerdo con su tipo y secuencia para poder modelar una escena. El contenido de cada uno de los nodos está delimitado por "{" y "}", dentro pueden encontrarse otros nodos o parámetros que lo describen. Los nodos tienen las siguientes características:

- Naturaleza: Es el tipo del nodo referido ya sea una primitiva (cubo, esfera) o una transformación (rotación), etc.
- Parámetros: Son los valores de las características que puede tener el nodo.
- Nombre: Se le puede añadir un nombre a un nodo (esto no es estrictamente necesario), con esto se puede tener ya definida una estructura compleja (como una copa, una silla, etc.) y mandarse a llamar en el mismo archivo o desde otro distinto simplemente con el nombre que se le asigno sin tener que agregar la definición completa del objeto.
- Nodos hijos: Ya que VRML se basa en una estructura jerárquica de nodos se tienen nodos padres y nodos hijos, donde los nodos hijos están contenidos en los padres, estos últimos son llamados nodos de grupo y pueden contener cero o más nodos hijos.

Los nodos con respecto a su naturaleza pueden clasificarse en nodos de forma, propiedad, grupo e interpolación. Los primeros se refieren a los que se pueden agregar a la escena, los de propiedad son las características que afectan directamente a la apariencia de los objetos, los de grupo son los que pueden contener varios nodos y los de interpolación ayudan a realizar cambios en la posición o en la apariencia de los objetos logrando de esta manera una animación dentro del mundo virtual.



Los nodos que contiene la primera versión de VRML2.0 están divididos en 9 grupos, los cuales son:

- Nodos de agrupación:
  - Anchor
  - Billboard
  - Collision
  - Group
  - Transform
- Grupos especiales:
  - Inline
  - LOD
  - Switch
  - Script
  - Shape
- Nodos comunes:
  - AudioClip
  - DirectionalLight
  - PointLight
  - Sound
  - SpotLight
  - WorldInfo
- Sensores:
  - CylinderSensor
  - PlaneSensor
  - ProximitiSensor
  - SphereSensor
  - TimeSensor
  - TouchSensor
  - VisibilitySensor
- Geometría:
  - Box
  - Cone
  - Cylinder
  - ElevationGrid
  - Extrusion
  - IndexedFaceSet
  - IndexedLineSet
  - PointSet
  - Sphere
  - Text
- Propiedades geométricas:
  - Color
  - Coordinate
  - Normal
  - TextureCoordinate
- Apariencia:
  - Appearance

TESIS CON
FALLA DE ORIGEN

- FontStyle
- ImageTexture
- Material
- MovieTexture
- PixelTexture
- TextureTransform
- Interpoladores:
  - ColorInterpolator
  - CoordinateInterpolator
  - NormalInterpolator
  - OrientationInterpolator
  - PositionInterpolator
  - ScalarInterpolator
- Nodos de amarre:
  - Backgroung
  - Fog
  - NavigationInfo
  - ViewPoint

4.2.2. Modelado.

A continuación se ven algunas de las instrucciones que ayudarán a realizar las escenas, estas estarán contenidas en el nodo Shape para que los objetos puedan ser visibles ya que de otra manera no se verán, esto es porque "geometry" (que es el grupo al que pertenecen estos nodos) sirve como uno de sus campos como se ve a continuación:

```
Shape {
	geometry objeto{...}
}
```

En objeto se ingresa una de las siguientes instrucciones. Hay que tomar en cuenta que el lenguaje si hace distinción entre mayúsculas y minúsculas por lo que se debe de respetar la forma en que se presentan.

Instrucción:

```
	Sphere {
		radius radio
	}
```

Descripción: Agrega una esfera a la escena donde radius indica su radio.

Instrucción:
```
Box {
        size X Y Z
}
```
Descripción: Define una caja donde size contiene el tamaño sobre X, Y y Z.

Instrucción:
```
Cone {
        bottomRadius radio
        height altura
        side booleano
        bottom booleano
}
```
Descripción: Define un cono, el radio de su base esta dado por bottomRadius mientras que height indica su altura. Debemos tomar en cuenta que las figuras se manejan como polígonos por lo que podemos decir cual cara se dibuja, esto se indica en side, con el valor TRUE dibuja la cara exterior y con FALSE la interior, bottom es algo parecido, con TRUE si es visible la base del cono mientras que con FALSE esta desaparece.

Instrucción:
```
Cylinder {
        bottom booleano
        height altura
        radius radio
        side booleano
        top booleano
}
```
Descripción: Define un cilindro donde la altura está dada por height y su radio por radius. Los campos bottom, side y top se manejan de la misma forma que en el cono con la diferencia de que bottom y top se refieren a la base inferior y superior respectivamente.

Instrucción:
```
Text {
        string "texto"
        fontStyle FontStyle {...}
}
```
Descripción: Añade el texto definido en string, este va entre comillas. El nodo FontStyle contiene los atributos del texto.

Todos los nodos anteriores pertenecen al grupo de geometría y por tal motivo deben estar precedidos por "geometry".

TESIS CON
FALLA DE ORIGEN

Instrucción:

```
FontStyle {
        family "Tipo de letra"
        horizontal boleana
        justify justificación
        leftToRight booleano
        size tamaño
        spacing espaciado
        style "Estilo"
        topToBottom booleano
}
```

Descripción: Define la apariencia que va a tener el texto donde family es el tipo de letra a utilizar como SANS o TYPEWRITER, horizontal indica si el texto se va a escribir horizontalmente (TRUE) o verticalmente (FALSE), justify indica la justificación del texto ya sea BEGIN, FIRST, MIDDLE o END, mientras que leftToRight dice la dirección en que se va a escribir y topToBottom indica si el texto se escribirá normal o de cabeza, size y spacing contienen el tamaño de la letra y el espaciado entre palabras respectivamente y por último style indica el estilo de letra ya sea ITALIC, BLOD, etc.

Instrucción:

```
Coordinate {
        Point [
                X1 Y1 Z1,
                X2, Y2, Z2,
                ...
                ]
}
```

Descripción: Define una serie de coordenadas en 3D contenidas en el arreglo point cada conjunto de 3 números indica un valor en X, Y y Z donde cada coordenada está separada por una coma. No existe un límite para el número de coordenadas.

Instrucción:

```
Color {
        color [
                R1 G1 B1,
                ...
        ]
}
```

Descripción: Define el color que se le aplicará al objeto, este utiliza el modelo RGB donde sus valores van del 0 al 1, el 0 indica la ausencia del componente en cuestión y el 1 su presencia con la máxima intensidad. Se puede declarar más de un color separándolos por comas, cada uno debe tener sus tres componentes.

Instrucción:

```
PointSet {
        coord Coordinate{...}
        color Color{...}
}
```

Descripción: Muestra una serie de puntos donde la posición de cada uno de estos está contenida en el nodo Coordinate. Con Color agrega color a los puntos donde para cada punto se define un color.

Instrucción:

```
IndexedLineSet {
        coord Coordinate{...}
        coordIndex [...]
        color Color{...}
        colorIndex [...]
        colorPerVertex booleano
}
```

Descripción: Dibuja una serie de líneas utilizando las coordenadas contenidas en Coordinate, el orden en que se usan lo contiene coordIndex donde al primer punto definido en Coordinate se le asigna el índice 0, al segundo el 1 y así sucesivamente, de esta manera se van uniendo los puntos con líneas, para terminar de unir estos puntos se debe ingresar un –1 y cada índice va separado por una coma; por ejemplo, si queremos unir las primeras 3 coordenadas de manera que la segunda coordenada definida en Coordinate este en primer lugar, la tercera en segundo y la primera hasta el último se deberá poner como sigue:

```
coordIndex [ 1, 2, 0, -1]
```

Se pueden definir varias líneas no solo una, cada una de estas termina con un –1 como se vio en el ejemplo, para pasar a una nueva línea después del –1 se pone coma para dar paso a los índices de las coordenadas que utilizará esta nueva línea. Con Color se definen los colores que pueden ser utilizados, colorIndex contiene el orden en que van a ser aplicados los colores definidos dentro de Color, al primer color definido se le da el índice 0 y se va incrementando sucesivamente, estos pueden estar repetidos y en desorden, la cantidad de índices que se utilicen dependerá del número de coordenadas, de líneas y de la instrucción colorPerVertex ya que ésta indica si el color se aplicará a cada vértice haciendo un degradado sobre la línea de uno hacia otro (con TRUE) o se aplicará sobre la misma línea obteniendo un mismo color sobre la línea entera.

Instrucción:

```
IndexedFaceSet {
        coord Coordinate{...}
        coordIndex [...]
        color Color{...}
        colorIndex [...]
        colorPerVertex booleano
}
```

TESIS CON
FALLA DE ORIGEN

Descripción: Esta es muy parecida a la anterior, la única diferencia es que en vez de poner solo líneas, encierra áreas y forma las caras de un polígono. En esta es más notoria la diferencia entre los valores de colorPerVertex.

Los últimos 3 nodos pertenecen al grupo de geometría por lo que también deben estar precedidos por "geometry".

Otro campo que contiene el nodo Shape es el grupo appearance el cual utiliza el nodo Appearance como contenedor de otros nodos del mismo grupo como se muestra a continuación, se puede utilizar uno o más.

```
Shape{
        geometry objeto{...}
        appearance Appearance{
                material Material{...}
                texture textura{...}
                textureTransform transformación{...}
        }
}
```

Del campo material solo se tiene el nodo Material, en texture se tienen más nodos al igual que en textureTransform, algunos de estos nodos se ven a continuación.

Instrucción:

```
        Material {
                diffuseColor R G B
                emisiveColor R G B
                transparency transparencia
                shininess brillo
                specularColor R G B
        }
```

Descripción: Define los atributos del material del objeto. Los campos que manejan color como diffuseColor usan el modelo RGB y sus valores están entre 0 y 1 donde 0 es la ausencia de este componente y 1 es su máxima intensidad, transparency indica la trasparencia del material mientras que shininess es su brillo.

Instrucción:

```
        ImageTexture{
                url["imagen.extención"]
        }
```

Descripción: Añade una imagen fija al objeto como textura, el nombre de la imagen está especificado en el arreglo url, este se pone entre comillas. Si la imagen no se encuentra en el mismo directorio se debe especificar la ruta completa de donde se encuentre (con todo y nombre con extensión). Los formatos que soporta son jpg, gif y png.

Instrucción:
```
MovieTexture{
        loop booleano
        speed velocidad
        url["archivo.mpeg"]
}
```
Descripción: Añade un video con formato mpeg al objeto como textura, loop indica si el video se va a estar repitiendo o solamente se ve una vez y se detiene, speed indica la velocidad, con 1 es normal, y url es el nombre sólo o con la ruta completa del archivo. Para esta instrucción el único formato soportado es el mpeg.

Instrucción:
```
TextureTransform{
        scale X Y
        translation X Y
}
```
Descripción: Le aplica transformaciones a la imagen que se utilice como textura, scale indica las veces que se pone la textura sobre cual eje dividiendo el área del objeto.

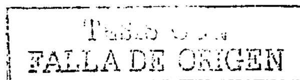
En VRML las transformaciones tienen un papel muy importante ya que todos los objetos siempre aparecen en el centro de los ejes coordenados, se aplican de un modo distinto a los otros modeladores ya que en este lenguaje se les da más peso jerárquico, por lo que una transformación contiene al nodo Shape, de esta forma a todos los objetos que contenga se les aplicaran las mismas transformaciones, quedando de manera general como sigue:

```
Transform{
        transformación
        children[
                Shape {...}
        ]
}
```

Hay que notar que el campo children es el que contiene la definición de todos los objetos, por lo tanto este debe contener a Shape. Las transformaciones que pueden ser aplicadas son las siguientes:

- center X Y Z: Indica la coordenada donde estará el centro de los objetos.
- rotation X Y Z ángulo: Aplica una rotación donde el ángulo está en radianes y con X Y Z se elige el eje con respecto del cual se va a rotar, si el valor es 1 el eje se toma en cuenta, si es 0 se ignora.
- scale X Y Z: Escala el objeto de acuerdo a los valores que contienen cada uno de los ejes.

En VRML la cámara tiene su propia luz por lo que la escena estará siempre iluminada, no obstante esta se puede desactivar para poder colocar distintos tipos de luces en los lugares deseados dando como resultado una escena más detallada. Para desactivar la luz se debe poner el siguiente nodo el cual no depende de nadie:


FALLA DE ORIGEN

```
NavigationInfo {
        headlight FALSE
}
```

Las siguientes instrucciones definen distintos tipos de luces, estas no tienen dependencia con otros nodos.

Instrucción:

```
PointLight {
        color R G B
        intensity intensidad
        location X Y Z
        radius radio
}
```

Descripción: Pone un punto que emite luz hacia todas direcciones, la instrucción contiene su color, su intensidad donde su valor está entre 0 e infinito, la coordenada donde se encuentra y el radio que abarca.

Instrucción:

```
DirectionalLight {
        color R G B
        intensity intensidad
        direction X Y Z
}
```

Descripción: Define una luz que cruza la escena de un lado hacia otro, también contiene su color y su intensidad, la dirección hacia donde se dirige está dada por el vector definido en direction.

Instrucción:

```
SpotLight {
        color R G B
        intensity intensidad
        beamWidth ángulo
        cutOffAngle ángulo
        location X Y Z
        radius radio
        direction X Y Z
}
```

Descripción: Define una luz de spot, los campos color, intensity, location, radius y direction funcionan igual que en las otras luces. Los ángulos beamWidth y cutOffAngle son la apertura que alcanza la luz más intensa (la del centro) y la desvanecida (la de las orillas), su rango de valores está entre 0 y $\Pi/2$.

En este lenguaje también se puede utilizar un fondo para darle más realismo a la escena, este puede ser simplemente colores o imágenes, estos nodos tampoco dependen de otros.

TESIS CON
FALLA DE ORIGEN

Instrucción:

```
BackGround {
        skyColor [R1 G1 B1,
                ...]
        skyAngle [ángulo1,
                ...]
}
```

Descripción: Define uno o más colores de fondo con skyColor y con skyAngle se indica el ángulo hasta el que llega cada uno, el rango va de 0 a Π. Si se utiliza más de un color para pasar de uno a otro se utilizan degradados.

Instrucción:

```
BackGround {
        topUrl []
        boottomUrl []
        leftUrl []
        rightUrl []
        backUrl []
        frontUrl []
}
```

Descripción: Encierra a la escena dentro de una especie de cubo (aunque sigue teniendo forma esférica) y le pone una imagen a cada cara, todos sus campos funcionan igual que el url visto con anterioridad.

Con este conjunto de instrucciones se puede modelar una escena por la cual se podrá navegar gracias a sus características.

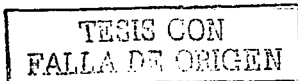
4.2.3. Animación.

Con VRML además de poder recorrer los mundos virtuales que se realizan, también se puede tener animaciones que se estarán ejecutando mientras el usuario explora la escena. Para lograr la animación se utilizan sensores e interpoladores por lo que la animación se realiza con un pequeño sentido de programación.

Para realizar una animación se deben declarar variables para referirnos al objeto involucrado y el interpolador que se va a utilizar, así como el sensor si se requiere. Para definir variable se utiliza la palabra DEF seguida del nombre de la variable que se utilizará como referencia, enseguida de esta se encontrará el nodo al que se quiera referir, esto queda como sigue:

DEF variable nodo{...}

Como se mencionó, el nodo puede ser cualquiera, aún estando dentro de otro (aunque para referirse a objetos se recomienda asignar la variable desde el nodo Transform


TESIS CON
FALLA DE ORIGEN

para que al realizar la animación se tengan en cuenta todos los cambios que se le han aplicado al objeto).

Instrucción:

```
CoordinateInterpolator {
        setFraction fracción
        key [tiempo1, ...]
        keyValues [X1 Y1 Z1, ...]
        value_changed valor
}
```

Descripción: Realiza una interpolación entre las coordenadas contenidas en keyValues para desplazar al objeto, la interpolación la realiza en el orden en que están las coordenadas. El arreglo key contiene el tiempo en que llega a cada una de las coordenadas por lo que hay el mismo número de coordenadas que de tiempos en este arreglo, los valores de key van del 0 (tiempo inicial) hasta el 1 (tiempo final), sus valores llevan un orden creciente dentro de este intervalo. El campo setFraction se utiliza para indicar el número de puntos que se van a utilizar para la interpolación entre dos coordenadas. El campo value_changed se utiliza para conocer si el valor a interpolar a cambiado y de esta manera censar el cambio para aplicarlo en el objeto.

Instrucción:

```
TimeSensor {
        cycleInterval tiempo
        enabled booleano
        loop booleano
        startTime tiempo
        stopTime tiempo
        cycleTime tiempo
        fraction_changed fracción
        isActive booleano
        time tiempo
}
```

Descripción: Define un sensor que genera eventos a lo largo del tiempo. El campo cycleInterval indica cuando el sensor del tiempo inicia en cada ciclo. Cuando enabled esta en TRUE el sensor puede ejecutarse, si es FALSE el evento debe ser recibido mientras el tiempo del sensor corre. Con loop se indica si el sensor se ejecutará continuamente después de iniciar (TRUE) o se ejecutara una sola vez (FALSE). Con startTime y stopTime se indican los tiempos de inicio y final respectivamente. Las variables que se pueden ocupar para obtener el estado y el tiempo del sensor son: cycleTime que indica cuando un ciclo inicia, fraction_changed es el valor que tiene el sensor en un tiempo dado, isActive indica si el sensor se está ejecutando o está pasivo y time contiene el tiempo actual del sensor.

VRML tiene más tipos de sensores (como el touchSensor), todos siguen el mismo principio de tener distintos valores en los diferentes lapsos de tiempo para posteriormente ocuparlos modificando algunos de los parámetros de los objeto.

La programación con este lenguaje no es realizada con estos sensores (aunque así parezca), sino que se puede insertar código escrito en java script para lograr otro tipo de efectos dentro de la escena. Este lenguaje no se explicará dentro de este trabajo ya que es muy extenso y no es parte de VRML.

TESIS CON
FALLA DE ORIGEN

CAPÍTULO 5

DISEÑO DEL FORMATO PARA MATERIAL3D

TESIS CON
FALLA DE ORIGEN

Basándose en los análisis que se realizaron en los capítulos anteriores se puede tener una idea de cómo se tuvo que estructurar el formato a diseñado. Se tienen 2 vertientes distintas, una es ver los objetos como colecciones de vértices especificando cada una de sus coordenadas, lógicamente esta es utilizada por los modeladores 3D donde el usuario maneja gráficamente los objetos y no tiene que estar escribiendo la escena directamente en el archivo con el formato. La segunda trata tanto a los objetos como a los demás efectos y características como instrucciones, de esta forma el usuario no tiene que estar especificando los vértices de cada objeto (aunque sí puede hacerlo) sino que simplemente ocupa la instrucción para crear la primitiva deseada indicando sus respectivos parámetros, esta vertiente es utilizada por los lenguajes de modelado 3D.

Cada una de las vertientes tiene sus ventajas y desventajas, por ejemplo el estar especificando vértices es muy tardado y tedioso (porque el usuario es el que decide donde se ubica cada uno) pero con los modeladores esto es automático además de que es más fácil escribir un programa que grafique una serie de coordenadas, a sacar las coordenadas de una instrucción con sus respectivas características para graficar.

Pero el especificar los vértices no es tan factible cuando solo se quiere realizar una escena simple y más si se utilizan puras primitivas ya que el archivo puede llegar a estar tan grande y revuelto a tal grado que ya no se entienda (aun siendo almacenado en código ASCII como es el caso del formato obj).

Una de las soluciones más factibles es poder utilizar las dos vertientes en un mismo formato, esto ya se ha implantado en el lenguaje POV-RAY, el problema que tiene es que por ser un lenguaje de modelado se tiene que realizar el render para poder revisar la escena y de esta forma ver si los objetos están en el lugar deseado. De esta forma si el formato tiene la posibilidad de manejar las 2 vertientes se tiene una gran ventaja ya que los objetos pueden ser vistos inmediatamente y los vértices se pueden crear automáticamente ya que hay que recordar que el formato va a ser para un modelador 3D. Tomando en cuenta estos factores se especificaron las instrucciones para poder modelar objetos.

5.1. Implantaciones del modelador.

Para el diseño de las instrucciones del formato primero se revisaron las distintas partes que tiene implantadas material3D ya sea crear objetos o aplicarles transformaciones, es importante ver la definición de cada una de estas funciones ya que los parámetros que utilice serán la base para diseñar las estructuras que almacenarán los datos de la escena.

Una cosa que se debe recordar es que las funciones que crean los objetos simplemente obtienen sus vértices ya que el despliegue del modelador se realiza con OpenGL y como ya se reviso en el capítulo 2 esta biblioteca trabaja solamente con

vértices, por lo que se necesita una estructura que los almacene junto con el número de vértices y de triángulos (ya que formar figuras con éstos da un mejor resultado, además de que no es complicado representarlos con OpenGL) que contenga el objeto, esta estructura junto con otras dos están actualmente implementadas y ayudan al modelador para almacenar los vértices. Las estructuras mencionadas son las siguientes (se debe tomar en cuenta que se están utilizando tipos de variables de GTK ya que con estas bibliotecas se programó la interfaz gráfica, por esa razón llevan una g antes del tipo).

```
typedef strcut {
        gdouble x, y, z;  // Coordenadas X Y Z
}point3D;

typedef struct {
        gint v1, v2, v3;  // Número del punto para el vértice 1, 2 y 3
        guchar r, g, b;  // Color con el modelo RGB en hexadecimal
}typTriangle;

typedef struct {
        gchar *name;  // Nombre del objeto
        gint numVertex;  // Número de vértices
        gint numTriangles;  // Número de triángulos
        gint typeObject;  // Tipo de objeto (esfera, cubo, etc.). Los valores ya
                          // están definidos.
        point3D *vertexArray;  // Contiene todos los vértices del objeto
        typTriangle *trianglesArray;  // Contiene la forma en que se conectan los vèrtices
        gboolean selected;  // Indica si el objeto está seleccionado (TRUE).
}m3DObject;
```

A simple vista la estructura contiene los datos necesarios para tener un formato que almacene los vértices de los objetos así como la forma en que se conectan, pero analizando más a fondo se puede observar que la estructura cuenta con un campo que indica el tipo del objeto (esfera, cubo, toroide, etc.) con el cual las instrucciones que definen objetos con los datos necesarios propios de la primitiva tienen un apoyo para encajar dentro de la estructura.

Para este tipo de instrucciones es necesario revisar el encabezado de las funciones que obtienen los vértices de cada una de las primitivas ya que estas son las que utilizan dichos parámetros recibiéndolos al mandarlas llamar. El cuerpo de la función no es de interés para el formato debido a que no importa el método con el que se obtengan los vértices si los datos que recibe no cambian. A continuación se muestra el encabezado de dichas funciones.

```
void create_sphere (gint numLevels, gint numSides, gdouble radio);
```

Crea una esfera donde numLevels es el número de cortes que tendrá de abajo para arriba la esfera y numSides es número de caras que tendrá alrededor de esta, con radio se indica el radio de la esfera.

TESIS CON
FALLA DE ORIGEN

```
void create_box (gint divX, gint divY, gint divZ);
```

Crea una caja, divX, divY y divZ indican el número de divisiones que tendrá a lo largo del eje correspondiente, a ésta no se le indica el tamaño que tendrá por lo que se debe utilizar un escalamiento para modificarlo.

```
void create_cylinder (gint numLevels, gint numSides, gdouble height,
                      gdouble radio);
```

Crea un cilindro done numLevels, numSides y radio indican lo mismo que para la esfera pero aplicados al cilindro, height es su altura.

```
void create_torus (gint numLevels, gint numSides, gdouble radioExt,
                   gdouble radioInt);
```

Crea un toroide o dona, numLevels y numSides indican lo mismo que los anteriores, radioExt y radioInt son sus radios externo e interno respectivamente, estos radios son como los del toroide que maneja POV-RAY.

```
void create_cone (gint numLevels, gint numSides, gdouble height,
                  gdouble radio);
```

Crea un cono, todos sus parámetros indican lo mismo que en las funciones anteriores donde el radio a diferencia del cilindro solamente se aplica en la base.

Es claro que algunos objetos utilizan parámetros en común pero que cada uno tiene su conjunto de parámetros en especial, de forma que podemos listar los parámetros utilizados para ver que variables se necesitan en una estructura que almacene dichos parámetros del objeto.

```
gint numLevels
gint numSides
gint divX
gint divY
gint divZ
gdouble radio
gdouble height
gdouble radioInt
gdouble radioExt
```

En esta serie de variables se tienen 3 que guardan radios siendo que solamente utilizamos 2 por lo que radio y radioExt se manejarán con la misma variable, por lo que solo quedará radio. Con esta estructura se puede llegar a pensar que todos los objetos tienen características que no le pertenecen, pero el programa internamente solo manejará los datos que necesite el objeto en cuestión dejando los otros datos que no le pertenecen con el valor nulo (NULL). En la estructura general del objeto es necesario

añadir dicha estructura de parámetros la cual puede o no utilizarse, esto depende de la variable "gboolean vertices" antes mencionada.

Existen otras funciones que afectan directamente a las coordenadas de los vértices del objeto, estas son las transformaciones, sus funciones reciben los mismos parámetros, estos son la transformación en cada uno de los ejes.

```
void translate(gdouble xDist, gdouble yDist, gdouble zDist);
void scale(gdouble xScale, gdouble yScale, gdouble zScale);
void rotate(gdouble xAngle, gdouble yAngle, gdouble zAngle);
```

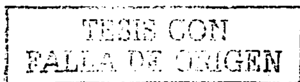
De esta manera solo se necesita el parámetro en cada uno de los ejes para aplicar la transformación, estas también deben de ser incluidas dentro de la estructura del objeto para poder conocer su ubicación, tamaño y orientación. Estos datos deben de ser almacenados en una estructura como la siguiente.

```
typedef struct {
        gint type;
        gdouble x, y, z;
}transform;
```

De esta manera la estructura del objeto queda como se muestra a continuación.

```
Typedef struct {
        gint numLevels;
        gint numSides;
        gint divX;
        gint divY;
        gint divZ;
        gdouble radio;
        gdouble height;
        gdouble radioExt;
} objParam;

typedef struct {
        gchar *name;
        gint numVertex;
        gint numTriangles;
        gint typeObject;
        point3D *vertexArray;
        typTriangle *trianglesArray;
        gint numTransf;
        transform *transfArray;
        gboolean vertices;
        objParam parametros;
        gboolean selected;
}m3DObject;
```



ESTA TESIS NO SALE
DE LA BIBLIOTECA

Una última estructura que debemos considerar es la que contiene todos los objetos de la escena, esta es muy simple ya que actualmente solo maneja objetos (porque su color está dentro de sus vértices).

```
typedef strcut {
	gint numObjects;
	m3DObject *objectArray;
} m3DScene;
```

Aun quedan algunos elementos que se proponen integrar a este modelador en futuras versiones, estos elementos se revisan a continuación.

5.2. Formato de Material3D.

Con las estructuras revisadas anteriormente se tienen los elementos para poder iniciar a diseñar el formato del modelador, este fue escrito en XML por las cualidades antes mencionadas sobre éste lenguaje. Obviamente la marca principal que contendrá a todas las otras marcas será la que pertenezca a la estructura de la escena mencionada en este capítulo, esta no tiene atributos simplemente contiene objetos por lo que puede quedar de la siguiente manera.

```
<M3DScene>
	Marcas de los objetos
</M3DScene>
```

Dentro de este elemento se tienen las marcas de los objetos que puede manejar el modelador, para estas marcas se toman en cuenta los datos recibidos por las funciones analizadas que obtienen el conjunto de vértices de los objetos debido a que son los atributos que los describen.

La esfera tiene como atributos necesarios el número de niveles, el número de lados y el radio por lo que estos tendrán que ser obligatorios dejando al nombre y su color como opcionales (ya que este último tiene un valor por omisión).

```
<sphere numLevels=gint numSides=gint radio=gdouble>
	Marcas de parámetros.
</sphere>
```

La caja tiene como atributos necesarios las divisiones en cada uno de los ejes donde sus parámetros son los mismos que en la esfera.

```
<box divX=gint divY=gint divZ=gint>
	Marcas de parámetros
</box>
```

Los atributos necesarios para el cilindro son el número se divisiones, el número de lados, la altura y el radio donde los parámetros que maneja son los mismos que la esfera.

<cylinder numLevels=gint numSides=gint height=gdouble radio=gdouble>
Marcas de parámetros
</cylinder>

Para el toroide o dona los atributos necesarios son el número de niveles, el número de lados, el radio exterior y el radio interior, este maneja los mismos parámetros de la esfera.

<torus numLevels=gint numSides=gint radioExt=gdouble radioInt=gdouble>
Marcas de parámetros
</torus>

El cono tiene como atributos necesarios el número de niveles, el número de lados, la altura y el radio de su base, manejando los mismos parámetros que la esfera.

<cone numLevels=gint numSides=gint height=gdouble radio=gdouble>
Marcas de parámetros
</cone>

Todos los objetos anteriores podrán utilizar las mismas marcas para sus parámetros ya que los que utilizan los objetos dentro del programa son del mismo tipo, estos son el nombre, el color y las transformaciones aplicadas sobre el objeto donde cada marca tiene relacionados sus parámetros con los datos que tiene la estructura que los almacena, en la estructura general del objeto está implícito el color, ya que este se encuentra dentro de la estructura del triángulo junto con los vértices pero para este tipo de marcas se puede utilizar su contenido como color general para todo el objeto.

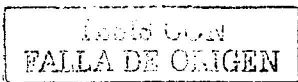
Para el parámetro del nombre solamente se necesita una cadena de caracteres que contenga el nombre que el usuario asigne al objeto.

<name>gchar</name>
<name>nombre</name>

Para el color se necesitan tres valores cada uno representando la intensidad de los componentes del modelo RGB, sus valores están en hexadecimal debido a la forma en que lo maneja y lo almacena en la estructura Material3D.

<color>guchar guchar guchar</color>
<color>R G B</color>

En las transformaciones únicamente se necesita tener el valor con el cual se está aplicando la transformación en cada uno de los ejes y el tipo de transformación que se va a aplicar, estas pueden ser ROTATE, TRANSLATE o SCALE.


FALLA DE ORIGEN

<transform type=gchar>gdouble gdouble gdouble</transform>
<transform type=tipo>X Y Z</transform>

Con las marcas anteriores se tiene la manera de representar a los objetos con los atributos que necesitan para ser representados y los parámetros que los afectan, pero falta una marca que presente los vértices de los objetos y su conexión por lo que se puede añadir la siguiente marca.

<M3DObject numVertex=gint numTriangles=gint typeObject=gchar>
Marcas de parámetros
</M3DObject>

De esta manera la marca tiene como atributos el número de vértices, el número de triángulos y el tipo de objeto, este último se refiere a la primitiva a la que pertenece el conjunto de vértices, este atributo no es necesario en las marcas anteriores debido a que ellas mismas indican el tipo de objeto que agregan. En caso de que no tenga un tipo en particular o no se sepa que tipo de objeto es, se debe poner "POLIGON", ya que este tipo de objetos siempre son almacenados con sus vértices.

Los parámetros que manejen estas marcas deben ser los vértices, los triángulos y el nombre (el cual utilizará la misma marca indicada anteriormente). La marca del color no la maneja ya que cada uno de los triángulos contiene su propio color. Para los vértices se tienen 3 parámetros que son X Y Z.

<vertex>gdouble gdouble gdouble</vertex>
<vertex>X Y Z</vertex>

Los triángulos deberán estas debajo de los vértices ya que estos últimos se irán numerando y dentro de esta marca se hará referencia a ellos conforme a su numeración como lo maneja el formato obj (analizado en el capítulo 3) donde el primer vértice es el 1, el segundo el 2 y así sucesivamente, pero en este caso el inicio y el final de la numeración esta delimitada por la marca del objeto, es decir no se tiene una sola numeración para todo el archivo sino que cada marca tiene la suya propia, de esta manera una marca no puede hacer referencia a un vértice declarado en otra distinta por consecuencia cada objeto debe contener su propio conjunto de vértices definidos en su marca correspondiente. Seguido de los 3 vértices que utiliza contiene el color de esta cara, este se maneja de la misma manera que con la marca color.

<triangle>gint gint gint guchar guchar guchar</triangle>
<triangle>V1 V2 V3 R G B</triangle>

Con este pequeño conjunto de marcas se tiene el formato actual de Material3D donde este puede representar todo lo que puede realizar el modelador.

3.3. Instrucciones a futuro.

Hasta el momento solamente se tiene la forma de crear objetos y aplicarles transformaciones, por lo que el formato solo tiene ese tipo de marcas, pero existen algunas partes que aunque no han sido implementadas ya se tienen contempladas como lo son las operaciones que contiene la GSC.

Al igual que estas operaciones, hace falta implantar una cámara y luces ya que solamente se tiene una luz por omisión, así que se diseñarán las instrucciones para estas partes del modelador que aún no se tienen implantadas para que puedan ser agregadas al formato ya diseñado.

Comenzando por la cámara al igual que con los objetos, debemos conocer los atributos y parámetros que esta utiliza para ver qué es lo que necesita llevar su marca. Sabemos que el despliegue se realiza con OpenGL por lo que nos podemos basar en la instrucción que hace el manejo de cámara que contiene, solo que se tiene un problema ya que no es solamente una instrucción quien la afecta sino que son varias por lo que se tomarán los atributos generales poniendo como parámetros otros más específicos.

Sabemos que OpenGL cuenta con la instrucción gluLookAt la cual contiene las coordenada donde está situada la cámara, las coordenadas del punto hacia donde está viendo y el vector que indica la dirección de donde es arriba, estos parámetros son muy importantes para poder implementar una cámara. Como parámetros secundarios se tendrían el tipo de cámara que se utiliza, ya sea ortogonal o en perspectiva por lo que la marca puede quedar de la siguiente manera.

<camera X=gdouble Y=gdouble Z=gdouble>
    Marcas de parámetros
</camera>

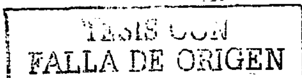
Ya que por omisión la cámara siempre ve hacia el origen y la dirección que indica hacia arriba está sobre el eje Y, no se incluyen en sus atributos; no así su posición pero no quiere decir que no se puedan modificar, para esto se implementan las marcas que se utilizarán como parámetros de la cámara.

Para indicar hacia donde ver, se tiene la siguiente marca.

<lookAt>gdouble gdouble gdouble</lookAt>
<lookAt>X Y Z</lookAt>

Para modificar la dirección de donde es arriba se tiene esta marca que contiene el vector que indica dicha dirección.

<upDirection>gdouble gdouble gdouble</upDirection>
<upDirection>X Y Z</upDirection>



Por último para indicar el tipo de cámara que se utilizará se tendrá la siguiente marca.

```
<camType>gchar</camType>
```

El tipo puede ser "PERSPECTIVE" o un "0", u "ORTOGRAPHIC" o un "1".

Dentro del programa se deberá tener una estructura que almacene estos datos que utiliza la cámara así como se tienen estructura para almacenar los datos de los objetos.

```
#define PERSPECTIVE 0
#define ORTOGRAPHIC 1
typedef struct {
	point3D position, lookAt, upDirection;
	gint camType;
}camera;
```

Ya teniendo la cámara se puede pasar a las luces, estas pueden ser separadas en dos tipos, el punto de luz y la luz de spot, donde esta última contiene los mismos atributos que la primera más otros propios. Para estas no se manejarán tantos parámetros como OpenGL lo permite, sino que simplemente se tomarán los básicos.

Para el punto de luz se necesita su posición como un atributo esencial y su color y atenuación como secundarios.

```
<pointLight X=gdouble Y=gdouble Z=gdouble>
	Marcas de parámetros
</pointLight>
```

Con esto se crea una fuente de luz en una cierta posición, como marca de su parámetro color puede utilizar la que ya pertenece al formato. Para la atenuación se tiene lo siguiente.

```
<attenuation type=gchar>gdouble</attenuation>
```

El tipo puede ser "CONSTANT", "LINEAR" o "QUADRATIC" y como parámetro tiene la constante de atenuación.

Para la luz de spot se tienen los mismos atributos y parámetros pero a parte se tienen la dirección hacia donde va dirigida, la apertura del cono y la intensidad dentro del cono aunque esta última puede ser la misma para todas las luces por lo que puede dejarse a un lado.

```
<spotLight X=gdouble Y=gdouble Z=gdouble cutOff=gdouble
	dirX=gdouble dirY=gdouble dirZ=gdouble>
	Marcas de parámetros
</spotLight>
```

De esta forma se puede crear una luz de spot indicando su posición con X Y Z, su ángulo de apertura con cutOff y la dirección hacia donde ilumina con dirX dirY dirZ. Sus parámetros pueden ser su color y su atenuación como con la luz anterior.

Para las luces pueden utilizar la misma estructura retomando la misma idea de la estructura de los objetos donde los atributos que no pertenecen a un objeto, toman el valor nulo.

```
#define POINT 0
#define SPOT 1
typedef struct {
        gint type;
        point3D position, direction;
        gdouble cutOff, attenuation;
        guchar r, g, b;
}light;
```

Ahora solo faltan las marcas de las operaciones de la GSC, estas básicamente sirven para agrupar objetos y algunas de ellas aplican un cambio a ese grupo. Las operaciones que se tienen pensado implementar en este modelador son la unión, la intersección y la diferencia. Como se mencionó, básicamente agrupan objetos por lo que su contenido son puros objetos. Sus atributos puede ser el tipo de operación, podría añadirse un atributo más que fuera el nombre, este último se podría manejar como parámetro utilizando la marca que ya se tiene en el formato.

```
<M3Dgsc type=gchar>
      Marcas de objetos y nombre.
</M3Dgsc>
```

En el tipo de transformación se indica cuál se va a aplicar con una cadena de caracteres o un entero, las cadenas que serán aceptadas serán "UNION", "INTERSECTION" y "DIFFERENCE". El siguiente pedazo de código muestra el número correspondiente para cada una y la estructura de estas operaciones. Se nota que aparte de tener el arreglo de objetos, su nombre y su tipo también contienen un arreglo de transformaciones, ya que se pueden aplicar al conjunto de objetos que son parte de la operación aplicada, De la misma manera se tiene un arreglo del mismo tipo de la estructura, ya que se puede tener un conjunto de operaciones dentro de otra que las contenga a todas y las pueda trabajar como un mismo objeto.

```
#define UNION 0
#define INTERSECTION 1
#define DIFFERENCE 2
typedef struct {
        gint type;
        gchar *name;
        gint numObjects;
        m3DObject *objectArray;
        gint numTransf;
```

```
        transform *transfArray;
        gint numGSC;
        GSC *gscArray;
}GSC;
```

Con estas nuevas marcas se modifica tanto la estructura como el posible contenido de la marca de la escena, las cuales quedan como sigue.

```
typedef struct {
        gint numObjects;
        m3DObject *objectArray;
        gint numGSC;
        transform *GSCArray;
        gint numLights;
        light *lightArray;
        camera cam;
}m3DScene;
```

Las marcas que puede contener la marca de escena pueden ser de la cámara, luces, objetos o de operaciones de GSC.

Aún quedan algunas partes que se quiere implantar en el modelador pero primero deben estar implantadas para saber cómo debe ser la marca que las identifique, ya que se tienen varias formas de realizarlas como la animación la cual tiene bastantes elementos con lo que se tendrían que ocupar varias marcas distintas solo en este módulo, pero la forma de diseñar dichas marcas seguirá basándose en las estructuras que contienen la información o en la forma de manejar ésta.

CAPÍTULO 6

DISCUCIÓN Y ANÁLISIS DE LOS RESULTADOS

El formato que se ha diseñado tiene las principales ventajas que brindan tanto los formatos de los modeladores 3D y los lenguajes de modelado 3D, ya que por una parte los objetos pueden llegar a ser almacenados con su conjunto de vértices o con las características que forman a cada uno de los objetos.

Al igual que en el formato de 3Dstdio analizado en el capítulo 3 tiene una marca en especial que indica los limites donde está definida la escena, y aunque no se tiene un bloque de definición de objetos cada una de ellas está delimitada por su propia marca. Aun contando con la desventaja de ocupar más espacio de almacenamiento debido a que el formato se guarda en ascii se cuenta con la ventaja de que puede ser revisado por un usuario y entendido con facilidad (debido a que no es muy complicado) sin necesidad de un programa especial que traduzca el archivo.

Otra de las grandes ventajas que podemos tener con este formato es que al estar escrito con XML se le pueden agregar con mayor facilidad nuevos elementos los cuales representen nuevas partes implementadas que contenga el modelador.

6.1. Analizador del formato

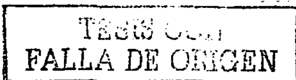
Ahora ya teniendo el formato, se necesita hacer el programa que revise que los archivos estén escritos correctamente y obtener los datos necesarios para poder mostrar los objetos, este programa puede ser considerado como un analizador léxico, sintáctico y semántico para el formato que se diseñó.

A continuación se muestra el código de este analizador, el programa utiliza las bibliotecas de libxml para facilitar la revisión de los archivos; ya que como se vió en el capítulo 2, éstas contienen varias instrucciones para el manejo de archivos de este tipo.

Para obtener los datos del archivo se utilizan otro tipo de estructuras que manejan tipos de datos distintos a las estructuras mostradas en el capítulo anterior, posteriormente estos datos se traspasan a sus estructuras correspondientes, esto es porque los datos que se obtienen del archivo son capturados como xmlChar, así que posteriormente se deberán pasar a su tipo correspondiente.

```
#include <stdio.h>
#include <stdlib.h>
#include <gnome.h>
#include <libxml/xmlmemory.h>
#include <libxml/parser.h>

/* Estructura que almacena temporalmente las transformaciones */
type struct {
	xmlChar *type;
	xmlChar *xyz;
}transform, *transformPtr;
```


TESIS CON
FALLA DE ORIGEN

```
/* Revisa y extrae la información de las transformaciones */
static transformPtr
parseTransform (xmlDocPtr doc, xmlNodePtr nodo) {
 transformPtr tmp = NULL;

 tmp = (transformPtr) malloc (sizeof (transform));
 if (tmp == NULL) {
  fprintf(stderr,"memoria insuficiente\n");
  return(NULL);
 }
 memset(tmp, 0, sizeof(transform));

 tmp->type = xmlGetProp (nodo, (const xmlChar *) "type");
 if (tmp->type == NULL) {
  fprintf(stderr,"falta el tipo de transformación\n");
  return (NULL);
 }
 tmp->xyz = xmlNodeListGetString (doc, nodo->xmlChildrenNode, 1);

 return (tmp);
}

/* Estructura que almacena temporalmente todos los datos del objeto */
typedef struct {
      xmlChar *name;
      xmlChar *numVertex;
      xmlChar *numTriangles;
      xmlChar *typeObject;
      xmlChar *vertexArray[1500];
      xmlChar *trianglesArray[1500];
      xmlChar *numLevels;
      xmlChar *numSides;
      xmlChar *divX;
      xmlChar *divY;
      xmlChar *divZ;
      xmlChar *radio;
      xmlChar *radio2;
      xmlChar *height;
      xmlChar *radioExt;
      gint ct;
      transformPtr transArray[30];
      xmlChar *color;
      gboolean vertex;
}xmlM3DObject, *xmlM3DObjectPtr;

/* Revisa las marcas que contienen las que definen el conjunto de vértices
almacenando los datos donde corresponde */
static xmlM3DObjectPtr
parsePoints3D(xmlDocPtr doc, xmlNodePtr nodo) {
 xmlM3DObjectPtr tmp = NULL;
 gint cv = 0, ct = 0;

 tmp = (xmlM3DObjectPtr) malloc(sizeof(xmlM3DObject));
 if (tmp == NULL) {
  fprintf(stderr,"memoria insuficiente\n");
  return(NULL);
 }
 memset(tmp, 0, sizeof(xmlM3DObject));
```

TESIS CON
FALLA DE ORIGEN

```
 nodo = nodo->xmlChildrenNode;
 while (nodo ¡= NULL) {
  if (¡xmlStrcmp(nodo->name, (const xmlChar *) "vertex")) {
   tmp->vertexArray[cv] = xmlNodeListGetString(doc,
                           nodo->xmlChildrenNode, 1);
   cv++;
  }
  if (¡xmlStrcmp(nodo->name, (const xmlChar *) "triangle")) {
   tmp->trianglesArray[ct] = xmlNodeListGetString(doc,
                              nodo->xmlChildrenNode, 1);
   ct++;
  }
  if (¡xmlStrcmp(nodo->name, (const xmlChar *) "name"))
   tmp->name = xmlNodeListGetString (doc, nodo->xmlChildrenNode, 1);
  if (¡xmlStrcmp(nodo->name, (const xmlChar *) "transform")) {
   tmp->transArray[ct] = parseTransform (doc, nodo);
   ct++;
  }

  nodo = nodo->next;
 }

 return (tmp);
}

/* Revisa las marcas que contienen las que definen los objetos con sus datos
particulares almacenandolos en su lugar correspondiente */
static xmlM3DObjectPtr
parseAttributes(xmlDocPtr doc, xmlNodePtr nodo) {
 xmlM3DObjectPtr tmp = NULL;
 gint ct=0;

 tmp = (xmlM3DObjectPtr) malloc(sizeof(xmlM3DObject));
 if (tmp == NULL) {
  fprintf(stderr,"memoria insuficiente\n");
  return(NULL);
 }
 memset(tmp, 0, sizeof(xmlM3DObject));

 nodo = nodo->xmlChildrenNode;
 while (nodo ¡= NULL) {
  if (¡xmlStrcmp(nodo->name, (const xmlChar *) "color"))
   tmp->color = xmlNodeListGetString (doc,
                  nodo->xmlChildrenNode,1);
  if (¡xmlStrcmp(nodo->name, (const xmlChar *) "name"))
   tmp->color = xmlNodeListGetString (doc,
     nodo->xmlChildrenNode,1);
  if (¡xmlStrcmp(nodo->name, (const xmlChar *) "transform")) {
   tmp->transArray[ct] = parseTransform (doc, nodo);
   ct++;
  }
  nodo = nodo->next;
 }

 tmp->ct = ct;
 return (tmp);
}
```

TESIS CON
FALLA DE ORIGEN

```
/* Revisa las marcas que definen los objetos, si se agregan las otras marcas
propuestas (por ejemplo la cámara) estas también deberán revisarse en esta
función */
static xmlM3DObjectPtr
parseObject (xmlDocPtr doc, xmlNodePtr nodo) {
 xmlM3DObjectPtr tmp = NULL;

 tmp = (xmlM3DObjectPtr) malloc(sizeof(xmlM3DObject));
 if (tmp == NULL) {
  fprintf(stderr,"memoria insuficiente\n");
  return(NULL);
 }
 memset (tmp, 0, sizeof(xmlM3DObject));

 if (¡xmlStrcmp(nodo->name, (const xmlChar *) "M3DObject")) {
  tmp = parsePoints3D(doc, nodo);
  tmp->numVertex = xmlGetProp (nodo, (const xmlChar *) "numVertex");
  tmp->numTriangles = xmlGetProp (nodo,
                              (const xmlChar *) "numTriangles");
  tmp->typeObject = xmlGetProp (nodo,
                              (const xmlChar *) "typeObject");
  tmp->vertex = TRUE;
 }
 if (¡xmlStrcmp(nodo->name, (const xmlChar *) "sphere")) {
  tmp = parseAttributes(doc, nodo);
  tmp->numLevels = xmlGetProp (nodo, (const xmlChar *) "numLevels");
  tmp->numSides = xmlGetProp (nodo, (const xmlChar *) "numSides");
  tmp->radio = xmlGetProp (nodo, (const xmlChar *) "radio");
  tmp->typeObject = nodo->name;
  tmp->vertex = FALSE;
 }
 if (¡xmlStrcmp(nodo->name, (const xmlChar *) "box")) {
  tmp = parseAttributes(doc, nodo);
  tmp->divX = xmlGetProp (nodo, (const xmlChar *) "divX");
  tmp->divY = xmlGetProp (nodo, (const xmlChar *) "divY");
  tmp->divZ = xmlGetProp (nodo, (const xmlChar *) "divZ");
  tmp->typeObject = nodo->name;
  tmp->vertex = FALSE;
 }
 if (¡xmlStrcmp(nodo->name, (const xmlChar *) "cylinder")) {
  tmp = parseAttributes(doc, nodo);
  tmp->numLevels = xmlGetProp (nodo, (const xmlChar *) "numLevels");
  tmp->numSides = xmlGetProp (nodo, (const xmlChar *) "numSides");
  tmp->height = xmlGetProp (nodo, (const xmlChar *) "height");
  tmp->radio = xmlGetProp (nodo, (const xmlChar *) "radio");
  tmp->typeObject = nodo->name;
  tmp->vertex = FALSE;
 }
 if  (¡xmlStrcmp(nodo->name, (const xmlChar *) "torus")) {
  tmp = parseAttributes(doc, nodo);
  tmp->numLevels = xmlGetProp (nodo, (const xmlChar *) "numLevels");
  tmp->numSides = xmlGetProp (nodo, (const xmlChar *) "numSides");
  tmp->radio = xmlGetProp (nodo, (const xmlChar *) "radioExt");
  tmp->radio2 = xmlGetProp (nodo, (const xmlChar *) "radioInt");
  tmp->typeObject = nodo->name;
  tmp->vertex = FALSE;
 }
 if (¡xmlStrcmp(nodo->name, (const xmlChar *) "cone")) {
  tmp = parseAttributes(doc, nodo);
```

TESIS CON
FALLA DE ORIGEN

```
  tmp->numLevels = xmlGetProp (nodo, (const xmlChar *) "numLevels");
  tmp->numSides = xmlGetProp (nodo, (const xmlChar *) "numSides");
  tmp->height = xmlGetProp (nodo, (const xmlChar *) "height");
  tmp->radio = xmlGetProp (nodo, (const xmlChar *) "radio");
  tmp->typeObject = nodo->name;
  tmp->vertex = FALSE;
 }

 return (tmp);
}

/* Estructura temporal que almacena la escena completa */
typedef struct {
      gint numObjs;
      xmlM3DObjectPtr objs[1500];
}xmlM3DScene, *xmlM3DScenePtr;

/* Reserva memoria para el árbol de marcas y comienza a construirlo colocando
en la raiz a la marca <M3DScene> */
static xmlM3DScenePtr
parseScene (char *archivo) {
 xmlDocPtr doc;
 xmlNodePtr nodo;
 xmlM3DScenePtr tmp;

 doc = xmlParseFile (archivo);
 if (doc == NULL) return (NULL);

 nodo = xmlDocGetRootElement (doc);
 if (nodo == NULL) {
  fprintf(stderr,"memoria insuficiente\n");
  xmlFreeDoc (doc);
  return (NULL);
 }

 if (xmlStrcmp(nodo->name, (const xmlChar *) "M3DScene")) {
  fprintf(stderr,"archivo no valido, falta marca de escena\n");
  xmlFreeDoc (doc);
  return (NULL);
 }

 tmp = (xmlM3DScenePtr) malloc (sizeof(xmlM3DScene));
 if (tmp == NULL) {
  fprintf(stderr,"memoria insuficiente\n");
  xmlFreeDoc (doc);
  return (NULL);
 }
 memset (tmp, 0, sizeof(xmlM3DScene));

 nodo = nodo->xmlChildrenNode;
 tmp->numObjs = 0;

 while(nodo != NULL) {
  tmp->objs[tmp->numObjs++] = parseM3DObject(doc, nodo);
  nodo = nodo->next;
 }
}
```

TESIS CON
FALLA DE ORIGEN

Con este programa, el modelador solo deberá llamar a la función parseScene con el nombre del archivo que contiene la escena y devolverá una estructura que contiene todos los datos de la escena, los cuales solamente se deberán separar y asignar a sus campos correspondientes dentro de las estructuras vistas en el capítulo 5.

6.2. Ejemplos del formato

Ya se tiene el formato y el programa que lo analizará, pero ¿cómo se vería una escena escrita con el formato diseñado?. A continuación se analizarán algunos ejemplos dando una descripción de ellos junto con una imagen que representa la forma en que debe verse dicha escena.

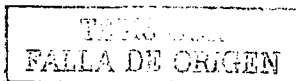
Un primer ejemplo que se puede tener en mente es el clásico cubo, este puede ser representado de dos formas.

```
<M3DScene>
	<box divX=2 divY=2 divZ=2>
		<name>Caja</name>
		<color>230 51 51</color>
	</box>
</M3DScene>
```

Esta es una forma muy simple de definir una caja y como no es afectada por ningún escalamiento, todos sus lados tienen tamaño 1 con lo que se forma el cubo. Una cosa que se debe notar es que todas las marcas están contenidas en la marca <M3DScene> ya que define una escena con este formato y solo se puede tener una marca de este tipo en todo el archivo, además de que ninguna marca puede estar fuera.

Se puede observar que este archivo es muy parecido a un lenguaje de modelado, lo cual hace que se facilite su lectura (cumpliendo de esta manera uno de los objetivos) aparte de no ser tedioso ya que elimina la definición de los vértices y su conexión, los únicos datos utilizados son las divisiones que tendrá a lo largo de cada uno de los ejes. La caja también utiliza otras dos marcas que indican su nombre y su color, estas son opcionales ya que sus valores por omisión son el blanco (255 255 255) en el caso del color y la cadena vacia en el caso del nombre, en el ejemplo el color de la caja es rojo ya que utiliza en modelo RGB donde cada uno de sus valores es la intensidad de cada uno de los componentes.

Esta forma de representar el cubo tiene sus ventajas (ya mencionadas) pero también tiene sus desventajas, esta es la falta de los vértices y su conexión ya que en algunas ocasiones pueden ser requeridos. Por esta razón se tiene la otra forma de representar esta misma escena la cual se muestra a continuación:



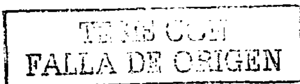
```
<M3DScene>
	<M3DObject numVertex=8 numTriangles=12 typeObject=BOX>
		<name>Caja</name>
		<vertex>0.0 0.0 0.0</vertex>
		<vertex>1.0 0.0 0.0</vertex>
		<vertex>0.0 1.0 0.0</vertex>
		<vertex>1.0 1.0 0.0</vertex>
		<vertex>0.0 0.0 1.0</vertex>
		<vertex>1.0 0.0 1.0</vertex>
		<vertex>0.0 1.0 1.0</vertex>
		<vertex>1.0 1.0 1.0</vertex>
		<triangle>1 2 3 230 51 51</triangle>
		<triangle>2 3 4 230 51 51</triangle>
		<triangle>1 3 5 230 51 51</triangle>
		<triangle>3 7 5 230 51 51</triangle>
		<triangle>1 5 6 230 51 51</triangle>
		<triangle>1 6 2 230 51 51</triangle>
		<triangle>2 6 8 230 51 51</triangle>
		<triangle>2 8 4 230 51 51</triangle>
		<triangle>3 4 8 230 51 51</triangle>
		<triangle>3 8 7 230 51 51</triangle>
		<triangle>5 7 8 230 51 51</triangle>
		<triangle>6 7 8 230 51 51</triangle>
	</M3DObject>
</M3DScene>
```

Con este ejemplo se puede observar que el archivo se vuelve más grande ya que contiene mayor información del objeto que el anterior, pero de la misma manera se vuelve más complicado de leer y escribir, ya que se tienen que estar calculando cada uno de los vértices que contiene la caja y la forma en que están conectados por lo que no es muy recomendable utilizar esta forma de poner un objeto para escribir en un editor de texto una escena ya que los objetos necesitan de una gran cantidad de datos. En el ejemplo se paso de tener solamente tres datos para formar la caja a tener 20 datos para obtener los mismos resultados.

Aún teniendo estos defectos esta forma de representación de los objetos es muy útil ya que de esta forma se pueden mover algunos de sus vértices de posición para tener mayor control del modelado o se puede utilizar para representar la escena con un lenguaje de programación añadiéndole interacción.

A la imagen se le han añadido los ejes para tener una mejor ubicación del objeto donde el eje "X" es de color rojo el "Y" verde y el "Z" azul.

Con esto se puede llegar a la siguiente conclusión, si se va a escribir una escena con este formato en un editor de texto es mejor utilizar la primera forma de representar los objetos ya que no requieren de muchos datos, en cambio si la escena se va a realizar utilizando Material3D entonces no se tiene ningún límite y los objetos pueden ser almacenados de las 2 formas dependiendo de lo que se requiera (lectura fácil o los



vértices y sus conexiones) incluso se pueden almacenar utilizando las dos formas en los distintos objetos que componen la escena. Un asunto importante que se debe notar es que si se mueve solamente un vértice de un objeto, este ya no podrá ser almacenado solo con su descripción, sino que será necesario almacenarlo con su conjunto de vértices y sus conexiones ya que de lo contrario perderá la forma como se modeló y obtendrá nuevamente su forma original (ya que su descripción no guarda la posición de sus vértices).

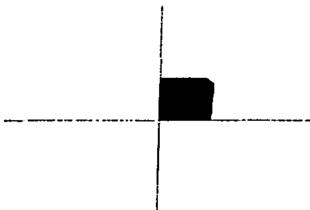


Figura 6.1. Ejemplo de la caja.

En la escena observamos que las caras de la caja están perpendiculares a los ejes correspondientes, se encuentra en el origen y que su tamaño es 1, la escena se ve de esta forma debido a que no se aplicó ninguna transformación y si se añadiera otro objeto sin aplicarle ninguna transformación los dos estarían en la misma posición. Esto es común en muchos modeladores ya que cada vez que se agrega un nuevo objeto a la escena este aparece en el centro, en estos casos se utilizan las transformaciones (rotación, traslación y escalamiento) para poder posicionar a los objetos en un lugar determinado, además de darle cierta orientación y poder cambiar su tamaño para que se pueda dar forma a la escena. En el siguiente ejemplo se tiene el mismo cubo pero con algunas transformaciones aplicadas.

```
<M3DScene>
	<box divX=2 divY=2 divZ=2>
		<name>Caja</name>
		<color>230 51 51</color>
		<transform type=ROTATE>0 45 0</transform>
		<transform type=TRANSLATE>-3 2 3</transform>
		<transform type=SCALE>1 1 2</transform>
	</box>
</M3DScene>
```

En este ejemplo se tiene la misma caja pero con 3 marcas de transformación, cada una es de distinto tipo de las otras con lo que se tiene una rotación sobre el eje "Y" de 45 grados, en los otros ejes no se tiene rotación ya que su ángulo indicado es 0. Para la traslación se indica que se mueva 3 posiciones sobre la parte negativa del eje "X", 2 sobre la parte positiva del eje "Y" así como 3 unidades sobre la parte positiva del eje

"Z". Por último, con el escalamiento se indica que se quiere aplicar únicamente sobre el eje "Z" aumentado su tamaño al doble, a pesar de que los otros 2 ejes tienen un valor de 1 no sufren ninguna transformación ya que este número indica que se quedará de su tamaño actual.

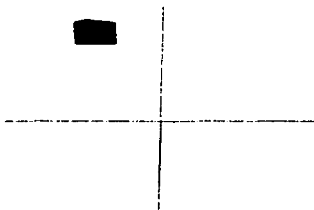


Figura 6.2. Transformaciones.

Sobre esta última transformación se debe tener cuidado de la forma en que se aplica ya que un valor mayor a 1 indica que el objeto aumentará de tamaño como se puede ver en el ejemplo anterior, un valor de 1 indica que el objeto queda del tamaño que tiene y un valor entre 0 y 1 indica que el objeto reducirá su tamaño, pero esta transformación no puede tomar un valor de 0 ya que con este valor el objeto desaparecería por lo que puede provocar un error en el modelador. También se pueden utilizar valores negativos obteniendo el objeto del tamaño indicado si el valor fuera positivo pero la orientación del objeto está como si hubiera sido reflejado por un espejo. El siguiente ejemplo muestra una caja de color rojo con tamaño 1 escalada positivamente para disminuir su tamaño por 0.9 sobre el eje "Y" y una caja azul con los mismos parámetros que la roja con la diferencia de que el valor de escalamiento será negativo, en ambos casos se trasladarán los objetos para separarlos una unidad, a la caja roja sobre el eje "X" positivo y a la azul sobre el mismo eje pero negativo, además se añadirá una rotación sobre el eje "Y" para que se vea realmente el efecto que se obtiene.

```
<M3DScene>
	<box divX=2 divY=2 divZ=2>
		<name>Caja 1</name>
		<color>230 51 51</color>
		<transform type=SCALE>0.9 1 1</transform>
		<transform type=ROTATE>0 45 0</transform>
		<transform type=TRANSLATE>1 0 0</transform>
	</box>
	<box divX=2 divY=2 divZ=2>
		<name>Caja 2</name>
		<color>51 51 230</color>
		<transform type=SCALE>-0.9 1 1</transform>
		<transform type=ROTATE>0 45 0</transform>
```


TESIS CON
FALLA DE ORIGEN

```
        <transform type=TRANSLATE>-1 0 0</transform>
    </box>
</M3DScene>
```

Primero se puede ver la forma de añadir más de un objeto a una escena, pero el ejemplo se enfoca en ver los efectos del escalamiento negativo que se muestran sobre la caja azul (la segunda definida). En la figura 6.3 se observa cómo la caja azul tiene las mismas dimensiones que la roja pero su orientación es distinta gracias a su escalamiento, otra cosa que se nota es que los valores de las transformaciones también se invierten por lo que al aplicar una traslación negativa ésta se aplica hacia el lado positivo. Este escalamiento se presta a utilizarse para añadir un efecto como de reflejo en un espejo.

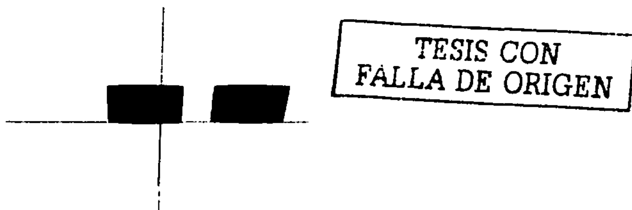

TESIS CON
FALLA DE ORIGEN

Figura 6.3. Escalamiento negativo.

Con estos ejemplos se vieron la mayoría de las instrucciones del formato que actualmente puede soportar Material3D, solo faltaron las otras primitivas gráficas que maneja el modelador, pero su definición es tan parecida a la caja (en lo único que cambia es en los parámetros de cada una de estas) que no se verá un ejemplo para cada una de estas.

A continuación se verán unos ejemplos en los cuales se utilizan las instrucciones que están planeadas a futuro para poder ver como se deberán manejar cuando estén disponibles. Para los ejemplos se utilizarán las instrucciones que utilizan la descripción del objeto, ya que hasta los objetos más simples están compuestos de varios vértices y esto provocaría que los ejemplos no queden completamente claros al ser bastante largos.

Primero se tiene que ver como poner una cámara y una luz, hay que recordar que solamente se puede añadir una sola cámara, en el caso de las luces se podrán tener cualquier cantidad. En el siguiente ejemplo se tiene la cámara, una luz y una caja para ver sus efectos.

```
<M3DScene>
    <camera X=0 Y=3 Z=-5>
        <lookAt>0 1 0</lookAt>
        <upDirection>0 1 0<upDirection>
```

```
        <camType>PERSPECTIVE</camType>
    </camera>
    <pointLight X=-1 Y=5 Z=-5>
        <color>255 255 255</color>
        <attenuation type= LINEAR >1.0</attenuation>
    </pointLight>

    <box divX=2 divY=2 divZ=2>
        <name>Caja</name>
        <color>255 51 51</color>
        <transform type=SCALE>2 2 2</transform>
    </box>
</M3DScene>
```

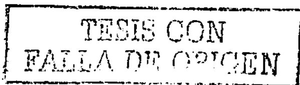
En este ejemplo lo primero que se ve es la cámara, ésta contiene su posición, hacia donde mira y la dirección de dónde es arriba (esta se da con un vector), además se tiene el tipo de cámara a utilizar (en este caso se tiene una vista en perspectiva). Es fácil notar que los parámetros indicados para la cámara son los que utiliza OpenGL para poder ingresar una cámara en una escena ya que el formato fue diseñado de esta manera para que se facilite la recolección de datos que se necesitan para realizar el despliegue.

Posteriormente se tiene un punto de luz, hay que notar que se tienen dos tipos de luz, una es esta y la otra es la luz de spot que se verá más adelante. Con la luz definida en el ejemplo se tiene su posición, su color y su atenuación, esta última contiene el tipo y una constante que servirá para ver que tanto se va atenuando la luz conforme se está más lejos de ella. Como la luz fue definida como un punto, alumbra hacia todas direcciones hasta que es obstruida por algún objeto dentro de la escena. Como se puede notar la sintaxis del archivo sigue siendo clara, concisa y muy completa ya que se pueden definir todos los datos necesarios para lograr los efectos, modelos y escenas deseados. La imagen que define este archivo se puede observar en la figura 6.4.

Figura 6.4. *Cámara en perspectiva y punto de luz*

TESIS CON FALLA DE ORIGEN

Ya se vio el efecto de una cámara en perspectiva, ahora se verá una cámara ortográfica, como ya se explicó estos dos tipos de cámaras se ven distinto ya que la primera da un efecto a los objetos con lo que se ve que tienen un volumen y la segunda


TESIS CON FALLA DE ORIGEN

al meter a toda la escena en una especie de cubo (alargado o no) quita gran parte de ese efecto.

```
<M3DScene>
	<camera X=0 Y=5 Z=-5>
		<lookAt>0 1 0</lookAt>
		<upDirection>0 1 0<upDirection>
		<camType>ORTOGRAPHIC</camType>
	</camera>
	<pointLight X=-1 Y=5 Z=-5>
		<color>255 255 255</color>
		<attenuation type= LINEAR >1.0</attenuation>
	</pointLight>

	<box divX=2 divY=2 divZ=2>
		<name>Caja</name>
		<color>255 51 51</color>
		<transform type=SCALE>2 2 2</transform>
	</box>
</M3DScene>
```

En este ejemplo se tiene la misma escena pero ahora con la cámara ortográfica por lo que la escena no se ve igual como se muestra en la siguiente figura.

Ahora se verá un ejemplo con una luz de spot la cual solamente ilumina hacia una dirección en un área delimitada por un ángulo.

```
<M3DScene>
	<camera X=0 Y=5 Z=-5>
		<lookAt>0 1 0</lookAt>
		<upDirection>0 1 0<upDirection>
		<camType>PERSPECTIVE</camType>
	</camera>
	<spotlight X=-1 Y=8 Z=-5 cutOff=7 dirX=1 dirY=1 dirZ=0>
		<color>255 255 255</color>
		<attenuation type= LINEAR >10.0</attenuation>
	</spotLight>
	<box divX=2 divY=2 divZ=2>
		<name>Caja</name>
		<color>255 51 51</color>
		<transform type=SCALE>2 2 2</transform>
	</box>
</M3DScene>
```

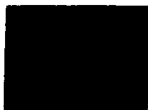


Figura 6.5. Cámara ortográfica.

Como se puede ver, se tiene la misma escena pero ahora con una luz de spot, ésta define su posición, el ángulo de apertura, hacia que punto alumbra, su color y la atenuación. Al igual que con la otras marcas (como la cámara), ésta tiene todos los datos que necesita OpenGL para poder agregar una luz de este tipo en un programa y a pesar de que contiene toda esta información, el archivo aún continua siendo simple. En la siguiente figura se muestra la luz de spot definida en este ejemplo.

TESIS CON
FALLA DE ORIGEN

Figura 6.6. Luz de spot.

Por último se tienen el conjunto de operaciones definidas con la GSC, como se mencionó en el capítulo anterior, las únicas operaciones que se podrán aplicar son la unión, la intersección y la diferencia. Estas deben contener al conjunto de objetos a los cuales se afecta junto con las transformaciones aplicadas a la operación, las cuales se aplican a todos los objetos que contenga.

Primeramente se tiene la unión, como se explicó en el primer capítulo, ésta no afecta a los objetos en su forma, solo los agrupa para poder manejarlos como uno sólo, esto se muestra con el siguiente ejemplo.

```
<M3DScene>
	<camera X=0 Y=3 Z=-4>
		<lookAt>0 0 0</lookAt>
		<upDirection>0 1 0<upDirection>
		<camType>PERSPECTIVE</camType>
	</camera>
	<pointLight X=-1 Y=5 Z=-5>
		<color>255 255 255</color>
```

```
        <attenuation type= LINEAR >1.0</attenuation>
    </pointLight>

    <M3Dgsc type=UNION>
        <box divX=2 divY=2 divZ=2>
            <name>Caja1</name>
            <color>255 51 51</color>
            <transform type=SCALE>2 2 2</transform>
        </box>
        <box divX=2 divY=2 divZ=2>
            <name>Caja2</name>
            <color>51 51 255</color>
            <transform type=SCALE>0.3 0.3 3</transform>
            <transform type=TRANSLATE>1 1 –0.5</transform>
        </box>
        <transform type=ROTATE>70 0 0</transform>
    </M3Dgsc>
</M3DScene>
```

Aquí se puede ver cómo las cajas son agrupadas y manejadas como un mismo objeto ya que al aplicar la transformación sobre la operación las dos giran el mismo ángulo sobre el mismo punto quedando como un solo objeto, pero también pueden ser manejadas como objetos individuales al aplicar las transformaciones dentro de la definición del objeto ya que el escalamiento de la primer caja no afecta en lo absoluto a la otra y viceversa con las transformaciones que tiene definidas. En la siguiente figura se muestra la imagen que genera esta escena.

TESIS CON FALLA DE ORIGEN

Figura 6.7. Union.

Para aplicar una diferencia o una intersección simplemente se cambia el tipo de operación que se va a realizar, la forma en que se verían, se muestra en la figura 6.8 donde la de la izquierda es la diferencia y la de la derecha es la intersección. Para estos ejemplos la rotación que afecta a la operación es de 40 grados sobre el mismo eje.

a) b)

Figura 6.8. a)Diferencia e b)Intersección.

Se puede notar algo extraño al aplicar estas operaciones, esto es porque están diseñadas para trabajar con objetos sólidos y no con polígonos huecos, la representación correcta de estas operaciones debe ser como se muestra en la siguiente figura.

a) b)

Figura 6.9. a)Diferencia e b)Intersección en sólidos.

Aquí observamos que los objetos están rellenos en su interior por lo que esa parte también se toma en cuenta para poder aplicar la operación, por lo que Material3D deberá trabajar con polígonos rellenos para que tenga sentido tener este tipo de operaciones dentro del modelador.

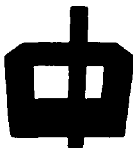
Ya vistos todos estos ejemplos solamente falta ver como se puede tener una operación dentro de otra operación, para esto se tendrá la suposición de que los polígonos están rellenos como muestra la figura anterior. Para el ejemplo se tomará el siguiente conjunto de marcas.

```
<M3DScene>
	<camera X=0 Y=3 Z=-4>
		<lookAt>0 0 0</lookAt>
		<upDirection>0 1 0<upDirection>
		<camType>PERSPECTIVE</camType>
	</camera>
	<pointLight X=-1 Y=5 Z=-5>
		<color>255 255 255</color>
		<attenuation type= LINEAR >1.0</attenuation>
	</pointLight>
```

TESIS CON
FALLA DE ORIGEN

```
<M3Dgsc type=UNION>
        <M3Dgsc type=DIFFERENCE>
                <box divX=2 divY=2 divZ=2>
                  <name>Caja1</name>
                  <color>255 51 51</color>
                  <transform type=SCALE>2 2 2</transform>
                </box>
                <box divX=2 divY=2 divZ=2>
                  <name>Caja1</name>
                  <transform type=SCALE>1.6 1.6 3</transform>
                  <transform type=TRANSLATE>0.2 0.2 -0.2</transform>
                </box>
        </M3Dgsc>
        <box divX=2 divY=2 divZ=2>
                <name>Caja2</name>
                <color>51 51 255</color>
                <transform type=SCALE>0.3 4 0.3</transform>
                <transform type=TRANSLATE>1 -1 1</transform>
        </box>
    </M3Dgsc>
</M3DScene>
```

En este ejemplo se puede observar que la unión esta compuesta por una diferencia entre dos cajas y una caja más a través de ellas, de esta manera se puede llegar a construir modelos o escenas más complejas. Esta escena se puede ver en la figura 6.10, una cosa que se puede notar es que la caja2 no es necesario que lleve la marca de color ya que solamente se utilizará para remover el área en donde se intercepte con la caja1.



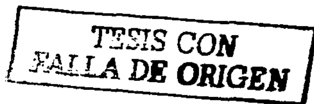


Figura 6.10. *Contención de operaciones por operaciones.*

Por último se puede ver una de las grandes ventajas que tiene el utilizar la definición de los objetos usando vértices ya que de esta manera cada una de las caras puede tener un color distinto a las otras, este efecto no es posible lograrlo utilizando la definición del objeto por su descripción como se ha estado utilizando para los ejemplos.

El siguiente ejemplo muestra una pirámide donde cada una de sus caras tiene distinto color.

```
<M3DScene>
        <M3Dobject numVertex=4 numTriangles=4 typeObject=POLYGON>
                <name>Piramide</name>
                <vertex>-2 0 -1</vertex>
                <vertex>2 0 -1</vertex>
                <vertex>0 0 1</vertex>
                <vertex>0 3 0</vertex>
                <triangle>1 2 3 0 0 255</triangle>
                <triangle>1 4 2 120 120 120</triangle>
                <triangle>1 3 4 255 0 0</triangle>
                <triangle>2 4 3 0 255 0</triangle>
                <transform type=ROTATE>-30 10 0</transform>
        </M3DObject>
</M3DScene>
```

Aquí se puede ver como a un objeto definido por sus vértices también se le pueden aplicar transformaciones, además de que se pueden definir objetos distintos a las primitivas que maneja el modelador (como la del ejemplo). La pirámide se rotó para que se pudieran observar 3 de sus caras ya que el objeto se ve de frente, ésta se muestra en la siguiente figura.

Figura 6.11. Pirámide.

Es importante notar que este ejemplo contiene solamente instrucciones que puede soportar actualmente el modelador

Al analizar estos ejemplos se ha podido notar que el formato diseñado ha cumplido con los objetivos planteados ya que al estar escrito en XML se le pueden agregar nuevas marcas para representar las implementaciones que se le vayan añadiendo (como el módulo de animación) y al estar escrito con este lenguaje se facilita su lectura.

Una característica muy importante es que puede ser manejado como formato de un modelador 3D al hacer uso de las marcas de vértices y triángulos, o ser usado como lenguaje de modelado utilizando las marcas de las primitivas de los objetos, pero además se pueden mezclar estas dos características en una misma escena sin problemas.

En el apéndice A se tiene un programa que lee una escena en 3DS y lo transforma a un archivo con el formato diseñado, este archivo debe tener extensión m3d. La forma de utilizarlo es poner después del nombre del programa el nombre del archivo 3ds a

convertir y el segundo argumento que es opcional es el nombre del archivo donde se quiere almacenar la escena con el nuevo formato. El programa fue realizado utilizando el lenguaje C estándar con lo que puede ser compilado en varias plataformas sin tener que realizar cambios en el código fuente.

CONCLUSIONES

Su conjunto de características hacen ver que el formato diseñado es realmente óptimo para el modelador debido a que puede trabajar con el conjunto total de los datos que maneja el modelador además de poder añadir instrucciones para nuevas funciones que se puedan implantar sin crear conflictos enriqueciendo al formato, con estas nuevas marcas se tiene la posibilidad de implementar operaciones de la GSC como ya se analizo en este capítulo.

La forma de definir un objeto por medio de sus vértices brinda la oportunidad de poder modificar al objeto trasladándolos de su posición original a nuevas posiciones dentro del espacio de la escena, pero también permite exportar de una manera muy sencilla las escenas realizadas con este formato a otros formatos o poder leer sus componentes para manejarlos en un programa donde se le agregue interacción.

En conclusión, este es un formato poderoso por las características antes mencionadas que permite prescindir del modelador para crear o modificar una escena, pero además se pueden añadir objetos o modificarlos e incluso eliminarlos de una manera muy simple editando una escena ya creada con el modelador gracias a su simplicidad, que además hace que el formato sea comprensible y se aprenda rápidamente.

TESIS CON
FALLA DE ORIGEN

LITERATURA CITADA.

[1]. Análisis y diseño de un sistema de modelado y animación 3D GPL
Marcelo Pérez Medel
UNAM, 2002.
[2]. Introducción a la graficación por computador.
J. Foley, Avan Dam, S. K. Feiner, J. F Hughes, R. L. Philips
Addison Wesley, Segunda edición, 1996.
[3]. Computer graphics. A programing aproach.
Steven Harrington
Mc. Graw Hill, International student edition, 1983.
[4]. Fast Algorithms for 3D-Graphics.
Georg Glaeser
Springer-Verlag, 1994.
[5]. Gráficas por computadora.
Donald Hearn, M. Pauline Baker
Prentice Hall Hispanoamericana, segunda edición, 1995.
[6]. Manual de XML.
C. F. Goldfarb, P. Prescod
Prentice Hall, 1999.
[7]. Applied XML: A toolkit for programmers.
Alex Ceponkus, Faraz Hoodbhoy
Wiley & Sons Inc, 1999.
[8]. XML con ejemplos
Benoît Marchal
Pearson Educación de México, primera edición, 2001.
[9]. OpenGL Programing Guide.
M. Woo, J. Neider, T. Davis, D. Shreiner
Addison Wesley, Tercera edición. 2000.
[10]. 3D grphics programing with OpenGL.
Clayton Walnum
Que Corporation, primera edición, 1995.
[11]. VRML biblioteca del programador.
Kris Jamsa, Phil Schmauder, Nelson Yee
McGraw-Hill/Interamericana de México, 1998.
[12]. VRML para internet.
Mark Pesce
Prentice-Hall Hispanoamericana, 1996

TESIS CON
FALLA DE ORIGEN

REFERENCIAS.

Tutorial de libxml.
http://cs1.mcm.edu/tutorial/doc/libxml-1.8.10/html/

Página de libxml.
http://xmlsoft.org

Tutorial de XML.
http://html.programacion.net/xml/principal.htm

Tutorial de OpenGL.
http://odra.dcs.fi.uva.es/area2/opengl/index2.html

Tutorial de OpenGL.
http://www.iua.upf.es/~aramirez/docencia/infografia/

Especificación del formato 3ds.
http://astronomy.swin.edu.au/~pbourke/geomformats/3ds/

Especificación del formato obj.
http://astronomy.swin.edu.au/~pbourke/geomformats/obj/

Especificación de los formatos 3ds y obj, y tutorial de POV-RAY.
http://www.wotsit.org/default.asp

Página oficial de POV-RAY.
http://www.povray.org

Tutorial de POV-RAY
http://astronomy.swin.edu.au/~pbourke/geomformats/pov/pov31

Referencias de POV-RAY.
http://www.3dlinks.com/tutorials_povray.cfm

Tutorial de VRML.
http://astronomy.swin.edu.au/~pbourke/geomformats/vrml1/

Tutorial de VRML.
http://astronomy.swin.edu.au/~pbourke/geomformats/vrml97/

Tutorial de VRML.
http://vrml.org/VRML1.0/vrml10c.html

Tutorial de VRML.
http://www.web3d.org/technicalinfo/specifications/vrml97/index.html

TESIS CON
FALLA DE ORIGEN

Referencias de VRML.
http://www.3dlinks.com/tutorials_vrml.cmf

Nota: Las referencias enunciadas aquí pueden no contener la información utilizada (ya sea que muestren otro tipo de información o que la referencia ya no exista) debido al constante cambio de la información que se maneja en internet.

TESIS CON
FALLA DE ORIGEN

APÉNDICE A

Convertidor de formato 3Dstudio a Material3D

TESIS CON
FALLA DE ORIGEN

3DS2M3D_v1.0.h

```
typedef struct
{
 float x, y, z;
}M3DVert;

typedef struct
{
 unsigned short v1, v2, v3;
 float r, g, b;  /* De 0.0 (minimo) a 1.0 (maximo) */
}M3DTriang;

typedef struct
{
 unsigned int numVertices;
 unsigned int numTriangs;
 M3DVert vertArray[1500];
 M3DTriang triangArray[1500];
 char name[20];
}M3DObj;

typedef struct
{
 unsigned int numObjs;
 M3DObj objetos[1500];
}M3DScene;
```

3DS2M3D_v1.0.c

```
#include <stdio.h>
#include <string.h>
#include "3ds2m3d_v1.0.h"

M3DScene escena;

FILE *fin, *fout;

unsigned short chunkID()
{
 unsigned short id;
```

TESIS CON
FALLA DE ORIGEN

```
 fread(&id,2,1,fin);

 return (id);
}

unsigned short chunkTam(int *tam2)
{
 unsigned short tam1;

 fread(&tam1,2,1,fin);
 fread(&tam2,2,1,fin);

 return(tam1);
}

void chunkVert()
{
 unsigned short numVert, cont;
 float coord;

 fread(&numVert,2,1,fin);
 escena.objetos[escena.numObjs].numVertices = numVert;

 for(cont=0; cont<numVert; cont++)
 {
  fread(&coord,4,1,fin);
  escena.objetos[escena.numObjs].vertArray[cont].x = coord;
  fread(&coord,4,1,fin);
  escena.objetos[escena.numObjs].vertArray[cont].y = coord;
  fread(&coord,4,1,fin);
  escena.objetos[escena.numObjs].vertArray[cont].z = coord;
 }

}

void chunkTrg()
{
 unsigned short numCaras, a, bandera;
 unsigned long tam;
 int x;

 fread(&numCaras,2,1,fin);
 escena.objetos[escena.numObjs].numTriangs=numCaras;
 for (x=0; x<numCaras; x++)
 {
  fread(&a,2,1,fin);
  escena.objetos[escena.numObjs].triangArray[x].v1 = a;
  fread(&a,2,1,fin);
```

TESIS CON
FALLA DE ORIGEN

```
    escena.objetos[escena.numObjs].triangArray[x].v2 = a;
    fread(&a,2,1,fin);
    escena.objetos[escena.numObjs].triangArray[x].v3 = a;
    fread(&bandera,2,1,fin);
   }
  a=chunkID();
  while((a == 0x4130) || (a == 0x4150))
   {
    fread(&tam,4,1,fin);
    fseek(fin, tam-6, SEEK_CUR);
    a=chunkID();
   }

  fseek(fin, -2, SEEK_CUR);

 }

 /*Veritces y triangulos*/
 void chunkMallaTrg(int *aux1, int *aux2)
 {
  unsigned short id;
  int tam1, tam2;

  id = chunkID();
  tam1 = chunkTam(&tam2);

  switch (id)
  {
   case 0x4110: chunkVert();
              break;
   case 0x4120: chunkTrg();
              break;
   default: fseek(fin, tam1 - 6, SEEK_CUR);
           break;
  }

  *aux1 = *aux1 - tam1;
  *aux2 = *aux2 - tam2;

  if(*aux1 < 0)
  {
   *aux2 = *aux2 - 1;
   *aux1 = 0xFFFF + *aux1;
  }
 }

void chunkObj()
{
```

TESIS CON
FALLA DE ORIGEN

```
 unsigned short id;
 int tam1=0, tam2;
 char c,objName[20];

 do {
  fread(&c,sizeof(char),1,fin);
  id = c;

  objName[tam1]=c;

  tam1++;
 } while (id != 0x0);
 objName[tam1]='\0';

 id = chunkID();
 tam1 = chunkTam(&tam2);
 while(((id != 0x4100) && (id != 0x4700) && (id != 0x4600))
   && (!feof(fin)))
 {
  fseek(fin, tam1 - 6, SEEK_CUR);
  id = chunkID();
  tam1 = chunkTam(&tam2);
 }

 if(id == 0x4100)
 {
  tam1-=6;
  if (tam1 < 0)
  {
   tam2-=1;
   tam1=0xFFFF+tam1;
  }

  do {
   do {
    chunkMallaTrg(&tam1, &tam2);
   } while(tam1 > 0);
  } while( tam2 > 0);

  strcpy(escena.objetos[escena.numObjs].name,objName);
  escena.numObjs++;
 }
if(id == 0x4700)
{
 fseek(fin, tam1 - 6, SEEK_CUR);
}
if(id == 0x4600)
{
```

TESIS CON
FALLA DE ORIGEN

```
  fseek(fin, tam1 - 6, SEEK_CUR);
 }

 if (feof(fin))
 {
  printf("ERROR: Fin de archivo irregular\n");
 }

}

void chunkGen()
{
 unsigned short id;
 int tam1, tam2;

 id = chunkID();
 tam1 = chunkTam(&tam2);

 switch (id)
 {
  case 0x4D4D: break;
  case 0x3D3D: break;
  case 0x4000: chunkObj();
            break;
  default: fseek(fin, tam1 - 6, SEEK_CUR);
          break;
 }
}

void guardaEsc()
{
 int c,d;

 fprintf(fout,"<M3DScene numObjs=%d>\n\n",escena.numObjs);

 for(c=0;c<escena.numObjs;c++)
 {
  fprintf(fout," <M3DObject numVertex=%d numTriangles=%d
typeObject=POLYGON>\n",
                 escena.objetos[c].numVertices,
                 escena.objetos[c].numTriangs);
  fprintf(fout,"  <name>%s</name>\n",escena.objetos[c].name);
  for(d=0; d < escena.objetos[c].numVertices; d++)
   fprintf(fout,"  <vertex>%f %f %f</vertex>\n",
                 escena.objetos[c].vertArray[d].x,
                 escena.objetos[c].vertArray[d].y,
                 escena.objetos[c].vertArray[d].z);
  for(d=0; d < escena.objetos[c].numTriangs; d++)
```

TESIS CON
FALLA DE ORIGEN

```
   fprintf(fout,"  <triangle>%d %d %d 0.9 0.9 0.9</triangle>\n",
                escena.objetos[c].triangArray[d].v1,
                escena.objetos[c].triangArray[d].v2,
                escena.objetos[c].triangArray[d].v3);
  fprintf(fout," </M3DObject>\n\n");
 }

 fprintf(fout,"</M3DScene>");
}

int main(int argc,char **argv)
{
 int x=1;
 unsigned short c;
 char m3df[25];

 escena.numObjs = 0;

 if (argc == 3)
  strcpy(m3df,argv[2]);
 else
  strcpy(m3df,"escene.m3d");

 if (argc > 3)
 {
  printf("ERROR: Demaciados argumentos, solo debe ir el archivo 3ds a\n");
  printf("convertir, y opcionalmente el archivo donde escribir con extencion m3d\n");
 }

 if ((fin=fopen(argv[1],"rb")) == NULL)
 {
  printf("\nERROR: No se puede leer el archivo de entrada\n");
  exit (0);
 }
 if ((fout=fopen(m3df,"w")) == NULL)
 {
  printf("\nERROR: No pude crear el archivo de salida\n");
  exit (0);
 }

 fread(&c,1,1,fin);
 while(!feof(fin))
 {
  fseek(fin, -1, SEEK_CUR);
  chunkGen();
  fread(&c,sizeof(char),1,fin);
  x++;
 }
```

TESIS CON
FALLA DE ORIGEN

```
 guardaEsc();

 fclose(fin);
 fclose(fout);

 printf("La escena en formato M3D se a guardado en el archivo %s\n",m3df);

 return 0;
}
```

TESIS CON
FALLA DE ORIGEN

APÉNDICE B

CHUNKS DE 3DSTDIO

TESIS CON
FALLA DE ORIGEN

Bloques de modelado y generales.

Chunk ID: 0x4D4D
Nombre: chunk principal
Nivel: 0
Tamaño: 6 + sub-chunks
Padre: no tiene
Datos: no tiene
Descripción: Contiene a todos los demás chunks y debe estar localizado en los primeros bytes del archivo.

Chunk: 0x0002
Nombre: Versión de 3DS
Nivel: 1
Tamaño: 10
Padre: 0x4D4D (chunk principal)
Datos: dword versión
Descripción: Indica la versión de 3Dstudio que se está utilizando.

Chunk ID: 0x3D3D
Nombre: Editor de 3Dstudio
Nivel: 1
Tamaño: 6 + sub-chunks
Padre: 0x4D4D (chunk principal)
Datos: no tiene
Descripción: Da inicio a la edición de la escena.

Chunk ID: 0x0100
Nombre: Una unidad
Nivel: 2
Tamaño: 10
Padre: 0x3D3D (Editor de 3Dstudio)
Datos: float una_unidad
Descripción: Define un flotante.
Chunk ID: 0x1100
Nombre: Mapa de bits de fondo
Nivel: 2
Tamaño: 6 + variante
Padre: 0x3D3D (Editor de 3Dstudio)
Datos: strz Nombre
Descripción: Contiene el nombre del mapa de bits que estará de fondo aun si no se utiliza.

Chunk ID: 0x1101
Nombre: Usar mapa de bits de fondo
Nivel: 2

Tamaño: 6
Padre: 0x3D3D (Editor de 3Dstudio)
Datos: BOOLEAN
Descripción: Si está presente indica que si se utilizará el mapa de bits de fondo, su ausencia indica lo contrario.

Chunk ID: 0x1200
Nombre: Color de fondo
Nivel: 2
Tamaño: 6 + sub-chunks
Padre: 0x3D3D (Editor de 3Dstudio)
Datos: no tiene
Descripción: Indica el color del fondo aun si no se utiliza, el color está dado por sus respectivos sub-chunks los cuales generalmente son:
    0x0010 chunk de color RGB flotante
    0x0013 chunk de color de corrector de gama RGB flotante

Chunk ID: 0x1201
Nombre: Usar color de fondo
Nivel: 2
Tamaño: 6
Padre: 0x3D3D (Editor de 3Dstudio)
Datos: BOOLEAN
Descripción: Si está presente indica que si se utilizará el color del fondo, su ausencia indica lo contrario.

Chunk ID: 0x1300
Nombre: Colores gradientes del fondo
Nivel: 2
Tamaño: 4 + sub-chunks
Padre: 0x3D3D (Editor de 3Dstudio)
Datos: float posición_del_gradiente
Descripción: Indica el color del gradiente del fondo aun si no se utiliza, el flotante da la posición de los tres colores gradientes, este indica la posición del color medio (que está entre 0.0 y 1.0). Este utiliza tres sub-chunks de color para indicar el color de cada uno de los gradientes.

Chunk ID: 0x1301
Nombre: Usar los colores gradientes del fondo
Nivel: 2
Tamaño: 6
Padre: 0x3D3D (Editor de 3Dstudio)
Datos: BOOLEAN
Descripción: Si está presente indica que se utilizarán los colores gradientes del fondo, su ausencia indica lo contrario.

Chunk ID: 0x1400
Nombre: inclinación del mapa de sombras
Nivel: 2
Tamaño: 10
Padre: 0x3D3D (Editor de 3Dstdio)
Datos: float inclinación
Descripción: Da la inclinación del mapa de sombras.

Chunk ID: 0x1420
Nombre: Tamaño del mapa de sombras
Nivel: 2
Tamaño: 8
Padre: 0x3D3D (Editor de 3Dstdio)
Datos: word tamaño
Descripción: Indica el tamaño del mapa de sombras.

Chunk ID: 0x1450
Nombre: rango de muestra del mapa de sombras
Nivel: 2
Tamaño:10
Padre: 0x3D3D (Editor de 3Dstdio)
Datos: float rango
Descripción: Indica el rango de muestra del mapa de sombras.

Chunk ID: 0x1460
Nombre: Inclinación del trazado de rayos
Nivel: 2
Tamaño: 10
Padre: 0x3D3D (Editor de 3Dstdio)
Datos: float inclinación
Descripción: Indica la inclinación del trazado de rayos.

Chunk ID: 0x1470
Nombre: Usar trazado de rayos
Nivel: 2
Tamaño: 6
Padre: 0x3D3D (Editor de 3Dstudio)
Datos: BOOLEAN
Descripción: Si está presente indica que se utilizará el trazado de rayos, su ausencia indica lo contrario.

Chunk ID: 0x4000
Nombre: Bloque de objeto
Nivel: 2
Tamaño: 6 + variante + sub-chunks
Padre: 0x3D3D (Editor de 3Dstdio)
Datos: strz nombre

TESIS CON
FALLA DE ORIGEN

Descripción: Inicia la definición de un objeto, la cadena es el nombre del objeto.

Chunk ID: 0x4010
Nombre: Ocultar objeto
Nivel: 3
Tamaño: 6
Padre: 0x4000 (Bloque de objeto)
Datos: BOOLEAN
Descripción: Si está presente indica que el objeto debe ocultarse, su ausencia indica lo contrario.

Chunk ID: 0x4012
Nombre: Objeto sin modelar
Nivel: 3
Tamaño: 6
Padre: 0x4000 (Bloque de objeto)
Datos: BOOLEAN
Descripción: Si está presente indica que el objeto no tiene cambios, su ausencia indica lo contrario.

Chunk ID: 0x4013
Nombre: Material del objeto
Nivel: 3
Tamaño: 6
Padre: 0x4000 (Bloque de objeto)
Datos: BOOLEAN
Descripción: Si está presente indica si el objeto tiene asignado un material, su ausencia indica lo contrario.

Chunk ID: 0x4015
Nombre: Activar proceso externo
Nivel: 3
Tamaño: 6
Padre: 0x4000 (Bloque de objeto)
Datos: BOOLEAN
Descripción: Su presencia indica que si existe algún proceso externo este si influirá sobre el objeto, su ausencia indica lo contrario.

Chunk ID: 0x4017
Nombre: El objeto no recibe sombras
Nivel: 3
Tamaño: 6
Padre: 0x4000 (Bloque de objeto)
Datos: BOOLEAN
Descripción: Si está presente indica que las sombras no influyen sobre el objeto, su ausencia indica lo contrario.

Chunk ID: 0x4100
Nombre: Malla triangular
Nivel: 3
Tamaño: 6 + sub-chunks
Padre: 0x4000 (Bloque de objeto)
Datos: No tiene
Descripción: Inicia el bloque que definirá una malla hecha a base de triangulos.

Chunk ID: 0x4110
Nombre: Lista de vértices
Nivel: 4
Tamaño: 6 + variante
Padre: 0x4100 (Malla triangular)
Datos: word número_vértices

(Para cada vértice)
vector posición
...

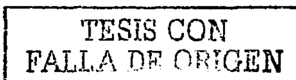
Descripción: Es la lista de los vértices que compondrán la malla, la variable tipo word indica el número de vértices y las variables tipo vector indican la posición de cada uno de los vértices. El número de variables de tipo vector está dado por el número de vértices indicado en la variable de tipo word.

Chunk ID: 0x4120
Nombre: Descripción de caras
Nivel: 4
Tamaño: 6 + variante + sub-chunks
Padre: 0x4100 (Malla triangular)
Datos: word número_caras

(Para cada cara)
word vértice_esquina_A (Referencia numérica)
word vértice_esquina_B (Referencia numérica)
word vértice_esquina_C (Referencia numérica)
word bandera_de_cara

Descripción: Contiene la conexión de cada vértice formando así las caras del polígono (lógicamente son 3 vértices por cara ya que son triangulares). De sus datos el primer tipo word contiene el número de caras, los tres siguientes contienen el número de los vértices que utilizara como esquinas, el siguiente dato tipo word indica la visibilidad de sus lados donde el bit 0 se refiere al lado formado por los vértices CA, el bit 1 por BC y el bit 2 por AB. Los últimos 4 datos se repiten para cada una de las caras.

Chunk ID: 0x4130
Nombre: Lista de materiales de caras
Nivel: 5
Tamaño: 6 + variante
Padre: 0x4120 (Descripción de caras)
Datos: strz nombre_del_material



word número de entradas

(Para cada entrada)
word cara_asignada_al_material (Referencia numérica)
Descripción: Contiene la lista de los materiales que va a utilizar el objeto y en que cara va a estar asignado.

Chunk ID: 0x4140
Nombre: Lista de coordenadas mapeadas para cada vértice
Nivel: 4
Tamaño: 6 + variante
Padre: 0x4100 (Malla triangular)
Datos: word número_vértices

(Para cada vértice)
float coordenada_U
float coordenada_V
Descripción: Contiene la lista de coordenadas U y V que dan el modo en que sé mapea cada vértice.

Chunk ID: 0x4150
Nombre: Lista de grupo de suavizado
Nivel: 5
Tamaño: 6
Padre: 0x4120 (Descripción de caras)
Datos:
Descripción:

Chunk ID: 0x4160
Nombre: Sistema de coordenadas local
Nivel: 4
Tamaño: 54
Padre: 0x4100 (Malla triangular)
Datos: vector x1
vector x2
vector x3
vector 0
Descripción: Define el sistema de coordenadas local donde x1, x2 y x3 representan los ejes y 0 es el origen.

Chunk ID: 0x7001
Nombre: Ambiente de ventana
Nivel: 2
Tamaño: 6 + variante
Padre: 0x3D3D (Editor de 3Dstdio)
Datos: No tiene

TESIS CON
FALLA DE ORIGEN

Chunk ID: 0x3D3E
Nombre: Versión de malla
Nivel: 2
Tamaño: 10
Padre: 0x3D3D (Editor de 3Dstdio)
Datos: dword versión
Descripción: Indica la versión de la malla.

Luces, cámara, color.

Chunk ID: 0x0010
Nombre: Color RGB flotante
Nivel: chunk global
Tamaño: 18
Padre: Variante
Datos: float rojo
float verde
float azul
Descripción: Define un color con el modelo RGB dando las intensidades de cada componente, las intensidades están entre 0.0 y 1.0.

Chunk ID: 0x0011
Nombre: Color RGB byte
Nivel: chunk global
Tamaño: 9
Padre: Variante
Datos: byte rojo
byte verde
byte azul
Descripción: Define un color con el modelo RGB dando las intensidades de cada componente, las intensidades están entre 0 y 255.

Chunk ID: 0x0012
Nombre: Corrector de gama de color RGB byte
Nivel: chunk global
Tamaño: 9
Padre: variante
Datos: byte rojo
byte verde
byte azul
Descripción: Contiene los valores del corrector de gama siguiendo el modelo RGB donde sus valores están dentro del rango de 0 a 255

TESIS CON
FALLA DE ORIGEN

Chunk ID: 0x0013
Nombre: Corrector de gama de color RGB flotante
Nivel: chunk global
Tamaño: 18
Padre: variante
Datos: float rojo
float verde
float azul
Descripción: Contiene los valores del corrector de gama siguiendo el modelo RGB donde sus valores están dentro del rango de 0.0 a 1.0.

Chunk ID: 0x0030
Nombre: Porcentaje entero
Nivel: chunk global
Tamaño: 8
Padre: variante
Datos: word porcentaje
Descripción: Indica un porcentaje entero del cual su rango va de 0 a 100.

Chunk ID: 0x0031
Nombre: Porcentaje flotante
Nivel: chunk global
Tamaño: 10
Padre: variante
Datos: float porcentaje
Descripción: Indica un porcentaje flotante del cual su rango va de 0 a 100

Chunk ID: 0x2100
Nombre: Color ambiental
Nivel: 2
Tamaño: 6 + sub-chunks
Padre: 0x3D3D (Editor de 3Dstdio)
Datos: No tiene
Descripción: Contiene otros chunks de color.

Chunk ID: 0x4165
Nombre: Color del objeto en el editor de 3Dstdio
Nivel: 4
Tamaño: 7
Padre: 0x4100 (Malla triangular)
Datos: byte color
Descripción: Indica el color que tiene el objeto dentro del editor de 3Dstdio.

Chunk ID: 0x4600
Nombre: Luz
Nivel: 3
Tamaño: 18 + sub-chunks
Padre: 0x4000 (Bloque de objeto)

TESIS CON
FALLA DE ORIGEN

Datos: vector posición
Descripción: Indica que hay una fuente de luz y da su posición.

Chunk ID: 0x4610
Nombre: Luz de spot
Nivel: 4
Tamaño: 26 + sub-chunks
Padre: 0x4600 (Luz)
Datos: vector objetivo
float hotspot
float degradado
Descripción: Si esta presente indica que la luz será una luz de spot, su ausencia indica que será un punto de luz que ilumina en todas direcciones.

Chunk ID: 0x4651
Nombre: El spot es rectangular
Nivel: 5
Tamaño: 6
Padre: 0x4610 (Luz de spot)
Datos: BOOLEAN
Descripción: Su presencia indica que la luz de spot proviene de un rectangulo, su ausencia indica que es circular.

Chunk ID: 0x4653
Nombre: Mapa de spot
Nivel: 5
Tamaño: 6 + variante
Padre: 0x4610 (Luz de spot)
Datos: strz archivo
Descripción: Indica el nombre de un archivo con el que se mapeará la luz de spot.

Chunk ID: 0x4656
Nombre: Giro de spot
Nivel: 5
Tamaño: 10
Padre: 0x4610 (Luz de spot)
Datos: float giro
Descripción: Es el giro que tiene la luz de spot, este se indica en grados.

Chunk ID: 0x4700
Nombre: Cámara
Nivel: 3
Tamaño: 38
Padre: 0x4000
Datos: vector posición
vector objetivo
float orilla
float lente

TESIS CON
FALLA DE ORIGEN

Descripción: Define la cámara, el vector de posición indica donde se localiza, el objetivo es hacia donde ve y la orilla se especifica en grados.

Chunk ID: 0xAFFF
Nombre: Editor de material
Nivel: 2
Tamaño: 6 + sub-chunks
Padre: 0x3D3D
Datos: No tiene
Descripción: Indica el inicio de los atributos del material.

Chunk ID: 0xA000
Nombre: Nombre del material
Nivel: 3
Tamaño: 6 + variante
Padre: 0xAFFF (Editor de material)
Datos: strz nombre
Descripción: Indica el nombre del material que se va a utilizar.

Chunk ID: 0xA010
Nombre: Color ambiental del material
Nivel: 3
Tamaño: 6 + sub-chunks
Padre: 0xAFFF (Editor de material)
Datos: no tiene
Descripción: Este contiene chunks de color (generalmente de color rgb y corrector de gama tipo byte).

Chunk ID: 0xA020
Nombre: Color difuso del material
Nivel: 3
Tamaño: 0 + sub-chunks
Padre: 0xAFFF (Editor de material)
Datos: no tiene
Descripción: De igual manera contiene chunks de color (generalmente de color rgb y corrector de gama de tipo byte).

Chunk ID: 0xA030
Nombre: Color especular del material
Nivel: 3
Tamaño: 6 + sub-chunks
Padre: 0xAFFF (Editor de material)
Datos: no tiene
Descripción: También contiene chunks de color (generalmente los mismos chunks de color que los anteriores).

TESIS CON
FALLA DE ORIGEN

Chunk ID: 0xA040
Nombre: Porcentaje de brillo del material
Nivel: 3
Tamaño: 6 + sub-chunk
Padre: 0xAFFF (Editor de material)
Datos: no tiene
Descripción: Contiene un chunk de porcentaje (generalmente entero)

Chunk ID: 0xA041
Nombre: Porcentaje de brillo intenso del material
Nivel: 3
Tamaño: 6 + sub-chunk
Padre: 0xAFFF (Editor de material)
Datos: no tiene
Descripción: Contiene un chunk de porcentaje (generalmente entero)

Chunk ID: 0xA200 – 0xA34C
Nombre: Mapa
Nivel: 3
Tamaño: 6 + sub-chunk
Padre: 0xAFFF (Editor de material)
Datos: no tiene
Descripción: Define los distintos mapas, contiene los chunks para todos estos mapas tal como el nombre de archivo o la escala U/V.

Chunk ID: 0xA300
Nombre: Nombre de archivo del mapeo
Nivel: 4
Tamaño: 6 + variante
Padre: 0xA200 – 0xA34C (Mapa)
Datos: strz nombre
Descripción: Contiene el nombre del archivo a mapear.
Chunk ID: 0xA354
Nombre: Escala V
Nivel: 4
Tamaño: 10
Padre: 0xA200 – 0xA34C (Mapa)
Datos: float V
Descripción: Contiene la escala V del archivo a mapear.

Chunk ID: 0xA356
Nombre: Escala U
Nivel: 4
Tamaño: 10
Padre: 0xA200 – A34C (Mapa)
Datos: float U
Descripción: Contiene la escala U del archivo a mapear.

TESIS CON
FALLA DE ORIGEN

Chunk ID: 0xA358
Nombre: Desplazamiento en U
Nivel: 4
Tamaño: 10
Padre: 0xA200 – A34C (Mapa)
Datos: float U
Descripción: Contiene el desplazamiento(offset) en U.

Chunk ID: 0xA35A
Nombre: Desplazamiento en V
Nivel: 4
Tamaño: 10
Padre: 0xA200 – A34C (Mapa)
Datos: float V
Descripción: Contiene el desplazamiento (offset) en V.

Chunk ID: 0xA35C
Nombre: Angulo de rotación
Nivel: 4
Tamaño: 10
Padre: 0xA200 – A34C (Mapa)
Datos: float angulo
Descripción: Contiene un ángulo de rotación.

Animación.

Chunk ID: 0xB000
Nombre: keyframe
Nivel: 1
Tamaño: 6 + sub-chunks
Padre: 0x4D4D (Chunk principal)
Datos: no tiene
Descripción: Indica el inicio de la descripción del movimiento que tienen los objetos, las luces y la cámara.

Chunk ID: 0xB001 – 0xB007
Nombre: Bloque de información
Nivel: 2
Tamaño: 6 + sub-chunks
Padre: 0xB000 (Keyframe)
Datos: no tiene
Descripción: Estos bloques contienen la información del movimiento de los elementos a animar.

TESIS CON
FALLA DE ORIGEN

0xB001 Bloque de información de la luz ambiental
0xB002 Bloque de información de malla
0xB003 Bloque de información de cámara
0xB004 Bloque de información del objetivo de la cámara
0xB005 Bloque de información del punto de luz hacia todas direcciones
0xB006 Bloque de información del objetivo de la luz de spot
0xB007 Bloque de información de la luz de spot

Chunk ID: 0xB008
Nombre: Cuadros (inicio y final)
Nivel: 2
Tamaño: 14
Padre: 0xB000 (Keyframe)
Datos: dword inicio
Dword final
Descripción: Contiene el cuadro de inicio y de final de la animación.

Chunk ID: 0xB010
Nombre: Nombre del objeto, parámetros y jerarquía del padre
Nivel: 3
Tamaño: 6 + variante
Padre: 0xB001 – 0xB007 (Bloque de información)
Datos: strz objeto
word bandera1
word bandera2
word jerarquía
Descripción: El primer dato contiene el nombre del objeto, en el segundo dato el bit 11 es escondido. En el tercer dato el bit 0 muestra la ruta, el bit 1 contiene la suavidad de la animación, el bit 4 el movimiento borroso y el bit 6 la metamorfosis de los materiales. El último dato contiene la jerarquía del padre, el enlace al objeto con parentesco (-1 para ninguno).

Chunk ID: 0xB013
Nombre: Punto de pivote del objeto
Nivel: 3
Tamaño: 18
Padre: 0xB001 – 0xB007 (Bloque de información)
Datos: vector pivote
Descripción: Contiene el punto que sirve de pivote para el movimiento del objeto.

Chunk ID: 0xB020 – 0xB029
Nombre: Camino
Nivel: 3
Tamaño: 6 + variante
Padre: 0xB001 – 0xB007 (Bloque de información)
Datos: word bandera
8 bytes desconocidos
dword keys

TESIS CON
FALLA DE ORIGEN

(para cada key)
dword posición
word bandera
floats
? dato

Descripción:

Bandera, bits 0-1: 0 simple, 2 repetir, 3 ciclo
bit 3: fijar X
bit 4: fijar Y
bit 5: fijar Z
bit 7: desenlazar X
bit 8: desenlazar Y
bit 9: desenlazar Z

Los 8 bytes siguientes son desconocidos, la siguiente variable es la posición dentro del camino.

bandera (de aceleración del dato presente)
bit 0: Seguir tensión (rango de –1.0 a 1.0)
bit 1: Seguir continuidad (rango de –1.0 a 1.0)
bit 2: Seguir vía (rango de –1.0 a 1.0)
bit 3: Facilidad a seguir (rango 0.0 a 1.0)
bit 4: Facilidad desde donde sigue (rango 0.0 a 1.0)

Siguen n flotantes que indican la aceleración de los datos y la última variable indica un dato que especifica el camino.

El dato especifico del camino es:

0xB020 Posición del camino:
1 vector posición

0xB021 Rotación del camino:
1 flotante ángulo (en radianes)
1 vector eje
0xB022 Escala del camino:
3 flotantes tamaño
0xB023 FOV del camino:
1 flotante ángulo (en grados)
0xB024 giro del camino:
1 flotante ángulo (en grados)
0xB025 Color del camino:
0xB026 Metamorfosis del camino:
1 strz nombre_del _objeto
0xB027 Hotspot del camino
1 flotante ángulo (en grados)
0xB028 Descendencia del camino
1 flotante ángulo (en grados)
0xB029 Esconder camino
BOOLEANO

Chunk ID: 0xB030
Nombre: Posición jerárquica
Nivel: 3
Tamaño: 8
Padre: 0xB001 – 0xB007 (Bloque de información)
Datos: word jerarquía
Descripción: Contiene un valor único para el objeto, el cual es usado para enlazar el árbol de jerarquías.

TESIS CON
FALLA DE ORIGEN

APÉNDICE C

ELEMENTOS DEL FORMATO OBJ

TESIS CON
FALLA DE ORIGEN

Vértice de geometría (v).
Sintaxis:
v x y z w
Estos son los vértices que utilizan los objetos como coordenadas para posicionar sus partes y de esta manera que tomen forma. La "v" indica que se trata de este tipo de vértice donde sus coordenadas x y z son flotantes, la cuarta coordenada **w** es opcional, esta se utiliza solo para las curvas y superficies racionales donde se utiliza para el valor del peso, si no se agrega su valor de default es 1.0.

Vértice de parámetro espacial (vp).
Sintaxis:
vp u v w
Especifica un punto de control en el parámetro espacial de una curva o una superficie. Se puede utilizar solamente el punto u (en un punto especial para una curva), o las 2 coordenadas u v (en un punto especial para una superficie), o las 3 u v w (como puntos de control para una curva racional recortada) donde w es el peso, si este último valor no se especifica su valor de default es 1.0. Los tres valores son flotantes.

Vector normal (vn)
Sintaxis:
vn i j k
Este especifica un vector normal que se utilizará sobre cada uno de los vértices de un polígono, sus coordenadas i j k son flotantes.

Vértices de textura (vt)
Sintaxis:
vt u v w
Este se utiliza para especificar el vértice de una textura que se quiera mapear en el objeto ya sea un polígono o una geometría de forma libre. Dependiendo de la dimensión en que se quiera mapear la textura será el número de variables que se utilicen (por ejemplo para mapear en 2D se necesitan u, v).

Puntos (p)
Sintaxis:
p v1 v2 ...
Especifica un conjunto de vértices que son utilizados como puntos para un polígono, cada punto requiere un solo vértice "v".

Líneas (l)
Sintaxis:
l v1/vt1 v2/vt2 ...
Especifica un conjunto de líneas donde los vértices referenciados por su número son sus extremos. En este elemento mínimo se debe tener 2 vértices geométricos. Los vt son opcionales y le aplican una textura a la línea, estos siempre deben de ir después del primer "/".

Caras (f)
Sintaxis:
f v1/vt1/vn1 v2/vt2/vn2 ...
Especifica una cara utilizando los vértices para un polígono. Se debe tener mínimo 3 puntos (con sus respectivos vértices), los vértices vt y vn son opcionales donde vt son los vértices de la textura que se va a mapear en la cara que sé esta declarando, y los vn son la normal de cada uno de los vértices de la cara. Si una textura ha sido declarada para la cara en cuestión y esta no contiene vértices para mapearla entonces la textura es ignorada.

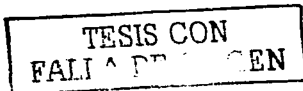
Algunos elementos que definen curvas o superficies de forma libre hacen referencia a vértices utilizando las mismas indicaciones.

Curva (curv)
Sintaxis:
curv u0 u1 v1 v2 ...
Especifica una curva con su rango de parámetros (u0 u1) y sus puntos de control (v). Las curvas no pueden ocultarse o desplegarse al realizar el render pero pueden ser utilizadas de cierta forma por los modeladores. Con u0 u1 se indica el valor de inicio y final de la curva respectivamente mientras que los vértices se utilizan como puntos de control, estos deben ser dos o más.

Curva 2D (curv2)
Sintaxis:
curv2 vp1 vp2 ...
Especifica una curva en 2 dimensiones sobre una superficie, utiliza los vp como sus puntos de control. Este tipo de curvas es utilizado para recortar un objeto por dentro o fuera, como una curva especial o para realizar la conexión de superficies. Como mínimo se debe tener 2 vértices que se utilizan como puntos de control de la curva. Si la curva es racional se puede utilizar 3 coordenadas en cada punto donde la tercera coordenada es el peso (w), si se omite su valor es 1.0.

Superficie (surf)
Sintaxis:
surf s0 s1 t0 t1 v1/vt1/vn1 v2/vt2/vn2 ...
Define una superficie con su conjunto de vértices, s0 s1 son los parámetros de inicio y final respectivamente en la dirección u, t0 t1 son los parámetros de inicio y final respectivamente en la dirección v. Los vértices v son utilizados como puntos de control para la superficie, cada uno de estos pueden tener opcionalmente vértices de textura y normales, estos vértices van en el orden en que se mencionan. Estas también pueden ser superficies racionales donde se podrán utilizar cuatro coordenadas donde la cuarta es el peso, si esta no se indica su valor de default es 1.0.

Estos últimos elementos son utilizados para darle forma a las curvas o superficies de forma libre en conjunto con los siguientes elementos que son su forma general, por lo que las anteriores son parte del bloque definido por los elementos que a continuación se muestran.



Se debe tomar en cuenta que estos elementos deben declararse de la siguiente manera.

cstype rat tipo

Este elemento especifica el tipo de la curva o la superficie donde rat es opcional, su presencia indica que la curva es racional y su ausencia indica que es no racional. En el tipo puede ir cualquiera de los siguientes elementos.

- **bmatrix** (matriz básica).
- **bezier** (Bezier).
- **bspline** (línea especial B).
- **cardinal** (Cardinal).
- **taylor** (Taylor).

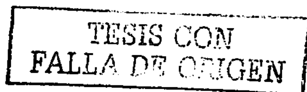
Cada tipo puede representar curvas o superficies sin necesidad de utilizar los elementos que hacen referencia a vértices, pero para hacer uso de esos elementos se debe utilizar también el siguiente.

Grados (deg)
Sintaxis:
deg degu degv
Indica los grados polinomiales para las curvas o superficies. Con degu se indica el grado en la dirección u, este siempre debe estar presente. Con degv se indica el grado en la dirección v, este se utiliza solo en superficies, no tiene valor de default únicamente con cardinal su valor siempre será 3 (aun si se especifica otro). Aunque no son estrictamente necesarios pueden llegar a utilizarse.

Parámetros utilizados para curvas o superficies de forma libre (entre cstype y end).

Valores de los parámetros (param)
Sintaxis:
param u p1 p2 ...
param v p1 p2 ...
Especifica los valores de los parámetros globales, para curvas y superficies bspline estos especifican vectores knot. Cada uno especifica los parámetros en su dirección correspondiente, para definirlos en ambas direcciones para una misma curva o superficie se deben declarar en líneas separadas (como se indica en la sintaxis). Las p son los valores de los parámetros y se debe tener mínimo 2 en la dirección que se esté utilizando.

Recorte (trim)
Sintaxis:
trim u0 u1 curv2d u0 u1 curv2d ...
Especifica una serie de curvas para construir una curva cerrada simple que recortará el exterior del objeto. Con u0 u1 se especifica el valor del parámetro de inicio y final respectivamente, las curvas especificadas con curv2d deben ser definidas antes de



esta línea y este elemento realizará la referencia a estas utilizando el mismo método que con los vértices.

Hoyo (hole)
Sintaxis:
hole u0 u1 curv2d u0 u1 curv2d ...
Especifica una serie de curvas para construir una curva cerrada simple que recortará el interior del objeto, este elemento se maneja igual que el anterior.

Curva especial (scrv)
Sintaxis:
scrv u0 u1 curv2d u0 u1 curv2d ...
Especifica una serie de curvas que yacen en una superficie dada para formar una curva especial. Los parámetros se manejan igual que los elementos anteriores.

Puntos geométricos especiales (sp)
Sintaxis:
sp vp1 vp2 ...
Especifica puntos geométricos especiales para asociarlos a una curva o superficie. Para curvas se utilizan coordenadas en 1D, para superficies son 2D, este utiliza los parámetros de vértice (vp) con su número de referencia.

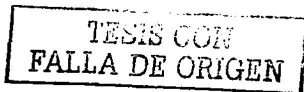
Final (end)
Sintaxis:
end
Especifica el término de la declaración de una curva o superficie.

Conexión (con)
Sintaxis:
con surf_1 q0_1 q1_1 curv2d_1 surf_2 q0_2 q1_2 curv2d_2
Especifica la conexión entre 2 superficies, las curvas y las superficies utilizadas son referenciadas con el mismo método de los vértices, q0 q1 son los parámetros de inicio y final respectivamente de las curvas a las que se hace referencia, hay que notar que los parámetros de este elemento están divididos en 2 partes que son las 2 superficies a conectar por lo que las q0 q1 solo afectan a la curva del lado done están declaradas.

Para agrupar objetos se tienen 4 elementos.

Especificación del nombre del grupo (g)
Sintaxis:
g nombre_grupo1 nombre_grupo2 ...
Especifica el nombre del grupo para los elementos que le siguen, se pueden tener varios nombres para el mismo grupo, los nombre pueden ser alfanuméricos. La información del grupo es opcional.

Grupo de suavizado (s)
Sintaxis:
s número_de_grupo


TESIS CON
FALLA DE ORIGEN

Pone un grupo de suavizado para los elementos que lo siguen, si no se quiere aplicar este grupo de suavizado se pone el valor 0 o también puede ponerse off. El elemento es seguido por el número que identifica al grupo de suavizado.

Grupo de unión (mg)
Sintaxis:
mg número_grupo res
Pone la unión de grupo y la resolución de la unión para superficies de forma libre, de la misma manera que el elemento anterior si no se quiere utilizar se pone off o 0. El elemento es seguido del número de la unión de grupo y res indica la distancia máxima que deben tener los objetos para realizar la unión de grupo.

Nombre de objeto (o)
Sintaxis:
o nombre_objeto
Especifica el nombre de un objeto definido por el usuario.

Todos los grupos contienen todos los elementos que están debajo de ellos incluyendo otros grupos.

El formato también cuenta con comentarios para poder ordenar de mejor forma los archivos, estos se ponen con el símbolo # al inicio de la línea.

TESIS CON
FALLA DE ORIGEN