Diseño e implementación de una herramienta de trabajo para la conversión de formatos y la evaluación de BRDFs

Rosana Montes Soldado

Departamento de Lenguajes y Sistemas Informáticos. Universidad de Granada

Director: *Carlos Ureña Almagro*

Granada, Septiembre de 2001

Introducción

Memoria del trabajo realizado durante el periodo de investigación tutelada correspondiente al Programa de Doctorado: *Métodos y Técnicas avanzadas del desarrollo del software (1999/2001)*. Este trabajo se presenta para la valoración por parte del Tribunal de Doctorado de la capacidad científica y metodológica de la autora para obtener la *suficiencia investigadora*.

En esta memoria se comentan las actividades realizadas durante el segundo año del Programa de Doctorado, así como los resultados obtenidos. Estos se podrían resumir en:

1. Concreción de las líneas de trabajo propuestas por los distintos cursos ofrecidos por el Departamento en el primer año del presente programa.
2. Incardinación dentro de la línea de investigación del "*Grupo de Investigación de Informática Gráfica*"
3. Puesta al día bibliográfica
4. Formación científica general

Para poder describir estos aspectos con mas detalle se ha dividido la memoria en varios capítulos o apartados. En el primero se describen los objetivos principales que se pretendían conseguir, así como los logros alcanzados. En el segundo capitulo se introduce la API gráfica Java3D dando una breve descripción de esta, que nos ayudara a entender los medios de los que se ha dispuesto en cuanto a la parte de implementación. En el siguiente capítulo se detalla el diseño de la aplicación **Wannabe Amazing** que ha sido implementada para conseguir los objetivos principales propuestos. Por último, se comentarán las conclusiones obtenidas y el plan de trabajo previsto, que dejan entrever el camino a recorrer para la realización de la tesis, así como las líneas de trabajo futuras.

Índice General

Capítulo 1

Un proyecto de trabajo

1.1 El periodo de Investigación Tutelada

Con la realización de este Programa de Doctorado se pretenden dos objetivos principales: la realización de la tesis del beneficiario y la formación completa de este como investigador y docente. Para ello integrará su trabajo con miembros del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

En lo referente a la consecución del primer objetivo los puntos a seguir son:

1. Realización de una herramienta para la conversión de distintos formatos comerciales a un formato interno. Esto facilitará el diseño de modelos geométricos y, una vez finalizada, será empleada en la simulación de la iluminación para escenas complejas, con aplicaciones al incremento de realismo en los escenarios de realidad virtual.

2. Realizar un estudio sobre la Reflexión local de la luz y las diferentes funciones empleadas para describir la reflectancia en un punto de la superficie. Se extraerá las características deseables de una **Función Bi-direccional de Distribución de la Reflectancia** (BRDF) para disponer de mejores resultados.

3. Diseño e implementación de una herramienta interactiva para diseño y evaluación de BRDFs. Se ampliará la herramienta anteriormente desarrollada, para poder comprobar la incidencia de la BRDF en la percepción de las superficies.

4. Probar los resultados del punto anterior en la simulación de la iluminación de escenas complejas, con aplicación en la mejora de los tiempos de calculo y calidad de síntesis de la imagen.

Aparte de trabajar en la tesis, la doctorando pretende formarse como investigadora y docente. Para ello son válidos los siguientes puntos:

- Colaboración en los proyectos de investigación asignados a los grupos de investigación del departamento
- Estudio de las diferentes líneas de investigación que se llevan a cabo en el departamento (bases de datos federadas, sistemas concurrentes, ingeniería del software, sistemas interactivos, modelado y visualización...).
- Asistencia a congresos
- Estancias temporales en otros centros de investigación en el extranjero
- En general, colaboración en las diferentes actividades que realizan los integrantes del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

También se han realizado los siguientes cursos de doctorado, enmarcados dentro del programa Métodos y Técnicas avanzadas del desarrollo del software (1999/2001):

Desarrollo Sistemático de Sistemas Concurrentes. Manuel I. Capel Tuñón

Diseño de Sistemas Interactivos aplicados a entornos gráficos y distribuidos. Miguel Gea Megías

Especificación y Garantía de Calidad del Software. Juan Carlos Granja Álvarez

Programación de Aplicaciones para Intranet-Internet. José María Guirao Miras

Especificación y diseño orientados a objetos: aplicación a entornos visuales de desarrollo. José Parets Llorca

Técnicas de modelado, visualización y análisis de datos científico-técnicos. Juan Carlos Torres Cantero y Miguel Vega López

Técnicas Avanzadas para Síntesis de Imágenes. Carlos Ureña Almagro De todos los cursos ofrecidos por el Departamento de Lenguajes y Sistemas Informáticos para el programa: *Métodos y Técnicas avanzadas del desarrollo del software (1999/2001),* la doctorante ha elegido centrar su investigación en Visualización Realista, y nos podemos preguntar el por qué.

1.2 Elección de la línea de investigación

Hoy en día disfrutamos de muchos adelantos técnicos, nos acostumbramos a la comodidad, cubrimos todas nuestras necesidades básicas, y en definitiva hemos olvidado lo que significaría vivir sin todo lo que nos rodea. La sociedad de hoy, da por sentado tantas cosas que estar un día sin luz o sin agua provocaría grandes pérdidas, por no mencionar lo incómodo que nos resultaría.

De similar manera, los detalles del mundo real son ricos y variados, pero perdemos la costumbre de observar lo pequeño. La visualización realista es un campo de la informática gráfica que intenta cubrir la necesidad de realismo en este área. Para ello se plantea el objetivo de generar imágenes realistas, que posean muchos de los efectos que produce la luz cuando interactúa con los objetos físicos reales.

Este objetivo constituye una gran empresa, ya que imitar la naturaleza nunca ha sido una tarea sencilla. Se marcarán objetivos pequeños que ayuden a la obtención de realismo. Varios son los aspectos a tratar:

- perspectivas en modelos de alambre
- eliminación de partes ocultas
- iluminación
- texturas y transparencias
- técnicas de sombreado: variación del color y sombras
- implementación de técnicas por hardware: visualización en tiempo real

Cada una de estas tareas tiene suficiente potencial como para cubrir libros y libros, que traten en profundidad la materia como se merece. No es nuestra intención tratar de explicar aquí en qué consiste cada uno de ellos. Solo queremos que el lector observe que trabajar en visualización realista implica muchos temas, todos ellos relacionados, pero que siempre será mejor concentrarse en un aspecto concreto que intentar abarcarlos todos.

La doctorante por ejemplo, ha elegido profundizar en los aspectos relacionados con la apariencia de los objetos dentro de una escena que es iluminada por una serie de fuentes de luz extendidas. La presentación de la superficie, es decir, la intensidad con la que se observa un objeto dependerá de:

- las condiciones de iluminación, por ejemplo la luz ambiente.
- las características de la superficie: color, grado de transparencia, reflectancia,...
- intensidad y posición de la fuente de luz

Siendo un poco más descriptivos, un buen sistema de renderizado que incluya todos los efectos que produce la luz cuando interactúa con objetos reales considerará:

- reflexión difusa, cuando la luz se refleja igualmente en todas las direcciones, y
- reflexión especular, cuando la luz se refleja en una dirección en particular.

Haciendo uso de una *"Función Bi-direccional de Distribución de la Reflectancia"* **(BRDF)**, se puede describir el comportamiento reflexivo de la superficie del objeto. Dado un punto de la superficie, la BRDF es una función de dos direcciones, una hacia la fuente de luz y otra hacia el observador. Las características de la BRDF determina el *"tipo"* de material que el observador percibe como el material del que el objeto está compuesto. Es por ello, que la elección de un modelo de BRDF así como sus parámetros, constituyen una labor de relevancia dentro del proceso general de síntesis de imágenes realistas.

Más formalmente, la expresión de la BRDF notada por $\rho(\mathbf{k1}, \mathbf{k2})$ nos da el ratio de radiancia observada ($\mathbf{L1}$) por un visor en la dirección $\mathbf{k2}$, proporcional a la irradiancia ($\mathbf{L2}$) procedente de un ángulo sólido infinitesimal ($\delta\omega\mathbf{1}$) alrededor de $\mathbf{k1}$. Es decir:

$$\mathbf{L2} = \rho(\mathbf{k1}, \mathbf{k2})\ \mathbf{L1}\ (\mathbf{k1}\cdot n)\delta\omega\mathbf{1}$$

siendo n la normal a la superficie observada.

Para el modelado de BRDFs podemos optar por dos estrategias de trabajo:

- medición directa. La BRDF puede ser medida directamente usando un gonioreflectómetro, que mecánicamente varía la dirección de una fuente de luz pequeña, y un sensor espectral que recoge un conjunto de medidas para la BRDF.
- métodos empíricos. Existe una gran variedad de modelos de reflexión empíricos, algunos de los cuales nos resultan muy familiares, como son los modelos introducidos por Gouraud en 1971 y por Phong en 1975. Desde entonces, otros muchos modelos han sido desarrollados.

Capítulo 2

Estudio de una API gráfica

2.1 Java3D

El API de Java 3D es un conjunto de clases para crear aplicaciones y applets con elementos 3D. Ofrece a los desarrolladores la posibilidad de manipular geometrías complejas en tres dimensiones. La principal ventaja que presenta este API 3D frente a otros entornos de programación 3D es que permite crear aplicaciones gráficas 3D independientes del tipo de sistema y presentables en la Web.

Las aplicaciones están constituidas por elementos gráficos individuales, como objetos separados que se ven conectados dentro de una estructura en árbol llamada **grafo de escena** que contiene una descripción completa de la escena, o como suele denominarse *universo virtual*.

2.1.1 Introducción

Java 3D es un conjunto de clases, interfaces y librerías de alto nivel que permiten aprovechar la aceleración gráfica por hardware que incorporan muchas tarjetas gráficas, ya que las llamadas a los métodos de JAVA3D son transformadas en llamadas a funciones de OpenGL o Direct3D. Aunque tanto conceptualmente como oficialmente JAVA3D forma parte del API JMF (*Java Media Framework*), se trata de unas librerías que se instalan independientemente del JMF. Al ser síntesis de otras APIs, elimina las *partes oscuras* de estas e introduce un conjunto rico de capacidades descriptivas de escenas 3D, así por ejemplo, es posible hacer uso de la clase *PhysicalBody* para definir el sistema de coordenadas de la cabeza del usuario, situando el origen entre el ojo izquierdo y el derecho, en el plano de la cara. De igual forma podemos encontrarnos objetos con filtros para simular distancia en el sonido, y otros que iremos describiendo en sucesivas secciones.

Es cierto también que JAVA3D puede no soportar directamente todas las posibles necesidades de una persona que trabaje en 3D, aunque sí proporciona la capacidad de implementarlo a través del código java.

Emplea una interfaz de programación de alto nivel basado en el paradigma orientado a objetos, lo que permite obtener todas las ventajas de este: desarrollo simple y rápido de aplicaciones.

La programación de aplicaciones 3D está basada en el *modelo de grafos de escena*. Conecta objetos separados en una estructura arbórea que incluye los datos geométricos, los atributos e información de visualización, que dan una descripción de la escena al completo. Esto nos permite centrarnos en objetos geométricos en lugar de pensar en los triángulos de los que consta la escena.

Ventajas:

- proporciona una visión de alto nivel - orientado a objetos - de los gráficos 3D,
- no es necesario acceder a las operaciones de bajo nivel de renderizado,
- optimizado donde es posible,
- dispone de paquetes para la implementación de cargadores de otros formatos, y otras utilidades que en un futuro se ampliará aún más,
- soporta dispositivos exóticos de realidad virtual.

Inconvenientes:

- es una extensión estándar de la API, tienes la opción de implementarla, pero nadie lo exige, lo cual introduce el riesgo de perder portabilidad,
- no está disponible en todas las plataformas. Sun solo implementa para Solaris y Win32. Sin embargo OpenGL está en cada Unix, Windows y otros sistemas operativos,
- no hay suficiente documentación, comparándolo con OpenGL que cuenta con multitud de webs y libros,
- permanece oculto el proceso de renderizado y, a veces es necesario acceder a esta parte,
- el renderizado es realizado por código nativo no-java, lo que da lugar a componentes heavyweight.

2.1.2 Modelo de Visión Java3D

JAVA3D introduce una nueva visión del paradigma de Java **write once, run anywhere** y lo generaliza para incluir dispositivos de visualización, periféricos de entrada de hasta seis grados de libertad, joysticks, etc., adecuados para la visión de entornos virtuales. El nuevo modelo **write once, view everywhere** significa que un applet escrito usando JAVA3D se puede mostrar en un monitor estándar, dispositivos multi-proyectores, dispositivos montados sobre

una cabeza como un casco virtual, e incluso los de visualización estereoscópica. Todo ello sin necesidad de realizar modificaciones. Tanta versatilidad proviene de la separación de lo que es el mundo virtual del físico. El modelo distingue cómo la aplicación posiciona, orienta, y escala un viewpoint (representado por el objeto ViewPlatform) dentro del mundo virtual, del que finalmente se renderiza y que emplea la posición y orientación dentro del entorno físico.

El modelo de ejecución y renderizado de JAVA3D asume la existencia de un objeto VirtualUniverse que lleva asociado un grafo de escena. Primero explicaremos de que consta un grafo de escena y como se construye, para posteriormente describir los tres modos que hay de renderizar un grafo de escena.: immediate mode, retained mode, y compiled-retained mode.

2.2 Jerarquía de Objetos 3D

La jerarquía de objetos 3D incluye clases básicas para construir y manipular grafos de escenas así como para controlar su visualización y renderizado. Se denominan **core classes** y son unas 100 las incluidas en el paquete *javax.media.j3d* De ellas comentaremos algo en los siguientes apartados.

javax.media.j3d
VirtualUniverse
Locale
View
PhysicalBody
PhysicalEnvironment
Screen3D
Canvas3D (extends awt.Canvas)
SceneGraphObject
    Node
    Group
    Leaf
    NodeComponent
        Various component objects
Transform3D

javax.vecmath
Matrix classes
Tuple classes

Por otro lado, se denominan **utility classes** a las incluidas en *com.sun.j3d.utils*. Son propias de Sun y por tanto no son estándar. Se agrupan en:

1. cargadores de contenidos
2. ayudantes en la construcción de grafos de escenas
3. clases geométricas
4. utilidades convenientes

Posiblemente en un futuro las clases que ahora son de utilidades pasen al núcleo de JAVA3D, además de añadirse nuevas utilidades con lo que todavía no trae implementado. Las clases asociadas con las matemáticas de matrices están en *javax.vecmath*. A su vez podemos seguir haciendo uso de *java.awt* y *javax.swing* para la interfaz de usuario.

2.2.1 Objeto SceneGraph

Se puede decir que una aplicación J3D consiste en indicar el grafo de escena que se renderizará. Un grafo de escena esta formado por una serie de objetos Java 3D, llamados nodos, dispuestos en una estructura de árbol. El usuario decidirá cuantos grafos va a crear y añadir al universo virtual. Las conexiones entre nodos siempre representan la relación "padre-a-hijo".

El grafo a construir es un DAG (grafo dirigido acíclico) y como tal no puede contener ciclos. O sea, tendremos un árbol con una única raíz y cada nodo hoja, solo podrá tener un único padre. Si esto no fuese así existirían más de un camino o *scene graph path* que me llevase del raíz a un hoja en concreto y la información del estado del nodo hoja que me da este camino sería ambiguo. Esto supone un problema, y en este caso (grafos ilegales) se realiza bien la compilación, pero no se renderizarán en tiempo de ejecución.

Además de la relación jerárquica padre-hijo, también se puede conectar los nodos mediante una *relación de referencia*, mediante la cual los nodos pueden compartir información localizada en diferentes ramas. Claramente se ve el alto grado de reusabilidad que esto proporciona.

Se puede decir que todas las *clases core* heredan de una superclase llamada *SceneGraphObject* que a su vez cuenta con dos subclases abstractas:

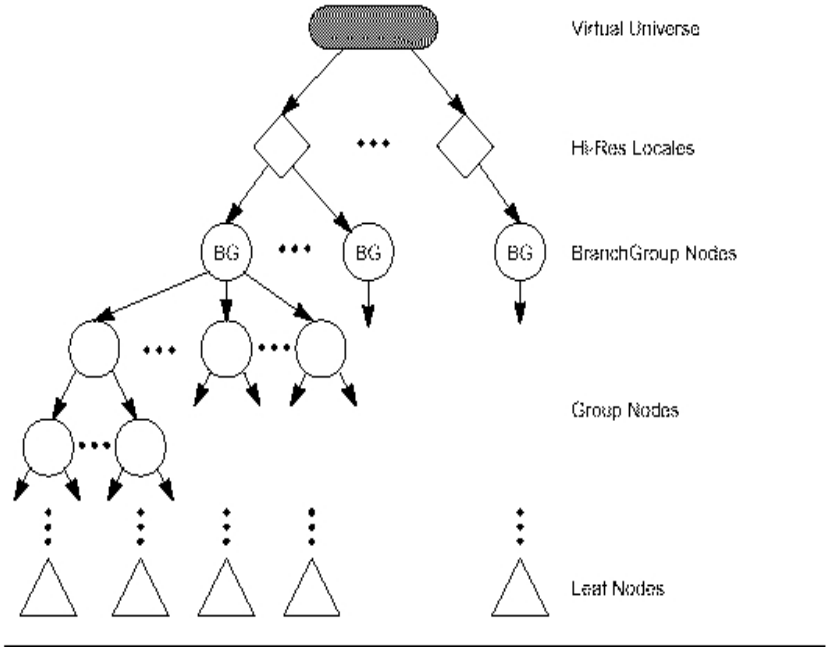
Node que a su vez contiene a *Group* y *Leaf*

NodeComponent Estos no son parte del grafo pero son referenciados por estos, ya que especifican la geometría, apariencia, textura, material, ... de los objetos del grafo.

En cuanto a los subgrafos hay dos categorías:

- **view branch group**: propiedades de la cámara (localización y dirección)
- **content branch group**: focos, sonidos, comportamiento, geometría, ...

Cada subgrafo o branch graph tiene como raíz un *BranchGroup*. Son los únicos que pueden ser insertados en un Locale. Pueden tener múltiples hijos ya sean Group o Leaf.



2.2.2 Objeto Group

Los nodos GROUP son los elementos usados como "pegamento" en la construcción del grafo de escena. Hay siete tipos distintos:

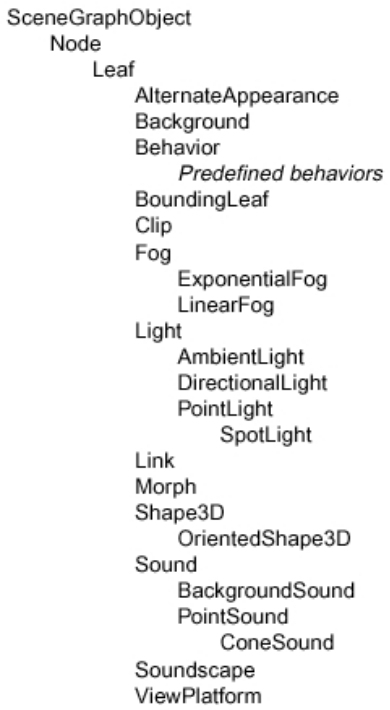
```
SceneGraphObject
    Node
        Group
            BranchGroup
            OrderedGroup
                DecalGroup
            SharedGroup
            Switch
            TransformGroup
            ViewSpecificGroup
```

Todos ellos pueden tener un número variable de objetos nodo hijos, así como otros objetos group. Cada uno lleva asociado un índice que permite realizar operaciones especificas con un objeto hijo en particular, a menos que se utilice un orden especial con el nodo *OrderedGroup*.

Java 3D se reserva el derecho de elegir el orden de renderizado que desee (incluida la posibilidad de renderizar en paralelo).

2.2.3 Objeto Leaf

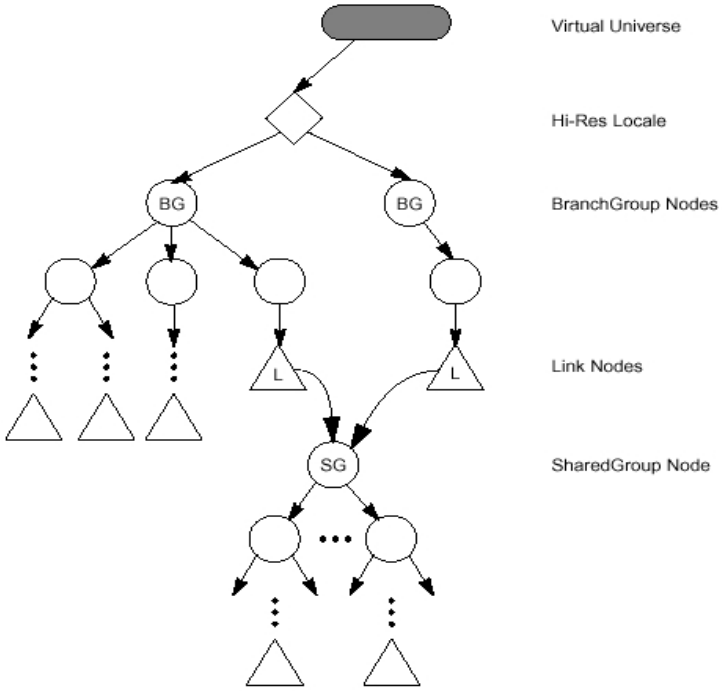
El nodo LEAF define entidades atómicas como son la geometría, las luces y el sonido. En realidad es una clase abstracta para aquellos nodos que no tienen nodos hijos.



Generalmente las entidades que representa tienen determinado por medio de sus propiedades el ser compartidos dentro del grafo de escena de forma que no tenga que ser duplicado su contenido.

JAVA3D le da dos significados distintos a la reutilización de grafos de escena, Para empezar, múltiples grafos de escena pueden compartir un mismo subgrafo. Segundo, la jerarquía de nodos de un subgrafo común puede ser clonada y seguir compartiendo un gran número de componentes de geometría, texturas, ... En el primer caso, los cambios en el subgrafo compartido afectan a todos aquellos grafos de escena que lo referencien. En el segundo caso, cada instancia es única y por tanto los cambios no transcienden a otras instancias.

Una aplicación que desee compartir un subgrafo para que pueda ser utilizado desde varios sitios, debe hacerlo mediante el uso de nodos hoja *Link* asociados a un nodo *SharedGroup* que actúe de raíz del subgrafo, tal y como se muestra en la figura.



2.2.4 Objeto Node

Los objetos NODE son componentes que incluyen la geometría actual y los atributos de apariencia que necesita conocer el renderizador para mostrar los resultados deseados. Toda esta información es especificada en atributos asociados a entidades, que suelen darse en forma de valores en coma flotante o enteros (normales, coordenadas, ...). En muchos casos esa información puede consistir en referencias a entidades mas complejas, denominadas componentes nodo. Su labor va a consistir en encapsular toda la información contenida en una única entidad.

2.2.5 Objeto Behavior

El nodo BEHAVIOR proporciona la forma de animar objetos y procesar los eventos que provienen del teclado y del ratón, reaccionando a movimientos y a eventos de selección. Un nodo Behavior realmente es código Java que puede interaccionar con objetos Java, cambiando los valores de los nodos que se encuentran en el grafo de escena. Generalmente, cambiar los valores significa modificar variables de estado tras realizar el computo deseado. De esta forma se pueden añadir efectos interesantes a la escena. La clase *Behavior* especifica animación con objetos visuales.

La diferencia entre animación e interacción es que el primero responde al paso del tiempo y el segundo lo hace con respecto a actividades del usuario. Hay varios tipos de comportamientos predefinidos en Java3D, que el programador puede personalizar, además de los que puede crear. Entre las posibles acciones podemos cambiar:

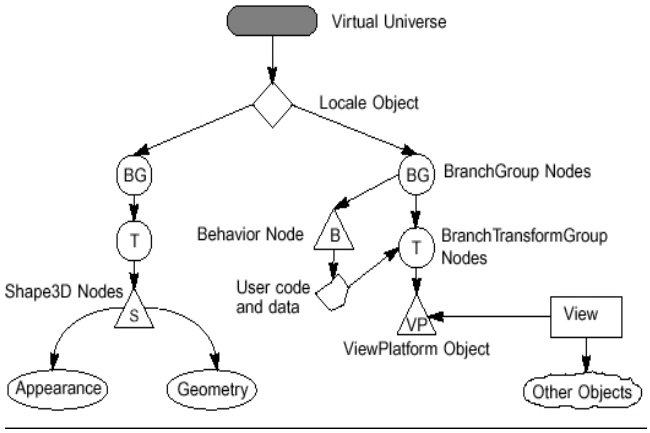
- la localización de un objeto: *PositionInterpolator*
- su orientación: *RotationInterpolator*
- su tamaño: *ScaleInterpolator*
- su color: *ColorInterpolator*
- o su transparencia: *TransparencyInterpolator*

Un objeto visual puede tener varios Behaviors, pero debemos tener en cuenta que disminuirá considerablemente el rendimiento de la aplicación.

Para paliar un poco esto, se definen unos límites espaciales de activación del comportamiento, llamados *scheduling region* dentro de cada *Behavior*, de forma que no se active mientras no se intersecte su volumen. Se puede especificar esta región utilizando el método setSchedulingBounds() o bien con un objeto visual que lo contenga, como un *BoundingSphere*.

La clase *Alpha* proporciona una función que varía en el tiempo generando valores alpha en un rango del 0 al 1 (ambos inclusive). Posee 10 parámetros modificables, lo que da una gran flexibilidad al programador para que lo pueda utilizar para generar movimientos pendulares, rotaciones simples, lanzamientos de cohetes, aperturas de puertas, etc. Se le puede indicar el número de veces que ha de ciclar (-1 infinito) y el tiempo en milisegundos que ha de tardar en ir de 0 a 1.

Con estos nodos, un grafo de escena puede tener el siguiente aspecto:



2.3 Construcción de un Grafo de Escena

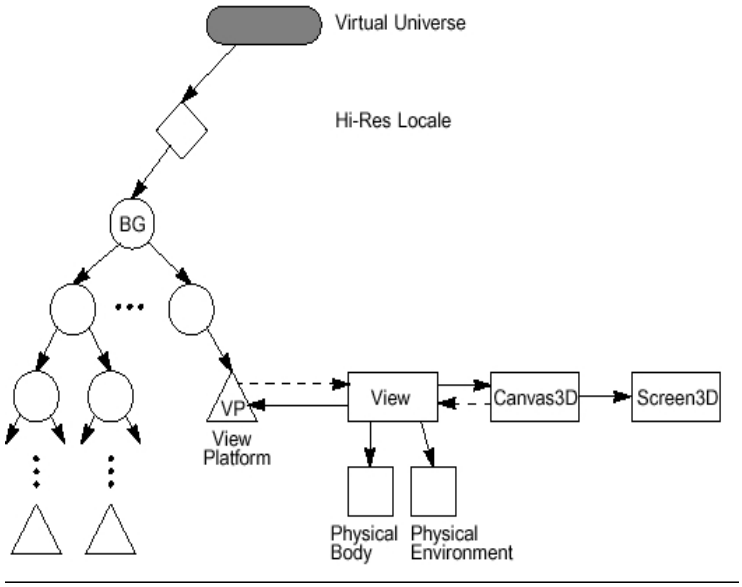
Como se permite la creación de varios universos, los identificamos mediante un objeto *VirtualUniverse*. Este y *Locale*, son las dos superestructuras de objetos de Java3D.

- Un *VirtualUniverse* es un espacio tridimensional destinado a representar un conjunto de objetos asociados.
- Mediante el objeto *Locale* definimos el origen del sistema en coordenadas de alta resolución, también llamadas HiResCoord que sirven como origen para todos los objetos contenidos en el Locale. El origen suele estar situado en el punto (0.0, 0.0, 0.0). *Locale* el contenedor de todos los subgrafos cuya raíz es un nodo *BranchGroup*, por eso sólo estos pueden estar ligados a los *Locale*. Cada *Locale* establece un sistema de coordenadas Cartesiano virtual, y puede haber más de uno.
- El nodo *Behavior* es el que contiene el código para manipular las matrices de transformación asociadas con la geometría del objeto.
- Los *TransformsGroup* son subclase de Group y especifican propiedades como posición (relativa al Locale), orientación y escala del objeto geométrico situado en el universo virtual. Normalmente son creados en un objeto *Transform3D*, los cuales no son parte de los objetos del grafo.
- *Shape3D* es un nodo hoja asociado a dos objetos componentes:
  - *Geometry*: describe la forma geométrica del objeto. Ej. Cubo
  - *Appearance*: describe el color, textura, características de reflectancia del material, etc.
- El nodo hoja *ViewPlatform* transforma las posiciones relativas al Locale especificadas en el TransformGroup a un sistema de coordenadas de usuario dentro del universo virtual. Finalmente *ViewPlatform* es referenciado por un objeto *View* que especifica todos los parámetros necesarios para renderizar la escena utilizando el sistema de coordenadas (el observador) dado por el ViewPlatform.

Hay cinco objetos que no son parte del grafo, pero que sirven para definir los parámetros de vista y proporcionar conexiones (*hooks*) con el mundo físico.

- *Canvas3D*. Encapsula los parámetros de la ventana que es renderizada. Al menos un objeto Canvas3D debe ser referenciado por el viewing graph
- *Screen3D*. Encapsula los parámetros de la ventana física que contiene al canvas, como sus dimensiones en pixels.
- *View*. Especifica la información necesaria para renderizar. Java3d permite múltiples views activos simultáneamente.

- *PhysicalBody*. Encapsula los parámetros del observador: posición del ojo izquierdo y derecho, posición de la cabeza, etc.
- *PhysicalEnvironment*. Encapsula información asociada a dispositivos físicos (*hand tracker*) indicando calibración y otros detalles.



Otras clases empleadas:

- *Transform3D* son usados en la creación de objetos *TransformGroup*. Representa transformaciones como traslación, rotación, escalado o combinaciones de estas. Primero se construye el objeto *Transform3D*, posiblemente de una combinación de otros objetos *Transform3D*, y después se construye el *TransformGroup*.
- *Vector3f* es una clase del paquete javax.vecmath utilizado generalmente para especificar translaciones de geometría, normales a la superficie y demás. No son usados directamente en la construcción del grafo de escena.
- *ColorCube* es una utilidad del paquete *com.sun.j3d.utils.geometry* que define a un cubo con diferentes colores en cada cara, centrado en el origen y con 2 metros por lado.

Como criterio podemos emplear los siguientes pasos para crear una aplicación Java3D

1. Creamos un objeto *Canvas3D* y lo añadimos al panel del applet. Esto es así porque Java3D necesita una ventana en la pantalla donde mostrar el grafo de escena que está renderizando.

2. Crear un *BranchGroup* que actúe como raíz de nuestro grafo.

3. Construir un nodo *Shape3D* con su correspondiente nodo ancestro *TransformGroup*.

4. Adjuntar un comportamiento al *TransformGroup*, digamos por ejemplo un *RotationInterpolator*.

5. Realizar un paso de mensaje a la utilidad constructora del universo para:

   (a) Establecer un universo virtual con un *Locale*

   (b) Crear los objetos: *PhysicalBody*, *PhysicalEnvironment*, *View* y *ViewPlatform*

   (c) Crear un *BranchGroup* que actúe como raíz del subgrafo *ViewPlatform*.

   (d) Insertar la rama *View* al *Locale*.

6. Insertar el grafo dentro del *Locale* del constructor del universo.

La estructura del view-branch-graph es casi siempre igual y por eso nos podemos permitir simplificaciones, como usar el *SimpleUniverse* (utility class). En cambio la parte del content-branch-graph varía de un programa a otro. El constructor de SimpleUniverse se encarga de los objetos VirtualUniverse, Locale y de toda la rama view branch. Las instancias de esta clase realizan los pasos 5a, 5b, 5c y 5d. Reduce por tanto el tiempo y esfuerzo necesario para crear el grafo de escena, y el programador se centra más en la parte de contenido que es la que varía.

Los pasos para construir un programa queda notablemente reducidos a:

1. Crear un objeto *Canvas3D*

2. Crear un objeto *SimpleUniverse* pasándole el *Canvas3D* anterior

   (a) Personalizar el objeto *SimpleUniverse*

3. Construir la rama content graph

4. Compilar el grafo (rama content)

5. Insertar la rama content en el *Locale* del *SimpleUniverse*

Cuando insertamos una rama en un Locale, le damos ***vida*** a todos los objetos de esa rama. Como tales, los objetos vivos están sujetos a ser renderizados. Sus características (*capabilities* en adelante) no pueden ser modificadas a menos que específicamente se habiliten. Los objetos BranchGroup pueden ser compilados, de esta forma se convierten en formas más eficientes para ser renderizadas.

2.3.1 El proceso de renderizado

Es un bucle infinito que comienza cuando hay algún objeto vivo. Permite tres modos de renderizado dejándonos decidir la optimización de la ejecución de nuestra aplicación:

1. Inmediate Mode. No incluye optimización global a nivel del grafo de escena. Resultados pobres pero rápidos.

2. Retained Mode. Requiere no solo construir el grafo, sino que además debemos especificar cómo la aplicación va a animar y modificar los objetos durante el renderizado.

3. Compiled-Retained Mode. Igual que el anterior con la opción de poder compilar algunos de los subgrafos o todos. Este modo es el que proporciona los mejores resultados.

Todos los detalles de renderizado es gestionado por Java3D proporcionando una abstracción de alto nivel de dicho proceso, sin perder por eso prestaciones, poniendo todo el peso en el hardware subyacente. El renderizado con Java3D se ve acelerado por la capa gráfica a bajo nivel sobre la que descansa y está disponible en el sistema, que suele ser OpenGL o Direct3D.

Java3D tiene una representación interna del grafo de escena, que realizará cuando:

- compilamos cada subgrafo (pasando un mensaje *compile* al objeto *BranchGroup*)

- hacemos viva una rama al insertarla al *VirtualUniverse*

Además de la conversión, la representación interna se puede optimizar:

- combinando TransformGroups

- combinando *Shape3D* que estén relacionados de forma estática. Estos objetos influyen mucho; cuanto menor sea su número mejor es el resultado.

Una vez que los elementos del grafo de escena están vivos no es posible modificar sus valores a menos que expresamente sea especificado en las capabilities (capacidades, propiedades, o características) del objeto. Se trata de una lista de parámetros que indica cuales pueden ser accedidos y el modo en el que se accede. Así, por ejemplo, para permitir transformaciones en el tiempo (rotaciones, renderizado sólido o en alambre, etc.) se debe activar en la clase TransformGroup el bit ALLOW_TRANSFORM_WRITE antes de compilar y que se convierta en vivo. Si no se hace así, resultará en una excepción en tiempo de ejecución y el renderizado no se verá bien.

2.4 Uso de la API gráfica Java3D

Observaciones El termino *"J3D"* es comúnmente usado para referirse a JAVA 3D. Dicha convención se empleará también en algunas partes del documento.

2.4.1 Herramientas de Java 2 SDK

Referencia: http://java.sun.com/products/jdk/1.3/docs/tooldocs/tools.html
Dentro del kit de desarrollo podemos encontrar los siguientes ejecutables:

javac Compilador para el lenguaje de programación JavaTM.

java Interprete que lanza las aplicaciones Java technology.

javadoc Generador de documentación, con el mismo estilo que el resto de APIs Java.

appletviewer Ejecuta los applets sin necesidad de un navegador web.

jar Maneja los archivos comprimidos JAR (Java Archive).

jdb Un depurador para Java.

javah Generador de cabeceras para C, usado para escribir código nativo.

javap Desensamblador de clases.

extcheck Utilidad para detectar conflictos en los ficheros Jar.

2.4.2 Versiones actuales

J3D requiere el entorno de ejecución de Java 2, disponible por separado. El lanzamiento más reciente de Java 2 corresponde a la versión 1.3. y esta disponible para Solaris, Linux y Win32.

En cuanto a **JAVA3D** el ultimo lanzamiento de Sun es la versión **1.2.1**. Está disponible para descargar la versión actual y las anteriores para varias plataformas, así como la especificación 1.3 Alpha de la API.

Tanto Java3D como Java2 se pueden encontrar en la web de Sun.

2.4.3 Instalación de JAVA3D

La instalación de J3D requiere realizar previamente los siguientes pasos:

1. Instalar java2 (jdk1.3.0).
   (a) En Linux:
      i. Nos situamos en el directorio que deseemos emplear: /usr/local o bien /usr/lib

ii. Copiamos en dicho directorio el archivo j2sdk-1_3_0_02-linux.bin y lo lanzamos con el shell.

iii. Instalamos el "java runtime environment" llevando el archivo java3d1_2-FCS-linux-i386-re_tar.bz2 para descomprimir en el directorio /usr/lib/jdk1.3.0/jre.

(b) En Windows:

i. por defecto se emplea c:\jdk1.3.0

ii. Solo hay que lanzar el programa Setup.exe para instalar el JDK (kit de desarrollo) y el JRE (entorno de ejecución)

2. Instalar la librería Java3D

(a) En Linux nos situamos en el directorio que nos ha creado la anterior instalación (ej. jdk1.3.0) y descomprimimos el archivo java3d1_2-FCS-linux-i386-sdk_tar.bz2 con la opción -I del programa tar.

(b) En Windows es mucho mas sencillo puesto que el asistente para la instalación se encarga de crear todos los directorios, y depositar de forma adecuada los archivos jars correspondientes a las clases core y utility.

3. Una vez instalado se dispondrán de varios ejemplos de applets (tanto de java2 como de java3d) en el directorio *demos* de la instalación. En algunos casos se requiere incrementar la memoria usada por la aplicación a 64MB. Eso se puede hacer con la siguiente línea de comandos:

- java -Xmx128m
- appletviewer -J-Xmx128m

Actualmente no es necesario incluir ninguna variable CLASSPATH en el entorno. Tan solo se deben localizar los ejecutables dentro de la variable PATH.

Capítulo 3

Wannabe Amazing

3.1 Introducción

Como en cualquier área de investigación, cuando se contempla la posibilidad de desarrollar un software, se hace pensando en los usos que se van a hacer de él, y a quién va a beneficiar. Nuestro caso no es diferente del resto. Si hemos dedicado tiempo y esfuerzo a la implementación de este programa es porque sirve para algo y se va a utilizar. Concretamente dentro de la labor realizada por el Grupo de Investigación de Informática Gráfica, nos encontramos trabajos sobre modelado, métodos formales y especificación de sistemas interactivos, renderizado y visualización realista.

Centrándonos en el trabajo del grupo sobre visualización, el principal objetivo se encuentra en la investigación de métodos y el desarrollo de algoritmos para calcular eficientemente la iluminación global en escenas arbitrarias, así como la visualización de las imágenes resultantes.

Esta línea de trabajo comenzó con la implementación de un sistema clásico de trazado de rayos. Después de esto el grupo rescribió el sistema en C++, usando una análisis orientado a objetos. A continuación, se extendió el sistema con el fin de calcular iluminación global, en lugar del modelo local restringido propio del trazado de rayos clásico.

Este software se conoce como GIRT (*Global Illumination Ray Tracer*). Emplea un método de dos pasadas. La primera es una simulación del transporte de fotones, que proporciona como resultado una aproximación a la emisión difusa de las superficies. La segunda pasada es, en esencia, un proceso de trazado de rayos distribuido, que usa la información obtenida en la primera pasada, y también tiene en cuenta reflexiones especulares perfectas. En los dos pasos, para obtener resultados correctos a un coste razonable se utilizan métodos de indexación espacial, y se optimiza el proceso mediante muestreo aleatorio utilizando métodos de Monte-Carlo.

Compañeros del grupo como Miguel Lastra, continúan con el desarrollo del sistema GIRT, con trabajos basados en el trazado de partículas centrándose en

el cálculo de densidades dado un plano tangente a un punto de una superficie geométrica cualquiera, para la estimación del número de rayos que alcanzan dicho punto.

El sistema de cálculo de iluminación global mediante trazado de rayos GIRT desarrollado por el grupo de Informática Gráfica del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada permite soportar el proceso completo de cálculo de iluminación realista.

Sin embargo, en el contexto de un proyecto más amplio de creación de sistemas usables de síntesis de imágenes realistas se vio la necesidad de facilitar el diseño de modelos geométricos, así como la aceptación de modelos generados por programas comerciales de amplio uso entre los profesionales del sector. Dada la no disponibilidad de aplicaciones de este tipo dentro del software desarrollado por el grupo, por lo específico de sus características, se considero la necesidad de su diseño e implementación.

Es nuestro objetivo el desarrollo de dicha herramienta.

3.2 Estudio de las necesidades del sistema

Básicamente, para la consecución de nuestro objetivo será necesario que nuestro sistema:

1. Facilite el diseño de modelos geométricos y de BRDFs.
2. Realice la conversión de datos de formatos 3D comerciales.
3. Ofrezca la posibilidad de enriquecer modelos geométricos 3D con la información necesaria para el proceso de simulación de la iluminación.
4. Exporte los modelos geométricos 3D extendidos a un formato legible por el sistema GIRT de simulación de la iluminación.

Se requiere una herramienta que nos permita trabajar con escenas complejas basadas en una representación de mallas. Implementar un programa que nos permita crear dichas escenas es algo que se escapa de nuestro objetivo. Es por tanto, mas fácil hacer uso de escenas ya creadas con otras aplicaciones profesionales, que desarrollarlas por nosotros mismos.

Ya que se va a hacer uso de escenas procedentes de otras aplicaciones, se hace necesario hacer una revisión de cuáles son los formatos más empleados, fáciles de obtener o en definitiva, los estándares del modelado en 3D.

La búsqueda nos llevó a estos formatos:

- *Kinetix 3D Studio*
- *Autocad Drawing eXchange Format*
- *Wavefront Technologies*

Además de estos, se vio interesante el añadir a los formatos anteriores:

- *MultiGen OpenFlight*
- *Virtual Reality Modelling Language VRML*

El permitir la carga de un modelo dado en su formato original no es lo único que necesitamos. Es parte de los requerimientos el formato de las salidas, es decir cuál es la sintaxis que describe los ficheros de salida, para que sea legible por el sistema GIRT de simulación de la iluminación. Para ello el formato debe expresar la geometría del modelo empleando un conjunto de mallas del que más adelante daremos detalles.

3.2.1 Características de la aplicación

En cumplimiento con los requisitos mencionados anteriormente se ha desarrollado un programa en Java, que ha venido a llamarse **Wannabe Amazing** sobre el cuál versa el capítulo en el que está el lector, y del que irá obteniendo un mayor conocimiento en los sucesivos apartados.

Las características a destacar de este programa son:

1. Portabilidad: La principal ventaja que presenta el uso de J3D y Java frente a otros entornos de programación 3D es que la aplicación generada es independiente del tipo de sistema, pudiéndose ejecutar en cualquier plataforma. Otro punto fuerte es que puede ser ejecutado en la Web como applet.

2. Trabajo con una librería de alto nivel. El mundo está lleno de objetos, con un estado y un comportamiento. Es más lógico trabajar abstrayendo los detalles propios de cada sistema.

3. Interfaz gráfica amigable: Ha habido grandes cambios en la interfaz gráfica de la aplicación, que ha ido ganando en aspecto desde que se le dio nombre a la aplicación, hace mas o menos un año.

4. Cargador de contenidos extensible: Actualmente está implementado la conversión de cinco formatos comerciales, pero gracias a que Wannabe Amazing es una aplicación diseñada y desarrollada siguiendo la metodología orientada a objetos, está preparada para ampliar la gama de formatos sin necesidad de rediseñar las clases.

5. Específica: No es fácil encontrar en el mercado aplicaciones que se ajuste a la medida de lo que se necesita. Esta aplicación se desarrolla para cubrir unas necesidades concretas que ninguna otra aplicación posee en su totalidad.

3.2.2 Capacidades

Sabemos que con esta aplicación podemos leer y cargar el contenido (geometría, material,...) procedente de un fichero con alguno de los formatos ya comentados,

para visualizarlo y poder exportar dicha información al formato interno GRF comprensible por GIRT.

Nuestro formato expresa cada malla como una lista de vértices y una lista de caras. Adicionalmente se puede incluir normales a los vértices, color de vértice, irradiancia en cada vértice, así como reflectancia y otras propiedades del material. El formato permite a su vez la definición y uso posterior de cada una de estas características (color, reflectancia,...) para que no haya necesidad de repetir datos ya dados. Por último añadir que también es posible realizar "includes" de otros ficheros de escenas con formato GRF.

Sin embargo esto no es todo. Una vez que tenemos el contenido traducido a objetos Java3D interconectados como parte de un grafo de escena y lo visualizamos, el usuario dispone de varias opciones.

El programa nos permitirá visualizar el modelo 3D en modo alambre, sólido con color plano y en modo sólido con sombreado. A su vez permite movernos por el modelo con traslaciones, rotaciones y zooms. Otros aspectos que utilizan los elementos de visualización de Java3D, son la personalización del fondo a color plano, o bien a una imagen (centrada, mosaico, estirado). También permite la inclusión en la escena de luces (ambiental, direccional, focos,...) y su modificación.

Uno de los aspectos más interesantes de la aplicación es la **composición de escenas**. Cuando lanzamos el programa se crea un grafo de escena (*content-branch-graph*) vacío. Cada vez que importamos una escena generamos un subgrafo que añadimos al content-branch-graph, a no ser que expresamente decidamos generar una escena nueva (reinicializando el grafo de escena).

Resta por implementar que la herramienta nos permita editar y aplicar propiedades a los objetos seleccionados (por ejemplo aplicar una BRDF especial a un polígono, o a una malla, y otra al resto). Dichas propiedades se verían reflejadas en el formato de salida tras grabar y exportar la escena.

Para todo esto se requiere una interfaz fácil e intuitiva, accesible a cualquier usuario, así como personalizable y con facilidades para seleccionar y modificar el máximo número de propiedades posibles, para que un usuario especializado pueda sacar el máximo provecho a este "editor de escenas".

3.2.3 Formatos de descripción de escenas comerciales

Gran parte del tiempo de investigación, se ha empleado en el conocimiento de los diferentes formatos comerciales anteriormente mencionados, así como en la implementación de cada uno de los parsers y en la integración de todos estos en una misma aplicación Java. Daremos una breve descripción de las peculiaridades de cada uno de los formatos.

Formato DXF

DXF es posiblemente el formato más extenso de intercambio de datos del software CAD para sistemas informáticos pequeños. El formato DXF (Drawing Interchange Format) posibilita el intercambio de escenas entre AutoCAD® y

otros programas, y es posible encontrarlos tanto en formato ASCII como en binario.

Original de la firma Autodesk, este formato ha sido mantenido en cada nueva versión del programa, desde su lanzamiento hace unos 15 años. Hoy en día encontramos cientos de aplicaciones en el mercado que también apuestan por este formato para la importación y exportación, sobre todo si es un programa CAD.

El formato DXF cambia constantemente con cada nueva versión de AutoCAD (tm), es por ello que nos hemos centrado en la versión más extendida como es AutoCAD R12. De todas formas, las versiones posteriores son compatibles con las más antiguas y dentro del programa siempre tendremos la posibilidad de guardar el modelo con la versión R12.

A grosso modo, en un fichero DXF los datos se distribuyen en capas, y cada dato se indica de forma independiente por líneas. Cada capa consiste en una serie de secciones cabecera, vectores, bloques, y entidades, que describimos a continuación:

- Cabecera (*Header*): Contiene los parámetros de visión y las variables asociadas con el modelo
- Clases (*Classes*): Tiene la información necesaria para definir clases cuyas instancias pueden aparecer en secciones del tipo: bloque, entidades y objetos.
- Tabla de símbolos (*Tables*): Cada tabla puede contener un número variable de entidades. Sin entrar en detalles, comentar que existen varios tipos de tablas.
- Bloques (*Blocks*): Contienen una entrada para todos los bloques referenciados en el modelo. Es decir, todas las definiciones, las entidades que constituyen un bloque, etc. No existe anidamiento en este tipo de sección.
- Objetos (*Objects*): Término empleado para los objetos sin representación gráfica.
- Entidades (*Entities*): Es el término empleado para los objetos gráficos.

La sección cabecera contiene la información de carácter general sobre el gráfico. Se suele incluir aquí el número de versión, especificación de la unidad en la que se encuentra, capa actual, espaciado de la rejilla, tolerancias, límites del gráfico, modo y modelo del terraplén, etc. Puesto que estamos solamente interesados en la transferencia de la geometría no necesitamos una sección de la cabecera, aunque es buena costumbre incluir dicha sección. Una convención estándar es describir al autor del fichero DXF en esta parte, así como fecha y la hora de la creación.

Para que el lector se haga una idea de qué se entiende por objeto gráfico, a continuación se presenta una tabla con las entidades soportadas actualmente por el cargador de contenidos para ficheros DXF.

Entidad	Estado Actual
POINT	soportado
LINE/3DLINE	soportado
CIRCLE	soportado
ARC	soportado
TRACE	soportado
SOLID	soportado
TEXT	casi soportado
SHAPE	no soportado
POLYLINE/VERTEX	soportado
3DFACE	soportado
VIEWPORT/VPORT/VIEW	no soportado
DIMENSION	soportado
BLOCK/INSERT	soportado
ATTRIB/ATTDEF	no soportado

Las diversas entidades del formato DXF pueden contener a su vez diferentes propiedades, alguna de las cuales no están disponibles aun:

Propiedad	Estado Actual
color	soportado
layer	soportado
line types	no soportado
line width	no soportado
extrusion	soportado
font	no soportado

Esencialmente un fichero DXF se compone de pares de códigos con valores asociados. Los códigos, conocidos como *códigos de grupo*, indican el tipo de valor que les sigue. Usando estos códigos y los valores, el fichero se organiza en secciones, que están compuestos por registros, los cuales a su vez podrán contener, códigos de grupo e ítem de datos. El par código - valor irá situado cada uno en una línea. Los códigos de grupo utilizan un número entero de 0 a 999. Por ejemplo el código para los comentarios es 999. Cada código va en una línea, así como cada valor que indique el código de grupo.

Todo esto se presenta de forma no muy legible en un fichero dxf de texto, que no suele ser muy grande (menos de 1MB).

Formato OBJ

Los ficheros objeto definen la geometría así como otras características de un objeto. Puede encontrarse en formato ASCII (obj) o bien en formato binario (mod). Nosotros nos hemos centrado en el primero.

Para describir la geometría podemos encontrarnos tanto objetos poligonales como objetos de libre-forma. El primero de ellos utiliza vértices, líneas y caras para definir los objetos. En segundo emplea curvas y superficies.

Sobre los vértices podrá aparecer en el fichero obj diferentes tipos de listas de datos de información variada. Las listas se indican con una palabra clave (en negrita) que ira seguido del dato en cuestión.

- v vértices que especifican la geometría
- **vt** vértices de textura
- **vn** normales a los vértices
- **vp** (parameter space vertices) atributos que describen las curvas y superficies de libre forma
- **cstype** tipo de superficie o curva: de matrices base, Bezier, B-spline, Cardinal y de Taylor
- **deg** grados
- **bmat** matriz base
- **step** incrementos

Además de dicha información sobre los vértices podemos establecer listas de elementos, como son: puntos (**p**). líneas (**l**), caras (**f**), curvas (**curv**), curvas bidimensionales (**curv2**) y superficies (**surf**).

No es nuestra intención dar una descripción amplia de cuales son las sentencias de libre-forma para la descripción de curvas y superficies, tampoco tratar sobre la posibilidad de conectividad entre superficies de libre-forma. Tan solo destacar que es posible lo anteriormente comentado así como también es posible, especificar atributos de renderizado (interpolación del color, materiales, etc.) y visualización (sombras y trazado de rayos) en la descripción del objeto.

Nótese que este formato se centra en la descripción de los objetos de una escena y no en la escena en sí. No obstante, el concepto escena aparece en este formato gracias al elemento agrupación (**g**) que nos permite agrupar varios objetos por su nombre, definido anteriormente (**o**).

Formato FLT

OpenFlight es un estándar abierto que describe un formato de fichero para 3D en tiempo real, actualmente de dominio público y de gran uso en la industria de la simulación visual.

Una base de datos de descripción de escenas en OpenFlight, soporta varios niveles de detalle, grados de libertad, sonido, instanciaciones (tanto dentro del fichero como en ficheros externos), replicación, animación, secuencias, volúmenes de envoltura, algunas características de iluminación como cadenas de luces puntuales, transparencia, mapeado de texturas, propiedades de materiales y otras características.

Nuestra aplicación implementa un subconjunto de la especificación OpenFlight con los siguientes elementos: paleta de colores, caras, vértices objetos y grupos, ignorando otras características más sofisticadas.

La estructura jerárquica del fichero FLT facilita la agrupación lógica. Se basa principalmente en relaciones padre-hijo, así como en agrupaciones de objetos, polígonos, superficies y vértices. Cada tipo de nodo tiene sus atributos específicos en la base de datos. Una forma de ver la capacidad de este formato, es ver brevemente los principales tipos de nodos que pueden aparecer en un fichero:

- Cabecera: Los nodos son los elementos de la base de datos que aparecen en el gráfico anterior como cajas etiquetadas. La cabecera es el nodo raíz de dicha vista jerárquica. Se crea de forma automática con cada fichero nuevo.

- Grupo: Un nodo grupo distingue un subconjunto lógico de la base de datos. Si aplicamos a un grupo una matriz para trasladar, escalar, rotar o lo que sea, esta matriz será heredada por todos los nodos englobados en el grupo.

- Objeto: Un nodo objeto contiene de forma lógica toda la información de la geometría. Presenta varios atributos, para dar dicha información a varios niveles de detalle.

- Cara: Este nodo representa la geometría. Sus nodos hijos solo podrán ser vértices, que describen un polígono, líneas, puntos o subcaras. Para un polígono, la cara frontal es aquella que se recorre en orden transversal de los vértices. Los atributos usuales de las caras suelen ser: color, textura, material y transparencia.

- Subcara: Este tipo de nodo describe a un nodo cara que es coplanar a otra. En caso de ser dibujada sobre la otra, se refiere a esta como supercara. Las subcaras pueden ser a su vez supercaras.

- Vértice: Un nodo vértice representa una lista de una o mas coordenadas 3D de doble precisión. Para cada coordenada, se almacenan atributos correspondientes a las coordenadas x, y, z y opcionalmente se incluye color, normal, y coordenadas uv para el mapeado de la textura. En la vista jerárquica de la base de datos no se suele mostrar los nodos vértices.

- Nivel de Detalle: LODs son conjuntos de modelos que representan el mismo objeto en la base de datos, empleando distintos números de polígonos, y son fundamentales a la hora de optimizar la base de datos para el dibujado en tiempo real. Un LOD cambia el display basándose en la distancia con el observador.

- Vértice de Morphing: El Morphing es la transición gradual entre un LOD y otro. En el caso de pasar a un LOD superior, el morphing se encarga

de añadir un conjunto extra de vértices, de forma suave. El vértice de morphing define el punto de inicio y el punto final en el camino existente entre estos y el vértice actual, que debe ser interpolado. De los dos puntos extremos, uno representa el mínimo peso (no morphed) y el otro el máximo (fully morphed).

- Switch: El nodo switch contiene una mascara para controlar la visión o no de sus nodos hijos. La mascara es el atributo de todo nodo switch, y permitirá seleccionar cero o mas hijos. No hay límite en cuanto al número de mascaras que se pueden definir.

- Referencia Externa: Un nodo de este tipo, referencia a otro base de datos externa de forma completa. Los contenidos de esta jerarquía hija serán incluidos en la jerarquía parental, sin realizar ningún tipo de comprobación a la geometría. No se podrá editar ningún nodo de la referencia externa.

- Grados de Libertad: Un nodo DOF (degree of freedom) puede ser insertado en cualquier momento para añadir movimiento a la geometría que se encuentre por debajo de su jerarquía. Lo que hace el nodo DOF es establecer un sistema de coordenadas local y controlar que la geometría utiliza estos ejes para moverse. Todos los parámetros de movimiento serán aplicados a todos los descendientes, pudiéndose acumular los efectos de varios DOFs.

- Fuentes de Iluminación: Un nodo de este tipo actúa como un nodo agrupación dentro de la jerarquía. Los usuarios podrán variar tanto la posición como la iluminación de la fuente de luz, y restringir sus efectos a partes especificas de la jerarquía.

- Sonido: De igual forma que los anteriores, estos nodos funcionan como agrupación de otros nodos dentro de la jerarquía de la base de datos. Estos nodos podrán tener asignados nuevos ficheros de sonido en cualquier momento. Dependiendo de la coordenada de localización el sonido tendrá un espacio tridimensional desde el que empezar a emitir.

- Región de recorte: Un nodo de recorte define un plano de recorte. Moviendo dicho plano se puede incrementar la calidad de visualización. Cualquier geometría que caiga fuera de la región de recorte no será mostrada.

- Texto: Un nodo de texto, dibuja una cadena de caracteres, con un tipo de letra determinado con la particularidad de que la geometría de dicho texto no se incluirá en la jerarquía como nodos caras.

- Luces puntuales: Una luz puntual es un nodo que representa una colección de luces representadas por puntos (vértices).

Formato VRML

El *Virtual Reality Modelling Language* (VRML) es un formato estándar de ficheros para gráficos en la web. Puede considerarse como el equivalente tridimensional del HTML. De hecho estos dos lenguajes son similares en cuanto a que ambos:

- están escritos en texto plano y puede utilizarse cualquier editor de textos para su edición.
- son multiplataforma: independientes de cualquier hardware, así como de cualquier sistema operativo.
- usan hiperenlaces para poder hacer referencia a documentos y recursos situados en cualquier lugar de la web, quedando identificados tan solo por su URL (Uniform Resource Locator).
- permiten la inclusión de elementos multimedia. De la misma forma que los documentos HTML presentan embebidos imágenes, sonido y multimedia, los mundos en VRML pueden usar imágenes de textura, vídeo, sonido y ficheros script.
- son un estándar abierto. La especificación de ambos está controlada por el World Wide Web Consortium, siendo VRML un estándar internacional ISO.

Por supuesto que la estructura de cada lenguaje es diferente, así como sus propósitos. VRML describe la geometría y la posición de objetos dentro de un espacio tridimensional, usando coordenadas 3D. De la misma forma, especifica la apariencia de dichos objetos y otras propiedades como iluminación e interacción. VRML es en si orientado a objetos y utiliza un modelo jerárquico de objetos para representar la escena. De hecho los objetos y las primitivas geométricas encuentran sus homónimos en Java3D de forma natural.

La primera versión de VRML fue desarrollada entre 1994 y 1995. Un número de diferentes formatos fue propuesto como base de VRML 1.0, pero tras muchos debates la mayoría de los miembros decidió que el formato ASCII Open Invertor de Silicon Graphics, era el que mejor se ajustaba. El hecho de que tras presentar esta nueva especificación, apareciesen navegadores VRML gratuitos (como el WebSpace de Silicon Graphics), seguido de herramientas software para VRML, potenció el crecimiento de sitios webs conteniendo mundos VRML y tutoriales.

La especcificación 1.0 describe mundos estáticos, permitiendo:

- Crear mundos virtuales 3D basados en formas primitivas como: cubos, conos, esferas y texto, o bien mediante formas definidas.
- Indicar propiedades del material, incluyendo mapas de texturas.
- Indicar el observador inicial, y dejar que este se mueva por la escena de forma más libre.

- Los objetos son "pinchables" y pueden contener enlaces a otros documentos o mundos VRML.
- Objetos inline, aquellos cuya geometría está definida en un fichero VRML a parte.
- Objetos con múltiples niveles de detalle.
- Definiciones de objetos a los que se les asigna nombre permitiendo ser reutilizados.

La demanda de más interactividad y comportamientos creó la necesidad de una nueva especificación VRML 2.0 también conocido como VRML97. En nuestro caso, y puesto que solo necesitamos importar la geometría de un modelo, nos hemos centrado en el formato VRML 1.0

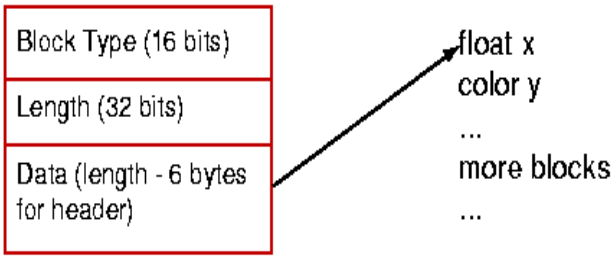
Formato 3DS

El formato de ficheros de 3D Studio de Autodesk Ltd. es uno de los más extendidos, por la potencia del programa y por contar con múltiples adeptos.

Los ficheros 3DS están compuestos básicamente por bloques jerárquicos. Cada bloque comienza con una cabecera, que debe ir seguida por datos. A los datos deben seguirles más bloques.

inicio	fin	tamaño	nombre
0	1	2	ChunkID
2	5	4	Puntero al siguiente chunk

A estos bloques se les conoce como ***chunks***, y describen qué información les debe seguir y de qué se compone esta. Todo queda referenciado mediante el uso de identificadores con la información del siguiente bloque. Realmente este idendificador nos está dando el tamaño del bloque actual.



Finaliza cuando encontramos en el bloque con la posición *start + length*, siendo start la posición del primer bloque y length el tamaño del fichero.

Un fichero 3DS está escrito en binario, pero en un modo muy especial: el byte menos significativo se indica primero en un entero. Por ejemplo, si tenemos un

dato de tipo long como: 4A 5C 3B 8F quedará que la palabra menos significativa y 3B8F es la más significativa.

Son muchos los diferentes tipos de bloques que 3DStudio emplea en su formato (material, vistas, iluminación, texturas, cámaras, etc.), sin embargo todos ellos tienen una jerarquía impuesta por el identificador del chunk. Puesto que al identificador se le asigna un nombre, nos es más fácil poner un ejemplo:

```
EDIT_OBJECT(0x4000)
|
+-- OBJ_TRIMESH (0x4100)
|   |
|   +-- TRI_VERTEXL (0x4110)
|   |
|   +-- TRI_FACEL1(0x4120)
|   |   |
|   |   +-- TRI_SMOOTH (0x4150)
|   |   |
|   |   +-- TRI_MATERIAL (0x4130)
    ...
+-- OBJ_CAMERA (0x4700)
```

Como el lector apreciará, nos vamos a centrar en los aspectos geométricos de este formato. La lista de vértices (0x4110) sigue la siguiente sintaxis:

inicio	fin	tamaño	tipo	nombre
0	2	2	unsigned int	nº total de vértices
2	5	4	float	valor X
6	9	4	float	valor Y
10	13	4	float	valor Z

Los bytes del 2..13 se repiten tantas veces como el número total de vértices que posee el objeto. Por otro lado la lista de caras, va a contener los vértices, expresados de tres en tres, empezando a contar desde el 0. Le sigue información sobre la cara dada mediante un número binario que se expande en tres valores. Por ejemplo 0x0006 se expande a 110 en binario, que en este formato se expresa como AB:1 BC:1 AC:0. Esto indica el orden de los vértices, ya que si AB:1 la línea se dibujará de A a B pero si AB:0 entonces, la línea vendrá de B a A. La sintaxis de la lista de caras es como sigue:

inicio	fin	tamaño	tipo	nombre
0	1	2	unsigned int	nº total de polígonos
2	3	2	unsigned int	nº del vértice A
4	5	2	unsigned int	nº del vértice B
6	7	2	unsigned int	nº del vértice C
8	9	22	unsigned int	información cara

3.3 Diseño e Implementación

Para el desarrollo se ha llevado a cabo un análisis orientado a objetos, y para la implementación se ha elegido el lenguaje de programación Java, junto con las librerías Java3D para almacenamiento y visualización de modelos 3D. Esta elección permite aprovechar las capacidades del lenguaje y la librería citada, reduciendo los tiempos de desarrollo respecto a otros sistemas.

3.3.1 Herramientas de desarrollo

A continuación se pretende identificar cierta información obtenida durante el periodo de investigación tutelada, realizando una revisión de las características de JAVA3D con respecto a otras herramientas basadas en el lenguaje C/C++ y otras librerías gráficas.

Elección del lenguaje de programación: Java

Para empezar intentaremos justificar la decisión de Java como lenguaje de programación, frente a otros como C++. Las características más destacables son:

- Portable: Además de la arquitectura independiente de la máquina, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y con la misma representación interna. Además construye sus interfaces de usuario empleando un sistema abstracto de ventanas, de forma que las ventanas pueden ser implantadas en cualquier entorno.

- Multihilo: Al ser multihilo Java permite varias actividades de forma simultanea. Todo esto aporta un mayor rendimiento interactivo y mejor comportamiento en tiempo real (aunque está limitado a la capacidad del sistema operativo en uso).

- Dinámico: Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el mismo tiempo de ejecución. Las librerías nuevas o actualizadas no paralizarán la ejecución de las aplicaciones actuales, siempre que mantengan el API anterior. Java también simplifica el uso de protocolos nuevos o actualizados. Si el sistema ejecuta una aplicación Java sobre la red y encuentra una pieza de la aplicación que no sabe manejar, Java es capaz de traer automáticamente cualquier pieza que el sistema necesite para funcionar.

- Interpretado: Para conseguir ser un lenguaje independiente del sistema operativo y del procesador, es tanto compilado (generando ByteCode) como interpretado. Aquí entra en juego el *java runtime environtment*, completamente dependiente de la máquina y del sistema operativo que interpreta dinámicamente el ByteCode y añade el 20% de las instrucciones

que restantes para su ejecución. De aquí viene la causa de que Java sea más lento que otros lenguajes como C/C++.

- Completo conjunto de librerías adicionales: la especialización de Java que cuenta con una amplia gama de APIs que entran en contacto con el hardware, facilita el desarrollo rápido de aplicaciones, y amplia el abanico de programadores potenciales de Java. Algunos ejemplos son:
  - Java2D: Gráficos 2D y manipulación de imágenes.
  - Java Animation: animación de objetos 2D.
  - Swing: Mejora de las herramientas para la implementación de interfaces de usuario
  - Java Media Framework: Elementos críticos en el tiempo.
  - Java Telephony: Integración con telefonía
  - Java Share: Interacción entre aplicaciones multiusuario.

Se dice que Java es el lenguaje ideal para el desarrollo de aplicaciones por incluir todas las herramientas que los programadores necesitan, desde las que van a bajo nivel, como las tablas hash y las listas enlazadas, hasta las que van a alto nivel, como CORBA y seguridad en red. Para el caso de aplicaciones científicas, no queda tan claro.

Java ofrece tanto ventajas como desventajas para aquellas áreas más específicas como es el caso del software científico. Mucho se ha hablado del paradigma orientado a objetos. El mundo está formado por objetos y el propósito de los programas es describir estos. Es un punto de vista interesante, pero no se aplica del todo al mundo científico que centra su atención puramente en números y operaciones matemáticas. Sin embargo, Java proporciona muchas herramientas de modo que el programador no necesite ser especialista de la programación orientada a objetos, permitiendo que la fase de aprendizaje se realice de forma fácil y rápida.

Como hemos comentado no todo en Java son ventajas. Un elemento importante que se encuentra en la mente de todo el mundo es el rendimiento. Todas aquellas aplicaciones en la que la velocidad es primordial no son buenas candidatas para Java. Tal vez, para un software de síntesis de imágenes realistas sí entra en juego el uso de supercomputadores, o bien optimizadores de código, que obtengan hasta la más mínima décima de segundo que se desperdiciaba. Sin embargo, para una herramienta como la que se comenta (diseño de modelos geométricos y BRDFs), un PC es suficiente y si ahora se ejecuta con dificultad, en un año un ordenador nuevo lo ejecutará cómodamente.

Distintas alternativas de APIs gráficas

1. PHIGS (*Programmers Hierarchical Interactive Graphics System*) es un estándar internacional para gráfico 2D y 3D, que tubo su influencia a finales de los 80, pero que fue ampliamente suplantado por OpenGL. Estaba orientado al trabajo en modo gráfico retenido.

2. OpenGL es una librería que incorpora un amplio conjunto de funciones de visualización, incluyendo renderizado, mapeado de texturas y efectos especiales. Proporciona aceleración y por ello se utiliza tanto en juegos populares como en simulaciones de alta envergadura. Es soportado por casi todos los sistemas operativos y con casi todo el hardware gráfico 3D.

3. DirectX y Direct3D son interfaces gráficas de Microsoft muy usadas por los juegos para la plataforma Windows. No obstante Windows NT no soporta la ultima versión y es por ello que muchos juegos no pueden ejecutarse en NT (a excepción de Quake que usa OpenGL, por ejemplo).

4. Fahrenheit es una nueva API actualmente en desarrollo, que comenzó siendo un proyecto conjunto entre Microsoft y Silicon Graphics, aunque este último ha preferido abandonarlo y centrarse en código abierto. Microsoft pretende sustituir su DirextX y Direct3D por Fahrenheit.

5. Java3D es un conjunto de clases para crear aplicaciones y applets con elementos 3D. Emplea un modelo jerárquico e incluye el mismo tipo de características que OpenGL.

De todos ellos OpenGL es el que cuenta con más influencia tanto en el mercado, como entre los desarrolladores. Esto se entiende por las ventajas que esta librería aporta:

- proporciona un modelo procedural de los gráficos, próximo a muchos de los algoritmos y métodos que históricamente se han empleado en Informática Gráfica.

- proporciona acceso directo a la secuencia de renderizado, permitiendo a los programadores especificar directamente cómo se va a renderizar los gráficos.

- optimizado en todas las formas imaginables, ya que está optimizado tanto por hardware como por software.

- todos los vendedores de hardware gráfico soportan OpenGL, siendo el estándar sobre el que los vendedores prueban y miden su tecnología.

- OpenGL es un estándar en la industria gráfica y muchos programadores están familiarizados con su API.

Nada es perfecto, y OpenGL también tiene algunos inconvenientes significantes:

- el modelo procedural de los gráficos que puede complacer a muchos programadores, puede no resultar muy adecuado para aquellos acostumbrados a una filosofía orientada a objetos y a las buenas formas de la ingeniería software.

- principalmente se comporta como una máquina de estado.

- optimizaciones en el hardware para OpenGL puede decrementar el abanico de hardware. Cuando un vendedor optimiza mucho un hardware para una plataforma, disminuye la portabilidad y el rendimiento para otras plataformas.

- la exposición de detalles internos en el renderizado por parte de OpenGL puede complicar significativamente un simple programa gráfico 3D. Todo el poder y la flexibilidad de acceder a dichos procesos a bajo nivel tiene un costo, la complejidad.

OpenGL está orientado principalmente a trabajar en modo gráfico inmediato, en el cual cada primitiva mencionada es enviada directamente al servidor gráfico para visualizarla. De esta forma no es necesario ningún alojamiento de memoria para las primitivas. La otra alternativa es el modo gráfico retenido. En este se separan las tareas de definición del modelo y su visualización. En algunos casos el modelo puede ser compilado para generar una visualización más rápida. Este modo no facilita la programación interactiva. OpenGL proporciona soporte al modo retenido mediante el empleo de display lists. Con estas, el programador puede pre-calcular funciones necesarias y almacenar sus resultados.

OpenGL no tiene soporte para modelos de objetos jerárquicos, conocidos como modelos de grafos de escena. Tampoco proporciona soporte directo a efectos como reflexión, refracción, sombras o radiosidad. Es por ello que se hizo un estudio más profundo de Java3D. El lector dispone del Capitulo 2 de la actual memoria como resultado de dicho estudio, para conocer más detalles sobre Java3D.

Elección de la librería gráfica: Java3D

No solo es suficiente con justificar Java, sino que hay que considerar a su vez, las características en las que J3D destaca frente a otras APIs. Los aspectos más interesantes de J3D son:

1. Librería de alto nivel. El desarrollador se centra en el modelo 3D sin necesidad de preocuparse por los detalles de bajo nivel, que se realizarán de forma automática. Los usuario avanzados disponen de medios para poder acceder al proceso de renderizado. Esta librería no solo se encarga del renderizado 3D sino, que incluye otras aportaciones:

   (a) Comportamientos basados en Java.

   (b) Detección de colisiones.

   (c) Sonido envolvente 3D.

   (d) Nuevo modelo de dispositivos virtuales.

   (e) Nuevo paradigma de visión.

2. Multihenebrado y paralelismo, que se traduce en un renderizado eficiente de estructuras de escena.

(a) El renderizado de J3D no esta restringido a ningún orden o recorrido del grafo, y proporciona soporte a la compilación de grafos de escena.

(b) El compilador optimiza el contenido del *branch group* para mejorar el renderizado.

3. Compresión geométrica. Se puede especificar la geometría utilizando un formato binario comprimido al igual que en otras APIs. Las aplicaciones son varias: como formato de tiempo de ejecución, otra forma de describir la geometría, como almacenamiento, e incluso para transmisión eficiente de un modelo por la red. La geometría será representada empleando menos espacio y con poca perdida de calidad.

4. Cargadores de contenidos: dentro de la distribución tenemos la posibilidad de abrir y parsear ficheros de un formato determinado, para convertir su contenido a objetos J3D, y renderizarlos dentro de la vista del modelo. El principal inconveniente de estos cargadores es que devuelven un objeto *scene* y se pierden las características más representativas que dicho formato pudiera tener con respecto a otro. Tan solo se incluye en las utilidades cargadores para:

   (a) Lightwave 3D Scene

   (b) Wavefront .obj

El lector se habrá dado cuenta ya, de las desventajas Java3D:

- no forma parte del paquete estándar de Java. Tampoco Java2D aparecía en sus comienzos como parte de Java y actualmente se incluye en Java2. Solo es cuestión de tiempo.

- necesita el soporte de OpenGL o DirectX, que han de ser instalados previamente.

3.3.2 Descripción del formato de representación

Una vez que tenemos un modelo tridimensional cualquiera en la aplicación, podemos modificarlo, o bien exportar la escena a un formato propio, para trabajo interno. Este formato propio denominado GRF (Granada File) permitirá describir con mayor riqueza sintáctica, los aspectos de iluminación de un modelo de mallas, y será usado por otros compañeros del grupo de investigación, también para simulación de la iluminación. El lector debe entender que posiblemente este formato se verá ampliado en el futuro para enriquecer en la medida de lo necesario el modelo geométrico.

Tanto la descripción del formato GRF como ejemplos de ficheros generados pueden encontrarse en la web: http://lsi.ugr.es/~rosana.

A continuación se muestra el formato presentado, empleando la notación BNF para representarlo. A su vez se irá comentando poco a poco los aspectos más relevantes del formato GRF.

escena ::= *bloques* .

bloques ::= *bloque* | *bloque bloques* | *nada* .

bloque ::= *seccion* | *malla* | *nada* .

seccion ::= *declaracion* | *uso* | *include* | *nada* .

De forma general una escena va a estar dividida en secciones. Cada sección aporta una semántica extra a la dada por el conjunto de mallas que describen la geometría de los objetos de la escena. Las secciones permiten:

- El nombrado de elementos. En lugar de repetir una y otra vez la descripción de una misma propiedad, es posible asignar un identificador a la declaración de un elemento, para posteriormente nombrarlo cuando sea necesario.
- Instanciación. Denominamos instanciación o uso, a la reutilización de una definición anterior. Es una forma de ahorrar espacio en el fichero y de hacerlo más legible.
- Elementos reusables. Se permite la inclusión de toda información sobre mallas, declaraciones y uso que estén descritas en otro fichero GRF, dentro del actual. Se realiza indicando el nombre de dicho fichero, siendo el resultado el mismo que si se hubiese realizado un típico Copiar / Pegar bloque. La diferencia estriba en que aparece más estructurada la información, sobre todo para el caso de escenas complejas.

declaracion ::= *declaracionTextura* | *declaracionLuzExterna* | *declaracionColor* | *declaracionBrdf* .

declaracionTextura ::= *texture textureId nombreFichero* .

declaracionLuzExterna ::= ***light begin*** (*vectoresParalepipedo*) ***end [light]***.

vectoresParalepipedo::= *coords3D arista arista*.

arista ::= *coords3D*.

declaracionColor ::= ***rgb*** *colorValue [colorId]* .

declaracionBrdf ::= ***brdf*** *brfdId nombreFichero* .

uso ::= *usoTextura* | *usoColor* | *usoBrdf* .

usoTextura ::= ***use texture*** *textureId* .

usoColor ::= ***use color*** *colorId* .

useBrdf ::= ***use brdf*** *brfdId* .

include ::= ***include*** *nombreEscena* .

Los elementos susceptibles a asignarles un identificador son: texturas, luces externas, color y BRDFs. No existe una sintaxis común como el caso de los nodos DEF del formato VRML, optándose por una palabra clave que informe del tipo de descripción asociada al identificador. Mas parecidos son los casos de instanciación, que comienzan con la palabra clave *use* y le sigue la palabra empleada en la definición y el identificador. Tanto las declaraciones como los usos de esa, no tienen ningún tipo de restricción en cuanto a orden de aparición u posición dentro del fichero de escena.

---

malla ::= ***begin mesh*** *listaVertices listaCaras* ***end mesh*** .

listaVertices ::= ***begin vertexs*** *vertices* ***end vertexs*** .

vertices ::= *vertice | vertice vertices | nada* .

vertice ::= *coords3D [coordTexture].*

coords3D ::= *(valor, valor, valor)* .

coordTexture ::= *(valor, valor)* .

listaCaras ::= ***begin faces*** *caras* ***end faces***.

caras ::= *[declaracionColor] cara | cara caras | nada* .

cara ::= *nvertices posicion* | (posicion , posicion*)* .

nvertices ::= *literal_entero* .

posicion ::= *literal_entero* .

Nuestro formato expresa cada malla como una lista de vértices y una lista de caras. Adicionalmente se puede incluir normales a los vértices, color de vértice, irradiancia en cada vértice, así como reflectancia y otras propiedades del material (directamente o bien usando el nombrado de elementos y la posibilidad de realizar "includes" de otro fichero). Dentro de cada malla, se describen los vértices que van a aparecer, junto a la información asociada a cada vértice. Posteriormente se describen las caras como una lista de vértices dados mediante un índice o posición ocupada dentro de la lista de vértices de dicha malla. De esta forma si cada vértice aparece en más de una cara, que es lo usual, no es necesario dar de nuevo toda esa información. Se permiten dos sintáxis distintas de expresar las caras, semejantes a la empleada por otros formatos.

Otros elementos que aparecen en la descripción BNF son:

---

nombreEscena := *nombreFichero* .

nombreFichero ::= *cadena* .

colorValue::= *(valor, valor, valor)* .

cadena::= *"conjunto de caracteres Unicode"* .

valor::= *literal_real | literal_entero* .

literal_real = (*decimales* "." [*decimales*] [*exponente*] [*sufijo_tipo*]) | ("." *decimales* [*exponente*] [*sufijo_tipo*]) | (*decimales* [*exponente*] [*sufijo_tipo*]) .

decimales ::= *0..9* .

exponente ::= "e" ["+" / "-"] *decimales* . .

sufijo_tipo ::= "f" | "d" .

literal_entero ::= *0..9* .

textureId ::= *identificador* .

colorId ::= *identificador* .

materialId ::= *identificador* .

brdfId ::= *identificador* .

identificador ::= *a..z, A..Z, ,0..9, $, _* .

comentario ::= // *cadena* .

nada ::= ;

Estamos seguros de que el lector deseará ver el típico ejemplo de un cubo tridimensional expresado en el formato GRF.

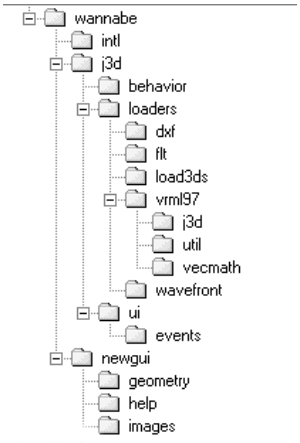
```
// Wannabe Amazing may 16, 2001
// Canvas:  Width 601 Height 512 Scale 0.2
// Background Color : 0
// Image : 0
// Components: 1
// CubeBox : 1  width : 1.0
begin mesh
  begin vertexs
     normal(0.0, 0.0, 1.0)
     (1.0, -1.0, 1.0)
     normal(0.0, 0.0, 1.0)
     (1.0, 1.0, 1.0)
     normal(0.0, 0.0, 1.0)
     (-1.0, 1.0, 1.0)
     normal(0.0, 0.0, 1.0)
     (-1.0, -1.0, 1.0)
     normal(0.0, 0.0, -1.0)
     (-1.0, -1.0, -1.0)
     normal(0.0, 0.0, -1.0)
     (-1.0, 1.0, -1.0)
     normal(0.0, 0.0, -1.0)
     (1.0, 1.0, -1.0)
     normal(0.0, 0.0, -1.0)
     (1.0, -1.0, -1.0)
```

```
  end vertexs
  begin faces
     (0,1,2,3)
     (4,5,6,7)
     (7,6,1,0)
     (3,2,5,4)
     (1,6,5,2)
     (3,4,7,0)
  end faces
end mesh
```

Este ejemplo se almacena en un fichero de unos 831 bytes. En general las escenas grabadas en GRF ocupan poco, del orden de los KBytes, como es el caso de la famosa tetera de OpenGL, con tan solo 69KB. Escenas más complejas, como el galeón de Wavefront, que cuenta con 2372 vértices y 2896 caras, queda descrito usando 265KB.

3.3.3 Diseño de clases. La interfaz gráfica

Siguiendo la filosofía orientada a objetos y el estilo de programación de Java, se ha hecho uso de paquetes para organizar las clases, así como de herencia para aprovechar las clases y utilidades de Java.



Básicamente los paquetes que se ven en la imagen y que ya aparecen organizados pueden clasificarse en:

1. Clases dedicadas a la interfaz gráfica.
2. Clases de utilidades para la interacción con la escena creada en Java3D.
3. Cargadores y exportadores para los diferentes formatos gráficos comentados

Como no tiene sentido hablar de las cerca de trescientas clases de una en una, creo más conveniente tratar la descripción del diseño de las clases más relevantes de los apartados mencionados con anterioridad. Y para estas describir su estructura, variables de instancias, de cada método sus parámetros, prerrequisitos, resultados, etc.

Clases dedicadas a la interfaz gráfica

Dentro de este apartado consideraremos las clases pertenecientes a los paquetes:

- *wannabe.newgui*
- *wannabe.intl*

Este último paquete implementa la parte de los idiomas, pudiéndose elegir entre Ingles y Español. Por defecto el idioma es español, y cuando el usuario selecciona otro, todos los elementos de la interfaz gráfica cambian automáticamente y en tiempo de ejecución el idioma del texto. Esto se puede realizar gracias al empleo de la clase *wbResourceBundle* que hereda de *java.util.ListResourceBundle*. Este contiene una lista de pares clave-valor, donde valor es el texto a mostrar como etiquetas de los objetos de la interfaz para el idioma por defecto y clave es el nombre que se le pasa a los componentes de la interfaz (Menuítems o Buttons) como propiedad texto.

Para cada idioma a implementar, se creará una clase de similares características, en donde se repite la lista de pares-valor y lo único que cambia es el idioma con el que se expresa dicho valor.

Las instancias de la clase *wbResourceBundle* son capaces de responder a los eventos *Locale* de cambio de idioma, y cargar dinámicamente la subclase adecuada.

La aplicación cuenta con una serie de elementos gráficos propios (newgui) que se han elaborado a partir de los dados en el paquete *javax.swing*. Algunos de estos elementos son:

- *AboutBox* - Un "Acerca de" que visualiza el fichero about.html del directorio help, permitiendo el uso de hiperenlaces.
- *ArrowPanel* - Panel de botones norte, sur, este y oeste.
- *Bar* - *Frame* que incluye la barra de opciones y las distintas barras de herramientas entre estas:
  - subclases de *Menu* propias: *BackgroundMenu*, *ActionMenu*, *AligmentMenu*, *HelpMenu*, *LightMenu*, *ViewMenu*, ...
  - subclases de *MenuItem* propias: *AmbientColor*, *ScreenColor*, *ColorAdjust*, *FloatAdjust*, *PercentAdjust*, *MultiItemArray*, ...
  - subclase de *JPopupMenu* propias: *ComponentMenu*, *DeleteMenu,* *MultiMenu*

  - subclase de *MultiMenu*: *CompColor*

- *Dummy* - Componente de la awt al que se puede inicializar con una dimensión especifica. Se utiliza para ajustar los elementos de la interfaz en la posición deseada wannabe.newgui.

- *ESLabel* - Clase JLabel modificada en cuanto al color del texto (negro) y el tipo de letra de la fuente ESUtils.MSG_FONT

- *HelpBox* - Ventana independiente de la aplicación que muestra una ayuda en html. Se implementan los enlaces de forma que es posible "navegar" por la ayuda. Los recursos utilizados se encuentran en el directorio wannabe/newgui/help

- *LibButton* - *JButton* propio que cuenta con la propiedad de cambiar su color cuando el ratón se posiciona sobre él, indicando que es seleccionable. Junto al texto del botón incorpora un pequeño icono verde en redondo.

- *PressPanel*- Panel que utiliza dos botones +/- para permitir que un usuario introduzca un valor numérico. Es utilizado por todas aquellas clases que implementan la interface Adjuster.

- *ToolBar* - Panel que incluye una serie de botones para general una nueva escena, leer una escena existente o exportar el modelo actual al formato GRF.

Recordamos lector que Java no implementa el mecanismo de herencia múltiple, pero que es posible simularla mediante el uso de clases abstractas o interfaces. En este paquete se hace uso de las siguientes interfaces:

- *Adjuster* - Interface que implementan todas aquellas clases que necesiten solicitar un valor numérico al usuario utilizando los botones +/- de la interfaz gráfica cuando estos están habilitados. Entre las clases que implementan esta inferface encontramos:

  - *AngleAdjust* - Emplea los métodos alterValue() y setValue() para modificar el parámetro ángulo de una fuente de luz puntual.
  - Otro ejemplo está en *AttnBox* , para el ajuste de la atenuación de una fuente de luz.
  - *FloatChange*, *IntChange*, etc.

- *CompType* - Interface que implementan todos los componentes wannabe. Son: *BoxBox*, *ConeBox*, *CubeBox*, *CylinderBox*, *SphereBox*, *ModelBox*, *Text2DBox* y *Text3DBox*. En general estas crean los objetos Shape3D necesarios para generar la primitiva en concreto, dentro del grafo de escena. También se encargan de la parte de exportación, es decir, cada componente genera la malla que lo describe en formato GRF, cuando se crea. Esta descripción se añadirá al fichero de escena que puede contener

este y otros elementos, solo cuando el usuario decida grabar el modelo en disco. Los constructores de estas clases tienen algo en común: toman valores por defecto en la creación, y permiten modificar los parámetros de cada primitiva mediante el botón 'Propiedades' de la interfaz de usuario. Estos parámetros suelen ser el alto, ancho, radio, o bien número de divisiones sobre el eje X y el Y.

- *Props* - Interface que define un conjunto de variables de clase, consideradas como globales para todos aquellos que implementen esta inferface.

- *MultiSelector* - Lo implementan aquellas clases que ofrecen más de una opción. Por ejemplo el usuario dispone en los menús de la aplicación de una gama de colores entre los que elegir el color de fondo. Se dice entonces que *BackgroundMenu* implementa esta interface.

Diálogos empleados en la aplicación

- *EndZap* - Clase que se lanza al comienzo de la aplicación para comprobar si la librería Java3D esta instalada. En concreto es una ventana (hereda de la clase Windows) con el mensaje de que se necesita instalar la librería

- *ESDialog* - Es una clase que hereda de JDialog para permitir que este tenga una dimensión proporcional al tamaño actual de la aplicación. Controla también la posición del dialogo. Todos los mensajes que se dirijan al usuario (error, pregunta, etc.) se realizaran con una clase que herede de esta. Subclases conocidas de esta son:

  - *ESMsg* - Diálogo de información en blanco, con el típico botón de continuar tras leer el mensaje. Este texto se le pasa al constructor del diálogo que hereda de ESDialog wannabe.newgui.
  - *ESQuestion* - Diálogo mediante blanco, con los típicos botones de Aceptar/Cancelar. El constructor acepta el texto para la pregunta además de los textos para los dos botones, por si queremos personalizarlos. Extiende *ESDialog*.
  - *ESZap* - Diálogo para indicar mensajes de error. Hereda de *ESDialog* y cuenta con un botón tipo *LibButton*, para continuar.

- *EditPrefs* - Permite al usuario editar las preferencias.

3.3.4 Diseño de clases. Interacción con Java3D

Clases dedicadas al procesamiento

CUtils - Clase que engloba una serie de métodos ampliamente utilizados por los componentes

- métodos de configuración de las propiedades de renderizado:

  - initAppearance(). Inicialización de componentes antes de añadir al grafo de escena.
  - polAttrib.flipFillState(). Permite alternar entre visualización en modo alambre o sólido.

- métodos para la lectura de propiedades de un objeto, procedente de un fichero de escena:

  - readTransform(datainputstream, this). Lee un objeto fuente de luz desde un archivo
  - readPoint(datainputstream). Lee un valor puntual, por ejemplo la atenuación de la luz.
  - readVector(datainputstream). Lee un vector desde fichero, por ejemplo la dirección del foco métodos
  - readColor(datainputstream). Color del Text2DBox

- métodos para la escritura de propiedades de un objeto, procedente de un fichero de escena:

  - writeTransform(dataoutputstream, this). Escribe un objeto fuente de luz.
  - writePoint(dataoutputstream, point3f), writeText(dataoutputstream, ""). Idem para puntos y cadenas de caracteres.
  - writeColor(dataoutputstream, color)j. Utilizado para cada propiedad del color de la fuente: difuso, especular y emisivo.
  - setTexture((TextType)obj, s3d.getAppearance()). Devuelve un objeto textura para *Text3DBox*.
  - getTextAlignment(alignment) ej. para el constructor de un objeto *Text3D*.
  - getTrans(obj, s3d.getAppearance()) ej. devuelve el valor de la propiedad transparencia para un objeto appearance.

DataUtils - Estas es una de las clases de utilidades con métodos y variables estáticas para poder realizar el paso de mensajes sin necesidad de crear una instancia de la clase, de forma que se pueden utilizar sus métodos desde cualquier parte de la aplicación.

- replaceCurrent() - Cuando se modifica las propiedades de un componente, lo que se hace es reemplazar el actual por uno inicializado con las nuevas características.

- bnds - Instancia de Bounds inicializado a un valor muy grande (infinito) para que los siguientes elementos tengan los mismos limites de activación: ambientlight, background, grid, screencolor, light, fog, boundingsphere, etc.

- writeAll(dataoutputstream) - Lo llama la clase *LayoutFile* para grabar todos los elementos usados por la aplicación: propiedades generales, luces y los datos de la escena. Este método llama a writeObject() y writeSubObject() que están implementados en los elementos anteriormente comentados, de forma que cada uno de ellos es el encargado de almacenar sus datos, ya que todos reciben la instancia de dataoutputstream correspondiente al fichero GRF que se generará.

- externFile - Variable de instancia de la clase *Vector* con un registro de todos los ficheros en uso por la aplicación, ej. imagen background, texturas.

- getCanvas(datainputstream, new Dimension(i,j)) - Lee de un fichero de *wbf* las dimensiones del canvas3d para inicializarlo correctamente .

- nComponent - Número de componentes diferentes que se deben exportar en el fichero GRF.

- Este método llama a writeObject() y writeSubObject() que están implementados en los elementos anteriormente comentados, de forma que cada uno de ellos es el encargado de almacenar sus datos, ya que todos reciben la instancia de dataoutputstream correspondiente al fichero GRF que se generará.

ESUtils - clase con utilidades para realizar entradas y salidas a ficheros. Algunos de sus métodos se comentan a continuación:

- getESURL("noway.jpg"), loadImage(image) - Obtiene un recurso dado el nombre de este.

- checkVersion() - Comprueba la versión de java2 (mínimo necesario jdk1.2).

- MSG_FONT - Tipo fuente por defecto para todas las instancias de ESLabel.

- getMI(REMOVE,this) - Devuelve el MenuItem cuyo nombre coincida con el texto del primer argumento.

- getSystemImage(Start.SYS_ICON) - Icono de sistema.

- PRODUCT - String con el nombre de la aplicación: Wannabe Amazing.

Grid - Se trata de la ventana que contiene el Canvas3D necesario para visualizar el universo virtual. Esta ventana es independiente de la que contiene la interfaz de usuario y tiene la particularidad de almacenar como preferencias la última dimensión dada por el usuario.De esta forma la aplicación la abre leyendo desde disco las dimensiones del alto y el ancho.

Clases empleadas en la construcción y uso de los componentes.

La clase *wannabe.newgui.CompType* es la interface que implementan todos los componentes wannabe. Un componente es un objeto gráfico, catalogado como común dentro de una librería gráfica. El programa nos permitirá modificar las propiedades de estos (color, resolución, tamaño, etc.), así como generar un fichero de salida con los datos de su geometría en formato GRF.

- *BoxBox* - Representa al objeto Rectángulo.
- *CubeBox* - Clase especializada de *BoxBox* que representa a un Cubo (Rectángulo con la misma longitud de lados).
- *CylinderBox* - Clase que representa a un objeto Cilindro.
- *ConeBox* - Clase que hereda de *CylinderBox*, para representar un caso especial de Cilindro: un Cono.
- *SphereBox* - Clase que representa una esfera de radio unidad.
- *ModelBox* - Componente no exportable a GRF, empleado para representar la 'caja englobante' de la escena que se esté visualizando en un determinado momento.
- *Text2DBox* - Componente no exportable a GRF, empleado para representar una determinada cadena de caracteres en dos dimensiones.
- *Text3DBox* - Componente no exportable a GRF, empleado para representar una determinada cadena de caracteres tridimensional.
- *LightBox* - Representa tres tipos de luces existentes en Java3D: direccional, puntual y foco. Una vez que la fuente de luz se añade a la escena, el efecto que produce en los objetos queda perceptible al usuario. Esto es así porque al ser un componente wannabe, podemos interaccionar con él y moverlo por la escena para encontrar la posición que de el efecto deseado.

Todas las clases mencionadas son objetos *TransformGroup* para poder ser agregados al grafo de escena del modelo, independientemente que ya exista contenido procedente de la carga de un fichero. A su vez comparten muchas características como los métodos para establecer o consultar las propiedades del componente, la forma en la que se exporta las mallas, el modo de reemplazar un componente, etc. En definitiva, se vio más constructivo la creación de una superclase *ShellBox* que permitiese agrupar toda esta funcionalidad.

En resumidas cuentas, cuando se procede a guardar una escena a formato GRF, se produce una llamada al método writeAll() de la clase *DataUtils* que comienza a generar una serie de llamadas para:

1. Generar la cabecera:
   (a) Nombre de la aplicación, fecha y hora de generación del fichero.

(b) Escala, tamaño y posición de las ventanas, para una próxima ejecución.

(c) Uso de recursos externos: imágenes de fondo, texturas, etc.

(d) Número de componentes exportados (incluyendo fuentes de luz).

2. Comprobar si hay fuentes de luz para exportar. *LightBox* es la clase encargada de esta parte.

3. Exportar las mallas mediante una llamada writeObj() a todos los objeto *CompType* de los que se tenga constancia que se han creado. Esto se traducirá en una llamada a writeObj() de *ShellBox*, seguido writeSubObj() de la clase a la que el objeto pertenezca en tiempo de ejecución. De esta forma en *ShellBox* se exporta la información común a todos los componentes y en writeSubObj() solo lo concerniente a lo particular del componente en cuestión.

Los componentes Wannabe emplean clases para crear la geometría de la primitiva a la que representan. Las primitivas se encuentran en el paquete:

- *wannabe.newgui.geometry*,

Es cierto que tanto *ConeBox*, *CylinderBox*, etc. podrían haber realizado llamadas a primitivas básicas de Java3D que se encuentran dentro de las **utility classes**. Sin embargo y puesto que necesitábamos conocer cómo se implementan dichas primitivas, así como acceder a métodos y variables que quedaban encapsuladas, se ha procedido a la inclusión de dichas clases en un paquete de la aplicación.

- *AnnotationAxes* - Primitiva geométrica que representa los ejes de un sistema de coordenadas. Con su constructor podemos decidir la longitud de los ejes y el grosor de la línea. wannabe.newgui.

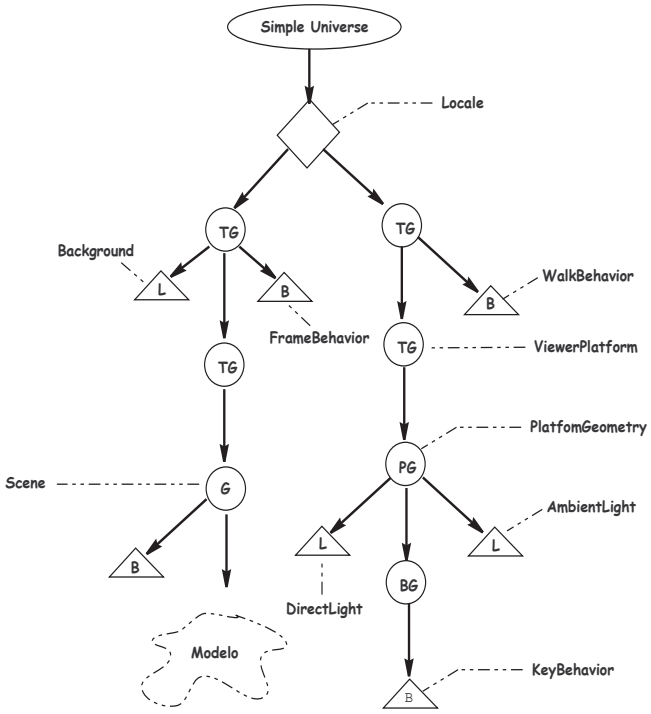
- *Primitive* - Superclase de las siguientes primitivas geométricas: Box, Cone, Cylinder y Sphere.

- *Cuadrics* y *GeomBuffer* - Clases empleadas en la construcción de las primitivas

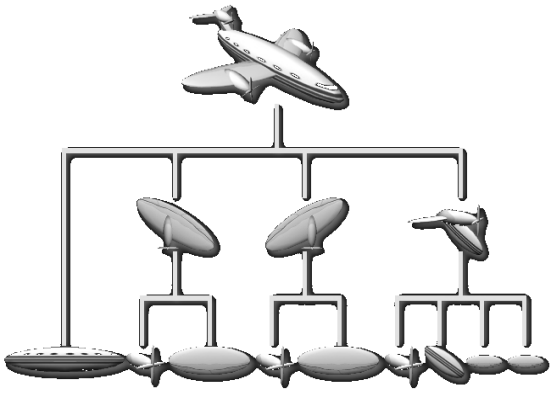
Nótese que no se incluye aquí las primitivas Text2D ni Text3D ya que estas se instancian directamente a las clases de utilidades de Java.

Clases de utilidades para la interacción con la escena Java3D.

La clase más importante es aquella que crea el universo virtual. Se denomina *Amazing,* está dentro del paquete *wannabe.* Su función consiste en crear el siguiente grafo de escena:



La parte del modelo es variable, y en cada caso dependerá del contenido cargado. Dando el ejemplo de un avión, podría venir estructurado de la siguiente forma:



Para la construcción del grafo de escena al completo la clase Amazing cuenta con los siguientes métodos:

- public void init() - Este es el método de iniciación de todo applet Java.

Crea una instancia de Frame que contenga al applet, así como la de la superclase *ModelLoader* de la que heredan todos los cargadores de contenidos implementados.

- public static void load(String filename) - Dado el nombre del fichero de contenidos, se utiliza la instancia de *ModelLoader* para obtiene un nodo *Scene* con todo el modelo cargado, mediante el mensaje al método load(filename) de la clase anteriormente mencionada. La escena leída se añade al universo con el método addGroup().

- private void setTitle(String s) - La ventana de Wannabe Amazing lleva como título no solo el nombre de la aplicación, sino que además:

  - Si no hay ningún modelo cargado pondrá "Sin Titulo" y el nombre de la autora.
  - Si hay algún modelo cargado, llevará el nombre del modelo sin la extensión.

- public static void fileOpenAction(String filename) - Método que responde al evento generado cuando el usuario selecciona el menú "Archivo/Abrir". Realiza una llamada con el nombre del archivo al método load() pasándole el nombre del archivo elegido.

- public static void viewResetAction() - Aplica las transformaciones necesarias al Universo para colocar al modelo en la posición original tras la carga.

- public static String getModelInfo() - Obtiene y presenta al usuario información de la geometría del modelo cargado. Esto es:

  - Número de líneas leídas.
  - Vértice mínimo y vértice máximo leido.
  - Tamaño y centro de la caja englobante del modelo.
  - Número de entidades encontradas, de las cuales se especifican el número de vértices y de caras leídas.
  - Número de errores o warnings ocurridos durante el proceso de carga.

- public static String getVMInfo() - Obtiene la información sobre la Maquina Virtual Java. Los datos recogidos y presentados al usuario son:

  - Vendedor.
  - Versión de la Maquina Virtual - JVM.
  - Estado de la variable classpath.
  - Versión de las clases.
  - Versión del sistema operativo en uso.

- public static void buildUniverse(SimpleUniverse simpleUniverse) - Con este método se crea un universo virtual mediante una utilidad de Java3d (*SimpleUniverse*). La instancia de SimpleUniverse construye e inicializa los siguientes objetos por defecto, a los que posteriormente deberemos acceder (a través de la instancia de SimpleUniverse) a los objetos que queramos modificar:

  - un *Locale* usando el origen en coordenadas "hi-res".
  - un *ViewingPlatform* que usaremos para poder añadir una luz que siempre de hacia el observador (headlight) y cambiar la dirección en la que miramos para simular el estilo de navegación "walk". Hay que decir que esta luz principal que posibilita la visión de los objetos, ya que de otra forma, seria el usuario el encargado de esta tarea, sin la cual no podríamos de dejar de verlo todo en la mas absoluta penumbra. Aquí se añaden los objetos que interactúan con el *ViewPlatform* (empleando el *TransformGroup* más interno), para manejar distintos modos de navegación: *KeyNavigatorBehavior* para el teclado y *WalkViewerBehavior* para realizar la misma tarea a través del ratón. Se construye en esta parte las siguientes instancias:
    * un *MultiTransformGroup* con el cual mover el *ViewPlatform,*
    * un *ViewPlatform* que soporta la vista,
    * un *BranchGroup* para la geometría,
    * un *BranchGroup* para la geometria del view platform,
    * un *Viewer* del que modificaremos las distancias para los planos de corte frontal y trasero. Este objeto a su vez construye:
      · un *PhysicalBody* que caracteriza las preferencias de visualización de los usuarios, así como sus habilidades
      · un *PhysicalEnvironment* que caracteriza el hardware y software de renderizado.
      · un *JavaSoundMixer* que inicializa el sonido dentro del entorno,
      · un View que renderiza la escena en el *Canvas3D*.

El lector notara que cuando utilizamos el vocablo añadir, nos estamos refiriendo a la construcción de un grafo de escena, empleando nodos ancestros y nodos descendientes. Hasta aquí queda construida la rama View-Branch-Graph, que permanecerá invariante frente a contenidos cargados.

La parte variable del grafo de escena, Content-Branch-Graph se inicializa desde el método que pasamos a comentar.

- public static *BranchGroup* createBranchGraph() - Creamos un nodo raíz de la escena, como un objeto *BranchGroup* para el "contenido". A este se añadirán otros, esto es:

  - Un *TransformGroup* que sera modificado por el tipo de navegación "examine" cuando este habilitado. Este comportamiento queda ofrecido por la clase *ExamineViewerBehavior* del paquete *wannabe.j3d.behavior*. La localización de este behavior es distinta a los anteriormente mencionados (*WalkViewerBehavior* y *KeyNavigatorBehavior*) que actuaban en el *TranformGroup* asociados al View-Branch-Graph, frente a este asociado al Content-Branch-Graph.
  - Geometria en 3d que veremos. En principio no debe haber ningún tipo de geometría ya que es el usuario el que debe determinar si se agregar un Componente Wannabe y/o se cargar el contenido geométrico de algún formato conocido. Sin embargo para facilitar la orientación de los objetos en mundo virtual, se crean unos ejes de coordenadas unitarios que se añadirán al grafo de escena, mediante un nodo *Group*.
  - Otros comportamientos. Haciendo uso de las posibilidades de la clase *Behavior*, como código java asociado a un nodo, se ha implementado la clase *FrameBehavior* que atiende a los eventos de repintado expresados en frames por segundo.

- public static Group buildScene() - Este método inicializa el contenido geométrico a unos ejes de coordenadas, creados mediante la clase *AnnotationAxes* que podemos encontrar en el paquete *wannabe.newgui.geometry*.
- public static void addGroup(Scene rscene) - La escena leída se añade al universo con el método addGroup()

Otros métodos de esta clase son:

public void centerView()
public static void reload()
public static void setNavigationType(int nav)
public static int getNavigationType( )
public static void setHeadlightEnable(boolean onOff)
public static boolean getHeadlightEnable( )
public static void setLightEnable(boolean onOff)
public static boolean getLightEnable( )

Es importante establecer las propiedades (Capability) de los objetos *BranchGroup*, *TransformGroup*, etc. que nos permite leer, o modificar las características de la grafo de escena construido y evitarnos así errores en tiempo de ejecución.

Otro punto de gran repercusión es la compilación de la escena (nodo *BranchGroup* raíz creado por este método) para que Java3D pueda efectuar las optimizaciones oportunas y su inclusión en el grafo. Es aquí cuando se añade al universo (creado en el método anterior) cuando se dice que está vivo (lived) y Java3D comienza a ejecutarlo en un bucle infinito.

Amazing se apoya en otras clases como son:

Grid - Con anterioridad comentábamos que la clase *Grid* es la ventana que soporta la visualización de la escena que el usuario percibe y maneja, pero Grid no es solo un contenedor para el Canvas3D. A su vez alberga los datos y los métodos relacionados con el modelo incluido en el Universo Virtual.

- widgets: que es el vector de componentes creados y añadidos a la escena (los que no proceden de ningún fichero en particular, sino que son creados por Wannabe Amazing)
- current: es un valor entero que informa de cual es el componente activo en el momento actual.
- bounds: valor que identifica el espacio ocupado por todo el modelo, o sea, los limites que abarca.
- canvas: instancia de la clase *MainPanel* consistente en un *Canvas3D* capaz de reaccionar a eventos de ratón (picking) y redimensión (resize)
- simpleuniverse: instancia de la utilidad *SimpleUniverse* de Java3D, que inicializa a valores por defecto, la parte no concerniente con la geometría de un Universo Virtual.
- scene: objeto *Scene* correspondiente a una escena en blanco (sin geometria), inicializada para contener unos ejes de coordenadas unidad por defecto. La instancia de este objeto se obtiene tras llamar al método buildUniverse() de la clase *Amazing*
- sceneTransform: procedente de la misma llamada se obtiene de la clase *Amazing* un *TransformGroup* aplicable a toda la escena, inicializado a la matriz al factor de escala de la escena.
- po: instancia de la clase *PickObject* de la utilidad com.sun.j3d.utils.behaviors.picking.* que posibilita la selección de un objeto.

Métodos de la clase *Grid*:

- static void addObject(Object obj, boolean flag): Añade un componente al grafo de escena.
- public static void checkSize(): Comprueba el tamaño y la posición que tenía el Grid en la ejecución anterior para restaurar dichas propiedades en la ejecución actual.
- static void deleteAll(): Elimina del modelo los componentes que hayan sido agregados. No modifica la geometría que proceda de formatos importados.
- static void deleteCurrent(): Elimina el componente que este seleccionado. Solo podrá eliminarse un componente cada vez y si no hay ningún componente seleccionado, se procede a avisar al usuario de ello.

- static void deselect(): lleva al sistema a un estado en el cual ningúncomponente se haya seleccionado.
- static CompType getCurrent(): Obtiene la instancia de *CompType* (tipo estático) correspondiente al componente seleccionado. En tiempo de ejecución esa variable es del tipo de la subclase (tipo dinámico) en cuestión y de esa forma se actúa sobre el objeto adecuado.
- static void pickWidget(int i, int j): Obtiene el objeto más cercano a la posición (i,j). Si no se encuentra ningún objeto a esa altura se llama al método setOrig() de esta misma clase.
- static void replaceComp(int i, CompType comptype) : permite la modificación de las propiedades de un componente, cambiando el actual por el modificado. Ambos son de la misma subclase de *CompType*.
- static void setCurrent(): Método que activa el botón de Propiedades de un componente, ya que este ha sido seleccionado.
- static void setDimension(Dimension dimension): Establece las dimensiones de la ventana *Grid*.
- public static void setOrig(): Muestra las dimensiones actuales del *Grid* en la barra de la ventana que lo contiene.
- static void setPos(int i, int j): Muestra la posición (X,Y) actual del *Grid* en la barra de la ventana que lo contiene.
- static void setPos(int i, int j, int k, int l): Muestra la posición (X,Y), así como el ancho y el alto.

Clases dedicadas al comportamiento de la escena

Dentro de este apartado consideraremos a su vez las clases pertenecientes a los paquetes:

- *wannabe.j3d.behavior*
- *wannabe.j3d.ui*
- *wannabe.j3d.ui.events*
- *wannabe.j3d.ui.record*
- *wannabe.j3d.ui.tools*

Una clase *Behavior* proporciona la forma de animar objetos y procesar los eventos que provienen del teclado y del ratón, reaccionando a movimientos y a eventos de selección. Actúa generando las transformaciones que son necesarias aplicar a un objeto *TransformGroup* dado en el constructor, modificandolo automáticamente para obtener los resultados deseados.

Objetos *Behavior* típicos (Rotación, Traslación, Zoom) son proporcionados por la librería Java3D. Las clases que heredan de esta, imponen una semántica específica. Este es el caso de "Examine" o "Walk" que ofrecen un modo de navegación similar a los visores de VRML.

Para empezar se ha implementado una clase abstracta, denominada *Viewer-Behavior*, inspirada en las utilidades *MouseBehavior*, *MouseRotate*, *Mouse-Translate*, y *MouseZoom*. Consigue los siguientes objetivos:

1. Encapsular los tres comportamientos en uno sólo, de forma que se refuerza la semántica de visualización de la escena.

2. Soportar métodos set/get para controlar los parámetros de velocidad de movimiento para operaciones de rotación y traslación.

3. Soporta el modificador "Control" como alternativa para aquellos ratones que tienen menos de tres botones (uno para cada operación).

Los comportamientos implementados son:

- *WalkViewerBehavior* y *KeyNavigatorBehavior*. Ya sea a través de ratón o de las teclas, el usuario puede utilizar el estilo 'walk' para moverse sobre el centro del observador como si este se pudiera girar sobre si mismo viendo la escena en la que esta inmerso. Ambos actuan sobre el *TranformGroup* asociado al View-Branch-Graph.

- *ExamineViewerBehavior*. La localización de este behavior es distinta a los anteriormente mencionados, ya que modifica la parte del Content-Branch-Graph. Lo que el usuario apreciará es que "examine" es un estilo en el que la rotación será sobre el origen de la escena, como si estuviera bajo observación.

- *FrameBehavior*. Atiende a los eventos de repintado del contenedor de orden superior de la escena. Si la escena se ve en un objeto *Canvas3D*, el contenedor al que nos referimos es la ventana (clase *Frame*). De esta obtendremos un valor numérico expresado en frames por segundo (fps), que el usuario podrá ver en la parte correspondiente a la barra de estado.

3.3.5 Diseño de clases. Cargadores de contenidos

Con anterioridad se ha hablado de los formatos que es capaz de leer la aplicación. Cada formato tiene sus propias particularidades, sin embargo el diseño de un *Loader* o cargador para un fichero es muy similar. Este es el caso de las clases: Loader_DXF, Loader_FLT, Loader_OBJ, Loader_3DS, Loader_WRL y los respectivos paquetes:

- *wannabe.j3d.loaders*

- *wannabe.j3d.loaders.dxf*

- *wannabe.j3d.loaders.flt*
- *wannabe.j3d.loaders.load3ds*
- *wannabe.j3d.loaders.vrml97,*
  - *wannabe.j3d.loaders.vrml97.j3d,*
  - *wannabe.j3d.loaders.vrml97.util*
  - *wannabe.j3d.loaders.vrml97.vecmath*
- *wannabe.j3d.loaders.wavefront*

Con esta premisa, los comentarios que a continuación voy a realizar no están sujetos a ningún cargador en concreto. Dependiendo del caso se podrán ejemplos de uno u otro formato.

Los pasos a seguir son:

1. Comenzar con un formato, encontrando su especificación.
   (a) Consideraciones del formato: qué ficheros puedo obtener, disponibilidad de herramientas, etc. Se resuelve gracias a los múltiples sitios webs sobre gráficos 3D (ver apartado 3.6.1).
   (b) Dónde puedo obtener información sobre la especificación: sitio oficial para la casa de cada formato, listas de distribución, webs para programadores.
2. Conseguir algunos modelos de prueba
   (a) Encontrar un repositorio de modelos gratuito. La mayoría de webs de recursos, cuentan con descargas gratuitas de modelos, además de otros de pago.
   (b) Intentar encontrar un rango de modelos simples, de forma que no superen 1 MB de espacio. Deben incluir geometría pero no texturas u otros aspectos más avanzados. Probar con diferentes versiones y diferentes características.
3. Entender la especificación del formato
   (a) Cada formato tiene sus propias peculiaridades: cabeceras, agrupaciones, normales a los vértices, texturas, etc.
   (b) Identificar la forma de expresar los vértices y las caras
4. Resolver todos los asuntos relacionados con los tipos de datos
   (a) Ver si emplea big endian o little endian
   (b) Conocer cuantos bits emplea el tipo base

(c) De qué forma se especifica el color u otra propiedad: entero, real, porcentaje, ...

(d) Establecer rangos arbitrarios. Así en el caso de 3DS:

Java3D	3DS
Color3b	COLOR_F
Color3f	COLOR_24
Color4b	LIN_COLOR_F
Color4f	LIN_COLOR_24

5. Implementar la *interfaz Loader*

   (a) Crear clases que hereden de *com.sun.j3d.loaders.BasicLoader* . Cuando usamos el diálogo abrir fichero, vemos que este contiene un filtro para los formatos dxf, obj, 3ds, wrl y flt, correspondientes a los cargadores implementados. La clase *wannabe.j3d.loaders.ModelLoader* que hereda de com.sun.j3d.loaders.BasicLoader permite, dado un cargador especifico para cada formato (subclase de ModelLoader), decidir en tiempo de ejecución y con el archivo abierto, cual sera la subclase a instanciar. Se hace patente el empleo clases abstractas (*interfaces* en Java) que nos permite una especialización del comportamiento de un objeto. Queda pues preparado para permitir la carga de más de un formato gráfico.

   (b) Crear clases que hereden de *com.sun.j3d.loaders.SceneBase* ajustándose a lo específico de cada formato.

6. Primero identificar la geometría

   (a) Encontrar los vértices y las caras. Usar como apoyo las clases de *com.sun.j3d.utils.geometry.*

   (b) Usar un material por defecto y emplear un background discreto.

   (c) Indicar los parámetros de visualización de la clase Appearance: modo alambre, sólido o sombreado.

   (d) Usar una escala fija (1m por ejemplo) y posicionar los objetos a una misma distancia respecto del origen

7. Ejemplo de código: Geometría

```
Shape3D s3d = new Shape3D();
int faceCount = readShort();
s3d.setAppereance(defaultAppearance);
GeometryInfo gi = -
new GeometryInfo(GeometryInfo.TRIANGLE_ARRAY);
Point3f points[] = new Point3f[count*3];
```

```
for (int i=0; i<faceCount; i++) {
    int index = readShort();
    points[i*3] = faces[index][0];
    points[i*3+1] = faces[index][1];
    points[i*3+2][2] = faces[index][2];
}
gi.setCoordinates(points);
s3d.setGeometry(gi.getIndexedGeometryArray(true));
```

8. Añadir materiales y appearance
   (a) Identificar los materiales en el formato
   (b) Establecer propiedades de color (emisivo, ambiente, especular y difuso), así como de brillo y transparencia.
   (c) Asociar los materiales con la geometría
   (d) Probar los modelos con y sin texturas.
   (e) Aquellos modelos que usen texturas, deben ignorar las propiedades de color del material
   (f) Dejar el modelo con Appereance en modo alambre.
9. Ejemplo de código: Materiales

```
switch (block_type) {
    case TYPE_MATERIAL_NAME:
        materials.put(readString(), appearance);
        break;
    case TYPE_MAT_AMBIENT:
        material.setAmbientColor(readColor());
        break;
    case TYPE_MAT_SPECULAR:
        material.setSpecularColor(readColor());
        break;
    case TYPE_MAT_DIFFUSE:
        material.setDiffuseColor(readColor());
        break;
    case TYPE_MAT_SHININESS:
        float shn = 1.0f + 127.0f * readPercentage();
        material.setShininess(shn);
        break;
}
```

10. Añadir superficies y normales
    (a) Comprobar la información de las normales

(b) Si el fichero incluye normales para cada vértice, entonces se usará dicha información. En caso contrario calculamos las normales usando la clase de Sun *NormalGenerator* que además normaliza cada vector normal.

(c) Comprobar casos en los que las normales estén mal. Por ejemplo si se puede ver a través de las superficies

11. Ejemplo de código: Superficies

```
case TYPE_MAT_2_SIDED:
    appearance.setPolygonAttributes(
        new PolygonAttributes(
            PolygonAttributes.POLYGON_FILL,
            PolygonAttributes.CULL_NONE,
            0.01,
            true
        )
    );
```

12. Ejemplo de código: Normales

```
GeometryInfo gi = ...
if (needToReverseNormals) {
    gi.reverse();
}
newÑormalGenerator().generateNormals(gi);
s3d.setGeometry(gi.getIndexedGeometryArray(true));
```

13. Añadir el resto de características

(a) Llegados a este punto, se puede ver qué más incluye el fichero:

i. color o imágenes de fondo
ii. texturas
iii. efecto de niebla
iv. iluminación: ambiental, fuentes de luz extendidas, ...
v. diferentes cámaras o vistas.

(b) Otros efectos: sombras, sonido, comportamientos (behaviors) como en el caso de VRML, etc. Aunque estos elementos no son relevantes para la consecución de los objetivos planteados, quizá en un futuro se utilice esta información para ampliar las funcionalidades de la aplicación si se considerase oportuno.

Cargadores y exportadores de formatos comerciales

Después de esta revisión sobre cómo se ha implementado cada uno de los cargadores quedaría por comentar brevemente algunas notas referentes a cada uno en particular.

- wannabe.j3d.loaders.dxf - En este caso el peso del parseado del formato dxf recae sobre la clase DXFLoader, que se apoya de vectores para el almacenamiento de los bloques leidos: *PointList*, *NodeList*, *LayerList*.

- wannabe.j3d.loaders.flt - Implementa un subconjunto de la especificación OpenFlight con las siguientes clases como representación de cada elemento: *FLTVertex*, *FLTtextureMapping*, *FLTtexture*, *FLTobject*, *FLTnode*, *FLTmaterial*, *FLTmaterialPalette*, *FLTlightPoints*, *FLTinstanceDef*, *FLTinstanceRef*, *FLTheader*, *FLTgroup*, *FLTface*, *FLTcolorPalette* y *FLTDOF* entre otros.

- wannabe.j3d.loaders.load3ds - Este cargador mezcla la filosofía empleada para los dos formatos anteriores. Por un lado existen clases que reflejan la forma en la que se especifica una cara (clase *Face*) o el hecho de que el fichero vienen por bloques (clase *Chunk*). Y por otro lado empleamos vectores para almacenar todas las características leídas: *AppearanceList*, *ObjectList*, etc.

- wannabe.j3d.loaders.vrml97 - Es con diferencia el paquete más estructurado en cuando a la existencia de una clase para cada uno de los aspectos más relevantes. Se han separado en diferentes paquetes las clases que cubren la especificación de VRML 1.0 de las que lo implementan usando Java3D, que encontramos en el paquete wannabe.j3d.loaders.vrml97.j3d. Así por ejemplo si *VRMLCylinder* es el nodo que construye un cilindro con unos determinados parámetros, *J3DCylinder* es la clase que llama a la primitiva *Cylinder* con dichas propiedades. Se añaden a estas clases las utilidades para el manejo de estructuras indexadas.

- wannabe.j3d.loaders.wavefront - Para la implementación de este paquete se ha considerado sobre todo la especificación ofrecida por Sun para su clase: com.sun.j3d.loaders.objectfile.ObjectFile, aunque no así su código, ya que distaba demasiado de la tendencia tomada por todos los cargadores de nuestra aplicación. Aquí también tenemos clases que agrupan las instancias de elementos leidos, como son: *FaceManager*, *GroupManager*, *VertexManager* y *VertexNormalManager* entre otros.

3.4 Versiones Software

Siguiendo un desarrollo basado en prototipos, se disponen de varias versiones de la aplicación, que han ido modificando considerablemente su aspecto y su funcionalidad.

El primer prototipo estaba disponible en Julio del 2000. Comenzó por llamarse DXFViewer, e incluía una interfaz simple con *java.awt,* y no empleaba Java3D, sino *java.graphics2D* Solo convertía el formato AutoCAD DXF.

Posteriores prototipos, ya de Wannabe Amazing, introdujeron interfaz *javax.swing* y la librería JAVA3D. Se amplió la gama de formatos aceptados y se incluyó la posibilidad de exportar la geometría de los modelos en formato GRF. Se puede habilitar y deshabilitar la luz directa situada en el observador, así como la ambiental para mejorar la visualización del modelo cargado. Dos posibilidades a la hora de cargar un modelo: Abrir/Añadir. Mediante el primero nos aseguramos que solo visualizamos el modelo que vamos a cargar. Con el segundo vamos añadiendo la nueva escena cargada a otras existentes. El inconveniente es que todas las escenas se sitúan en el mismo origen. Se añaden por defecto en el *VirtualUniverse* Flechas correspondientes a los ejes de coordenadas.

La aplicación queda dividida en dos ventanas: la que incluye los elementos de interfaz con el usuario, y la que muestra la escena en un *Canvas3D*. A esta última se la denomina *Grid*, y no aparecerá en la primera ejecución de la aplicación, ya que no encontrará los ficheros en donde queda almacenado el tamaño y la posición de dicha ventana (y de otras) en la anterior ejecución. Se utilizan tres ficheros texto con extensión **wbf** (wannabe file), que serán almacenados en el directorio props:

- para almacenar las preferencias del usuario: prefs.wbf
- los archivos mas recientemente grabados: recent.wbf
- la propiedades de la aplicación: props.wbf

3.4.1 Breve manual de usuario

Si nos disponemos a utilizar el programa, será necesario primeramente conocer cómo usarlo, qué es posible hacer desde los menús y lo más importante, cuáles son los pasos para importar y exportar un modelo.

Anteriormente se ha comentado que la aplicación utiliza dos ventanas. La primera contiene todos los elementos de interacción como menús y botones. La segunda se utiliza para visualizar el modelo y podremos interaccionar con este mediante las teclas de cursor del teclado y el ratón. Ambas ventanas son redimensionables y pueden ser situadas donde el usuario prefiera. Nos centraremos por tanto en los elementos de interfaz de la primera ventana, cuyo aspecto podrá ver el lector en la siguiente imágen.

Menú Archivo

- Abrir escena: Muestra el dialogo que nos permite seleccionar el archivo de escena a abrir. El directorio en el que nos sitúa es el más recientemente utilizado, y en su defecto el directorio "export", creado por la aplicación en el directorio de ejecución. El universo virtual se inicializa y se muestra tan solo el modelo que abrimos.
- Añadir escena: Es igual que el anterior, salvo que el modelo abierto se añade a otros modelos anteriores si los hay. Por defecto cada modelo se muestra centrado en el origen de coordenadas como si fuese el único modelo mostrado.
- Guardar: Procedimiento que permite salvar la información del modelo actual empleando el formato GRF. El archivo se generará en el mismo directorio, y con el mismo nombre que el último fichero importado, con la extensión como única diferencia. Los datos exportados corresponderán a una o más mallas cada una de las cuales estará formada por una lista de vértices y normales a los vértices, así como una lista de caras con los índices de los vértices empleados. Las luces añadidas, forman parte del fichero GRF. Datos como la dimensión del *Grid*, o el color de fondo se exportarán como comentarios.
- Guardar como... : Al igual que el apartado anterior, este menú se utiliza para generar un archivo de salida GRF. La diferencia estriba en que podremos indicar el directorio en el que se almacenará el archivo y el nombre que tomará. Por defecto se utiliza el directorio "exports" que se crea la primera vez que se ejecuta la aplicación.
- Archivos recientes: La aplicación guarda una entrada por cada archivo exportado, con la información de su ruta absoluta. El primero corresponde al más reciente, y la última entrada (la sexta) corresponde al más antiguo. Al seleccionarlo se abre una ventana que nos muestra (si se encuentra) el texto del fichero GRF exportado.
- Salir**:** Finaliza la aplicación. No hay ningún tipo de diálogo que avise sobre los últimos cambios realizados.

Menú Fondo

- Color Fondo: Disponemos de un conjunto de 16 colores predefinidos para poner como color de fondo (objeto *Background*) del universo virtual.
- Color Ambiente: Selección del color que va a ser utilizado como luz ambiental. Por defecto el view-branch-graph viene con una luz ambiental blanca y una luz direccional gris.
- Niebla: Se trata de un efecto interesante para la visualización de la escena. Por ahora la información del los parámetros de niebla que podamos escoger, no repercute en el formato de salida. Desde este submenú podremos añadir y eliminar niebla, así como modificar la densidad de esta.

- Imagen de fondo: Por defecto el universo aparece con fondo en negro y sin imagen, pero podemos escoger archivos de imagen en formato JPEG y GIF para sustituir el color de fondo. La imagen que escojamos podemos disponerla de forma centrada, mosaico o estirada. En cualquier momento podemos quitar la imagen para permitir que el fondo sea el color plano que hemos seleccionado en el Menú Fondo \ Color Fondo.

Menú Luces

No solo es importante disponer cargada una geometría en el programa, si no añadimos en el grafo de escena ninguna luz Java3D, no se verá nada renderizado. Por ello contamos por defecto con una luz ambiental y una direccional, que el usuario puede modificar, así como añadir nuevas luces a la escena. Lo primero que se nos pide cuando añadimos una luz es indicar el color de emisión de la fuente. Después las modificaciones de esta dependerán del tipo de luz añadida.

Las luces son también componentes wannabe, por lo que podremos seleccionarlos para trasladarlos, aplicarles zoom, etc. Sin embargo el usuario debe saber que por defecto este componente no aparece visible (para no interferir con la geometría de la escena) y que por tanto deberá ser seleccionado para permitir apreciar su posición dentro del mundo.

Los diferentes tipos de luces que Java3D nos ofrece los encontramos como submenús de Luces y son:

- Direccional
  - Añadir direccional
  - Cambiar dirección
  - Eliminar direccional
- Puntual
  - Añadir puntual
  - Seleccionar puntual. Necesario para visualizarla.
  - Alterar atenuación direccional. La atenuación puede ser constante, lineal o bien cuadrática
  - Eliminar puntual
- Foco
  - Añadir spot
  - Seleccionar spot. Necesario para visualizarla.
  - Alterar atenuación spot
  - Alterar ángulo spread
  - Alterar dirección

  - Alterar concentración
  - Eliminar spot

- Esconder luces: Cada luz adicional que añadimos a la escena es mostrada junto con el resto del modelo (excepto la direccional que está en el infinito), situada en un punto y con un color. Es posible que se confunda con otros elementos del modelo, por lo que el usuario puede decidir ocultar su posición.

- Eliminar todas las luces: Quita del grafo de escena las fuentes posteriores a la Ambiental y Directa que están colocadas por defecto.

Menú Visualización

- Reiniciar: Tras cargar el modelo, visualizamos a este de frente. Tras esto el usuario puede rotar, trasladar el modelo, así como aumentar y disminuir el zoom. Si aplicamos la opción de reiniciar, volveremos a encontrar el modelo colocado en la misma forma que cuando se cargó.

- Caminar: Comportamiento (Behavior) de ratón por el cual nos movemos por el mundo trasladándonos y girando, como si estuviésemos dentro de la escena. Similar al estilo de navegación "Walk" usado en los visores de VRML. Dependiendo del empleo del ratón podremos:

  - Clic izquierdo y movimiento horizontal: rotación sobre el eje Y.
  - Clic izquierdo y movimiento vertical: rotación sobre el eje Z.
  - Clic botón central (o modificador ALT) y movimiento horizontal: rotación sobre el eje Y.
  - Clic botón central (o modificador ALT) y movimiento vertical: rotación sobre el eje X.
  - Clic derecho de ratón (o modificador Ctrl) y movimiento horizontal: traslación sobre el eje X
  - Clic derecho de ratón (o modificador Ctrl) y movimiento vertical: traslación sobre el eje Y

- Examinar: Comportamiento (*Behavior*) de ratón que genera transformaciones en el estilo "Examine" de los visores de VRML. Es decir, el objeto se moverá como si estuviese situado a la distancia del brazo. El ratón originará distintas transformaciones dependiendo de:

  - Clic izquierdo y movimiento horizontal: rotación sobre el eje Y.
  - Clic izquierdo y movimiento vertical: rotación sobre el eje X.
  - Clic botón central (o modificador ALT) y movimiento vertical: rotación sobre el eje Z.

  - Clic derecho de ratón (o modificador Ctrl) y movimiento horizontal: traslación sobre el eje X.
  - Clic derecho de ratón (o modificador Ctrl) y movimiento vertical: traslación sobre el eje Y.

- Luz Directa: Enciende y apaga la luz directa incluida por defecto en el modelo. Esta luz es blanca y está dirigida en el sentido negativo del eje Z, tratando de iluminar por encima del observador.

- Luz Ambiente: Enciende y apaga la luz ambiental incluida por defecto en el modelo. Emite un gris (rgb(0.4,0.4,0.4)) con la misma intensidad en todas las direcciones.

- Sombreado: Cambia las propiedades de renderizado en tiempo real para alternar entre sombreado plano (caras con un único color) y sombreado de Gouroud.

- Sólido / Alambre: Cambia las propiedades de renderizado de los polígonos para mostrarlos en modo alambre (wireframe) o bien en modo sólido.

Menú Opciones

- Dimensión exacta de vista: Esta opción guarda relación con el tamaño con el que esté definido en la ventana *Grid* o *Canvas3D* con el que visualizamos el modelo. El diálogo nos permite modificar los valores de pixels para el alto y el ancho. Esto mismo se puede realizar con el ratón redimensionando la ventana *Grid*, como cualquier ventana del Sistema Operativo.

- Normales a las caras: El usuario puede decidir cómo se trabajará con las normales en el fichero de salida. En este caso se indica expresamente que las normales a los vértices no sean el promedio de las normales a las caras en las que intervienen, sino que sea directamente la normal de una de las caras.

- Promediar las normales: El la opción predeterminada del generador de fichero GRF de salida. El valor de la normal en un vértice será la media de las normales a las caras en las que el vértice interviene.

- Ajuste de escala: Modifica la escala aplicada al modelo.

- Poner todo por defecto: Restablece todos los parámetros por defecto de la aplicación.

- Obtener información. A través de un diálogo podemos obtener información del modelo cargado, o bien, de la Máquina Virtual Java.

Menú Ayuda

- Manual de Usuario: Muestra otra pantalla con la ayuda de la aplicación e Hipertexto. Se pueden usar los enlaces para navegar en manual del usuario de la aplicación. Esta ventana de ayuda se puede mantener durante la ejecución de Wannabe Amazing, ya que no interfieren.

- Acerca de ... Dialogo sobre la autora, versión y fecha de la aplicación así como del SDK empleado.

Barra de herramientas

Al igual que en otras aplicaciones, se añade una barra de herramientas que hace más cómoda la localización de algunas funciones. Y no solo eso, también encontramos otras funciones que no aparecen en los menús. Es mejor ver una imagen de la aplicación y comentar cada uno de estos botones conforme aparecen situados. Los tres primeros botones corresponden al menú Archivo y son Abrir, Añadir y Guardar, que ya han sido comentados con anterioridad.

Barra de componentes

Se puede añadir a la escena objetos gráficos simples, que proporciona J3D en el paquete *com.sun.util.geometry*, en lugar de abrir un fichero gráfico. Utilizamos para ello los botones de Componentes y Propiedades. Cuando hablamos de componentes nos referimos a primitivas geométricas básicas muy comunes en programas que trabajan en 3D. Es decir: Cubo, Rectángulo, Cono, Cilindro y Esfera. Además de estas formas básicas y como demostración de las posibilidades de Java3D. También se puede insertar en la escena Texto 2D y 3D (previa petición de un texto a mostrar) aunque estos dos actualmente no se exportan. Cada vez que insertamos un componente este se selecciona como activo.

Con el botón Propiedades podemos modificar las características del componente que esté activo. Tras pulsar el botón aparece un menú emergente con una serie de ítems que variará según el componente que vayamos a modificar. Desde el botón Eliminar Componente podemos quitar el componente activo, así como eliminar todos los componentes insertados (que no hayan sido importados desde un fichero).

Otros elementos de la barra de herramientas son los botones: Caja Englobante e Información. El primero de ellos añade a la escena un cubo tridimensional en alambre cuyos extremos coinciden con el punto mínimo y máximo de la escena al completo. Actualmente la caja es un componente y para dejar de visualizarlo hay que eliminarlo como tal. Mediante el segundo botón, podemos obtener la información del modelo cargado y/o los componentes wannabe añadidos. Incluye el número de mallas exportadas, así como el número de vértices, caras y los datos de la caja englobante para cada malla.

Por debajo de la barra de herramientas tenemos un indicador de la frecuencia de repintado del canvas que muestra la escena. Esta etiqueta aparece con el nombre de fps: y le sigue un valor que corresponde a los frames por segundo mostrados. También nos encontramos con un cuadro de texto que nos informa de los componentes que estén seleccionados en cada momento. Para distinguir unos

casos de otros se ha utilizado una notación con color. El fondo amarillo indica selección y un fondo blanco corresponde a la ningún componente seleccionado. El texto indica qué componente se ha seleccionado.

Barra de ajuste y Barra de rotación

Dentro de la primera tenemos un display y botones "Menos", "Más", y "OK". Siempre aparecen deseleccionados salvo cuando el usuario ha de introducir un valor numérico en cuyo caso puede utilizar los botones +/- para incrementar o decrementar en una cantidad determinada el valor que se presenta como el actual. El botón "OK" introduce el nuevo valor en la variable que está siendo editada. La segunda suele utilizarse cuando modificamos parámetros de las luces que hemos añadido a la escena, como son la dirección del foco.

Todos los botones presentan una descripción de la función que realizan cuando el ratón se sitúa sobre estos.

3.5 Uso de la aplicación

3.5.1 Requerimientos hardware

Los únicos requerimientos para ejecutar la aplicación es el disponer instalado en el sistema la Maquina Virtual Java (JVM), así como la librería JAVA3D que viene, de forma independiente (por ahora), como una serie de ficheros jar, localizados dentro del directorio jre (Java Runtime Environtment).

Navegadores como el Microsoft Internet Explorer y Netscape Navigator pueden mostrar aplicaciones escritas usando Java y Java3D, pero es necesario descargar un plug-in especial para esto. Lo más fácil es disponer del entorno de ejecución Java2 instalado y utilizar el interprete de éste.

Actualmente trabajo con un *AMD K6 "3D Processor"* a 350MHz y 64MB de RAM, sin embargo para obtener unos tiempos razonables se recomienda un Pentium III con 256MB de RAM.

3.5.2 Ejecución de la aplicación

Para ejecutar Wannabe Amazing, no es necesario cambiar nada de las variables de entorno de Java (CLASSPATH), lo único que necesitamos es localizar los ejecutables en el path añadiendo el subdirectorio "bin" del jdk1.3.0. Ahora podremos usar los siguientes comandos:

- java ClasePrincipal para lanzar una aplicación java
- java -jar Aplicacion.jar para lanzar una aplicación comprimida con jar
- appletviewer claseprincipal.html para lanzar el applet que este indicado dentro del fichero html, utilizando la etiqueta <APPLET CLASS= WIDTH= HEIGHT= ></APPLET>

- javac ClasePrincipal.java para compilar la clase y todas las que de esta dependan que hayan sufrido alguna modificación
- jar -x Aplicacion.jar para descomprimir una aplicación comprimida con jar

En el caso de WannabeAmazing emplearemos: java -jar wannabeamazing.jar

En cuanto a las necesidades del equipo recomendado, resaltamos la importancia de la memoria principal, quizás mas relevante que la velocidad de reloj del procesador. De todas formas si no se le indica al interprete en la línea de comandos, el interprete siempre reservará 32MB para la ejecución, independientemente de la cantidad de memoria libre en el sistema. Conviene pues, emplear el modificador -Mmx64m para aumentar el tamaño a 64MB u a otra cantidad deseada.

3.6 Referencias

3.6.1 Web

Un buen modo de comenzar con Java3D es "Introduction to Programming with Java 3D". Tutorial realizado en colaboración con Sun y San Diego Supercomputer Center: *http://www.sdsc.edu/~nadeau/Courses/SDSCjava3d/*

La referencia obligada es la pagina de Sun, con tutoriales (Java 3D Guide), así como suscripciones a listas de distribución mantenidas por los mismo programadores de Sun. *http://java.sun.com/products/java-media/3D/*

Por otro lado, diversas comunidades web independientes de Sun incluyen tutoriales, fuentes, artículos, y referencias a otras webs. Entre estas están:

- Sobre Java 3D:
  - http://web3d.about.com/compute/web3d/msubjava3d.htm
  - http://gladstone.uoregon.edu/~bbarrett/bookmarks.html
  - El centro de información de VRML y Java3D de la Universidad de Leeds. *http://www.comp.leeds.ac.uk/vrmljava3d*
  - The 3D Web Repositori. *http://www.web3d.org*
  - Java3D Comunity Site. *http://www.j3d.org*
  - Columnas en JavaWorld sobre programación escritas por Bill Day. *http://www.javaworld.com/topicalindex/jw-ti-media.html*
  - Documento de especificación de la API, en formato pdf y html. *http://java.sun.com/products/java-media/3D/j3dguide/j3dTOC.doc.html*
  - La lista Java 3D FAQ mantenida por Steve Pietrowicz, del grupo Java 3D en NCSA. *http://tintoy.ncsa.uiuc.edu/~srp/java3d/faq.html*
  - Tutoriales y recursos sobre Java en Java Land: *http://www.epm.ornl.gov*

- Formatos gráficos y modelos
  - The programmer's File Format Collection. *http://www.wotsit.org*
  - Centro de recursos para el Arte y el Diseño de la Universidad de Teesside. *http://vr3.tees.ak.uk/rachael*
  - Repositorio gratuito de modelos. *http://www.3dcafe.com*
- Sobre OpenGL:
  - *http://www.eecs.tulane.edu/www/Terry/OpenGL/Introduccion.html*
  - Sitio oficial. *http://www.opengl.org*

3.6.2 Libros

No es fácil encontrar libros orientados a Java3d (no es el caso de OpenGL), pero se pueden comentar algunos:

Título: Java 3D Programming, Daniel Selman (Julio 2000) , Manning Publications Company; ISBN: 188477797X

Título: Especificación de la API Java3d 1.3 Alpha en formato pdf, que se puede descargar de la web de Sun

Título: "Java for 3D and VRML Worlds". Rodger Lea, Kouichi Matsuda y Ken Miyashita. Ed. New Riders Publishing, Indianapolis, Indiana. 1996. ISBN: 1-56205-689-1.

Tìtulo: "Al Descubierto Java 1.0". Traducción de "Java Unleased", Ed. Sams.Net. 1996. ISBN: 1-57521-049-5

Título: "Maximum Java 1.1". Glenn Vanderburg, et al. Ed. Sams.Net Publishing, Indianapolis, Indiana. 1997. ISBN: 1-57521-290-0

Título: "Descubre Java 1.2". Mike Morgan. Ed. Prencite Hall, 1999. ISBN: 84-8322-142-X

La especificación de la API también esta disponible en formato impreso por la editorial Addison-Wesley dentro una Serie para Java. Para más información se puede consultar la siguiente dirección:
http://www.amazon.com/exec/obidos/ASIN/0201325764/billday/

Capítulo 4

Conclusiones y trabajos futuros

4.1 Conclusiones

A estas alturas del Programa de Doctorado: *Métodos y Técnicas avanzadas del desarrollo del software (1999/2001),* nos sentimos muy contentos de como se está llevando a cabo la integración de nuestro trabajo con con miembros del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada, así como por el grado de formación adquirido en el plano investigador y docente.

Con la realización de este trabajo de investigación se pretende cumplir los objetivos propuestos al inicio del Programa de Doctorado. En lo referente a la realización de una herramienta para la conversión de distintos formatos comerciales a un formato interno, los resultados obtenidos pudeden resumirse en:

1. Facilita el diseño de modelos geométricos, al permitirnos utilizar el trabajo hecho desde otras herramientas.

2. Realiza la conversión de datos de formatos comerciales, exportándolos a un formato legible por el sistema GIRT.

3. Actualmente está siendo empleada en la simulación de la iluminación para escenas complejas, como parte del trabajo del grupo de investigación.

4. Emplea un formato propio en la descripción de la información geométrica de la escena, lo cual nos permite que se ajuste de forma específica a nuestras necesidades. Si las necesidades cambian, nuestro formato también cambia.

5. Utiliza una interface amigable y sin necesidad de modificar el código, está sujeto a las mejoras que Sun realice en la Máquina Virtual Java y en el núcleo de clases de la librería Java3D.

6. Aporta un banco de pruebas para futuros trabajos.

4.2 Trabajos futuros

La línea de trabajo que se abre camino desde ahora está íntimamente relacionada con los métodos empíricos para generar funciones de distribución de reflectancia bidireccionales. En concreto el estudio de una BRDF óptima que saque provecho de la sensibilidad humana, pasa por una serie de fases:

- Estudio de los modelos actuales, viendo las aportaciones que realizan a los modelos clásicos de reflexión
- Propuesta en su caso, de un modelo de BRDF que mejore en la medida de lo posible las deficiencias de los modelos estudiados.
- Puesta en práctica del modelo de BRDF propuesto. El desarrollo de una herramienta que permita asignar valores de BRDF a las superficies de los objetos de una escena para su posterior visualización empleando algoritmos de iluminación global puede sernos de utilidad para:
  - obtener imágenes sintéticas de alto realismo
  - comparar la BRDF propuesta con los modelos estudiados, viendo que realmente los mejora
  - utilizar esta información como retroalimentación para la mejora de la BRDF

Adviértase que la valoración de la BRDF está sujeta a la objetividad o subjetividad de la persona que observe los resultados. Esta persona podrá indicar si es o no buena la BRDF evaluada, pero no cómo de buena es. Se ve por tanto necesaria la búsqueda de una métrica adecuada para la valoración de esta función, ajustándose a las características específicas de esta como son:

- reciprocidad o simetría
- conservación de la energía

En resumidas cuentas, nos queda por completar el estudio sobre la Reflexión local de la luz y las diferentes funciones empleadas para describir la reflectancia en un punto de la superficie. Se extraerá las características deseables de una **Función Bi-direccional de Distribución de la Reflectancia** (BRDF) para disponer de mejores resultados. Posteriormente ampliaremos nuestra herramienta para la evaluación de BRDFs y poder así comprobar la incidencia de la BRDF en la percepción de las superficies.

Conscientes del trabajo y el esfuerzo que la implica, nos planteamos en estos momentos continuar en el futuro con el estudio de estas ideas y el desarrollo de modelos aplicables a la síntesis realista de imágenes.

Bibliografía

[1] *Arvo, J. "The Role of Functional Analysis in Global Ilumination". Program of Computers Graphics. Cornell University*

[2] *Arvo, J. "Backward Ray Tracing". Course notes of Computers Graphics. Cornell University. 1986*

[3] *Ashdown, I. "Radiosity: A Programmer's Perspective" New York John Wiley & Sons, Inc. 1994.*

[4] *Ashikhmin M. "A Microfacet-based BRDF Generator". University of Utah. Siggraph 2000*

[5] *Cohen, Wallace. "Radiosity and Realistic Image Synthesis" Ed. Academic Press, 1993.*

[6] *Glassner. "Principles of Digital Image Synthesis". Ed. Morgan Kaufmann Publisher, 1995.*

[7] *Foley, van Dam, Feiner, Hughes. "Computer Graphics: Principles and Practice (2nd. Ed. in C)". Ed. Addison-Wesley Publishing Company, 1993.*

[8] *Varios autores. "Computer Graphics Proceedings. Annual Conference Series." Ed. ACM SIGGRAPH (ACM Special Interest Group on Computer Graphics). Desde 1980 hasta la actualidad.*

[9] *Varios Autores. "Rendering Techniques". Proceedings EG Workshop on Rendering Ed. Springer Verlag, años 1994 hasta 1999.*

[10] *Heckbert P.S." Adaptative Radiosity Textures for Bidirectional Ray Tracing", Computer Graphics, 24 (4), august 1990. ACM Siggraph'90 Conference Proceedings.*

[11] *Hill, F.S. "Computer Graphics. Using OpenGL (2nd. Ed). Ed. Prentice Hall*

[12] *Kajiya J.T. "The Rendering Equation". Computers Graphics, 20(4):143-150, 1986. ACM Siggraph'86 Conference Proceedings*

[13] *Lastra, M. "Sintesis Eficiente de Imágenes Fotorealistas por Simulación del Modelo de Partículas de la Luz". Trabajo de investigación. Departamento de Lenguajes y Sistemas Informáticos. Universidad de Granada. 2000*

[14] *Lastra, M.; Marín, J.J; Moreno, J., Revelles, J. "Estudio comparativo de métodos de optimización de Ray Tracing basadas en subdivisión espacial uniforme y no uniforme". Internal report LSI-1996-1*

[15] *Neumann, L; Bekaert, P. "Radiosity with Well Distributed Ray Sets". Eurographics'97*

[16] *Nishita T.; Dobashi Y. "Display of Clouds Taking into Account Multiple Anisotropic Scattering and Sky Light"*

[17] *Pharr, M.; Hanrahan, P. "Monte Carlo Evaluation Of Non-Linear Scattering Equations For Subsurface Reflection". Stanford University. Siggraph 2000*

[18] *Pattananaik, S.; Greenberg, D. "Time-Dependent Visual Adaptation For Fast Realistic Image Display". Cornell University. Siggraph 2000*

[19] *Revelles, J.; Torres, J.C. "Acceleration Techniques Formalization based on Spatial Indexing". Internal report LSI-1996-2*

[20] *Revelles, J.; Ureña, C.; Lastra, M. "An Efficient Parametric Algorithm for Octree Traversal". University of Granada. 2001*

[21] *Rusinkiewicz, S. "A Survey of BRDF Representation for Computers Graphics". CS348c. Universidad de Standford. 1997*

[22] *Smints, B.; Wann, H. "Global Illumination Test Scenes". University of Utah Technical Report UUCS-00-013. 2000*

[23] *Sillion François X., Puech C. Morgan Kaufmann Publishers, inc. "Radiosity & Global Illumination". San Francisco, California. 1994*

[24] *Ureña, C. "Métodos de Monte-Carlo Eficientes para Iluminación Global". Tesis doctoral. Departamento de Lenguajes y Sistemas Informáticos. Universidad de Granada. 1998*

[25] *Ureña, C. "Computation of Irradiance from Triangles by Adaptative Sampling". Internal report LSI-2000-1*

[26] *Ureña, C.; Parets, J.; Torres, J.C.; del Sol, V. "An Object-Oriented aproach to Ray Tracing Image Synthesis implementations". Computers & Graphics Vol. 16, No. 4, pp. 363-368, 1992*

[27] *Ureña, C.; Torres, J.C. "A Bidirectional Monte Carlo Method for Image Rendering". Internal report LSI-1993-4*

[28] *Ureña, C.; Torres, J.C. "Improved Final Gather for Radiosity with Importance Sampling". Internal report LSI-1997-3*

[29] *Ureña, C.; Torres, J.C.; Revelles, J.; Cano, P.; del Sol, V.; Cabrera, M. "GIRT: Un sistema orientado a objetos para simulación precisa de la iluminación". Internal report LSI-1997-1*