Sistemas Informáticos
Curso 2002-03

Motor gráfico para el desarrollo de videojuegos.

Autores:

Pablo José Beltrán Ferruz
Alfonso del Cerro Aguilar
Daniel Cuenca Pascual
Gustavo García Aceituno

Dirigido por:
Prof. Purificación Arenas Sánchez
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

Resumen

El objetivo del proyecto es el diseño e implementación de un motor gráfico y sonoro para realizar un juego en un entorno tridimensional. El motor soporta la visualización de objetos en tres dimensiones, de forma optimizada, mostrando sólo los elementos que entran en el campo de visión de la cámara. También es capaz de manejar animaciones, efectos de luces y sombras, sonido en 3D y música. El motor también proporciona funciones para la detección y manejo de colisiones entre objetos, de forma sencilla, eficiente y transparente para el programador.

El motor es independiente de la librería gráfica utilizada, pero solamente se ha implementado para DirectX.

Paralelamente, se ha creado un videojuego sencillo que muestra las prestaciones del motor gráfico implementado.

Abstract

The aim of this project is to design and implement a 3D graphic and sound engine to build a game in a three dimensional environment. The engine allows the rendering of three dimensional objects in an optimized way, drawing only those elements that fit in the camera's field of view. It also allows animations and light and shadow effects, together with 3D sound and music. The engine also provides collision detection and handling functionality in a simple, efficient and transparent way for the programmer.

The engine does not depend on the graphics library used, the current implementation uses DirectX.

We have also implemented a simple videogame that shows the engine's capabilities.

Palabras clave: juego, gráfico, motor 3D, árboles BSP, colisiones, lista de visibilidad, DirectX, sonido, proyección de sombras, mallas animadas.

Índice

1.- Prefacio

El desarrollo de este proyecto surgió como idea tras cursar el año pasado la asignatura de informática gráfica. Dicha asignatura despertó en nosotros un interés especial por el desarrollo de los gráficos en el PC.

Planteando como idea inicial la curiosidad por implementar nosotros mismos algunos de los algoritmos que habíamos estudiado en clase y otros que habíamos leído en los libros, también nos suscitaba interés el poder desarrollar un videojuego al estilo de los antiguos programadores de videojuegos, es decir, un grupo de amigos que sin ningún tipo de gran estructura como pueden tener las actuales empresas, se "plantan" delante de sus ordenadores y a partir de las librerías existentes llegan a desarrollar un videojuego.

Tras esta idea inicial nos dirigimos a hablar con las profesoras que impartieron la asignatura de informática gráfica y empezamos a concretar lo que iba a ser nuestro proyecto. A lo largo de la conversación se notaba la ambición por realizar algo que mereciera la pena, pero no faltaron los comentarios de realismo. Por un lado estaban las ideas de realizar un gran videojuego que incluyera características tales como el gran detalle en las animaciones de los personajes, con las sombras y las luces, la posibilidad de multijugador, el uso de inteligencia artificial para los personajes controlados por el ordenador y algunas otras cosas más. Por otro lado nos planteamos la realidad de las limitaciones que puede tener un proyecto de este estilo y observamos lo realizado por otros grupos en años anteriores que pudiera parecerse a esto. Nos encontramos que los proyectos que se han realizado sobre este tema tenían la característica de tratarse de paseos virtuales con una cámara por un escenario.

Finalmente pensamos que un grupo formado por cuatro integrantes podía llegar a plantearse algo más ambicioso que dar un paseo por un escenario y nos fijamos los objetivos que expondremos más adelante.

Otro de los asuntos que tratamos de afrontar inicialmente era las herramientas que íbamos a emplear para el desarrollo de nuestro proyecto. En la asignatura que habíamos estudiado empleábamos C++ Builder como compilador y OpenGL como librería gráfica. Sin embargo nosotros optamos por emplear Visual C++ como compilador y DirectX como librería gráfica por dos motivos fundamentalmente. El primero de ellos, aunque no tiene porque serlo por importancia, es que en DirectX existían una serie de herramientas que nos permitían importar los gráficos realizados en el 3D Studio (aunque posteriormente descubrimos que no funcionaban bien y hemos tenido que implementar nosotros mismos alguna de estas utilidades) y el segundo motivo era aprender

otra de las técnicas que se emplean ampliamente en el desarrollo de videojuegos.

Uno de los asuntos que también nos costó decidir fue si íbamos a utilizar algunos conceptos de inteligencia artificial en el desarrollo del proyecto, pero dejamos esta opción casi por descartada ya que consideramos que nos faltaría tiempo.

Finalmente, lo que en las páginas que siguen se muestra es el trabajo resultante de estas ideas.

Captura gráfica del videojuego implementado con nuestro motor gráfico

2.- Introducción

Nuestro objetivo principal a la hora de abordar este proyecto fue la realización de un motor 3D útil para la creación de videojuegos. Este motor debe proporcionar a las aplicaciones que lo usen las funcionalidades básicas que requieren los juegos en 3D, que son programas cada vez más complejos; y que estos servicios sean fácilmente utilizables. Dichas funcionalidades son:

- Escenarios y personajes en 3D
- Movimientos articulados de los personajes
- Efectos de luces y sombras
- Diseño de los gráficos con el 3D Studio y capacidad para importarlos a la aplicación
- Uso de las librerías gráficas de DirectX 8
- Manejo de las colisiones entre los objetos media estructuras complejas que se busquen el libros
- Utilización de algoritmos de visibilidad para reducir el numero de polígonos que se dibujan

Sin embargo, también hemos desarrollado un pequeño programa a modo de juego, acompañando a nuestro motor. Este programa tiene dos finalidades:

- Demostrar el funcionamiento correcto de nuestro motor, incorporando todas sus características de una forma visible.

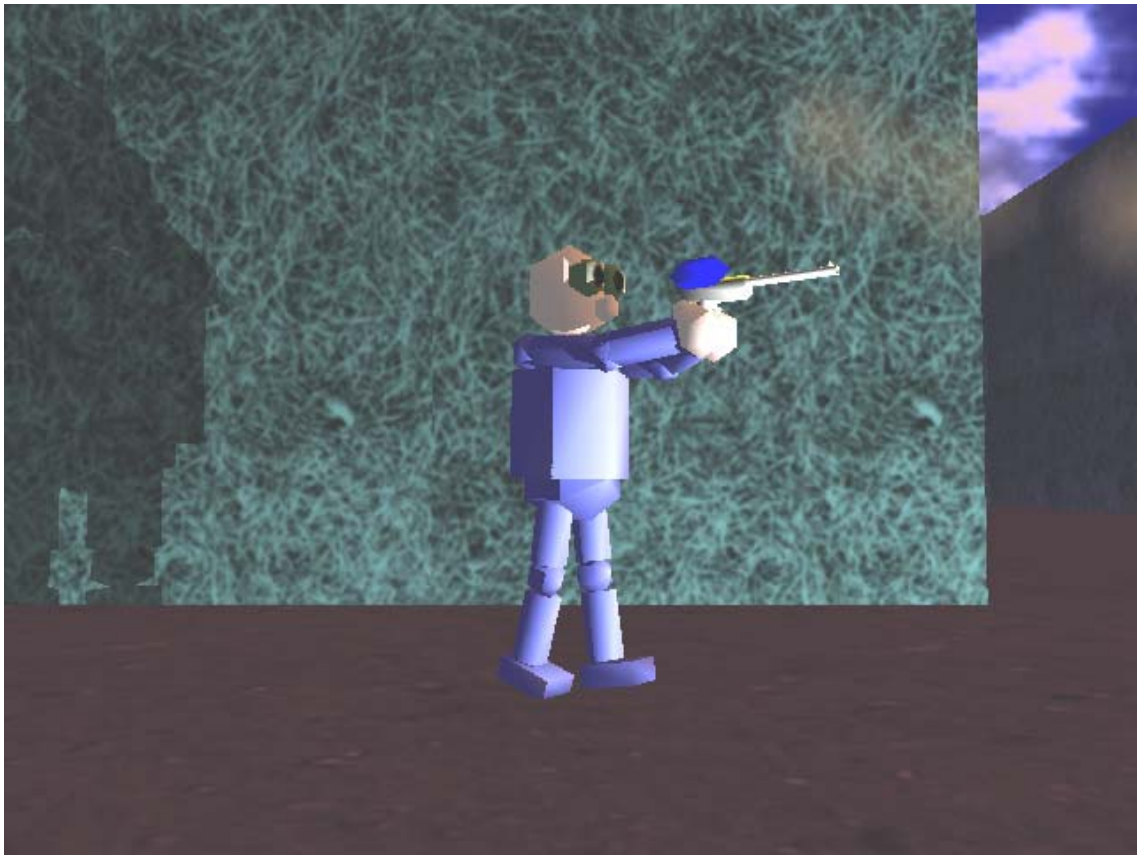
- Servir de ejemplo de uso de nuestro motor, que los clientes que lo empleen puedan tener como referencia y punto de partida.

Este programa consiste básicamente en lo siguiente. El jugador controla a un personaje en un entorno tridimensional, usando la interfaz típica de los juegos en tercera persona, en los que se combina el uso del teclado (las flechas de los cursores) y del ratón. Como en estos juegos, la cámara sigue al personaje en su movimiento por el escenario.

El escenario consiste en un laberinto formado por setos altos, situado en exteriores, con un cielo con nubes como fondo.

Además de pasear por el escenario, comprobando el correcto funcionamiento de la iluminación, la animación del muñeco, la colisión con las paredes, etc., el personaje lleva dos armas diferentes que puede disparar, con dos diferentes proyectiles de pintura, usando el botón izquierdo del ratón. Puede cambiarse de arma con el botón 0. Además, es posible lanzar una bengala con el botón derecho. Esta bengala ilumina lo que la rodea y permite observar también el sistema de proyección de sombras.

Esperamos que este pequeño programa sirva como una demostración vistosa y entretenida de las posibilidades del motor que hemos desarrollado.



Captura gráfica del videojuego realizado con nuestro motor gráfico

3.- Diseño

Desde el comienzo del proyecto discutimos ampliamente cómo debía ser el diseño de nuestra aplicación, ya que ello iba a repercutir ampliamente tanto en la facilidad para programar, como en la capacidad para poder repartirnos el trabajo de modo que cuatro personas pudieran estar trabajando simultáneamente sin que nadie estropeara el trabajo de los demás. Y lo que también resultaba muy importante, era conseguir un alto nivel de eficiencia, pues era altamente probable que hasta que no estuviéramos en un punto avanzado del proyecto no pudiéramos comprobar que la velocidad conseguida era satisfactoria o si por el contrario iba a resultar un programa totalmente absurdo, dado que al plantearse como un juego en tiempo real no se puede permitir que el tiempo de reacción del sistema fuera lento.

A continuación se muestra la historia del diseño de este proyecto que ha pasado por diversas etapas, en las que, en cada momento primaba un aspecto del diseño.

3.1.- Diseño versión 1

En esta primera aproximación del diseño hemos tratado de dar cabida a todos los elementos de los que estará compuesto nuestro juego, sin atenernos a grandes detalles ni a las cuestiones de eficiencia en la implementación.

Como podemos observar en el gráfico, planteamos dar cabida a la música, a las luces... Pero si nos centramos en cómo se va a implementar la escena, podemos ver que vamos a tener un escenario y éste se va a encargar de tratar con todos los objetos que lo integran. Podemos ver dos estructuras de datos que van a contener la información sobre el escenario. Por una parte el árbol BSP, que nos va a permitir detectar las colisiones, y por otra parte la tabla de regiones que va a contener la información gráfica para dibujar la pantalla.

En la figura 3.1 podemos ver las clases que hemos ido creando en esta versión del diseño. También podemos ver que ya tienen métodos y relaciones entre las clases. Se trata de una primera versión que nos permite ver los fallos y las limitaciones, pero a la vez se pueda compilar.

Ya a partir de este diseño podemos obtener algunas conclusiones que son algunas de las divisiones claras. Podemos ver que tendremos por un lado el tratamiento de las regiones y por otro, todo lo referente a los polígonos, mallas, etc.

Hemos de tener en cuenta que en este primer diseño hemos pasado bastante por alto la estructura de datos necesaria para implementar los árboles BSP, parte importante de nuestro trabajo.

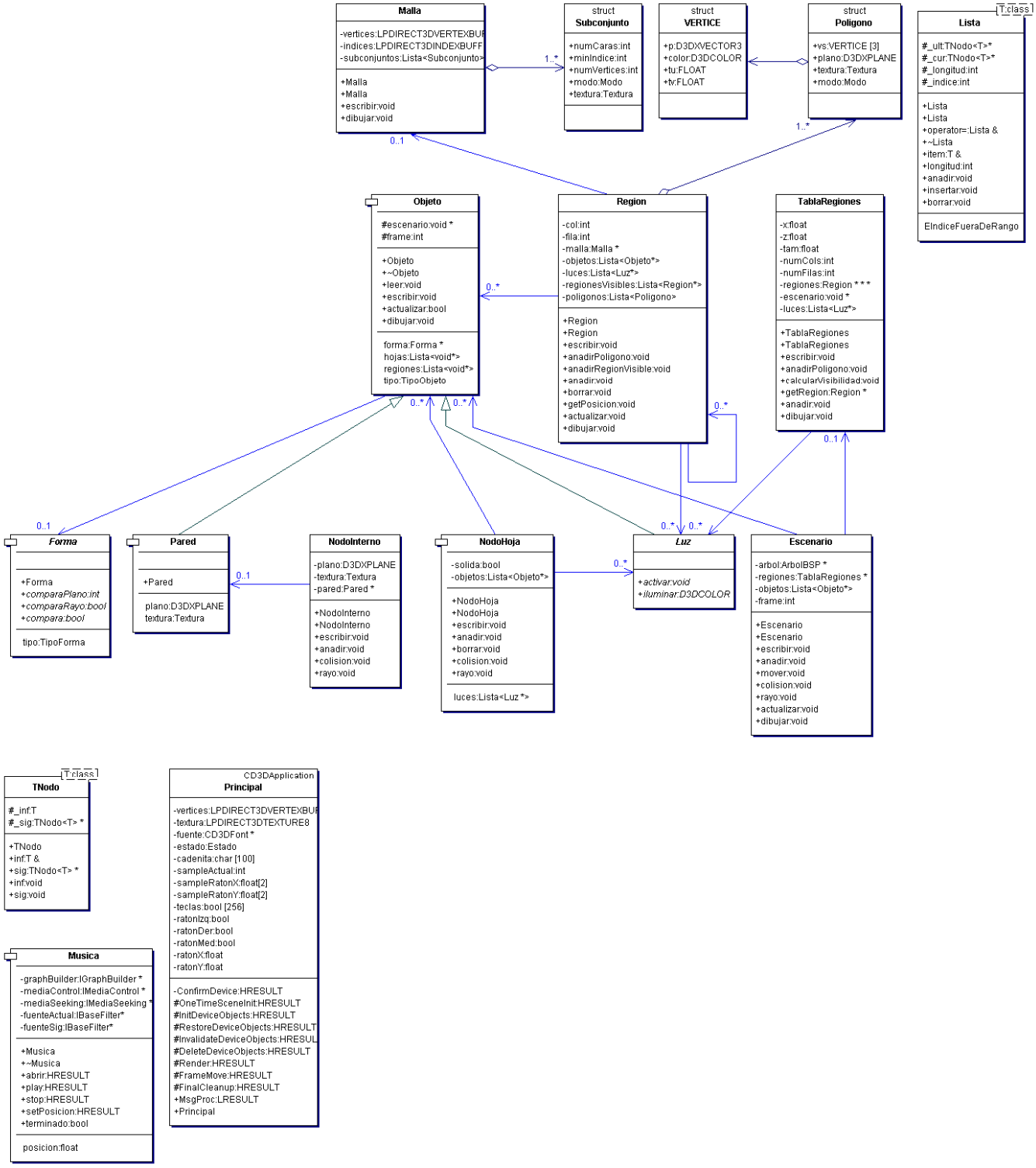

Malla
-vertices:LPDIRECT3DVERTEXBUF
-indices:LPDIRECT3DINDEXBUFF
-subconjuntos:Lista<Subconjunto>
+Malla
+Malla
+escribir:void
+dibujar:void
struct
Subconjunto
+numCaras:int
+minIndice:int
+numVertices:int
+modo:Modo
+textura:Textura
struct
VERTICE
+p:D3DXVECTOR3
+color:D3DCOLOR
+tu:FLOAT
+tv:FLOAT
struct
Poligono
+vs:VERTICE [3]
+plano:D3DXPLANE
+textura:Textura
+modo:Modo
T:class
Lista
#_ult:TNodo<T>*
#_cur:TNodo<T>*
#_longitud:int
#_indice:int
+Lista
+Lista
+operator=:Lista &
+~Lista
+item:T &
+longitud:int
+anadir:void
+insertar:void
+borrar:void
EIndiceFueraDeRango
1..*
0..1
Objeto
#escenario:void *
#frame:int
+Objeto
+~Objeto
+leer:void
+escribir:void
+actualizar:bool
+dibujar:void
forma:Forma *
hojas:Lista<void*>
regiones:Lista<void*>
tipo:TipoObjeto
Region
-col:int
-fila:int
-malla:Malla *
-objetos:Lista<Objeto*>
-luces:Lista<Luz*>
-regionesVisibles:Lista<Region*>
-poligonos:Lista<Poligono>
+Region
+Region
+escribir:void
+anadirPoligono:void
+anadirRegionVisible:void
+anadir:void
+borrar:void
+getPosicion:void
+actualizar:void
+dibujar:void
TablaRegiones
-x:float
-z:float
-tam:float
-numCols:int
-numFilas:int
-regiones:Region ***
-escenario:void *
-luces:Lista<Luz*>
+TablaRegiones
+TablaRegiones
+escribir:void
+anadirPoligono:void
+calcularVisibilidad:void
+getRegion:Region *
+anadir:void
+dibujar:void
0..*
0..1
Forma
+Forma
+comparaPlano:int
+comparaRayo:bool
+compara:bool
tipo:TipoForma
Pared
+Pared
plano:D3DXPLANE
textura:Textura
NodoInterno
-plano:D3DXPLANE
-textura:Textura
-pared:Pared *
+NodoInterno
+NodoInterno
+escribir:void
+anadir:void
+colision:void
+rayo:void
NodoHoja
-solida:bool
-objetos:Lista<Objeto*>
+NodoHoja
+NodoHoja
+escribir:void
+anadir:void
+borrar:void
+colision:void
+rayo:void
luces:Lista<Luz *>
Luz
+activar:void
+iluminar:D3DCOLOR
Escenario
-arbol:ArbolBSP *
-regiones:TablaRegiones *
-objetos:Lista<Objeto*>
-frame:int
+Escenario
+Escenario
+escribir:void
+anadir:void
+mover:void
+colision:void
+rayo:void
+actualizar:void
+dibujar:void
T:class
TNodo
#_inf:T
#_sig:TNodo<T> *
+TNodo
+inf:T &
+sig:TNodo<T> *
+inf:void
+sig:void
CD3DApplication
Principal
-vertices:LPDIRECT3DVERTEXBUF
-textura:LPDIRECT3DTEXTURE8
-fuente:CD3DFont *
-estado:Estado
-cadenita:char [100]
-sampleActual:int
-sampleRatonX:float[2]
-sampleRatonY:float[2]
-teclas:bool [256]
-ratonIzq:bool
-ratonDer:bool
-ratonMed:bool
-ratonX:float
-ratonY:float
-ConfirmDevice:HRESULT
#OneTimeSceneInit:HRESULT
#InitDeviceObjects:HRESULT
#RestoreDeviceObjects:HRESULT
#InvalidateDeviceObjects:HRESUL
#DeleteDeviceObjects:HRESULT
#Render:HRESULT
#FrameMove:HRESULT
#FinalCleanup:HRESULT
+MsgProc:LRESULT
+Principal
Musica
-graphBuilder:IGraphBuilder *
-mediaControl:IMediaControl *
-mediaSeeking:IMediaSeeking *
-fuenteActual:IBaseFilter*
-fuenteSig:IBaseFilter*
+Musica
+~Musica
+abrir:HRESULT
+play:HRESULT
+stop:HRESULT
+setPosicion:HRESULT
+terminado:bool
posicion:float

Figura 3.1

3.2.- Diseño versión 2

En esta segunda aproximación del diseño hemos pasado a estudiar un poco más en profundidad el diseño de las estructuras de datos que van a formar la escena. De esto que podamos ver las clases *ÁrbolBSP* y *TablaRegiones* dentro de una jerarquía que nos va dando pistas de cómo va a ser la implementación de estas clases.

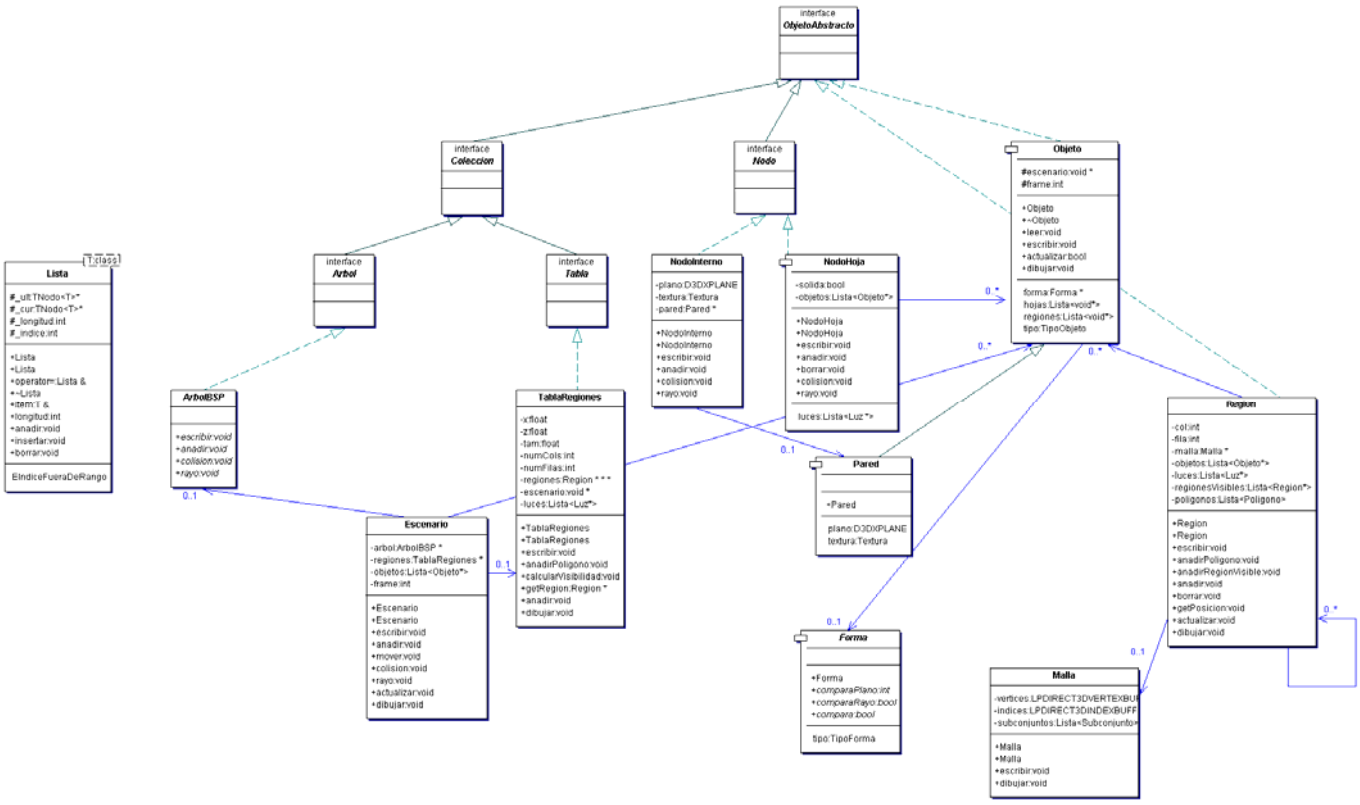


Figura 3.2

En esta figura ya podemos ver como se distinguen dos tipos de estructuras de datos. Por un lado vamos a tener una estructura de árbol para la implementación de los BSP y por otro tablas multidimensionales para la implementación de las regiones.

La funcionalidad de la estructura de datos de los BSP es permitir la detección de colisiones entre objetos, mientras que las tablas de regiones nos van a permitir hacer una selección de los polígonos que se van a mandar dibujar, de modo que cuantos menos polígonos mandemos dibujar mayor será la velocidad de la aplicación.

3.3.- Diseño versión 2.1

Esta versión es una ampliación de la anterior, en la que vamos detallando algunos de los métodos que contiene cada clase y las relaciones que hay entre las diversas clases.

Además hemos añadido una nueva clase que es el *Frustrum* y va a representar el campo de visión.

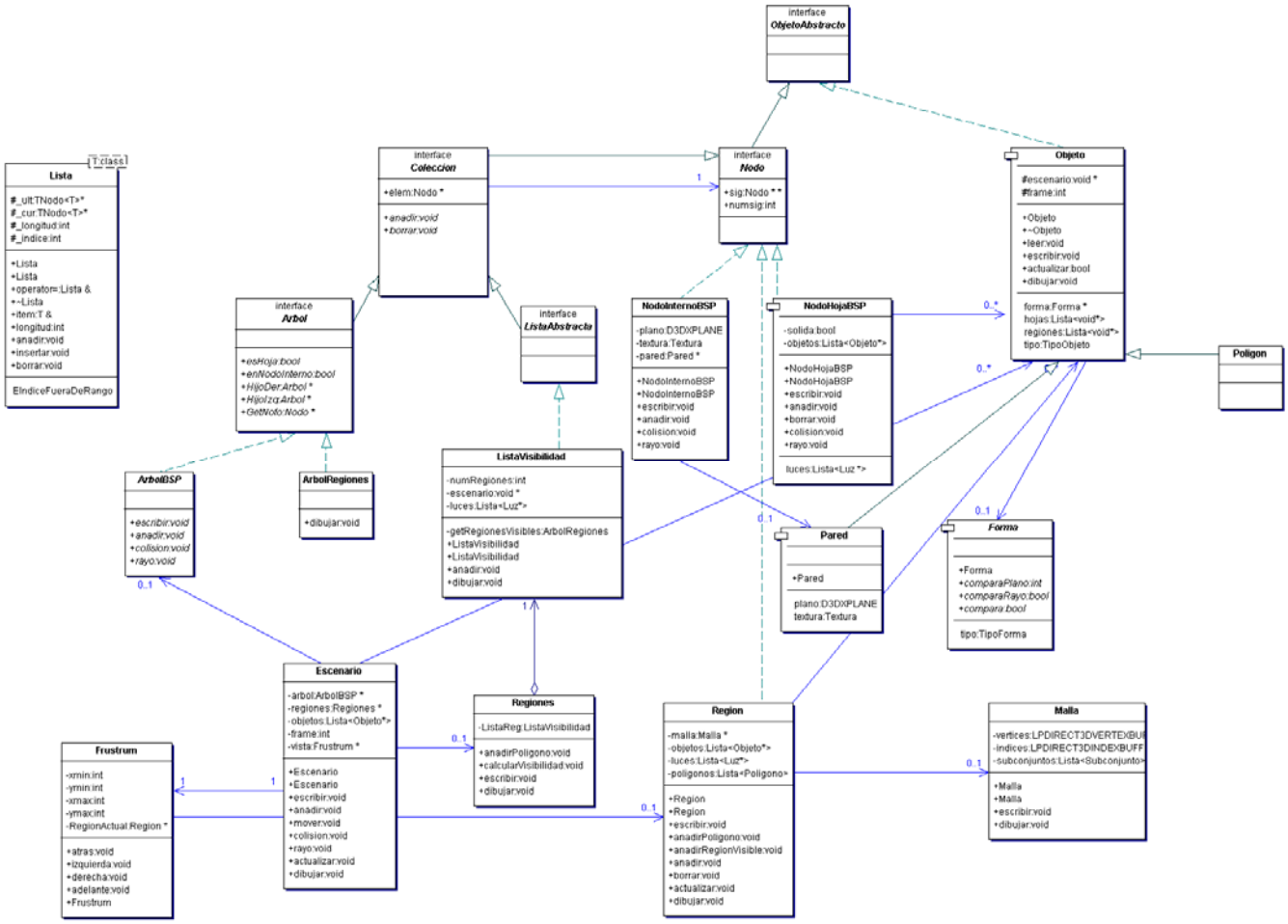


Figura 3.3

Podemos ver aquí las distintas relaciones que hay entre las clases. La clase escenario es la que lleva el control de todas las demás clases y ya nos podemos ir haciendo una idea de cómo van a ir interactuando unas clases con otras.

En primer lugar el escenario, en su constructor, creará las estructuras fundamentales que son: el *ArbolBSP* y la *ListaVisibilidad.* Después todo el proceso consistirá en llamar a la función actualizar, que realizará las operaciones necesarias para mover los objetos que se encuentren en movimiento, realizará las comprobaciones necesarias para evitar las colisiones mediante el *ArbolBSP* y luego dibujará las regiones que sean visibles haciendo uso de la *ListaVisibilidad*.

3.4.- Diseño versión 3

Llegamos ya a una fase clave del diseño, a partir de ahora vamos a comenzar a implementar. Empezamos a dividir el diseño en una serie de paquetes. Por un lado vamos a tener un paquete al cual llamaremos *Paquete3D* y que se va a encargar de proporcionar una interfaz con DirectX, de modo que el resto del programa sea prácticamente transparente al uso de DirectX.

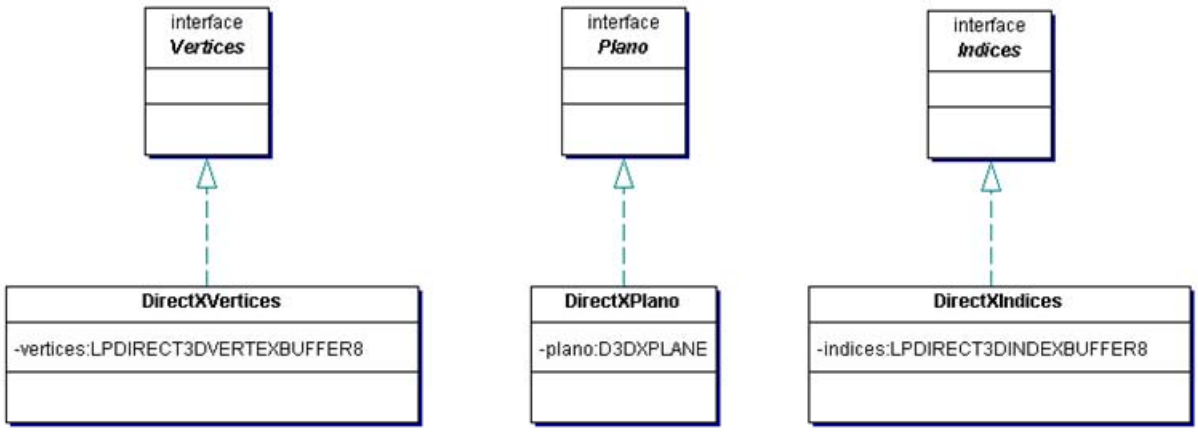


Figura 3.4.1

En la figura 3.4.1 podemos ver un fragmento del Paquete3D. La idea de este paquete es encapsular las clases de DirectX, de modo que para el resto del programa sea totalmente transparente la librería gráfica utilizada y que de manera simple, si interesara se pudiera cambiar a OpenGL tan solo cambiando la implementación de este paquete.

Una vez decidido que íbamos a crear una interfaz con DirectX hemos decidido poner nombres sencillos a las clases en español de manera que sean fácilmente recordables.

Tendremos otro paquete que va a ser el *paquete BSP* y que va a contener todo lo necesario para generar los árboles BSP que nos permitirán detectar las colisiones.

En este diseño se puede observar que hemos aplicado patrones. El patrón *composite* nos va a permitir tratar los nodos y las hojas de la misma manera, de modo que el código de la clase escenario sea más sencillo, evitando las distinciones de tipo.

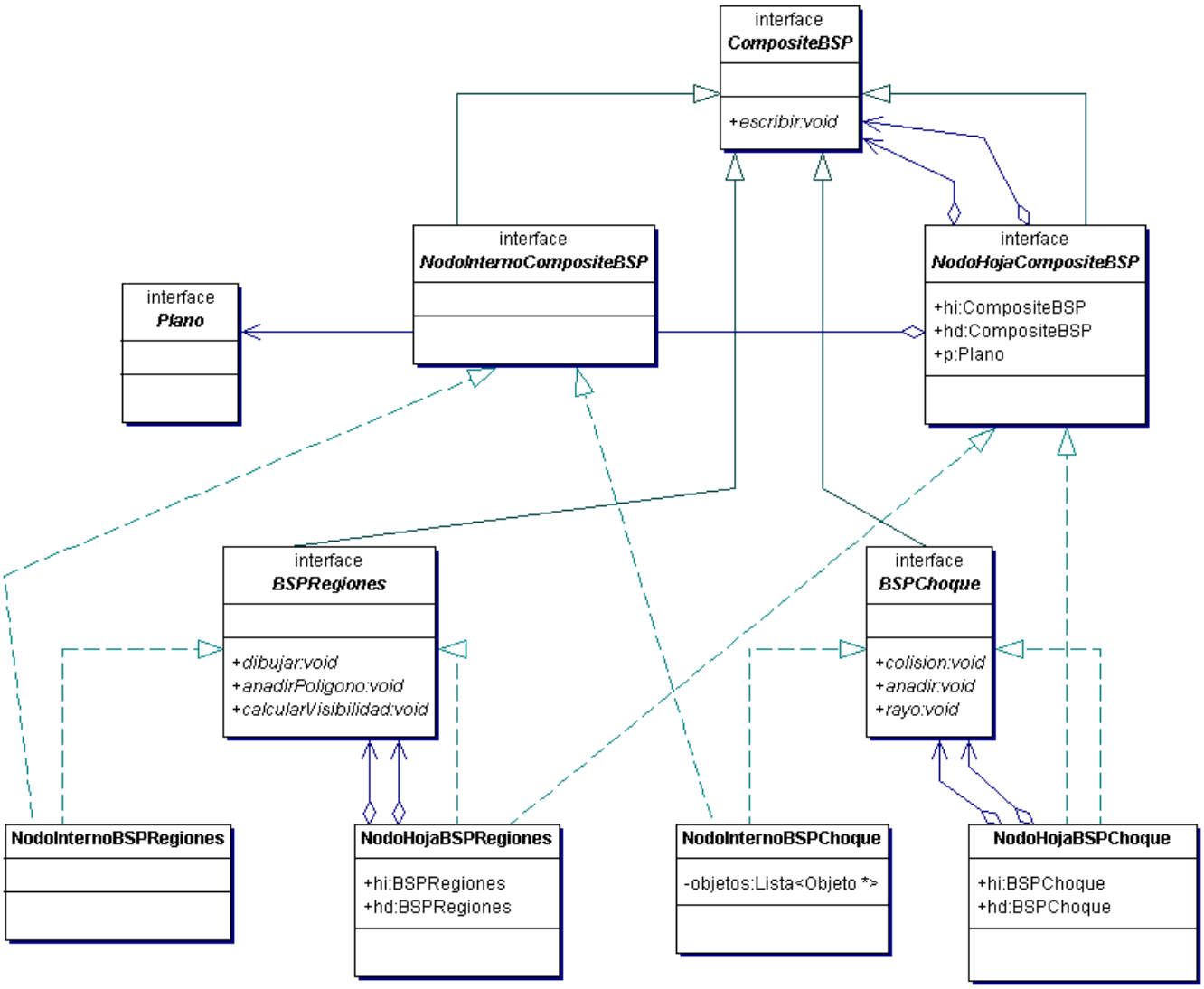


Figura 3.4.2

También tenemos el paquete *Listavisibilidad*, que va a contener todo lo referente a las regiones, es decir, se encargará de dividir el escenario en regiones de modo que, a la hora de dibujar el escenario, no sea necesario dibujarlo entero, sino que sea posible reducir al máximo el numero de polígonos que se le mandan dibujar a DirectX.

En la figura 3.4.2 también podemos observar una parte de lo que va a ser el paquete *ListaVisibilidad*, ya que en un principio creemos que podremos utilizar la estructura de árbol de los BSP para implementar las listas de visibilidad.

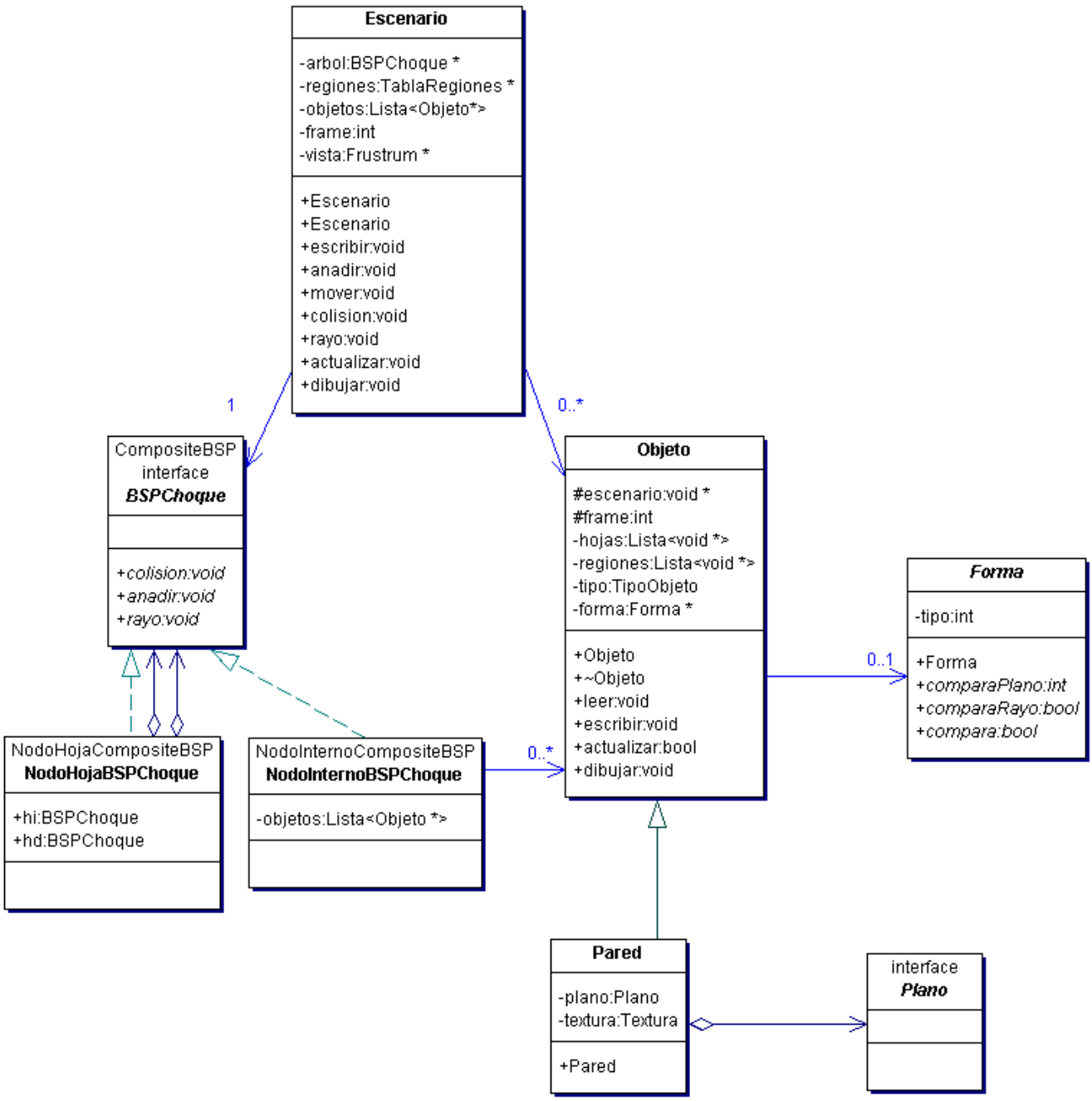


Figura 3.4.3

En la figura 3.4.3 se pretende mostrar como el escenario llamará a los árboles BSP para comprobar las colisiones entre los objetos.

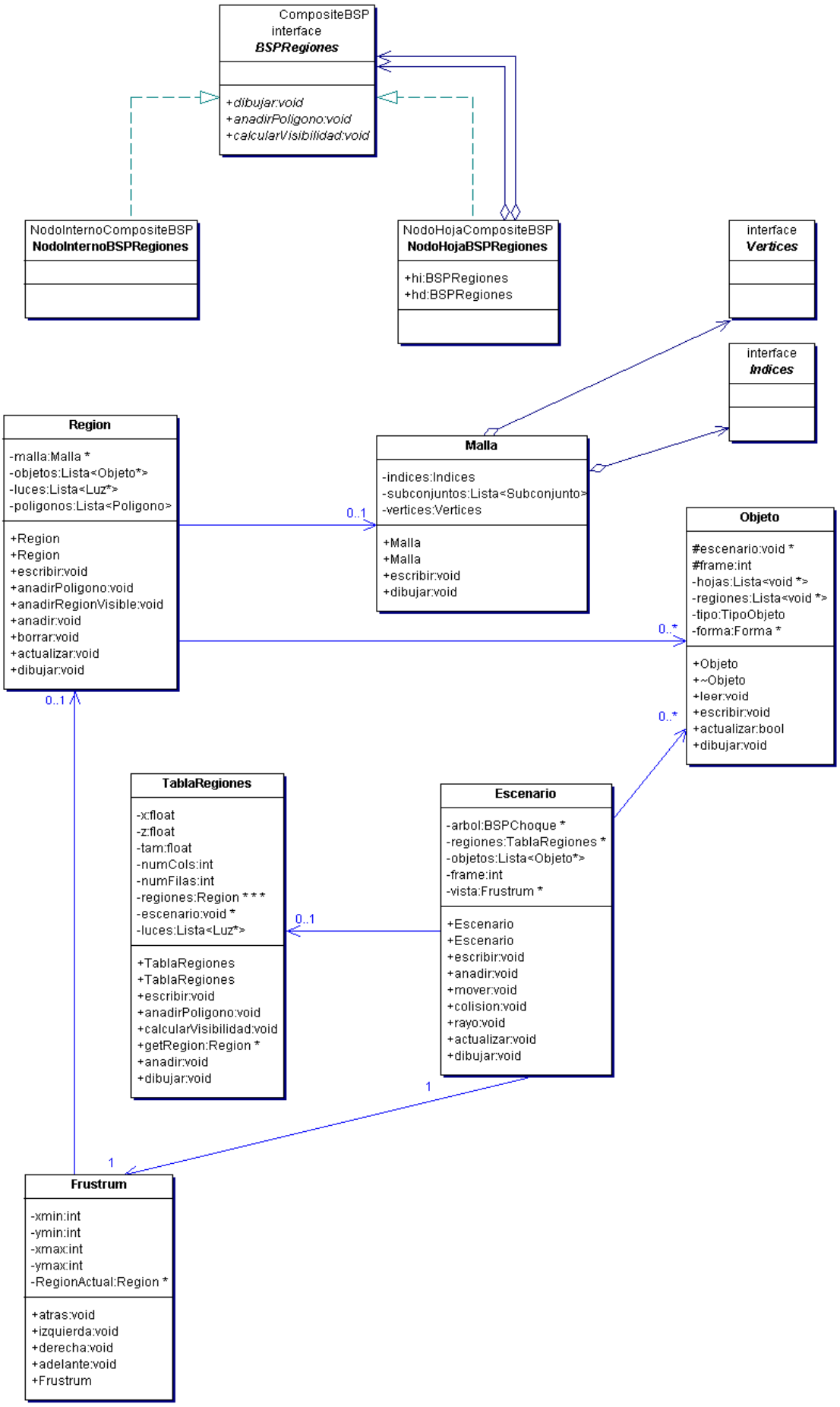
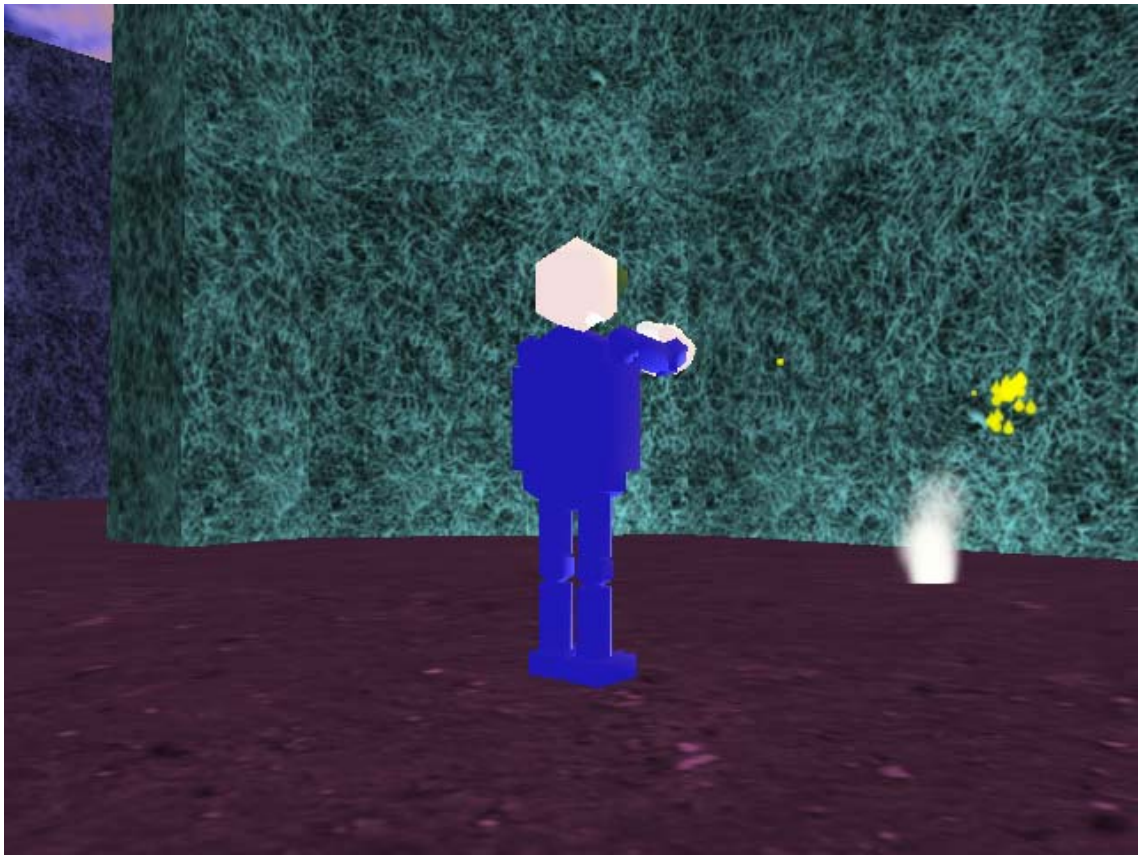

CompositeBSP
interface
BSPRegiones
+dibujar:void
+anadirPoligono:void
+calcularVisibilidad:void
NodoInternoCompositeBSP
NodoInternoBSPRegiones
NodoHojaCompositeBSP
NodoHojaBSPRegiones
+hi:BSPRegiones
+hd:BSPRegiones
interface
Vertices
interface
Indices
Region
-malla:Malla *
-objetos:Lista<Objeto*>
-luces:Lista<Luz*>
-poligonos:Lista<Poligono>
+Region
+Region
+escribir:void
+anadirPoligono:void
+anadirRegionVisible:void
+anadir:void
+borrar:void
+actualizar:void
+dibujar:void
Malla
-indices:Indices
-subconjuntos:Lista<Subconjunto>
-vertices:Vertices
+Malla
+Malla
+escribir:void
+dibujar:void
Objeto
#escenario:void *
#frame:int
-hojas:Lista<void *>
-regiones:Lista<void *>
-tipo:TipoObjeto
-forma:Forma *
+Objeto
+~Objeto
+leer:void
+escribir:void
+actualizar:bool
+dibujar:void
0..1
0..*
0..1
0..*
TablaRegiones
-x:float
-z:float
-tam:float
-numCols:int
-numFilas:int
-regiones:Region * * *
-escenario:void *
-luces:Lista<Luz*>
+TablaRegiones
+TablaRegiones
+escribir:void
+anadirPoligono:void
+calcularVisibilidad:void
+getRegion:Region *
+anadir:void
+dibujar:void
Escenario
-arbol:BSPChoque *
-regiones:TablaRegiones *
-objetos:Lista<Objeto*>
-frame:int
-vista:Frustrum *
+Escenario
+Escenario
+escribir:void
+anadir:void
+mover:void
+colision:void
+rayo:void
+actualizar:void
+dibujar:void
0..1
1
1
Frustrum
-xmin:int
-ymin:int
-xmax:int
-ymax:int
-RegionActual:Region *
+atras:void
+izquierda:void
+derecha:void
+adelante:void
+Frustrum

Figura 3.4.4

En la figura tenemos, por una parte, cómo va a interactuar el escenario con la tabla de regiones o lista de visibilidad y por otro lado, cómo va a estar implementada interiormente.

Y por último vamos a tener lo que llamaremos paquete *Juego* que va a contener las clases necesarias para implementar el juego haciendo uso de todo lo anterior. El paquete está compuesto por las clases principales de pantalla y las clases que heredaran de la clase objeto y que implementarán los objetos concretos de nuestra aplicación, es decir, el *Queco*, la *Bengala*, el *Disparo*, la *Particula*...



Captura gráfica del videojuego en la que se puede ver una bengala ardiendo en el suelo y las balas de pintura estrellándose en la pared.

3.5.- Diseño versión 4

En esta última etapa de diseño, que se solapa con la etapa de implementación, hemos realizado los cambios que hemos ido considerando convenientes y hemos evaluado como positivos mientras estamos en la etapa de implementación.

A continuación vamos a ver las clases principales de cada uno de los paquetes

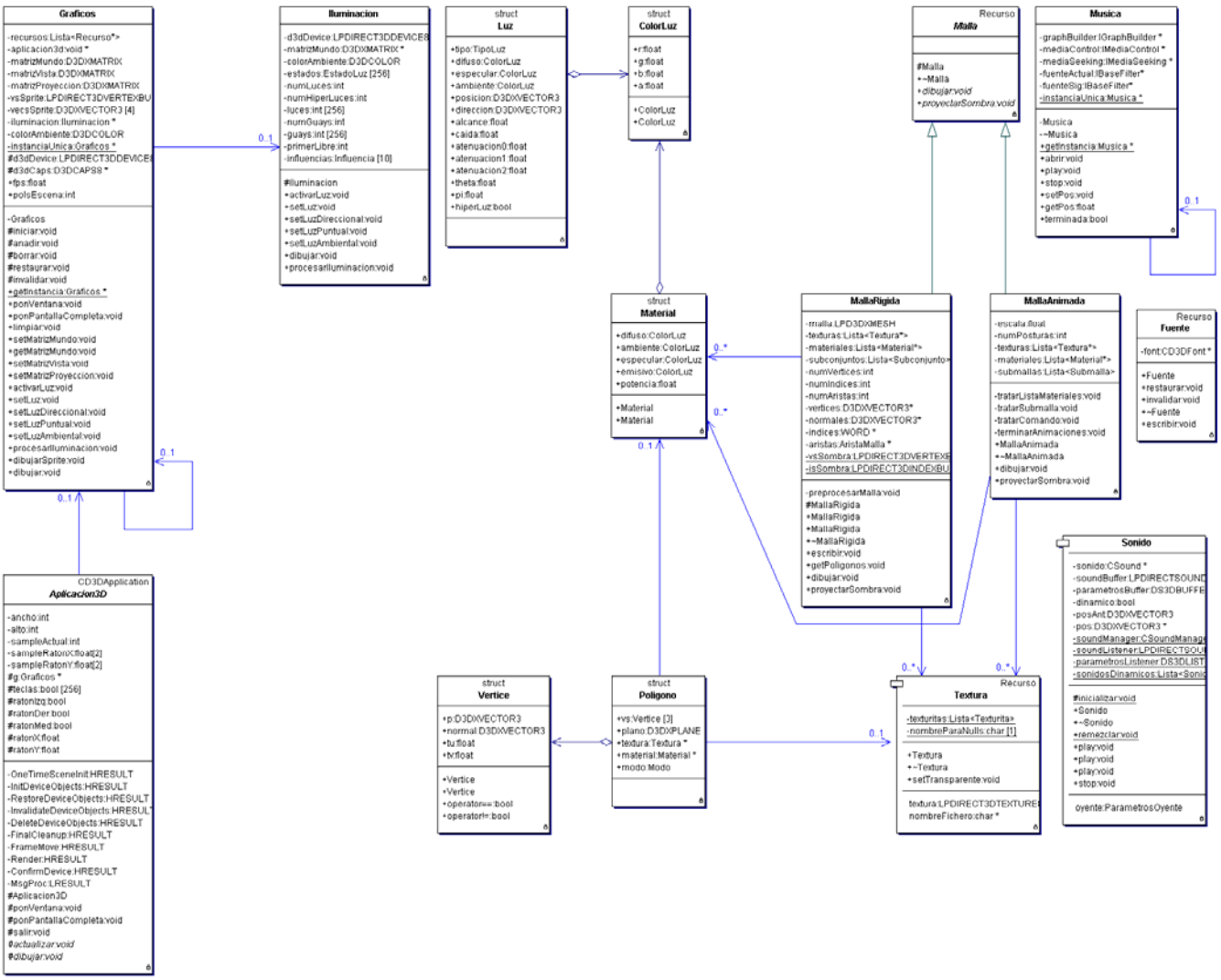


Figura 3.5.1

En la figura 3.5.1podemos ver como el *Paquete3D* que en un principio sólo iba a ser un envoltorio para DirectX, ahora ha cobrado mayor importancia, pues también maneja la iluminación y la música, además de contener un conversor de las mallas animadas que genera el 3D studio.

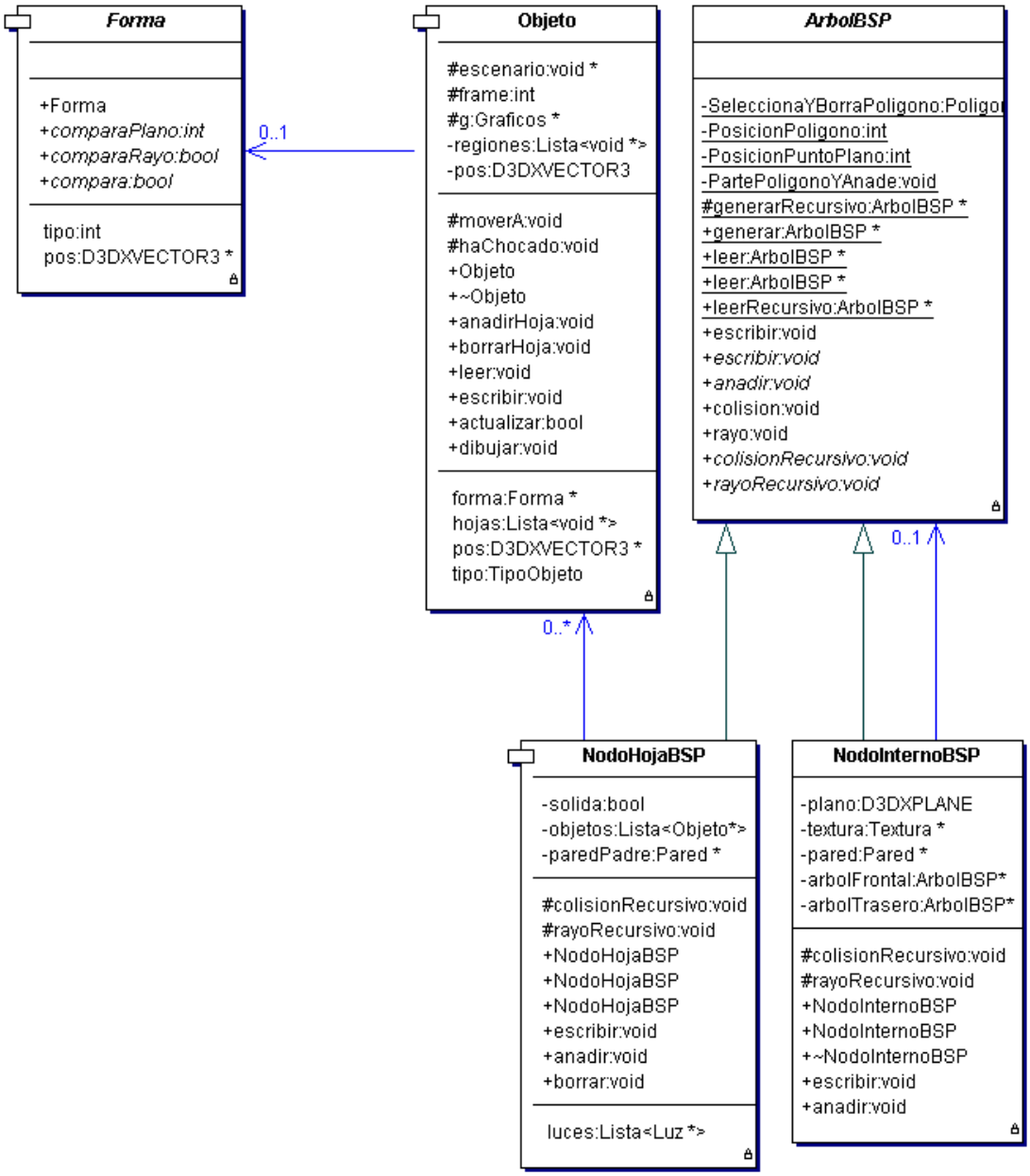


Figura 3.5.2

El Paquete de *árbol BSP* es el que menos ha cambiado, tan solo ha sido necesario crear nuevas subclases de *Forma* que se adecuasen a nuestras necesidades.

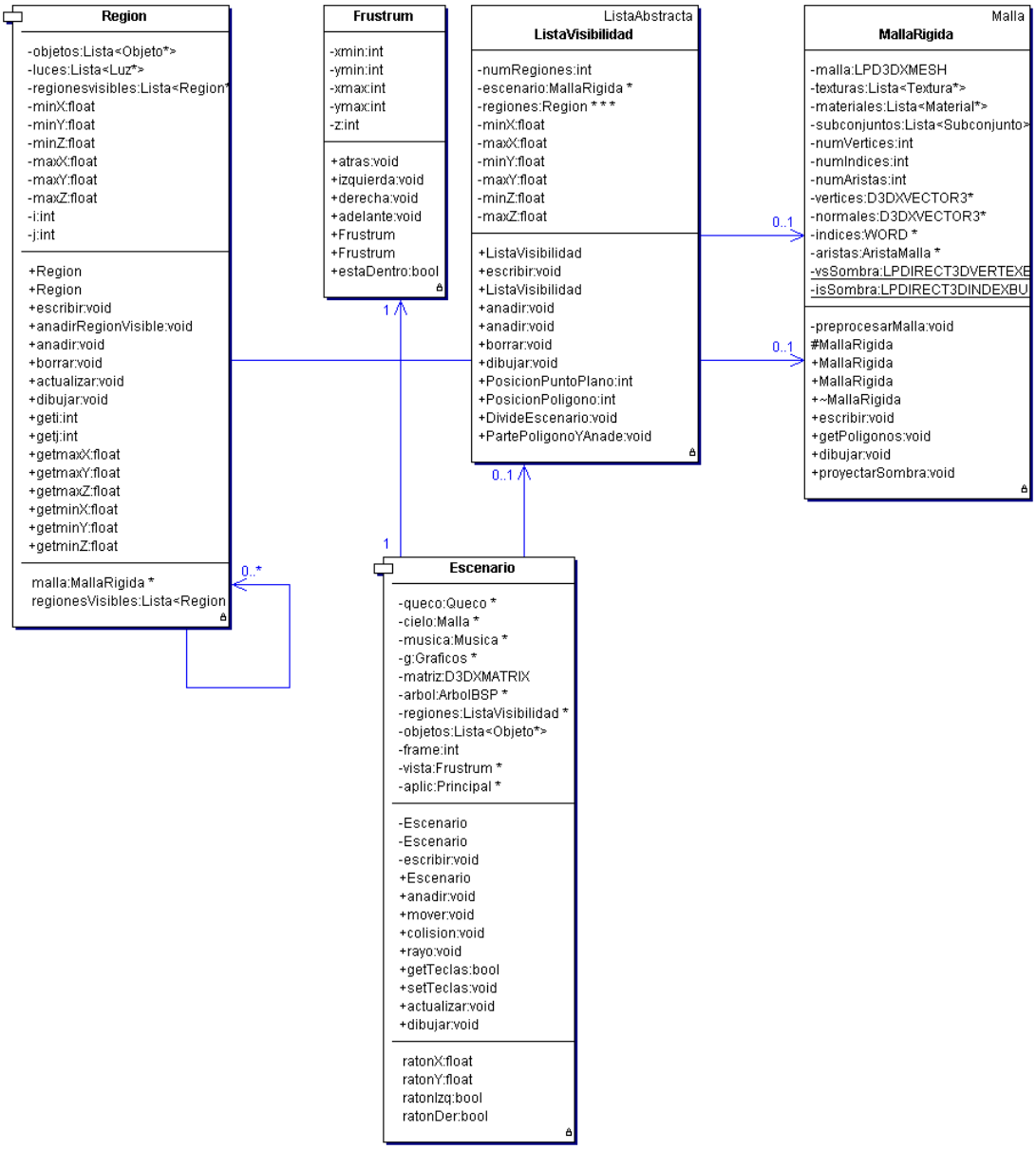


Figura 3.5.3

En el caso del paquete *ListaVisibilidad* los cambios en las interfaces han sido mínimos, sin embargo ha sufrido constantes cambios en la estructura interna, ya que hemos pasado por muchas ideas para su implementación. Al principio pensamos que podríamos usar árboles para su implementación, luego consideramos el uso de estructuras de datos más complejas, pero finalmente optamos por una estructura más sencilla consistente en un tabla bidimensional de listas enlazadas.

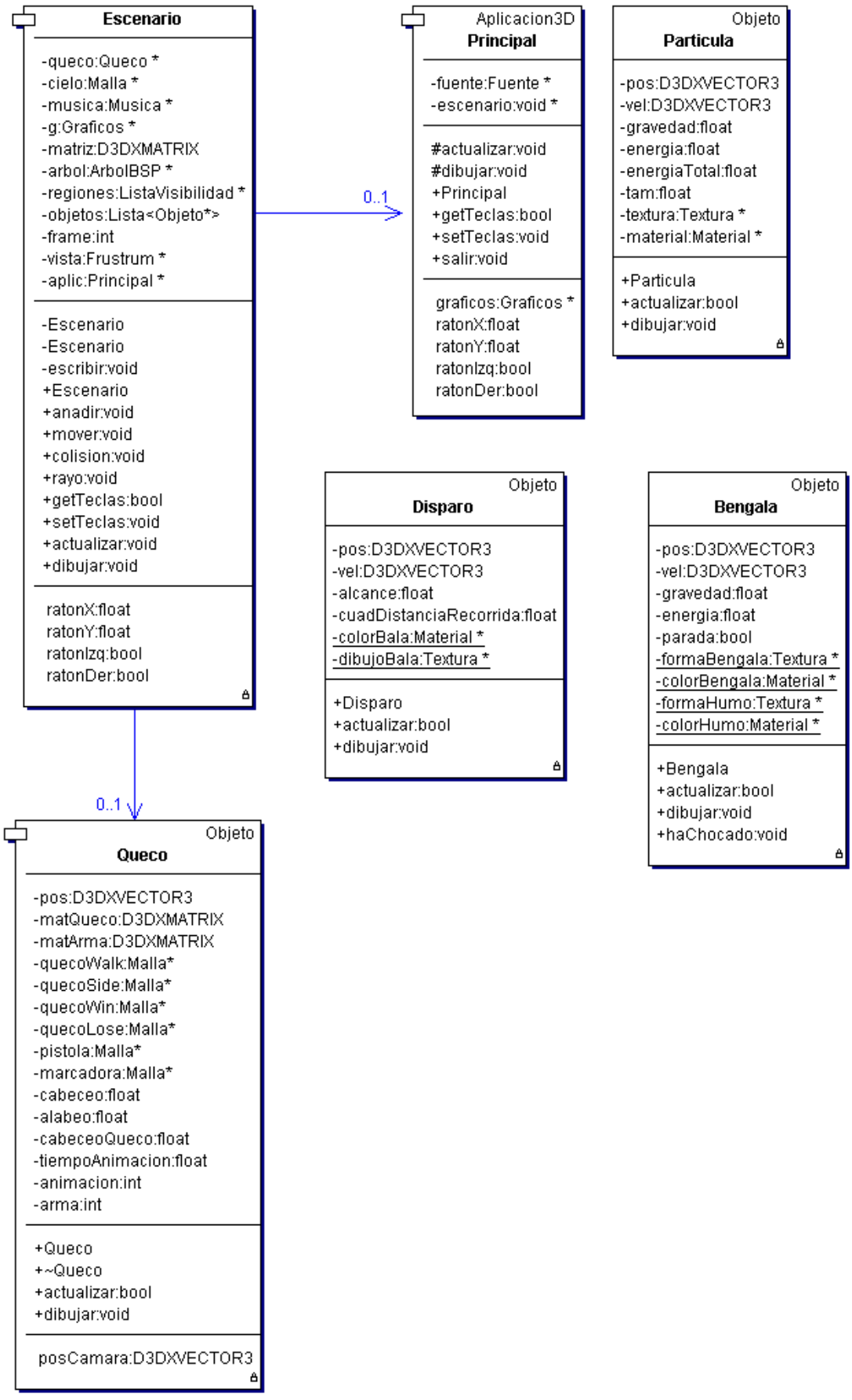

Escenario
-queco:Queco *
-cielo:Malla *
-musica:Musica *
-g:Graficos *
-matriz:D3DXMATRIX
-arbol:ArbolBSP *
-regiones:ListaVisibilidad *
-objetos:Lista<Objeto*>
-frame:int
-vista:Frustrum *
-aplic:Principal *
-Escenario
-Escenario
-escribir:void
+Escenario
+anadir:void
+mover:void
+colision:void
+rayo:void
+getTeclas:bool
+setTeclas:void
+actualizar:void
+dibujar:void
ratonX:float
ratonY:float
ratonIzq:bool
ratonDer:bool
0..1
Aplicacion3D
Principal
-fuente:Fuente *
-escenario:void *
#actualizar:void
#dibujar:void
+Principal
+getTeclas:bool
+setTeclas:void
+salir:void
graficos:Graficos *
ratonX:float
ratonY:float
ratonIzq:bool
ratonDer:bool
Objeto
Particula
-pos:D3DXVECTOR3
-vel:D3DXVECTOR3
-gravedad:float
-energia:float
-energiaTotal:float
-tam:float
-textura:Textura *
-material:Material *
+Particula
+actualizar:bool
+dibujar:void
Objeto
Disparo
-pos:D3DXVECTOR3
-vel:D3DXVECTOR3
-alcance:float
-cuadDistanciaRecorrida:float
-colorBala:Material *
-dibujoBala:Textura *
+Disparo
+actualizar:bool
+dibujar:void
Objeto
Bengala
-pos:D3DXVECTOR3
-vel:D3DXVECTOR3
-gravedad:float
-energia:float
-parada:bool
-formaBengala:Textura *
-colorBengala:Material *
-formaHumo:Textura *
-colorHumo:Material *
+Bengala
+actualizar:bool
+dibujar:void
+haChocado:void
0..1
Objeto
Queco
-pos:D3DXVECTOR3
-matQueco:D3DXMATRIX
-matArma:D3DXMATRIX
-quecoWalk:Malla*
-quecoSide:Malla*
-quecoWin:Malla*
-quecoLose:Malla*
-pistola:Malla*
-marcadora:Malla*
-cabeceo:float
-alabeo:float
-cabeceoQueco:float
-tiempoAnimacion:float
-animacion:int
-arma:int
+Queco
+~Queco
+actualizar:bool
+dibujar:void
posCamara:D3DXVECTOR3

Figura 3.5.4

El paquete Juego, que se ha diseñado casi por completo en esta última fase es bastante sencillo en cuanto a su estructura, pues tal como originalmente estaba pensado, basta con tener una implementación de la clase *Aplicación3D* en la que se declaren los objetos que van a formar parte del juego. La complejidad de estas clases reside en la funcionalidad que se le quiera otorgar a cada uno de los objetos que formen parte de ella.

Captura gráfica del videojuego

4.- Implementación

Esta fase del desarrollo del proyecto, que se ha solapado en un primer momento con la última fase del diseño, es la que mayor parte de tiempo nos ha llevado. A continuación se muestra una breve descripción de cuál es el comportamiento y algunas ideas sobre la implementación de cada uno de los paquetes que integran la aplicación.

Captura gráfica del videojuego

4.1.- El preprocesamiento

Para poder comprender algunos aspectos importantes de la implementación de los paquetes, es importante tener en cuenta un detalle: el preprocesamiento. Cuando en un inicio nos planteábamos que la aplicación debía ser eficiente y ejecutarse rápido nos dimos cuenta que había muchos cálculos, necesarios para preparar un escenario de manera que se pueda visualizar, que no era necesario recalcular en cada ejecución de nuestro programa. Con esta idea surge el preprocesamiento, que consiste en: una vez generadas las estructuras de los árboles BSP y las Listas de Visibilidad, éstas pueden ser almacenadas en ficheros, de manera que, la próxima vez que se inicie el juego no sea necesario volver a calcularlas.

Esto se ha traducido finalmente en que tanto el árbol BSP como las listas de Visibilidad tengan métodos para escribirse y leerse de un fichero.

La influencia en nuestra aplicación del preprocesamiento ha sido mucho mayor de lo que en un principio esperábamos, pues, para el escenario que hemos diseñado, el tiempo necesario para generar las estructuras correctamente es de aproximadamente una hora en uno de nuestros ordenadores, mientras que las siguientes ejecuciones que leen las estructuras de un fichero, apenas tardan unos diez segundos.

4.2.- Paquete 3D

Introducción

El *Paquete3D* es una parte del motor para juegos que hemos desarrollado. Su función es hacer a la aplicación independiente del entorno gráfico en que se ejecute (DirectX, OpenGL,...), de modo que los programas puedan portarse reimplementando el *Paquete3D* sin tocar el resto de la aplicación.

Todas las funciones principales de cualquier entorno gráfico tienen su equivalente en el *Paquete3D*. En los próximos apartados iremos viendo una a una las diferentes características soportadas por este conjunto de clases, que incluyen la carga de mallas y animaciones de ficheros, texturas, materiales, textos, iluminación con proyección de sombras volumétricas en tiempo real, reproducción de sonido en 3D y música en mp3.

Actualmente el *Paquete3D* está implementado para DirectX. Escogimos esta librería gráfica por varias razones. La principal era que, al contrario que OpenGL, incluye "de serie" funciones para cargar mallas de ficheros, y conversores de uno de los tipos de ficheros que se pueden generar con la herramienta de diseño utilizada, 3D Studio Max, al tipo de los ficheros soportados por aquellas funciones. Más tarde veremos que esta característica no nos fue tan útil como pretendíamos.

Otra de las razones por las que escogimos DirectX es porque también incorpora abundante documentación, así como muchos programas de ejemplo en los que puede aprenderse el funcionamiento de cada una de sus características.

La filosofía de diseño del *Paquete3D* es, además de ofrecer independencia del entorno gráfico, conseguir la máxima facilidad de uso posible. Iremos viendo en cada uno de los apartados como en general se proporciona la posibilidad de hacer la mayoría de las cosas con simples llamadas a métodos con pocos parámetros, donde en DirectX u OpenGL se obliga a rellenar estructuras de datos con muchos parámetros, en muchas ocasiones con valores estándar. La gestión automática de la liberación y restauración de recursos cuando la aplicación en pantalla completa se minimiza y se restaura, el tratamiento automatizado de la proyección de sombras o de los sonidos dinámicos en 3D, son otros ejemplos del intento de facilitar al máximo posible la vida al cliente.

Por todo ello, el resultado principal del *Paquete3D* no son las características que pone en funcionamiento, sino sobre todo la pequeñez y la simplicidad del código. Esperamos que el código del juego en sí sea una muestra de esto y de la cantidad de problemas que

el uso de este conjunto de clases ahorra a los clientes en la creación de juegos, problemas que pasamos a ver uno a uno.

Estructura

Para crear una aplicación que use gráficos y sonido mediante el *Paquete3D,* lo primero que hay que hacer es crear una clase que herede de la clase *Aplicacion3D*, sobrescribiendo sus métodos *actualizar* y *dibujar*. A estos métodos se los llamará una y otra vez hasta que se use el método salir.

```
virtual void actualizar(float tiempo) = 0;
virtual void dibujar() = 0;
```

El método actualizar debe actualizar la aplicación en cada fotograma, incluyendo todo el mundo del juego, los personajes, etc.; el método dibujar debe dibujarlos, usando todos los recursos proporcionados por el *Paquete3D*. Se podría haber fundido ambos métodos en uno solo, pero así se mantiene la filosofía de las aplicaciones 3D, en que ambas funciones deben estar diferenciadas porque no tienen por qué dibujarse todos los objetos (se deben excluir aquellos no visibles desde el punto de vista de la cámara), mientras que sí deben actualizarse todos los objetos en todos los fotogramas (no queremos que, por ejemplo, los proyectiles dejen de avanzar cuando no les miramos).

Las aplicaciones gráficas deben tener en cuenta que los gráficos tardan en renderizarse una cantidad de tiempo diferente en cada fotograma (debido a la distinta cantidad de objetos a dibujar, la superficie que ocupan estos en la pantalla, etc.), y diferente en cada ordenador (debido a la diferencia de potencia). Por esta razón, en el método actualizar se recibe como parámetro la cantidad de tiempo transcurrido desde el fotograma anterior, de modo que las aplicaciones hagan "avanzar" el mundo una cantidad de tiempo proporcional al tiempo que se está tardando en generar cada imagen. Así, el juego se ejecutará siempre a la misma velocidad, aunque el número de imágenes por segundo pueda variar, dando lugar a que los gráficos se muevan con más o menos fluidez.

El funcionamiento de la clase *Aplicacion3D* está basado en el de una clase que viene de DirectX que se llama *CD3DApplication*. Esta clase contiene una gran cantidad de código que se encarga de realizar tareas bastante tediosas que deben hacerse en todos los programas que usen DirectX. Entre otras cosas, hace una lista de todas las tarjetas gráficas (generalmente una) disponibles en el sistema; para cada una de ellas hace una lista de "dispositivos" disponibles para renderizar (generalmente dos: uno renderiza por hardware y otro, usado sólo en casos especiales, por software); y para cada uno de ellos hace una lista de todos los modos gráficos disponibles (resoluciones, profundidades de color, etc.). La cantidad de código que se necesita

para todo ello es enorme y por eso la clase *Aplicacion3D* se limita a reusarlo, heredando de ella.

Normalmente, las aplicaciones que usan DirectX pueden crearse heredando directamente de *CD3DApplication*, y sobrescribiendo nada menos que 10 métodos:

```
HRESULT OneTimeSceneInit();
HRESULT InitDeviceObjects();
HRESULT RestoreDeviceObjects();
HRESULT InvalidateDeviceObjects();
HRESULT DeleteDeviceObjects();
HRESULT FinalCleanup();
HRESULT FrameMove();
HRESULT Render();
HRESULT ConfirmDevice(D3DCAPS8*,DWORD,D3DFORMAT);
LRESULT MsgProc(HWND hWnd,UINT uMsg,WPARAM wParam,LPARAM lParam);
```

La clase del *Paquete3D*, *Aplicacion3D*, siguiendo la filosofía de diseño del paquete, permite una facilidad de uso mucho mayor exigiendo al cliente solamente sobrescribir los dos métodos citados. Esos 10 métodos han sido heredados de forma privada y sobrescritos con contenido útil de modo que el cliente no tiene ni que saber que existen. Hagamos un recorrido, uno a uno, de todos esos métodos para ver cómo se ha conseguido simplificar enormemente la interfaz con la que el cliente debe trabajar:

- Los métodos *FrameMove* y *Render* son los equivalentes a actualizar y dibujar, respectivamente. Como puede intuirse, la implementación de dichos métodos en *Aplicacion3D* la constituyen las llamadas a los dos métodos nuevos.

- Los métodos *OneTimeSceneInit*, *InitDeviceObjects, DeleteDeviceObjects* y *FinalCleanup* dan la oportunidad al cliente de crear todos los recursos (texturas, sonidos, etc.) que el programa va a necesitar al principio de la aplicación, y de borrarlos al final. Esos son los momentos en que son llamados esos métodos. Sin embargo, esta estructura no se corresponde con la naturaleza habitual de los juegos. Los juegos no cargan todos los recursos que necesitan al principio de su ejecución y los borran al final, porque suelen requerir borrar y cargar diferentes recursos a lo largo de su ejecución (entre fase y fase, o incluso durante el propio juego), debido a la gran cantidad de memoria que ocupan. Por esa razón, es necesario de todos modos cargar y borrar recursos en los momentos adecuados de la actualización de la aplicación, así que se hace más ortogonal la programación de la misma cargando y borrando todos los recursos en la actualización. Esos métodos tienen su sentido en los programas de ejemplo que vienen con DirectX, que son programas muy sencillos en los que es posible ese comportamiento.

- *ConfirmDevice* es un método al que se llama con un puntero a una estructura que contiene las características disponibles de la tarjeta gráfica, y los programas deben devolver si son suficientes o no para ejecutar ese juego en particular. Pues bien, dado que el Paquete3D intenta ocultar todos los detalles de la implementación de los gráficos, no sería adecuado obligar al cliente a conocerlos para rellenar este método. Por ejemplo, para usar proyección de sombras son necesarias ciertas características que el cliente tendría que especificar, pero que puede que sean distintas a las características que se usarían al reimplementar el Paquete3D en otro entorno gráfico, y eso haría que tuvieran que modificarse las aplicaciones al cambiar el entorno gráfico, alejándonos del objetivo que el Paquete3D persigue. Por esta razón, es el Paquete3D el que se encarga de responder a la llamada conociendo las características que usa.

- El método *MsgProc* es llamado cuando la aplicación recibe mensajes de Windows causados por eventos (teclado, ratón,...). Se ofrece un servicio mejor a los clientes del Paquete3D leyendo por ellos esos eventos en variables. Hay un array de booleanos que contiene el estado de pulsación o no de cada una de las teclas del teclado, 3 booleanos más para los 3 botones del ratón y un par de coordenadas que indican el movimiento del ratón. Esto es más apropiado porque así la programación de las respuestas al control de un personaje puede hacerse junto con el resto del código de actualización de ese personaje, consultando las variables, y no en un método aparte de gestión de eventos, que dispersaría código muy relacionado. Además, así ya está solucionado el problema no trivial de medir el movimiento del ratón en cada fotograma (hay que medir la distancia del puntero al centro de la pantalla y luego volverlo a poner en el centro de la pantalla).

- Por último, los métodos *RestoreDeviceObjects* e *InvalidateDeviceObjects* son una de las cargas más tediosas de llevar al hacer programas normales en DirectX, de las que el Paquete3D libra al cliente completamente, tal y como explicaré a continuación.

El Paquete3D ofrece una solución muy elegante al problema de tener que implementar los métodos *RestoreDeviceObjects* e *InvalidateDeviceObjects* en las aplicaciones de DirectX. Al método *InvalidateDeviceObjects* se le llama cuando el usuario pulsa Alt+Tab estando la aplicación funcionando en pantalla completa, y se vuelve a Windows. Al método *RestoreDeviceObjects* se le llama cuando se

restaura la aplicación después de la situación anterior. Las aplicaciones normalmente deben encargarse de liberar ciertos recursos y volver a reservarlos (ciertas clases de texturas y de geometría) porque dejan de tener "autoridad" para tener reservados para ellos toda la memoria de vídeo, etc.

En el Paquete3D hay una clase Recurso de la que heredan todos los recursos así. Cuando el cliente crea uno de estos recursos, por ejemplo, un tipo de letra para escribir, el constructor heredado de Recurso se encarga de añadir el recurso creado a una lista oculta al cliente. Cuando el cliente destruye un recurso, el destructor heredado lo borra de la lista. Así, cuando el programa debe "invalidar" y "restaurar" los recursos, esos dos métodos internamente recorren la lista y mandan liberar y restaurar los recursos, respectivamente.

Gracias a eso el uso de los recursos por parte del cliente es mucho más sencillo e inmune a fallos. La lista de recursos se encuentra en la instancia única de la clase *Graficos*, a la que la clase *Aplicacion3D* llama desde esos dos métodos.

La clase *Graficos* es un singleton, esto es, una clase de la que sólo puede existir una instancia, que está accesible para todo el mundo. Esta clase es la que se encarga de ofrecer la mayoría de los servicios que el cliente necesita para renderizar la escena en el método dibujar. Para conseguir que sea un singleton basta con declarar privados el constructor y una variable de clase puntero a la instancia única, y declarar público un método que devuelve el puntero a dicha instancia (creándola si es la primera vez que se le llama).

He aquí un extracto de los métodos principales de Graficos:

```
    // MÉTODOS PARA CAMBIAR EL MODO GRÁFICO
    void ponVentana();
    void ponPantallaCompleta(int ancho,int alto,bool colorVerdadero);
    inline void limpiar(bool limpiarColor,Color color,
                        bool limpiarProfundidad,float profundidad=1.0f)

    // MÉTODOS PARA CAMBIAR LAS MATRICES DE TRANSFORMACIÓN
    inline void setMatrizMundo(Matriz *matriz);
    inline void getMatrizMundo(Matriz *matriz);
    void setMatrizVista(Matriz *matriz);
    inline void setMatrizProyeccion(Matriz *matriz);

    // MÉTODOS PARA CONTROLAR LA ILUMINACIÓN
    inline void activarLuz(int num,bool activar=true);
    inline void setLuz(int num,Luz *luz);
    inline void setLuzDireccional(int num,Vector direccion,ColorLuz color);
    inline void setLuzPuntual(int num,Vector posicion,ColorLuz color,float
alcance,bool hiperLuz=false);
    inline void procesarIluminacion();

    // MÉTODOS PARA DIBUJAR COSAS
    void dibujarSprite(Vector pos,float tam,Textura *tex,Material *mat,float
opacidad=1.0f);
    inline void dibujar(Malla *malla,Modificador flags=0);
```

Veamos unos cuantos detalles sobre los métodos disponibles:

- Los métodos para cambiar el modo gráfico, que permiten pantalla completa a diferentes resoluciones o ventana, se implementan en realidad invocando a la clase Aplicacion3D, la cual hereda de la clase en la que se encuentra toda la información necesaria para hacer este trabajo, como se dijo antes.

- De los métodos para cambiar las matrices de transformación puede que llame la atención a las personas acostumbradas a trabajar con OpenGL el hecho de que haya 3 matrices distintas. La matriz de modelado y vista de OpenGL en DirectX viene desglosada en dos: la matriz de mundo (que se encarga de trasladar las mallas que se dibujen a la posición en que se encuentran los objetos a los que representan), y la matriz de vista (que indica la posición y orientación de la cámara). En una hipotética implementación futura del Paquete3D a OpenGL esto no tendría ningún problema: bastaría con establecer como matriz de modelado y vista el producto de esas dos matrices.

- Los métodos usados para configurar la iluminación se tratarán extensamente en un apartado posterior de esta memoria.

- Existe un método para dibujar sprites fácilmente con una sola línea de código. Un sprite es un cuadrilátero orientado hacia la cámara, usado para dibujar partículas, nubes de humo y similares. El Paquete3D se encarga de apuntar la posición de la cámara cada vez que se cambie la matriz de mundo, para así poder calcular las coordenadas de las cuatro esquinas del sprite en función de la posición del sprite y de la cámara. Gracias a esto pueden incluirse en los juegos efectos visuales a base de sistemas de partículas o similares con facilidad. El último parámetro (opacidad) de la llamada, permite hacer que los sprites desaparezcan o aparezcan gradualmente.

- Muchos de los métodos son inline porque se implementan con una línea de código equivalente para DirectX, y así es más eficiente su uso.

Por último, queda por comentar en esta sección algunos detalles más sobre el Paquete3D:

- La clase *Fuente* permite escribir texto en la pantalla. Encapsula a la clase *CD3DFont* que viene con DirectX,

añadiéndole además la funcionalidad de la gestión automática de minimizar y restaurar la aplicación, como se comentó anteriormente.

- Los tipos de datos usados para las matrices, los vectores y los planos están declarados en Paquete3D.h, donde se identifican con los usados en DirectX para no necesitar de ninguna transformación en tiempo real, y donde vienen declaradas las principales funciones que permiten operar con ellos:

    - los vectores pueden normalizarse, interpolarse, transformarse con matrices, reflejarse o proyectarse con planos, calcular su módulo o el cuadrado de éste y multiplicarse escalar o vectorialmente;

    - los planos pueden normalizarse, transformarse, calcularse a base de un punto y una normal o de 3 puntos, intersecarse con líneas y calcular la distancia con puntos;

    - de las matrices pueden hallarse productos, inversas, traspuestas, matrices adecuadas para la matriz de mundo (traslación, escalado, varias clases de rotaciones, reflejo...), para la matriz de vista (indicando posición, dirección y vector arriba) y para la matriz de proyección (se incluyen varias clases de proyecciones ortogonales y en perspectiva).

  Algunas de esas operaciones invocan a funciones contenidas en la librería de DirectX; otras no. Al usar los tipos y operaciones declaradas se elimina totalmente la necesidad de hacer la menor referencia a DirectX en el código de la aplicación, con lo que no tendría que alterarse en lo más mínimo al trasladarla a otro entorno gráfico. Lo peor que podría pasar es tener que hacer traducciones internas en esas operaciones porque los tipos de datos usados actualmente por el Paquete3D (los de DirectX) no se correspondan con los del nuevo entorno gráfico, perjudicando la eficiencia de la aplicación; pero esto no es muy probable porque todos los tipos de datos de los entornos gráficos están declarados de formas similares y respetando los tipos usados en C++, y además aunque fueran distintos, la compatibilidad sin necesidad de traducciones se mantendría si siempre se usaran las operaciones proporcionadas para acceder a los datos y no a los campos de los datos directamente (componentes x, y y z de un vector, por ejemplo), ya que aquéllas siempre respetarían el nuevo orden de las componentes, etc. Más

allá de eso, el problema de la necesidad de traducciones sería inevitable.

- Una función secundaria pero presente en la clase *Aplicacion3D* es suavizar el movimiento del ratón. Sucede que si un juego en el que la cámara se controla por el ratón, se está ejecutando en un ordenador en que los gráficos van a 60 imágenes por segundo, y los eventos del ratón a 40 Hz, la sensación visual que se tiene es que los gráficos van a 40 imágenes por segundo, porque periódicamente hay pares de fotogramas consecutivos en que la cámara no se mueve. La solución es usar como entrada para el movimiento de la cámara la media del movimiento del ratón en los últimos fotogramas. De este modo, el movimiento se reparte entre fotogramas consecutivos, es decir, se suaviza. La mejora en la fluidez de movimientos, comparando ambas versiones, se nota claramente.

Mallas y animación

En este apartado veremos todos los problemas existentes en la carga de ficheros con mallas de polígonos y animación, generados por una herramienta de diseño (3D Studio Max), en un programa que no sólo debe mostrarlos en tiempo real, sino también acceder a su contenido para realizar modificaciones o ejecutar los diferentes algoritmos necesarios; particularmente el procesamiento del escenario para generar las estructuras de datos necesarias para la detección de colisiones y la visibilidad.

El motivo fundamental por el que escogimos la librería de DirectX para realizar la implementación del *Paquete3D* fue que ésta incorporaba directamente servicios de carga de mallas y de animaciones. Más tarde veremos que estas funciones se quedaron un poco cortas con respecto a nuestras necesidades.

DirectX incluye un formato de fichero, los ficheros .x, que pueden contener mallas o animaciones. También incluye una pequeña utilidad llamada conv3ds.exe para convertir los ficheros .3ds, que el 3D Studio Max puede generar, a ficheros .x. Por último, incluye también funciones para cargar tales mallas, al menos sin animación. Así, en un principio, el trabajo para tener a nuestras mallas en el programa, es exportarlas a formato .3ds con el 3D Studio Max, convertir los ficheros resultantes al formato .x, y usar las funciones incorporadas por DirectX para cargarlos.

La clase *MallaRigida* es la que encapsula esta última funcionalidad. Su constructor recibe el nombre del fichero a cargar y su método dibujar dibuja la malla, que aparecerá en la ubicación especificada por la matriz de mundo.

```
MallaRigida(char *nombreFichero);
MallaRigida(Lista<Poligono> &lp);
~MallaRigida();

void escribir(char *nombreFichero);
void getPoligonos(Lista<Poligono> *lp, Lista<Textura*> *lt=NULL,
                  Lista<Material*> *lm=NULL);
void dibujar(float tiempo);
void proyectarSombra(float tiempo,Vector posLuz);
```

Para realizar el procesamiento de las mallas cargadas se proporcionan los siguientes métodos: *getPoligonos* devuelve una lista de polígonos con todos los polígonos de esta malla. Si es necesario modificar una malla tras realizar cualquier procesamiento (por ej., trocearla para dibujar sólo los trozos adecuados en la visibilidad), se usa el otro constructor, que genera una malla a partir de una lista de polígonos; y el método *escribir*, que permite escribir en un fichero la malla procesada, permitiendo que los juegos no tengan que volver a hacerlo cada vez que se carguen.

El tipo *Poligono,* definido en *Poligono.h*, contiene 3 vértices (cada uno con sus coordenadas de posición y de textura y su normal), el plano determinado por dichos vértices, punteros a su textura y material, y su modo de dibujo. La textura la contiene un objeto de tipo *Textura*, que es una clase de sencilla interfaz que encapsula la funcionalidad de DirectX de cargar texturas de diferentes formatos; el material especifica el modo en que le afecta la luz, y el modo de dibujo es un entero que permite especificar diferentes maneras de dibujar el polígono. En la versión final de nuestro programa no ha llegado a usarse, pero su utilidad es que diferentes polígonos de una misma malla se dibujen con diferentes efectos. Por ejemplo, una nave espacial podría estar formada de partes opacas, partes trasparentes (la carlinga), partes con *environment mapping* o brillo metálico, etc.

Falta por comentar el método *proyectarSombra*, cuya función se explicará en el apartado de iluminación. Dicha función requiere un preprocesamiento especial de la malla que realizan internamente sus constructores.

Pero el verdadero problema de satisfacer nuestra especificación sobre la carga de ficheros generados por el 3D Studio Max nos lo dieron las animaciones.

En primer lugar, 3D Studio Max no es capaz de exportar animaciones a un fichero .3ds. En consecuencia no se pueden leer de los ficheros .x traducidos por nuestro conversor. La solución parecía dárnoslas los exportadores de 3D Studio Max. Resulta que esta herramienta de diseño ha sido creada de modo que los programadores pueden extenderla incorporándole los llamados *plug-ins*. En particular, existen plug-ins para exportar directamente desde 3D Studio Max a ficheros .x.

Las versiones más actualizadas del SDK de DirectX vienen con un plug-in así. Después de instalar una versión de 3D Studio Max posterior a la que teníamos, porque el plug-in no era compatible con ella, y de compilar el plug-in, que estaba proporcionado en forma de código fuente porque debía compilarse con librerías incorporadas por la versión exacta del 3D Studio Max para la que se quiera que funcione, descubrimos que dicho plug-in sólo exporta correctamente las mallas rígidas o carentes de animación. La animación de derrota de nuestro personaje aparecía totalmente desfigurada: los miembros se metían unos dentro de otros, y tenía 3 cuerpos, uno con brazos y piernas y otros dos que flotaban en el aire...

Dedicamos horas a buscar otros plug-ins por Internet. En esa búsqueda descubrimos la cantidad de gente que también tenía quejas sobre el plug-in incluido por Microsoft en su SDK, y pudimos probar diferentes correcciones del código fuente que la gente proponía, ninguna de las cuales nos ayudaron. Después de varias horas nos

convencimos de que en la red sólo podíamos encontrar un plug-in más para lo mismo, esta vez desarrollado por Pandasoft.

Desgraciadamente, el plug-in sólo nos enseñó otra manera de distorsionar incorrectamente la animación. Dedicamos más horas a probar ambos plug-ins con todas las combinaciones de parámetros que admiten, y a probarlos con diferentes versiones del SDK (8.0, 8.1 y 9.0) por si fueron desarrollados para una versión específica de las funciones de carga de mallas, pero no hubo manera. Y lo peor fue que todo esto era posterior al tiempo que le habíamos dedicado a investigar el funcionamiento de la carga de animaciones de ficheros .x, que como ya dijimos antes no es tan directa como la carga de mallas rígidas. Muchas más horas perdidas.

Estos problemas, junto con un adelantamiento de la fecha de entrega de este proyecto que no habíamos previsto, nos llevaron a la idea de abandonar nuestra intención de satisfacer completamente la especificación y de no incorporar animaciones en nuestro motor para juegos.

Pero no fue así. Con sólo unos días por delante, el que escribe se armó de valor y se dedicó a fondo a desarrollar una clase que cargue manualmente las animaciones de uno de los formatos generados por 3D Studio Max. Esto es precisamente lo que intentamos evitar cuando elegimos DirectX y no OpenGL para implementar el Paquete3D.

El formato elegido fue el .ase, que se caracteriza por tener formato de texto ASCII. Esto tiene la desventaja de que cargarlo es mucho más lento que cargar un fichero binario donde no son necesarias todas las traducciones de los valores de las cadenas de texto. Sin embargo, tiene la ventaja principal de que podemos leer el contenido, e incluso modificarlo, pero sobre todo podemos leerlo. Gracias a esto pude comprobar con bastante certeza que los datos exportados eran correctos antes de ponerme a escribir código, porque en vista del éxito de los intentos de exportación anteriores no estábamos como para fiarnos de cualquier formato.

El resultado de mi trabajo de estos últimos días ha sido la clase *MallaAnimada.* Su constructor recibe un nombre de fichero .ase y, como la clase *MallaRigida,* puede dibujarse y proyectar sombras. Como puede adivinarse, ambas clases derivan de una clase base común *Malla*, que permite usar el polimorfismo para simplificar el funcionamiento de la iluminación, a la que le viene bien poder tratar ambas clases de mallas de forma homogénea, como se verá en el siguiente apartado.

Internamente, la clase *MallaAnimada* contiene una lista de submallas. Cada submalla es una *MallaRigida* junto con un array de matrices. Cada matriz representa la ubicación de esa submalla en un momento dado de la animación. Así, una malla animada se compone

de una serie de mallas rígidas que se mueven independientemente. Para dibujar o proyectar la sombra de una malla animada, simplemente se utiliza el tiempo dado como parámetro para escoger el momento concreto de la animación que se quiere mostrar. La matriz que se escoja de una submalla debe combinarse con la matriz de mundo actual para seguir permitiendo que las mallas animadas se ubiquen en cualquier punto del espacio como cualquier otra.

Como cabe esperar, la mayor parte del código de esta clase está destinado a leer los ficheros ase. Todo este código se ha escrito desde cero, empezado por la lectura de tokens del fichero (eliminar espacios, comillas de las cadenas), la lectura de comandos (cada uno de los cuales puede contener un número variable de parámetros y de subcomandos), etc., etc. En la versión final es capaz de leer mallas, con sus vértices, coordenadas de texturas, materiales, referencias a ficheros de texturas, pero sin duda lo más complicado ha sido leer la animación.

El hecho de que las mallas compuestas de submallas se almacenen en forma de jerarquía ha hecho necesario hacer investigación sobre cuál es la manera en que la información contenida en las submallas padre afectan a las hijas, especialmente cuando varias partes del cuerpo heredan informaciones combinadas de rotación: muchas pruebas, hipótesis descartadas,... en definitiva ingeniería inversa.

Después de decenas de horas de trabajo, el resultado es la posibilidad de leer mallas y animaciones fieles a las versiones originales en 3D Studio Max, cosa que constituye una de las características más interesantes de nuestro proyecto.

Iluminación

Sin duda alguna, este es el apartado más interesante de todo el Paquete3D, debido a que incorpora una característica muy avanzada de los juegos de última generación, como es la proyección de sombras volumétricas en tiempo real. Y, como todas las otras partes del Paquete3D, la filosofía de diseño de maximizar la facilidad de uso ha conducido a la obtención de una interfaz realmente sencilla para el cliente:

```
inline void activarLuz(int num,bool activar=true);
inline void setLuz(int num,Luz *luz);
inline void setLuzDireccional(int num,Vector direccion,ColorLuz color);
inline void setLuzPuntual(int num,Vector posicion,ColorLuz color,
                          float alcance,bool hiperLuz=false);
inline void procesarIluminacion();
inline void dibujar(Malla *malla,Modificador flags=0);
```

Todo lo que necesita el cliente conocer para añadir un gran atractivo a la ambientación de las escenas con luces y sombras es esos 6 métodos, pertenecientes a la clase singleton *Graficos*.

Funcionan del mismo modo que lo hacen las luces en DirectX u OpenGL, existen 256 luces disponibles que se pueden configurar con sus parámetros (posición, color, alcance,...), activar y desactivar. En general, las tarjetas gráficas no permiten más de 8 luces activadas simultáneamente.

En la primera versión que hubo de la iluminación en el Paquete3D, estos métodos se limitaban a invocar al método equivalente de DirectX sin más. Lo único interesante era que mientras DirectX obliga al cliente a rellenar los diferentes parámetros de una estructura que describe a una fuente de luz para configurarla, el Paquete3D proporcionaba métodos para realizar configuraciones de luces sencillas en una sola línea: *setLuzPuntual* y *setLuzDireccional* son versiones particulares de *setLuz*, al que de hecho invocan internamente.

Pero veamos ya el funcionamiento de la proyección de sombras en tiempo real. Este término se refiere a sombras que adoptan la forma de los objetos que las proyectan, que permiten apreciar todos sus detalles, su movimiento, y que se distribuyen sobre la geometría del escenario con una exactitud total. Se han inventado dos filosofías diferentes para realizarla.

La primera de ellas consiste básicamente en usar una textura blanca sobre la que se dibujan, en negro, los objetos que deben proyectar sombras, usando como cámara la fuente de luz. Esa textura se proyecta sobre los objetos que deben recibir la sombra (por ej. el escenario), mezclándose con sus texturas normales. Esta técnica obliga a realizar las siguientes operaciones:

- Averiguar sobre qué polígonos se debe proyectar la sombra. Esta es una operación de visibilidad equivalente a averiguar qué polígonos deben dibujarse por estar en el campo de visión de la cámara. Puede preprocesarse si la fuente de luz es estática (inmóvil).

- Calcular las coordenadas de textura que deben usarse para proyectar la textura con las sombras sobre el escenario. Se trata de transformar las coordenadas 3D del escenario en coordenadas 2D usando las mismas matrices de transformación que se usaron para dibujar las sombras. También puede precalcularse para fuentes de luz estáticas.

- La técnica obliga a dibujar una vez más los objetos que proyectan sombras (para generar sus sombras), y una vez más los objetos que las reciben (para añadirles las sombras), aumentando la complejidad de la escena.

Una ventaja de esta técnica es que no es demasiado costosa. De hecho, nada impide, como hacen algunos juegos, utilizar modelos de bajo nivel de detalle para dibujar las sombras. Además, gracias a que cuando se dibujen las sombras sobre el escenario, se les aplicarán los filtros bilineales que realizan las tarjetas gráficas a todas las texturas, los bordes de la sombra no son abruptos, sino que están más suavizados y resultan más realistas que con la otra técnica.

Sin embargo, la principal desventaja de esta técnica es su falta de versatilidad. Tal y como se ha explicado, sólo vale para fuentes de luz cónicas, tal y como una linterna o una farola que ilumina el espacio debajo de ella. Si se quiere implementar para una fuente de luz direccional, tal como el Sol, se debe hacer de una forma diferente, tratando cada objeto por separado, ya que no se puede tener una textura lo bastante grande como para incluir sombras detalladas de muchos objetos distantes a los que el Sol puede iluminar.

También hay que introducir modificaciones específicas para las fuentes de luz puntuales, como una antorcha llevada por un personaje. No se puede dibujar sobre una textura todos los objetos que están alrededor de la antorcha porque no se puede abarcar 360º. La manera sería usar 6 fuentes de luz, hacia arriba, abajo, izquierda, derecha, delante y detrás, y combinar sus efectos. Esto conllevaría un aumento del coste enorme, ya que probablemente cada objeto se tendría que redibujar varias veces, incluyendo tanto los que proyectan sombras como los que las reciben, por estar incluidos simultáneamente en varias de las 6 zonas.

Esta técnica tampoco proporciona un método sencillo para hacer *self-shadowing*, es decir, objetos que proyectan sombras sobre sí mismos, y si se quiere que los objetos proyecten sombras unos sobre otros, deben ordenarse según su distancia a la fuente de luz e irse

procesando de cerca a lejos, alternando el dibujo de las sombras con su aplicación, para que no acaben afectando las sombras de objetos lejanos a la fuente de luz a objetos más cercanos.

Por todas esas razones es previsible que la segunda filosofía de proyección de sombras en tiempo real se vaya extendiendo y haciendo más habitual en el mundo de los videojuegos. También por esas razones nosotros hemos escogido esta segunda filosofía. Nuestro trabajo lo hemos basado en el excelente artículo de Eric Lengyel incluido en la bibliografía, que trata esta filosofía con todo detalle.

La segunda técnica se basa en calcular los *volúmenes de sombra* de los objetos. El volumen de sombra de un objeto es el espacio al que el objeto impide llegar a la luz por estar tapado por él. El algoritmo consiste básicamente en lo siguiente:

- Una vez dibujada la escena con todos los objetos, se generan las mallas de polígonos que cubren los volúmenes de sombra de cada uno de los objetos que deban proyectar luces. Estas mallas contienen los polígonos que separan el espacio iluminado de los volúmenes de sombra, orientados hacia fuera.

- De todos esos polígonos, unos estarán orientados hacia la cámara y otros no. Se dibujan los que no estén orientados hacia la cámara, pero no sobre el frame buffer (no afectando a la escena), sino marcando los píxeles que no pasen el test de profundidad sobre el stencil buffer. El stencil buffer es una zona de la memoria de vídeo de las tarjetas gráficas que permite anotar diferentes cosas en cada píxel.

- Después se dibujan los polígonos orientados hacia la cámara, desmarcando del stencil buffer los píxeles que no pasen el test de profundidad.

El resultado de esas operaciones es que solamente han quedado marcados sobre el stencil buffer los píxeles que formen parte de la intersección de los volúmenes de sombra con lo que se haya dibujado. Resulta obvio que estos píxeles son los que deben quedar oscurecidos por las sombras; quizá no tan evidente sea el por qué. Lo mejor es verlo por uno mismo en una escena con un objeto, una luz y una pared y ver que píxeles pertenecen a cada una de las categorías por las que hemos pasado.

Las operaciones descritas son la evolución de una versión anticuada que tenía el problema de que las sombras dejaban de funcionar cuando la cámara se introducía dentro del volumen de sombra.

Hasta aquí la primera parte del algoritmo. Hay dos opciones para la segunda parte:

- bien se oscurecen todos los píxeles de la pantalla excluyendo los que queden fuera de las sombras.

- bien se redibujan todos los objetos que se desee que reciban sombras, teniendo activadas las luces que las causan, y excluyendo los píxeles que queden dentro de las sombras.

Como siempre, las dos versiones tienen sus ventajas y sus inconvenientes. La primera es que es sin duda mucho más eficiente. Tan sólo una pasada más sobre la imagen oscureciendo los píxeles interiores a las sombras y ya se habría terminado. De hecho tiene la gran ventaja de que el coste de la proyección de sombras es independiente de la complejidad de la geometría sobre la que se proyecta. Con esta versión se obtiene que los objetos proyectan sombras sobre sí mismos y sobre otros objetos, de una forma precisa.

Pero tiene algunos defectos. En primer lugar, el modo en que se iluminan las cosas cuando hay varias luces simultáneamente no es muy realista. Si en una única pasada se oscurecen todos los píxeles interiores a alguna sombra, se oscurecerán igual superficies que deberían recibir más luz por estar tapadas por menos objetos que otras. Otro problema es que, de hecho, las sombras se ven incluso en zonas a las que no llegan las luces que se supone que las causan. Por ejemplo, si hay una luz puntual debida a la presencia de una antorcha, un objeto podría hacer oscurecerse una zona del escenario que está fuera del alcance de la antorcha. Esto es un problema importante en los juegos, en los que se intenta mejorar la eficiencia desactivando las luces que se encuentran lejos de los objetos: las sombras se verán aparecer y desaparecer bruscamente al acercarse o alejarse de las luces sin posibilidad de evitarlo.

La segunda versión evita todas las desventajas a costa de una mayor complejidad computacional: después de dibujar toda la escena, generar los volúmenes de sombra y dibujar sus dos pasadas, hay que redibujar todos los objetos que se desea que reciban sombras, esta vez teniendo activadas las luces que las causan, y enmascarados los píxeles a los que no llegan.

Si por cada luz que haya se añade a la escena su contribución, mezclándose con el resto, se puede obtener correctamente el resultado de iluminar la escena con varias luces: cada píxel recibe exactamente las luces que lo alcanzan y que no son tapadas por algún objeto intermedio. Y además las sombras se funden perfectamente con las zonas a las que las luces no llegan sin apariciones ni desapariciones bruscas.

Adicionalmente, a pesar de su claramente menor eficiencia, permite optimizaciones que no admite la otra versión para atenuar esto un poco. Pueden usarse modelos de bajo nivel de detalle para simplificar la generación de los volúmenes de sombra, que la otra versión no admitía porque aparecerían sombras inconsistentes en el modelo que las proyecta debidas a la diferencia entre los dos niveles de detalle. Con esta versión no tienen por qué aparecer tales inconsistencias porque los modelos no tienen por qué proyectar sombras sobre sí mismos.

Pues bien, nosotros hemos optado por esta segunda versión y todas estas características pueden probarse en el juego.

Independientemente de la versión que se elija, la técnica de los volúmenes de sombra tiene la gran ventaja de la versatilidad: es apto para fuentes de luz de cualquier tipo (direccionales, puntuales, cónicas...), para que los objetos proyecten sombras mutuamente o sobre sí mismos, etc.

Las primera desventaja es que en general es mucho más ineficiente. El cálculo del volumen de sombra puede compensarse con el cálculo de las coordenadas de texturas en el método anterior y ambas técnicas requieren redibujar los objetos de una forma u otra. Pero la gran diferencia estriba en la necesidad de tasa de relleno. Con los volúmenes de sombra, puede que, en un momento dado, varias mallas de las que delimitan los volúmenes de sombra pasen delante de la cámara (aunque no se vean), obligando a rellenar la pantalla muchas veces.

Otra desventaja es que existen objetos que por su naturaleza pueden proyectar sombras mucho mejor con el método de dibujar las sombras en texturas. Por ejemplo, una verja se dibuja en general con una textura que tiene zonas opacas y transparentes: eso es perfectamente compatible con el primer método, pero el método de los volúmenes de sombra generaría dichos volúmenes en base a la geometría de los polígonos y no de las texturas. Y, por supuesto, sería catastróficamente lento generar volúmenes de sombra en base a una geometría que contuviera todos los recovecos de una verja introducidos de forma poligonal. También está la desventaja de que los bordes de las sombras son abruptos: a los píxeles o les afecta la luz del todo o no les afecta (según si el valor correspondiente en el stencil buffer está o no enmascarado).

Para la *generación del volumen de sombra* hemos usado un algoritmo que es más eficiente que el algoritmo trivial que viene con DirectX como ejemplo de programación. El algoritmo tiene como entrada una malla de polígonos y como salida otra, que es el volumen de sombra. Esta última malla consta de 3 clases de polígonos:

- Los polígonos de la malla que estén orientados hacia la fuente de luz.

- El resultado de proyectar los polígonos anteriores en la dirección de la fuente de luz (por ejemplo, sumándoles la diferencia entre sus vértices y la fuente de luz multiplicada 1000 veces) y orientados hacia el lado contrario.

- Los polígonos que unen a las dos clases de polígonos anteriores. Estos polígonos resultan de extender en la dirección de la fuente de luz la silueta de la malla vista desde la fuente de luz.

Lo más complicado de lo anterior es obtener la silueta de la malla vista desde la fuente de luz. El algoritmo trivial y menos eficiente es de orden cuadrático en el número de polígonos. Consiste en recorrer todos los polígonos de la malla, insertando sus aristas en una lista. Como cada arista participa de dos polígonos, el resultado será una lista que tiene cada arista dos veces. Si se borran todas las aristas que estén repetidas, todas se cancelarán unas a otras. Pues si se hace eso no para todos los polígonos, sino sólo para aquellos orientados hacia la fuente de luz (producto escalar de su normal con el vector que lleva a la luz mayor que 0), el resultado es que se cancelarán todas las aristas excepto las de la silueta.

El algoritmo que hemos usado nosotros es de orden lineal en el número de aristas. Se basa en precalcular la lista de aristas de la malla, almacenando para cada una de ellas las normales de los dos polígonos en los que participa. Para obtener la silueta de la malla vista desde la fuente de luz basta recorrer esa lista, quedándonos con las aristas que tengan a sus dos normales una apuntando hacia la fuente de luz y otra hacia el otro lado.

Otro problema que puede surgir con esta técnica de proyección de sombras es que los volúmenes de sombra se recorten por el plano lejano al dibujarse (algo que cabe esperar si al proyectarlos pretendemos que cubran todo lo que tengan detrás sin límite de distancia y en consecuencia los agrandemos cuanto podamos). La solución está en usar matrices de proyección infinita, que permiten dibujar sin plano lejano. La clave para calcularlas es usar las mismas fórmulas habituales para la generación de matrices de proyección, solo que haciendo el límite cuando f tiende a infinito siendo f la distancia al plano lejano. El artículo al que hicimos referencia al principio de este apartado incluye todos los detalles, así como una demostración de que la pérdida de precisión en el buffer de profundidad al usar este tipo de matrices no es muy significativa. De todos modos, tuvimos que rehacer los cálculos nosotros porque el artículo está basado en OpenGL, que usa un sistema de coordenadas diferente al de DirectX. El resultado puede encontrarse en las funciones que generan este tipo de matrices

en *Paquete3D.h,* que son las versiones que no incluyen la distancia al plano lejano como parámetro.

Sólo queda comentar cómo se ha obtenido la interfaz tan sencilla que hemos visto al principio del apartado. Recordemos que basta con activar las luces, configurar sus parámetros y dibujar los objetos. Después de todo ello sólo hace falta hacer una llamada al método *procesarIluminacion*.

En primer lugar hay que decir que el Paquete3D permite elegir para cada malla que se dibuje individualizadamente si se desea que proyecte y/o reciba sombras, y para cada luz también. Lo primero se especifica en los flags que se pueden incluir en la llamada al método que dibuja las mallas. Lo segundo es un parámetro más que se puede configurar de las luces.

Internamente, se mantiene una lista de todas las luces activadas que no van a causar sombras, y otra de las que sí, a las que en el código fuente se denomina "hiperluces". Para cada hiperluz se tiene, además, la lista de todas las mallas que se han dibujado con ella activada y que querían proyectar sombras, y la lista correspondiente para las que querían recibir sombras. Cuando se dibuja una malla especificando que se quiere que proyecte sombras, se añade a la lista correspondiente de todas las hiperluces activadas. Cuando se dibuja una malla que se quiere que reciba sombras, se añade a la lista correspondiente también. Además, si una malla se dibuja y quiere recibir sombras, cuando se dibuja inicialmente primero se apagan todas las hiperluces activas y luego se vuelven a encender (porque las contribuciones de las hiperluces activas ya se añadirán al final enmascarando los píxeles interiores a las sombras).

Al final, cuando el cliente ordena procesar la iluminación, se recorren todas las hiperluces que fueron activadas. Para cada una de ellas hay que añadir su contribución sobre la escena. Para ello, primero se apagan todas las demás luces, incluyendo la luz de ambiente general. Después se generan todos los volúmenes de sombra de las mallas dadas de alta en su lista de mallas que deben proyectar sombras, y se enmascaran los píxeles adecuados siguiendo el algoritmo descrito antes. Por último, se redibujan todos los objetos dados de alta en la lista de mallas receptoras de sombras, sin afectar a los píxeles enmascarados. Al redibujar las mallas se hace de modo transparente y aditivo, esto es, el color resultante de la malla iluminada, únicamente con esta luz se suma al color presente en la escena, permitiendo que se acumulen las contribuciones de las diferentes hiperluces. Al final se vuelven a encender todas las luces activas tal y como estaban.

Todas estas operaciones se realizan de la forma más eficiente posible. Como se ha visto, son necesarios recorridos rápidos de todas las luces (para desactivarlas y reactivarlas en el procesamiento final), y de todas las hiperluces (para desactivarlas y reactivarlas antes y

después de dibujar por primera vez una malla de las que quieren recibir sombras). Todo ello sin alterar el funcionamiento habitual de DirectX u OpenGL, que permiten tener 256 luces que se pueden activar, desactivar y configurar independientemente (excepto que no se pueden tener más luces activadas simultáneamente de las que permite la tarjeta gráfica).

Las estructuras de datos necesarias para realizar todo esto eficientemente han sido: el array con los datos de configuración y estado de activación o no de cada una de las 256 luces disponibles, una lista de los índices de ese array que contienen luces activas, y una lista de los índices de ese array que contienen hiperluces activadas. Por el hecho de tener que mantener esa información actualizada, las llamadas a los métodos de activación y configuración de las luces ya no están implementadas directamente con llamadas a DirectX, sino que requieren comportamiento adicional, pero el resultado obtenido, no tener que recorrer las 256 luces haciendo búsquedas de luces de una u otra clase cada vez que se dibuje una malla que reciba sombras, merece la pena. Además, para no tener que almacenar para cada una de las 256 luces un array con espacio suficiente para almacenar gran cantidad de mallas (las listas de emisores y receptores de sombras), ni tampoco sufrir la penalización en tiempo real de usar una estructura de datos dinámica (los news y deletes deberían evitarse en lo posible, y más en operaciones muy frecuentes como éstas), se ha usado un array preasignado de espacio para listas de mallas para hasta 10 luces simultáneas, cuyas posiciones se van asignando y desasignando dinámicamente a las diferentes luces a medida que van siendo necesarias, con el consiguiente mantenimiento de los punteros de la siguiente posición libre, etc.

Y todo esto permanece oculto al cliente, al que le bastan unas sencillas líneas de código para crear escenas con una iluminación interesantísima y realizada de la manera más eficiente posible.

Hay que añadir que dado que la interfaz de la iluminación es de tan alto nivel de abstracción, no sólo se cumple el objetivo de que sea trasladar la aplicación a otro entorno gráfico sin cambiar más que el Paquete3D, de hecho es posible cambiar el algoritmo de proyección de sombras (por ejemplo, pasar a la técnica de dibujar sobre texturas) sin que el código de la aplicación deba cambiarse en lo más mínimo.

Por último, debo decir que todas estas funcionalidades se encuentran implementadas en la clase *Iluminacion*, a la que la clase Graficos llama cuando se lo piden los clientes. El motivo de la no inclusión de todo esto en la clase Graficos es el evitar su excesivo crecimiento y que se convierta en una clase monolítica difícil de mantener. La generación de los volúmenes de sombra se encuentra en la clase MallaRigida, que es donde está accesible la información necesaria: la geometría de las mallas.

Sonido

A pesar de que lo más importante de éste proyecto y lo más complicado de la programación de los juegos en general es la generación de los gráficos, nuestro motor no estaría completo si no diera la oportunidad a los juegos que lo usen de añadir el interés y el mayor grado de ambientación que proporcionan los sonidos y la música.

Por esa razón, el Paquete3D también dispone de clases para reproducir sonido en 3D y música en mp3, usando DirectX. Se trata de las clases Sonido y Musica. Ambas clases son totalmente independientes del resto, del mismo modo que el sonido es independiente de los gráficos. En otras palabras, es posible reimplementar el Paquete3D para OpenGL o cualquier otro entorno gráfico y que el sonido siga funcionando mediante DirectX; o al revés, podría usarse otra librería que no sea DirectX para implementar la parte de sonido del Paquete3D sin tener que cambiar el resto.

La clase *Musica* no consta más que del código del programa de reproducción de música en mp3 que viene como ejemplo de DirectShow, introducido en los diferentes métodos que la forman. DirectShow es la parte de DirectX que permite reproducir música y vídeo de diferentes formatos.

La clase *Musica* es un singleton, al igual que la clase Graficos: tiene constructor privado y variable de clase puntero a la única instancia también privada; y un método público para acceder a dicha instancia, creándola si es la primera vez que se le llama. Es lógico que lo sea puesto que no se necesita (ni se puede) reproducir más de una música simultáneamente.

Una vez obtenida la instancia única, el cliente no tiene más que usar los métodos disponibles para cargar y reproducir un fichero de música, cuyas interfaces son las más sencillas posibles:

```
void abrir(char *nombreFichero);
void play();
void stop();
void setPos(float pos);
float getPos();
bool terminada();
```

Como puede verse, los métodos no requieren explicación alguna. Se proporcionan incluso métodos para escoger la parte de la canción por la que se quiere empezar a reproducir que, a modo de curiosidad, internamente se puede especificar u obtener con una precisión de décimas de microsegundo.

La clase *Sonido* es la última que queda por tratar en todo el *Paquete3D*. Permite reproducir un número arbitrario de fuentes de sonido simultáneas, ya sean estáticas o dinámicas, y en 3D.

Sonido en 3D quiere decir que la posición del sonido con respecto a la del oyente se tiene en cuenta para reproducir el sonido de una forma u otra. Por ejemplo, en una configuración con 2 altavoces, si uno tiene una fuente de sonido delante puede notar como el sonido pasa de un altavoz a otro si la cámara gira. Incluso se nota una diferencia si el sonido se encuentra detrás de la cámara: se oye como más grave. Por supuesto, la intensidad con la que se oye el sonido depende también de la distancia del sonido a la cámara. Además, se percibe la modificación de todos estos parámetros durante la reproducción de un solo sonido con total continuidad. Por último, el sonido en 3D permite también que sean los drivers de la tarjeta de sonido los que se encarguen de reproducir el sonido del mejor modo posible, aprovechando las características del hardware disponible: los drivers sólo reciben la información "de alto nivel" sobre la ubicación relativa del sonido y el oyente y ellos se encargan del resto. Así, en una configuración con 4 altavoces la percepción de la dirección de la que viene el sonido es aún más nítida. Pero la característica principal de la clase Sonido es la sencillez con que el cliente puede hacerlo todo.

Normalmente para reproducir un sonido en 3D usando DirectX hay que rellenar una estructura de datos que incluye los vectores de posición y velocidad del sonido, la intensidad o alcance y el modo en que quiere reproducirse (con o sin 3D), entre otras cosas; y usar un puntero a esa estructura como parámetro de una llamada. Además, si hay sonidos dinámicos (las fuentes de sonido se mueven), el cliente debe volver a rellenar tales estructuras para cada fuente de sonido y para cada fotograma. Con el *Paquete3D*, basta una sola línea de código para reproducir sonidos, ya sean estáticos o dinámicos. Echemos un vistazo a la interfaz de la clase sonido y veamos el modo en que esto se ha conseguido.

```
static void inicializar(HWND hWnd);
static void setOyente(ParametrosOyente &parametros);
static void remezclar();

Sonido(char *nombreFichero);
~Sonido();
void play(bool bucle=false);
void play(Vector pos,float dist,bool bucle=false);
void play(Vector *pos,float dist,bool bucle=false);
void stop();
```

Como puede comprobarse existen 3 métodos play para reproducir sonidos. El primero de ellos es el de interfaz más sencilla y permite reproducir un sonido sin efecto 3D, esto es, el sonido se reproduce con total intensidad por todos los canales de sonido. Equivale a reproducir un sonido en 3D que se encuentra en la misma posición que el oyente.

El segundo de ellos reproduce un sonido estático (no se mueve) con efecto 3D. Se especifica la posición y el alcance, y la clase internamente se encarga de rellenar la estructura de datos a enviar a DirectX para permitir que al cliente del Paquete3D le baste con una línea de código.

El tercero de ellos, como puede verse, se diferencia del anterior únicamente en que recibe la posición del sonido no como un vector, sino como un puntero a un vector. Se trata de un sonido dinámico. Gracias a este método la clase Sonido tiene toda la información que necesita para reproducir un sonido dinámico. Mientras dure la reproducción, el sonido emanará de la posición indicada por el vector al que apunte el puntero especificado. Esta es la interfaz más adecuada posible porque en los juegos los sonidos dinámicos van asociados a objetos que se mueven (por ejemplo, a un proyectil), de los que se mantiene siempre en alguna parte el vector de posición. Con este método, pueden reproducirse sonidos 3D dinámicos con una única línea de código, siempre que se tenga la única precaución necesaria de que mientras dure el sonido el puntero debe apuntar a una dirección válida: no es correcto borrar el objeto que contiene la posición del sonido cuando éste aún no ha terminado. Pero eso no es ningún problema, porque si el objeto asociado a una fuente de sonido ha desaparecido, lo más normal es que se desee parar el sonido con el método stop (en su destructor, por ejemplo).

El alcance del sonido viene especificado en las mismas unidades de distancia que los vectores, y es la distancia por debajo de la cual el sonido se oye con intensidad máxima; si la distancia es d, el sonido se oirá al 100% en distancias menores que d y, por encima de ella, al 50% en 2*d; al 25% en 4*d, etc.

Una característica más es la posibilidad de reproducir un sonido en bucle, una y otra vez, tal y como permiten los 3 métodos. Esto permite la inclusión de sonidos de ambiente asociados a lugares en un escenario. Por ejemplo, en una catarata puede ponerse un sonido de agua 3D estático que se reproduzca en bucle.

Aún falta por comentar las características de la clase Sonido que eliminan la necesidad de hacer una llamada por cada sonido dinámico y por cada fotograma. Existen 3 métodos de clase como puede verse: el primero, inicializar, sólo arranca la reproducción de sonidos con DirectSound y lo llama automáticamente la clase Aplicacion3D de modo que el usuario no tiene que conocerlo. El segundo, setOyente, es el que especifica los parámetros del oyente en cada fotograma, y que las aplicaciones deberían actualizar junto con la cámara, aunque no hay ningún impedimento a que el "ojo" se encuentre en un sitio y los "oídos" en otro sitio diferente. Debe especificarse una estructura que contiene posición, velocidad y orientación, así como factores por los que se multiplican las unidades de distancia, la atenuación de los sonidos y el efecto doppler de forma universal. Afortunadamente, dicha

estructura viene equipada con un constructor que recibe todos esos parámetros, consiguiendo también que la actualización de los sonidos pueda realizarse con una sola línea de código. Ejemplo:

```
Sonido::setOyente(ParametrosOyente(pos,pos-posAnt,dir,Vector(0,1,0)));
```

Como puede verse, en la llamada no ha sido necesario especificar los últimos 3 parámetros porque es muy inusual el tener que cambiarlos por lo que el constructor contiene los valores habituales en parámetros por defecto.

Al último de los 3 métodos de clase lo llama automáticamente la clase *Aplicacion3D* en cada fotograma después de actualizar. Se trata del método *remezclar* que le dice a DirectSound que recalcule las proporciones relativas con que deben mezclarse los diferentes sonidos en función de las actualizaciones realizadas en sus posiciones, el oyente, etc. Este proceso podría realizarse tras cada una de esas actualizaciones y así no sería necesaria la llamada a *remezclar*, pero es más eficiente realizar los cálculos una sola vez en cada fotograma.

Además, así el procedimiento *remezclar* se usa también para realizar la tarea necesaria para no tener que hacer una llamada para cada sonido dinámico y para cada fotograma. Básicamente, en la clase *Sonido* existe una lista de todos los sonidos dinámicos que se están reproduciendo en un momento dado. Cuando se reproduce un nuevo sonido dinámico, se añade a la lista. Cuando un sonido dinámico termina o el cliente lo para, se borra de la lista. El método *remezclar* recorre esa lista rellenando por el cliente la estructura de datos que DirectSound necesita con todos los nuevos parámetros del sonido. La nueva posición la obtiene accediendo al puntero que fue especificado en la llamada a *play*. La nueva velocidad la obtiene automáticamente haciendo la diferencia entre la posición actual, y la posición en la última llamada a *remezclar*, que debe irse almacenando de fotograma a fotograma.

La especificación de la velocidad de los sonidos, de la que el Paquete3D libra a los clientes, constituye una de las partes más tediosas de usar DirectSound, pero es necesaria para producir el efecto Doppler. El efecto Doppler es la distorsión que sufren los sonidos debida a su velocidad relativa al oyente. Si un sonido se aleja del oyente a gran velocidad, las "cimas" o los "valles" de la onda se alejan entre sí y el sonido es más grave (como en las carreras de Formula 1); si el sonido se acerca rápido al oyente, se hace más agudo. También se llama efecto Doppler a distorsiones de esta misma naturaleza en otros tipos de ondas (la luz, etc.).

Con la clase *Sonido*, pues, se obtiene reproducción de sonidos 3D estáticos o dinámicos y con efecto Doppler "de serie" sin que el cliente tenga que hacer un esfuerzo adicional más allá de las sencillas llamadas a los métodos *play* que se han visto.

4.3.- Árboles BSP

Introducción

Los árboles BSP son unas estructuras de datos muy usadas en motores 3D y en muchas ocasiones con diversos propósitos. Su nombre proviene de las siglas, en inglés, de árbol binario de partición espacial. Como su propio nombre indica, un árbol BSP es un árbol binario usado para dividir el espacio en regiones.

Los árboles BSP y sus algoritmos fueron desarrollados por Fuchs y Naylor. Permiten el acceso a una región cualquiera del espacio en tiempo lineal con el número de polígonos del escenario. Su utilidad en nuestro motor 3D es la detección de colisiones, tanto con polígonos del escenario para evitar atravesar las paredes, como con otros objetos.

Esta estructura consiste en un árbol binario donde cada nodo contiene un polígono. El plano del polígono del nodo divide lógicamente el espacio en dos subespacios, siendo el frontal aquel que se encuentra del lado hacia donde apunta la normal del plano y el subespacio trasero el otro. Además del polígono, cada nodo contiene un puntero a cada uno de estos dos subespacios, los cuáles a su vez son subárboles.

Esta representación de escenas no es única, es decir, un mismo escenario puede ser representado utilizando diferentes árboles BSP, unos con mayor número de nodos que otros o más o menos balanceados, lo que se refleja en que unos sean más rápidos que otros para desplegar objetos.

Escenario y árbol BSP

Mediante el árbol BSP se puede saber si un punto en concreto es habitable o sólido. Por habitable entendemos un punto en el que puede haber un objeto tal como un personaje. Las zonas habitables son aquellas que pueden ser visitadas, es decir, forman la parte visible del escenario. Por sólido entendemos un punto que se encuentra tras una pared, bajo el suelo o encima del techo. Determina aquellas zonas del escenario que no van a ser vistas y en las que no pueden haber objetos. Un personaje puede moverse libremente por las zonas habitables pero no podrá entrar en aquellas zonas sólidas. Las paredes imponen el límite entre ambas zonas. Para generalizar, un polígono del escenario determina una pared, aunque el polígono realmente represente el suelo o el techo de una estancia.

Como ya se ha visto, el árbol BSP está formado por nodos que pueden ser internos o nodos hoja. Un nodo interno contiene el plano que divide al espacio y un nodo hoja tiene información de si la zona a la

que representa es habitable o sólida. En la generación del árbol se determina si un nodo hoja es habitable si queda en la parte frontal del polígono y sólido si queda en la parte trasera.

Generación del árbol BSP

El árbol BSP que representa una escena es generado una vez que se tiene el conjunto de polígonos que conforman la escena. En nuestro caso, estos polígonos forman parte de una lista de polígonos. Estos polígonos deben ser convexos. Nuestro motor 3D usa únicamente triángulos por lo que nos aseguramos que se cumple esa condición.

El árbol BSP es generado eligiendo cualquier polígono como polígono raíz. Este polígono es utilizado para dividir el espacio en dos subespacios. Uno contiene los polígonos que se encuentra enfrente del polígono raíz, relativo a su normal, y el otro subespacio los polígonos detrás de él. Si algún polígono pertenece a ambos subespacios es dividido por el plano en dos partes y cada uno de estos polígonos es introducido en su respectivo subespacio. Si un polígono está en el mismo plano del polígono raíz puede colocarse en cualquier subespacio. Un polígono del subespacio frontal y otro del trasero se vuelven el hijo frontal e hijo trasero del nodo y cada hijo es recursivamente usado para dividir su subespacio de la misma manera. El algoritmo continúa hasta que cada nodo contiene un solo polígono. A continuación se muestra el código del motor para generar un árbol BSP:

```
ArbolBSP *ArbolBSP::generar(Lista<Poligono> *poligonos)
// construye este subárbol BSP partiendo de la lista de polígonos
// especificada. Aquí va el algoritmo de generación del árbol BSP
// (ésto se hace sólo en el preprocesador)
{
  return ArbolBSP::generarRecursivo(poligonos, false, NULL);
}

ArbolBSP *ArbolBSP::generarRecursivo(Lista<Poligono> *poligonos,
                                     bool solido, Pared *paredPadre)
// genera el arbol BSP de forma recursiva. Se indica si el nodo es
// solido (solo para nodos hoja). Recibe la pared del nodo padre
// (necesario si es un nodo hoja)
{
  Lista<Poligono> frontal, trasero;
  Poligono poli, pdiv;

  if(poligonos->longitud()==0) {
    return new NodoHojaBSP(solido, paredPadre);
  } else {
    pdiv = SeleccionaYBorraPoligono(poligonos);
    for(int i = 1; i<=poligonos->longitud(); i++) {
      poli = poligonos->item(i);
      switch(PosicionPoligono(poli, pdiv)) {
      case POSPL_ENFRENTE:
        frontal.anadir(poli);
        break;
      case POSPL_DETRAS:
        trasero.anadir(poli);
        break;
      case POSPL_DENTRO:
```

```
        // Si el poligono ya esta se le ignora
        break;
      default: // Poligono debe ser partido
        PartePoligonoYAnade(poli, pdiv, frontal, trasero);
        break;
      }
    }
    poligonos->anadir(pdiv);
    return new NodoInternoBSP(pdiv, &frontal, &trasero);
  }
}
```

Véase también el código de las constructoras de NodoHojaBSP y NodoInternoBSP

El árbol estará formado por nodos internos, que a su vez tienen el subárbol frontal y el trasero, y nodos hoja que indican si esa hoja es sólida o habitable. Una hoja es sólida si representa la zona del espacio que queda detrás de un polígono. Para que el algoritmo funcione bien los polígonos solo se pueden tocar en alguna de sus aristas.

Debido a que durante la creación del árbol BSP pueden ocurrir divisiones de polígonos, en la mayoría de las escenas el número de polígonos después de la creación del árbol BSP es mayor que el número de polígonos originales. La elección del polígono para particionar cada subespacio repercute drásticamente en el número de polígonos divididos.

De todos los árboles BSP que representan la misma escena, aquel con el menor número de nodos será el que permita un desplegado más rápido del mismo, debido a que para cada polígono que compone el escenario, se deben realizar operaciones para determinar la posición de un punto respecto al plano de ese polígono, por lo que la disminución del número de polígonos reduce el número de cálculos. En caso de que dos o más árboles contengan el mismo número de nodos, aquel mejor balanceado será más rápido, pues será menor la profundidad de recursión. Este árbol es llamado el árbol óptimo, sin embargo, el algoritmo para encontrarlo tarda demasiado tiempo pues el número de combinaciones de árboles es muy grande.

Un escenario debe ser válido para que el árbol que se genere sea correcto y no haya ambigüedades. Existe ambigüedad si dos árboles diferentes sobre el mismo escenario no resultan en la misma distinción entre zonas sólidas y zonas habitables. Esto se entenderá mejor con un ejemplo. Supóngase un escenario compuesto por un polígono que es el suelo y una pared que reposa en el suelo y deja la mitad de este a cada lado. La normal del suelo apunta hacia arriba y la normal de la pared apunta hacia la derecha, por ejemplo. Si en la generación del árbol se toma como polígono raíz el suelo nos quedaría el árbol 1, mientras que si se toma la pared como raíz nos quedaría el árbol 2.

```
+ = frontal
- = trasero

1.                suelo
                 /+    \-
            pared       (sólido)
           /+    \-
(habitable)       (sólido)

2.                    pared
                      /+       \-
            suelo der.           suelo izq.
           /+         \-       /+         \-
      (habitable) (sólido)  (habitable) (sólido)
```

Según el árbol 1, sólo es habitable la parte que queda a la derecha de la pared y encima del suelo. Según el árbol 2, todo lo que está encima del suelo es habitable. Este escenario es ambiguo pues dependiendo de cómo se genere el árbol se tendrán diferentes zonas sólidas y habitables.

Un requisito que debe cumplir un escenario para que no sea ambiguo es que los polígonos solo se pueden tocar entre sí en las aristas.

Posición de un polígono respecto a un plano

Para determinar si un polígono poli se encuentra enfrente, detrás o a ambos lados del plano de otro polígono plano es necesario evaluar cada vértice de poli para determinar el lado donde se encuentra. Esto se hace calculando la distancia del vértice al plano. Si la distancia es positiva el punto está enfrente del plano, si la distancia es negativa se encuentra detrás y si la distancia es cero el punto se encuentra dentro del plano. Para el cálculo de la distancia de un punto al plano se usa un método del Paquete 3D llamado PlanoDistancia. La constante ME define un margen de error para comparaciones y otros cálculos. Se va a considerar que un punto está dentro del plano si la distancia en valor absoluto no supera el valor de ME.

El siguiente código realiza la evaluación del lado donde se encuentra un polígono respecto a un plano tomando en cuenta el factor ME para el error de redondeo:

```
int ArbolBSP::PosicionPoligono(Poligono poli, Poligono plano)
{
	int front = 0, back = 0, planar = 0;
	Plano dplano;

	for(int i = 0; i<3; i++) {
		PlanoCon3Puntos(&dplano, &plano.vs[0].p, &plano.vs[1].p,
				&plano.vs[2].p);
```

```
			switch(PosicionPuntoPlano(&poli.vs[i].p, &dplano)) {
			case POSPL_ENFRENTE: front++; break;
			case POSPL_DETRAS: back++; break;
			default: front++; back++; planar++; break;
			}
		}
		if(planar==3)
			return POSPL_DENTRO;
		else if(front==3)
			return POSPL_ENFRENTE;
		else if(back==3)
			return POSPL_DETRAS;
		return POSPL_PARTE;
}

int ArbolBSP::PosicionPuntoPlano(Vector *punto, Plano *plano)
{
		float dist;
		dist = PlanoDistancia(plano, punto);
		if (dist>ME)
			return POSPL_ENFRENTE;
		else if (dist<-ME)
			return POSPL_DETRAS;
		else
			return POSPL_DENTRO;
}
```

Los métodos *PlanoCon3Puntos* y *PlanoDistancia* son propios del paquete 3D.

División de polígonos

Durante la creación del árbol BSP un polígono puede ser dividido por un plano. Esta división generará dos polígonos, cada uno a un lado del plano. Es muy posible que uno de los dos polígonos tenga un número de vértices superior a tres. Como solo trabajamos con triángulos, ese polígono deberá ser dividido a su vez en otros dos polígonos que ahora sí tendrán tres vértices. El código que se encarga de partir un polígono mediante un plano es el siguiente:

```
void ArbolBSP::PartePoligonoYAnade(Poligono poli, Poligono plano,
Lista<Poligono> &frontal, Lista<Poligono> &trasero)
{
		Lista<Poligono> frontales, traseros;
		PartirPoligono(&poli, plano.plano, &frontales, &traseros);
		for (int i = 1; i<=frontales.longitud(); i++)
			frontal.anadir(frontales[i]);
		for (int j = 1; j<=traseros.longitud(); j++)
			trasero.anadir(traseros[j]);
}
```

Este código usa el método PartirPoligono del paquete 3D que devuelve los polígonos que quedan a cada lado del plano mediante dos listas.

Elección del polígono base

La elección del polígono raíz para cada subespacio es muy importante, ya que repercute en el número de polígonos divididos y en la profundidad del árbol. Un árbol profundo o muy desequilibrado es contraproducente a la hora de ser utilizado.

Existen varias técnicas para la elección de este polígono, unas más rápidas que otras. Aquí solo se explica la técnica utilizada en este motor.

El polígono elegido se determina seleccionando el mejor de todos. El mejor polígono es aquel que deje un número más o menos igual de polígonos en cada subespacio y que corte a un número mínimo de polígonos. La elección del mejor polígono genera árboles muy cercanos al óptimo pero es la técnica que más tiempo consume. No obstante, como el procesamiento del árbol solo se va a realizar una vez por escenario, es conveniente emplear algo más de tiempo en su generación para mejorar su rendimiento.

A continuación viene el código que selecciona un polígono de la lista. Como se puede ver, a cada polígono se le da una puntuación que viene determinada por:

- La diferencia entre número de polígonos que deja a cada lado. Si se coge un polígono que equilibre los polígonos a cada lado esta diferencia será cercana a cero.

- El número de polígonos que parte. Este valor se multiplica por 3 para dar más peso a aquellos polígonos que corten un número menor de polígonos. Es preferible coger un polígono que deje ambos lados un poco desequilibrados antes que tomar uno que parta un número grande de polígonos.

Según esto, el mejor polígono será aquel con puntuación más baja.

```
Poligono ArbolBSP::SeleccionaYBorraPoligono(Lista<Poligono> *poligonos)
{
  int topscore = 1000000; // puntuacion inicial
  int back, front, planar, splits, score, index;
  Poligono splitter, bestsplit;

  for(int i = 1; i<=poligonos->longitud(); i++) {
    splitter = poligonos->item(i);
    front = 0;
    back = 0;
    planar = 0;
    splits = 0;
    for(int j = 1; j<=poligonos->longitud(); j++) {              if(i!=j) {
        switch(PosicionPoligono(poligonos->item(j), splitter)) {
        case POSPL_ENFRENTE: front++; break;
        case POSPL_DETRAS: back++; break;
        case POSPL_DENTRO: planar++; break;
        default: splits++; break;
```

```
        }
      }
    }
    score = abs(front - back) + 3 * splits;
    if(score<topscore) {
       topscore = score;
       bestsplit = splitter;
       index = i;
    }
  }
  poligonos->borrar(index);
  return bestsplit;
}
```

Lectura y escritura de los árboles BSP

Para cada escenario, es posible generar el árbol BSP solamente una vez y tenerlo almacenado en un fichero. A la hora de utilizar un escenario, se lee el árbol BSP del fichero y se ahorra tener que procesar el árbol todas las veces.

La clase que implementa los árboles BSP incorpora métodos para leer y para escribir en un fichero recibiendo tanto el nombre del fichero o bien un fichero ya abierto. La información del árbol que es necesaria guardar y posteriormente leer es la siguiente:

- Nodos hoja: información de si la hoja es sólida o no
- Nodos internos: plano que divide al espacio, información del subespacio frontal y trasero

La escritura es un proceso recursivo, ya que al escribir un nodo interno hay que escribir sus dos hijos. Para diferenciar si un nodo es hoja o interno se escribe un valor booleano antes de la información del nodo.

La lectura es bastante simple y consiste en leer el valor booleano y a continuación leer la información del nodo. Si es un nodo interno habrá que usar recursión y para cada hijo volver a leer el tipo de nodo y su información.

Los métodos para leer y escribir en fichero son: ArbolBSP::leer y ArbolBSP::escribir. Ambos pueden recibir el nombre de un fichero o un descriptor de fichero (FILE *) y leerán o escribirán en ese fichero la información de los árboles. Esto último es útil si la información de los árboles BSP se encuentra incluida en un fichero con más datos.

Objetos en los árboles

Como ya se ha dicho, los árboles BSP se van a utilizar para realizar toda la detección de colisiones. Estas colisiones se pueden tener tanto con paredes del escenario como con otros objetos que estén en él. Más

adelante se hablará un poco de cómo se realiza la detección de colisiones.

En los nodos hoja del árbol hay listas que contienen referencias a los objetos que se encuentran en esas hojas. La clase árbol provee métodos para manejar estos objetos:

```
void ArbolBSP::anadir(Objeto *objeto)
```

Añade el objeto al árbol. Para ello, según la posición del objeto, va descendiendo por el árbol hasta un nodo hoja donde ese objeto es añadido. Como los objetos tienen forma y tamaño es posible que se encuentre ocupando más de una hoja. En este caso el objeto se añade a todas las hojas que ocupe. A su vez, el objeto debe saber en que hojas se encuentra por lo que cada vez que se añade a una hoja esta misma hoja es añadida a la lista de hojas del objeto.

Si un objeto no tiene forma, no va a chocar con nada y por tanto no será añadido al árbol.

```
void NodoHojaBSP::borrar(Objeto *objeto)
```

El método para borrar un objeto solo se encuentra en los nodos hoja. Esto es así porque solo el objeto se va a borrar de las hojas que ocupa. Para eso el objeto debería borrarse de todas las hojas que tiene en su lista de hojas. Cuando un objeto se mueve primero se borra de todas las hojas y luego se vuelve a añadir.

Cuando un objeto se borra de una hoja, esa hoja es borrada de la lista de hojas del objeto.

Si se intenta borrar un objeto que no existe no se hará nada.

Detección de colisiones

Para la detección de colisiones se dispone de dos métodos diferentes. En ambos métodos se detectan colisiones con las paredes del escenario y con objetos que se encuentren en él que hayan sido añadidos al árbol.

```
void ArbolBSP::colision(Objeto *objeto, Objeto **choque)
```

Detecta si el objeto choca con algún otro objeto o con una pared. Si no se produce ningún choque el valor del puntero choque será NULL, en caso contrario apuntará al objeto con el que ha

chocado. Si el choque se produce con una pared el objeto será de la clase Pared. Para la detección de colisiones se tiene en cuenta la forma que tenga el objeto.

Lo que hace este método es obtener la posición del objeto e ir bajando por el árbol hasta llegar a un nodo hoja. Si la hoja es sólida entonces es porque ha chocado con una pared. Si la hoja es habitable comprueba si choca con alguno de los objetos que están en esa hoja. Si en un nodo interno un objeto ocupa algo de la parte frontal y de la parte trasera se comprueba si hay colisión en ambos subespacios. Cuando choca con una pared (se llega a una hoja sólida) hay que determinar con que pared se ha chocado. Esto se hace al descender por el árbol mediante un parámetro que es la pared donde es posible que el objeto choque. Inicialmente no hay ninguna pared, pero si se llega a un nodo interno en el que el objeto esté en ambos subespacios y no hay choque en el árbol frontal, es posible que el objeto choque con la pared de ese nodo interno. En este caso se pasa esa pared como posible pared de choque al árbol trasero. Al final, la pared con la que choca será aquella que pertenezca al nodo interno más profundo en el que el objeto se encuentre en los dos subespacios. Se puede dar el caso que el objeto esté por completo en una hoja sólida y no se tendría ninguna pared candidata para el choque. Entonces se toma la pared del nodo inmediatamente superior al nodo hoja.

```
void rayo(Vector *origen, Vector *destino, Vector *puntoChoque, Objeto
**objetoChoque)
```

El método rayo se usa para comprobar si un rayo que va de origen a destino choca con algún objeto. Devuelve el objeto con el que ha chocado y el punto donde se ha producido el choque. Si no choca con nada el objeto devuelto apuntará a NULL. Este método comprueba si existen colisiones con paredes y con objetos en el árbol.

El funcionamiento de este método es básicamente el siguiente. Para un nodo interno comprueba en que parte del espacio se encuentra el punto origen y destino. Si ambos están en la misma parte, baja por el árbol correspondiente. Si están en diferente parte, primero comprueba si hay colisión en la parte del punto origen y si no la hay mira en la parte del punto destino. Para detectar con que pared choca el rayo se usa la misma técnica explicada en el método colision, pero aquí sólo se actualiza la posible pared de choque si el rayo cruza la pared desde su parte frontal a su parte trasera. Si se produce al revés es imposible que el rayo choque con esa pared ya que, si el rayo atravesara realmente esa pared, el punto de origen se encontraría en una zona sólida y se detectaría colisión en el árbol trasero.

Usando objetos

La clase Objeto es una clase abstracta de la que heredan todos los tipos de objetos diferentes que puede haber en el escenario. Si queremos crearnos un tipo de objeto en particular, tenemos que crear una clase hija de Objeto y redefinir los métodos virtuales que necesitemos. A la hora de crear la clase hija hay que tener en cuenta los siguientes puntos:

- Hay que llamar al constructor de la clase Objeto.
- Todos los objetos tienen una forma que se usa para detectar colisiones. Inicialmente al construirse un objeto tiene forma nula y no choca con nada (su forma es NULL). La clase Objeto tiene un método para definir la forma llamado setForma. Este método se usará también para cambiar la forma de un objeto.
- Cada objeto es de un tipo concreto que se especifica al llamar al constructor de la clase Objeto. En Objeto.h hay definidos varios tipos de objetos diferentes pero se pueden añadir más. Mediante el método getTipo se puede obtener el tipo de un objeto. Esto es útil cuando dos objetos colisionan para interactuar entre ellos.
- Todo objeto que quiera añadir a otros en el escenario debe usar el método Escenario::anadir(). El puntero al escenario es de tipo void por lo que habrá que realizar una conversión. La llamada quedaría así:

```
((Escenario *)escenario)->anadir(new Objeto(...))
```

- El método usado para actualizar el objeto se llama actualizar. En él, el objeto se puede mover o realizar cualquier otra acción. Si el objeto debe ser destruido devolverá true en este método. Esta es la única manera de borrar un objeto del escenario.
- La actualización de un objeto debe ser proporcional al tiempo transcurrido desde el frame anterior.
- Para saber la posición de un objeto se usa el método getPos y para cambiar su posición se hace moverA. Si al mover un objeto este choca con algo se vuelve a su posición anterior y se ejecutaría el método haChocado. Este es un método virtual que en la clase Objeto no hace nada. En un objeto creado por nosotros podemos redefinir este método para que interactúe con el objeto con el que choca. Al cambiar

la forma de un objeto es posible que choque con objetos con los que antes no chocaba. Para detectar esto, tenemos dos alternativas:

1. Cambiar la forma del objeto y llamar a moverA(getPos()). Esto no mueve al objeto pero mira si colisiona con algo. Si hay colisión se ejecutaría haChocado.

2. Cambiar la forma del objeto y llamar a (Escenario *)escenario)->colision(this, &objetoChoque), donde objetoChoque es un puntero a un objeto. Si objetoChoque es distinto de NULL es porque se ha producido un choque.

En ambos casos, habría que restaurar su forma original e interactuar con el objeto con el que ha chocado si se quiere.

- Como ya se ha dicho antes, un objeto que no deba chocar no tiene forma. Pero además de esto se pueden ahorrar las llamadas a moverA para mover el objeto. El método moverA tiene sentido para objetos con forma. Un objeto sin forma puede tener un vector con su posición y acceder a él directamente cuando se mueva sin tener que llamar a moverA.

- Un objeto se dibuja en su método dibujar.

Con esto, se ha visto un breve resumen de cómo se manejan objetos. A continuación se habla con detalle de los métodos y propiedades de Objeto y de Forma.

```
Objeto::Objeto(void *escenario, TipoObjeto tipo)
```

Es el constructor del objeto. Un objeto heredado en su constructor deberá llamar a este constructor indicando el escenario en el que se encuentra el objeto y su tipo. En *Objeto.h* están definidos varios tipos de objetos diferentes, pero se puede añadir cualquier tipo que se quiera. El tipo de los objetos es útil a la hora de interactuar entre ellos, por ejemplo cuando dos objetos chocan. El constructor de objeto crea un objeto sin ninguna forma y, por tanto, no choca con nada.

En el constructor de la clase hija, para darle una forma al objeto hay que llamar al método *setForma*. Se puede establecer la posición inicial del objeto llamando a *moverA.*

```
void Objeto::anadirHoja(void *hoja)
```

Los objetos tienen una lista de las hojas del árbol en las que están. Este método es llamado por el árbol BSP cuando se añade un objeto para notificar al objeto que se encuentra en una determinada hoja. Generalmente, este método no será necesario llamarlo.

```
void Objeto::borrarHoja(void *hoja)
```

Este método se usa para borrar una hoja de la lista de hojas en las que se encuentra este objeto. Igual que el método anterior, este método es llamado por el árbol BSP cuando se borra un objeto de una hoja y por lo tanto no es necesario usarlo.

```
Lista<void *> Objeto::getHojas()
```

Devuelve una lista con punteros a las hojas del árbol donde se encuentra el objeto. Este método es usado por el escenario.

```
virtual void Objeto::leer(FILE *fichero)
```

Lee del fichero ya abierto la información del objeto. Por defecto, en la clase Objeto no hace nada. Se puede redefinir para objetos heredados. Se puede usar para leer los objetos de un escenario de un fichero.

```
virtual void Objeto::escribir(FILE *fichero)
```

Escribe el objeto en el fichero ya abierto. Por defecto, en la clase Objeto no hace nada. Se puede redefinir para objetos heredados. Se puede usar el método leer para recuperar los objetos escritos con este método.

```
virtual bool Objeto::actualizar(float tiempo)
```

Se encarga de actualizar el objeto en cada frame. Recibe el tiempo en segundos que ha transcurrido desde la última vez que se actualizó. Aquí es donde el objeto puede moverse de manera proporcional al tiempo que ha pasado desde la anterior actualización. Para moverse puede llamar al método moverA. El método debe devolver un valor bool que será true si el objeto debe destruirse.

```
virtual void Objeto::dibujar(int frame)
```

Este método debe ser redefinido para que se dibuje el objeto. Antes de dibujarse debe comprobar que no se haya hecho ya en el frame actual. Esto se haría así:

```
if (this->frame == frame)
  return;
 this->frame = frame;
```

Al dibujarse, en general, debe cambiar la matriz del mundo (WORLD_MATRIX) para situar su malla en la posición del objeto y a continuación dibujar la malla. Si se modifica algún modo en el que se dibuja (transparencias, texturas, etc.), tras terminar de dibujar habrá que restaurar a su estado original.

```
Forma *Objeto::getForma()
```

Devuelve la forma del objeto. Si el objeto no tiene forma devuelve NULL.

```
Vector *Objeto::getPos()
```

Devuelve la posición en la que se encuentra el objeto. Se obtiene un puntero al vector de posición del objeto. Este puntero se debe usar únicamente para leer la posición. Si se quiere cambiar la posición a un objeto hay que llamar al método moverA que detecta si hay colisiones.

```
TipoObjeto Objeto::getTipo()
```

Devuelve el tipo del objeto. En el archivo Objeto.h hay definidos algunos tipos de objetos.

```
void Objeto::setForma(Forma *nuevaForma)
```

Cambia la forma del objeto. La forma de hacer esto es:

```
setForma(new Forma(...))
```

No hace falta destruir la forma anterior del objeto antes de darle una nueva forma porque ya lo hace setForma. En el destructor por defecto de Objeto también se destruye la forma. En el código anterior se debe sustituir Forma por una clase hija de Forma, aquella que represente a la forma que queremos dar al objeto.

```
void Objeto::moverA(Vector *nuevaPos)
```

Desde el objeto se debe llamar a este método para actualizar su posición y comprobar las colisiones. Si se produce una colisión este método llamará a haChocado y el objeto no será movido.

```
virtual void Objeto::haChocado(Vector *pos, Objeto *objetoChoque)
```

Un objeto que quiera implementar interacción con los objetos con los que choca o simplemente ser notificado de este hecho debe redefinir este método que por defecto no hace nada. Recibe la posición a la que el objeto se ha intentado mover y donde se ha producido el choque. También recibe el objeto con el que ha chocado.

```
void *Objeto::escenario
```

Puntero al escenario donde se encuentra el objeto. Necesario para interactuar con él. Es necesario realizar la conversión del puntero a Escenario *.

```
int Objeto::frame
```

Indica cuál fue el último frame en el que se dibujó. En el método dibujar se debe modificar su valor.

Usando formas

Las formas son útiles para la detección de colisiones. Un objeto puede ser dibujado con una malla muy compleja formada por muchos polígonos. Calcular si dos mallas de polígonos intersecan entre sí es demasiado costoso, por ello se usa una forma asociada que debe ser lo más aproximada posible a la forma y tamaño real del objeto. Existe una clase Forma con unos métodos definidos de la que pueden heredarse clases para cada forma concreta que se quiera implementar. Con el motor 3D se incluye una forma básica que es *FormaEsfera* y que puede ser útil para la mayoría de los objetos simplemente especificando un tamaño diferente. Calcular si una esfera choca con una pared o con otra esfera es muy sencillo y rápido y por eso se ha tomado esta forma como una forma básica en el motor 3D. No obstante, para ciertos objetos puede que esta forma no sea conveniente y sea necesario crear otro tipo de formas. Para no caer en una disminución de rendimiento importante, las formas creadas deberían ser sencillas y el cálculo de su intersección con otros objetos muy rápido. Ejemplos de formas que se podrían implementar son *FormaCaja* (un paralelepípedo) o *FormaCapsula* (una esfera estirada en uno de sus ejes).

Las formas tienen un tipo asociado para poder identificarse fácilmente e incorporan métodos para detectar si tienen intersección con un plano, un rayo y otras formas.

A continuación se muestran los métodos de *Forma* y sus variables. Los métodos virtuales deben ser redefinidos en las formas que se creen.

```
Forma::Forma(TipoForma tipo, Vector *pos)
```

Es el constructor de la clase Forma. Una forma creada por nosotros debe llamar a este constructor e indicarle el tipo de forma y la posición del objeto al que da forma. Los diferentes tipos de forma se encuentran en el fichero Forma.h y pueden ampliarse con aquellos que se implementen. El puntero a la posición del objeto sirve para que la forma quede asociada al objeto y la forma siempre tenga acceso a la posición actual del objeto.

```
virtual int Forma::comparaPlano(Plano *plano)
```

Método que debe redefinirse en cada forma concreta. Permite comparar una forma con un plano. Debe devolver 1 si la forma está al lado positivo del plano (hacia donde apunta la normal), -1 si está en al lado negativo y 0 si el plano corta a la forma.

```
virtual bool Forma::comparaRayo(Vector *origen, Vector *destino, Vector
*choque)
```

Método que debe redefinirse en cada forma concreta. Devuelve cierto si el rayo que va de origen a destino corta a la forma, falso en caso contrario. Si el rayo corta a la forma también devuelve el punto de choque.

```
virtual bool Forma::compara(Forma *forma)
```

Método que debe redefinirse en cada forma concreta. Devuelve cierto si ambas formas intersecan en algún punto. Para ello, debe ver el tipo de la otra forma. Por cada tipo posible de forma habrá un algoritmo diferente.

```
int Forma::getTipo()
```

Devuelve el tipo de la forma.

```
Vector *Forma::getPos()
```

Devuelve la posición donde está la forma que es la misma que la del objeto al que está asociada. El puntero a vector que devuelve solo debe ser usado para leer la posición y en ningún caso para modificarla. Si se quiere modificar la posición de una forma, se debe mover el objeto asociado.

Clase FormaEsfera

Esta clase implementa la forma de una esfera de un radio arbitrario cuyo centro se encuentra en la posición del objeto. El constructor tiene la siguiente forma:

```
FormaEsfera(float radio, Vector *pos)
```

A continuación se hace un breve comentario de los métodos redefinidos de Forma:

- comparaPlano: calcula la distancia del centro de la esfera al plano, si es menor que el radio, el plano corta la esfera.
- comparaRayo: usa un algoritmo para calcular el punto del rayo que queda más cerca del centro de la esfera. Si ese punto está a una distancia igual o menor al radio es que el rayo choca con la esfera. El punto de choque que devuelve no es exactamente el primer punto en el que el rayo toca a la esfera, pero devuelve un punto que queda dentro de la esfera y que estará cercano al punto real donde el rayo choca. Se ha usado este algoritmo porque es más rápido que el que calcula el punto exacto donde el rayo toca a la esfera.
- compara: la única forma con la que se puede comparar es con otra esfera. Para comparar dos esferas se mira la distancia entre sus centros y si es menor que la suma de sus radios es porque ambas esferas se chocan.

4.4.- Listas de Visibilidad

Introducción

La implementación del paquete de las listas de visibilidad nos va a proporcionar la funcionalidad del dibujo del escenario en la pantalla.

La clase principal de este paquete y con la que va a interactuar el escenario en la clase *ListaVisibilidad*. El escenario al principio del juego construirá una instancia de esta clase y delegará en ella la responsabilidad de dibujar correctamente el escenario en la pantalla.

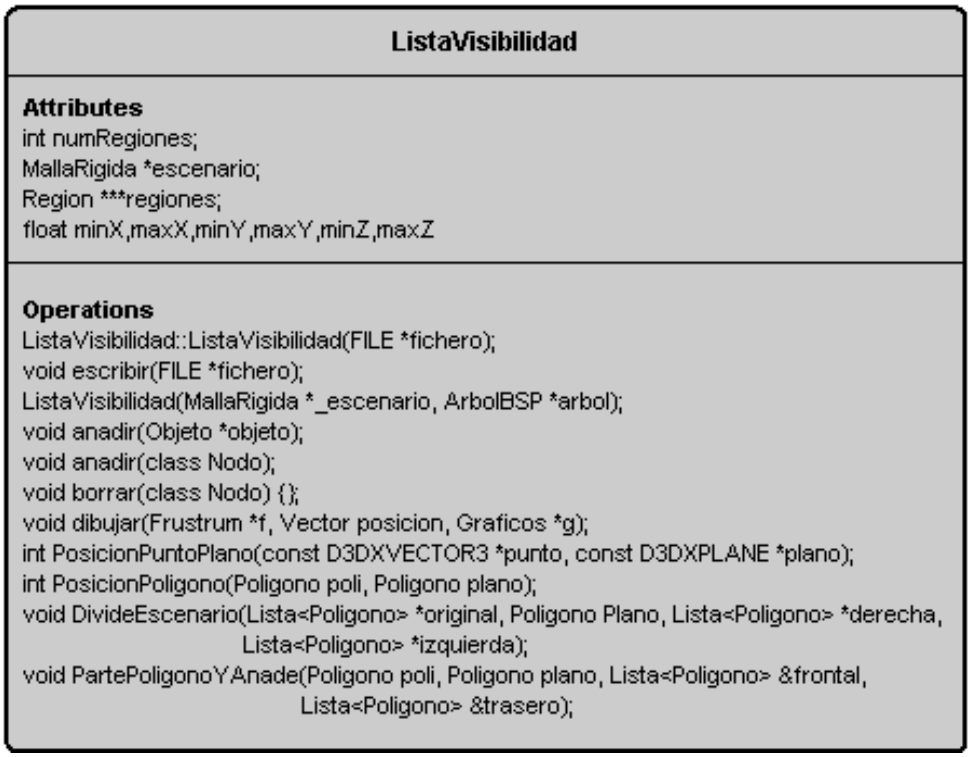


Figura 4.4.1

En la figura 4.4.1 podemos ver la clase ListaVisivilidad que nos da un poco de idea de lo que hace esta clase.

Cuando construimos una instancia de esta clase lo podemos hacer de dos modos: a partir de un fichero que haya sido guardado o a partir de la malla del escenario y del árbol BSP que debe haber sido generado con anterioridad. Nos vamos a centrar en este último método ya que es de un interés especial.

Generación de la lista de visibilidad

Generar la estructura de datos para manejar la lista de visibilidad es una tarea compleja que podríamos dividir en dos fases: la primera consistiría en obtener los polígonos que forman una región, y la segunda en calcular las regiones visibles desde cada región. A continuación pasamos a detallar cada una de las etapas.

Lo primero que tenemos que hacer con el escenario es calcular sus dimensiones, para así averiguar el tamaño de las regiones, pues el número de estas va a ser fijado por un parámetro. De momento éste parámetro lo hemos fijado en 10, que significa que se va a trocear el escenario en 100 regiones.

Después de haber calculado las dimensiones del escenario, ahora debemos construir una lista de polígonos por región que contenga los polígonos que se encuentran en dicha región.

Esto lo hacemos mediante dos bucles: uno que va haciendo "rodajas" el escenario y otro que corta dichas rodajas en trocitos.

```
for (i = 1; i <= numRegiones; i++)
{
	// Construimos un plano paralelo al eje X

	plano.vs[0] = Vertice(Vector((i*(maxX-minX)/numRegiones)+minX,
	minY,(maxZ-minZ)/2), Vector(1,0,0),0,0);
	plano.vs[1] = Vertice(Vector((i*(maxX-minX)/numRegiones)+minX,
	(maxY-minY)/2, maxZ), Vector(1,0,0),0,0);
	plano.vs[2] = Vertice(Vector((i*(maxX-minX)/numRegiones)+minX,
	maxY, minZ), Vector(1,0,0),0,0);

	PlanoCon3Puntos(&(plano.plano),&(plano.vs[0].p),&(plano.vs[1].p),
	&(plano.vs[2].p));

	columna = new Lista<Poligono>;
	restocolumna = new Lista<Poligono>;

	// Partimos el escenario con este plano, de manera que a la
	//izquierda tenemos una columna y a la derecha nos queda todo lo
	//demas

	DivideEscenario(&lp,plano,columna,restocolumna);

	// Ahora pocedemos a partir esta columna en cubos

	for (j = 1; j <= numRegiones; j++)
	{
		// Construimos un plano paralelo al eje Z

		plano.vs[0] = Vertice(Vector((maxX-minX)/2, minY, (j*(maxZ-
		minZ)/numRegiones)+minZ), Vector(1,0,0),0,0);
		plano.vs[2] = Vertice(Vector(maxX, (maxY-minY)/2, (j*(maxZ-
		minZ)/numRegiones)+minZ), Vector(1,0,0),0,0);
		plano.vs[1] = Vertice(Vector(minX, maxY, (j*(maxZ-
		minZ)/numRegiones)+minZ), Vector(1,0,0),0,0);

		PlanoCon3Puntos(&(plano.plano), &(plano.vs[0].p),
		&(plano.vs[1].p), &(plano.vs[2].p));
```

```
                fila = new Lista<Poligono>;
                restofila = new Lista<Poligono>;

                // Al dividir el escenario con este plano, en fila tenemos
                //los cubos y en restofila obtenemos todo lo que nos queda

                DivideEscenario(columna,plano,fila,restofila);

                if (fila->longitud() != 0)
                        lps[i-1][j-1] = *fila;

                *columna = *restofila;
                delete fila;
                delete restofila;
        }

        lp = *restocolumna;
        delete columna;
        delete restocolumna;
}
```

Como podemos observar este algoritmo utiliza el método *DivideEscenario*, que es el que se encarga de dividir una lista de polígonos en dos, dejando en la primera los polígonos que se encuentran a su izquierda y en la segunda los que se encuentran a su derecha. Tras el uso de varias funciones auxiliares la función *DivideEscenario* acaba usando a la función *PartirPoligono* que ya esta implementada en el paquete de los árboles BSP.

Una vez que ya tenemos las listas de polígonos lo que debemos hacer es ir generando cada una de las regiones de las que formarán la tabla de regiones.

Esto consiste en crear una tabla bidimensional de objetos *Region*. Estos objetos contendrán fundamentalmente dos cosas: la malla de polígonos que hay en esa región y la lista de regiones que es visible desde esta región.

La tarea más costosa del preprocesamiento es calcular las regiones visibles desde una región, pues teóricamente sería necesario lanzar rayos desde cada punto de la región en todas direcciones y ver cuáles de estos rayos tienen una colisión en otra región.

Lo que hacemos nosotros para que no sea una tarea infinita es lanzar una cantidad representativa de rayos, que nos da cierta seguridad de que se van a calcular todas las regiones visibles, pero que puede dar lugar a situaciones como se ven en la figura 4.4.1

Figura 4.4.1

```
// Hacemos un recorrido por todas las regiones lanzando rayos

	for (i = 0; i < numRegiones; i++)
		for (j = 0; j < numRegiones; j++)
		{
			// Preparamos las variables las variables para procesar la
			//region [i,j]

			for (i1 = 0; i1 < numRegiones; i1++)
				for (j1 = 0; j1 < numRegiones; j1++)
					anadidas[i1][j1] = false;
			anadidas[i][j] = true;

			char cad[100];
			sprintf(cad,"Tratando la region [%d,%d]",i,j);
			logFile(cad);

			for (nr=0; nr <= 10000; nr++)

			{
				o = ((float)rand())/(RAND_MAX);
				l1 = ((float)rand())/(RAND_MAX);
				l2 = ((float)rand())/(RAND_MAX);
				l3 = ((float)rand())/(RAND_MAX);
				l4 = ((float)rand())/(RAND_MAX);
				l5 = ((float)rand())/(RAND_MAX);

				if ((i==8) && (j==4))
				{
					sprintf(cad, "o=%4f, l1=%4f, l2=%4f, l3=%4f,
					l4=%4f, l5=%4f", o, l1, l2, l3, l4, l5);
					logFile(cad);
				}
				// Aqui hay que elegir un punto de la region que
				//estamos evaluando
```

```
                origen = Vector( (i*(maxX-minX)/numRegiones)+ minX+
                ((maxX-minX)/numRegiones)*l1,
                (maxY/4)+((maxY/2)*l5), (j*(maxZ-minZ)/numRegiones)+
                minZ+((maxZ-minZ)/numRegiones)*l2);


                // Aqui hay que elegir un punto de la frontera del
                //escenario

                if (o <= 0.2) // Hemos elegido un punto del suelo
                      destino  =  Vector(minX  +  (maxX-minX)*l3,
                      minY*2, minZ + (maxZ-minZ)*l4);
                else if (o <= 0.4) // Hemos elegido la pared X =
                //minX
                      destino = Vector(minX*2, minY + (maxY-minY) *
                      l3, minZ + (maxZ-minZ)*l4);
                else if (o <= 0.6) // Hemos elegico la pared X =
                //maxX
                      destino = Vector(maxX*2, minY + (maxY-minY) *
                      l3, minZ + (maxZ-minZ)*l4);
                else if (o <= 0.8) // Hemos elegido la pared Z =
                //minZ
                      destino = Vector(minX + (maxX-minX)*l3, minY
                      + (maxY-minY) * l4,minZ*2);
                else // Hemos elegido la pared Z = maxZ
                      destino = Vector(minX + (maxX-minX)*l3, minY
                      + (maxY-minY) * l4,maxZ*2);

                // lanzamos un rayo desde el origen al destino

                pchoque = Vector(0,0,0);

                objeto = NULL;

                arbol->rayo(&origen,&destino,&pchoque,&objeto);


                if (objeto != 0) // se produce un choque el pchoque
                //y la region a que pertenece
                      // este punto debe ser añadida si no lo ha
                      //sido ya
                {
                      // averiguamos en que región se encuentra el
                      //punto de choque

                      i1 = 0;
                      j1 = 0;

                      while (pchoque.x > minX + (i1+1) * ((maxX-
                      minX)/numRegiones))
                            i1++;

                      while (pchoque.z > minZ + (j1+1) * ((maxZ-
                      minZ)/numRegiones))
                            j1++;

                      // Si la region aún no había sido añadida a
                      //la lista de regiones
                      // la añadimos ahora

                      if (!anadidas[i1][j1])
                      {
                            regiones[i][j]->anadirRegionVisible
                            (regiones[i1][j1]);
                            anadidas[i1][j1] = true;
                      }
                }
          }
    }
```

Uso de la listas de visibilidad

Las listas de visibilidad nos van a permitir dibujar en pantalla el menor número posible de polígonos. El mecanismo que se emplea es el siguiente: el escenario le indica a la lista de visibilidad que dibuje los polígonos que se encuentran dentro del *frustrum* y la posición de la cámara. Entonces la lista de visibilidad realiza los siguientes pasos:

En primer lugar calcula la región en la que se encuentra la cámara, y dibuja esta región. Después recorre todas las regiones que están en la lista de visibilidad, comprobando antes de dibujarlas si están dentro o no del *frustrum*, y en caso de encontrarse dentro las dibuja.

Dibujar una región consiste en dibujar todos los polígonos y la lista de objetos incluidos en dicha región. Es decir, las balas, las bengalas y demás objetos se incluyen en la lista de visibilidad para que no se dibujen los objetos que queden fuera del alcance de la vista.

```
void ListaVisibilidad::dibujar(Frustrum *f, Vector posicion, Graficos *g);
```

4.5.- Los elementos del Juego

CLASE: Disparo

Implementa los disparos de los dos tipos de arma del juego. La pistola dispara balas, mientras que la marcadora lanza chorros de pintura. Cada disparo emite 2 sonidos: uno al ser disparado y otro al chocar. Las texturas y sonidos sólo se cargan una vez de fichero y todos los objetos de la clase Disparo las comparten.

METODOS:

- dibujar(int frame): Pinta en pantalla los disparos de la pistola, utilizando sprites con el dibujo de un balín de paintball.

- actualizar(float tiempo): Desplaza el disparo, avanzando en la dirección en la que fue disparado. Termina cuando la distancia que ha recorrido es superior a su alcance.

CLASE: Bengala

Implementa las bengalas que puede lanzar el muñeco como iluminación adicional a la incluída en el escenario. Las bengalas humeantes salen disparadas y siguiendo una trayectoria parabólica caen al suelo. Si chocan con las paredes, se quedan adheridas en el punto de choque.

MÉTODOS:

- dibujar(int frame): Dibuja la bengala, junto con un montón de partículas que forman el humo. Todos son sprites.

- actualizar(float tiempo): Avanza la bengala describiendo una parábola, siempre y cuando no haya chocado con una pared o el suelo, en cuyo caso se queda parada. La bengala va perdiendo energía luminosa y termina cuando se queda sin ella.

CLASE: Partícula

Implementa una partícula que describe un movimiento. Es muy versátil y se utiliza para simular el humo de las bengalas, los chorros de pintura y los choques de los disparos con las paredes.

MÉTODOS:

- dibujar(int frame): Dibuja la partícula, con un sprite.

- actualizar(float tiempo): Desplaza la partícula, siguiendo el comportamiento especificado en los parámetros vel y gravedad del constructor.

CLASE: Queco

Implementa el comportamiento del muñeco protagonista del juego. También controla la cámara desde la que se visualiza el escenario y el arma. El código de interacción con el jugador también está incluído aquí. Es el único elemento del juego que tiene animaciones.

MÉTODOS:

- dibujar(int frame): Coloca la cámara, dibuja la malla correspondiente al estado de la animación del muñeco y dibuja el arma que está empuñando en ese momento.

- actualizar(float tiempo): Aquí se desplaza al muñeco en la dirección en la que está mirando, y se actualiza la cámara en consecuencia. También se lee el estado de las teclas que controlan el movimiento del muñeco, y se actualizan las matrices para dibujar las mallas del muñeco y del arma.

CLASE: Escenario

Esta clase es la encargada de manejar todos los objetos anteriores. El constructor se encarga de crear las instancias de los objetos y de construir el escenario, además controla la función del preprocesamiento, es decir, comprueba si existen los ficheros precalculados y sino los genera.

MÉTODOS:

- dibujar(int frame): Este método se encarga de preparar la pantalla para el dibujado de todos los objetos y los manda dibujar en el orden correcto, es decir, lo último a dibujar son las balas y bengalas, con sus partículas.

```
void Escenario::dibujar()
  // dibuja todo
{
  int i;
       Vector posCamara = queco->getPosCamara();

       // borra el buffer de profundidad (no hace falta borrar el frame buffer
       //porque lo vamos a sobrescribir entero dibujando un fondo)
  g->limpiar(false,0x00000000,true);
```

```
  // Dibujar el cielo de fondo
  // el cielo es un cubo que acompaña a la cámara (de modo que parece que está
  //lejísimos)
  // lo dibujamos sin z-buffer (de modo que todo lo que se dibuje después
  //salga delante)
  MatrizTraslacion(&matriz,posCamara.x,posCamara.y,posCamara.z);
  g->setMatrizMundo(&matriz);
  g->setLuzAmbiental(0xffffffff); // el cielo ya es lo bastante oscuro por sí
                                  //mismo
  g->dibujar(cielo,modSinZBuffer);
  g->setLuzAmbiental(0x30303030); // dejamos la luz de ambiente como estaba

  // Dibujar el escenario
  MatrizIdentidad(&matriz);
  g->setMatrizMundo(&matriz);
  regiones->dibujar(NULL,posCamara,g);

  // Dibujar los objetos que no sean bengalas, disparos,...
  for (i=1; i<=objetos.longitud(); i++)
        if (objetos[i]->getTipo()!=OBJ_BENGALA &&
                  objetos[i]->getTipo()!=OBJ_DISPARO &&
                  objetos[i]->getTipo()!=OBJ_PARTICULA)
      objetos[i]->dibujar(0);

  // Procesar la iluminación
  g->procesarIluminacion();

  // Dibujar las  bengalas, disparos,...
  for (i=1; i<=objetos.longitud(); i++)
        if (objetos[i]->getTipo()==OBJ_BENGALA ||
                  objetos[i]->getTipo()==OBJ_DISPARO ||
                  objetos[i]->getTipo()==OBJ_PARTICULA)
      objetos[i]->dibujar(0);

  MatrizIdentidad(&matriz);
  g->setMatrizMundo(&matriz);
 }
```

- actualizar(float tiempo): En este método se comprueba si el usuario desea finalizar el juego, si ha terminado la música se vuelve a reproducir y después se manda actualizar a todos los objetos.

5.- Grafismo

Los gráficos son uno de los elementos más importantes en la valoración de un videojuego. No es suficiente tener un buen motor gráfico que permita renderizar escenas a gran velocidad, sino que esas escenas deben ser lo bastante representativas como para permitir que el jugador se sienta involucrado en el desarrollo del juego.

Los gráficos están sujetos a una restricción muy importante: el número de polígonos. Actualmente, la potencia de las plataformas de juego ha atenuado enormemente el impacto de esta restricción. Sin embargo, sigue siendo una limitación bastante importante.

Para un juego 3D se necesitan tanto objetos tridimensionales que representen los elementos de juego como gráficos 2D, que se usan para mostrar información sobre el estado de juego, o texturas para recubrir las mallas de los objetos tridimensionales.

Existe en el mercado una gran cantidad y variedad de productos software que facilitan el trabajo de grafistas y animadores. Programas como 3DS MAX, Bryce 3D, Maya o SoftImage son muy comunes en la creación de elementos tridimensionales. Para el procesamiento de gráficos 2D hay programas como Photoshop, o Paint Shop Pro.

Antes de comenzar a trabajar con estos programas es imprescindible tener una idea muy clara de los elementos que se pretenden a modelar. Por eso, en la creación de videojuegos, los grafistas y animadores no comienzan su trabajo hasta que no se ha diseñado el juego con claridad, especificando bien los distintos objetos que aparecerán en él. Es muy usual realizar bocetos y dibujos en papel de esos objetos, que son utilizados como referencia en el modelado de los mismos.

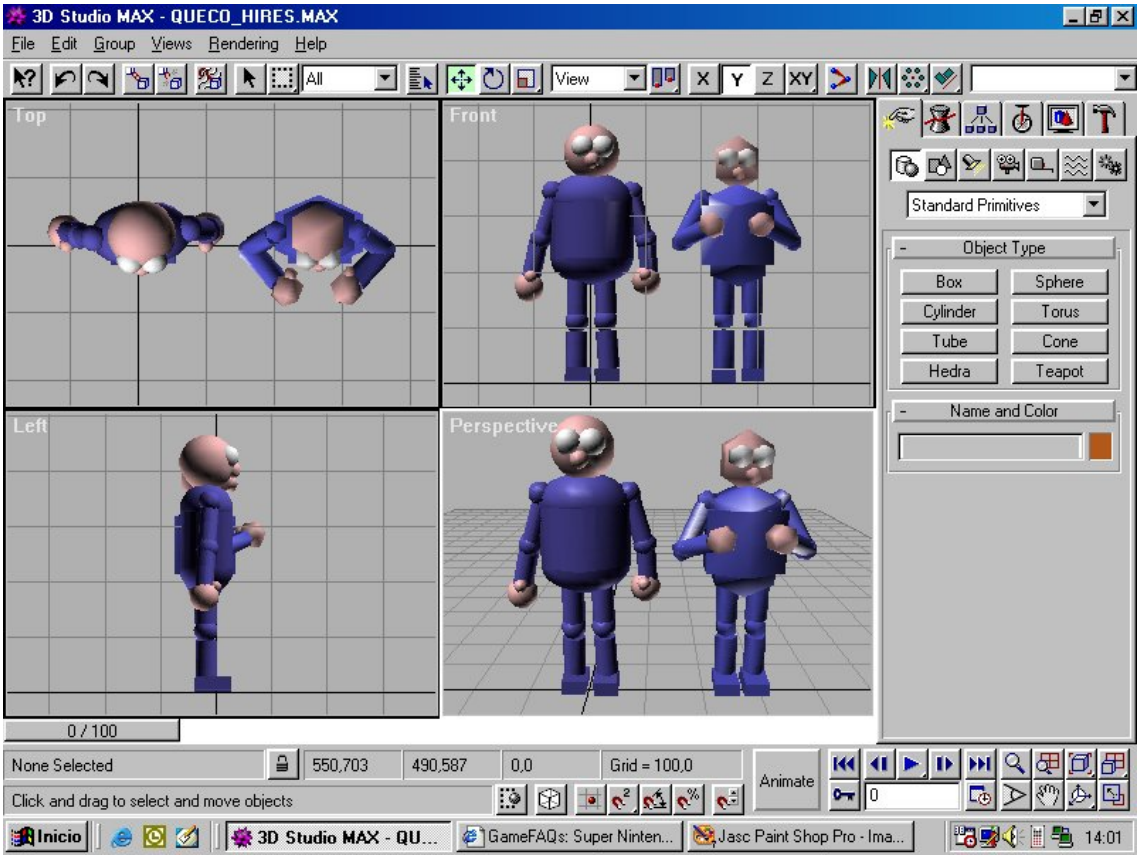
5.1 Creación de los gráficos del juego

Para construir las mallas 3D de todos los elementos del juego (escenario, muñecos y armas) se utilizó el software de modelado 3D Studio MAX 5. Las texturas del escenario se encontraron en Internet, mientras que las de los ojos del muñeco se crearon a mano usando el programa Paint Shop Pro 7.

· El muñeco

El modelo tridimensional del protagonista del juego debía disponer de elementos articulables para aplicar animación, así como tener una cantidad de polígonos que permitiera gestionar la malla eficientemente en tiempo real durante el juego.

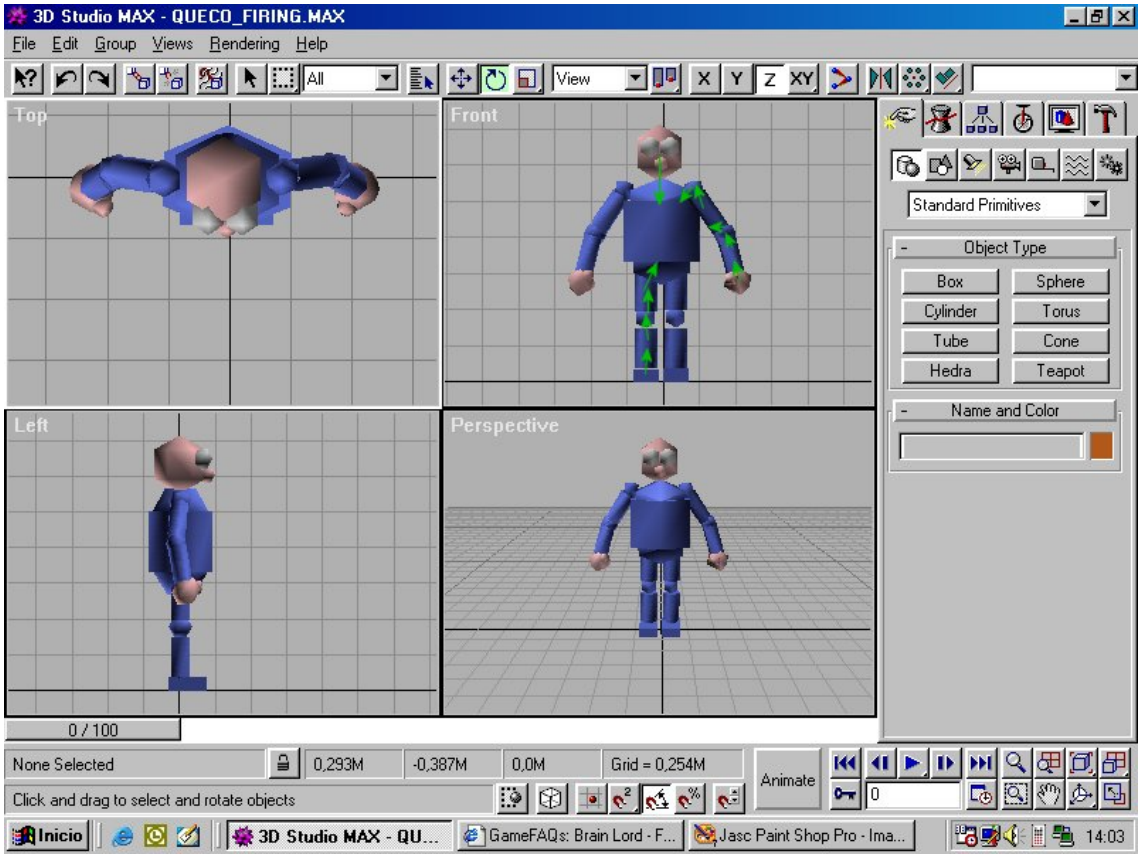
El muñeco tiene una estructura antropomórfica, creada a partir de elementos simples como esferas y cilindros. Se generaron 2 modelos distintos: uno con muchos polígonos, para generar imágenes 2D de adorno durante los tiempos de carga, y uno con pocos polígonos, que será el muñeco que el jugador controla durante la partida.



Los dos modelos del muñeco, frente a frente

Para crear las animaciones en 3DMAX, se aplicó a los distintos componentes del modelo 3D de pocos polígonos una jerarquía que, a modo de esqueleto, permitiera realizar los movimientos de forma sencilla. La jerarquía está establecida como el esqueleto de un humano, con el cuerpo como elemento principal. Así, por ejemplo, al girar el hombro derecho, todos los componentes que forman el brazo

derecho giran a la vez, permitiendo simular el movimiento de rotación del brazo. Lo mismo sucede con los codos, o las rodillas, por ejemplo.



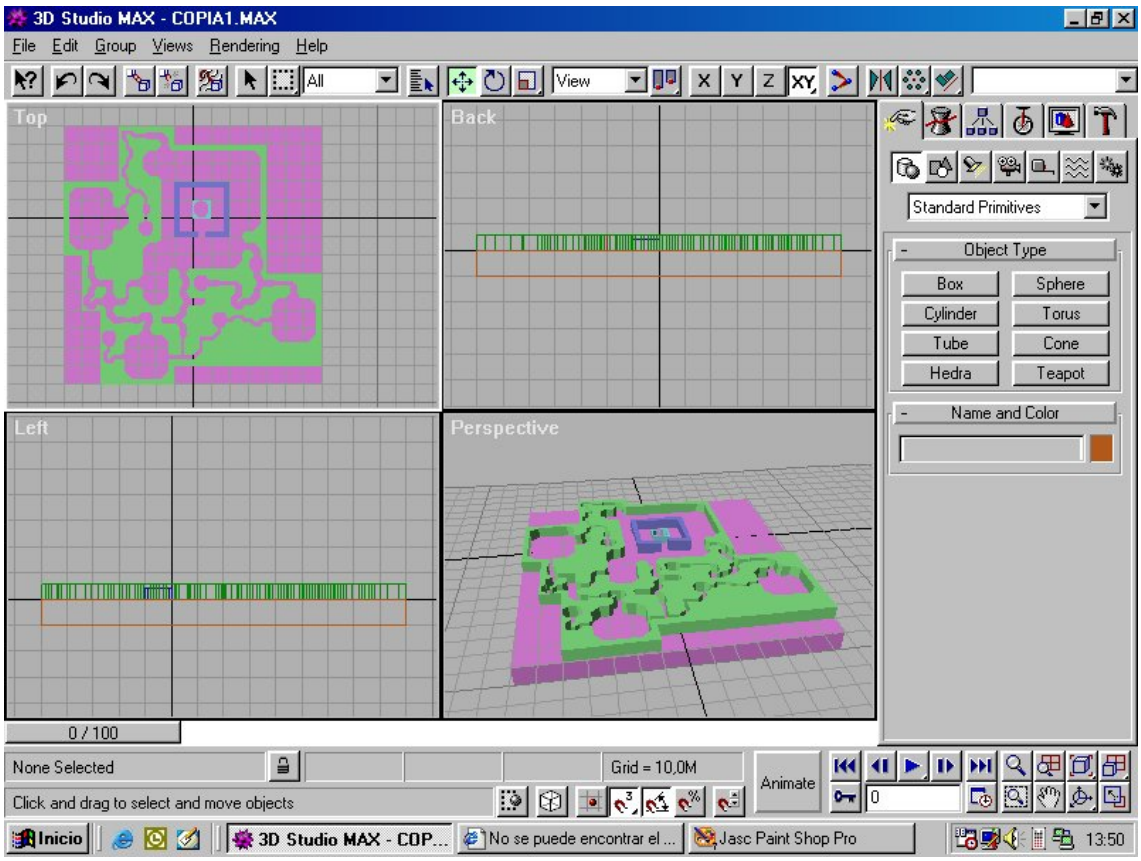
Las flechas representan la estructura de la jerarquía

El muñeco tiene un abanico de movimientos bastante limitado, pudiendo desplazarse hacia delante o hacia atrás, o lateralmente. También se crearon animaciones para las victorias y las derrotas.

· El escenario

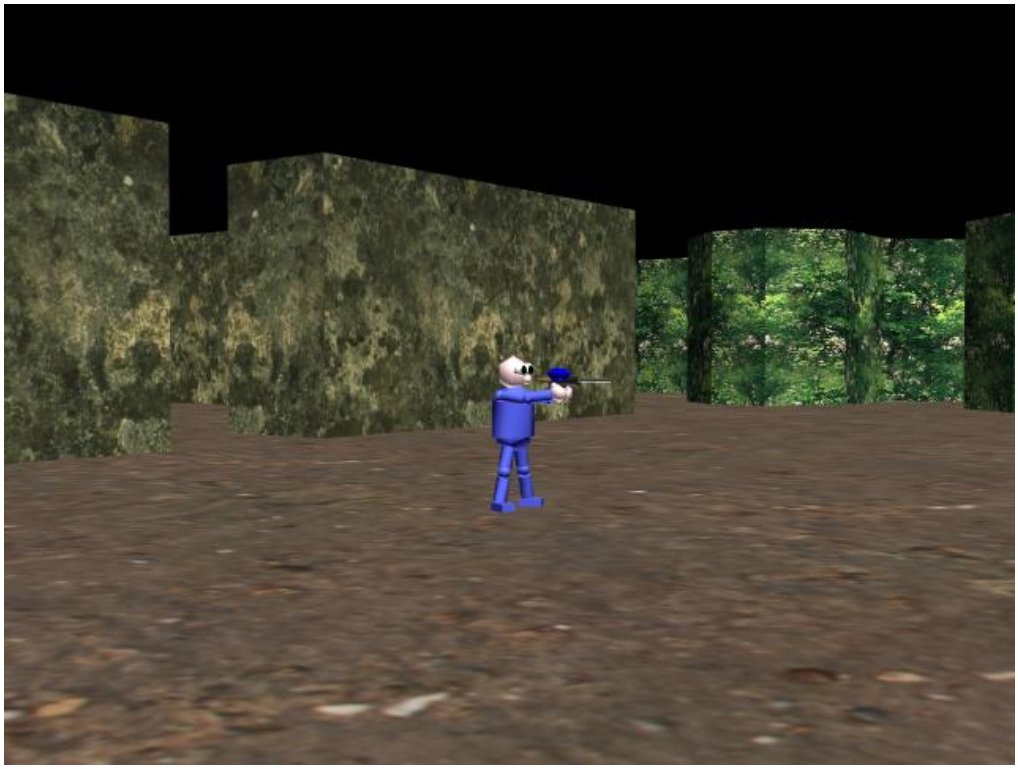
La malla del escenario fue la más problemática, porque era muy importante que el árbol BSP correspondiente fuera correcto y no permitiera que el personaje atravesara las paredes o chocara contra muros invisibles.

Para generar el escenario se utilizó un sistema muy común en los juegos 2D y que se ha adaptado para muchos juegos 3D: el sistema de mosaico o "tiles". Con este sistema, a partir de unos componentes básicos (los "tiles"), que representan trozos de calles, árboles, casas,... se puede construir un escenario gigantesco simplemente colocando tiles de forma adyacente. El escenario generado no es nada más que un mosaico de esos tiles.



El modelo 3D del escenario

El problema de este sistema reside en que los tiles adyacentes comparten varios polígonos, lo cual puede desembocar en incorrecciones en el árbol BSP. Para evitar esto, se eliminaron esos polígonos después de construir el escenario.

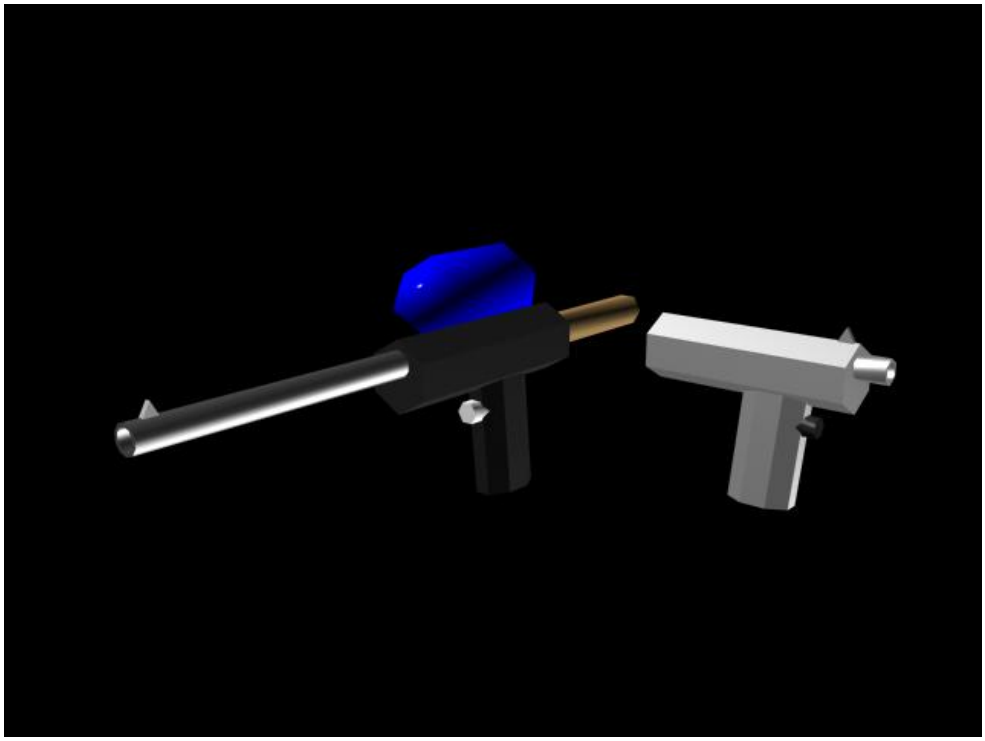


Una vista del escenario, desde la altura del muñeco

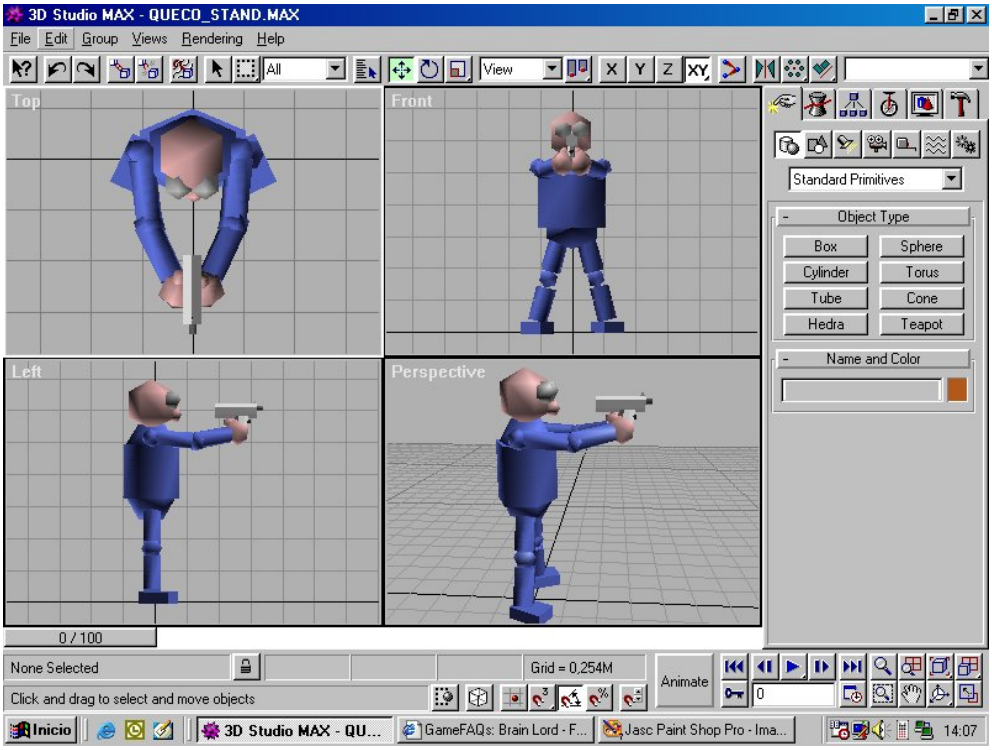
· Las armas

Las mallas de las dos armas disponibles en el juego debían cumplir el mismo requisito que el modelo del muñeco, en cuanto al número de polígonos.

Los modelos de la marcadora y de la pistola se crearon después del modelo del muñeco, con el fin de poder adaptarlas al tamaño de éste.



Las armas del juego



Los modelos de las armas se adaptaron al tamaño del muñeco

5.2.- Conversores

La descripción completa de todos los problemas que nos hemos encontrado para que nuestro motor importe los ficheros generados por el 3D Studio Max se encuentra en el apartado “Mallas y animaciones” de la sección “Paquete 3D”.

El conversor conv3ds.exe

Como allí se indica, lo primero que hicimos para convertir los ficheros fue usar la herramienta conv3ds.exe, incluida con el DirectX SDK. A esta herramienta se le indica el nombre de un fichero .3ds exportado por el 3D Studio Max, y genera un fichero equivalente traducido a formato .x (formato que se puede cargar directamente usando funciones de la librería de DirectX). Su sintaxis es:

```
conv3ds queco.3ds
```

Esa ejecución convierte el fichero queco.3ds en queco.x. Debido a que los modelos que generamos tienen una escala muy diferente que la que usamos en el juego (el muñeco sale alrededor de 200 veces más grande de lo que nosotros queremos) fue necesario usar un parámetro que permite reescalar las mallas:

```
conv3ds –s0.005 queco.3ds
```

Otros parámetros empleados fueron el –m, que funde todas las submallas de las que se componga el fichero especificado en una sola, ya que en un principio no disponíamos del código para reproducir los movimientos independientes de dichas submallas (animaciones) y no nos servían; y el –x, que genera un fichero .x en formato texto. Existen tres clases de fichero .x: en formato texto (más lentos de cargar por tener que realizar todas las conversiones de cadena a números, pero permiten leerse para verificar la corrección de los datos o cualquier particularidad), en formato binario (ventajas y desventajas complementarias a las anteriores), y en formato binario comprimido (más lentos de cargar pero ocupan menos espacio en disco duro).

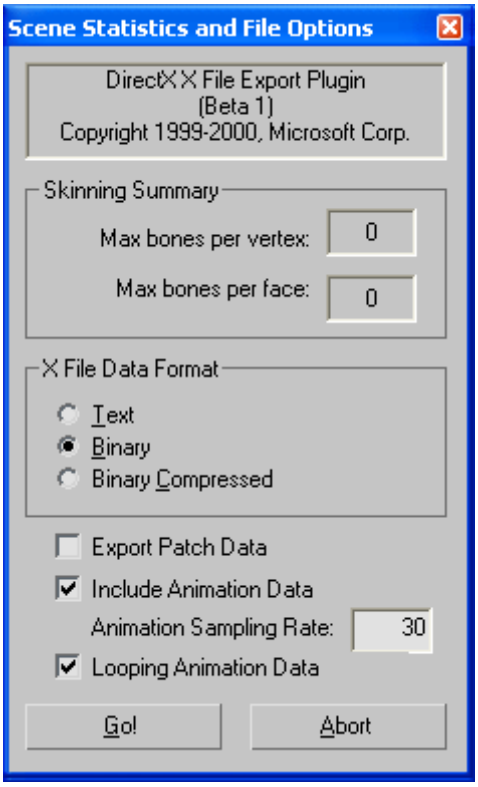
El conv3ds.exe puede generar a veces mallas corruptas. Por ejemplo, nuestro personaje aparecía con un brazo muy reducido, parecía manco. Además, los ficheros .3ds no pueden contener más que una postura de animaciones así que es imposible limitarse a este conversor para crear nuestro motor.

Los plug-ins para 3D Studio Max

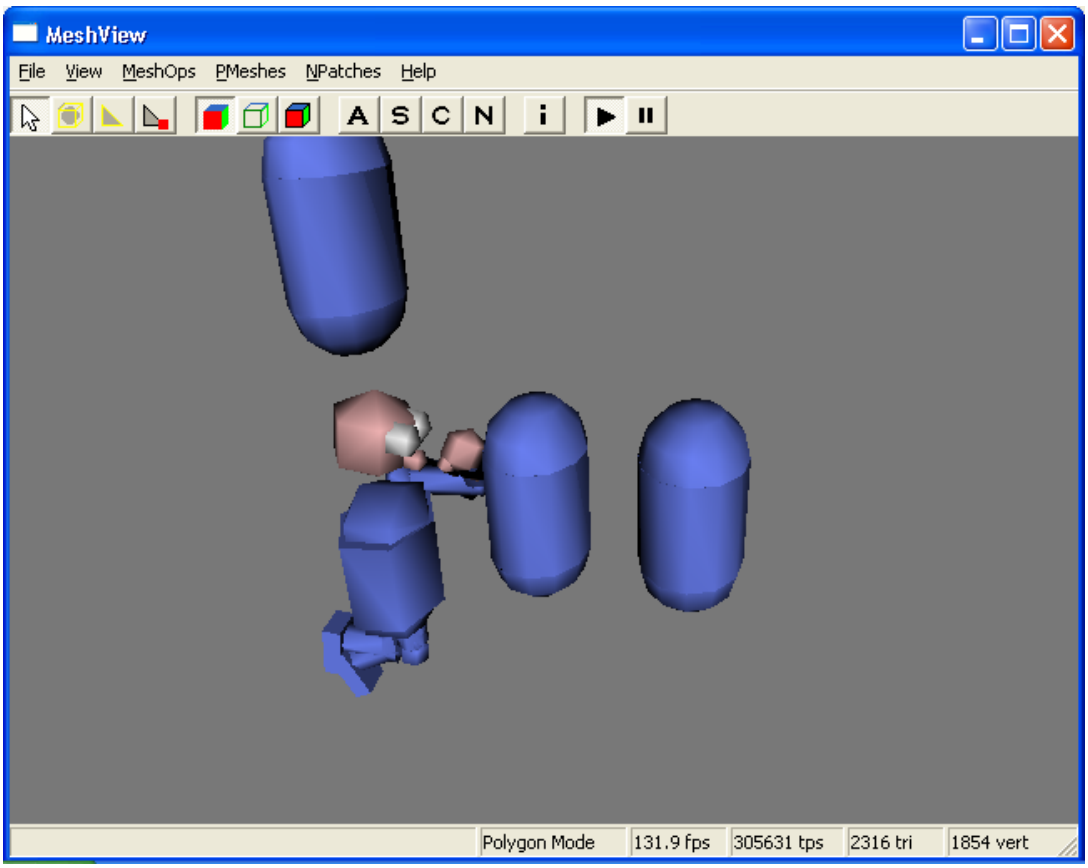
Ahora veamos el funcionamiento de los plug-ins que intentamos usar para exportar las animaciones del 3D Studio. Un plug-in es una

extensión de la herramienta de diseño, programada por unos terceros. Los plug-ins, una vez compilados, constituyen un fichero .dle (algo parecido a un dll), que hay que copiar en la subcarpeta plugins de 3D Studio Max. Los plugins que usamos nosotros tienen la funcionalidad de exportar mallas y animaciones a formato .x. Fueron el XSkinExp.dle, desarrollado por Microsoft e incluido (aunque sólo en código fuente) en el DirectX SDK; y el PandaDXExport5.dle, desarrollado por Pandasoft y distribuido ya en formato binario.

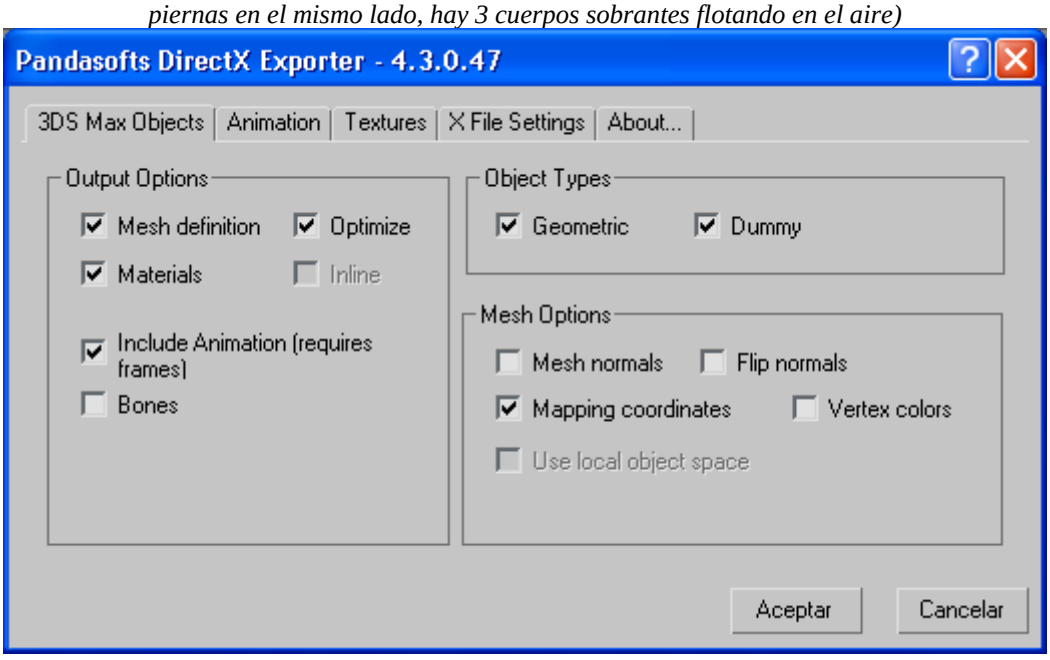
Una vez instalados los plugins, para usarlos hay que seleccionar dentro del 3D Studio Max la opción File -> Export, seleccionar alguno de los nuevos formatos existentes en la lista, e introducir los parámetros que se estimen necesarios en los diferentes cuadros de diálogo que salen. Como se dijo, ninguno de estos plugins sirvió para otra cosa más que para perder el tiempo, porque las animaciones se exportaban incorrectamente en todos los casos.



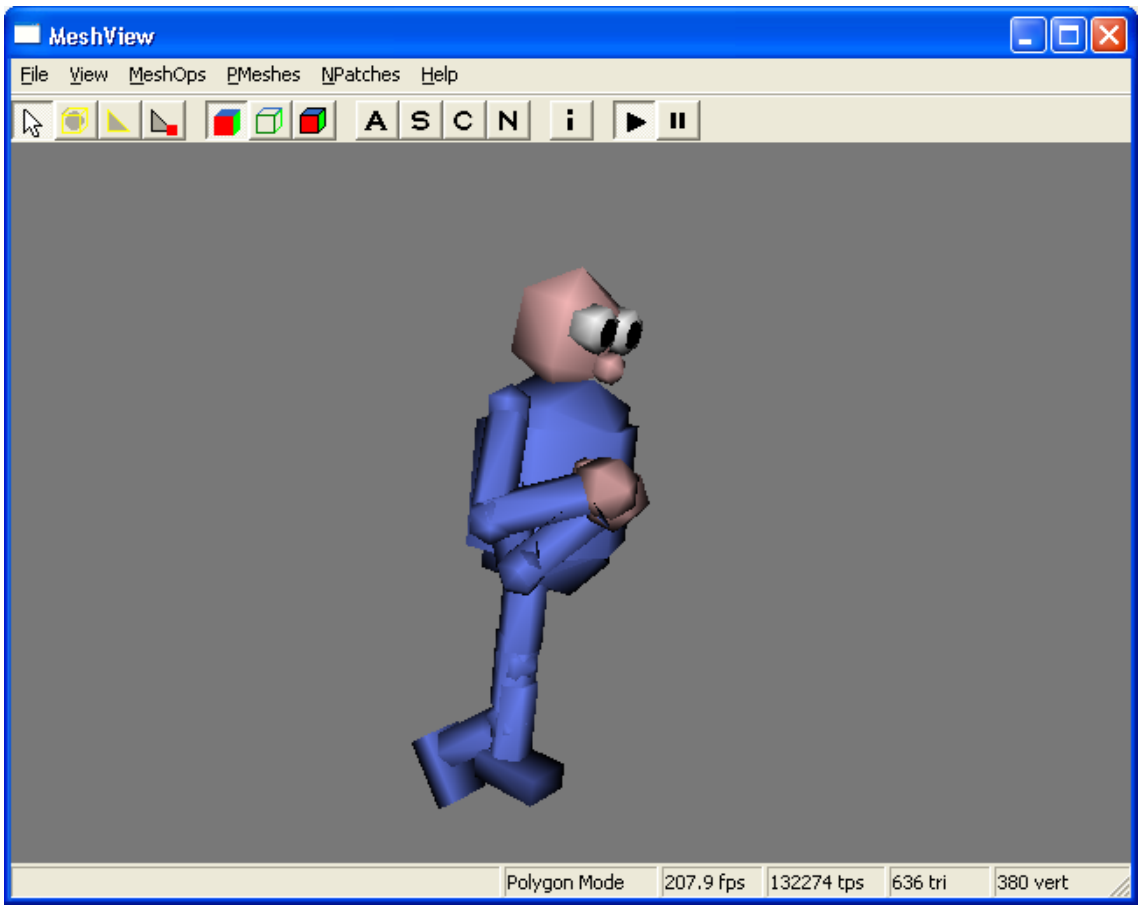
Plug-in XskinExp.dle



Animación resultante (totalmente incorrecta: la cabeza y los brazos salen separados del cuerpo, las dos piernas en el mismo lado, hay 3 cuerpos sobrantes flotando en el aire)



Plug-in PandaDXExport5.dle



Resultado de la animación (piernas y brazos de ambos lados coexistiendo en el mismo, unos se mueven y otros no)

Nuestro propio lector de ficheros .ase

Cumpliéndose el dicho de que si uno quiere que algo salga bien debe hacerlo él mismo, acabamos programando nuestra propia clase capaz de leer los ficheros con animación, en formato de texto .ase. El método de trabajo para introducir las animaciones en el juego consiste ahora en exportar los ficheros a formato ase desde 3D Studio Max (File -> Export -> seleccionar formato y configurar parámetro), y simplemente pasarle el nombre del fichero generado como parámetro al constructor de la clase que se encarga de leerlos, *MallaAnimada.*

Resultado de cargar nuestras 4 animaciones de prueba (caminar, caminar lateralmente, victoria y derrota), correctamente

6.- El Juego

Actualmente se ha popularizado un juego que se practica al aire libre. Solamente son necesarios unos trajes impermeables, una pistola de bolas de pintura y un grupo de gente dispuesto a acabar llenos de pintura hasta las cejas. El juego es conocido como Paintball y básicamente consiste en formar dos equipos. Cada jugador tiene una pistola de bolas de pintura y debe disparar a los miembros del equipo contrario. Aquel equipo que acabe con menos manchas de pintura se proclama vencedor del combate. La base del juego es muy sencilla, es el clásico juego de vaqueros e indios adaptado a la época moderna y con pistolas de verdad, aunque su munición no sean balas sino inofensivas bolas de pintura. Posiblemente la sencillez del juego y el hecho de que sea un juego al que todos hemos jugado alguna vez, aunque hayamos tenido que imaginar nuestras armas, ha hecho que este entretenimiento haya ganado muchos adeptos. Es una forma muy sana y barata de divertirse y, además, se dice que es una buena forma de descargar ese estrés, problema que hoy en día pesa sobre las cabezas de un número cada vez mayor de gente.

De los integrantes del grupo de desarrollo de este juego, solamente uno de ellos ha practicado este juego, pero su experiencia fue tan positiva que rápidamente surgió la idea de trasladar ese juego real al mundo virtual de un ordenador. A todos los demás la idea nos pareció original y divertida. Realmente, hasta ahora no se ha hecho ningún juego de esta clase, por lo menos no con esta ambientación. Juegos de combate entre equipos o de manera individual, en la que haya que abatir a un cierto número de enemigos, hay muchos, pero ninguno en el que el objetivo sea ensuciar al enemigo lo máximo posible.

Para todos aquellos que no hayan tenido la oportunidad de jugar al Paintball, le recomiendo que lo practique que seguramente se lleva un buen recuerdo. Pero mientras llega la ocasión, aquí está este juego para ir despertando la curiosidad. Para aquellos que ya hayan jugado al Paintball, seguramente prefieran el juego en la vida real, pero en este juego tendrán la oportunidad de desahogarse y liberar el estrés por unos momentos. Y para todos aquellos a los que les gusta los juegos tipo arcade, aquí tienen uno que es muy posible que sea más gracioso que muchos otros. En cualquier caso, nosotros hemos pasado un buen rato programando este juego, espero que aquellos que jueguen a él igualmente pasen un buen rato.

6.1.- Manual de usuario

1. Controles

El juego utiliza la interfaz de control habitual de los juegos 3D en 3ª persona, que combina teclado y ratón:

- **Flechas arriba / abajo / izqda. / dcha**.: desplazan al personaje hacia delante, detrás, y lateralmente hacia la izquierda y derecha, respectivamente.
- **Ratón**: se usa para girar la cámara alrededor del personaje
- **Botón izquierdo del ratón:** sirve para disparar en la dirección en que se esté mirando.
- **Botón derecho del ratón:** sirve para lanzar una bengala, que ilumina temporalmente el escenario.
- **Botón 0:** sirve para cambiar de arma, alternando entre la pistola y la marcadora.
- **Botones + y – (teclado numérico):** aumentan o disminuyen, respectivamente, la luz ambiental.

2. Objetivo

Dado que nuestro objetivo fundamental con la programación de este juego es demostrar el funcionamiento del motor para juegos que hemos desarrollado, la funcionalidad del juego se reduce a pasear por el escenario, sin más que hacer que observar el funcionamiento correcto de las colisiones, la iluminación puesta en marcha por las bengalas, etc.

7.- Bibliografía

Articulos

The Mechanics of Robust Stencil Shadows
Eric Lengyel
Gamasutra
October 11,2002
URL: http://www.gamasutra.com/features/20021011/lengyel_01.htm

Libros

3D Game Engine Design - A practical approach to real-time computer graphics"
Autor: David H. Eberly
Morgan Kaufmann Publishers
An imprint of academic press
1999

3d Games: Real-Time Rendering and Software Technology Volume One
Alan Watt & Fabio Policarpo
2001 ACM Pre

Enlaces web

http://symbolcraft.com/graphics/bsp/bsptreedemo_spanish.html

http://rt00149b.eresmas.net/Tecnicas/Bsp.PDF

http://info.pue.udlap.mx/~tesis/lis/cardona_a_jf/capitulo5.pdf

http://www.dcc.uchile.cl/~cc52b/Apuntes/Clase14-Ocultamiento.pdf

http://www.melax.com/bsp/melaxbsp.pdf

http://www.dccia.ua.es/dccia/inf/asignaturas/RG/temas/sesion15.pdf

Los autores de este proyecto: Pablo José Beltrán Ferruz, Alfonso del Cerro Aguilar, Daniel Cuenca Pascual y Gustavo García Aceituno, autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Madrid a 8 de julio de 2003

Firmado: