Sistemas Informáticos
Curso 2004-2005

Motor 3D con soporte de red aplicado a un videojuego

Roberto Blanco Ancos
José Antonio García Gimeno
Gustavo de Santos García

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

Facultad de Informática
Universidad Complutense de Madrid

Resumen

En este proyecto se desarrolla un juego de rol por ordenador, el cual tiene la posibilidad de ser utilizado en modo multijugador por medio de una red a la que se conecta el usuario. Cada cliente conectado a una partida en la red podrá explorar un escenario virtual común junto con el resto de participantes, además de interactuar y luchar contra ellos.

La implementación gráfica se basa en las librerías OpenGL. Los modelos de personajes, tanto humanos como generados por el servidor, han sido creados en *3D Studio MAX* y exportados con una herramienta que hemos programado.

Abstract

In this project we will be developing a computer *role playing game*, whose main feature is the possibility of being used in multiplayer mode through a net to which the user must be connected. Each client connected to the net will be able to explore a common virtual scenery together with the rest of participants, besides fighting and interacting with them.

The graphics implementation is based on the OpenGL libraries. Each character model (human as well as server-generated ones), has been created using 3D Studio MAX, and exported with a plug-in we have programmed.

Palabras clave

OpenGL, SDL, sockets, threads, 3DStudio, multijugador, geomipmapping, billboarding, sistema de partículas, lenguaje de script

Keywords

OpenGL, SDL, sockets, threads, 3DStudio, multiplayer, geomipmapping, billboarding, particle system, scripting

Índice

1 - Introducción

Entre la amplia gama de géneros de videojuego existentes en la actualidad, tanto para plataformas PC como en videoconsolas, los comúnmente conocidos como RPGs (*Role Playing Game*) se han ido abriendo un hueco entre los consumidores en los últimos años[MASS]. Aunque los RPGs son bien conocidos desde siempre, últimamente parecen haber vuelto a estar de moda. Esto ha provocado una nueva oleada de adelantos tecnológicos en este campo: argumentos complejos y no lineales, mejoras gráficas sustanciales… El realismo y la jugabilidad han aumentado considerablemente.

Es por ello que en este proyecto hemos querido explotar las nuevas posibilidades de este tipo de videojuegos. En concreto, hemos decidido desarrollar un RPG multijugador, en el cual se mezclarán los aspectos que definen los RPG clásicos con la ventaja de poder jugar en red. Siempre es más interesante saber que los enemigos a los que te enfrentas y los personajes que te ayudan en tu camino son en realidad seres humanos, y no simples algoritmos que repiten siempre lo mismo.

Cuando se intenta convertir una aplicación monousuario en algo equivalente pero con varios usuarios simultáneos, hay que tener en mente que no sólo consiste en hacer que las distintas instancias de los clientes se envíen mensajes entre sí, sino que habrá que modificar e incluso rediseñar la estructura interna del programa.

En este caso, al empezar el proyecto desde cero, lo que hacemos es diseñar el juego incluyendo la parte del multijugador como una parte más de la estructura del programa que vamos a crear. Aún así, siempre hay que cuidarse de procurar mantener la mayor independencia posible entre unas cosas y otras. Así, tendremos un módulo encargado de las comunicaciones en red claramente diferenciado del resto. Lo mismo ocurre con la parte gráfica, ya que todo juego actual que se precie necesita un motor gráfico.

Participar en un proyecto de este tipo nos ha permitido aprender mucho sobre el esfuerzo y la organización que se requieren para hacer un videojuego. Al principio pretendíamos introducir muchos requisitos para hacer que el juego fuera más complejo y entretenido. Pronto nos dimos cuenta de que, a pesar de la organización y el reparto de trabajo, muchos aspectos de la implementación que parecían triviales (como mostrar un menú de comandos o animar el movimiento de un modelo) costaban más trabajo del esperado. De hecho, ya pudimos comprobar en la asignatura de Informática Gráfica lo laborioso que puede resultar generar desde cero una escena con su iluminación, cámaras, objetos, materiales… Por eso decidimos no detenernos demasiado tiempo en aquello que no fuese realmente importante.

Este trabajo se estructura en los siguientes apartados (por orden de aparición):

- *Implementación*: Comentaremos las herramientas utilizadas durante el proceso de desarrollo, así como las tecnologías y requerimientos de la aplicación.

- *El sistema de juego*: Se describen todos los elementos que conforman el juego, desde los atributos de un personaje hasta las posibles acciones que podemos realizar. Además se explica la construcción de una escena y los distintos tipos de escena del juego.

- *Representación de objetos en el mundo 3D*: Todos los personajes, objetos, edificios, etc. que podemos ver en una escena, se dibuja a partir de un modelo. En este apartado veremos cómo está hecho un modelo, cómo animarlo y cómo se calculan las colisiones entre varios modelos.

- *Implementación de exteriores*: El terreno sobre el que descansan los modelos no es un simple polígono fijo. Aquí se veremos en qué consiste el Geomipmapping y cómo generar terrenos predefinidos a través de archivos de imagen. Además, explicamos cómo se crea la iluminación de escena y se recorta la parte de la escena que no entra en la cámara para aumentar el rendimiento con *Frustum culling*. Se dedicará un apartado final al Skybox y los efectos de partículas.

- *Editor de escenas*: Gracias al lenguaje de script reconocible por el juego, podemos crear escenas fácilmente con un editor y cargarlas en el juego sin tener que hacer compilaciones intermedias.

- *Exportador de modelos*: Es muy difícil crear un modelo "a mano". Podemos servirnos de herramientas de modelado como 3D Studio y generar los modelos desde una interfaz gráfica. El exportador de modelos nos permitirá coger los modelos de 3D Studio para poder introducirlos en el juego.

- *Multijugador*: En una partida multijugador, los clientes tienen que conocer el estado de los demás jugadores en todo momento. Así mismo, el servidor debe tener constancia de las acciones de cada cliente para poder procesarlas, sincronizarlas y reenviarlas a otros clientes. El módulo multijugador implementa un sistema de envío de mensajes para poder realizar esta tarea.

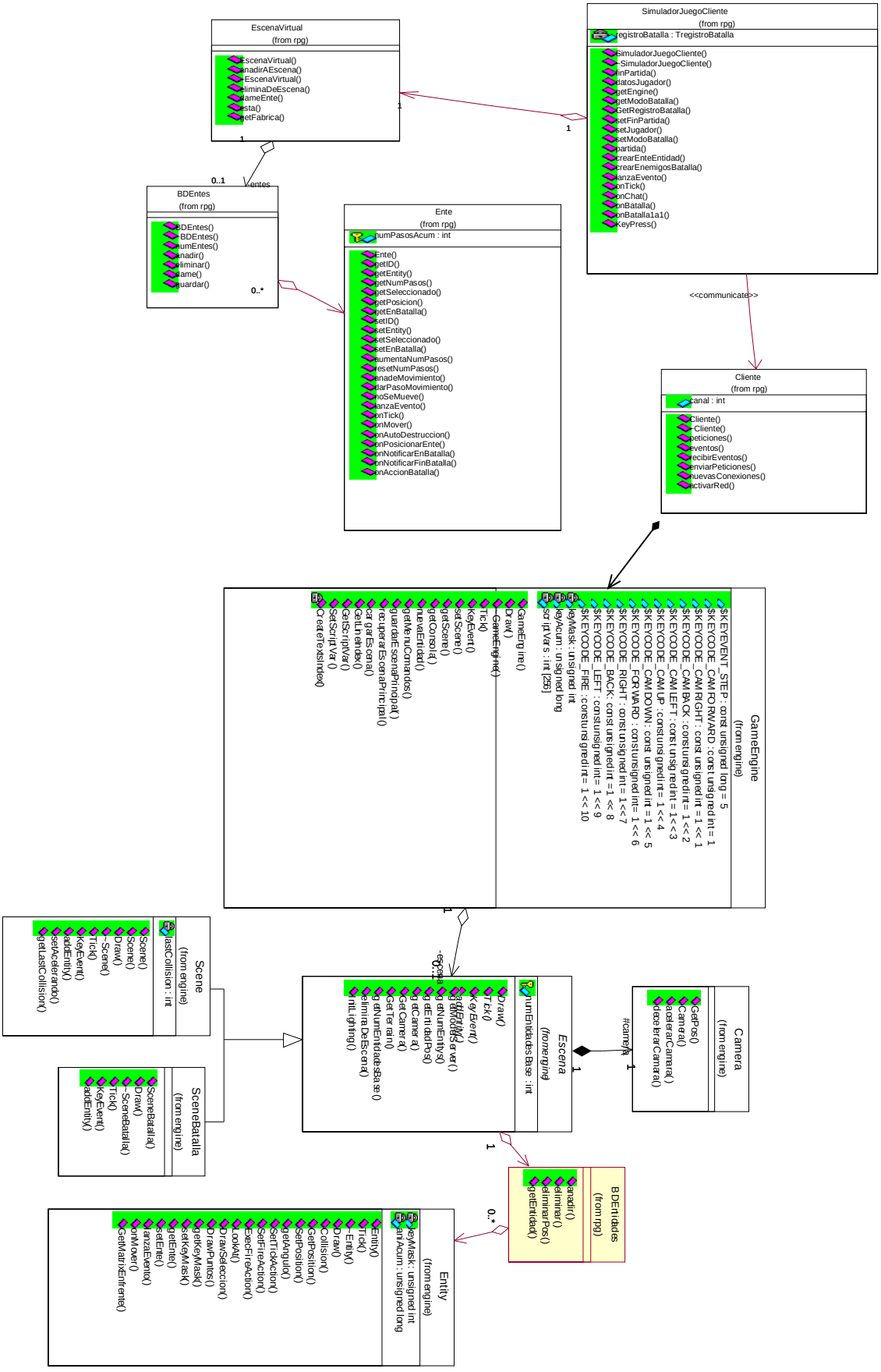
2 – Implementación

Para la implementación del juego nos hemos apoyado en OpenGL, ya que son unas librerías muy completas, de libre distribución y utilizadas en multitud de proyectos, especialmente en videojuegos.

Al utilizar OpenGL, tuvimos que implementar todo el proyecto en lenguaje C. Concretamente, hemos usado el entorno de Microsoft Visual C++. Al ser un proyecto de cierta envergadura, nos hemos valido de la programación orientada a objetos (C++) para poder hacer un diseño más intuitivo y fácil de manejar.

Haciendo referencia al diseño, en el gráfico de la página siguiente se muestran los elementos más importantes de la estructura interna del juego en un cliente. Por una parte, tenemos el "simulador de juego" (la clase *SimuladorJuegoCliente*), encargada de gestionar y comunicar entre sí los distintos módulos que componen el juego. Así, el simulador contiene una referencia al motor gráfico (*GameEngine*), y también una referencia a la *EscenaVirtual*, donde se guardan los datos del juego. Así, cuando se produzca algún cambio en la escena virtual, el simulador de juego podrá notificar al motor gráfico para que éste muestre algo por la pantalla.

Así es como se comunican el motor y la implementación interna del juego, pero… ¿y el multijugador? En este caso, el simulador de juego contiene referencias a las colas de eventos y peticiones que el gestor de red (clase *Cliente*) enviará a través de la red. Como explicaremos más adelante, el contenido de estas colas son los mensajes que se envían entre sí clientes y servidor para comunicarse. De este modo, al producirse una acción en la lógica interna del juego, el simulador generará un nuevo mensaje de red y lo introducirá en la cola correspondiente, para que sea enviado a su destinatario.


EscenaVirtual
(from rpg)
EscenaVirtual()
anadirAEscena()
~EscenaVirtual()
eliminaDeEscena()
ameEnte()
esta()
etFabrica()
SimuladorJuegoCliente
(from rpg)
egistroBatalla : TregistroBatalla
SimuladorJuegoCliente()
~SimuladorJuegoCliente()
inPartida()
atosJugador()
getEngine()
getModoBatalla()
GetRegistroBatalla()
setFinPartida()
setJugador()
setModoBatalla()
partida()
crearEnteEntidad()
crearEnemigosBatalla()
anzaEvento()
onTick()
onChat()
onBatalla()
onBatalla1a1()
KeyPress()
1
1
1
0..1
entes
BDEntes
(from rpg)
BDEntes()
~BDEntes()
numEntes()
anadir()
eliminar()
ame()
guardar()
0..*
Ente
(from rpg)
numPasosAcum : int
Ente()
getID()
getEntity()
getNumPasos()
getSeleccionado()
getPosicion()
getEnBatalla()
setID()
setEntity()
setSeleccionado()
setEnBatalla()
aumentaNumPasos()
esetNumPasos()
anadeMovimiento()
darPasoMovimiento()
noSeMueve()
anzaEvento()
onTick()
onMover()
onAutoDestruccion()
onPosicionarEnte()
onNotificarEnBatalla()
onNotificarFinBatalla()
onAccionBatalla()
<<communicate>>
Cliente
(from rpg)
canal : int
Cliente()
~Cliente()
peticiones()
eventos()
recibirEventos()
enviarPeticiones()
nuevasConexiones()
activarRed()
GameEngine
(from engine)
SKEVENT_STEP : const unsigned long = 5
SKEYCODE_CAM FORWARD : const unsigned int = 1
SKEYCODE_CAM RIGHT : const unsigned int = 1 << 1
SKEYCODE_CAM BACK : const unsigned int = 1 << 2
SKEYCODE_CAM LEFT : const unsigned int = 1 << 3
SKEYCODE_CAM UP : const unsigned int = 1 << 4
SKEYCODE_CAM DOWN : const unsigned int = 1 << 5
SKEYCODE_FORWARD : const unsigned int = 1 << 6
SKEYCODE_RIGHT : const unsigned int = 1 << 7
SKEYCODE_BACK : const unsigned int = 1 << 8
SKEYCODE_LEFT : const unsigned int = 1 << 9
SKEYCODE_FIRE : const unsigned int = 1 << 10
script Vars : int [255]
keyAcum : unsigned long
keyMask : unsigned int
GameEngine()
~GameEngine()
Draw()
Tick()
KeyEvent()
setScene()
getScene()
getConsola()
nuevaEntidad()
getMenuComandos()
guardarEscenaPrincipal()
recuperarEscenaPrincipal()
cargarEscenal()
GetLinIndex()
GetScriptVar()
SetScriptVar()
CreateTextoIndex()
1
1
Escena
(from engine)
numEntidadesBase : int
Draw()
Tick()
KeyEvent()
GetTerrain()
GetCamera()
getNumEntidadesBase()
eliminaDeEscena()
ntLighting()
1
1
#camera
Camera
(from engine)
GetPos()
Camera()
~Camera()
aceleraCamara()
doceleraCamara()
Scene
(from engine)
lastCollision : int
Scene()
~Scene()
Draw()
Tick()
KeyEvent()
addEntity()
setAceleRando()
getLastCollision()
SceneBatalla
(from engine)
SceneBatalla()
Draw()
Tick()
KeyEvent()
addEntity()
1
BDEntidades
(from rpg)
anadir()
eliminar()
eliminarPos()
getEntidad()
0..*
Entity
(from engine)
keyMask : unsigned int
keyAcum : unsigned long
Entity()
~Entity()
Draw()
Tick()
GetCollision()
GetPosition()
SetPosition()
getAngulo()
SetTickAction()
SetFireAction()
ExecFireAction()
LookAt()
DrawSeleccion()
DrawPuntos()
getKeyMask()
setKeyMask()
getEnte()
setEnte()
anzaEvento()
onMover()
GetMatrixEnte()

2 - El sistema de juego

Las reglas que rigen este juego no son distintas a las de cualquier R.P.G. actual. Nuestra misión, como jugadores, es la de sobrevivir a los retos que se nos puedan presentar, y hacer evolucionar a nuestro personaje para que este se haga más fuerte y disponga de más recursos para vencer en los combates. Cuantas más victorias cosechemos, más fácil será asegurar nuestra permanencia en la partida.

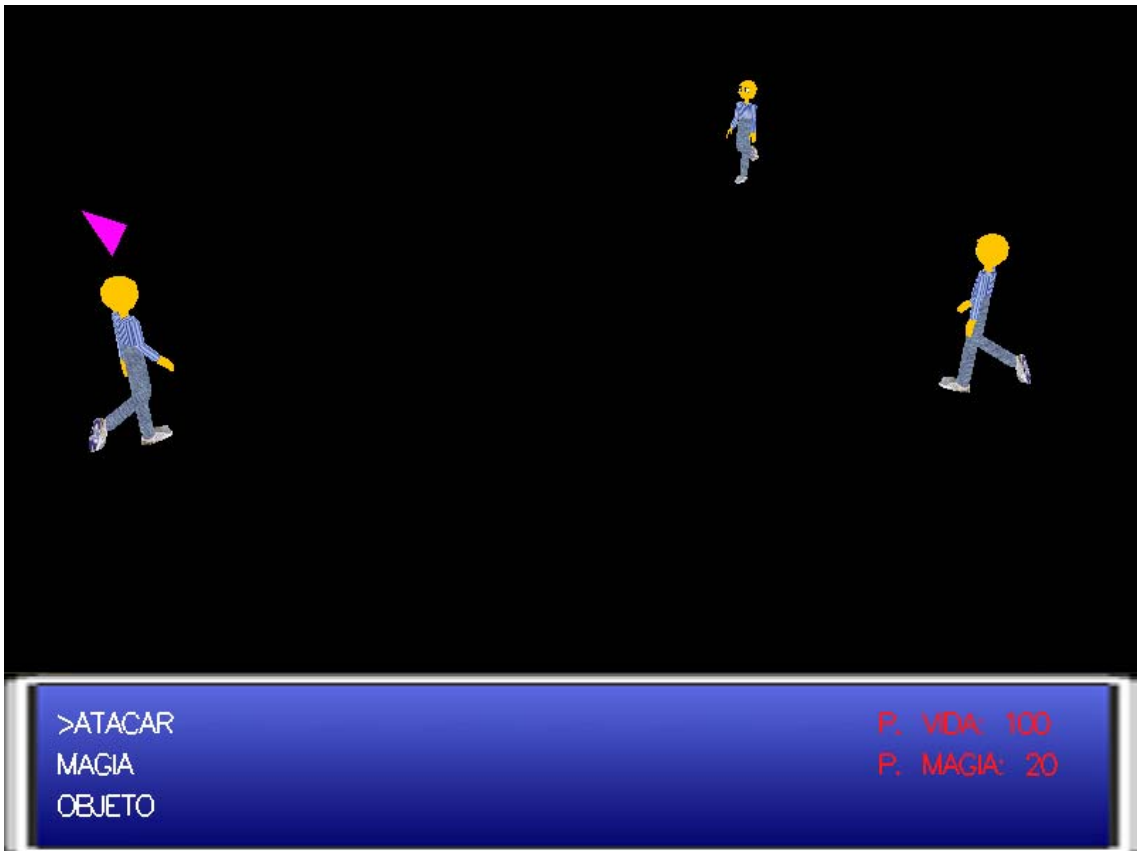
2.1 Escenarios

El juego se desarrollará siempre en alguno de los dos siguiente escenarios:

- **La escena del mundo:** Es el entorno en el que el personaje se puede mover libremente y comunicarse con el resto de jugadores y personajes artificiales. Aquí nos dedicamos básicamente a recorrer (caminando) la escena buscando enemigos, personajes neutrales que nos den alguna misión, o simplemente algún jugador con quien hablar.

Ejemplo de imagen de la escena del mundo exterior

- La escena de batalla: Es un espacio bastante más reducido que el de la escena del mundo, y donde no tenemos la misma libertad de movimientos. Es donde tienen lugar los combates. Los participantes de un combate se colocan formando un círculo alrededor del centro de la escena. No pueden actuar todos al mismo tiempo, sino que la acción se desarrolla por turnos. Cuando es el turno de un jugador, el resto de jugadores o personajes artificiales deben esperar. El jugador en posesión del turno realizará una acción, y entonces el turno pasará al personaje que corresponda. El combate que estamos jugando termina cuando conseguimos eliminar a todos nuestros contrincantes o cuando alguno de ellos nos aniquila.



La flecha sobre nuestro personaje indica que ha llegado nuestro turno

2.2 Elementos del juego

2.2.1 Atributos

Cada personaje involucrado en una batalla dispone de una serie de atributos. Cada vez que efectuamos una acción en una batalla, o que alguien la efectúa sobre nosotros, uno o varios de estos atributos cambian como consecuencia de esa acción. El objetivo está en optimizar nuestros atributos para que nuestro personaje sea mejor que los demás. Estos son los atributos de un personaje:

- Puntos de vida (PV): Se representa con un número entero comprendido entre el cero y un valor máximo (al igual que la mayoría de los atributos). Es el dato más importante a tener en cuenta, ya que cuando nuestros PV lleguen a 0, significará que nuestro personaje ha sido derrotado. Así, podríamos decir que el objetivo de la partida consiste en dejar a todos los enemigos sin puntos de vida.

- Puntos de magia (PM): Si queremos atacar a un enemigo lanzándole un hechizo, tenemos que consumir parte de nuestro poder mágico. Este poder está representado por los puntos de magia (PM). Cada hechizo que hagamos consumirá 5 PM, con lo cual no podremos hacer un hechizo cuando nuestro nivel de PM sea demasiado bajo.

- Puntos de vida y de magia máximos (PVmax y PMmax): Un personaje no puede tener todos los puntos de vida o de magia que desee. Siempre se cumple que "PV<=PVmax" y "PM<=PMmax".

- Fuerza: Los ataques y magias tienen distinta potencia dependiendo de quién y en qué momento los realice. Si un personaje tiene más nivel (es decir, más experiencia en combate), es lógico que sus ataques sean más poderosos, y el atributo 'Fuerza' nos da una idea de este poder.

- Defensa: Al igual que ocurre con la Fuerza, un personaje va aumentando durante el juego su capacidad de resistir los ataques. A mayor defensa, menor será el daño recibido.

- Nivel y Experiencia: Si conseguimos salir victoriosos de toda situación peligrosa y mantenernos en el juego mucho tiempo, el personaje que manejamos evolucionará con cada batalla ganada, aumentando sus puntos de Experiencia. Los puntos de Experiencia se van acumulando, y cuando llegan a un cierto valor, nuestro personaje sube de Nivel. Esto quiere decir que se hace más poderoso, con lo que a otros jugadores les será más difícil vencerle.

- Estado: Hay métodos más sutiles para inclinar la balanza a nuestro favor en una lucha que a priori pueda parecer imposible de ganar. Algunos objetos pueden ser utilizados contra nuestros enemigos para causarles lo que denominaremos "Estados Alterados". Un estado alterado hace que cambie el comportamiento y los atributos de nuestro personaje sin que nosotros podamos controlarlo. A continuación se detallan los posibles estados alterados que puede tener un personaje del juego:

· Normal: No le ocurre nada extraño al personaje.

· Dormido: Nos quedamos dormidos. Cuando llega nuestro turno, éste salta hasta el siguiente jugador porque nosotros no podemos hacer nada. El personaje seguirá dormido hasta que alguien le cure o hasta que reciba un ataque físico (los ataques mágicos no le despertarán).

· Mudo: Los hechizos suelen requerir de un conjuro previo. Cuando nos encontramos en el estado Mudo, nos veremos incapacitados para hacer magias.

· Ciego: Nuestra visión se vuelve borrosa. Esto no afecta a los hechizos, ya que no necesitamos apuntar para efectuarlos, pero los ataques físicos fallarán el blanco, haciéndose inservibles.

· Berserk: Es un estado de furia incontrolable. Perderemos la capacidad de controlar a nuestro personaje, y este atacará indiscriminadamente a cualquier personaje. La ventaja es que los ataques tendrán mucha más fuerza.

· Zombi: Al ser un muerto viviente, estamos más vinculados con la muerte que con los vivos. Cualquier poción de vida que tomemos nos quitará puntos de vida (PV) en vez de restaurarlos.

· Veneno: Cuando un personaje está envenenado, pierde algo de vida en cada comienzo de su turno. La vulnerabilidad al veneno que nos está atacando depende directamente de nuestro atributo de Defensa.

Menú de comandos de nuestro personaje en el modo batalla.

2.2.2 Acciones

Cada vez que un jugador obtiene el turno de la partida, tiene derecho a realizar una sola acción, teniendo que esperar a que vuelva a ser su turno para poder volver a actuar. Las acciones están orientadas principalmente a causar algún daño a los enemigos, y también a defender nuestros puntos de vida (PV) para no quedar fuera de combate.

Las posibles acciones son las siguientes:

- Atacar: Es la acción más básica y sencilla. Seleccionando 'Atacar' en el menú de comandos de la pantalla, sólo tenemos que elegir un personaje enemigo y nuestro personaje se acercará a él y le atacará. Entonces aparecerá una cifra sobre el enemigo que nos informa de los PV que este ha perdido.

- Magia: Una magia es más potente que un ataque sencillo, pero al efectuarla consumimos puntos de magia (PM). Podemos elegir entre tres tipos de magia (Fuego, Hielo y Rayo). Después de seleccionar al enemigo, se conjurará la magia sobre él y, al igual que con los ataques, podremos ver cuántos PV ha perdido.

- Objeto: Todos los personajes, tanto los que son controlados por un jugador como los que son controlados por inteligencia artificial, tienen un inventario. En este inventario hay objetos que pueden ser utilizados para hacer daño o para curar. La acción "Objeto" consiste en seleccionar un objeto del inventario y a un personaje sobre el cual utilizar ese objeto. La siguiente es una lista de los tipos de objeto que podemos llevar:

  · *Poción:* El personaje que se tome una poción aumentará en 50 sus puntos de vida (PV). Hay que tener en cuenta que el nuevo valor de PV nunca podrá ser superior a 'PVmax'.
  · *Eter:* Aumenta los puntos de magia (PM) en 15 puntos.
  · *Tila:* Causa a quien lo toma el estado alterado "Dormido".
  · *Esparadrapo:* Causa el estado alterado "Mudo".
  · *Tinta china:* Causa el estado alterado "Ciego".
  · *Salsa Tabasco:* Causa el estado alterado "Berserk".
  · *Calavera vudú:* Causa el estado alterado "Zombi".
  · *Veneno:* Causa el estado alterado "Veneno".
  · *Café:* Cura el estado alterado"Dormido". Si no estabas dormido antes de tomarlo, no tiene ningún efecto.
  · *Bocina:* Cura el estado alterado "Mudo".
  · *Lentillas:* Cura el estado alterado "Ciego".
  · *Té de valeriana:* Cura el estado alterado "Berserk".
  · *Agua bendita:* Cura el estado alterado "Zombi".
  · *Antídoto:* Cura el estado alterado "Veneno".

2.2.3 El menú de comandos

En la parte inferior de la pantalla de una escena de combate, podemos ver una consola azul con la información necesaria para manejar a nuestro personaje y decidir la mejor estrategia que nos lleve a la victoria. Este es el menú de comandos:

El menu de comandos nos permite componer una acción y enviársela a un personaje. Cuando llega nuestro turno, podemos ver en la parte de la izquierda una lista con los tipos de acciones que podemos realizar. Si seleccionamos "Atacar", podremos seleccionar con los botones de dirección de movimiento *'adelante' y 'atrás'* al enemigo que recibirá el ataque. Seleccionando "Magia", se muestra en el menú de comandos una lista de tipos de magias, y después de elegir una seleccionamos al personaje de destino del mismo modo que antes. Finalmente, con "Objeto" nos aparecerá una lista de tipos de objetos, en la cual podremos navegar con los botones 'adelante' y 'atrás' (igual que cuando seleccionamos un personaje). Si el objeto que se está mostrando en el menú aparece de color gris, significa que no disponemos de ningún objeto de ese tipo en el inventario. Si aparece de color blanco, sí que podemos utilizarlo.

En la parte derecha del menú de comandos se visualizan los puntos de vida (PV) y los puntos de magia (PM) que tenemos actualmente. Así sabremos cuándo es conveniente tomarse una poción o un éter en vez de atacar (recordemos que sólo se puede realizar una acción por turno, y el orden que elijamos para nuestras acciones puede ser decisivo en la victoria final). Si además estamos sufriendo algún estado alterado, el menú de comandos también nos informará de ello, debajo de los marcadores de PV y PM.

2.3 Modos de lucha

Existen dos formas de enzarzarse en un combate:

- Viajando por el mundo exterior: Cada vez que hayamos recorrido una cierta distancia, entraremos automáticamente en un combate. Este combate se generará enteramente en el equipo cliente, ya que los enemigos serán todos artificiales. El servidor solamente será informado de la situación al comienzo y al final de la batalla.

Este tipo de combates es el más sencillo, ya que los enemigos no tienen demasiada fuerza ni PV. Sin embargo no hay que descuidarse, estos enemigos valoran sus posibilidades en cada turno y eligen la acción más conveniente en función de su estado actual y de la observacion que hacen de nuestros movimientos durante la batalla. A este fín se crea en cada batalla un registro de batalla, con los siguientes campos:

```
//Registro de datos para moldear el comportamiento de los jugadores no humanos
typedef struct TregistroBatalla {
        int numTurnosJugados;           //Numero de turnos que el jugador ha hecho
        int potenciaMediaAtaques;       //Cantidad de puntos aproximada que quitan sus ataques
        int puntosVidaPerdidos; //Puntos de vida que lleva perdidos el jugador en la batalla
        int objetosCurativosUsados;     //Cantidad de objetos beneficiosos que ha utilizado
        int defensaEstimada;            //Valor estimado del atributo de defensa del jugador
        int estadoActual;               //Estado alterado en el que se encuentra el jugador
};
```

- Batalla 1 contra 1: En el mundo exterior, podemos acercarnos a otro jugador humano. Si nos pegamos a él y pulsamos el botón de *'lucha'* (tecla Z), se generará un nuevo escenario de combate, en el que sólo estaremos presentes él y nosotros. En este caso el servidor recibe todas las acciones producidas durante la partida, para así poder sincronizar la escena de lucha en los dos equipos cliente participantes. Este tipo de lucha es en general más encarnizado, debido principalmente a que desconocemos totalmente la fuerza de nuestro contrincante. El jugador que quede K.O. (con PV=0) acabará su partida, mientras que el ganador podrá continuar jugando y será transportado de nuevo al mundo exterior.

2.4 Fin del combate

Como ya hemos mencionado, un combate termina cuando acabamos con todos o cuando acaban con nosotros. Si perdemos, se acaba el juego y tendríamos que empezar de cero. Si conseguimos ganar, se mostrarán en el menú de comandos los siguientes resultados:

- *PV y PM restantes:* Serán los puntos con los que comenzaremos la siguiente batalla.
- *Estado alterado:* Si tenemos alguno, desaparecerá cuando volvamos al mundo exterior.
- *Nivel:* Muestra el nivel actual.
- *Experiencia:* Muestra los puntos de experiencia que tenemos acumulados después del combate y los puntos que nos faltan para pasar al siguiente Nivel.
- *Premio:* En ocasiones, después de ganar el combate recibiremos un objeto como premio. Puede ser cualquier tipo de objeto (poción, antídoto, tila…).

2.5 Manual de manejo

Ahora que ya conocemos las reglas que rigen este juego y las posibilidades que se nos ofrecen, aprenderemos a utilizar a nuestro personaje en el mundo virtual que hemos creado. El manejo es sencillo, utilizaremos el teclado de la siguiente forma:

- Teclas de movimiento: Para movernos por el escenario, y también para navegar por los menús. Las teclas asignadas al movimiento son I, J, K y L.

- Chat: En cualquier momento podemos conversar con otro jugador, siempre que podamos visualizarle en el mundo virtual. Solo tenemos que caminar hasta colocarnos cerca de él, y pulsar la tecla TAB. En la parte superior de la pantalla aparecerá una consola de chat, y podremos escribir cualquier cadena de caracteres y enviársela a nuestro compañero pulsando ENTER. Para salir del modo de consola de chat, sólo tenemos que volver a pulsar TAB.
**NOTA: En las batalla 1 contra 1, también es posible utilizar el chat para hablar con nuestro contrincante. El manejo es el mismo que en el mundo exterior.*

- Salir: Para salir en cualquier momento de la partida, se pulsará la tecla ESC. El programa cliente enviará un último mensaje de desconexión al servidor y entonces se cerrará.

- Batalla 1 contra 1: Para empezar a pelear con otro personaje humano, debemos acercarnos a él hasta que estemos casi pegados, y seguidamente pulsar la tecla Z. Entonces comenzará la batalla y se cambiará la escena en los dos clientes implicados.

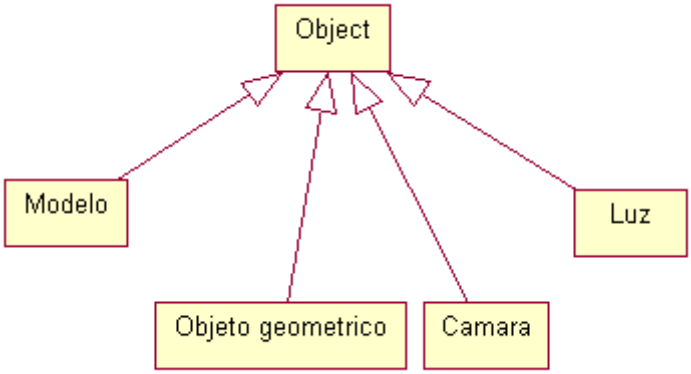
- Personajes neutros: Para hablar con un personaje neutral (no humano) de la escena del mundo exterior, tenemos que pegarnos a él y pulsar la tecla M.

- Menú de comandos: Durante nuestro turno en la batalla, podemos navegar por las opciones del menú de comandos con las teclas de movimientos, y seleccionar la opción que deseemos pulsando ENTER. Si nos equivocamos al entrar en un submenú y queremos rectificar, hay que pulsar la tecla RETROCESO.

3 – Representación de objetos en el mundo 3D

3.1 ¿Qué es un objeto?

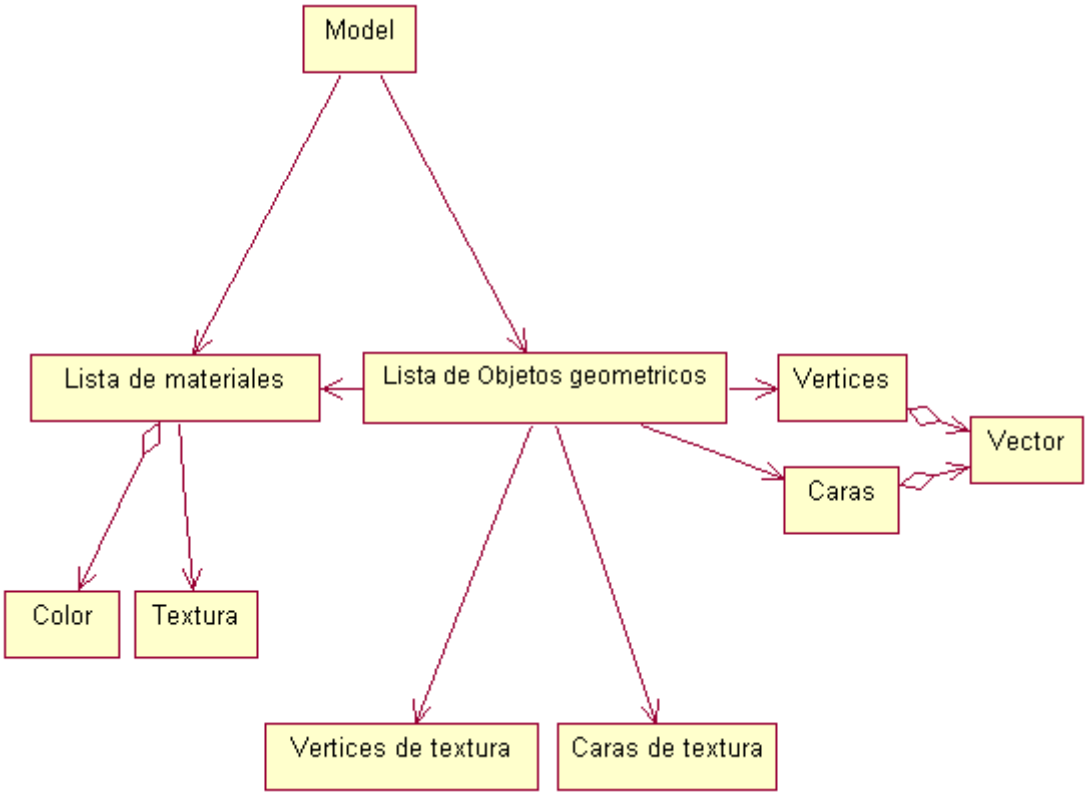
En este apartado se va a explicar qué representación interna se ha utilizado para los objetos en nuestro mundo en tres dimensiones. Básicamente para nosotros un objeto es cualquier "cosa" que tiene una posición y una orientación en el mundo (con excepción del terreno, para el cual hay un capítulo específico en esta memoria). Para este cometido, la representación de nuestro objeto contiene una matriz. Dentro de esta calificación de objeto, tenemos la cámara, las luces, los objetos con representación geométrica y los modelos. Un modelo, como se verá en el capítulo siguiente, es una composición de objetos geométricos.



3.2 Los modelos

Como hemos visto, los modelos son un tipo de objetos. Son los más complejos ya que son las estructuras que se van a encargar de llevar toda la información asociada a la representación del modelo: geometría, materiales, texturas , animación, etc.

En el diagrama siguiente se puede ver resumidamente cómo ésta compuesto un modelo. Contiene una lista de materiales, que mantienen propiedades sobre el color así como una textura opcional. Contiene también una lista de objetos geométricos (por ejemplo el modelo "hombre" podría estar compuesto de los objetos geométricos cabeza, tronco, brazos y piernas).



3.2.1 Los materiales

Los materiales están formados por tres parámetros: dos obligatorios y uno opcional. Los obligatorios son la reflexión de color difusa y ambiental, dadas en componentes RGB. El parámetro opcional es la textura. El formato gráfico elegido para cargar gráficos en nuestro motor ha sido el .TGA, dada su gran versatilidad, ya que a diferencia de otros formatos admite además de los canales básicos RGB, un canal de alpha para transparencias, y también admite compresión sin pérdida de calidad con el algoritmo RLE (Run Length Encoding). Se ha desarrollado un lector de este formato para el proyecto.

3.2.2 Los objetos geométricos

El modelo está compuesto por varios objetos geométricos, que definen su representación en el mundo. Como hemos visto en los diagramas anteriores, tanto el modelo como el objeto geométrico heredan de Objeto, gracias a lo cual tienen matrices independientes relacionadas de manera jerárquica, es decir cualquier variación que sufra el modelo en cuanto a su posición u orientación afectará a todos los objetos geométricos que contiene.

El objeto geométrico almacena toda la información relacionada con sus vértices y caras. Tanto los vértices como las caras poseen vectores normales utilizados para la iluminación.

También contiene un puntero a un material de la lista de materiales que contiene el modelo, que se utiliza para pintar el objeto. Si el material tiene textura, se deberán usar también los vértices de textura y los índices a las caras de textura.

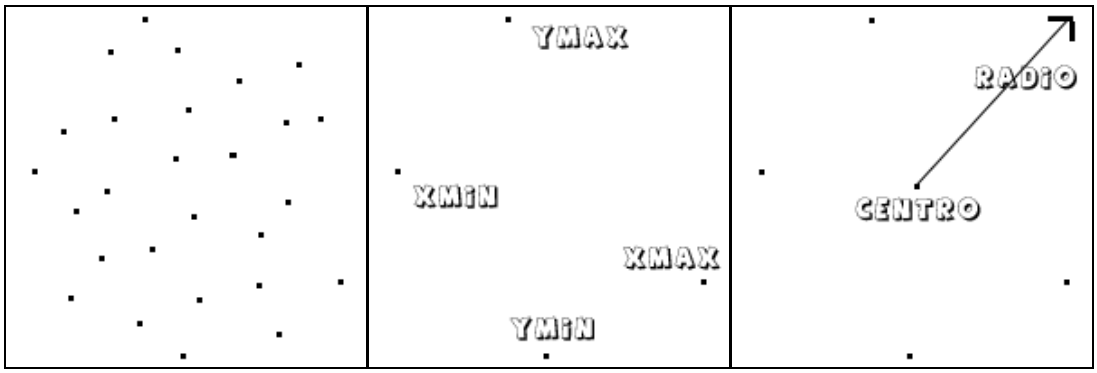
3.2.3 Detección de colisiones entre modelos

En cualquier motor 3D orientado a videojuegos, debe aparecer algún tipo de soporte de detección de colisiones [EBE01]. Aquí hemos utilizado una detección de colisiones de dos etapas. En la primera etapa se hace una detección simplificada que requiere poco cálculo para descartar la mayoría de las colisiones. Si alguna colisión cumple los requisitos de la primera etapa, pasa a una segunda etapa más precisa para comprobar si realmente se ha producido una colisión.

3.2.3.1 Primera etapa de la detección de colisiones

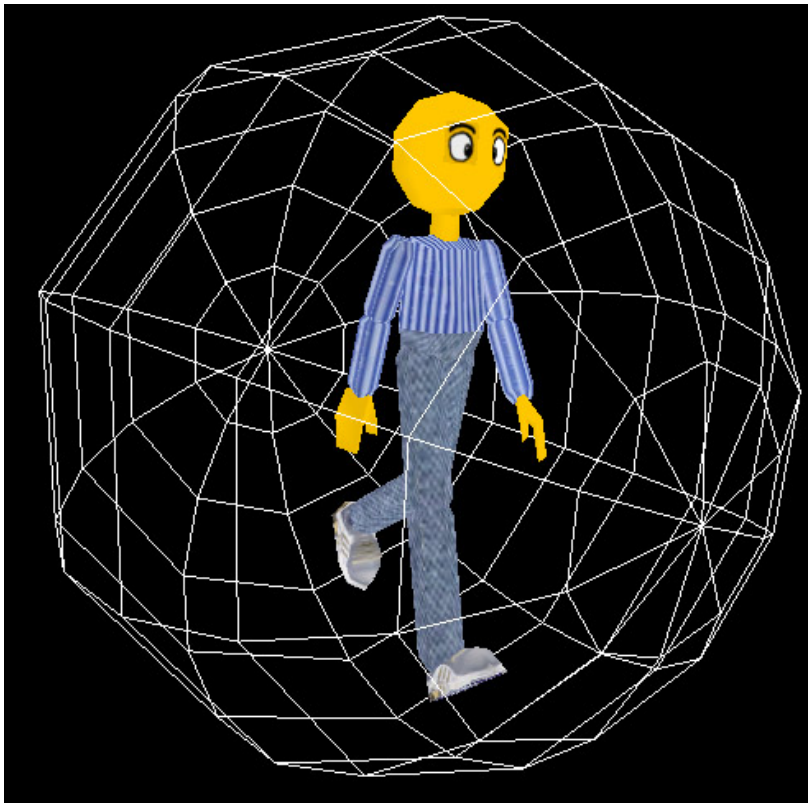
Esta primera etapa tiene como objetivo desechar la mayoría de las colisiones, con lo cual no se busca precisión si no rapidez de cálculo. Con este objetivo, la detección se va a realizar a nivel del modelo, y no a nivel de cada uno de sus componentes geométricas. Para ello, todo modelo tiene asociado a sí mismo una esfera que lo envuelve por completo, definida mediante un centro y un radio. El procedimiento para calcular esta esfera envolvente es el siguiente:

- Ya que la esfera va a contener a todo el modelo, se van a tratar todos los vértices de los objetos geométricos que tiene el modelo como si fueran un solo objeto.
- Se va a recorrer todo este conjunto de vértices, y se van a hallar los valores máximos y mínimos para cada una de las tres coordenadas. Con lo cual, al acabar esta pasada del algoritmo tendremos 6 valores: Xmin, Xmax, Ymin, Ymax, Zmin y Zmax.
- El centro de la esfera se va a colocar en el punto medio de cada una de estas coordenadas, es decir:
    - Centro.X = (XMin + XMax) / 2
    - Centro.Y = (YMin + YMax) / 2
    - Centro.Z = (ZMin + ZMax) / 2
- Una vez hallado el centro, nos queda calcular el segundo parámetro, el radio de la esfera. Para asegurarse de que la esfera envuelve a todos los vértices, el radio será la distancia del centro al punto (XMax, YMax, ZMax).
- A continuación se pone un ejemplo gráfico del proceso en dos dimensiones:



Paso 1 Paso 2 Paso 3

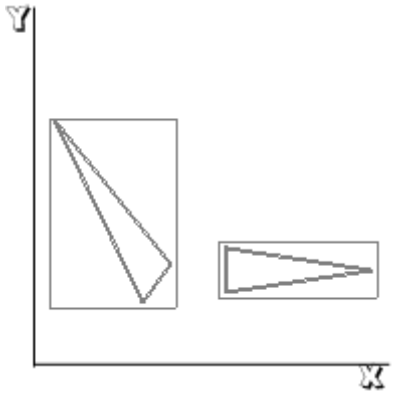
Para comprobar si hay colisión entre dos modelos, basta con comprobar si sus esferas envolventes asociadas intersecan. Para hacer esta comprobación se calcula la distancia entre los centros de las esferas y se comprueba si es menor que la suma de los radios. Si se pasa este primer test, se ejecutará la segunda parte del algoritmo para ver si ha habido una colisión real.



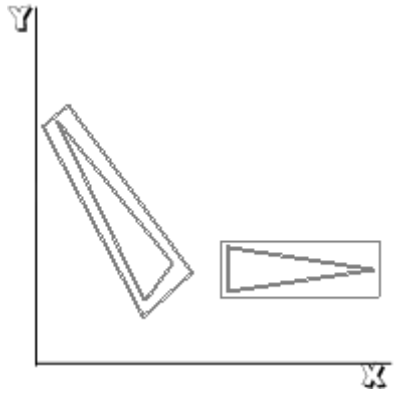
3.2.3.2 Segunda etapa de la detección de colisiones

Para esta segunda etapa donde se requiere mayor precisión, la colisión se va a comprobar al nivel de cada objeto geométrico de los que consta el modelo. Se ha utilizado un algoritmo más complejo que en el de la primera etapa. En vez de utilizar esferas, vamos utilizar cajas. A la hora de utilizar cajas envolventes (más comúnmente

llamadas bounding boxes) hay varias alternativas. La más sencilla de todas es utilizar cajas que tengan siempre la misma orientación. La comprobación de colisión entre este tipo de cajas es trivial, requiere muy poco cálculo, pero no envuelven bien al objeto al que representan.

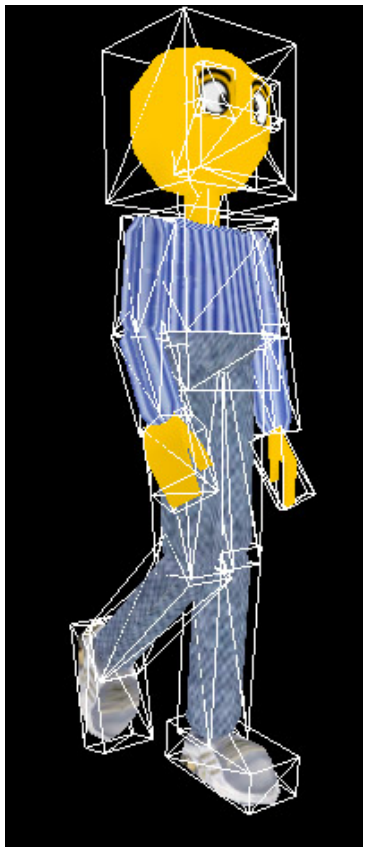


Como puede verse en el gráfico, aún siendo de forma similar ambas figuras, debido a su diferente orientación la segunda está mucho mejor envuelta que la primera. Para mejorar esta deficiencia, se utilizan un segundo tipo de cajas envolventes llamadas OBB (Oriented Bounding Boxes). Estas cajas tienen la ventaja de poseer cada una su propio sistema de coordenadas dependiente de la figura que están envolviendo lo que da cajas envolventes más precisas. En el ejemplo anterior, la envoltura quedaría así:

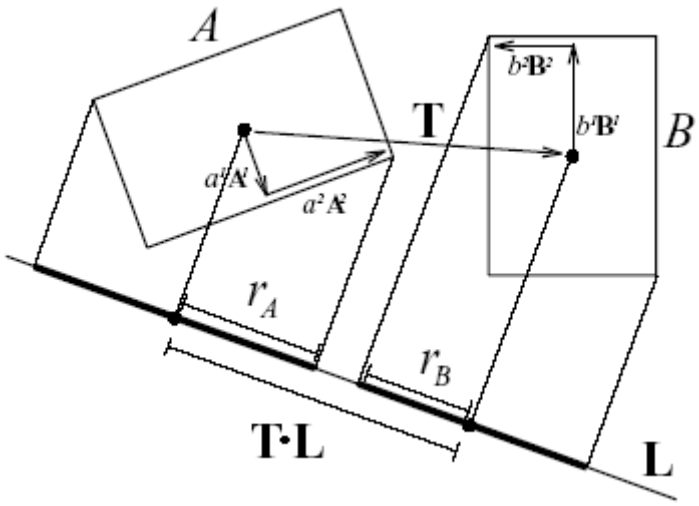


Como puede verse en este gráfico, ahora las cajas envolventes no están orientadas de la misma manera con lo que se consigue una mejora de la envoltura en el caso de la primera figura, esto permitirá mayor precisión en la detección de las colisiones.

Las cajas que utilizamos en nuestro motor son OBBs. A continuación se muestra un gráfico de un modelo cargado en nuestro motor, y con las OBBs de sus correspondientes objetos geométricos calculadas:



El cálculo de intersección entre OBBs se ejecuta aplicando el teorema de los ejes de separación de Gottschalk [GOTT]. En este teorema se define el eje de separación entre dos objetos convexos como aquella recta tal que las proyecciones ortogonales de los objetos sobre ella no se solapan. Nuestras OBBs no colisionarán entre sí, si nuestro algoritmo consigue encontrar un eje de separación entre ellas. En el siguiente dibujo se puede ver la representación gráfica de lo que acabamos de comentar en un ejemplo en dos dimensiones:



L es un eje de separación para las OBBs A y B porque las proyecciones ortogonales de A y B sobre L no se solapan.

Para comprobar si nuestras OBBs colisionan, deberemos hacer el chequeo sobre 15 ejes de separación: uno por cada eje de orientación de la primera OBB, uno por cada eje de orientación de la segunda OBB, y por último uno por cada producto vectorial del eje de orientación de una OBB con el de la otra OBB. Con lo cual salen 3 + 3 + 9 = 15 ejes de separación.

Siguiendo la nomenclatura del dibujo que hemos puesto de ejemplo, lo primero que se hace para comprobar si el eje L es eje de separación, es proyectar los centros de las OBBs sobre este eje. Seguidamente calcularemos Ra y Rb de la siguiente manera:

$$Ra = \sum | ai \cdot Ai \cdot L | \qquad Rb = \sum | bi \cdot Bi \cdot L |$$

Si la distancia entre la proyección sobre el eje de los centros de las OBBs es mayor que la suma de Ra y Rb , entonces es un eje de separación, y por tanto las OBBs no colisionan entre sí.

3.3 La animación en los objetos

Como hemos dicho tanto la cámara, los modelos, los objetos geométricos y las luces heredan de la clase Objeto, que les permite tener una posición y una orientación en el mundo. Pero muchas veces es necesario dotar a un objeto de animación, para que una cámara siga una ruta por ejemplo, o lo que es más importante, para dotar a los modelos de animaciones geométricas (saltar, andar, …)

Para conseguir esto, el Objeto no contiene sólo una matriz, si no que contiene una matriz por cada cuadro de animación que se quiera representar. De ésta manera, si queremos pintar un objeto animado, deberemos ir pasando por todas las matrices de animación del objeto a una velocidad determinada (frames por segundo) lo que dará la sensación de animación.

3.4 El fichero .GRT

Como se ha podido observar en los capítulos anteriores, la representación geométrica de un modelo es una estructura bastante compleja como para introducir esta información a mano. Para simplificar esta tarea, se ha desarrollado un plug-in para la herramienta de modelado 3DStudio Max , el cual genera como salida un archivo de formato propio mediante el cual construiremos nuestros modelos. El funcionamiento de este plug-in se explica en el capítulo 6.

4. Implementación de exteriores.

4. 1 Introducción.

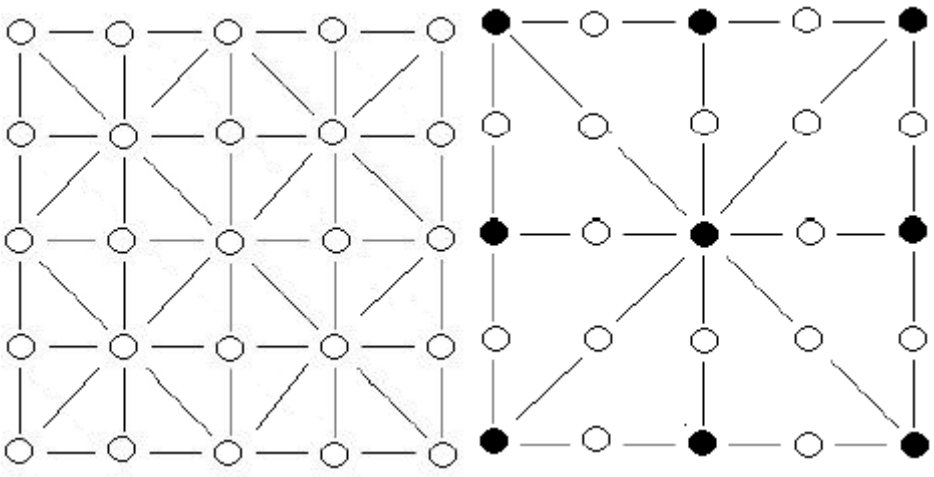
Actualmente existen una gran variedad de algoritmos para optimizar el renderizado de escenarios exteriores. Las aproximaciones más básicas a este problema son los algoritmos basados en mapas de alturas [TP]. Estos algoritmos utilizan como representación gráfica un archivo de imagen en escala de grises en el que, por ejemplo, los colores claros representan valores de altura mayores. De este modo, a cada píxel de la imagen le hacemos corresponder un valor de altura determinado entre 0 y 255. Por supuesto este valor puede ser escalado multiplicándolo por una constante y obtener mapas más o menos "escarpados", al igual que se puede determinar la distancia entre cada punto de la imagen obteniendo mapas más o menos grandes a costa de un sacrificio en la calidad del escenario representado.

El principal inconveniente de estas técnicas es que, por lo general, los mapas a representar suelen ser de gran tamaño, lo que requiere dibujar un gran número de triángulos con el consecuente perjuicio en el rendimiento. Bajo este contexto se han desarrollado los algoritmos de nivel de detalle continuo (CLOD)[TP]. Su objetivo es reducir la cantidad de polígonos enviados a la tarjeta a costa de trabajo extra para la CPU. Los principales exponentes de este grupo son ROAM [RTOAM] (Real-time Optimally Adapting Meshes) y Geomipmapping [BO].

Recientemente, con las mejoras en hardware en este campo, la GPU de las nuevas tarjetas son capaces de controlar una carga de trabajo mayor y es necesario considerar cuidadosamente hasta que punto es beneficioso penalizar a la CPU. Por este motivo hemos considerado que la mejor opción en este proyecto es el uso de Geomipmapping, ya que con él se obtienen buenos resultados con una sobrecarga de CPU inferior a los de las otras opciones consideradas.

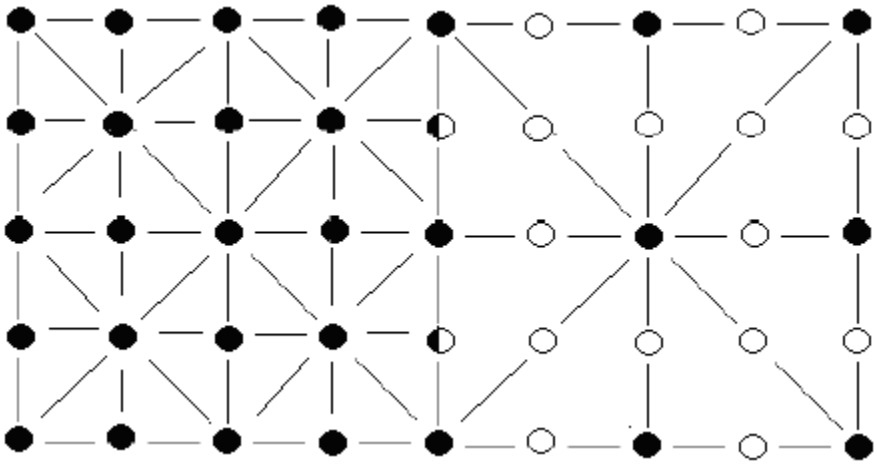
4. 2 Geomipmapping.

El nombre de este algoritmo viene del término "Geomipmaps". Esto es una analogía con los Mipmaps de las texturas. La idea general es que las zonas del terreno que están más alejadas de la cámara no necesitan tantos polígonos para dibujarse como las que están cerca, ya que de todos modos se van a ver más pequeños y no se van a apreciar. Esta es la misma idea que en las técnicas de mipmapping de texturas, en éstas preparamos una misma textura con varias resoluciones generando una serie de niveles y según la distancia de la cámara a la textura escogemos un nivel u otro. Si aplicamos este concepto a los mapas de alturas explicados en la introducción, podemos crear bloques de puntos de un tamaño dado (n x n). En un nivel de detalle máximo (nivel 0) dibujaríamos todos los vértices mientras que si la cámara se aleja, el detalle cambiaría a un nivel más bajo en el que se dibujan menos vértices.



En la imagen se aprecia un bloque de 5x5. En la izquierda se pintan todos los vértices, pues es el de máximo detalle, el de nivel 0. En el dibujo de la derecha solo se pintan los vértices coloreados.

Esta organización de los triángulos tiene el inconveniente de que dos bloques contiguos de distinto nivel de detalle forman “agujeros” en sus fronteras. Esto se produce porque el bloque de más detalle dibuja vértices en sus extremos que el otro no dibuja. Estos se dibujan a su altura correspondiente en el mapa de alturas, pero en el bloque con menos detalle el triangulo formado por los vértices que sí se dibujan no tiene porque pasar por ese vértice. Los dos triángulos del bloque con más resolución estarían en planos distintos, y entre estos y el triángulo del otro bloque quedarían huecos.



La solución a este problema es omitir este vértice en los bloques que se encuentren situados junto a un bloque de menor resolución. Esto nos obliga a hacer una serie de comprobaciones antes de dibujar cada bloque, ya que la misma situación se da en el resto de direcciones. Hay que averiguar el nivel de detalle de los bloques circundantes y decidir que vértices vamos a omitir si esto fuera necesario.

Si almacenamos los bloques en un array, cada vez que la cámara se moviese tendríamos que recorrerlo para actualizar todos los valores del nivel de detalle y comprobar antes de dibujar si los vértices frontera forman huecos. Para

esto se pueden usar cuatro valores booleanos, uno para cada posible vértice a omitir (superior, inferior, izquierdo y derecho), si la variable tiene el valor false, el vértice correspondiente no se dibuja. La función de dibujar iría recorriendo el array de bloques y actualizando los valores booleanos comprobando el nivel de detalle de los bloques adyacentes. Si uno de estos bloques tiene menos nivel de detalle, entonces el valor de la variable correspondiente debe ser false y ese vértice no debe dibujarse.

4.2.1 Implementación del algoritmo.

Para generar los exteriores hemos usado una clase abstracta Paisaje, que contiene todas las funciones necesarias en un terreno basado en mapa de alturas, independientemente del algoritmo usado para dibujarlo. Entre estas funciones básicas se encuentran las de cargar un mapa de alturas a partir de un archivo tga, crear la textura del mapa, generar las sombras estáticas en función de la geometría del mapa y las funciones para obtener el valor de la altura en un punto dado del mapa de alturas, o en un punto del terreno, para lo que es necesario interpolarla conociendo las de los puntos más cercanos al buscado. Esto es necesario para que los personajes puedan moverse con fluidez por el paisaje y no "saltando" de vértice en vértice.

Los bloques se gestionan mediante una estructura GEO_PARCHE que contiene dos campos: la distancia de este parche a la cámara y el nivel de detalle que le corresponde. La clase Geomipmapping contiene un vector de todos los bloques. En cualquier momento se puede calcular la posición de cada bloque, ya que conocemos el tamaño de cada uno de ellos y las dimensiones del mapa.

Para comprobar la mejora de rendimiento de este algoritmo con respecto del consistente en dibujar directamente todos los puntos del mapa de alturas también se implementó esta aproximación, que extiende también a la clase Paisaje.

4.2.2. Atributos configurables

La clase que implementa el geomipmapping admite una serie de atributos que nos permiten configurar el terreno para aproximarlo más al objetivo buscado, así, tenemos los siguientes:

-**Tamaño del bloque**: representa el valor del lado del parche que deseamos, es decir, el número de píxeles que representarán un bloque en el mapa de alturas.

-**Separación entre puntos**: es el valor de la separación entre los puntos contiguos en el mapa de alturas. Aumentando este valor podemos aumentar el tamaño del mapa con facilidad, usando un mismo mapa de alturas.

-Escala: es un factor que se aplica a los valores de alturas contenidos en el mapa. Como estos valores están, en principio, comprendidos entre 0-255, le aplicamos este factor, permitiéndonos obtener un escenario más montañoso.

-Detalle: con este atributo podemos marcar el nivel de detalle que deseamos en el mapa. Se aplica a la fórmula que calcula el nivel de detalle para cada bloque.

4.2.3 - Actualización del mapa.

El objeto Terreno posee una función encargada de actualizar el detalle de todos los bloques del terreno cada vez que la cámara cambia de posición. Eso se produce cada vez que se pulsa una tecla para moverse. A partir de la matriz de la cámara obtenemos el punto exacto donde se encuentra la cámara en el sistema de coordenadas del mundo. Para eso es necesario obtener los valores de la última fila de la inversa de la matriz.

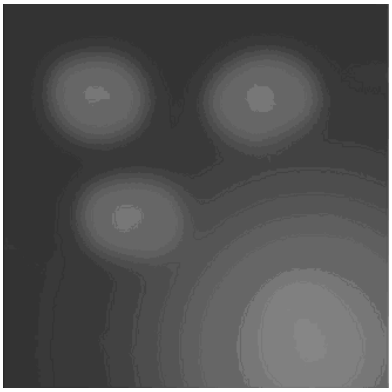
El proceso de actualización recorre el vector de bloques calculando la distancia de estos a la cámara. En función de este valor le asocia el nivel de detalle que le corresponda según la fórmula siguiente:

```
if (parches[i].distancia < (70*paso/detalle)) {
                        parches[i].LOD = 0;

} else if (parches[i].distancia < (150*paso/detalle))
                        parches[i].LOD = 1;

else if (parches[i].distancia < (250*paso/detalle))
                        parches[i].LOD = 2;

else if (parches[i].distancia >= (250*paso/detalle) )
                        parches[i].LOD = 3;
```

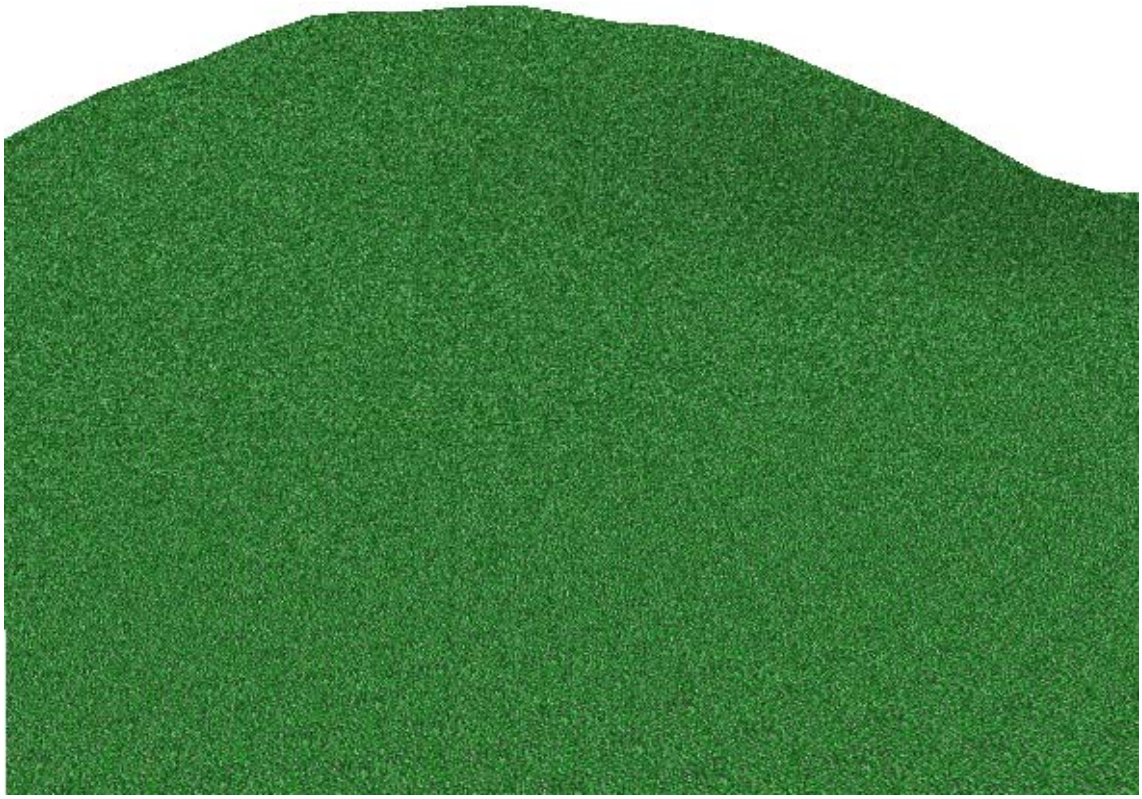
La próxima vez que se pinte el mapa cada uno de los bloques tendrán el número de vértices que le corresponda según su nivel de detalle.

4.2.4 - Resultados.

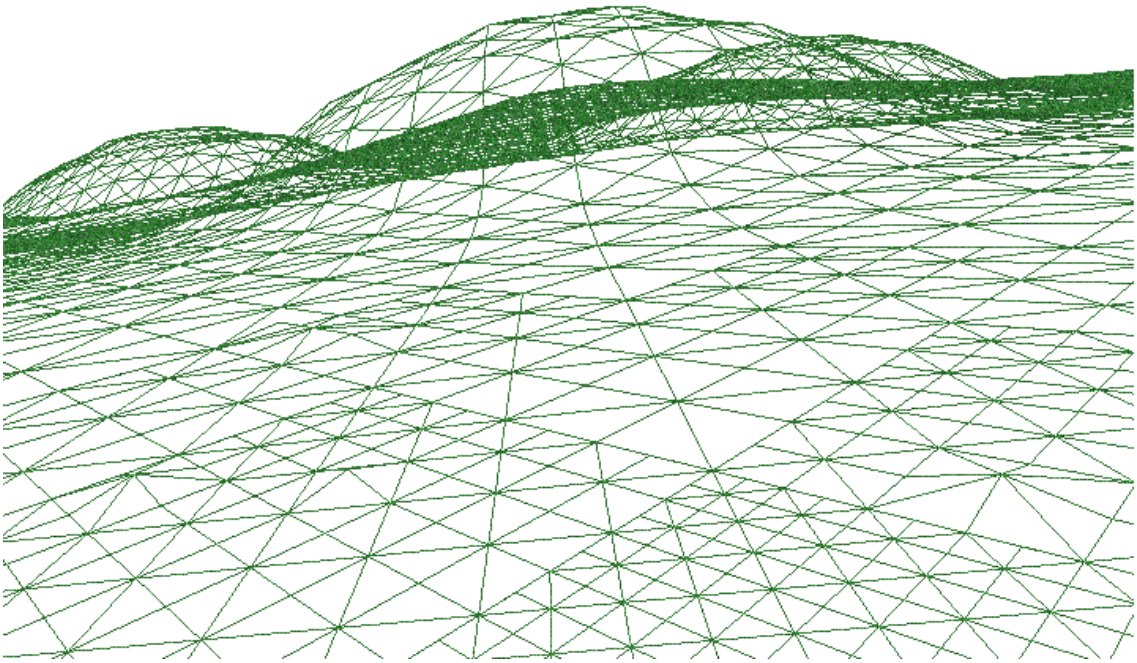
Los resultados obtenidos han sido los siguientes:



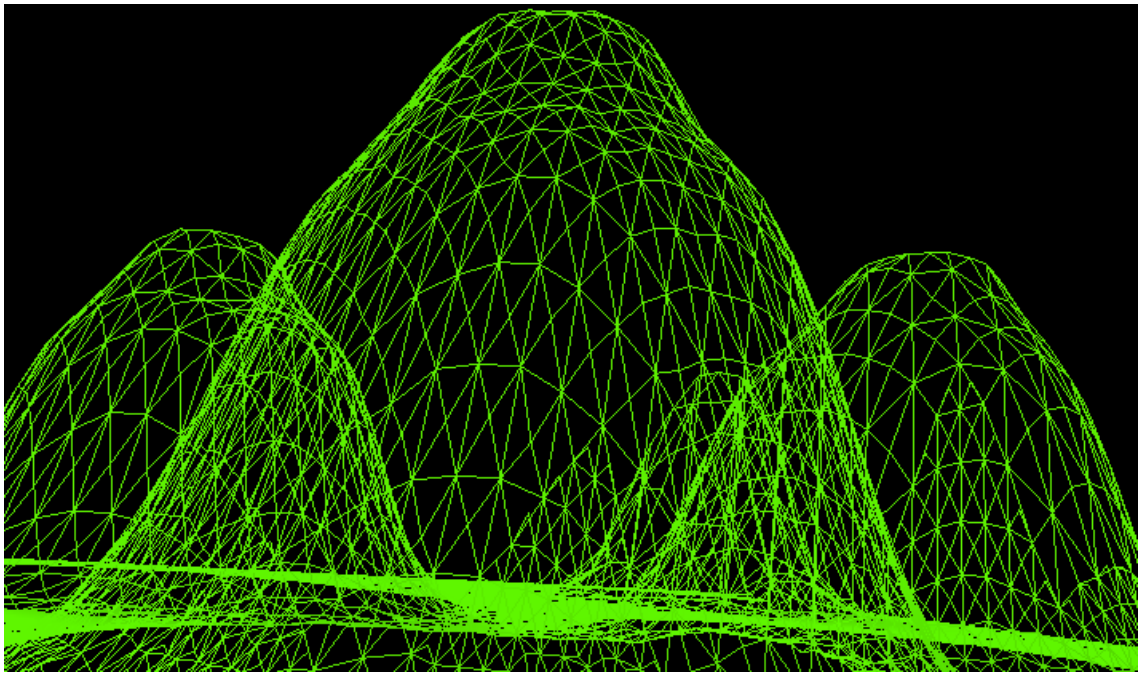
Mapa de alturas



Terreno con una textura aplicada.



Vista de los vértices del terreno.



Un terreno un poco más escarpado.

4.3.- Iluminación estática del terreno.

4.3.1- Introducción.

En un principio se descartó utilizar la iluminación ofrecida por OpenGl ya que ésta se calcula a partir de las normales. Como los vértices del terreno varían según la posición de la cámara, habría que recalcular las normales cada vez que ésta se desplazase. Esto implica un alto coste y por tanto un mal rendimiento general.

Un terreno gana una gran cantidad de realismo si posee sombras correctamente generadas. Hay varias formas de resolver este problema. Mediante sombras dinámicas, que se generan en tiempo real o mediante sombras estáticas, que se generan una sola vez al comienzo del programa. Las sombras generadas dinámicamente se recalculan cada vez que la escena se pinta, aunque esta no haya cambiado, para que se proyecten correctamente si los objetos se mueven. Esto requiere una gran cantidad de cálculos, haciendo descender el rendimiento general de la escena. Mediante las sombras estáticas podemos permitirnos unos cálculos más precisos, ya que como sólo se calculan una vez no sufrimos una pérdida de rendimiento constante.

Enfocando esto a un terreno, la aproximación más ventajosa es la segunda, ya que como el terreno no se mueve no hay necesidad de recalcular todas las sombras que éste proyecta cada vez que se va a pintar. Nosotros hemos seguido esta idea.

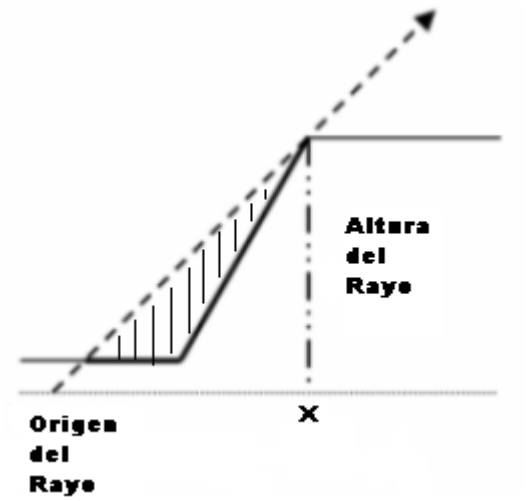
El método utilizado para generar las sombras en el terreno está basado en las técnicas de trazado de rayos. El procedimiento consiste en precalcular las sombras a partir del mapa de alturas mediante el ángulo de incidencia de los rayos del sol. El brillo de cada punto es un valor de escala de grises que simula la cantidad de luz que alcanza a cada punto. Estos valores luego se difuminan, es decir, si los puntos colindantes a uno concreto son más brillantes que éste, este punto a su vez se vuelve más brillante. De este modo se obtienen “sombras suaves”, como las generadas por la luz del sol, en contraposición con las generadas por luces puntuales.

Para almacenar los brillos de los puntos basta utilizar un vector de unsigned char del mismo tamaño que el mapa de alturas. Luego estos valores se añaden como colores en cada vértice, oscureciendo o aclarando el tono de la textura.

Esta técnica ha sido desarrollada por Robert Nagy en su artículo “Ray Lighting. Creating realistic shadows for terrain” [RG].

4.3.2- Descripción del algoritmo.

El algoritmo consiste en generar rayos teóricos con un ángulo determinado desde cada punto del eje x. Si algún punto del terreno interseca con el rayo, ese punto genera una sombra sobre cada uno de los puntos que le precede en el trazado de ese rayo.



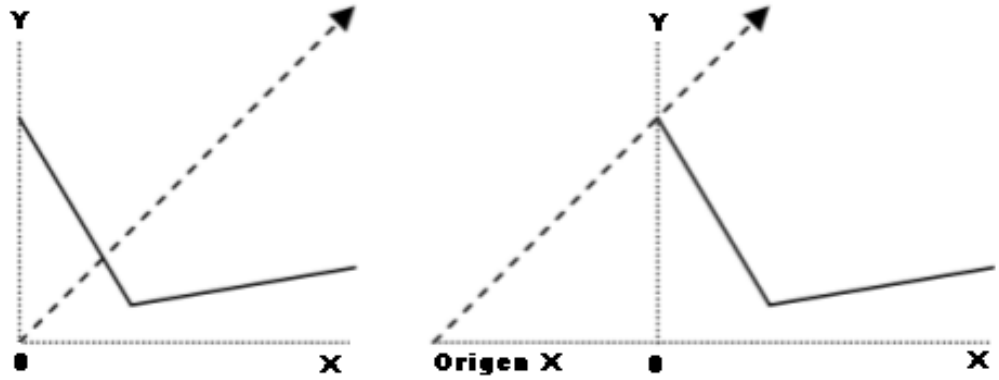
La forma de calcular la altura del rayo en cualquier punto de su trayectoria es muy sencilla. A partir del triángulo rectángulo formado por la horizontal y el rayo. Conociendo la longitud del cateto contiguo y el ángulo del rayo, podemos usar la fórmula de la tangente para calcular la longitud del cateto opuesto.

$$AlturaRayo = (X - OrigenDelRayo) * Tangente(Angulo)$$

Si comenzamos a generar los rayos desde la posición x = 0 (suponemos que el terreno se encuentra en los valores positivos del eje X y que comienza en x = 0) se puede dar un problema. Supongamos que el valor del mapa de alturas en esa posición tiene una altura mayor que cero. Como el rayo comienza ahí, todos los puntos que queden por encima del trazado de éste quedarán excluidos de los cálculos y nada podrá proyectar sombra sobre ellos. Para solucionar este problema es necesario que los rayos comiencen a proyectarse desde las posiciones negativas del eje X. Exactamente tan atrás como sea necesario para que el primer rayo tenga la altura del mapa de alturas en x = 0.

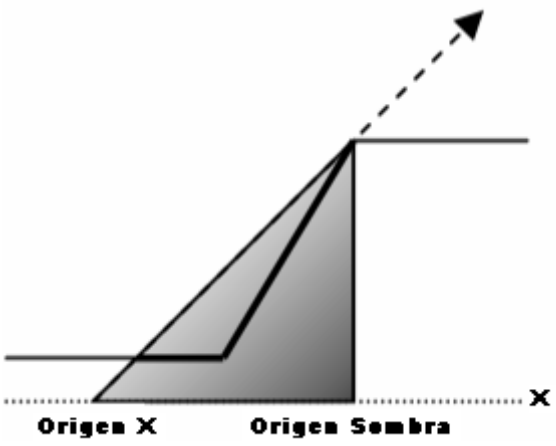
Al igual que anteriormente, el valor del origen de los rayos en el eje X se puede calcular a partir del triángulo rectángulo formado por el rayo, la vertical del punto en x = 0 con la altura máxima posible y la horizontal. La formula sería la siguiente:

$$Origen\ X = AlturaMáxima / tangente(Angulo)$$



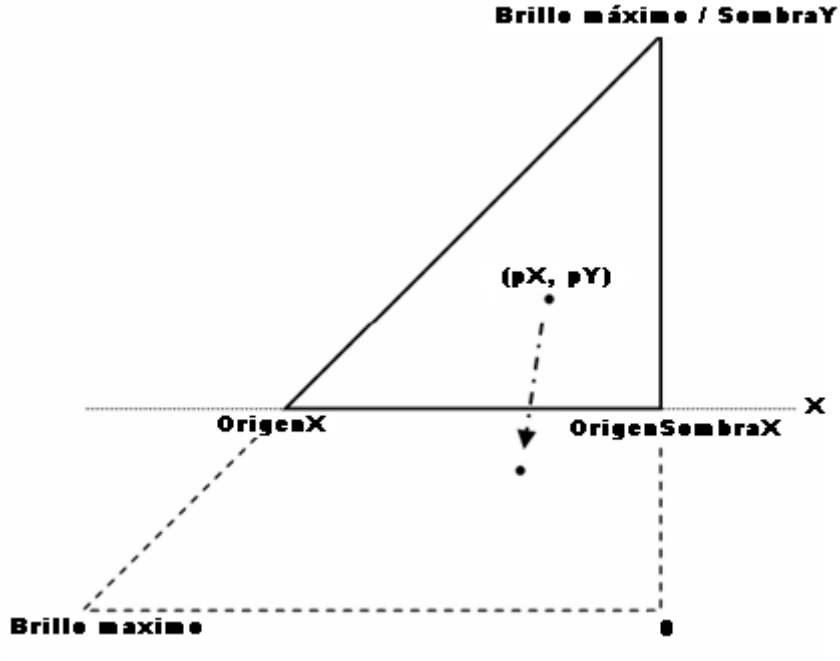
4.3.3- Brillo.

Cuando el sol ilumina un objeto, diferentes cantidades de luz inciden en cada uno de sus puntos. En este algoritmo, este factor se traduce en el valor del brillo. Conseguimos el efecto deseado mediante una escala de grises. De este modo, los puntos más cercanos al origen de la sombra serán más oscuros y los más alejados más claros.



Por tanto calculamos el brillo de un punto dado según su posición relativa a la sombra. Para mantener los niveles de brillo al mismo nivel, usamos una constante de brillo máximo. Con la siguiente fórmula escalamos los valores de la escala de grises:

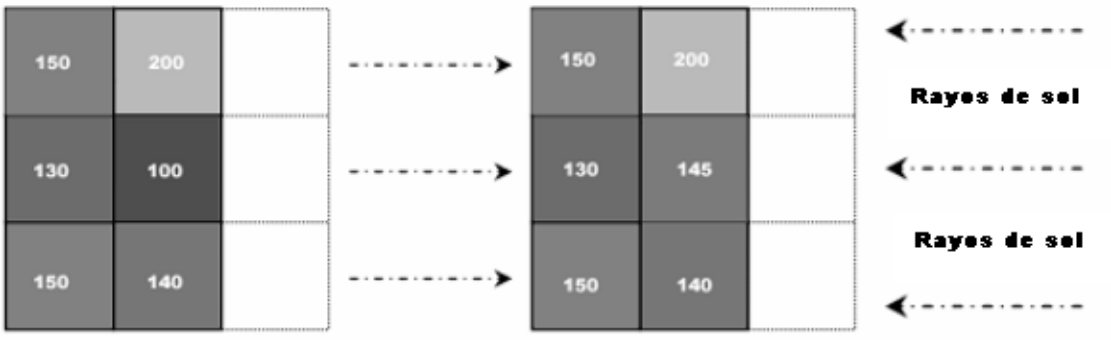
*Brillo = BrilloMáximo * [(OrigenSombraX – pX) / (OrigenSombraX – OrigenX) + (pY / SombraY)]*



4.3.4- Difuminado de sombras.

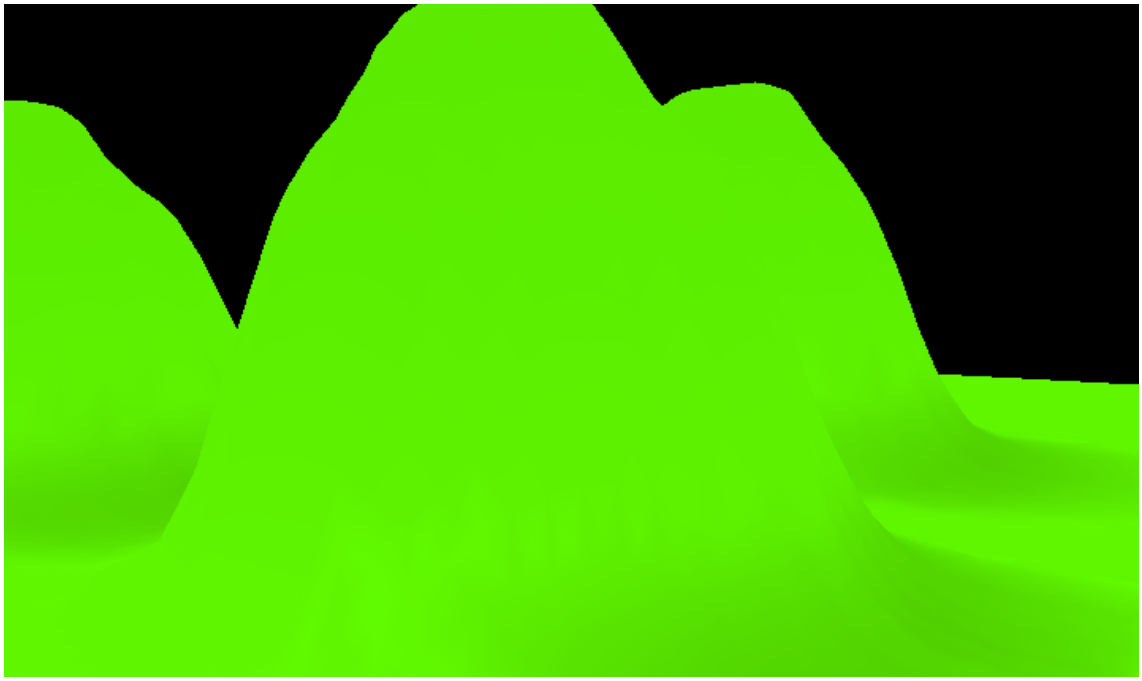
Como las fuentes de luz grandes dan como resultado sombras difuminadas, si queremos obtener sombras realistas en exteriores, deberemos, por tanto, difuminarlas. Una vez calculados todos los valores de brillo de los puntos deberemos hacer una nueva pasada a todo el vector que almacena estos valores para comparar cada uno de ellos con sus adyacentes y de este modo poder aclarar u oscurecer el brillo de cada vértice como corresponda.

Para hacer los cálculos sólo se tienen en cuenta los vértices adyacentes en dirección de los rayos de sol, como muestra el siguiente dibujo.

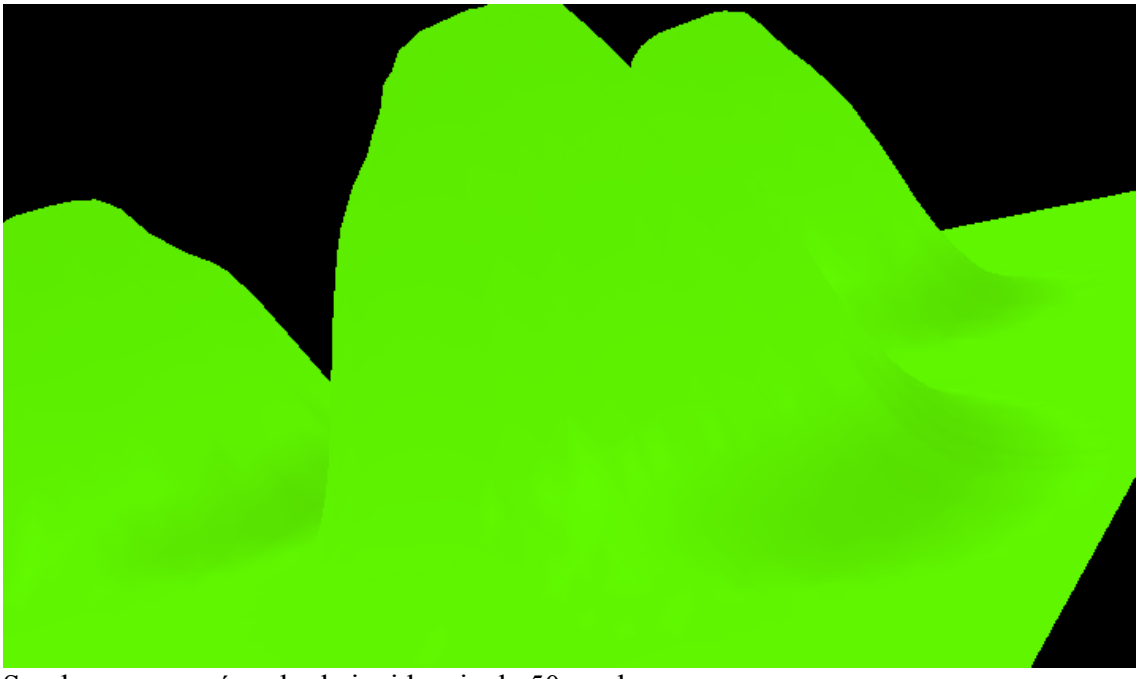


El difuminado de cada punto viene dado por la media aritmética del brillo de cada punto y de sus puntos adyacentes.

4.3.5- Resultados.



Sombras con un ángulo de incidencia de 30 grados.



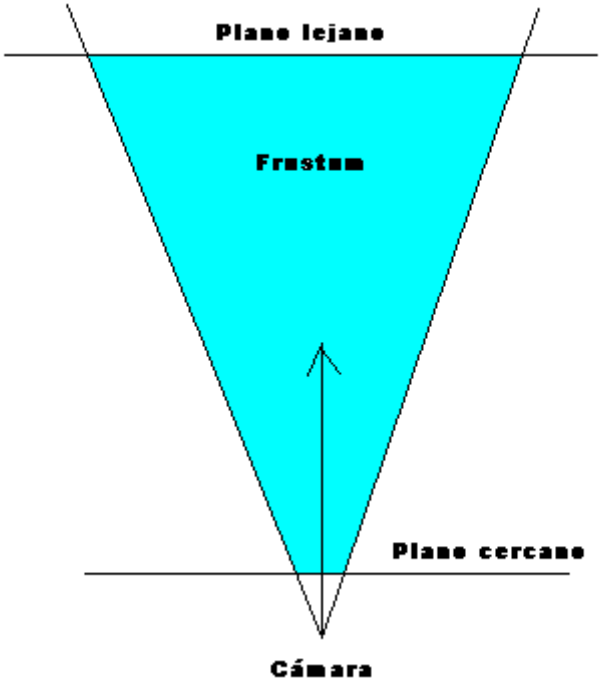
Sombras con un ángulo de incidencia de 50 grados.

4.4.Frustum culling

4.4.1- Introducción.

Por Frustum culling se entiende el algoritmo encargado de recortar la parte de la escena que no se va a ver en pantalla, por caer fuera del campo de vista de la cámara. Si evitamos enviar a la tarjeta gráfica todos los vértices que no va a necesitar dibujar en pantalla la aliviaremos de una gran carga de trabajo y obtendremos una mejora de rendimiento, sobre todo en escenas grandes.

El frustum es el volumen del mundo que será visible para una cámara determinada. Su forma es una pirámide truncada en la punta. Por tanto podemos representar el frustum mediante seis planos, cercano, lejano, izquierdo, derecho, alto y bajo. El algoritmo consiste en averiguar que objetos están dentro de esta pirámide.



4.4.2- Obtención de los planos.

Para empezar vamos a suponer que la matriz de modelado y vista es la identidad. Estamos, por tanto, situados en el origen de coordenadas del mundo, mirando hacia el eje Z positivo. Partimos de la matriz de proyección.

Supongamos un vértice v = (x,y,z,w=1) y M la matriz de proyección. Al transformar el vértice v con la matriz M nos queda:

$$M = \begin{vmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{vmatrix}$$

$$Mv = v' \Rightarrow \begin{vmatrix} x' \\ y' \\ z' \\ w' \end{vmatrix} = \begin{vmatrix} x * m11 + y*m12 + z*m13 + w*m14 \\ x * m21 + y*m22 + z*m23 + w*m24 \\ x * m31 + y*m32 + z*m33 + w*m34 \\ x * m41 + y*m42 + z*m43 + w*m44 \end{vmatrix}$$

El vértice v' esta en coordenadas homogéneas [CG], en este sistema de coordenadas un punto en 3D viene representado por cuatro coordenadas (x, y, z, w) en vez de por tres. Un punto en coordenadas homogéneas representa el mismo punto que otro cuyas coordenadas son múltiplos distintos de cero de las suyas. La representación típica de un punto (x, y, z, w) con w distinta de cero es (x/w, y/w, z/w, 1). Los puntos cuya coordenada w es cero son puntos en el infinito. La ventaja de utilizar las coordenadas homogéneas para comprobar el corte con el volumen de la cámara es que este, en una proyección en perspectiva (su volumen de vista canónico es una pirámide truncada) puede transformarse en el de una proyección paralela (un cubo), en el cual las operaciones necesarias son más sencillas. Esta transformación está contenida en la matriz de proyección de openGL, por tanto la posición relativa del punto v con respecto del volumen de la cámara en perspectiva es la misma que la posición relativa de v' con respecto de un volumen de vista de proyección paralela.

Si el vértice v' está dentro esta caja, entonces el vértice v está dentro de la pirámide que forma el frustum. Para garantizar que el vértice v' está dentro de esta caja, debe cumplir las siguientes inecuaciones:

$$-w' < x' < w'$$
$$-w' < y' < w'$$
$$-w' < z' < w'$$

De esto se pueden sacar las siguientes conclusiones:

Si $-w' < x'$ entonces x' está en el semiespacio interno del plano izquierdo.
Si $w' > x'$ entonces x' está en el semiespacio interno del plano derecho.
Si $-w' < y'$ entonces y' está en el semiespacio interno del plano bajo.
Si $w' > y'$ entonces y' está en el semiespacio interno del plano alto.
Si $-w' < z'$ entonces z' está en el semiespacio interno del plano cercano.
Si $w' > z'$ entonces z' está en el semiespacio interno del plano lejano.

Ahora, para obtener, por ejemplo, el plano derecho, tenemos que comprobar si la siguiente inecuación es cierta:

$$w' > x'$$

Si sustituimos estos valores por los representados en la matriz resultado anterior (Mv = v') nos da la siguiente ecuación (en la que el punto cae dentro del plano):

$$(x * m41 + y*m42 + z*m43 + w*m44) > (x * m11 + y*m12 + z*m13 + w*m14)$$

$$x(m41 – m11) + y(m42 – m12) + c(m43 – m13) + (m44 – m14) = 0$$

Esto es la ecuación de un plano de la forma:

$$ax + by + cz + d = 0$$

Procediendo de la misma manera con el resto de planos podemos obtener las demás ecuaciones.

Para particularizar estas operaciones a otra situación en que la matriz de modelado y vista no sea la identidad basta con premultiplicar esta matriz de modelado por la matriz de proyección, es decir:

$$M = Modelado * Proyección$$

Para representar los planos definimos una clase con los cuatro campos a,b,c,d como atributos:

```
//Plano de la forma ax + by + cz + d = 0
//Normal = (a,b,c)
class Plano{

public:
        float a,b,c,d;

        void normaliza();

        //si d<0 el punto esta detras del plano, d>0 esta delante, d=0
        //esta dentro
        float distanciaP(Punto3 p);

        //posicion relativa del punto al plano
        Posicion clasificaP(Punto3 p);

        //calcula normal
        Vector3 calcNormal();   };
```

Para contener la información de los seis planos creamos una clase *Frustum* que contendrá un vector de planos de seis componentes, donde implementaremos las funciones necesarias para actualizarlo cada vez que la cámara se mueva.

4.4.3- Corte de un bloque con el frustum

Una vez tenemos las ecuaciones de los seis planos debemos comprobar antes de dibujar cada bloque si éste está contenido o corta al frustum. Si no es así ese bloque no debe ser dibujado, ya que cae fuera del volumen de vista. Para esto, lo más sencillo es calcular la esfera que contiene al bloque y calcular el corte de ésta con el frustum.

La forma de calcular la esfera es inmediata, ya que al ser todos los bloques cuadrados, ésta queda definida con el centro de la esfera, que es el punto central del bloque y el radio, que será la distancia desde este punto al punto más distante del bloque, es decir, cualquiera de las cuatro esquinas.

Para calcular el corte de una esfera con cada uno de los planos, añadimos una función a la clase Frustum que recibe como parámetros el centro y el radio de la esfera. Después hay que calcular la posición relativa de la esfera con respecto a los seis planos:

```
int Frustum::contieneEsfera(float radio, Punto3 centro){

    float distancia;

    for(int i =0; i < 6; i++){

    distancia = planos[i].calcNormal().pEscalar(centro) +     planos[i].d;

            //estamos fuera;
            if (distancia < -radio) return -1;

            //estamos en corte;
            if( (float)fabs(distancia) < radio) return 0;

    }
    //si despues de pasar por todos los planos estoy en la cara
    //interna de todos es que estoy dentro del frustum

    return 1;

}
```

4.4.4- Aplicación al motor.

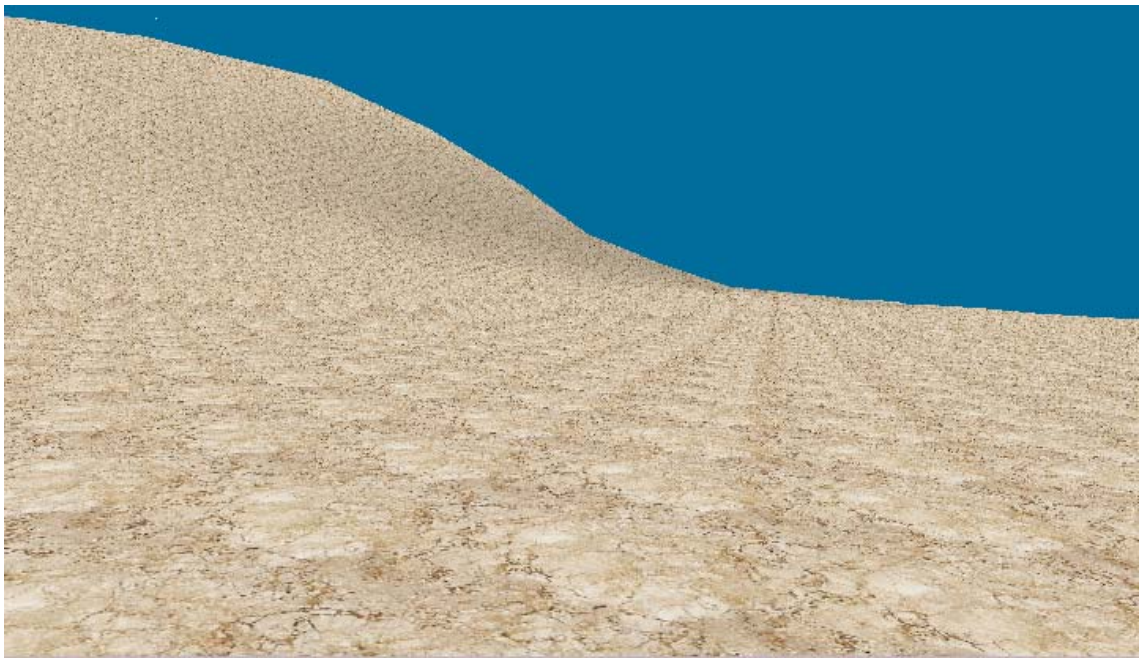
Con las estructuras definidas anteriormente podemos determinar qué bloques están dentro de la visión de la cámara y de este modo evitar dibujar todos aquellos que no serán visibles. Para conocer esta información, la clase geomipmapping contiene un puntero a la clase Frustum de la escena y antes de dibujar cada bloque comprueba si su esfera está contenida entre los seis planos. Si lo está, o si corta a alguno de ellos, entonces ese bloque se dibuja.

4.5-Efectos.

Para dotar de un poco más de detalle a los terrenos se han desarrollado unos cuantos efectos, entre los que se encuentran el skybox, un cubo que contiene a todo el terreno para simular el cielo y un sistema de partículas que simula una tormenta de nieve.

4.5.1- Skybox.

Un skybox [HA] es una forma sencilla de representar el cielo en un escenario de exteriores. Está formado por un cubo que ocupa todo el terreno alcanzable por el usuario en cuyas caras visibles podemos poner distintas texturas para simular un paisaje determinado. De este modo, con unas texturas adecuadas se obtiene un efecto bastante realista.



4.5.2- Sistema de partículas.

Un sistema de partículas [HA] es un conjunto de elementos independientes (partículas) que poseen una serie de características propias, como velocidad, color, tiempo de vida etc. Cada una de estas partículas actúa de un modo independiente, no se ve afectada de ninguna manera por el resto de partículas del sistema. Definiendo la forma de actuar de estas partículas se pueden desarrollar determinados efectos, como explosiones, chispas o lluvia.

Dada esta estructura, es fácil implementar un sistema de partículas usando las capacidades del diseño orientado a objetos. La idea es crear una clase básica que

contenga las características comunes a todos los sistemas y extenderla para desarrollar cada efecto particular.

4.5.2.1- Implementación de la base.

Para representar una partícula hemos desarrollado una estructura que contenga todos los atributos necesarios para guiar su comportamiento y que sea lo más general posible. Por este motivo la clase solo contendrá atributos sencillos, que bastará para simular efectos simples. De este modo evitamos el desperdicio de memoria que implicaría la reserva de una serie de atributos que no se usarán en la mayoría de los efectos (esto es significativo porque puede que tengamos miles de partículas en la escena en un momento dado).

```
struct PARTICULA{

    Vector3 posicion; //posicion de la particula
    Vector3 posPrev; //posicion previa de la particula
    Vector3 velocidad; //vector velocidad
    Vector3 aceleracion; //vector aceleracion

    float vida; //tiempo de vida de la particula

    float tam; //tamaño de la particula
    float tamDelta; //variacion de tamaño con el tiempo

    float peso; //relacionado con el efecto que produce la gravedad
    float pesoDelta; //cambio de peso con el tiempo

    float color[4]; //color de la particula
    float colorDelta[4]; //cambio de color con el tiempo

};
```

Ahora será necesario declarar una clase encargada de gestionar todas estas partículas, de inicializarlas, actualizar sus valores y dibujarlas en pantalla. Para esto creamos una clase *SistemaPartículas* que contenga un vector de partículas de un tamaño máximo definido en el constructor de la clase y una serie de funciones virtuales que realizan las funciones anteriormente descritas, algunas de las cuales deberán ser rellenadas por las clases hijas. También es necesario fijar un punto origen, desde el cual serán emitidas las partículas. Si deseamos que sean emitidas en un área cuadrada en vez de desde un punto fijo, solo tenemos que sumar un valor aleatorio acotado por los límites deseados a cada componente del punto para cada variable a emitir.

Las partículas serán emitidas en proporción al tiempo transcurrido, por eso la función de actualización recibe como parámetro una variable tipo float que representa el periodo de tiempo desde la última llamada a la función y la anterior. En algunos casos también puede ser útil conocer el tiempo transcurrido desde la creación del sistema.

La representación gráfica de cada partícula puede variar según el efecto deseado. Comúnmente suelen ser puntos, líneas o polígonos, que poseen la ventaja de poder añadirles una textura determinada. Por este motivo, en la clase básica no vamos a implementar el método encargado de dibujar las partículas en pantalla, sino que vamos a permitir a cada efecto concreto decidir la forma en que quiere representar sus propias partículas.

También hay que considerar que el movimiento de las partículas se verá modificado por la aplicación de una (o varias) fuerzas a cada una de ellas. Esta fuerza puede ser la gravedad que hace caer los copos de nieve o cualquier otra que pueda servir para simular un determinado efecto. Estas fuerzas serán representadas por un vector que se almacenará como atributo protegido de la clase.

```
class SistemaParticulas{

public:

        SistemaParticulas(){ };
        SistemaParticulas(int maxParticulas, Vector3 origen);

        virtual void actualiza(float tiempo) = 0;
        virtual void render(Matrix &cam) = 0;

        virtual int emite(int numeroParticulas);

        virtual void inicializaSistema();
        virtual void terminaSistema();

protected:

        virtual void inicializaParticula(int indice) = 0;

        PARTICULA *listaParticulas;
        int maximoPart; //numero máximo de particulas vivas
        int numParticulas; //numero total de particulas libres
        Vector3 origenSistema; //centro del sistema

        float tiempoAcumulado; //tiempo transcurrido desde la ult.
/                              //particula

        Vector3 fuerza; //comunmente gravedad


};
```

4.5.2.2- Efecto de tormenta de nieve.

Uno de los muchos efectos que pueden ser implementados fácilmente con el esquema anterior es la simulación de una tormenta de nieve. Para ello extendemos la clase SistemaPartículas comentada anteriormente y le añadimos una serie de atributos que caracterizan el efecto que deseamos implementar, como pueden ser las dimensiones del volumen en el que queremos que se generen los copos de nieve. Estos atributos, multiplicados por un factor aleatorio se sumarán a las componentes del origen del sistema, para calcular el punto donde se generará una partícula concreta.

También implementamos la función encargada de inicializar cada una de las partículas que vamos a emitir. En esta función es donde se calculará el origen de la partícula y su velocidad inicial, que se calcula a partir de un vector velocidad base y uno que representa la variación posible de velocidad, para que cada copo varíe su movimiento independientemente del resto.

De este modo, al actualizar las partículas sólo tenemos que recorrer el array que las contiene y modificar su posición conociendo su vector velocidad particular y el tiempo transcurrido desde la última actualización.

También debemos eliminar las partículas que han sobrepasado los límites que asignamos al marcar el volumen. En el caso de la tormenta de nieve, eliminamos los puntos que han alcanzado el suelo. Después sólo queda emitir las nuevas partículas que correspondan, según un ritmo que hemos definido en función del tiempo.

Para representar los copos hemos usado un quad al que le hemos añadido una textura para que simule un copo de nieve. La utilización de polígonos para representar las partículas tiene una dificultad añadida, probablemente el efecto que tengamos en mente requiera que los polígonos estén siempre orientados hacia nosotros y que si nos movemos hacia un lado no los veamos de canto. Esto se consigue con una técnica llamada **billboarding [HA].**

La técnica de billboarding es utilizada generalmente para simular un efecto de tridimensionalidad a partir de un objeto 2D en un mundo 3D. El efecto se consigue haciendo que el objeto 2D esté orientado siempre hacia la cámara evitando de este modo que notemos que no tiene profundidad. (esta técnica es la utilizada para crear a los enemigos en el videojuego DOOM). En la actualidad se usa para representar arbustos y árboles en entornos tridimensionales sin necesidad de sobrecargar al sistema utilizando una malla para cada uno. Como comentamos antes, este efecto también es muy útil para representar las partículas de nuestro sistema.

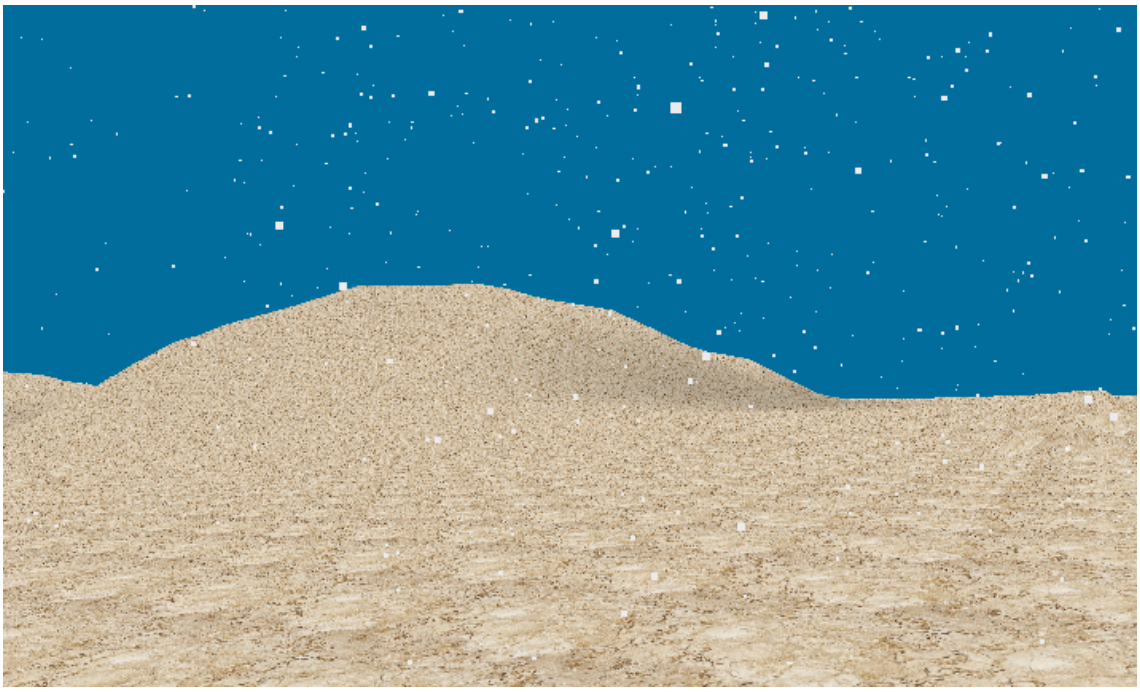
Para obtener la dirección a la que deben apuntar nuestras partículas deberíamos invertir los efectos de la matriz de la cámara multiplicando esta por su inversa. Afortunadamente esto no será necesario, ya que hay una forma más sencilla de obtener el mismo resultado. Nosotros solo necesitamos la submatriz 3x3 superior, la cual ha de ser ortogonal. Esto significa que su inversa es igual a su transpuesta, que es más fácil de calcular. Una vez transpuesta solo necesitaremos los dos vectores de las dos primeras filas, que representarán los vectores ortogonales a la dirección en que apunta la vista hacia arriba y hacia la derecha. Para evitar el paso de transponer la matriz solo tenemos que coger las componentes de las columnas en vez de las de las filas.

```
Vector3 derecho = Vector3( glMatrix[0], glMatrix[4], glMatrix[8]);
Vector3 arriba = Vector3( glMatrix[1], glMatrix[5], glMatrix[9]);
```

Si tenemos estos vectores en cuenta a la hora de dibujar los copos, y suponemos que el punto de la posición del copo representa el centro de este, la forma de pintar un copo es la siguiente:

```
glBegin(GL_QUADS);

for(int i=0; i<numParticulas;++i){
//usando billboarding para que los copos apunten a la cámara

partPos = listaParticulas[i].posicion;
t = listaParticulas[i].tam/2;

//inferior izquierda
glTexCoord2f(0.0, 0.0); glVertex3f( partPos.x + (derecho.x + arriba.x) * (-t),
                                    partPos.y + (derecho.y + arriba.y) * (-t),
                                    partPos.z + (derecho.z + arriba.z) * (-t));

//inferior derecha
glTexCoord2f(1.0, 0.0); glVertex3f( partPos.x + (derecho.x - arriba.x) * t,
                                    partPos.y + (derecho.y - arriba.y) * t,
                                    partPos.z + (derecho.z - arriba.z) * t);

//superior derecha
glTexCoord2f(1.0, 1.0); glVertex3f( partPos.x + (derecho.x + arriba.x) * t,
                                    partPos.y + (derecho.y + arriba.y) * t,
                                    partPos.z + (derecho.z + arriba.z) * t);




//superior izquierda
glTexCoord2f(0.0, 1.0); glVertex3f( partPos.x + (-derecho.x + arriba.x) * t,
                                    partPos.y + (-derecho.y + arriba.y) * t,
                                    partPos.z + (-derecho.z + arriba.z) * t);

}
glEnd();
```

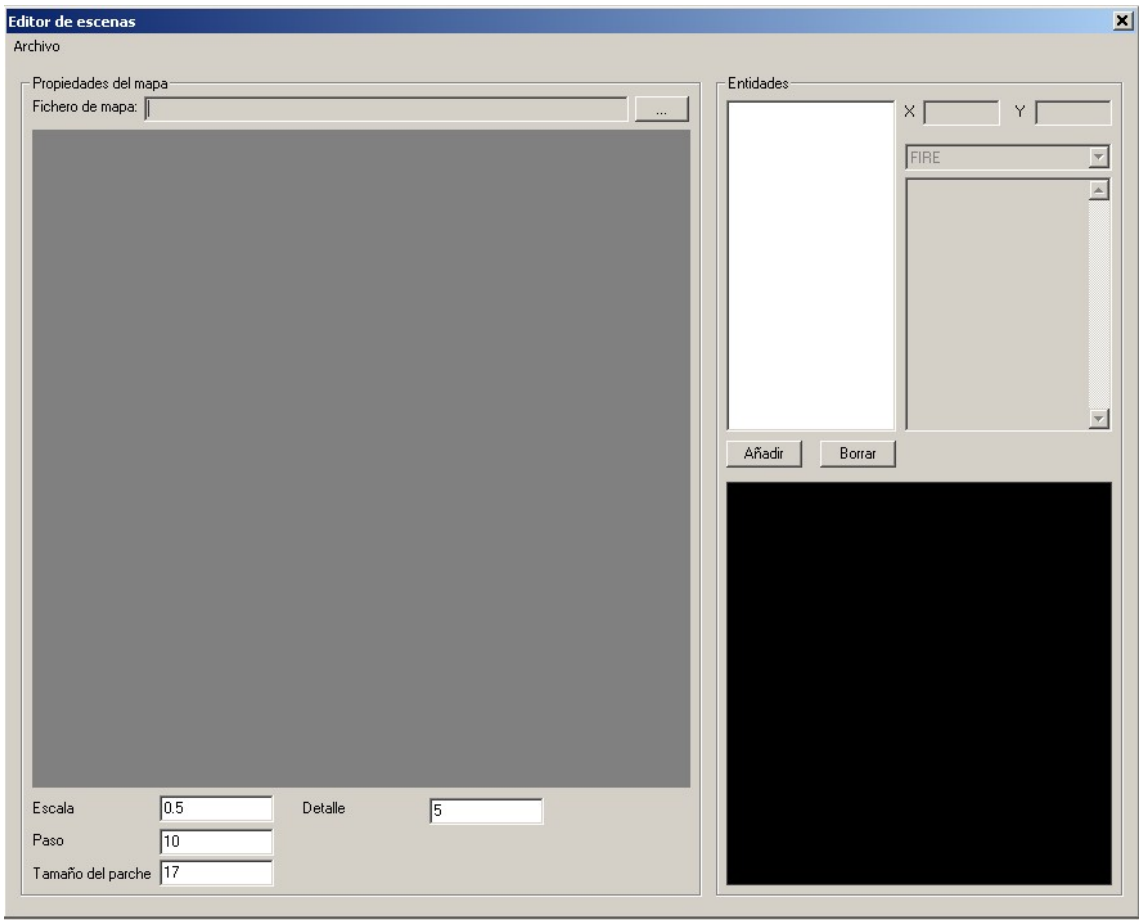
Los resultados obtenidos son los siguientes:



5 – El editor de escenas

5.1 Introducción

Esta herramienta se ha realizado para dar flexibilidad a la hora de poder crear escenas en nuestro juego. Con este programa el usuario puede introducir todos los parámetros que van a definir la escena, sin necesidad de tener que recompilar nada del código fuente del juego. Resumidamente el programa permite configurar el terreno con sus correspondientes parámetros, y configurar cada una de las entidades que formarán parte de la escena. Una entidad es cualquier cosa que pueda representarse en la escena, y está definida por una malla y por un comportamiento. El siguiente gráfico muestra el aspecto visual del editor de escenas:



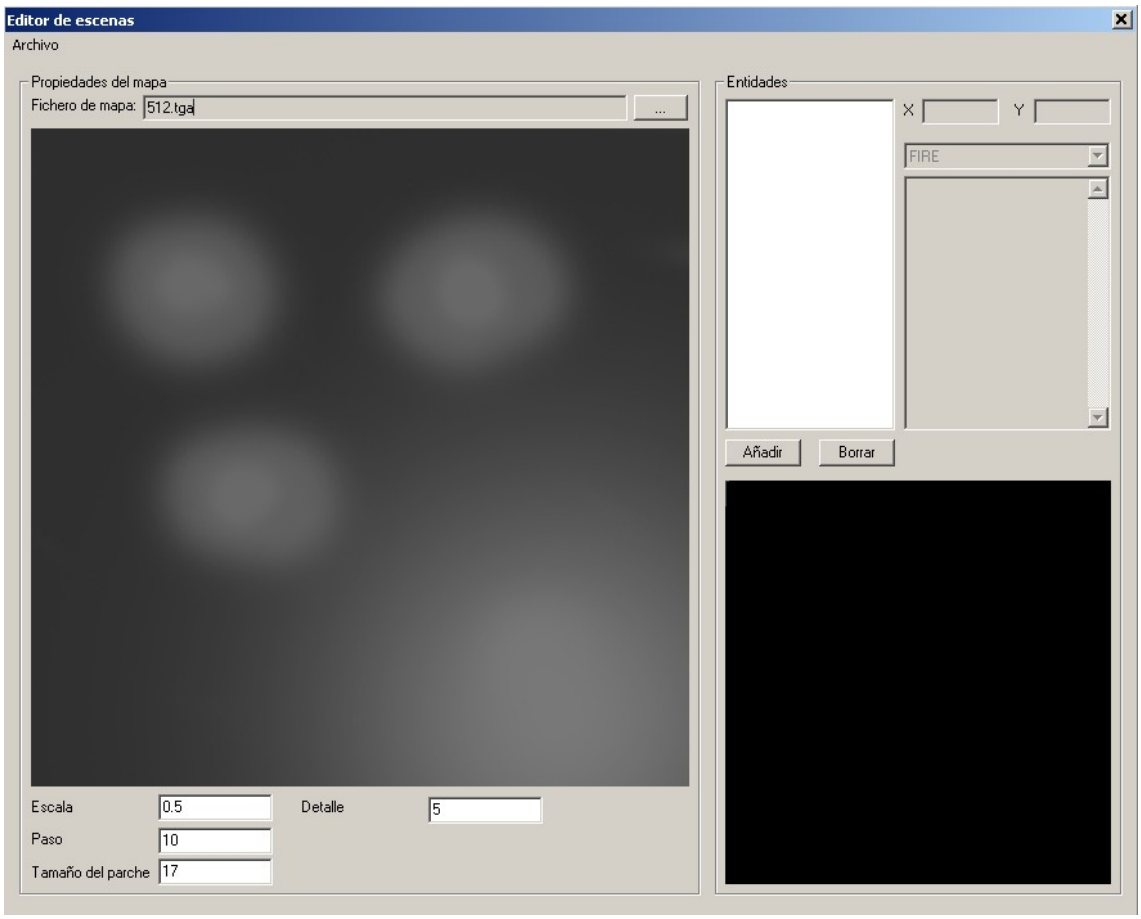
5.2 Configurando el terreno

Lo primero que tenemos que hacer para configurar el terreno de nuestra escena, es seleccionar un mapa de alturas. Para ello pulsamos el botón "..." y seleccionamos un fichero .TGA. Una vez seleccionado el mapa se mostrará en el cuadro correspondiente, solicitando el resto de los parámetros:

- Escala: es un factor que se aplica a los valores de alturas contenidos en el mapa.
- Paso: es el valor de la separación entre los puntos contiguos en el mapa de alturas.
- Tamaño del parche: representa el valor del lado del parche que deseamos, es decir, el número de píxeles que representarán un bloque en el mapa de alturas.

- Detalle: con este atributo podemos marcar el nivel de detalle que deseamos en el mapa.

Por ejemplo:



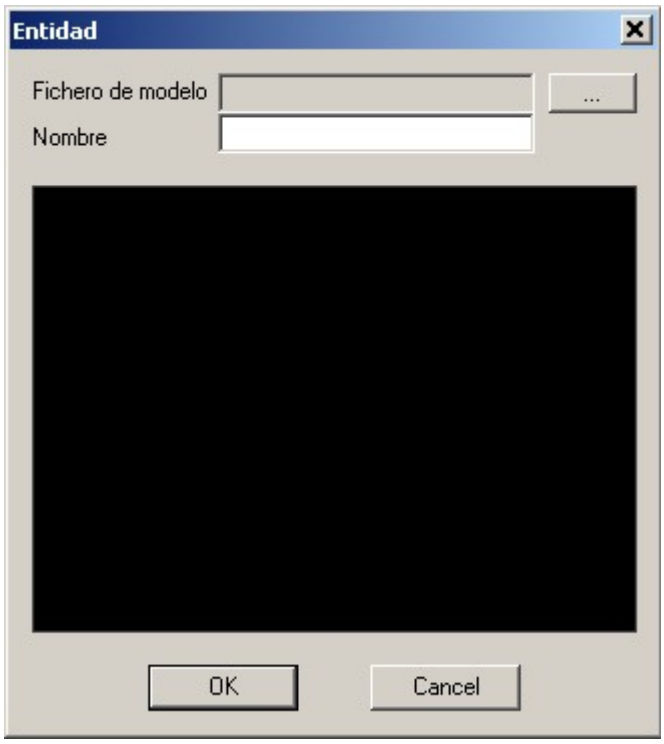
Ahora mismo ya tenemos una escena valida, ya que el único requisito indispensable de una escena, es que tenga un terreno que mostrar. Pero para que una escena tenga funcionalidad, debemos añadirle entidades, eso lo trataremos en el siguiente punto.

5.3 Añadiendo entidades

Esta es la parte que mas funcionalidad da a la escena. Gracias al lenguaje de scripting que se puede utilizar a la hora de modelar el comportamiento de una entidad, desde aquí podemos hacer casi cualquier cosa que se nos ocurra. Para crear una entidad debemos elegir una malla, un nombre que la identifique, y posteriormente escribir su comportamiento en el lenguaje de script.

5.3.1 Elegir malla y nombre

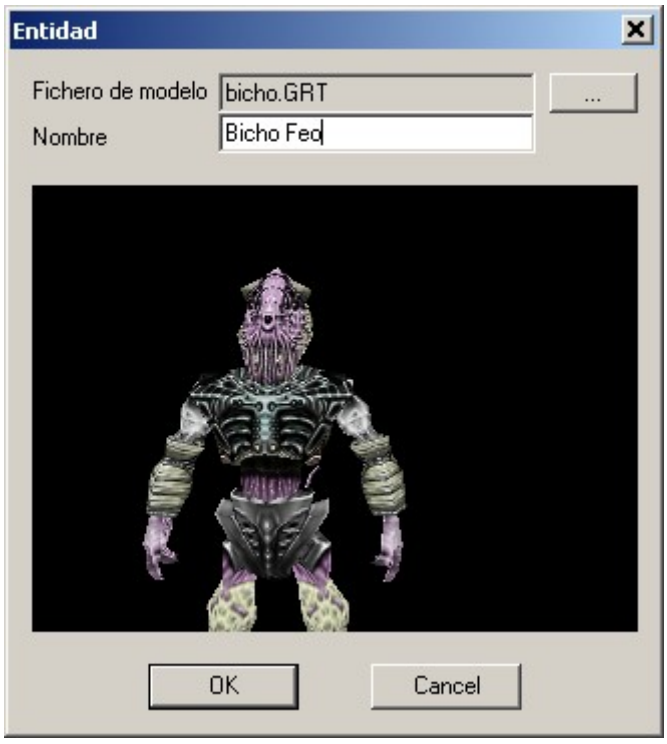
El primer paso para crear una entidad en nuestra escena es seleccionar una malla y un nombre. Para ello pulsamos el botón "Añadir" que tenemos debajo de la lista de entidades, y se nos abrirá un nuevo cuadro de diálogo:



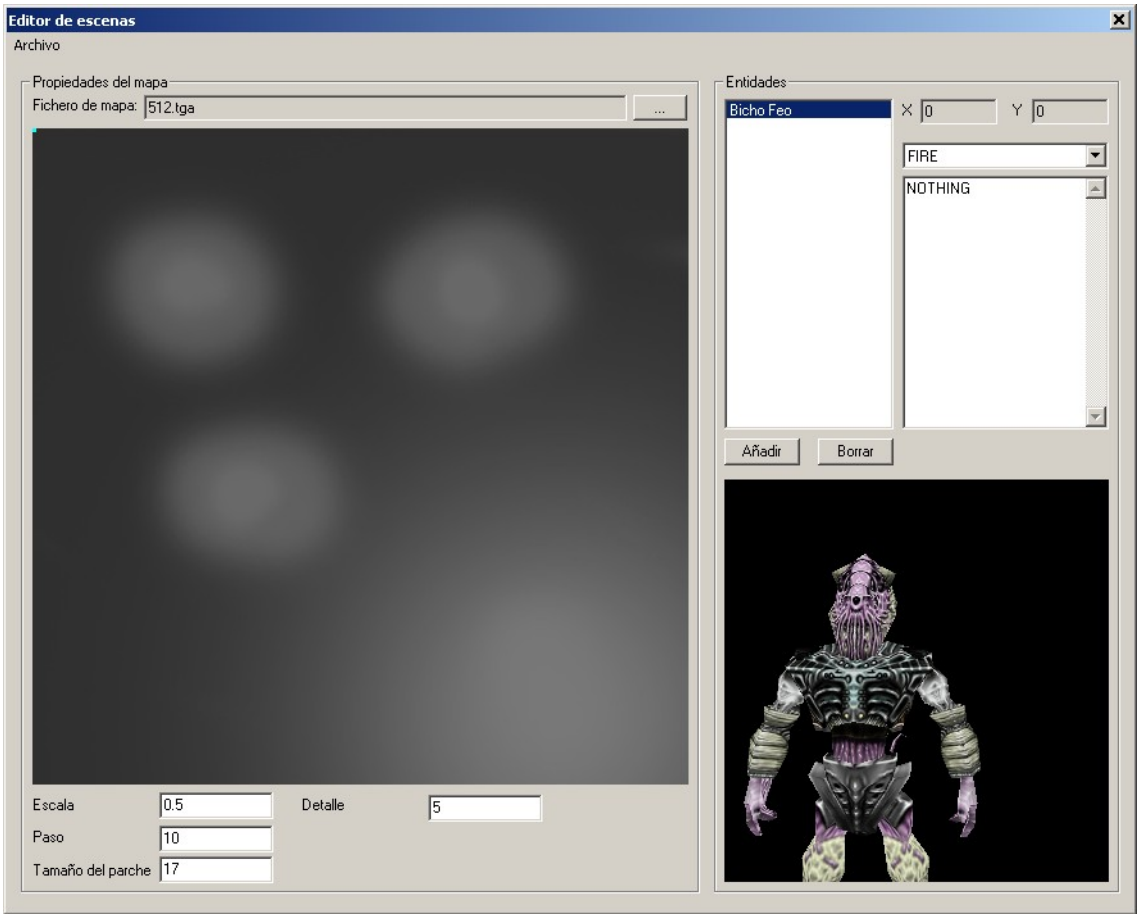
Para elegir la malla que representará a nuestra entidad en la escena, pulsamos el botón "...", y elegimos un fichero .GRT (este tipo de ficheros son los que se crean con el exportador de modelos de 3D Studio Max explicado detalladamente en el punto XX). Una vez seleccionamos la malla podemos ver una previsualización de la misma en el cuadro de abajo por si acaso queremos elegir otra.

En el campo del nombre debemos elegir una cadena de caracteres que identifique a nuestra entidad.

Una vez hemos seleccionado los dos campos tendremos una imagen de este tipo:



A continuación pulsamos “OK” e inmediatamente la nueva entidad será añadida a la lista de entidades de la escena:



5.3.2 Posicionando una entidad en el terreno

Si nos fijamos en la imagen de antes al lado de la lista de entidades aparecen dos casillas X e Y, con sus valores a cero. Estos dos valores representan dos de las tres coordenadas de la entidad en el terreno, la tercera se obtiene a partir del mapa de alturas. Como se puede observar , una vez creada la entidad, sus coordenadas son (0 , 0). En este punto vamos a ver como reposicionarla.

El mapa de alturas tiene su origen de coordenadas (0 , 0) en la esquina superior izquierda. Si nos fijamos, en ese punto tenemos ahora un "cuadradito", que es la representación de nuestra entidad en las coordenadas (0 , 0). Para reposicionarla, lo único que tenemos que hacer es arrastrarla con el ratón a la posición que deseemos.

5.3.3 Añadiendo comportamiento a una entidad

Llegados a este punto hemos conseguido colocar un terreno a nuestra escena, y posicionar una serie de entidades. Estas entidades ahora mismo no tienen comportamiento, simplemente son mallas colocadas en la escena. Este tipo de entidades podrían representar casas, árboles, ... es decir objetos estáticos y con los que el jugador no va a interactuar, son meramente posicionales. Pero si lo que queremos es hacer entidades que tengan un comportamiento y reaccionen a acciones del jugador debemos añadirles un comportamiento.

A una entidad se le pueden asignar dos tipos de comportamiento. Uno asociado al evento de reloj y otro asociado al evento de acción del jugador sobre él. El primero se ejecuta en cada ciclo de reloj de la escena, y el segundo se ejecuta cuando el jugador está cerca de nuestra entidad y pulsa el botón de acción.

Para editar cada uno de estos dos comportamientos, lo único que tenemos que hacer es seleccionar en el menú desplegable de debajo de las coordenadas el evento que queremos editar (TICK o FIRE) y escribir en el cuadro de edición de debajo el comportamiento que queramos asociarle. Como se puede observar, la entidad por defecto tiene en ambos eventos el comportamiento NOTHING. Este es el comportamiento más sencillo, y como su propio nombre indica, no hace nada.

Para crear el comportamiento de una entidad, se ha creado un lenguaje de script que es el que va a ser capaz de interpretar nuestro motor. Este lenguaje es de tipo imperativo, es decir, ejecutará las instrucciones que encuentra una detrás de otra de manera secuencial. Para dar versatilidad, se tiene la posibilidad de introducir bucles e instrucciones condicionales. Además de las instrucciones, nuestro lenguaje tiene también evaluadores, que son utilizados en las instrucciones como parámetros. Los evaluadores son instrucciones que, a diferencia de las instrucciones normales, devuelven valores. La introducción de los evaluadores ofrece más versatilidad al lenguaje, ya que las instrucciones pueden tener parámetros dinámicos en lugar de valores fijos. Los evaluadores también pueden tener como parámetros otros evaluadores. Un ejemplo de esto sería:

```
GOTO ( GETVAR ( 0 ) GETVAR ( 1 ) 500 )
```

En este caso, para los dos primeros parámetros de la instrucción GOTO se ha utilizado el evaluador GETVAR, mientras que para el tercero se ha introducido directamente un valor.

Además de instrucciones y evaluadores, el script posee una memoria de trabajo, y de esta manera mediante las instrucciones correspondientes, se pueden asignar y obtener valores de las variables de esta memoria.

A continuación se muestra el repertorio de instrucciones y evaluadores de nuestro lenguaje de script, los parámetros que toman, y una breve descripción de los mismos.

5.3.3.1 Instrucciones simples

Una instrucción es la unidad básica de ejecución de nuestro lenguaje. Las instrucciones son de la forma:

NOMBRE_INSTRUCCIÓN [Evaluador 1] [Evaluador 2] .....

Hay instrucciones que se ejecutan en un solo ciclo , pero hay algunas que requieren varios ciclos hasta que completan su ejecución.

5.3.3.1.1 La instrucción **NOTHING**

Formato: NOTHING

Como su propio nombre indica esta instrucción no hace nada. Es la instrucción que se asigna por defecto a una entidad. Se utiliza cuando queremos que nuestra entidad no haga nada ante un determinado evento.

5.3.3.1.2 La instrucción **SAY**

Formato: SAY Línea

Esta instrucción se utiliza cuando queramos que nuestra entidad diga una frase. La frase que se va a mostrar es elegida del archivo de textos Texts.txt que se encuentra en el directorio Data del motor.

Parámetros:

1. La línea de texto del archivo Texts.txt que se va a mostrar.

5.3.3.1.3 La instrucción **GOTO**

Formato: GOTO X Y Velocidad

Esta instrucción se utiliza cuando queramos que nuestra entidad se mueva a una determinada posición. La instrucción nos permite también determinar la velocidad a la que se moverá. Esta instrucción se ejecutará el número de veces que haga falta hasta que la entidad alcance la posición indicada.

Parámetros:

1. Coordenada X a la que se quiere que se mueva el objeto. Este valor viene dado en coordenadas de terreno, si se quiere pasar a coordenadas de mundo debe multiplicarse por el paso del terreno.
2. Coordenada Y a la que se quiere que se mueva el objeto. Este valor viene dado en coordenadas de terreno, si se quiere pasar a coordenadas de mundo debe multiplicarse por el paso del terreno.
3. Velocidad de movimiento. Este parámetro indica el número de milisegundos necesarios para que la entidad de un paso en su recorrido. Por tanto cuanto menor sea este valor, más rápido se moverá la entidad.

***5.3.3.1.4 La instrucción* WAIT**

Formato: WAIT Tiempo

Esta instrucción se utiliza cuando queramos que nuestra entidad espere una cantidad de milisegundos antes de ejecutar la siguiente instrucción. . Esta instrucción se ejecutará el número de veces que haga falta hasta que la entidad haya esperado el tiempo necesario.

Parámetros:

1. Tiempo en milisegundos que se quiere que dure la espera.

***5.3.3.1.5 La instrucción* SETVAR**

Formato: SETVAR Indice Valor

Esta instrucción se utiliza cuando queramos cambiar el valor de una variable de la memoria de trabajo.

Parámetros:

1. Indice de la variable que se quiere modificar.
2. El nuevo valor que se quiere asignar a la variable.

***5.3.3.1.6 La instrucción* END**

Formato: END

Esta es una instrucción especial que se utiliza como delimitador de final de otro tipo de instrucciones espciales, como las condicionales o las instrucciones compuestas (ver siguiente punto).

5.2.3.2 Contenedores de instrucciones

Un contenedor de instrucciones, es un tipo especial de instrucción que a su vez contienen a otras instrucciones. Si queremos que nuestra entidad realice más de una instrucción para un determinado evento, entonces deberemos asociarle un contenedor de instrucciones en vez de una instrucción simple. Un contenedor de instrucciones tiene la siguiente forma:

```
NOMBRE_CONTENEDOR
        NOMBRE_INSTRUCCIÓN_1  [Evaluador 1] [Evaluador 2] .....
        NOMBRE_INSTRUCCIÓN_2  [Evaluador 1] [Evaluador 2] .....
                .........
        NOMBRE_INSTRUCCIÓN_N [Evaluador 1] [Evaluador 2] .....
END
```

Hay instrucciones que se ejecutan en un solo ciclo , pero hay algunas que requieren varios ciclos hasta que completan su ejecución. La manera en que las instrucciones son ejecutadas depende del contenedor que se haya definido.

No hay que olvidar que los contenedores de instrucciones siguen siendo instrucciones, sólo que más complejas, por tanto en cualquier sitio que se requiera una instrucción puede colocarse también una instrucción compuesta.

***5.3.3.2.1 El contenedor* SEQ**

```
Formato:    SEQ
                    Instrucción_1 Parámetros
                    Instrucción_2 Parámetros
                    ....
                    Instrucción_N Parámetros
            END
```

Este contenedor ejecuta cada una de las instrucciones que contiene de manera secuencial, como mucho una por ciclo. Termina su ejecución cuando la instrucción Intrucción_N termina su ejecución.

***5.3.3.2.2 El contenedor* LOOP**

```
Formato:    LOOP
                    Instrucción_1 Parámetros
                    Instrucción_2 Parámetros
                    ....
                    Instrucción_N Parámetros
            END
```

Este contenedor tiene un comportamiento similar al contener SEQ (ver punto anterior) pero con una particularidad, que una vez la Instrucción_N termina su ejecución,

se vuelve de nuevo a ejecutar la Instrucción_1 y así indefinidamente. Esto quiere decir que este contenedor nunca terminará su ejecución.

5.3.3.3 Los evaluadores

Los evaluadores son tipos especiales de instrucciones que devuelven valores. Los evaluadores pueden ser usados como parámetros de otras instrucciones, y también pueden ser usados como parámetros de los evaluadores ya que ellos mismos son instrucciones.

El tipo más básico de evaluador es un número constante. Cuando nosotros en el lenguaje escribimos el número N, internamente es tratado como un evaluador cuyo valor es siempre el valor constante N.

5.3.3.3.1 El evaluador GETVAR

Formato: GETVAR Indice

Obtiene el valor de una variable de la memoria de trabajo.

Parámetros:

1. Indice de la variable que se quiere obtener.

5.3.3.4 Instrucciones condicionales

Las instrucciones de condicionales nos permiten ejecutar instrucciones en función del cumplimiento de una condición.

5.3.3.4.1 La instrucción IF

```
Formato 1:   IF Evaluador1 operador Evaluador2
                 Accion1
             ENDIF

Formato 2:   IF Evaluador1 operador Evaluador2
                 Accion1
             ELSE
                 Accion2
```

La instrucción IF lo que hace es evaluar el evaluador primero , evaluar el evaluador segundo y ejecutar la operación de comparación que indica el operador entre ambos. Los operadores admitidos son < , > , = . Si el resultado de la operación de comparación es cierto se ejecutará la acción uno. Si el resultado de la operación de comparación es falso lo que se hará dependerá del formato utilizado. Si la instrucción IF no tiene ELSE, entonces finaliza su ejecución. Si por el contrario se añade el ELSE a la instrucción entonces se ejecutará la acción dos, y tras esto finalizará su ejecución.

5.3.3.4.2 La instrucción IF2

Esta instrucción se ha añadido para poder realizar ciertas acciones que con el funcionamiento de la instrucción IF no se puede. En cuanto a sintaxis es exactamente igual que la instrucción IF, admitiendo los dos formatos. La diferencia radica en la manera de ejecutarse:

La instrucción IF como se ha visto en el punto anterior, evalúa la expresión de comparación , elige la acción a ejecutar, y termina su ejecución.

La instrucción IF2 se ejecuta infinitamente. En cada ciclo de reloj que se la llama, evalúa la expresión de comparación, y ejecuta un paso de ejecución en la acción que corresponda según la operación de comparación.

5.3.3.4.3 Diferencias entre IF *e* IF2

Para ver la diferencia entre ambas vamos a ilustrarlo con un ejemplo. Queremos que una entidad de nuestra escena esté moviéndose en una ruta hasta que por algún motivo cambie el valor de una variable, en cuyo caso queremos que se pare. El código de script habría que asociarlo al evento TICK de la entidad y sería algo así:

```
IF GETVAR 0 = 5
        LOOP
                GOTO 100 100 50
                GOTO 100 50 50
                GOTO 50 50 50
        END
ELSE
        NOTHING
```

```
IF2 GETVAR 0 = 5
        LOOP
                GOTO 100 100 50
                GOTO 100 50 50
                GOTO 50 50 50
        END
ELSE
        NOTHING
```

Este sería el codigo con las dos variantes del IF. Como vemos, la sintaxis es exactamente igual, pero vamos a ver como el primer IF no realizaría el comportamiento deseado:

El código de la izquierda haría lo siguiente: Primero evalúa la condición, que suponemos inicialmente cierta. Por tanto el siguiente paso de ejecución iría por la rama del LOOP. El LOOP es una instrucción infinita, y hemos dicho que el IF hasta que no termina la ejecución de la acción correspondiente, no termina, por tanto se quedaría encerrado para siempre en esta acción, y nuestra entidad estaría siempre moviéndose en su ruta, porque aunque se produjese un cambio en la variable 0, la operación de comparación del IF no va a volver a ser evaluada.

El código de la derecha sin embargo haría lo siguiente: Evaluará la condición, que como hemos dicho suponemos inicialmente cierta, y por tanto ejecutará un paso de ejecución en la acción de LOOP, y por tanto avanzaría un paso en su ruta. Pero sin embargo, si en algún momento el valor de la variable 0 cambia su valor, como la operación de comparación el IF2 la realiza en cada ciclo de reloj, se daría cuenta del cambio, y ejecutaría el NOTHING, por tanto la entidad dejaría de moverse.

5.4 El fichero de salida .SCN

Una vez hayamos configurado toda la escena a nuestro gusto, el último paso que nos queda es guardarla en un fichero que contenga toda la información para que después el intérprete sea capaz de interpretarla. A continuación se muestra el formato binario de los archivos de escena:

Número de bytes	Descripción	Tipo
1	Indica la longitud en bytes del campo que viene a continuación.	unsigned char
X	Nombre del fichero del mapa de alturas. Tantos caracteres como indicara el campo anterior.	char
4	Escala del mapa de alturas.	float
1	Paso del mapa de alturas.	unsigned char
1	Tamaño del parche del mapa de alturas.	unsigned char
1	Detalle del mapa de alturas.	unsigned char
1	Número de entidades que forman parte de la escena.	unsigned char
*	Esta parte se repite por cada entidad	*
1	Indica la longitud en bytes del campo que viene a continuación.	unsigned char
X	Nombre del fichero de modelo. Tantos caracteres como indicara el campo anterior.	char
1	Indica la longitud en bytes del campo que viene a continuación.	unsigned char
X	Nombre de la entidad.	char
4	Coordenada X de la entidad.	int
4	Coordenada Y de la entidad.	int
1	Código identificador de la acción asociada al evento FireAction.	unsigned char
X	Este campo es de longitud variable, ya que está en función de los parámetros requeridos por la acción asociada al evento.	X
1	Código identificador de la acción asociada al evento TickAction.	unsigned char
X	Este campo es de longitud variable, ya que está en función de los parámetros requeridos por la acción asociada al evento.	X

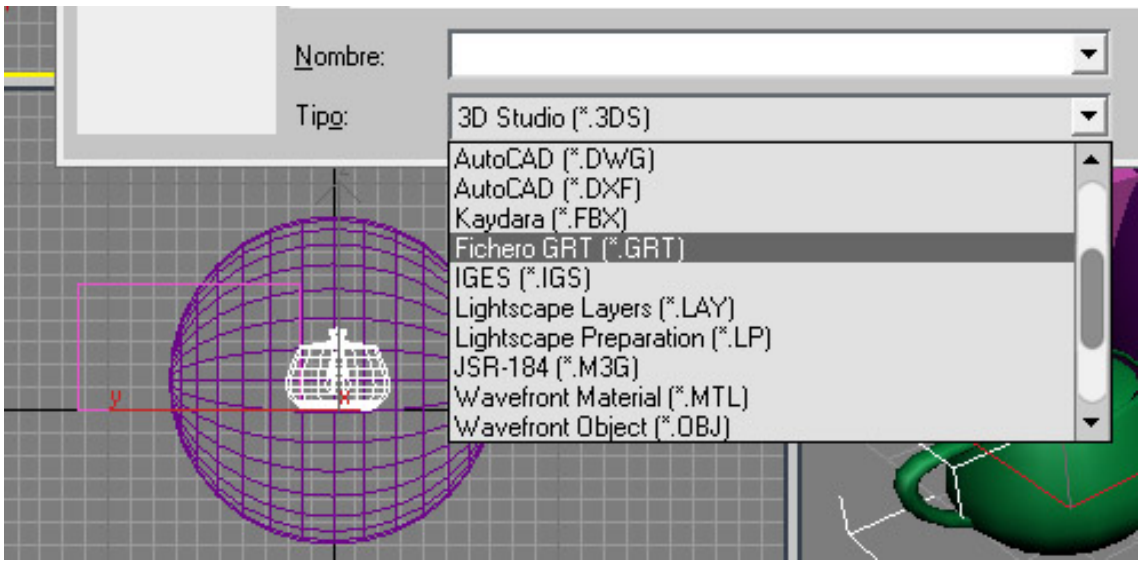
6 – Exportador de modelos

6.1 ¿Qué es el exportador?

Como se vió en el capítulo 3, que explica la representación geométrica de objetos en tres dimensiones en nuestro motor, un modelo es una estructura bastante compleja, con lo cual se necesita alguna herramienta externa de modelado en tres dimensiones para crearlo. Como la creación de una herramienta de este tipo se sale del alcance de nuestro proyecto, lo que se ha desarrollado es un plugin para el programa de modelado 3DStudio MAX. Este plugin es un exportador, y se ha desarrollado en C++ utilizando para ello la SDK para plugins de 3DStudio MAX. Este plugin se ha desarrollado con la versión 5.0 de la SDK , aunque debería ser compatible con cualquier versión superior.

6.2 Desarrollo

Para crear nuestro exportador de modelos utilizamos la clases proporcionadas por la SDK de 3DStudio MAX. Concretamente, para la realización de plugins de tipo exportador, la SDK proporciona una clase base llamada “SceneExport”. Esta clase contiene métodos para describir las propiedades del exportador (extensión del fichero, versión, autor, etc…) y un método para manejar el proceso de exportación. Este metodo, es el punto de inicio de nuestro plugin, y será llamado cuando el usuario elija desde la interfaz del programa que desea exportar un modelo con nuestro formato:



La cabecera de este método es la siguiente:

```
int SceneExporter::DoExport(const TCHAR *name,ExpInterface *ei,Interface *i,
                            BOOL suppressPrompts, DWORD options);
```

Los parámetros de esta función, relevantes para nosotros, son dos:

- const TCHAR *name: Este parámetro contiene la información del nombre del archivo que ha seleccionado el usuario para guardar el modelo.
- Interface *i: A través de esta clase vamos a poder acceder a toda la información de la escena del 3DStudio MAX.

La escena de 3DStudio esta jerarquizada en nodos. Hay un nodo raíz a partir del cual podemos acceder al resto de nodos hijos. Luego para obtener la escena, lo que vamos a hacer es ir recorriendo estos nodos en forma de árbol e ir procesando aquellos que nos interesen. La manera de acceder al nodo raíz es a través del método GetRootNode() de la clase Interface. A partir de este nodo, comenzaremos el procesamiento de la escena, que seguirá el siguiente esquema:

```
ProcessObject(node);

unsigned int childs = node->NumberOfChildren();

for(int i = 0;i < childs;i++){
        ProcessObjects(node->GetChildNode(i));
}
```

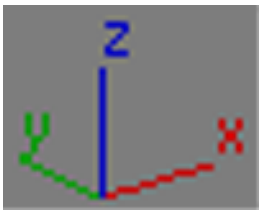
Como se puede observar, es un recorrido inorden del árbol de jerarquía. Lo primero que hacemos es procesar el nodo en el que nos encontramos, y seguidamente procesamos cada uno de los hijos que le sigan en la jerarquía.

Para el 3DStudio MAX, un nodo puede ser una gran cantidad de cosas: Objetos geométricos, luces, cámaras, helpers, objetos deformables, splines… Por tanto lo primero que debemos hacer al procesar un nodo, es ver si realmente contiene información que nos interese exportar. Para esto la SDK nos da unos métodos que nos permiten saber a qué clase pertenece un nodo. Nosotros nos vamos a centrar en exportar objetos de las clases GEOMOBJECT_CLASS_ID o LIGHT_CLASS_ID, es decir, objetos geométricos y luces.

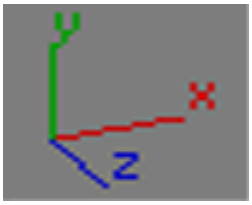
6.2.1 Procesando un objeto geométrico

Lo primero que vamos a querer obtener de un objeto geométrico es su matriz, para poder representar su posición y su orientación. Como se vió en el capítulo de la representación de objetos 3D de nuestro motor, nuestros objetos pueden tener más de una matriz si queremos que estén animados, por tanto en este paso no vamos a obtener una sola matriz, si no que vamos a obtener una matriz por cada cuadro de animación de la escena. Pero no podemos procesar la matriz de transformación del objeto tal cual nos

la proporciona el 3D Studio debido a una particularidad: nuestro sistema de coordenadas no coincide con el suyo:



Sistema de coordenadas de 3D Studio Max



Sistema de coordenadas de nuestro motor

Como se puede ver en ambos dibujos, lo que para el 3D Studio MAX es la Z, para nosotros es la Y, y lo que para el 3D Studio MAX es la Y, para nosotros es la Z negada. Por tanto para pasar de un sistema a otro, deberemos intercambiar la segunda fila de la matriz por la tercera, y además negar esta última.

También hemos tenido en cuenta otro detalle. Debido a que el factor de escala de una matriz nos era un poco engorroso a la hora de realizar ciertos cálculos, como por ejemplo en la detección de colisiones, ya que teníamos que medir distancias y preferimos tener todos los sistemas con el mismo factor de escala, antes de guardar la matriz también vamos a eliminar el factor de escala. Esto lo hemos hecho de la siguiente manera: la SDK nos proporciona una macro llamada "decomp_affine", que nos descompone una matriz en sus transformaciones afines básicas (rotación, escalación y traslación). Pues con esta macro sacamos el factor de escala de nuestra matriz, y le aplicamos a ella el factor de escala inverso, con lo cual, nos quedara una matriz con factor de escala unidad. Pero para no obtener valores incorrectos, si hemos sacado el factor de escala de la matriz, debemos tenerlo en cuenta a la hora de procesar los vértices como se verá más adelante.

Una vez tenemos obtenidas las matrices pasamos a procesar los vértices. Leemos cada uno de los vértices con las particularidades que hemos dicho antes, es decir:

```
Para i = 0 hasta numero_de_vertices hacer
        vertice[i].x  = malla.vertice[i].x * factor_de_escala_en_X;
        vertice[i].y  = malla.vertice[i].z * factor_de_escala_en_Y;
        vertice[i].z  =  - malla.vertice[i].y * factor_de_escala_en_Z;
FPara
```

Una vez procesados todos los vértices del objeto pasamos a las caras. Lo que hacemos simplemente es copiar los índices de los vértices de las caras que nos proporciona el 3DStudio. Hacemos igual con las caras de textura y con los vértices de textura (coordenadas u,v).

El siguiente paso es obtener el material que tiene asociado el objeto y sus propiedades. A través del método del nodo GetMtl() obtenemos un puntero al material que utiliza el objeto , o NULL si no tienen ningún material asociado.

Si el objeto no tiene material asociado, accedemos a la información del color de la malla para poder darle color de alguna manera. Esto se consigue a través del método GetWireColor(). El 3DStudio nos proporciona este color en el rango de 0 a 255, pero nosotros por comodidad para su uso en openGL, cambiamos esta información para amoldarla al rango 0 a 1.

En caso de que el objeto si tenga material asociado, con la llamada GetMtl() habremos obtenido un puntero a la clase Mtl , que es la que utiliza la SDK del 3DStudio para representar las propiedades de los materiales. Las propiedades que nosotros necesitamos y que vamos a obtener son:

- *Componentes de reflexión ambiental*: obtenidas a partir de Mtl.ambientRGB.
- *Componentes de reflexión difusa*: obtenidas a partir de Mtl. diffuseRGB.
- *Fichero de textura*: Este parámetro no es obligatorio, ya que el material puede tener mapa de textura o no.

Llegados a este punto, tenemos todos los valores necesarios para representar un objeto en nuestro motor 3D.

6.2.1 Procesando una luz

3DStudio Max define muchas clases de luces, pero ahora mismo el exportador sólo exporta luces posicionales. Por lo tanto si nos encontramos con una luz, únicamente vamos a exportar su matriz de animación que nos dará su posición en cada instante de tiempo.

6.3 El fichero de salida .GRT

Una vez se haya recorrido toda la jerarquía de nodos de la escena, el último paso que nos queda es guardar toda esta información en un fichero para que después el motor sea capaz de interpretarla. A continuación se muestra el formato binario del archivo:

Número de bytes	Descripción	Tipo
4	Número de materiales que utiliza el modelo.	unsigned int
*	Esta parte se repite por cada material.	*
4	Componente roja de reflexión ambiental.	float
4	Componente verde de reflexión ambiental.	float
4	Componente azul de reflexión ambiental.	float

4	Componente roja de reflexión difusa.	float
4	Componente verde de reflexión difusa..	float
4	Componente azul de reflexión difusa.	float
1	Indica la longitud en bytes del campo que viene a continuación.	unsigned char
X	Nombre del fichero de la textura asociada al material.	char
*	Fin de materiales	*
1	Número de cuadros de animación del modelo.	unsigned char
1	Velocidad de la animación del modelo dada en cuadros por segundo.	unsigned char
4	Número de objetos geométricos del modelo.	unsigned int
*	Esta parte se repite por cada objeto geométrico del modelo.	*
**	Esta parte se repite por cada cuadro de animación del objeto geométrico.	*
48	Los 12 valores para representar una matriz.	float
**	Fin de cuadros de animación.	*
4	Identificador de material. -1 si no utiliza material.	int
4	Componente roja del color de la malla. Sólo aparece si el identificador de material es -1.	float
4	Componente verde del color de la malla. Sólo aparece si el identificador de material es -1.	float
4	Componente azul del color de la malla. Sólo aparece si el identificador de material es -1.	float
4	Número de vértices del objeto geométrico.	unsigned int
**	Esta parte se repite por cada vértice del objeto geométrico.	**
4	Coordenada x del vértice.	float
4	Coordenada y del vértice.	float
4	Coordenada z del vértice.	float
**	Fin de la lista de vértices.	**
4	Número de caras del objeto geométrico.	unsigned int
**	Esta parte se repite por cada cara del objeto geométrico.	**
4	Indice del primer vértice de la cara.	unsigned int
4	Indice del segundo vértice de la cara.	unsigned int
4	Indice del tercer vértice de la cara.	unsigned int
**	Fin de la lista de caras	**

4	Número de vértices de textura del objeto geométrico.	unsigned int
**	Esta parte se repite por cada vértice de textura del objeto geométrico.	**
4	Coordenada U del vértice de textura.	float
4	Coordenada V del vértice de textura.	float
**	Fin de la lista de vértices de textura.	**
**	Las caras de textura sólo aparecen si el número de vértices de textura es mayor que cero. Hay tantas caras de textura como caras normales.	**
4	Indice del primer vértice de la cara de textura.	unsigned int
4	Indice del segundo vértice de la cara de textura.	unsigned int
4	Indice del tercer vértice de la cara de textura.	unsigned int
**	Fin de la lista de caras de textura.	**
*	Fin de la lista de objetos geométricos.	*
4	Número de luces del modelo.	unsigned int
*	Esta parte se repite por cada luz del modelo.	*
**	Esta parte se repite por cada cuadro de animación de la luz.	**
48	Los 12 valores para representar una matriz.	float
**	Fin de la lista de cuadros de animación.	**
*	Fin de la lista de luces	*

7 - Multijugador

Una gran parte del desarrollo del juego se centra en la creación de un módulo que gestione una partida de forma concurrente entre varios usuarios. Es decir, que desde un equipo en el que estemos ejecutando nuestro juego, podremos observar e interactuar con otros jugadores conectados a la partida a través de una red. La consistencia de los datos manejados por los distintos clientes se mantiene gracias a un equipo que actúa como servidor, sincronizando los datos y reenviando paquetes de unos clientes a otros.

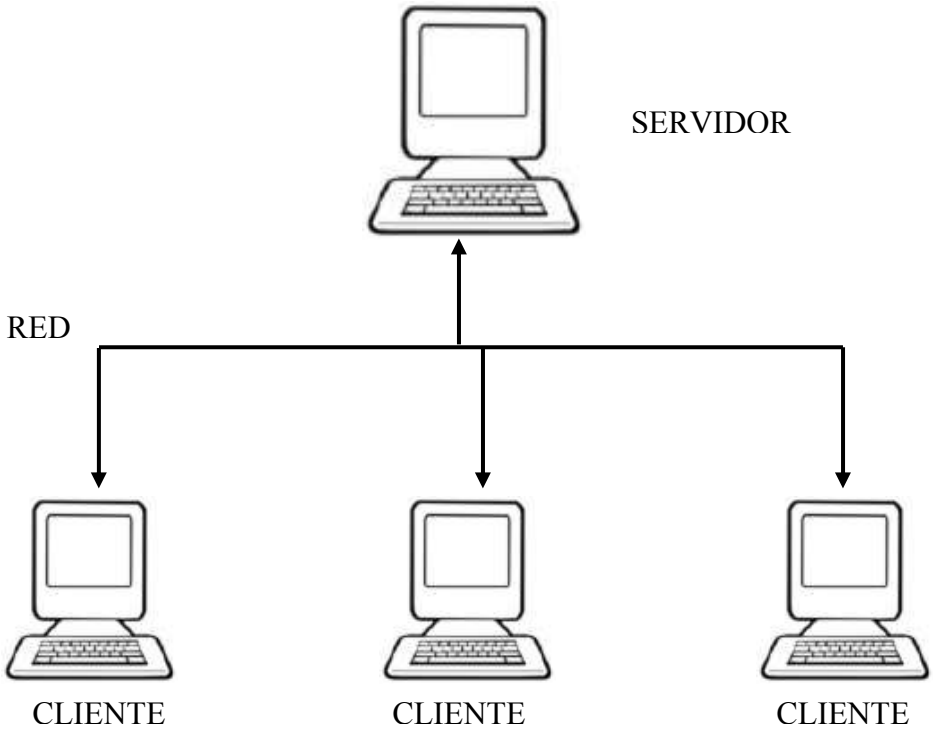
El servidor recibe todos los paquetes de los clientes, enviados en forma de petición. En el caso de que alguno de esos paquetes recibidos por el servidor esté destinado a otro cliente, el servidor se lo reenvía, actuando como un puente de comunicaciones. El resto de los paquetes son procesados por el servidor (por ejemplo, la notificación de que un nuevo cliente quiere conectarse a la partida) y, si es necesario, se notifica a todos los clientes que ese paquete ha sido procesado, mediante un mensaje de *broadcast*.

Cada paquete recibido por un cliente, es procesado como un evento. Por ejemplo, si al cliente A está en la partida, y entonces un nuevo cliente B se conecta, A recibe un evento de notificación de conexión. En ese momento, el programa cliente de A generará un nuevo modelo asociado al cliente B y lo mostrará por pantalla. De esta forma, un jugador puede saber de la existencia de otros jugadores en 'tiempo real'.

Algunos de los eventos que genera el servidor y reciben los clientes, no tienen su origen en una anterior petición de un cliente, sino que el servidor los genera espontáneamente. Por ejemplo, para que todos los jugadores conozcan en todo momento la posición en el escenario de sus compañeros, el servidor envía cada cierto tiempo eventos de posicionamiento a todos los clientes, informándoles de la posición de cada jugador en el mapa. Por tanto, si en una partida tenemos tres clientes A, B y C, el servidor enviará 9 mensajes de posicionamiento, 3 a cada cliente.

Si nos fijamos en lo anterior, el servidor debe tener información totalmente actualizada de todos los cliente. Por eso, algunas de las peticiones de los cliente al servidor son en realidad mensajes que informan a éste sobre el estado actual del cliente, como por ejemplo, su posición.

La siguiente figura ilustra la estructura de red:



La base de la estructura multijugador será el envío de paquetes de información a través de *sockets*. Los sockets son relativamente sencillos de manejar, y gracias a la librería de libre distribución "SDL_Net", se consigue que los sockets funcionen en multitud de sistemas operativos, abstrayéndonos de la implementación de detalles de bajo nivel.

Tanto servidor como clientes inician SDL al arrancar (con SDL_INIT(), como veremos más adelante). Aunque SDL puede también puede utilizarse para manejar gráficos, dispositivos de entrada, etc., nosotros nos limitaremos a usar SDL_Net para el manejo de paquetes de red, SDL_Thread para crear hilos de proceso, y SDL_Mixer para reproducir música durante la partida.

7.1 Protocolos

En un primer momento, se implementó la comunicación de red mediante conexiones TCP. Este tipo de conexión se caracteriza por su fiabilidad (no se pierden paquetes, y estos siempre llegan en orden), pero este aumento de la seguridad tiene un impacto sobre la eficiencia, ya que los paquetes pueden tardar demasiado en llegar a su destino. Por tanto, después de haber hecho numerosas pruebas de eficiencia en la red, se optó por sustituir las conexiones TCP por el envío de paquetes UDP. En este caso, al ser un servicio no orientado a conexión, los paquetes se envían sueltos por la red, con lo cual pueden llegar desordenados a su destino, o no llegar. Sin embargo, la comunicación es más rápida que con TCP y por eso la mayor parte de videojuegos multijugador comerciales están orientados a UDP.

Las funciones de SDL_Net que utilizamos tienen la siguiente sintaxis:

- SDL_Init() y SDLNet_Init(): Inicialización de los componentes de SDL_Net

- SDLNet_UDP_Open(int puerto): Abre un nuevo socket en el puerto especificado, para poder enviar o recibir paquetes a través de él.

- SDLNet_UDP_Bind(UDPsocket sock, int canal, IPaddress *address): Dentro de un mismo socket, podemos enviar información por varios canales. Con la función SDLNet_UDP_Bind() podemos asociar a cada uno de estos canales del socket una dirección IP remota, y así hacemos que el servidor tenga un canal asociado a cada cliente conectado a él.

- SDLNet_UDP_Send(UDPsocket sock, int canal, UDPpacket *paquete): Envía un paquete de información a través del canal 'canal' del socket 'sock'. Se supone que el canal ha sido previamente asociado a una dirección IP.

- SDLNet_UDP_Recv(UDPsocket sock, UDPpacket *packet): Comprueba si en el socket 'sock' hay información lista para recibir, y en caso afirmativo la recibe y la introduce en el paquete 'packet'.

- SDLNet_AllocPacket(int tam): Reserva una porción de memoria de tamaño 'tam' para asignársela a un paquete.

- SDLNet_ResolveHost(IPaddress *address, char *host, int puerto): Dado el nombre de un equipo en la red ('host'), generamos la dirección IP que le corresponde.

Estas funciones nos permitirán un control suficiente sobre los datos que cada equipo necesite enviar o intente recibir. Aunque el puntero de datos del paquete sea de tipo Uint8, mediante una simple conversión de datos podemos introducir información de cualquier tipo, desde cadenas de caracteres hasta números en punto flotante.

7.2 Procesos de red

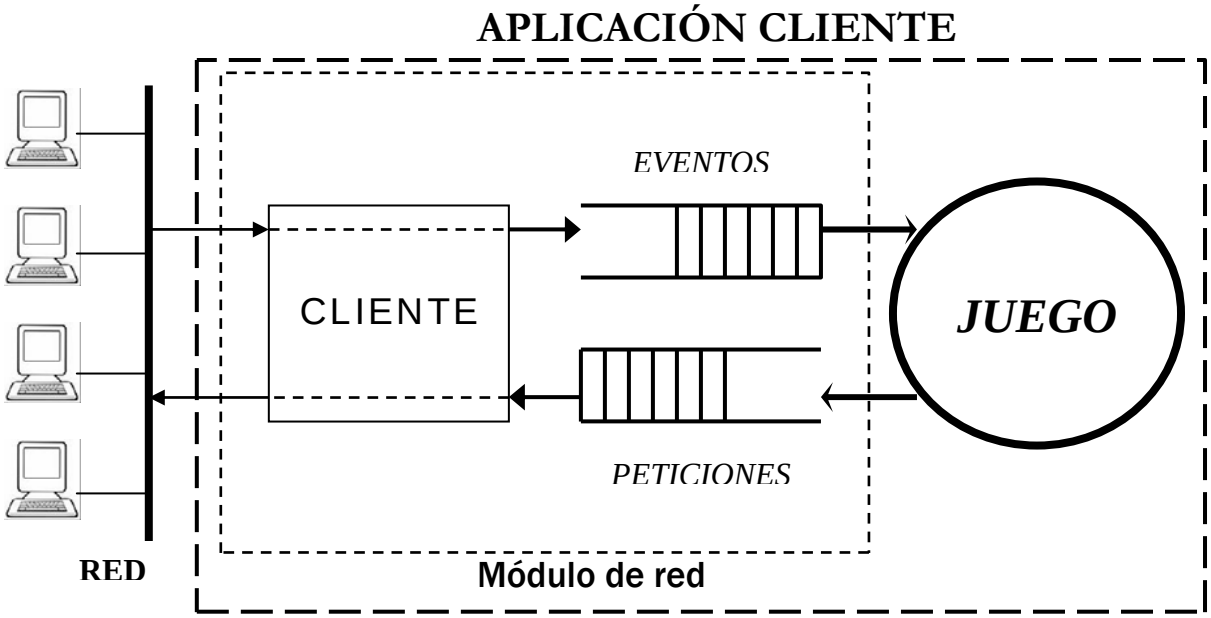
Para aumentar aún más la eficiencia de la red, tanto los clientes como el servidor realizarán el envío y recepción de paquetes desde un hilo de proceso independiente. Para crear un hilo, utilizamos la función "SDL_CreateThread()". Al llamar a esta función, tenemos que pasarle como parámetro la función o procedimiento que se ejecutará dentro del nuevo hilo. Generalmente está función debería consistir en un bucle *while*.

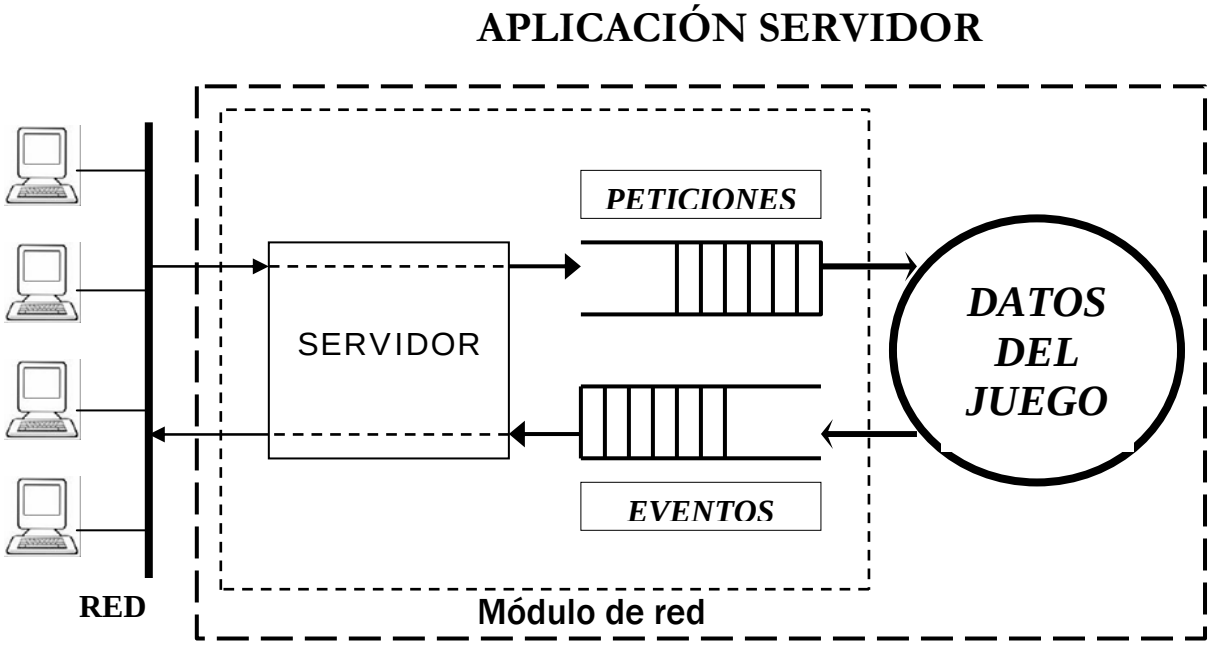
Hemos conseguido abstraer toda la tecnología de red en nuestro juego introduciendo todo el código de comunicaciones en las clases **Cliente** y **Servidor**. Estas clases se encargarán de crear el hilo de proceso mencionado antes, el cual a su vez ejecuta una función que compueba contínuamente el contenido de dos colas, "colaPeticiones" y "colaEventos":

→ El hilo del servidor comprueba si le han llegado paquetes, y en caso afirmativo, los introduce en "colaPeticiones". El hilo del cliente los introduce en "colaEventos".

→ El hilo del servidor también comprueba si hay datos listos para enviar en "colaEventos", y en caso afirmativo, los envía por la red. El hilo del cliente envía los paquete de "colaPeticiones".

Vemos que estas dos colas contienen eventos creados por el servidor y peticiones creadas por los clientes. Las colas son el único nexo de unión entre el módulo de red del juego y el simulador de juego propiamente dicho, es decir, entre el código de red y el código interno del juego. El módulo de red no llama a ninguna función del código de juego y viceversa. Lo que ocurre es que ambos módulos comprueban las colas y las van rellenando y vaciando con mensajes de red (en forma de eventos y peticiones).





De esta forma, el juego en sí sólo tiene que meter o sacar los paquetes de las dos colas mencionadas. Como son varios procesos los que acceden a las colas de forma concurrente, estas disponen de semáforos (*mutex*) que garantizan la exclusión mutua.

7.3 Paquetes de red

La estructura de un paquete UDP es:

```
typedef struct {
        int channel;            /* Canal de origen/destino del canal */
        Uint8 *data;            /* Datos del paquete */
        int len;                /* Tamaño de los datos del paquete */
        int maxlen;             /* Tamaño del buffer de datos */
        int status;             /* Estado del paquete tras el envío */
        IPaddress address;      /* Dirección de origen/destino del paquete */
} UDPpacket;
```

Los datos que nos interesan son los que están en el campo 'data'. Lo que metemos en 'data' es la información que hay en las colas, es decir, elementos del tipo ***Evento***. A efectos del programa, un *Evento* podría considerarse como un paquete de información del juego que va contenido dentro de un paquete de red (UDPpacket). Tanto la cola de peticiones como la cola de eventos del juego contienen elementos de este tipo *Evento*.

La estructura de un Evento es:

```
typedef struct {
        TipoEvento codigo;   // Identifica a un tipo de evento de los demás
        ID origen;           // Identificador del ente que genera la accion
        ID destino;          // Identificador del ente que recibe la accion
        ID para;             //Ident. del jugador que recibirá el mensaje(-1 broadcast)
        long int tiempo;     // Momento en el que se produce el evento
        int args[50];        // Información que cada evento necesita
} Evento;
```

El significado de cada campo de un *Evento* es el siguiente:

- *codigo*: Nos dice qué tipo de evento es. 'TipoEvento' es un tipo enumerado cuyos valores son EVENTO_CONECTARCLIENTE, EVENTO_MOVER, etc.
- *origen*: Identificador del ente del juego que ha generado el Evento.
- *destino*: Identificador del ente que recibirá el Evento.
- *para*: Identificador del cliente o jugador que recibirá el Evento.
- *tiempo*: Es un número entero largo que contiene el instante de tiempo en el que se creó el *Evento*.
- *args*: Es una serie de datos que variará según el tipo de Evento que sea.

Veamos algunos ejemplos de construcción de Eventos:

1- Dar un paso de movimiento (se mueve el jugador con ID=5):

codigo=EVENTO_MOVER
origen=5
destino=5
para=-1 → Si 'para' es -1, el mensaje se reenviará a todos los cliente(broadcast)
tiempo=34349202
args[0]=64 → Código ASCII de la tecla de movimiento que se ha pulsado
args[1]=false → True – La tecla ha sido presionada; False – Ha sido soltada
args[2]=10 → Nº de ticks de movimiento que debe dar el personaje
args[3]=320 → Máscara de todas las teclas que hay pulsadas actualmente

2- Mandar un mensaje de chat (por parte del jugador con ID=6):

codigo=EVENTO_MENSAJECHAT
origen=6
destino=0 → Si es 0, el evento no está dirigido a un jugador, sino al juego en sí
para=-1
tiempo=10003065
args[0]="Hola, qué tal?"

3- Efectuar un ataque (el jugador ID=5 contra el jugador ID=8):

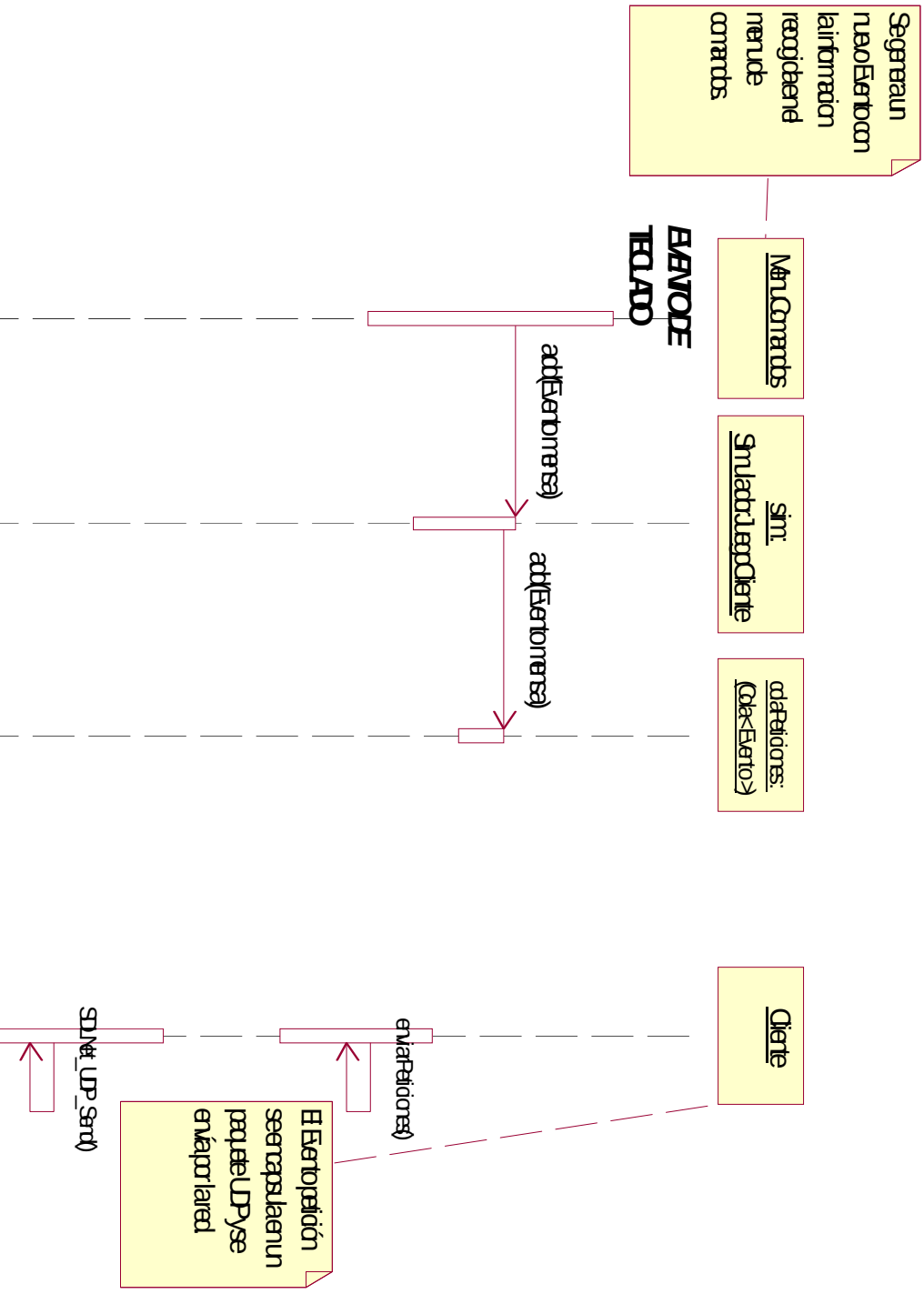
codigo=EVENTO_BATALLA_ATACAR
origen=5
destino=8
para=8
tiempo=4953454
args[1]=getJugador(5)->getFuerza()

4- Desconectarse de la partida, porque queremos salir o porque nos han matado (ID=9):

codigo=EVENTO_DESCONEXION_CLIENTE
origen=9
destino=0 → Si es 0, el evento no está dirigido a un jugador, sino al juego en sí
para=0
tiempo=45647734

De esta forma, cada vez que una instancia del programa cliente necesite enviar información a otro cliente o al servidor, debe generar un Evento, rellenando todos sus campos con los valores correctos, e introducirlo en la cola de Peticiones. Entonces el proceso encargado de las comunicaciones se encargará de recoger el Evento, encapsularlo dentro de un paquete UDP, y finalmente enviarlo por la red. Veamos un ejemplo de las llamadas a función realizadas en el diagrama a continuación:

Ejemplo: Acción de ataque en una batalla 1 contra 1:



7.4 Cómo se configura una partida multijugador

En este apartado vamos a ver los pasos que se deben tomar al arrancar el juego para conseguir vivir una experiencia en multijugador. Todas las aplicaciones que participan en este proceso (y sus respectivos *threads*) deben comunicarse entre sí de forma fiable y lógica. Sólo así se tendrá una conexión estable y que mantenga la consistencia de los datos.

El punto de partida está, logicamente, en el servidor. Al ser la pieza clave del sistema multijugador, será el primero en ejecutarse, y el primero también en inicializar sus conexiones de red. Estos son los pasos que se siguen:

→ Crear el objeto de la clase Servidor que comentamos anteriormente, aquel que se encarga de recibir y enviar paquetes UDP entre la red y las colas. El objeto Servidor configura la red en el equipo en el que se ejecuta, abriendo un puerto de comunicaciones (el 10001) y creando un nuevo socket. El socket estará asociado al puerto que hemos abierto, con lo que los paquetes que viajen por este puerto podrán ser accedidos por la aplicación servidor a través del socket.

→ Después de configurar los sockets, el servidor hace una llamada a "activarRed()", con lo que se genera un nuevo hilo de proceso con una doble función: comprobar el socket por si han llegado nuevos paquetes, y enviar por la red (es decir, por el socket) todos los Eventos que el servidor vaya generando. Este hilo de proceso permanece activo hasta que se desactive el servidor.

Una vez activado el servidor, cualquier cliente puede iniciar una "sesión" y comenzar a jugar. Pero antes de poder jugar es necesario establecer algún vínculo o conexión con el servidor.

Tras haberse iniciado todos los componentes gráficos y de audio, se genera un nuevo objeto Cliente, de características similares al objeto Servidor anterior, pero cuyas funciones son las contrapuestas a las de este último. Es decir, recibe los Eventos generados por el servidor y los mete en la cola de eventos, y a su vez envía los elementos que van llegando a la cola de peticiones por la red.

Para que el objeto Cliente sea capaz de hacer todo esto, primero debe conectarse al servidor. Veamos cómo hacerlo:

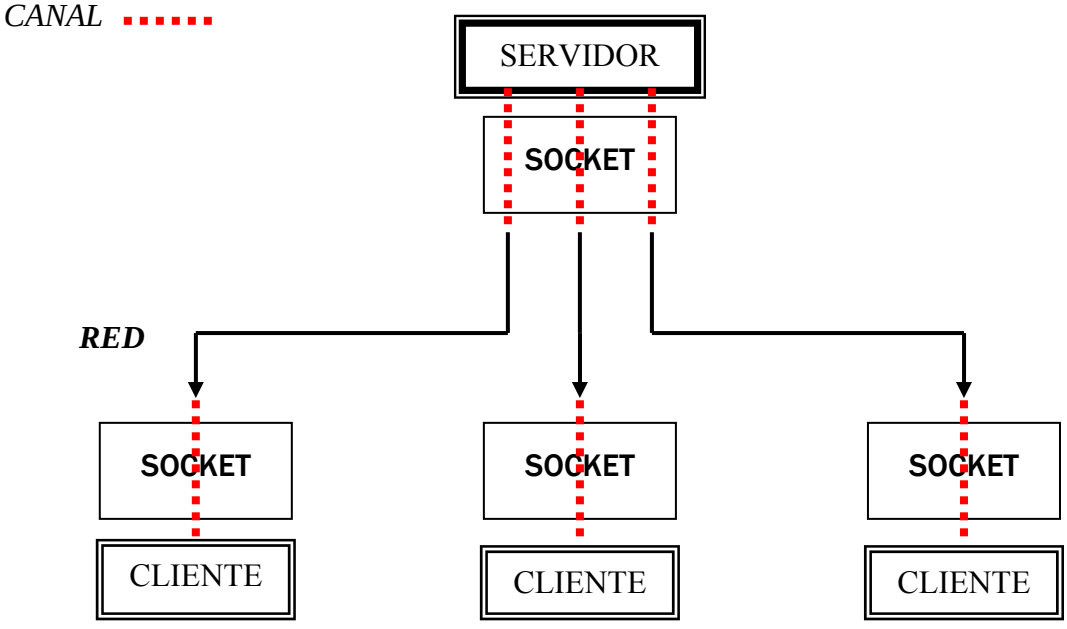
→ El cliente abre un puerto de comunicaciones y crea un nuevo socket asociado a él.

→ El cliente hace una llamada a la función *"SDLNet_UDP_Bind()"*, pasándole como parámetros el nuevo socket y la dirección IP del equipo donde está corriendo el servidor. Esta dirección IP la debemos conocer de antemano, o al menos el nombre de red del equipo al que corresponde, si estamos trabajando en una red de área local (LAN). La llamada a la función devuelve un entero, que corresponde al canal del socket asociado a la nueva conexión. En adelante, el cliente enviará todos sus paquetes a través de este canal del socket.

→ El cliente envía su primer paquete, que en nuestro caso contiene un *Evento* con el código "EVENTO_CONECTARCLIENTE".

→ El servidor recibe el intento de conexión por parte del cliente. Entonces, al igual que se hizo en el cliente, llamamos a *"SDLNet_UDP_Bind()"* pasándole como parámetros el socket del servidor y la dirección IP del paquete que acabamos de recibir (es decir, la IP del cliente). Ahora también obtenemos un identificador de canal. El servidor creará un nuevo objeto Jugador para el cliente que se acaba de conectar, y copiará en este objeto el canal obtenido. De esta forma, el servidor mantiene una lista de todos los cliente conectados, junto con el identificador de canal que cada cliente tiene asignado en el socket servidor.

Después de hacer estos pasos, el servidor está conectado a todos los clientes y cada cliente está conectado al servidor. En el servidor habrá generalmente varias conexiones simultáneas, por lo que se utilizarán varios canales del socket (cada uno asignado a un cliente). Los clientes en cambio sólo utilizan un canal de su socket.



7.5 Consideraciones sobre la capa de red

Para conectar nuestro cliente a una partida en un servidor de juego, debemos conocer la dirección IP del servidor. Sólo así podremos establecer contacto. En el caso de redes de área local (LAN), al ser un entorno reducido, generalmente tendremos un mayor control sobre él. Si conocemos el nombre que el equipo servidor ostenta en la red, podemos utilizar este nombre en lugar de la dirección IP.

Para un ámbito mayor, como puede ser Internet, debemos pensar a gran escala. En primer lugar, la aplicación servidor del juego tendrá que estar alojada en algún servidor remoto que tengamos o que hayamos contratado. Lo segundo y más importante, es que los paquetes tardarán bastante más en llegar a sus destinatarios, por lo que el rendimiento del juego se verá degradado. De hecho, al ser la comunicación mediante paquetes UDP, también aumenta la posibilidad de que se pierdan paquetes a medio camino y haya que reenviarlos.

En nuestras pruebas de implementación, hemos trabajado siempre dentro de redes LAN. Sería posible la ejecución del juego en Internet, pero como acabamos de ver, tiene sus inconvenientes.

8 – Conclusiones

Desarrollando este proyecto hemos podido observar la cantidad de dificultades que se presentan a la hora de desarrollar un videojuego, ya que hoy en día ésta es un área de la informática donde se aplican todo tipo de conocimientos: gráficos, sonido, red, inteligencia artificial... A todo esto hay que añadir la dificultad de crear una historia interesante así como todo el desarrollo artístico, lo que implica que para el buen término de un proyecto de estas características sea necesario un equipo más amplio y multidisciplinar que el nuestro.

Participar en un proyecto de este tipo nos ha permitido aprender mucho sobre el esfuerzo y la organización que se requieren para hacer un videojuego. Al principio pretendíamos introducir muchos requisitos para hacer que el juego fuera más complejo y entretenido. Pronto nos dimos cuenta de que, a pesar de la organización y el reparto de trabajo, muchos aspectos de la implementación que parecían triviales (como mostrar un menú de comandos o animar el movimiento de un modelo) costaban más trabajo del esperado. De hecho, ya pudimos comprobar en la asignatura de Informática Gráfica lo laborioso que puede resultar generar desde cero una escena con su iluminación, cámaras, objetos, materiales… Por eso decidimos no detenernos demasiado tiempo en aquello que no fuese realmente importante.

9 – Bibliografía

[TP] **Focus on 3D Terrain Programming**. Trent Polack. PREMIER PRESS. 2002

[RTOAM] Duchaineau, Wolinsky, Sigeti, Millery. **ROAMing Terrain: Real-time Optimally Adapting Meshes** http://www.llnl.gov/graphics/ROAM/

[BO] Willem H. de Boer **Fast Terrain Rendering Using Geometrical MipMapping.**. http://www.flipcode.com/articles/article_geomipmaps.shtml

[RG] Robert Nagy. **Ray Lighting. Creating realistic shadows for terrain**. http://www.gamedev.net/reference/programming/features/raylighting/

[HA] . Hawkins, Astle **OpenGL game programming**. MUSKA LIPMAN/PREMIER. 2002.

[GOTT] S. Gottschalk. **Collision Queries using Oriented Bounding Boxes**. Ph.D. thesis, Univ. North Carolina. Dept. Computer Science, 2000.

[CG] Foley, Van Dam, Feiner y Hughes.**Computer Graphics**. Addison-Wesley Professional; 2 edition (August 4, 1995)

[OGL] OpenGL architecture review board.**OpenGL Programming Guide.** *2004 Book News, Inc., Portland, OR*

[MASS] Thor Alexander**. Massively multiplayer game development.**. Ed. Charles River Media. 2003

[EBE01] David M. Eberey**. 3d Game Engine Design**. Morgan Koufmann, 2001

Se autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, (Gustavo de Santos García, Jose Antonio García Gimeno y Roberto Blanco Ancos) tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

Fecha:

Firmado: