Sistemas Informáticos
Curso 2003-2004

Sistema de chat 3D

Autores:

Javier Blázquez de Miguel
Gonzalo de Santos García
Gonzalo Fernández Hernández

Dirigido por:

Purificación Arenas Sánchez

Facultad de Informática
Universidad Complutense de Madrid

Resumen

El presente trabajo describe el diseño y la implementación de un sistema de conversación multi-usuario que muestra una representación en tres dimensiones de los usuarios y del escenario. El proyecto toma ideas del sistema de chat IRC y de juegos online masivos (MMORPG).

El objetivo principal del proyecto consiste en desarrollar un motor gráfico capaz de hacer una representación eficiente de escenarios complejos, así como de permitir el movimiento por mapas no triviales. El usuario dispone de una serie de animaciones que permiten expresar ciertas emociones prefijadas, lo que proporciona una mejor experiencia y una mayor interacción. El sistema de red implementa un modelo cliente-servidor y está diseñado para minimizar los efectos visuales que suelen aparecer en simulaciones en tiempo real que se ejecutan en redes de alta latencia como Internet.

Abstract

This work presents the design and implementation of a multi-user online conversation system which displays a three-dimensional representation of the participants and their surroundings. It borrows concepts from the Internet Relay Chat (IRC) system and Massively Multiplayer Online Role-Playing Games (MMORPG).

The main goal of the project concerns with the development of a graphics engine that supports the efficient representation of complex indoors scenes, as well as allowing the exploration of non-trivial environments. Social interaction is enhanced by associating animated sequences to a predefined set of emotions, which play an important role in the immersive experience. The networking system implements a client-server model and is designed to minimize the visual artifacts characteristic of real-time simulations running on non-ideal high-latency networks like the Internet.

Palabras clave

Chat, OpenGL, Motor 3D, Árbol BSP, Detección de Colisiones, Cálculo de Visibilidad, Algoritmos Genéticos, Sincronización Time-Warp, Formato ASE, Cliente-Servidor

Keywords

Chat, OpenGL, 3D Engine, Binary Space Partitioning, Collision Detection, Visibility Determination, Genetic Algorithms, Time-Warp Synchronization, ASE File Format, Client-Server

TABLA DE CONTENIDOS

0 - INTRODUCCIÓN

El documento está organizado de la siguiente manera.

En primer lugar se presenta la especificación de la aplicación. Los capítulos 1.1 y 1.2 corresponden a la especificación funcional y a los casos de uso considerados, respectivamente. Esta especificación describe un sistema de chat muy completo, con todo tipo de funcionalidades inspiradas en el sistema IRC. La mayor parte de estas funcionalidades se consideraron de baja prioridad y se decidió no implementarlas. El capítulo 1.3 describe por tanto el alcance final del proyecto.

El segundo capítulo describe el diseño del proyecto. Por un lado, la sección 2.1 describe el diseño inicial que se consideró, diseño que se construyó intentando en la medida de lo posible dar cabida a todas las funcionalidades consideradas en la especificación. Este diseño no se llegó a implementar en su totalidad. La sección 2.2 describe el diseño asociado a la aplicación entregada.

Finalmente, el tercer capítulo corresponde a la sección de apéndices donde se detallan los demás temas que no pertenecen estrictamente a los capítulos mencionados.

Un poco de historia sobre los chats

La historia de los chats comenzó allá por el final de la década de los setenta, con el desarrollo del sistema operativo unix.

Unix fue un sistema operativo diseñado en 1969 por Ken Thompson, Dennos Ritchie y Rudd Canaday. En principio, se llamaba unics (información uniplexada y sistema de computación). Debido a la portabilidad que le daba estar escrito en C, el sistema operativo se extendió con rapidez, y se convirtió en un lenguaje potente para redes aprovechando su capacidad multiusuario.

El hecho de que el sistema operativo soportase varios usuarios (algo novedoso), provocó la necesidad de comunicación entre ellos. Al principio mediante correo, pero más tarde se desarrolló un comando para poder mantener conversaciones en tiempo real. Hablamos del comando talk, el precursor de todos los chats.

Talk

Talk es un programa que permite a dos usuarios en el sistema comunicarse escribiendo en el teclado. Al invocar `talk` la pantalla se divide en dos partes, cada una correspondiente a uno de los usuarios. Ambos pueden escribir simultáneamente, y ambos ven la salida en su parte correspondiente de la pantalla.

Pero este sistema que permitía a dos usuarios comunicarse de forma local respecto al mismo ejemplar de sistema operativo corriendo, quedó desfasado con el surgimiento de las grandes redes de área extensa (WAN). Por ello se desarrollaron soluciones como las noticias y la red de chat.

Noticias

Se llaman noticias a una serie de mensajes electrónicos(llamados artículos) que, agrupados por tema, los usuarios podían mandar de forma que le llegase al resto de usuario. Aunque las noticias se siguen usando en internet, son más famosas las listas de correo.

La idea de noticias en la red nació en 1979 cuando dos estudiantes graduados, Tom, Truscott y Jim Ellis, pensaron en el uso de UUCP para conectar las máquinas con la finalidad de intercambiar información entre los usuarios de Unix. Instalaron una pequeña red compuesta de sólo tres máquinas en Carolina del Norte. Así nació el primer servidor de noticias : Usenet.

El primer chat de área extensa

En verano de 1988, Jarkko Oikarinen trabajaba en el departamento de procesamiento de información de la Universidad de Oulu. Su función era administrar el servidor SUN del departamento, tarea que no requería excesivo tiempo, de manera que empezó a escribir un programa de comunicaciones que le permitiera hacer su BBS un poco más fácil de usar por sus usuarios.

Estos BBS eran ordenadores dedicados conectados a una línea de teléfono también dedicada donde empresas ofrecían soporte a sus clientes y trabajadores, particulares daban servicios gratuitos de ficheros y correo, y tiendas mostraban sus catálogos y se les podían hacer pedidos a distancia.

El objetivo inicial era permitir discusiones tipo Usenet además de discusiones a tiempo real. Poco tiempo atrás, Jyrki Kuoppala había creado el programa rmsg para enviar mensajes a usuarios de máquinas remotas, aunque no tenía aun el concepto de canales, de manera que era utilizado únicamente para comunicación de usuario a usuario.

De esta manera, el nacimiento del IRC fue en Agosto de 1988. La fecha concreta es una incógnita, pero su autor estima que fue hacia finales de mes.

En Enero de 1991, tuvo lugar el hecho que probablemente revolucionó el IRC, la Guerra del Golfo, que catapultó el uso del IRC a un máximo histórico de 300 usuarios. A partir del año 91, el uso del IRC fue en aumento, con 135 servidores (69 de Estados Unidos y 66 de otros lugares del mundo) y una media de 240 usuarios.

Mensajería instantánea

La Mensajería Instantánea es un punto intermedio entre los sistemas de chat y los mensajes de correo electrónico, las herramientas de mensajería instantánea, son programas regularmente gratuitos y versátiles, residen en el escritorio y, mientras hay una conexión a Internet, siempre están activos.

La mayor diferencia entre los chats y la mensajería instantánea reside en que los usuarios no se agrupan en canales por temas, si no que cada usuario tiene una lista fija de amigos (otros usuarios) de los cuales puede ver su estado (conectado/desconectado). De esta forma, las conversaciones son más individuales.

El servicio de mensajería instantánea ofrece una ventana donde se escribe el mensaje, en texto plano o acompañado de iconos o "emoticons" (figura que representan estados de ánimo), y se envían a uno o varios destinatarios quienes reciben los mensajes en tiempo real, el receptor lo lee y puede contestar en el acto.

A las últimas versiones se les han añadido una serie de aplicaciones extra como la posibilidad de entablar conversaciones telefónicas, utilizando la infraestructura de Internet, lo mismo que contar con sistemas de información financiera en tiempo real, y el compartir diferentes tipos de archivos y programas, incluidos juegos en línea.

La mensajería instantánea ha ganado popularidad en forma arrasadora. En los últimos años, el número de usuarios de cualquiera de los principales servicios: Instant Messenger de

AOL, MSN Messenger de Microsoft, Yahoo! Messenger e ICQ, se ha incrementado en forma sustancial alcanzando casi el número total de usuarios de Internet.

Nuevas tendencias

Según hemos visto, los chats han ido evolucionando en busca de una mayor expresividad y una mejor simulación de la conversación. Como hemos visto, la mensajería instantánea ya permitía el uso de emoticonos e incluso mantener conversaciones de voz.

Las nuevas tendencias quieren simular no sólo la conversación, sino la interacción física de unos usuarios con otros. De esta forma, sitúan a los usuarios dentro de un espacio en el que se pueden ver, pueden moverse y pueden relacionarse con personas y objetos. Con estos nuevos chats se puede simular mejor la realidad, acercarte a un grupo de gente para oír lo que dicen, en definitiva: realismo.

Muestra de ello, es el hotel habbot. Este chat se desarrolla en un hotel, compuesto de varias habitaciones vistas en isométrica. La habitación está llena de mobiliario que los personajes pueden usar, como mesas y sillas.

Otra tendencia es la de los shooters multijugador. Son juegos que utilizan la vista cónica de 3D, que simulan combates con armas de fuego. La extensión de estos juegos para soportar partidas por red entre jugadores ha dado lugar a que se provea a estos de un motor de chat para que puedan comunicarse mientras juegan.

Probablemente sean estos últimos los que den lugar a la nueva generación de chats. En el mundo de los chats, el realismo es lo que importa.

1 – ESPECIFICACIÓN

1.1 - ESPECIFICACIÓN FUNCIONAL

A continuación se presenta la especificación funcional del proyecto tal como fue concebido en un principio. A lo largo del curso académico fueron cambiando las prioridades y se empezaron a descartar funcionalidades que no iba a ser posible añadir.

En el apartado Alcance se describe lo que se ha incluído finalmente en el proyecto.

1.1.1 - OBJETIVOS

El objetivo es desarrollar un sistema de chat en el cuál cada usuario es representado virtualmente como un personaje que interactúa en un mundo tridimensional, relacionándose con los demás usuarios mediante la conversación escrita y un conjunto de gestos y animaciones.

El usuario es capaz de:

- Unirse a una conversación casual en una sala pública en la que participan usuarios posiblemente desconocidos.
- Reunirse con un grupo de usuarios conocidos en un lugar previamente establecido, posiblemente privado.
- Mantener una conversación uno a uno con otro usuario.
- Llevar a cabo conversaciones seguras sin temor a que sean interceptadas por terceros.
- Utilizar cualquier lenguaje y sistema de escritura existente para comunicarse con los demás usuarios.
- Asumir la apariencia de uno de los múltiples personajes disponibles e identificarse por un nombre real o ficticio.
- Realzar distintas expresiones usando determinados movimientos o animaciones.
- Visitar las distintas salas que componen el escenario.
- Interactuar con el medio de distintas maneras.

1.1.2 - DESCRIPCIÓN GENERAL

El objetivo principal de la aplicación es permitir la comunicación de forma sencilla entre distintos usuarios que pueden estar separados físicamente por una gran distancia.

La acción principal que llevará a cabo el usuario es la **conversación escrita** con un grupo arbitrario de usuarios. El tamaño de este grupo, los individuos que lo componen y el tema de conversación dependen en gran medida de la **sala de conversación** en la que se encuentre el usuario. Las salas de conversación pueden dividirse en dos grupos:

- Salas públicas, en las que suelen encontrarse más usuarios y donde posiblemente algunos son desconocidos entre sí.
- Salas privadas, a las que sólo cierto grupo de usuarios puede acceder y donde normalmente los individuos se conocen.

Las primeras en principio permiten el acceso a cualquier usuario, a no ser que esté en efecto alguna limitación por la cuál se le deniegue el acceso, como por ejemplo haber superado el aforo máximo. Las salas privadas, en cambio, permiten el acceso sólo a aquellos usuarios que hayan sido invitados o conozcan la contraseña necesaria.

Por otro lado, las salas pueden ser generadas de forma estática o de forma dinámica. Dado que los escenarios son estáticos, en el sentido de que no se pueden modificar para añadir nuevas salas, todas las salas que creen los usuarios se encontrarán separadas del escenario principal y se accederá a ellas de una manera distinta. Normalmente todas las salas dinámicas serán generadas a partir de un patrón de sala genérica.

Independientemente de su tipo, una sala de conversación normalmente tiene una **temática** asociada. Esta temática viene caracterizada por el tema principal de conversación, aunque es posible que se usen otros criterios como la decoración de la sala o el propósito principal de ésta. El personaje sólo puede encontrarse en una sala de conversación en un momento dado. Si quiere conversar con un grupo diferente de usuarios o cambiar de temática tendrá que moverse hasta otra sala.

El usuario puede elegir entre distintos **modos de conversación**. El modo más básico le permite conversar con todos los usuarios que se encuentren en la misma sala. Un personaje que esté en una sala escuchará todos los mensajes que se manden a ésta. Aparte de este modo de conversación el individuo puede mantener por ejemplo conversaciones de carácter privado con las personas que elija, que no deben encontrarse necesariamente en la misma sala.

El usuario elige un **avatar** que es con el que será representado en el escenario. También debe elegir un **nombre** que le identifique. El usuario es libre de cambiar de apariencia y de nombre en distintas sesiones. El avatar es personalizable, de tal manera que el usuario sea capaz de crear una apariencia única y distintiva para su personaje.

1.1.3 - CARACTERÍSTICAS GENERALES

A continuación se muestran las distintas características de la aplicación divididas en categorías:

- Personajes
  - Representación en 3D
  - Animaciones y gestos
  - Aspecto configurable
- Conversación
  - Distintos modos de conversación
    - Con todos los usuarios de la sala
    - Con un grupo reducido de usuarios cercanos
    - Privado con otro usuario
  - Posibilidad de usar cualquier lenguaje y sistema de escritura
  - Uso de gestos para dar énfasis a distintas expresiones
- Salas de conversación
  - Disposición y diseño variados
  - Temáticas variadas
    - Tema de conversación
    - Decoración
    - Propósito
  - Control de acceso
    - A través de invitaciones
    - A través de una contraseña
    - Otros métodos
  - Distintos modos aplicables
    - Con aforo máximo
    - Moderadas
  - Posibilidad de mantener conversaciones encriptadas
- Escenario
  - Movimiento completamente libre
  - Elementos del escenario
    - Estáticos
    - Dinámicos
- Representación
  - Escenario tridimensional
  - Bocadillos para la visualización de los mensajes
  - Cámara en tercera persona posicionable
  - Iluminación realista
  - Física simple (gravedad, inclinación de superficies, etc)
- Interfaz de usuario
  - Poco intrusivo
  - Configurable
  - Uso de gestos de ratón

1.1.4 - INICIO DE LA APLICACIÓN

1.1.4.1 - Diagrama de inicio

La aplicación aparece frente al usuario de la forma siguiente:

- Aparece una pantalla inicial de presentación.
- Un menú para elegir entre **Jugar**, **Créditos** o **Elegir aspecto/nombre.**
- Cuando se selecciona Jugar, se aparece en una posición inicial prefijada en el escenario.
- El usuario puede entonces desplazarse hacia una de las salas disponibles, tras consultar cuáles existen y son accesibles.
- Al entrar en una **sala de conversación** el usuario se encontrará con los demás usuarios que estén en ese momento en la sala, y a partir de entonces podrá comunicarse con ellos.
- Para abandonar el chat el usuario puede volver a la posición inicial y abandonar el escenario. También es posible salir directamente, cerrando la aplicación con alguna opción en un menú o pulsando la combinación ALT+F4.

1.1.5 - PERSONAJES

1.1.5.1 - Manejo del personaje

1.1.5.1.1 - Movimiento

Para desplazar el personaje por el escenario se usan los comandos Avanzar, Retroceder, Girar a la izquierda y Girar a la derecha. Con este conjunto de comandos es posible acceder a cualquier parte del escenario, y no es necesario usar comandos adicionales para subir o bajar pendientes o escaleras.

1.1.5.1.2 - Gestos

Los distintos gestos que puede llevar a cabo el personaje se realizan mediante los correspondientes comandos, ya sea introduciéndolos explícitamente o seleccionando alguna opción del interfaz, o mediante la inclusión de representaciones textuales en la frase, como puede ser por ejemplo el smiley ":D".

Para más información ver el apartado Conversación.

1.1.5.1.3 - Interacción con el medio

Para interaccionar con ciertos objetos del escenario o con otros personajes se usan los comandos Interactuar y Opciones. Para ello es necesario primero seleccionar el destino de la acción, por ejemplo con el puntero del ratón.

El comando Interactuar es la acción que se llevará a cabo por defecto, y por ejemplo en el caso de que el destino sea un personaje se establecerá una conversación privada con ese jugador.

El comando opciones es similar al menú contextual de otras aplicaciones; es una lista de las posibles acciones que se pueden realizar sobre la entidad seleccionada.

1.1.6 - CONVERSACIÓN

1.1.6.1 - Introducción

La conversación es el método de interacción principal de la aplicación, y por ello es la parte de la que más funcionalidad dispone.

En general será necesario encontrarse en una sala de conversación para llevar a cabo cualquier comunicación, a no ser que se realice una conversación a distancia, como puede ser la conversación privada.

En el caso de conversación en abierto (es decir, pública), los receptores del mensaje pueden ser múltiples, y vienen dados por la proximidad relativa al emisor del mensaje. Así, si se elige el método de conversación estándar (**hablar**) solamente aquellos personajes que se encuentren en la cercanía del emisor recibirán el mensaje; si por el contrario se quiere ampliar el radio de acción del mensaje a la totalidad de la sala, habrá que utilizar el otro método (**gritar**). Para más información ver Tipos de Conversación más abajo.

El usuario es libre de enviar cualquier tipo de mensaje a una sala, y todos los receptores potenciales lo recibirán, a no ser que la sala se encuentre moderada, en cuyo caso el mensaje deberá ser aprobado previamente por alguno de los moderadores. El usuario es también libre de utilizar cualquier idioma y método de escritura, siempre que tenga algún modo de introducir el texto.

También será posible limitar o filtrar de alguna manera los mensajes que se reciben, como por ejemplo evitando recibir los mensajes de cierto usuario (**ignorar**) o filtrando los que se reciban de usuarios más allá de cierta distancia.

Aunque la conversación es la razón principal por la que se usará la aplicación, se permitirá en la medida de lo posible ofrecer otras maneras de comunicación que no impliquen necesariamente la escritura, como pueden ser los gestos y otras animaciones.

1.1.6.2 - Tipos de conversación

Hay tres modos diferentes de conversación, que se diferencian por el radio de acción y los posibles destinatarios del mensaje.

1.1.6.2.1 - Conversación local

Este es el modo de conversación por defecto.

Tiene un radio de acción prefijado. Todos los personajes que se encuentren dentro del radio de acción escucharán lo que se diga.

1.1.6.2.2 - Conversación global a la sala

En este modo el radio de acción es la totalidad de la sala de conversación.

1.1.6.2.3 - Conversación privada

Los mensajes enviados en este modo tienen como destinatario un usuario en concreto, y tienen un radio de acción infinito (es decir, el destinatario recibirá el mensaje se encuentre donde se encuentre).

Los mensajes privados no se representan gráficamente de la misma manera que los otros tipos de mensajes. Al contrario que la conversación local, en la que las frases enviadas por cada usuario podrán relacionarse inmediatamente con éste gracias a la representación en bocadillo, los mensajes privados, al provenir de un usuario que posiblemente se encuentre a una distancia grande o en una sala de conversación distinta, aparecerán en una **ventana de privado**.

1.1.7 - SALAS DE CONVERSACIÓN

1.1.7.1 - Modos

Las salas de conversación pueden tener asociados distintos **modos**, que afectan de diversas maneras a la comunicación entre usuarios y al acceso de éstos a la sala.

Los modos disponibles son los siguientes:

- **Normal**: Este es el modo que por defecto tienen todas las salas. En este caso no hay restricción alguna al acceso o al envío de mensajes a la sala por parte de cualquier usuario.
- **Aforo limitado**: En este modo el número de usuarios que pueden encontrarse en la sala al mismo tiempo está limitado. Si este límite se alcanza, el acceso a la sala no estará permitido hasta que o bien el límite se aumente o alguno de los usuarios presentes abandone la sala. Este modo es útil si se quiere evitar que haya demasiadas conversaciones al mismo tiempo.
- **Moderada**: En el caso de salas moderadas aparece la figura de los **moderadores**. Los moderadores son usuarios con privilegios especiales que se encargan de seleccionar, entre todos los mensajes que son enviados a la sala, cuáles deben aparecer y cuáles no. Este modo es útil si el número de usuarios en la sala es grande y se quiere organizar de alguna manera lo que será recibido en la sala.
- **Acceso restringido**: Cuando este modo está activo el acceso a la sala estará restringido y sólo aquellos usuarios que cumplan los requisitos necesarios podrán acceder a la sala. Los distintos tipos de control de acceso se explican en el apartado siguiente.

1.1.7.2 - Control de acceso

En algunos casos es deseable limitar de alguna manera qué usuarios pueden acceder a determinada sala. El método más básico de control de acceso es el de aforo limitado descrito más arriba. El problema de este método es que no permite discriminar entre los usuarios que intentan acceder a la sala; a todos ellos les es denegado el acceso.

Una manera más deseable de controlar quién puede acceder a una sala es construir una lista de los usuarios que pueden acceder, o proteger la sala con una contraseña y comunicársela sólo a determinados usuarios. Estos tipos de control de acceso son descritos a continuación, y se pueden combinar indistintamente.

1.1.7.2.1 - Acceso por identificador de usuario

Una manera de controlar los usuarios que pueden acceder a una sala es construir una lista con identificadores de usuario. Esta lista puede ser de dos tipos:

- **Inclusiva**: Especifica los únicos usuarios que pueden acceder a la sala. Si un usuario intenta entrar en la sala pero no se encuentra en la lista se le denegará el acceso.

- **Exclusiva**: Especifica los únicos usuarios que no pueden acceder a la sala. Todo usuario que no se encuentre en la lista podrá entrar en la sala.

1.1.7.2.2 - Acceso por contraseña

Otra manera eficaz de evitar que usuarios no deseados accedan a una sala es protegerla con contraseña. Al intentar entrar en la sala se requerirá la contraseña, y a no ser que se introduzca correctamente el acceso será denegado.

Los usuarios lógicamente tendrán que comunicarse la contraseña por algún método seguro, de tal manera que estén seguros de que sólo ellos la conocen. Si por alguna razón la contraseña de la sala deja de ser secreta es posible cambiarla en cualquier momento.

1.1.7.3 - Creación de nuevas salas

En principio el escenario tendrá definidas distintas salas, llamadas salas estáticas, que no pueden crearse o destruirse arbitrariamente. Que el número de salas sea fijo es un factor limitante que puede eliminarse permitiendo la creación de nuevas salas no disponibles en el escenario.

Dado que estas salas no son parte del escenario original, el acceso a ellas es diferente y habrá poca variedad entre ellas. Para más información ver el apartado Escenario.

Al crear una nueva sala, el usuario seleccionará el tipo y temática de la sala entre distintas plantillas genéricas disponibles. Una vez hecho esto la nueva sala aparecerá en la lista de salas disponibles y los demás usuarios podrán acceder a ella.

1.1.8 - MOTOR DE DIBUJO

1.1.8.1 - Introducción al motor

El motor de dibujo tiene las siguientes características:

- Usa una representación en tres dimensiones del escenario, los objetos y los distintos personajes que se encuentran en él.
- Representa la posición y el movimiento de los demás personajes de acuerdo con la información que recibe del servidor, intentando en la medida de lo posible minimizar la desincronización que pueda producirse por tener una comunicación de alta latencia con el servidor.
- Representa de manera intuitiva, en forma de bocadillo, los mensajes enviados por los distintos usuarios, evitando siempre que sea posible el uso de una ventana global de mensajes.
- Posee una cámara que mantendrá en todo momento al personaje en pantalla.
- Detecta y evita las colisiones entre distintas entidades, como jugador con escenario o cámara con jugador.

1.1.8.2.1 - Elementos 2D del interfaz de usuario

Los elementos del interfaz de usuario, aunque se intentarán minimizar, serán necesarios para poder acceder a toda la funcionalidad de la aplicación. Estos elementos serán como los que se encuentran normalmente en otros interfaces de usuario: ventanas, botones, listas, etc.

1.1.8.2.2 - Bocadillos

Los bocadillos son la manera en la que se representan los distintos mensajes enviados por cada usuario. Todo usuario que haya enviado un mensaje recientemente tendrá un bocadillo asociado, a no ser que se decida por alguna razón que no debe mostrarse (por ejemplo si la distancia al receptor es grande o si éste ya tiene muchos bocadillos en pantalla).

Un bocadillo no es más que un marco en el que están contenidos los mensajes que ha enviado recientemente un usuario. Cada bocadillo está representado de tal manera que es fácil saber a qué usuario pertenece.

El bocadillo puede ser opaco, facilitando la lectura del texto, o semitransparente, de tal manera que no se oculte en demasiada medida lo que se encuentra detrás. Siempre se encontrará de frente a la cámara, por lo que se puede considerar un objeto 2D.

Un bocadillo puede contener varias frases. Cuando el usuario envía un nuevo mensaje éste aparece en la parte inferior del bocadillo, desplazando el resto de frases hacia arriba, lo que puede hacer que una frase anterior deje de ser visible.

Siempre que es posible los bocadillos aparecen sobre la cabeza del personaje al que pertenecen, pero dado a que el escenario es un mundo tridimensional arbitrario es muy posible que en un momento dado dos usuarios que están hablando se encuentren uno detrás de otro con respecto a la posición de la cámara. También se puede dar el caso de que un usuario que se encuentra a la espalda de la cámara envíe un mensaje, y es deseable que se muestre.

Cuando un bocadillo no pueda mostrarse en la posición deseada (sobre la cabeza del personaje) se buscará una posición alternativa en pantalla lo más cercana posible. Si la posición donde finalmente aparece el bocadillo está demasiado alejada se mostrará alguna ayuda para identificar el personaje al que pertenece. Si el personaje que envía el mensaje no se encuentra dentro del ángulo de visión su bocadillo aparecerá en uno de los lados de la pantalla, con una flecha indicando la posición aproximada del jugador.

En el caso en que una gran cantidad de usuarios cercanos estén conversando en un momento dado y que la representación de todos los bocadillos requiera un espacio excesivo en pantalla, u oculte en gran medida el escenario, se mostrarán solamente los bocadillos de los personajes más cercanos.

1.1.8.3 - Detección de colisiones

La colisión entre jugador y jugador no será tenida en cuenta por el motor. Los personajes serán incorpóreos entre sí y podrán ocupar el mismo espacio.

De esta manera se evita que usuarios malintencionados o aquellos que han perdido la conexión bloqueen por ejemplo un pasillo estrecho o la entrada a una sala.

1.1.9 - SERVIDOR

1.1.9.1 - La figura del administrador

En todo sistema cliente-servidor es necesario que exista la figura del administrador: un usuario con privilegios especiales que se encarga de asegurar el buen funcionamiento del sistema y el ajuste del comportamiento de los usuarios a las reglas de uso establecidas.

1.1.9.2 - Interfaz de administración

El interfaz de administración será muy similar al de un usuario normal, pero tendrá opciones extra que facilitan la tarea de desplazarse a una posición determinada, encontrar y seleccionar a un usuario concreto y modificar las salas existentes o crear nuevas.

1.2 – CASOS DE USO

Se detallan a continuación los casos de uso más importantes considerados en el proyecto. Estos casos de uso sirvieron para describir exactamente el comportamiento que debía tener la aplicación y también para encontrar las dependencias entre los distintos subsistemas, de tal manera que el diseño que se hiciera posteriormente fuera apropiado.

Los casos de uso se encuentran divididos en clases.

1.2.1 – CASOS DE USO DEL INTERFAZ

1.2.1.1 - Iniciar Cliente

Descripción: Describe la secuencia de pasos que se han de llevar a cabo al ejecutar la aplicación cliente.

Prioridad: Alta
Ver También: Cerrar Cliente

Precondiciones: El cliente no se está ejecutando.
Secuencia:

1. El Usuario ejecuta el cliente.
2. Se inician los sistemas básicos (ficheros, logging y configuración)
3. Se leen los ficheros de configuración.
   3.1. Si los ficheros de configuración no se encuentran entonces se crean con valores por defecto.
4. Se inicia el Sistema de GráficosSe inicia el Sistema de Red.
5. Se muestra la ventana inicial de la aplicación.

1.2.1.2 - Cerrar Cliente

Descripción: Describe la secuencia de pasos que se han de llevar a cabo al dar la orden de cerrar la aplicación cliente.

Prioridad: Alta
Ver También: Iniciar Cliente

Precondiciones: El cliente se está ejecutando.
Secuencia:

1. El Usuario da la orden de cerrar el cliente.
2. La lógica de la aplicación se para.
3. Se inicia un temporizador para poder comunicarle al Servidor convenientemente la desconexión del cliente.
4. Se realiza la petición de desconexión al Servidor y se espera respuesta.
   4.1. Si el temporizador se activa antes de recibir respuesta entonces no se espera la respuesta del Servidor y se pasa a 5..
5. Se cierra la conexión con el Servidor.
6. Se desactivan los distintos sistemas y se liberan sus recursos.
7. Se cierra la aplicación.

1.2.1.3 - Cambiar Resolución de Pantalla

Descripción: Describe cómo se lleva a cabo el cambio de resolución de la pantalla.

Prioridad: Baja

Precondiciones: Ninguna.
Secuencia:

1. El Usuario selecciona, mediante la opción correspondiente, la nueva resolución. Si la nueva resolución es igual a la actual, fin.
2. Se comprueba que la nueva resolución es soportada.
3. El renderizado a la ventana se deshabilita.
4. Si se ha seleccionado una profundidad de color distinta de la actual se vuelve a crear el contexto de rendering con el nuevo formato de pixel.
5. Si se está ejecutando en modo ventana, pasar a 7.
6. Se cambia la resolución de pantalla.
7. Se redimensiona la ventana.
8. Se rehabilita el renderizado a la ventana.

1.2.1.4 - Minimizar Ventana Principal

Descripción: Describe cómo se lleva a cabo la minimización de la ventana principal. Cuando la ventana está minimizada la aplicación no recibe ningún tipo de entrada del usuario y el dibujo sobre ella se deshabilita.

Prioridad: Baja
Ver También: Restaurar Ventana Principal

Precondiciones: La ventana debe ser visible y estar activa.
Secuencia:

1. El Usuario da la orden de minimizar la ventana.
2. Se deshabilita el renderizado a la ventana.
3. Se minimiza la ventana. Si se está ejecutando en modo ventana, fin.
4. Se restaura la resolución de pantalla que había antes de pasar a modo pantalla completa.

1.2.1.5 - Restaurar Ventana Principal

Descripción: Describe cómo se lleva a cabo la restauración de la ventana principal desde su estado minimizado.

Prioridad: Baja
Ver También: Minimizar Ventana Principal

Precondiciones: La ventana debe estar minimizada.
Secuencia:

1. El Usuario da la orden de restaurar la ventana.
2. Si se estaba ejecutando en modo ventana, pasar a 4.
3. Se restaura la resolución de pantalla que usaba la aplicación.
4. Se restaura la ventana.
5. Se rehabilita el renderizado a la ventana.

1.2.1.6 - Paso de Ventana a Pantalla Completa

Descripción: Describe cómo se lleva a cabo el paso de modo Ventana a modo Pantalla Completa. Normalmente el modo Pantalla Completa requiere cambiar la resolución de la pantalla para que coincida con el de la ventana.

Prioridad: Baja
Ver También: Paso de Pantalla Completa a Ventana

Precondiciones: La ventana principal debe ser visible y encontrarse en modo ventana.
Secuencia:

1. El Usuario da la orden de pasar a pantalla completa.
2. Se deshabilita el renderizado a la ventana.
3. Si la resolución de pantalla y profundidad de color son iguales a las que necesita la aplicación, pasar a 5.
4. Se cambia la resolución de pantalla y la profundidad de color.
5. Se eliminan los bordes de la ventana.
6. Se reposiciona la ventana en la esquina superior izquierda y se redimensiona hasta ocupar la totalidad de la pantalla.
7. Se rehabilita el renderizado a la ventana.

1.2.1.7 - Paso de Pantalla Completa a Ventana

Descripción: Describe cómo se lleva a cabo el paso de modo Pantalla Completa a modo Ventana.

Prioridad: Baja
Ver También: Paso de Ventana a Pantalla Completa

Precondiciones: La ventana principal debe ser visible y encontrarse en modo ventana.
Secuencia:

1. El Usuario da la orden de pasar a modo ventana.
2. Se deshabilita el renderizado a la ventana.
3. Se restaura la resolución de pantalla que había antes de pasar a pantalla completa.
4. Se restauran los bordes de la ventana.
5. Se restaura la posición de la ventana.
6. Se rehabilita el renderizado a la ventana.

1.2.2 – CASOS DE USO DE LA SINCRONIZACIÓN

1.2.2.1 - Enviar Evento de Personaje

Descripción: Describe el proceso que se ha de llevar a cabo cuando un cliente realiza una acción que produzca la generación de un evento de personaje (por ejemplo, hacer avanzar al personaje).

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Usuario: El Usuario realiza una acción con su personaje (avanzar, retroceder, girar, saltar, sentarse, levantarse o comenzar un emote).

2. Lógica: Crea un evento de comienzo de movimiento y lo marca con tiempo t + *local lag*. El evento se añade a la lista de eventos de la simulación.
3. Cliente: Envía el evento inmediatamente al servidor.
4. Servidor: Recibe el evento del cliente. Si el tiempo actual de simulación es menor que el tiempo asignado al evento, pasar a 5.
   4.1 Servidor.Lógica: Asigna al evento el tiempo actual de simulación.
   4.2 Servidor: Envía al cliente que comenzó el movimiento notificación de que el evento se ha retrasado hasta el tiempo actual.
5. Servidor.Lógica: Añade el evento a la simulación (ver *Añadir Evento a Simulación*).
6. Servidor.Lógica: Determina los clientes que se encuentran en el área de interés del cliente que ha iniciado el movimiento.
7. Servidor: Envía el evento a todos los clientes que se encuentran en el área de interés del que inició el movimiento.

1.2.2.2 - Añadir Evento a Simulación

Descripción: Describe el proceso que se lleva a cabo cuando la lógica recibe un nuevo evento que ha de añadir a la simulación en el tiempo indicado.

Prioridad: Alta

Precondiciones: El tiempo del evento a añadir es mayor que el tiempo de simulación más antiguo del que se dispone información de estado del juego.
Secuencia:

1. Lógica: Decide añadir un nuevo evento a la simulación.
2. Lógica: Añade el evento a la lista de eventos en el tiempo correspondiente. Si este tiempo es mayor o igual que el tiempo de simulación actual, fin.
3. Lógica: Se notifica la necesidad de hacer rollback de simulación hasta el tiempo en el que se tuvo que haber ejecutado el evento (ver *Hacer Rollback de Simulación*).

1.2.2.3 - Hacer Rollback de Simulación

Descripción: Describe el proceso que se lleva a cabo cuando la lógica recibe un evento retrasado y se ve obligada a volver al estado de un punto pasado en el tiempo y volver a ejecutar la simulación hasta el tiempo actual, de tal manera que la simulación coincida con la del servidor.

Prioridad: Alta

Precondiciones: Existe información sobre el estado del juego en el tiempo hasta el que se quiere hacer rollback.
Secuencia:

1. Lógica: Detecta que es necesario hacer un rollback a un tiempo pasado para volver a simular lo ocurrido desde entonces.
2. Lógica: Guarda el estado actual de la simulación.
3. Lógica: Recupera el estado de la simulación en el punto desde el que se quiere hacer rollback.
4. Lógica: Se realizan los pasos de simulación necesarios para llegar hasta el ciclo actual.
   4.1 Lógica: Para cada evento que se procesa se comprueba si en su ejecución anterior dio lugar a otros eventos. Si no es así, pasar a siguiente evento.
   4.2 Lógica: Para cada evento al que el evento que se está considerando dio lugar en su ejecución anterior, pero al que no ha dado lugar en la ejecución actual, se envía un anti-evento anulándolo, ya que no es válido.
5. Lógica: Se comprueban las posiciones de todas las entidades que han cambiado de posición durante la resimulación.

5.1 Lógica: A cada entidad que requiera una corrección de posición se le asigna un controlador de corrección de trayectoria.

1.2.2.4 - Procesar Evento Retrasado

Descripción: Describe el proceso que se lleva a cabo cuando el cliente recibe notificación del servidor de que un evento que el cliente generó ha sido retrasado. Esto puede ocurrir cuando el evento no llega al servidor antes de que se cumpla el tiempo en el que estaba programado.

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Cliente: Recibe notificación del servidor de que un evento que este cliente envió anteriormente ha sido retrasado.
2. Lógica: Añade un evento del tipo del evento original a la lista de eventos en el nuevo tiempo asignado.
3. Lógica: Busca el evento original en la lista de eventos. Si no se encuentra, error.
4. Lógica: Elimina el evento original de la lista de eventos.
5. Lógica: Hace un rollback hasta el tiempo del evento original.

1.2.2.5 - Procesar Anti-evento

Descripción: Describe el proceso que se lleva a cabo cuando el cliente recibe un anti-evento. Un anti-evento es una notificación de que un evento que el cliente mandó anteriormente ha sido cancelado y que por tanto todos los eventos que se produjeran dependiendo de éste han de ser cancelados.

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Cliente: Recibe del Servidor un mensaje de anti-evento para cancelar un evento que se recibió en un momento anterior.
2. Lógica: Busca el evento que ha de ser cancelado en la lista de eventos. Si no se encuentra, fin.
3. Lógica: Elimina el evento al que se refiere el anti-evento de la lista.
4. Lógica: Se notifica la necesidad de hacer rollback de simulación hasta el tiempo en el que se ejecutó el evento.

1.2.2.6 - Inicializar Área de Interés

Descripción: Describe la inicialización del área de interés de un cliente.

Prioridad: Media

Precondiciones: No se dispone en el Servidor de información sobre el área de interés del cliente.
Secuencia:

1. Cliente: Realiza su entrada en el escenario.
2. Servidor.Lógica: Añade al área de interés del Cliente todos los clientes que estén dentro del radio de ésta (ver *Entrar en Área de Interés*).

3. Servidor.Lógica: Añade al área de interés del Cliente todos los clientes que son visibles desde su posición.
4. Servidor.Lógica: Añade el jugador al área de interés de los clientes correspondientes.

1.2.2.7 - Entrar en Área de Interés

Descripción: Describe el proceso que se realiza cuando una entidad entra en el área de interés de un cliente. Una entidad puede entrar en el área de interés de un cliente cuando: o bien está lo suficientemente próxima, o bien está a la vista, o bien tiene un contacto con una entidad que ha entrado en el área de interés.

Prioridad: Media

Precondiciones: La entidad que entra en el área de interés no se encontraba ya en ésta.
Secuencia:

1. Servidor.Lógica: Detecta que una entidad tiene que ser añadida al área de interés de un cliente, ya sea por cercanía o por visibilidad.
2. Servidor.Lógica: Si la entidad considerada está marcada para eliminación de este área de interés entonces se quita esta marca y fin.
3. Servidor.Lógica: Crea un evento con tiempo actual y de tipo "añadir entidad" con toda la información sobre el estado de la entidad que el cliente necesita para añadirla correctamente a su simulación.
4. Servidor.Lógica: Para cada entidad que tenga creado un contacto con la entidad considerada y no se encuentre ya en el área de interés del cliente se crea un nuevo evento de tipo "añadir entidad", de tal manera que el cliente pueda también añadirla a su simulación.
5. Servidor: Envía los eventos al cliente.

1.2.2.8 - Salir de Área de Interés

Descripción: Describe el proceso que se realiza cuando una entidad sale del área de interés de un cliente. Una entidad puede salir del área de interés de un cliente cuando está lo suficientemente lejos, no es visible por éste y además no tiene ningún contacto con alguna entidad que esté aún dentro del área de interés.

Prioridad: Media

Precondiciones: La entidad que sale del área de interés se encontraba en ésta, así como todas las entidades que tienen un contacto creado con ella.
Secuencia:

1. Servidor.Lógica: Detecta que una entidad tiene que ser eliminada del área de interés de un cliente.
2. Servidor.Lógica: Si la entidad tiene un contacto con alguna otra entidad que también está en el área de interés del cliente pero que no está marcada para eliminación de ésta, simplemente se marca para eliminación en este área de interés y fin.
3. Servidor.Lógica: Se crea un evento con tiempo actual y de tipo "eliminar entidad" para la entidad considerada.
4. Servidor.Lógica: Para cada entidad que tenga creado un contacto con la entidad considerada se crea un nuevo evento de tipo "eliminar entidad".
5. Servidor: Envía los eventos al cliente.

1.2.2.9 - Procesar Evento Añadir Entidad

Descripción: Describe el comportamiento del cliente cuando procesa una notificación que recibió del servidor de que una entidad ha entrado en su área de interés y por tanto ha de añadirla a su simulación. A partir de este momento el cliente recibirá información sobre esta entidad.

Prioridad: Media

Precondiciones: La entidad no existe en la simulación.
Secuencia:

1. Lógica: Se dispone a procesar un evento de tipo "añadir entidad" como parte del ciclo de simulación.
2. Lógica: Crea la nueva entidad e inicializa su posición, rotación y demás propiedades físicas y de animación con la información que contiene el evento.
3. Lógica: Crea los controladores correspondientes y los inicializa con los datos del evento.
4. Lógica: Añade el evento a la simulación en el tiempo correspondiente, posiblemente provocando un rollback (ver *Añadir Evento a Simulación*).

1.2.2.10 - Procesar Evento Eliminar Entidad

Descripción: Describe el comportamiento del cliente cuando procesa una notificación que recibió del servidor de que una entidad ha salido de su área de interés y por tanto ha de eliminarla de su simulación. A partir de este momento el cliente dejará de recibir información sobre esta entidad.

Prioridad: Media

Precondiciones: La entidad existe en la simulación.
Secuencia:

1. Lógica: Se dispone a procesar un evento de tipo "eliminar entidad" como parte del ciclo de simulación.
2. Lógica: Elimina la entidad de la simulación.
3. Lógica: Hace rollback hasta el momento marcado en el evento (ver *Hacer Rollback de Simulación*).

1.2.2.11 - Procesar Evento de Personaje

Descripción: Describe el comportamiento del cliente cuando procesa una notificación que recibió del servidor de que una entidad ha llevado a cabo alguna acción.

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Lógica: Se dispone a procesar un evento de personaje (avanzar/retroceder, girar, saltar, sentarse, levantarse o comenzar emote) como parte de la simulación.
2. Lógica: Añade el controlador correspondiente para el personaje y lo inicializa con el estado actual de éste.

1.2.2.12 - Ciclo de Controlador de Corrección de Trayectoria

Descripción: Describe el proceso que lleva a cabo el controlador que se le aplica a las entidades de una simulación local que hacen de *proxies* de entidades que no son controladas

por el cliente y que han visto su trayectoria extrapolada corregida por la llegada de un evento del servidor.

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Lógica: El ciclo del Controlador de Corrección de Trayectoria de una entidad es llamado.
2. Si la posición actual del fantasma de la entidad es igual al de la entidad, se elimina este controlador y fin.
3. Se determina la posición correcta de la entidad.
4. Se ajusta la posición del fantasma para acercarla a la posición correcta de la entidad.
5. Si se produce colisión del fantasma con algún objeto del escenario o con el escenario en sí, se elimina este controlador de la entidad y se reactiva el dibujado de ésta.

1.2.3 – CASOS DE USO DE LA RED

1.2.3.1 – Encender servidor

Descripción: Describe la secuencia de pasos que se han de llevar a cabo al inicializar la aplicación de servidor. El administrador de la red decide encender el servidor para que los clientes puedan comunicarse entre ellos. Este proceso lee las configuraciones y activa la escucha de paquetes.

Prioridad: Alta
Ver También: Apagar Servidor

Precondiciones: Ninguna.
Secuencia:

Actores: administrador, servidor

1. Administrador: ejecuta el comando del servidor
2. Servidor: Toma de los ficheros de configuración la información sobre los canales preexistentes
3. Servidor: Toma de los ficheros de configuración los puertos por los que escuchar
4. Servidor: Lanza el logger que abrirá el fichero o medio de log (registro) para añadir
5. Servidor: El logger escribirá en el log una referencia al inicio de sesión de servidor, con información de fecha, hora y la máquina que está sirviendo
6. Servidor: Comienza la escucha por los puertos

1.2.3.2 – Apagar Servidor

Descripción: Describe la secuencia de pasos que se han de llevar a cabo al dar la orden de apagar la aplicación servidora. El administrador decide que la máquina que está haciendo de servidor deje de ejecutar el servidor, haciendo que los usuarios no puedan seguir chateando.

Prioridad: Alta
Ver También: Encender Servidor

Precondiciones: El servidor se está ejecutando.
Secuencia:

Actores: Administrador, Usuario, Gráficos, Lógica, Red (Cliente, Servidor)

1. Administrador: ejecuta comando de apagado de servidor (como shutdown.bat de tomcat)
2. Servidor: Envía un mensaje de finalización a los usuarios
3. Servidor: Cierra conexiones
4. Servidor: El logger escribirá en el log información sobre el final de sesión (fecha, hora)
5. Cliente: Recibe la información de servidor y lo comunica a la lógica
6. Lógica: Finalizará los sistemas
7. Gráficos: Mostrarán un mensaje de desconexión por apagado de servidor

1.2.3.3 – Cambiar Nick

Descripción: Describe la secuencia de pasos que se han de llevar a cabo cuando un usuario quiere cambiar de nick. Un usuario quiere que el resto de usuarios le conozcan por medio de otro apodo.

Prioridad: Media

Precondiciones: El cliente se está ejecutando, el servidor se está ejecutando, y el cliente está conectado al servidor.
Secuencia:

Actores: Usuario, Gráficos, Lógica, Red (Cliente, Servidor)

1. Usuario: Indica a la interfaz su intención de cambiar de nick
2. Gráficos: Muestra una ventana formulario para introducir los datos del nuevo nick
3. Usuario: Introduce los datos y acepta
4. Lógica: Toma los datos, cambia su variable nick y se los pasa a Cliente
5. Cliente: Envía mensaje al Servidor proponiendo cambio de nick
6. Servidor: Recibe el mensaje con el cambio de nick
7. Servidor: Busca entre todos los clientes para buscar que no hay colisión del nuevo nick con los existentes
8. Servidor: Cambia el nick del usuario en su lista de usuarios
9. Servidor: El logger graba en el log el cambio de nick
10. Servidor: Comunica el cambio a todos los clientes conectados al canal en el que está el usuario que ha pedido su cambio de nick
11. Cliente: Los clientes reciben la información de cambio de nick y la aplican a su lista de nicks
12. Gráficos: Muestran por pantalla el cambio de nick

Casos alternos:

3b. Usuario: Cancela orden

8c. Servidor: Encuentra nick coincidente
9c. Servidor: Envía petición de cambio de nick al cliente que hizo la petición para que recupere su nick anterior
10c. Cliente: Recibe petición de volver al nick anterior por parte del servidor
11c. Lógica: Cambia nick

1.2.3.4 – Conectarse al Chat

Descripción: Describe la secuencia de pasos que se han de llevar a cabo cuando un usuario quiere conectarse a un servidor. El usuario ha arrancado su aplicación cliente y debe conectarse a un servidor para poder hablar con otros usuarios.

Prioridad: Alta

Ver También: Desconectarse del chat.

Precondiciones: El cliente se está ejecutando, el servidor se está ejecutando, y el cliente no está conectado al servidor.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: Indica a la interfaz el deseo de conectarse
2. Interfaz: Muestra formulario sobre información de nick, avatar
3. Usuario: Rellena el formulario
4. Lógica: Toma los datos del formulario, y los guarda en memoria, comprobando su validez
5. Cliente: Abre una conexión al servidor
6. Cliente: Manda la solicitud de participar en el chat
7. Servidor: Toma la solicitud y el logger mete en el log los datos del cliente (ip, nick)
8. Servidor: Comprueba que no hay colisión de nicks
9. Servidor: Inserta la información del usuario en la lista de usuarios
10. Servidor: El logger mete en el log que el usuario ha conectado con éxito
11. Servidor: Manda al cliente información de sincronización
12. Cliente: Sincroniza su reloj
13. Gráficos: Muestra que la conexión se realizó con éxito

Casos alternos:

3b. Usuario: Cancela orden

5c. Cliente: La conexión no se pudo establecer
6c. Gráficos: Se muestra información del error

8d. Servidor: Determina que hay colisión de nicks
9d. Servidor: Manda un mensaje indicando el error al cliente
10d. Servidor: Cierra socket y elimina la información usada
11d. Cliente: Recibe el mensaje
12d. Gráficos: Muestra que la conexión no se pudo establecer y la causa del error

1.2.3.5 – Desconectarse del chat

Descripción: Describe la secuencia de pasos que se han de llevar a cabo cuando un usuario se quiere desconectar del servidor. El usuario decide que ya no quiere chatear más y por lo tanto, se desconecta del servidor.

Prioridad: Alta
Ver También: Conectarse al chat.

Precondiciones: El cliente se está ejecutando, el servidor se está ejecutando, y el cliente está conectado al servidor.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: Indica a la interfaz el deseo de conectarse
2. Cliente: Manda al servidor su solicitud de salir
3. Servidor: Recibe el mensaje de solicitud
4. Servidor: Cierra el socket con el usuario
5. Servidor: Elimina al usuario de la lista de usuarios
6. Servidor: El usuario sale del canal (Ver abandonar canal)

7. Servidor: El logger mete en el log que el usuario ha desconectado
8. Lógica: Apaga los subsistemas
9. Gráficos: Muestra la desconexión

Casos alternos:

1b. Cliente: Se desconecta el socket
2b. Saltar a 5

6b. El usuario no estaba en ningún canal -> Saltar a 7

1.2.3.6 – Entrar a un canal

Descripción: Describe la secuencia de pasos que se han de llevar a cabo cuando un usuario quiere entrar en un canal. Un usuario conectado al chat quiere entrar a un canal dedicado a un tema específico para poder hablar con los usuarios que están en dicho canal.

Prioridad: Alta
Ver También: Abandonar un canal.

Precondiciones: El cliente se está ejecutando, el cliente está conectado al servidor, y el cliente está en un canal.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: Indica a la interfaz el deseo de entrar en un canal
2. Cliente: Solicita una lista de canales al servidor
3. Servidor: Recibe el mensaje de solicitud
4. Servidor: Manda al cliente la lista de canales
5. Cliente: recibe la lista de canales
6. Gráficos: muestra la lista de canales en pantalla
7. Usuario: elige el canal al que quiere entrar
8. Lógica: guarda el canal actual en su información
9. Cliente: Manda la solicitud de entrar en el canal elegido
10. Servidor: Recibe la solicitud
11. Servidor: Comprueba la existencia del canal
12. Servidor: Agrega el usuario a la lista del canal
13. Servidor: El logger mete en el log la entrada al canal
14. Servidor: Manda al cliente información sobre todos los usuarios
15. Servidor: Manda a todos los usuarios del canal el mensaje de que el nuevo usuario ha entrado
16. Lógica: Inicializa los subsistemas
17. Lógica: Crea los personajes e inicializa su posición mediante una predicción basada en la información recibida
18. Lógica: Muestra que la conexión se ha efectuado con éxito

Casos alternos:

4b. No hay canales-> Servidor: Manda un error diciendo que no hay canales
5b. Cliente: Recibe el error
6b. Gráficos: Muestra el error por pantalla

7c. Usuario: Cancela la petición

11d. El canal no existe-> Servidor: Envía un error diciendo que el canal no existe
12d. Cliente: Recibe el error

13d. Gráficos: Muestra el error por pantalla

1.2.3.7 – Abandonar un canal

Descripción: Describe la secuencia de pasos que se han de llevar a cabo cuando un usuario quiere abandonar un canal. Un usuario decide salir de un canal temático, probablemente para meterse en otro que le interese más.

Prioridad: Alta
Ver También: Entrar a un canal

Precondiciones: El cliente se está ejecutando, el cliente está conectado al servidor, y el cliente está en un canal.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: Indica a la interfaz el deseo de abandonar el canal
2. Cliente: Manda al servidor su solicitud de salir del canal
3. Servidor: Recibe el mensaje de solicitud
4. Servidor: Elimina al usuario de la lista del canal
5. Servidor: El logger apunta el evento
6. Servidor: Comunica a todos los clientes en el canal que el usuario ha salido del canal
8. Lógica: Apaga los subsistemas
9. Gráficos: Muestra la salida del canal

Casos alternos:

4b. El usuario no estaba en el canal -> Servidor envía mensaje de error
5b. Cliente: recibe mensaje de error.
6b. Gráficos: Muestra información del error por la pantalla

1.2.3.8 - Hablar

Descripción: Describe la secuencia de acciones a realizar cuando un usuario quiere decir una frase para que la oigan los usuarios más cercanos en el canal.

Prioridad: Media
Ver También: Gritar

Precondiciones: El cliente se está ejecutando, el cliente está conectado al servidor, y el cliente está en un canal.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: Escribe en la interfaz el mensaje que va a mandar
2. Cliente: Manda al servidor el mensaje del usuario
3. Servidor: Recibe el mensaje del usuario
4. Servidor: Toma el canal de la información del usuario y calcula las posiciones de los individuos
5. Servidor: Manda el texto a los usuarios cercanos del canal
6. Gráficos: Muestra el texto en el canal y en nuestro personaje

Casos alternos:

4b. El usuario no estaba en ningún canal -> Servidor: Envía un mensaje de error al cliente
5b. Cliente: Recibe el mensaje de error
6b. Gráficos: Muestra el mensaje de error por pantalla

1.2.3.9 - Gritar

Descripción: Describe la secuencia de acciones a realizar cuando un usuario quiere decir una frase en un canal para que lo oigan todos los usuarios en el canal.

Prioridad: Alta
Ver También: Hablar

Precondiciones: El cliente se está ejecutando, el cliente está conectado al servidor, y el cliente está en un canal.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: Escribe en la interfaz el mensaje que va a mandar
2. Cliente: Manda al servidor el mensaje del usuario
3. Servidor: Recibe el mensaje del usuario
4. Servidor: Toma el canal de la información del usuario
5. Servidor: Manda el texto a los todos los usuarios del canal
6. Gráficos: Muestra el texto en el canal y en nuestro personaje

Casos alternos:

4b. El usuario no estaba en ningún canal -> Servidor: Envía un mensaje de error al cliente
5b. Cliente: Recibe el mensaje de error
6b. Gráficos: Muestra el mensaje de error por pantalla

1.2.3.10 - Enviar Mensaje Privado

Descripción: Describe la secuencia de acciones a realizar cuando un usuario quiere decir una frase a otro usuario conectado al servidor. El usuario lo usa para hablar con otro usuario sin que ningún otro tenga la posibilidad de oírle.

Prioridad: Media
Ver También: Recibir Mensaje Privado

Precondiciones: El cliente se está ejecutando y el cliente está conectado al servidor.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: Escribe en la interfaz el mensaje que va a mandar y a quien
2. Cliente: Manda al servidor el mensaje del usuario
3. Servidor: Recibe el mensaje del usuario
4. Servidor: Busca al usuario que recibirá el privado en su lista de usuarios
5. Servidor: Manda el texto a dicho usuario
6. Interfaz: Mostramos el texto en ventana y nuestro personaje

Casos alternos:

4b. El usuario no conectado-> Servidor: Mandamos error
5b. Cliente: Recibe el mensaje de error
6b. Gráficos: Muestra el mensaje de error por pantalla

1.2.3.11 – Recibir Mensaje Privado

Descripción: Describe la secuencia de acciones a realizar cuando un usuario recibe una frase privada que otro usuario ha dicho.

Prioridad: Media
Ver También: Enviar Mensaje Privado

Precondiciones: El cliente se está ejecutando y el cliente está conectado al servidor.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Cliente: Recibe del servidor el mensaje del usuario
2. Lógica: Busca la correspondencia entre el que manda y los personajes
3. Interfaz: Mostramos el texto en ventana y personaje

Casos alternos:

2b. Lógica: No encontramos el personaje que nos ha enviado el mensaje
3b. Interfaz: Mostramos la ventana sin el personaje

1.2.3.12 – Comenzar Movimiento

Descripción: Describe la secuencia de acciones a realizar cuando un usuario comienza a realizar determinado movimiento de translación.

Prioridad: Alta
Ver También: Finalizar Movimiento

Precondiciones: El cliente se está ejecutando, el cliente está conectado al servidor, y el cliente está en algún canal.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: El usuario pulsa un cursor de avanzar o retroceder y lo mantiene pulsado
2. Cliente: Comunica al servidor que se ha comenzado un desplazamiento, la posición de inicio, el tiempo, y la dirección
3. Servidor: Recibe el mensaje del usuario
4. Servidor: Toma el canal de la información del usuario
5. Servidor: Mete el movimiento en la información del usuario eliminando los que se anulan con éste
5. Servidor: Manda el movimiento al resto de usuarios del canal
6. Gráficos: Muestra el movimiento en el canal

Casos alternos:

4b. El usuario no estaba en ningún canal -> Servidor: Envía un mensaje de error al cliente
5b. Cliente: Recibe el mensaje de error
6b. Gráficos: Muestra el mensaje de error por pantalla

1.2.3.13 – Finalizar Movimiento

Descripción: Describe la secuencia de acciones a realizar cuando un usuario termina de realizar determinado movimiento de translación.

Prioridad: Alta
Ver También: Comenzar Movimiento

Precondiciones: El cliente se está ejecutando, el cliente está conectado al servidor. El cliente está en algún canal y además había comenzado a realizar un movimiento.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: El usuario suelta un cursor de avanzar o retroceder que mantenía pulsado
2. Cliente: Comunica al servidor que se ha finalizado un desplazamiento, la posición de fin, el tiempo, y la dirección
3. Servidor: Recibe el mensaje del usuario
4. Servidor: Toma el canal de la información del usuario
5. Servidor: Mete el movimiento en la información del usuario eliminando los que se anulan con éste
6. Servidor: Manda el movimiento al resto de usuarios del canal
7. Gráficos: Muestra el movimiento en el canal

Casos alternos:

4b. El usuario no estaba en ningún canal -> Servidor: Envía un mensaje de error al cliente
5b. Cliente: Recibe el mensaje de error
6b. Gráficos: Muestra el mensaje de error por pantalla

1.2.3.14 – Comenzar Giro

Descripción: Describe la secuencia de acciones a realizar cuando un usuario comienza a realizar determinado movimiento de rotación.

Prioridad: Alta
Ver También: Finalizar Giro

Precondiciones: El cliente se está ejecutando, el cliente está conectado al servidor, y el cliente está en algún canal.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: El usuario pulsa un cursor de rotar y lo mantiene pulsado
2. Cliente: Comunica al servidor que se ha comenzado una rotación, la posición de inicio, el tiempo, y la dirección
3. Servidor: Recibe el mensaje del usuario
4. Servidor: Toma el canal de la información del usuario
5. Servidor: Mete el movimiento en la información del usuario eliminando los que se anulan con éste

5. Servidor: Manda el movimiento al resto de usuarios del canal
6. Gráficos: Muestra el movimiento en el canal

Casos alternos:

4b. El usuario no estaba en ningún canal -> Servidor: Envía un mensaje de error al cliente
5b. Cliente: Recibe el mensaje de error
6b. Gráficos: Muestra el mensaje de error por pantalla

1.2.3.15 – Finalizar Giro

Descripción: Describe la secuencia de acciones a realizar cuando un usuario termina de realizar determinado movimiento de rotación.

Prioridad: Alta
Ver También: Comenzar giro

Precondiciones: El cliente se está ejecutando, el cliente está conectado al servidor. El cliente está en algún canal y además había comenzado a realizar un giro.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: El usuario suelta un cursor de rotar que mantenía pulsado
2. Cliente: Comunica al servidor que se ha finalizado un desplazamiento, la posición de fin, el tiempo, y la dirección
3. Servidor: Recibe el mensaje del usuario
4. Servidor: Toma el canal de la información del usuario
5. Servidor: Mete el movimiento en la información del usuario eliminando los que se anulan con éste
6. Servidor: Manda el movimiento al resto de usuarios del canal
7. Gráficos: Muestra el movimiento en el canal

Casos alternos:

4b. El usuario no estaba en ningún canal -> Servidor: Envía un mensaje de error al cliente
5b. Cliente: Recibe el mensaje de error
6b. Gráficos: Muestra el mensaje de error por pantalla

1.2.3.16 - Saltar

Descripción: Describe la secuencia de acciones a realizar cuando un usuario quiere realizar un salto.

Prioridad: Baja

Precondiciones: El cliente se está ejecutando, el cliente está conectado al servidor y el cliente está en algún canal.
Secuencia:

Actores: Usuario, Interfaz, Lógica, Cliente, Servidor

1. Usuario: El usuario pulsa la tecla de saltar (da igual que la mantenga pulsada)

2. Cliente: Comunica al servidor que el usuario salta, la posición de inicio, el tiempo, y la dirección
3. Servidor: Recibe el mensaje del usuario
4. Servidor: Toma el canal de la información del usuario
5. Servidor: Mete el movimiento en la información del usuario eliminando los que se anulan con éste
5. Servidor: Manda el movimiento al resto de usuarios del canal
6. Gráficos: Muestra el movimiento en el canal

Casos alternos:

4b. El usuario no estaba en ningún canal -> Servidor: Envía un mensaje de error al cliente
5b. Cliente: Recibe el mensaje de error
6b. Gráficos: Muestra el mensaje de error por pantalla

1.2.4 – CASOS DE USO DEL MOVIMIENTO

1.2.4.1 - Mover Personaje

Descripción: Describe la manera en que se maneja la petición de un usuario de mover a su personaje, así como de girar o saltar.

Prioridad: Alta

Precondiciones: Se está en Modo Normal. El personaje está de pie sobre el suelo.
Secuencia:

1. Usuario: Ejecuta el comando correspondiente para hacer avanzar, retroceder, girar o saltar a su personaje.
2. Lógica: Crea el evento de movimiento correspondiente y lo envía al Servidor (ver *Enviar Evento de Personaje*)

1.2.4.2 - Sentar Personaje

Descripción: Describe la manera en que se maneja la petición de un usuario de hacer saltar a su personaje.

Prioridad: Baja

Precondiciones: Se está en Modo Normal. El personaje está de pie sobre el suelo. El personaje está parado.
Secuencia:

1. Usuario: Ejecuta el comando correspondiente para sentar a su personaje.
2. Lógica: Crea un evento de salto y lo envía al Servidor (ver *Enviar Evento de Personaje*).

1.2.4.3 - Levantar Personaje

Descripción: Describe la manera en que se maneja la petición de un usuario de hacer que su personaje se levante.

Prioridad: Baja

Precondiciones: Se está en Modo Normal. El personaje está sentado.
Secuencia:

1. Usuario: Ejecuta el comando correspondiente para levantar a su personaje.
2. Lógica: Crea un evento de levantar personaje y lo envía al Servidor (ver *Enviar Evento de Personaje*).

1.2.4.4 - Comenzar Emote

Descripción: Describe la manera en que se maneja la petición de un usuario de comenzar un emote.

Prioridad: Media

Precondiciones: Se está en Modo Normal. El emote es compatible con el estado actual del personaje.

Secuencia:

1. Usuario: Ejecuta el comando correspondiente para comenzar un emote.
2. Lógica: Crea un evento de comienzo de emote y lo envía al Servidor (ver *Enviar Evento de Personaje*).

1.2.4.5 - Ciclo de Controlador de Avanzar/Retroceder

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Lógica: El ciclo del Controlador de Avanzar/Retroceder de un personaje es llamado.
2. Si la velocidad vertical del personaje es lo suficientemente negativa (se detecta que está cayendo) se sustituye este controlador por un Controlador de Caer, y se restaura cuando termine este último. Fin.
3. Si el personaje no está de pie sobre el suelo, fin.
4. Se posiciona el personaje a la altura del suelo, a no ser que el cambio de altura sea demasiado alto.
5. Se aplican las fuerzas correspondientes al personaje: impulso y rozamiento.
6. Si la animación actual no es la de avanzar, se cambia por la animación de andar o la de correr, dependiendo de la velocidad actual del personaje.
7. Se ajusta la velocidad de la animación de acuerdo con la velocidad del personaje.

1.2.4.6 - Ciclo de Controlador de Girar

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Lógica: El ciclo del Controlador de Girar de un personaje es llamado.
2. Si la velocidad vertical del personaje es lo suficientemente negativa (se detecta que está cayendo) se sustituye este controlador por un Controlador de Caer, y se restaura cuando termine este último. Fin.
3. Si el personaje no está de pie sobre el suelo, fin.
4. Se aplican los torques correspondientes: impulso y rozamiento.
5. Si el personaje está avanzando o retrocediendo, fin.
6. Si la animación actual no es la de girar, se cambia.
7. Se ajusta la velocidad de la animación de acuerdo con la velocidad de giro del personaje.

1.2.4.7 - Ciclo de Controlador de Saltar

Prioridad: Media

Precondiciones: Se ha inicializado el controlador para dar el impulso inicial.
Secuencia:

1. Lógica: El ciclo del Controlador de Saltar de un personaje es llamado.
2. Si la velocidad vertical del personaje es positiva, ir a 3.

2.1 Se elimina este controlador del personaje.
2.2 Se pone un Controlador de Caer y fin.

3. Si la animación actual no es la de salto, se sustituye.

1.2.4.8 - Ciclo de Controlador de Caer

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Lógica: El ciclo del Controlador de Caer de un personaje es llamado.
2. Si la velocidad vertical del personaje es lo suficientemente pequeña (se detecta que ha dejado de caer), se elimina este controlador y se restaura el que tuviera previamente. Fin.
3. Si la animación actual no es la de caída, se sustituye.

1.2.4.9 - Ciclo de Controlador de Sentar

Prioridad: Media

Precondiciones: Ninguna.
Secuencia:

1. Lógica: El ciclo del Controlador de Sentar de un personaje es llamado.
2. Si la velocidad vertical del personaje es lo suficientemente negativa (se detecta que está cayendo) se cancela la acción de sentarse, se sustituye este controlador por un Controlador de Caer, y se restaura un Controlador de Inactividad cuando termine este último. Fin.
3. Si el personaje ha terminado de sentarse, se sustituye este controlador por un Controlador de Inactividad. Fin.
4. Si la animación actual no es la de sentarse, se sustituye.

1.2.4.10 - Ciclo de Controlador de Levantar

Prioridad: Media

Precondiciones: Ninguna.
Secuencia:

1. Lógica: El ciclo del Controlador de Levantar de un personaje es llamado.
2. Si la velocidad vertical del personaje es lo suficientemente negativa (se detecta que está cayendo) se sustituye este controlador por un Controlador de Caer, y se restaura un Controlador de Idle cuando termine este último. Fin.
3. Si se produce colisión durante la acción de levantarse, se cancela la acción y se restaura el controlador anterior. Fin.
4. Si el personaje ha terminado de levantarse, se sustituye este controlador por un Controlador de Inactividad. Fin.
5. Si la animación actual no es la de levantarse, se sustituye.

1.2.4.11 - Ciclo de Controlador de Emote

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Lógica: El ciclo del Controlador de Emote de un personaje es llamado.
2. Se modifica la animación del personaje acordemente.
3. Si el emote requiere movimiento del personaje, añade las fuerzas y torques correspondientes.

4. Si el personaje ha terminado el emote, se elimina este controlador.

1.2.4.12 - Ciclo de Controlador de Inactividad

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Lógica: El ciclo del Controlador de Inactividad de un personaje es llamado.
2. Si la velocidad vertical del personaje es lo suficientemente negativa (se detecta que está cayendo) se sustituye este controlador por un Controlador de Caer, y se restaura cuando termine este último. Fin.
3. Si el personaje no tiene creado ningún contacto con otra objeto dinámico, se desactiva la física sobre él.
4. Si la animación actual no es la de inactividad, se sustituye.

1.2.4.13 - Empujar Objeto/Jugador

Prioridad: Baja

Precondiciones: Existe un contacto entre las dos entidades.
Secuencia:

1. Lógica: Determina que una entidad está empujando a otra.
2. Física: La entidad que empuja aplica la fuerza correspondiente en el punto de contacto en una dirección paralela a su eje de movimiento y del mismo sentido.
3. Física: Se resuelven las fuerzas teniendo en cuenta las masas de las entidades y la posición del punto de contacto.

1.2.4.14 - Tirar de Objeto

Prioridad: Baja

Precondiciones: El personaje está lo suficientemente cerca del objeto.
Secuencia:

1. Usuario: Notifica que desea tirar de un objeto.
2. Física: El personaje se aleja del objeto en una dirección dada.
3. Física: Se resuelven las fuerzas teniendo en cuenta las masas y posiciones de las entidades y las propiedades de la juntura.

1.2.4.15 - Levantar Objeto

Prioridad: Baja

Precondiciones: El personaje está lo suficientemente cerca del objeto.
Secuencia:

1. Usuario: Notifica que desea levantar un objeto.
2. Física: Se aplica una fuerza en el punto más cercano del objeto en dirección vertical.

1.2.5 – CASOS DE USO DE LA CÁMARA

1.2.5.1 - Mantener Visible Objetivo de Cámara

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Lógica: Se dispone a asegurarse de que la cámara está posicionada y orientada de tal manera que su objetivo es visible.
2. Si la cámara se encuentra a la distancia deseada del objetivo al que debe seguir, pasar a 3.
   2.1 La cámara se acerca al objetivo.
3. Si no hay ningún objeto que esté bloqueando la vista del objetivo al que debe seguir, pasar a 4.
   3.1 Posiciona la cámara por delante del objeto para que el objetivo sea visible.
4. Si la cámara se encuentra mirando al centro del objetivo al que debe mirar, fin.
   4.1 Gira la cámara para apuntar al objetivo al que debe mirar, que no tiene por qué ser el mismo que al que tiene que seguir.

1.2.5.2 - Acercar/Alejar Cámara

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Usuario: Da la orden de acercar o alejar la cámara de su personaje.
2. Si se produce una colisión entre la cámara y el escenario o algún objeto, se cancela la operación y fin.
3. Se comienza el movimiento de acercamiento o alejamiento de la cámara.

1.2.5.3 - Girar Cámara

Prioridad: Alta

Precondiciones: Ninguna.
Secuencia:

1. Usuario: Da la orden de girar la cámara.
2. Se comienza el movimiento de giro de la cámara.

1.2.5.4 - Fijar Cámara en Objeto/Personaje

Prioridad: Baja

Precondiciones: Ninguna.
Secuencia:

1. Usuario: Da la orden de fijar la cámara sobre un objeto o personaje distinto de su avatar.
2. Lógica: Cambia el objetivo al que la cámara debe mirar para que sea el objeto o personaje seleccionado.
3. Si en algún momento el objetivo al que la cámara debe mirar sale del área de interés del cliente, se restaura el avatar del jugador como objetivo de la cámara.

1.3 – ALCANCE

Debido a lo ambicioso del proyecto, y teniendo en cuenta nuestra disponibilidad y productividad, se descartaron muchas de las características iniciales del chat en la implementación.

En este apartado se ofrece una visión general de qué características han sido implementadas y cuáles han sido desechadas.

1.3.1 - OBJETIVOS

El objetivo sigue siendo la realización de un mundo virtual al que varios usuarios se pueden conectar y se ven representados mediante un personaje. Los personajes interactuan entre ellos mediante una conversación escrita y gestos que el personaje realiza para dar expresividad a aquello que está diciendo. Estos gestos incluyen tanto la gesticulación típica en una conversación como los gestos que el usuario ordena explícitamente que su personaje realice (emoticonos). Estas conversaciones las puede tener en el mundo en el que se encuentran todos los usuarios conectados.

Por el contrario:

- El chat no dispone de diferentes salas de conversación, ni públicas ni privadas.
- El mundo es único para todos los usuarios.
- Las conversaciones privadas y encriptadas también han sido descartadas del proyecto.
- La aplicación tampoco soporta unicode(codificación de caracteres universal).
- Todos los personajes usan un avatar por defecto.
- La interacción con el medio se limita a la colisión básica con los escenarios, incluyendo subir y bajar escaleras.

Como hemos mencionado, nuestro chat tiene un mundo único. Tanto las salas públicas como las salas privadas quedan fuera del alcance del proyecto.

Éste mundo, por lo tanto, no tiene límite ni de acceso ni de aforo. Tampoco tiene temática asociada, y cualquier usuario puede escuchar lo que está diciendo otro usuario conectado.

El usuario no puede elegir avatar. El avatar es único para todos, y el usuario no tendrá nombre asociado al que se pueda referir otro usuario.

1.3.2 – APLICACIÓN DISEÑADA VS APLICACIÓN IMPLEMENTADA

1.3.2.1 - inicio

La aplicación especificada presenta el inicio de la aplicación como una pantalla de presentación, en la que el usuario puede elegir entre ver la pantalla de créditos, diseñar/elegir su avatar y su nick, o conectarse al chat.

En la aplicación implementada, en cambio, el cliente toma la configuración del servidor de un archivo y se conecta a éste, usando el avatar por defecto, y usando el mundo por defecto, sin interacción del usuario en ello. El nick es proporcionado por el servidor, pero el usuario no tiene constancia ni de su nick ni de los del resto de usuarios.

1.3.2.2 - Personaje

El personaje ha sido llevado a cabo tal como estaba en la especificación, de forma que es capaz de avanzar, retroceder, girar y realizar gestos mediante pulsaciones de teclas.

El personaje es capaz de subir escaleras, y otras interacciones básicas con el escenario.

En cambio, el personaje no interactúa con ningún objeto del escenario en la aplicación implementada. No sólo en el sentido de que no puede coger un libro, o sentarse en una silla, si no que las colisiones con los objetos tampoco han sido implementadas.

1.3.2.3 - Conversación

La aplicación especificada nos presentaba diferentes modos de comunicación entre usuarios (hablar, gritar) referidos al tipo de conversación (local, global, privada). Además daba opciones sobre filtrado, como puede ser ignorar a otro usuario.

El único modo de conversación que se ha preservado en la aplicación implementada es el modo "global a la sala", es decir, que cualquier cosa que el usuario diga, puede ser oída por cualquier otro usuario que esté en el mundo.

1.3.2.4 - Salas

Se especificaron varios tipos de sala en el diseño:

- Normal
- Aforo limitado
- Moderada
- Acceso restringido

En la aplicación implementada el concepto de sala no existe. El mundo es único para todos. Todos entran en él al conectarse, y no hay posibilidad de cambiar de mundo.

Este mundo corresponde a una sala de tipo normal de la especificación.

No es posible tampoco crear salas nuevas.

1.3.2.5 - Interfaz gráfica

La interfaz gráfica de la aplicación implementada nos muestra el mundo en 3D, con todos los personajes y objetos que en él se encuentran. El movimiento del resto de usuarios se interpola para dar la sensación de que no hay latencia de red de por medio.

Las conversaciones se llevan a cabo mediante bocadillos, que presentan el texto que ha dicho el personaje sobre el que se sitúa.

La interfaz posee una cámara que enfoca al personaje por detrás, y lo mantiene siempre visible, esquivando los obstáculos que se interpongan y colisionando con el escenario para no atravesar las paredes.

Con respecto a la aplicación especificada, tan sólo ha faltado por implementar:

- Ventana de privados: Puesto que no hay privados, estas ventanas no se han implementado.
- Ventana de conversación de canal: Una ventana que en 2D escribiera todo el texto del canal. Puesto que el mundo es grande, el número de conversaciones inconexas

sería demasiado. Además, los usuarios generarían demasiado tráfico como para ser leído.

1.3.2.6 - Detección de colisiones

Las colisiones se han implementado con el mundo, de forma que el jugador no puede atravesar paredes. También puede subir y bajar escaleras, o caer desde una altura. Se realiza mediante un BSP.

El punto no implementado en este sentido ha sido la colisión con objetos, puesto que queríamos usar un algoritmo más complejo que el usado con el escenario, que lo habría dotado de más realismo.

1.3.2.7 - Administrador

Se descartó implementar la parte de administrador. Ni interfaz ni protocolo. El servidor tampoco da soporte para superusuario.

Por tanto todos los clientes son iguales, y pueden realizar las acciones convenidas para un cliente normal del chat.

2 – DISEÑO

2.1 – INTRODUCCIÓN

Este capítulo se encuentra dividido en dos partes. La primera parte describe el diseño que se consideró y detalló en un principio. Este diseño no se ha implementado en su mayor parte, por lo que la segunda parte del capítulo detalla el diseño que corresponde a la aplicación realizada.

El diseño inicial de la aplicación se llevó a cabo tras considerar todos los casos de uso relevantes y detallar la mayoría de ellos. Aunque el alcance final de la aplicación sea solamente una parte de la especificación inicial, el diseño está preparado para acomodar el resto de funcionalidades consideradas sin necesidad de grandes reestructuraciones.

El diseño inicial se encuentra dividido en una serie de **módulos**, relativamente independientes, que se unen para formar la aplicación, ya sea el cliente o el servidor. La división en módulos de esta manera permite que ciertas funcionalidades que han de ser idénticas en el cliente y en el servidor puedan ser compartidas a nivel de código. Esta reutilización de código es importante y necesaria, por ejemplo en el caso del código que lleva a cabo los cálculos y las actualizaciones de las posiciones de los objetos, donde es imprescindible que no haya diferencias sensibles entre el cliente y el servidor, ya que podrían dar lugar a simulaciones distintas que complicarían en gran medida la sincronización.

Las siguientes secciones describen en todo detalle los distintos módulos que componen los diseños mencionados, acompañándose las explicaciones de diagramas de clases.

2.2 – DISEÑO INICIAL

Como se ha mencionado anteriormente, el diseño inicial se encuentra dividido en módulos. Estos módulos se correponden, a nivel de implementación, con bibliotecas estáticas que se enlazan para formar los respectivos ejecutables. La reutilización de código se consigue entonces simplemente enlazando el cliente y el servidor con las mismas bibliotecas.

Los módulos que componen la aplicación son los siguientes:

- **Client**: Este módulo contiene el código específico de la aplicación cliente. Se encarga de la lectura de los ficheros de configuración propios del cliente y del manejo de la ventana principal sobre la que se mostrará la escena.

- **Collision**: Este módulo contiene todo el código relacionado con la detección y la respuesta a colisiones entre las distintas entidades colisionables de la aplicación. El módulo Collision depende del módulo Physics para llevar a cabo alguna de sus funcionalidades.

- **Control**: Este módulo, que es específico del cliente, se encarga de todo lo relacionado con la entrada de teclado y de ratón. Contiene código que traduce los distintos tipos de eventos de los periféricos de entrada a comandos que tienen sentido en la aplicación.

- **Graphics**: Este módulo contiene el motor gráfico que se encarga de representar la escena, teniendo en cuenta los estados de todos los objetos que componen el escenario. El motor gráfico funciona de forma independiente de lo que se puede englobar como **lógica**, que básicamente es el conjunto de los módulos relacionados directa o indirectamente con la actualización de la simulación. Mientras que la lógica se ejecuta con una frecuencia fija, el *rendering* se produce a la máxima velocidad que permita la plataforma.

- **Logic**: Este módulo contiene la parte principal del código que se encarga de llevar a cabo la simulación. Recibe eventos a través de los módulos Control y Net que modifican el curso de la simulación.

- **Math**: Este módulo contiene las clases ayudantes que facilitan los cálculos con objetos matemáticos como vectores, matrices y cuaterniones.

- **Net**: Este módulo se encarga de todo lo relacionado con la conexión con el servidor.

- **Physics**: Este módulo contiene el código relacionado con el comportamiento dinámico de los objetos del escenario. El módulo Physics depende del módulo Collision para avisarle de que dos objetos han colisionado y por tanto estos objetos han de responder adecuadamente.

- **Server**: Este módulo contiene el código específico de la aplicación servidor.

2.2.1 – MÓDULO CLIENT

El módulo Client compone la parte inicial de la aplicación cliente.

Por un lado se encarga de leer los archivos de configuración y de inicializar los distintos subsistemas. Una vez hecho esto se encarga de realizar la conexión con el servidor y de entrar en el escenario. Finalmente muestra la ventana principal y da el control al usuario.

Por otro lado se encarga de todo lo relacionado con la ventana principal: visualización de la escena y recepción de eventos de los periféricos de entrada.

En cuanto a la visualización de la escena, la ventana del cliente proporciona un lugar donde llevar a cabo el render, que en terminología de Windows es el *device context*[1], al cuál se le asocia un *rendering context* que soporte OpenGL.

Los eventos de teclado y de ratón, por otro lado, son recibidos por la aplicación a través de mensajes de Windows. Estos mensajes son capturados y procesados en lo que se conoce como *message loop*. El bucle de mensajes se encarga de procesar uno a uno los mensajes que han ido llegando a la cola de mensajes de la ventana. Una vez inspeccionados los mensajes, si se detecta que están relacionados con el teclado o el ratón, se traducen a los eventos de entrada específicos de la aplicación y son enviados al módulo Control para que sean procesados.

[1] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/gdi/devcons_25pv.asp

2.2.2 – MÓDULO COLLISION

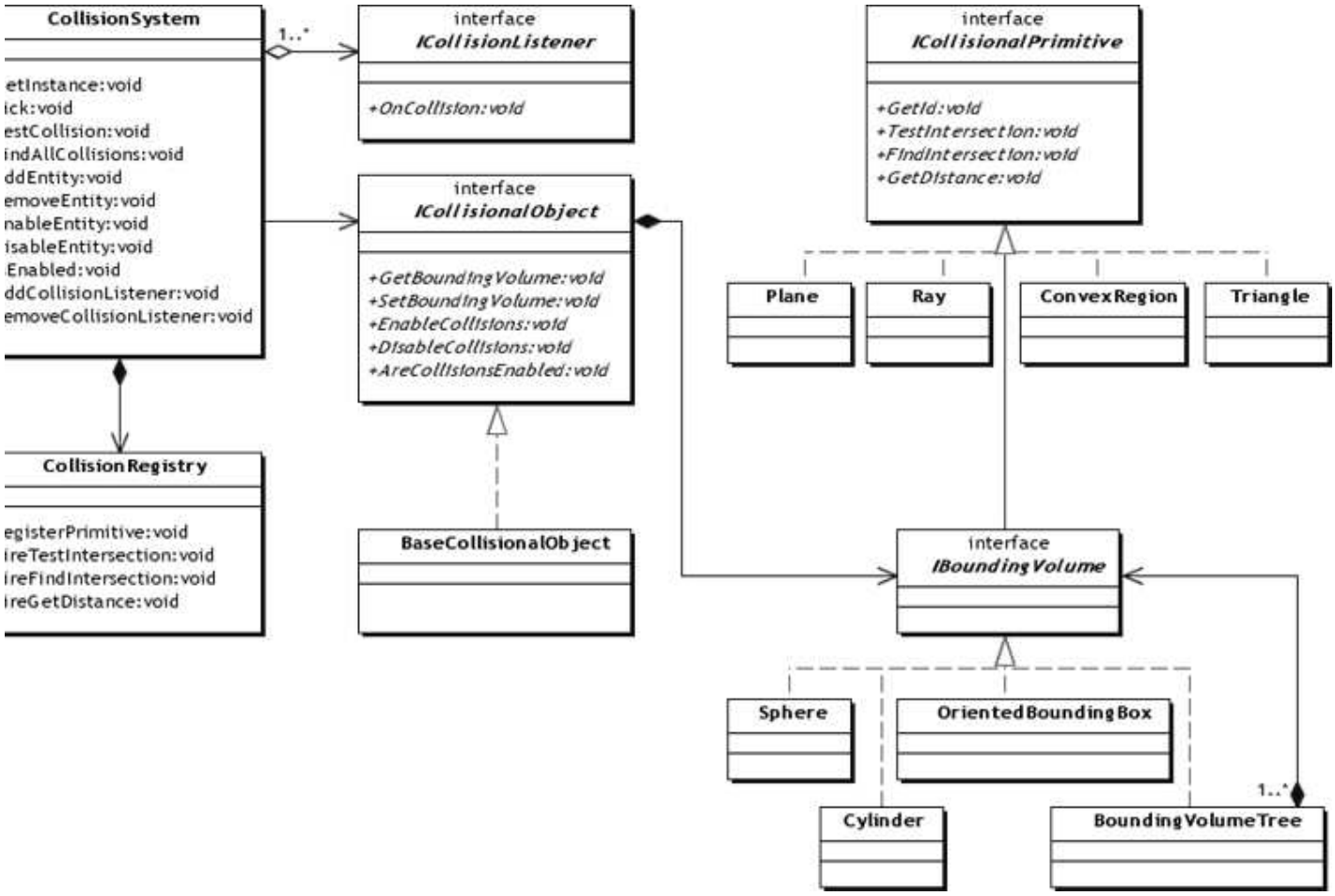
2.2.2.1 - Objetivos

El objetivo principal que se quería conseguir cuando se diseñó el módulo Collision fue permitir el movimiento del jugador por escenarios complejos. Debía ser posible moverse libremente por escenarios con subidas y bajadas progresivas o en forma de escalera, caídas, y paredes y otros elementos insalvables. También debía ser posible colisionar con objetos que no formaran parte del propio escenario, objetos que podrían encontrarse en lugares arbitrarios o tener algún tipo de movimiento.

Además, la colisión debía ser eficiente y no entorpecer el ritmo de la aplicación aún cuando el escenario constase de cientos de miles de polígonos y hubiera cientos de objetos complejos dispersos por él.

Estos requisitos han guiado las decisiones de diseño y han hecho que se consideren estructuras de datos concretas para llevar a cabo el trabajo.

2.2.2.2 – Descripción del Diseño



El interfaz principal al módulo Collision es la clase **CollisionSystem**. Esta clase proporciona un interfaz sencillo que permite a otros módulos hacer uso de los servicios de detección de colisiones. Proporciona métodos para detectar si hay colisión entre dos primitivas de colisión, para encontrar todas las colisiones que se han producido y para activar o desactivar la detección de colisiones sobre las entidades.

Las **primitivas de colisión** son los objetos básicos sobre los que se puede detectar una colisión. No son necesariamente objetos que tienen representación visual, sino que sirven para aproximar las propiedades de los objetos del escenario. Suele tratarse de formas sencillas (segmentos, planos, esferas, cubos, etc) de tal manera que sea posible determinar en poco tiempo si dos de ellas intersecan.

El número de primitivas es reducido porque si se quiere tener un sistema de colisiones completo es necesario tener código específico para la colisión entre cada par de primitivas de colisión. Esto hace que el número de casos que hay que implementar crezca de manera cuadrática con respecto al número de primitivas. En el sistema de colisiones tal como se diseñó se consideraron las siguientes primitivas: planos, segmentos (rayos), triángulos, conjuntos convexos de triángulos (regiones convexas) y finalmente volúmenes contenedores, que a su vez se dividen en esferas, cilindros, OBBs[2] y árboles de todas éstas. La razón de tener estos tipos de primitivas es que cada una es usada en un tipo de detección de colisión distinta, como se detalla en los siguientes apartados.

Una de las cosas que hay que decidir a la hora de implementar un sistema de detección de colisiones es el nivel de detalle al que se quiere llegar, es decir, con qué precisión se quiere calcular el punto de intersección entre dos objetos. La detección de colisiones al nivel de los polígonos individuales que componen los objetos es una **detección de colisiones exacta**.

Existen varios algoritmos eficientes que calculan colisiones exactas. En [2] se describe un algoritmo que detecta colisiones entre cuerpos rígidos en movimiento y que explota la coherencia temporal, es decir, la propiedad de que la posición de los objetos no cambia de manera significativa en cada paso de tiempo. El algoritmo consta de dos fases. La primera fase se encarga de detectar los posibles pares de objetos que colisionan, mediante la detección de intersección entre sus AABBs[3]. Esta detección se acelera usando un algoritmo *sweep and prune* que permite descartar los pares de AABBs que se encuentran demasiado lejos. Una vez que se tienen los pares de objetos candidatos a colisionar se procede a la segunda fase, donde se realiza una detección de colisiones exacta mediante diagramas de Voronoi, obteniéndose el par de polígonos que colisionan y el punto más cercano de ambos. Este algoritmo funciona tanto para polígonos convexos como para polígonos no convexos.

En [3] se presenta un algoritmo que se basa en el uso de jerarquías de volúmenes contenedores para la detección de colisiones entre objetos. La jerarquía planteada es un árbol de DOPs[4], estructura que se argumenta es igual de eficiente que los árboles de OBBs para los tests de colisión y más eficiente en espacio. El algoritmo consta de una sola fase, en la que para cada par de objetos se consideran sus árboles de DOPs y se hace un recorrido paralelo de ambos. Para el test de intersección, en cada nodo del árbol se transforma el DOP de un objeto al sistema de coordenadas del otro objeto, donde se lleva a cabo una detección eficiente. Si para cierto nodo del árbol se detecta que el DOP correspondiente del primer objeto no interseca con el DOP del segundo objeto entonces los objetos no pueden colisionar. Si no es así, se sigue bajando por el árbol hasta llegar a la raíz, tras lo cuál se realiza la detección de colisiones exacta entre los polígonos que forman ambos objetos.

Por otro lado, en [4] se presenta un algoritmo para la detección exacta de colisiones usando árboles de OBBs. De manera similar al algoritmo anterior, la detección hace un

[2] Oriented Bounding Boxes
[3] Axis-Aligned Bounding Boxes
[4] Discrete Oriented Polytopes

descenso por el árbol haciendo tests de intersección entre la OBB de cada nodo, hasta llegar a la raíz donde se procede a hacer una detección más precisa a nivel de polígono. El trabajo presenta una forma eficiente de hacer el test de intersección entre OBBs, y compara estos volúmenes contenedores con otras posibles elecciones como AABBs, argumentando que las OBBs proporcionan un ajuste mayor a la geometría del objeto, no necesitan ser recalculadas cuando el objeto cambia de orientación y no requieren una gran cantidad extra de operaciones. También presenta un algoritmo para la generación automática de árboles de OBBs para objetos poligonales arbitrarios.

Finalmente, [5] extiende el algoritmo anterior para soportar objetos en movimiento. Además presenta pseudocódigo para la implementación de los tests de intersección OBB-OBB, OBB-Triángulo y Triángulo-Triángulo, ya sean estacionarios o en movimiento. También describe dos maneras alternativas de crear el árbol de OBBs.

En el presente proyecto se ha decidido no implementar una detección de colisiones exacta, ya que se ha considerado que no es ventajosa la mayor precisión que proporciona a cambio de una mayor complejidad en tiempo. Tampoco se ha decidido soportar colisión entre objetos en movimiento, sino que se hace uso de la propiedad de coherencia temporal para comprobar solamente la intersección de los objetos en sus posiciones actuales.

Para la detección de colisiones se ha considerado utilizar un algoritmo de dos fases: una primera fase de grano grueso que descarta, mediante el uso de un árbol BSP[5], los posibles pares de objetos que están demasiado lejos para colisionar, y una segunda fase de grano fino que calcula intersecciones entre las jerarquías de OBBs que aproximan la superficie de los objetos, sin llegar a hacer tests entre los triángulos que componen estos objetos. En las secciones siguientes se detallan en más profundidad los algoritmos considerados.

Otra pieza principal del módulo Collision es la clase **CollisionRegistry**. Esta clase se encarga de llevar el control de las distintas primitivas de colisión soportadas y de dar acceso al algoritmo de detección de colisiones específico entre ellas. La clase CollisionRegistry implementa *double-dispatch* en C++, descrito por Meyers en [6] y más tarde extendido por Alexandrescu en [7]. La implementación elegida corresponde a un caso concreto de los *multimethods* de complejidad en tiempo constante descritos en [7]. El double-dispatch permite realizar una llamada virtual que dependa de las clases concretas de *dos* objetos; el mecanismo de llamadas virtuales de C++ sólo permite seleccionar la función a llamar dependiendo del tipo dinámico de un objeto.

Una vez descubiertos los tipos dinámicos de las dos primitivas que intervienen en la colisión se ejecuta el código específico de cálculo de intersección entre éstas. Para la implementación de estos cálculos de intersección se ha considerado usar la biblioteca Wild Magic [8], que proporciona funciones de cálculo de intersecciones eficientes entre una gran cantidad de primitivas.

Todas las entidades de la aplicación que deban poder participar en la detección de colisiones han de implementar el interfaz **ICollisionalObject**, o derivar de la clase concreta **BaseCollisionalObject**, que proporciona implementaciones por defecto de los métodos del interfaz. Todos los objetos colisionables tienen por tanto un volumen contenedor que les representa. Este volumen contenedor podrá ser un volumen contenedor simple o un árbol de éstos. En las siguientes secciones se detallan qué contenedores se asocian a los distintos tipos de entidades.

[5] Binary Space Partitioning

2.2.2.3 – Colisión entre personaje y escenario

La colisión entre el personaje y el escenario es la parte más importante del sistema de detección de colisiones, ya que permite que el personaje pueda moverse libremente por el escenario de una manera realista, sometiéndose a los distintos obstáculos que lo compongan.

Dado que los escenarios tienen que poder constar de varios cientos de miles de polígonos, no es factible usar un algoritmo de fuerza bruta y hacer la comprobación entre los polígonos del personaje y cada uno de los polígonos del escenario. Para reducir al máximo el conjunto de polígonos con los que se debe comprobar la colisión se ha considerado hacer uso de una jerarquía de subdivisión espacial, concretamente un árbol BSP. El uso de una estructura de datos de este tipo permite descartar rápidamente un gran número de polígonos con los que el personaje no puede colisionar.

Para la aproximación de la geometría del personaje se ha decidido usar un cilindro. El cilindro es una figura geométrica muy sencilla que permite hacer tests de intersección muy eficientes, se ajusta bastante bien a la forma de un personaje humanoide, y además tiene la ventaja de que permite al jugador girar sobre sí mismo sin ocupar más espacio, al contrario que lo que pasa con las AABBs y OBBs, simplificando el cálculo de colisiones cuando el personaje gira estando muy cerca de algún obstáculo. Otros volumenes menos sencillos pero que podrían ser más ajustados son también posibles, como por ejemplo las cápsulas [9].

A continuación se describe con detalle el uso y la creación del árbol BSP considerado en el proyecto.

2.2.2.3.1 - ¿Qué es un árbol BSP?

Un árbol BSP es una estructura de datos que representa una subdivisión recursiva del espacio. Se trata de un árbol binario donde cada nodo interno representa un plano (o hiperplano, si se generaliza a más de 3 dimensiones) que subdivide el espacio en dos mitades. Estas mitades son subdivididas recursivamente y forman los hijos del nodo. Las hojas representan regiones del espacio que ya no se considera necesario seguir subdividiendo, ya sea porque contienen a lo sumo un objeto, porque se ha llegado a un nivel de profundidad elegido o algún otro criterio de terminación.

2.2.2.3.2 – Otras estructuras de datos similares

Aparte de los árboles BSP existen otras estructuras de datos que pueden servir para llevar a cabo funciones semejantes. Entre ellas se encuentran los kD-Trees [10], que son una especialización de los árboles BSP donde los planos de subdivisión están siempre alineados con los ejes de coordenadas, los quadtrees y octrees [10][1], que representan, en 2D y 3D respectivamente, subidivisiones recursivas de las celdas en 4 y 8 hijos del mismo tamaño, y otras estructuras como BOXTREEs [11].

Las estructuras mencionadas proporcionan distinta eficiencia dependiendo del tipo de función que se quiere llevar a cabo. Más adelante se motiva el uso de árboles BSP en el proyecto en detrimento de otras estructuras.

2.2.2.3.3 – Funciones que se pueden implementar mediante un árbol BSP

Entre las múltiples funcionalidades que se pueden implementar usando árboles BSP, las dos más comúnmente utilizadas son la eliminación de superficies ocultas y la detección de colisiones.

Los árboles BSP fueron concebidos originalmente como estructuras que permitían llevar a cabo de manera eficiente la eliminación de superficies ocultas (*hidden surface removal*). Por la peculiaridad de la estructura de los árboles BSP es posible, dado un punto de vista en el espacio, recorrer el árbol de tal manera que se obtenga una ordenación de atrás a adelante de los polígonos con respecto al punto de vista dado, de tal manera que se pueda utilizar el algoritmo del pintor [12] para renderizar la escena correctamente sin necesidad de utilizar un Z-buffer[6] [13].

Posteriormente se comenzaron a utilizar los árboles BSP como estructuras muy útiles para la detección eficiente de colisiones con objetos complejos. La estructura de estos árboles nos permite descartar un subespacio completo si detectamos que el objeto se encuentra totalmente a un lado del plano de corte del nodo actual, ya que entonces es imposible que colisione con un objeto que se encuentre al otro lado.

Uno de los inconvenientes de los árboles BSP es que no soportan de manera eficiente la modificación de la geometría que representan, por lo que no son muy apropiados para escenarios dinámicos. De todas maneras pueden ser útiles en algunos casos concretos de escenarios no estáticos [18].

El uso de árboles BSP para la eliminación de superficies ocultas está descrito en muchos lugares [10][15][16][17][18]. En [17] se describe el uso de un árbol BSP como pilar principal de un motor de portales [19] y para el cálculo de iluminación estática, y se proporciona pseudocódigo para la implementación de las funciones más importantes. En [18] se discuten extensivamente los distintos usos de los árboles BSP, entre los que se encuentran la aplicación de este tipo de estructuras para la determinación de visibilidad en escenarios restringidos.

En cuanto al uso de este tipo de estructuras para la detección de colisiones, en [10] se describe el uso de árboles BSP para implementar un sistema de detección de colisiones de una sola pasada, usando segmentos que describen el cambio de posición de un objeto y comprobando la intersección de estos rayos con el escenario, y también se comentan especializaciones que son útiles para detectar colisiones de esferas. En [1] se describen también especializaciones que permiten detectar colisiones de cilindros y envolturas convexas (*convex hulls*) [12] contra el escenario representado por el árbol BSP.

2.2.2.3.4 – Creación de un árbol BSP

La generación de un árbol BSP se reduce a la elección sucesiva de planos de corte que dividan los subespacios en dos mitades. Estos planos normalmente produciran cortes en los polígonos del subespacio al que dividen, creando nuevos polígonos que pertenecerán exclusivamente a una de las dos mitades. La elección de planos terminará cuando se cumpla el criterio de terminación.

La creación del árbol se puede llevar a cabo basándose en distintos criterios, dependiendo del tipo de uso que se le quiera dar. Si por ejemplo se va a usar como base del algoritmo de eliminación de superficies ocultas, normalmente convendrá que el número de cortes producidos por los planos sea mínimo, de tal manera que el número de polígonos de la escena no crezca demasiado. Si por otro lado se quiere usar el árbol como base del sistema de

[6] Aunque se disponga de Z-buffer, para el correcto renderizado de objetos semitransparentes normalmente es necesario hacer una ordenación de los objetos de atrás a adelante [1], aunque otros métodos como el *depth peeling* se pueden utilizar para evitar esta ordenación previa [14].

detección de colisiones, se intentará en todo momento equilibrar el árbol eligiendo planos que dejen cerca de la mitad de los polígonos a un lado y al otro, para que la profundidad del árbol sea mínima y por tanto los tests de colisión sean más eficientes. Estos dos criterios no son compatibles; esta es una de las razones por las que la creación de un árbol BSP óptimo es un problema NP-completo y por tanto intratable para escenarios grandes.

Se tiende entonces a usar algoritmos que produzcan árboles casi óptimos y que tengan una complejidad en tiempo reducida. No es factible buscar por fuerza bruta en cada nodo el plano que más conviene para subdividir el espacio, ya que el número de planos que habría que comprobar puede ser muy elevado (igual al número de polígonos del escenario si se usan *auto-particiones* [12]) o incluso infinito. Se utilizan normalmente criterios sencillos que permiten seleccionar un plano relativamente bueno de manera eficiente. En [18] se hace mención a varios de estos criterios, entre los que destacan el criterio de el plano menos intersecado (*Least-crossed criterion*), donde se comprueban un número reducido de planos (alrededor de 5, aunque depende del tamaño de la escena) y se elige el que menos cortes produzca, y el critero del plano más intersecado (*Most-crossed criterion*), donde se elige de un pequeño conjunto de planos aquél que más cortes produzca (de manera intuitiva parece que este plano daría lugar a subdivisiones que luego producirían menos cortes).

Por otro lado, se ha estudiado también el uso de algoritmos evolutivos para la creación de árboles BSP casi óptimos. Más adelante se hace hincapié en este tema.

La creación de árboles BSP se ha tratado extensivamente [15][16][17][20][21]. En [17] se consideran distintas organizaciones del árbol BSP, con los polígonos en las hojas o con los polígonos en los nodos, y se proporciona pseudocódigo para la creación del primer tipo de árboles, que es la que se ha decidido utilizar en el proyecto (ver siguiente apartado para la motivación). En [20] se discute la creación de árboles BSP que representan particiones perfectas, es decir, particiones en las que ningún objeto del escenario es cortado por los planos separadores. Finalmente, en [21] se discute la creación de un árbol BSP por el método de fuerza bruta, pero como se ha comentado anteriormente ese método no es aplicable a escenarios del tamaño que se está considerando.

2.2.2.3.5 – Objetivo en el proyecto

El objetivo principal de la estructura de datos elegida en el proyecto es que sea capaz de calcular de manera eficiente la colisión entre los objetos y escenarios de varios cientos de miles de polígonos. Dado que no se va a usar el árbol BSP para la eliminación de superficies ocultas (se usará PVS[7], ver más adelante) se considera más apropiado que el árbol esté equilibrado, en contraste con que tenga un número menor de polígonos.

En cuanto al almacenamiento de los polígonos que componen el escenario, se consideraron dos posibilidades: almacenarlos en los nodos internos o almacenarlos en las hojas. El hecho de tener los polígonos en los nodos internos tiene la ventaja de que el algoritmo de detección de colisiones no necesita normalmente bajar hasta una hoja para detecta la colisión contra un polígono, ya que éstos se encuentran dispersados en niveles superiores, pero por otro lado tiene el inconveniente de que los planos de corte tienen que ser auto-particiones, es decir, tienen que coincidir con los polígonos del escenario. Dado que esta restricción se consideró importante, se decidió utilizar un árbol BSP en el que los polígonos están exclusivamente en las hojas (*leafy BSP*).

Por otro lado se consideraron distintos criterios de terminación. El primero que se consideró y que se desechó fue la subdivisión del escenario hasta que hubiese a lo sumo un polígono en cada región. El inconveniente principal de este criterio es que el tamaño del árbol es extremadamente grande para escenarios del tamaño de los que se estaban considerando. El

[7] Potentially Visible Sets

otro criterio de terminación que se consideró, y que fue el que se decidió utilizar, es el descrito en [17]: subdividir hasta que se tenga un conjunto convexo de polígonos[8].

2.2.2.3.6 – Uso de algoritmos evolutivos para la creación de árboles BSP

Como se ha mencionado en el apartado anterior, la creación de árboles BSP óptimos es un problema intratable. Entre los distintos métodos que se han investigado para conseguir particiones casi óptimas en un tiempo reducido está el uso de algoritmos evolutivos.

En [22] se detalla un método para la generación de estos árboles mediante un algoritmo evolutivo. El algoritmo se basa en buscar una ordenación óptima de los planos de corte que den lugar al mejor árbol (los criterios que se siguen para evaluar la optimalidad de un árbol son seleccionables). Los planos de corte que se consideran son únicamente los que coinciden con los polígonos del escenario. El algoritmo evolutivo lleva entonces a cabo la evolución de los indivíduos (i.e. los árboles BSP), representados como permutaciones de los planos posibles, a través de una serie de iteraciones en las que aleatoriamente se realiza el cruce y la mutación de un subconjunto de éstos. En cada paso los indivíduos son evaluados mediante una función de coste basada en ciertas propiedades estadísticas, y los que mayor aptitud tengan son seleccionados para formar parte de la siguiente población.

Según los experimentos detallados en el trabajo, la aplicación repetida de los pasos descritos lleva a generar árboles BSP mejores que los obtenidos por otros métodos y en un tiempo relativamente corto.

El algoritmo descrito tiene un problema importante que ha llevado a considerar un método alternativo para la generación de árboles BSP para el proyecto. El problema es el enorme requerimiento en espacio para escenarios de más de varios miles de polígonos. Se puede ver que los requerimientos de memoria son extremadamente altos para escenarios como los considerados para el proyecto, de varios cientos de miles de polígonos. Suponiendo un escenario de 500.000 polígonos, los indivíduos estarían representados por una lista de medio millón de planos, por lo que en el mejor de los casos requerirían 2 megabytes de espacio cada uno[9]. Dada la enorme variedad de posibles indivíduos la población tendría que ser relativamente grande, pero aún con una población pequeña de 1000 indivíduos los requerimientos de memoria se sitúan en los 2 gigabytes. La evaluación de tal número de indivíduos y el cruce y la mutación sobre ellos sería extremadamente costoso, por lo que no se podría llevar a cabo un gran número de iteraciones, reduciéndose considerablemente la calidad del mejor árbol que se puede conseguir.

Por estas razones se ha decidido utilizar un método híbrido que se describe a continuación.

La selección de los planos de corte se realiza de distinta manera dependiendo del número de polígonos que quedan en el subespacio actual. Si el número de polígonos es reducido entonces se utiliza el criterio de plano menos intersecado, comprobando un número de planos que depende de $\log n$ (con n número de polígonos en el subespacio) de todos los planos que coinciden con los polígonos (es decir, sólo se consideran auto-particiones).

Por otro lado, si el número de polígonos es mayor que cierto umbral, se procede a aplicar un algoritmo evolutivo para seleccionar el mejor plano de corte. Las diferencias con el algoritmo propuesto en [22] son múltiples. Primero, no se consideran solamente planos que coincidan con los polígonos existentes, sino que los planos considerados son planos arbitrarios que se encuentran en la cercanía de la región a subdividir. Además, el algoritmo evolutivo se ejecuta para encontrar cada plano de corte, no para encontrar directamente el árbol BSP completo más óptimo. Por esta razón el coste en tiempo y en espacio es sensiblemente menor,

[8] Conjunto de polígonos donde ningún polígono está detrás de ningún otro y por tanto se pueden dibujar en cualquier orden.

[9] Suponiendo que los índices se representan como enteros sin signo de 32 bits.

aunque también la calidad del árbol BSP conseguido finalmente es también menor ya que se están realizando optimizaciones locales, no una optimización global como es el caso de [22]. Finalmente, el cruce y la mutación son necesariamente distintos, ya que la propia representación de los indivíduos es diferente - en este caso los indivíduos son simplemente planos y se representan por sus coeficientes A, B, C y D. El cruce consiste en encontrar los dos planos cuyos puntos se encuentran a igual distancia de los dos planos padres, mientras que la mutación consiste en modificar ligeramente alguno de los coeficientes del plano, moviendo o cambiando la orientación de éste.

2.2.2.3.7 – Detalles de implementación

A continuación se describen las distintas peculiaridades de la implementación del algoritmo que genera árboles BSP casi óptimos del proyecto, así como los problemas que acarrea llevar a cabo esta implementación en un lenguaje como C++.

- Parámetros configurables: Es posible modificar una serie de parámetros para determinar la calidad del tipo de árbol BSP que se quiere conseguir. En primer lugar se encuentran los parámetros comunes de todo algoritmo evolutivo: tamaño de la población, número máximo de iteraciones y probabilidades de cruce y mutación. Aparte de estos parámetros se puede seleccionar el umbral bajo el cuál se dejará de utilizar el algoritmo evolutivo y se comenzará a utilizar el criterio de least-crossed. También es posible seleccionar el peso que se le da al equilibrio del árbol y al número de cortes, de tal manera que se puede calibrar el árbol para un uso u otro (detección de colisiones vs eliminación de superficies ocultas).

- Generación de la población inicial: Dado que los planos posibles son totalmente arbitrarios en cuanto a que no tienen que coincidir con polígonos existentes, es necesario asegurarse de que son planos *útiles*, es decir, que producen una subdivisión significativa de la escena, en vez de dejar todos los polígonos a uno de los lados. Para ello en la generación de la población inicial se crean planos que intersecan la AABB del escenario completo, de tal manera que muy posiblemente serán planos útiles. Los planos que aún así no dividan la escena en absoluto serán probablemente eliminados tras la primera iteración por el algoritmo evolutivo, ya que su evaluación será muy baja.

- Selección de la siguiente población: La selección de los indivíduos que formarán parte de la población en la siguiente iteración se hace de forma aleatoria, pero teniendo en cuenta las aptitudes de los distintos individuos, de tal manera que aquellos que son más aptos tienen más posibilidades de ser seleccionados para seguir evolucionando. En cuanto al reemplazamiento de los padres por los hijos generados a partir de ellos, se realiza un reemplazamiento directo en el cuál los hijos sustituyen directamente a sus padres en la siguiente población. Se podría haber hecho uso del *elitismo*, por el cuál los mejores indivíduos en una iteración pasan a formar parte directamente de la población siguiente, tal como se hace en [22].

- Criterio de terminación: El criterio de terminación implementado es simplemente el número máximo de iteraciones. Se podrían haber implementado criterios de terminación más sofisticados, de tal manera que se parara el algoritmo cuando se viera que la aptitud general no hubiera mejorado en las últimas iteraciones y no pareciera tener probabilidades de mejorar.

- Problemas de precisión: La implementación del algoritmo en C++ acarreó problemas de precisión en los cálculos en coma flotante. Estos problemas se manifiestan principalmente como grietas entre los polígonos resultantes en el escenario, debido a que los cálculos de los nuevos vértices no coinciden exactamente con sus posiciones anteriores o no se encuentran exactamente en la arista del triángulo del que proceden. También se manifiestan cuando se hacen divisiones por números muy próximos a cero (como puede ocurrir por ejemplo al dividir por el módulo de un vector entre dos puntos muy próximos), lo que obliga a añadir código que detecte estas situaciones y las corrija en la medida de lo posible. Estos problemas de precisión ocurren porque en C++ se utiliza el estándar IEEE 754 [23] de números en coma flotante, que

tienen precisión finita. Para reducir en la medida de lo posible estos errores se ha usado en todo momento el formato de precisión doble (`double`).

- Grosor de los planos: Otro de los problemas que hay que tratar en un algoritmo de generación de árboles BSP es el tema del grosor de los planos. Puede darse el caso de que se detecte que un polígono intersecta un plano pero que el vértice que se encuentra al otro lado se encuentre muy próximo al plano, de tal manera que al dividir el polígono se genere un polígono extremadamente pequeño. Este tipo de problemas se pueden subsanar si se dota a los planos de cierto *grosor*, de tal manera que los polígonos que lo intersecten ligeramente se consideren totalmente a uno de los lados del plano y se evite dividirlos.

- Triángulos degenerados: Un triángulo degenerado es un triángulo cuyos tres vértices están en línea, de tal manera que el área es cero. Este tipo de triángulos traen bastantes problemas a la hora de tratarlos en un algoritmo como el de generación de un árbol BSP, y por tanto se ha tenido que introducir código para detectarlos y eliminarlos.

2.2.2.3.8 – La utilidad BspGenerator

La utilidad BspGenerator implementa el algoritmo descrito de generación de árboles BSP casi óptimos. La entrada es un mapa de Everquest, y la salida que proporciona es la siguiente:

`mapa.map`:	Fichero XML que describe el mapa y los distintos objetos que contiene.
`mapa.rawbsp`:	Fichero RAWBSP que contiene el árbol BSP y la geometría del escenario.
`materials/`:	Directorio que contiene la definición de los materiales básicos del mapa.
`objects/`:	Directorio que contiene la definición en XML y los modelos de todos los objetos del mapa.
`textures/`:	Directorio que contiene las texturas usadas en el mapa transformadas a formato PNG.

Su uso es el siguiente:

```
BspGenerator escenario ruta_salida [ peso_splits peso_equilibrio [
limite_ag num_it tam_pob cruce mutacion ] ]
```

El argumento *escenario* identifica el mapa de Everquest del que se quiere extraer el escenario. El argumento *ruta_salida* identifica el directorio donde se crearán los distintos ficheros que componen el mapa. Los argumentos *peso_splits* y *peso_equilibrio* controlan la forma en que se quiere obtener el árbol, ya sea dando más importancia a la reducción de cortes o al equilibrio del árbol. El argumento *limite_ag* es el número de polígonos a partir del cuál se empieza a usar el algoritmo evolutivo para la selección del mejor plano de corte. El argumento *num_it* especifica el número máximo de iteraciones, y *tam_pob* el tamaño de la población. Finalmente, *cruce* y *mutacion* son las probabilidades de cruce y mutación respectivamente.

2.2.2.4 – Colisión entre objeto y objeto

La colisión entre dos objetos se lleva a cabo en dos fases.

La primera fase consiste en usar el árbol BSP para encontrar las parejas de objetos que se encuentran lo suficientemente cerca (concretamente, las parejas de objetos que comparten una región del escenario). Una vez encontradas estas parejas se pasa a la siguiente fase.

La segunda fase consiste en usar la jerarquía de volumenes contenedores, que en el caso de los objetos será un OBB-Tree, para hacer el test de intersección pormenorizado de cada pareja. Como se comentó anteriormente, esta detección se realiza mediante un descenso paralelo de ambos árboles, parando cuando las OBBs no intersequen o cuando se llegue a la raíz, en cuyo caso se calcula el punto intermedio entre ambos centros y se devuelve como punto de la intersección.

No se realiza una detección de colisión exacta porque se considera que la aproximación comentada es suficiente para esta aplicación en concreto.

El uso de OBB-Trees en detrimento de otro tipo de estructuras se ha considerado apropiado porque por un lado puede describir la geometría de objetos complejos y por otro lado es una estructura que puede ser generada automáticamente por una serie de métodos, como los descritos en [4] y [5].

2.2.2.5 – Colisión entre cámara y escenario

El último tipo de colisión que se ha considerado es el de cámara y escenario. Es necesario tener en cuenta esta colisión para que la cámara esté en todo momento libre de obstáculos y el jugador pueda ver a su personaje.

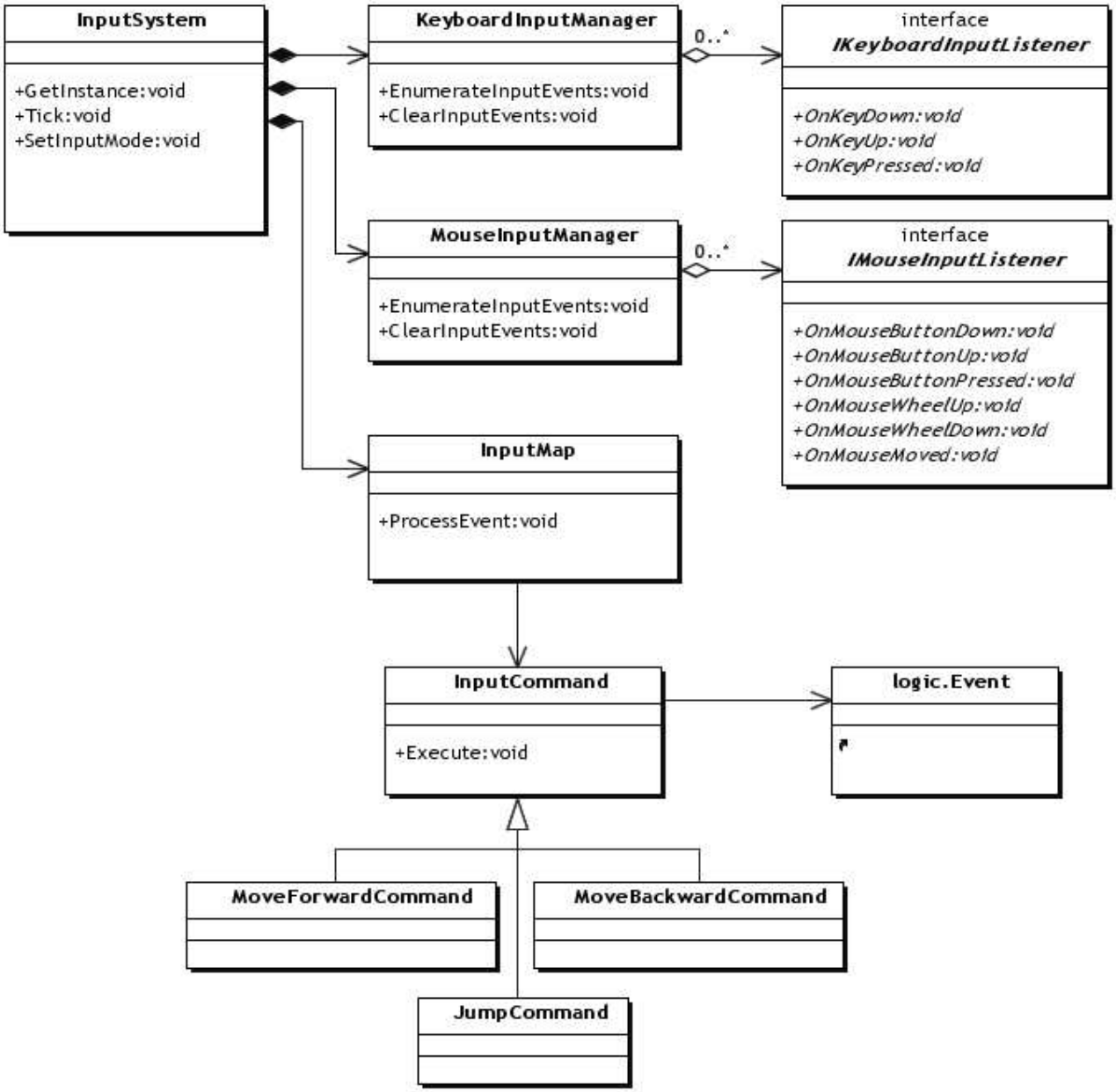
Este tipo de detección de colisiones se ha planteado de la siguiente manera. Primero se detecta si hay alguna obstrucción entre la cámara y el personaje, a través del test de intersección entre un rayo y el escenario. Si es así entonces hay un objeto o parte del escenario que se encuentra entre la cámara y el personaje, por lo que la cámara se mueve a un punto inmediatamente posterior al último punto de intersección entre ambos. Tras esto la cámara se encuentra libre de obstrucciones y el personaje es visible.

2.2.3 – MÓDULO CONTROL

2.2.3.1 - Objetivos

El objetivo principal del módulo Control es proporcionar una manera potente de traducir los eventos de teclado y de ratón en comandos que tienen sentido en la aplicación. Además, esta traducción debe ser configurable y puede estar compuesta por distintos *modos de entrada*, que básicamente no son más que distintas tablas que asocian los posibles eventos a los respectivos comandos.

2.2.3.2 – Descripción del Diseño



El punto de entrada al módulo Control es la clase **InputSystem**. Esta clase proporciona dos servicios importantes: el paso de un ciclo de simulación, que consiste en recoger todos los eventos que se hayan generado en el pasado ciclo y ejecutar los respectivos comandos a los que se traducen, y el cambio del modo de entrada, para pasar por ejemplo de modo escritura a modo interactivo o viceversa.

Las clases que se encargan de recibir los eventos de teclado y de ratón desde el sistema operativo son **KeyboardInputManager** y **MouseInputManager** respectivamente. Estas clases acumulan los eventos que se van produciendo hasta que otra clase llama a EnumerateInputEvents para recibir la lista de estos eventos y seguidamente los elimina llamando a ClearInputEvents.

Una vez recogidos los eventos que se han generado en el pasado ciclo, es responsabilidad de la clase **InputMap** traducir estos eventos a comandos de la aplicación. En cuanto a la implementación, la clase InputMap se implementa como una tabla hash que se indexa por el tipo de evento producido y que guarda una referencia al prototipo del comando correspondiente. Este prototipo entonces se clona y se devuelve al objeto llamante. Este comportamiento corresponde al patrón de diseño *Prototype* descrito en [24].

Los comandos son clases concretas que derivan de **InputCommand**, clase que declara un solo método Execute que lleva a cabo la acción asociada al comando. Los comandos de la aplicación normalmente no llevarán a cabo directamente el cambio de estado de los objetos de la simulación, sino que se limitarán a comprobar que la acción que representan es válida en el momento en que son ejecutados y acto seguido crearán un objeto de la clase **logic::Event** que añadirán a la simulación en el tiempo correspondiente. Será cuando este evento de simulación sea procesado cuando la acción se lleve realmente a cabo.

2.2.4 – MÓDULO GRAPHICS

2.2.4.1 - Objetivos

El objetivo del módulo Graphics es hacer una representación realista y eficiente de la escena. Se pretende hacer una eliminación de superficies ocultas agresiva que permita conseguir un *framerate* interactivo aunque el escenario consista de cientos de miles de polígonos.

Además, se pretende dar soporte a una serie de efectos como iluminación por pixel y sombras dinámicas e integrarlos en el bucle de rendering de forma eficiente.

2.2.4.2 – Descripción del Diseño

El interfaz principal del módulo Graphics es la clase **GraphicsSystem**. Esta clase proporciona una serie de servicios, entre los que se encuentran el dar acceso a los distintos *managers* del módulo.

Los managers son clases que se encargan de llevar un registro de los distintos recursos gráficos que maneja el motor. Entre estos recursos se encuentran las texturas, los materiales (detallados más adelante), los modelos y los buffers. Las clases que hacen de manager implementan el patrón de diseño *Factory Method* [24].

La clase GraphicsSystem proporciona también un método Tick que desencadena el renderizado de la escena. Este método es llamado a la máxima frecuencia posible, en contraste con la lógica de la aplicación que se ejecuta a una frecuencia fija. Este tipo de separación tiene una serie de ventajas, entre las que se encuentra la posibilidad de aprovechar al máximo la potencia gráfica de la plataforma. La implementación de esta separación se ha basado en [26].

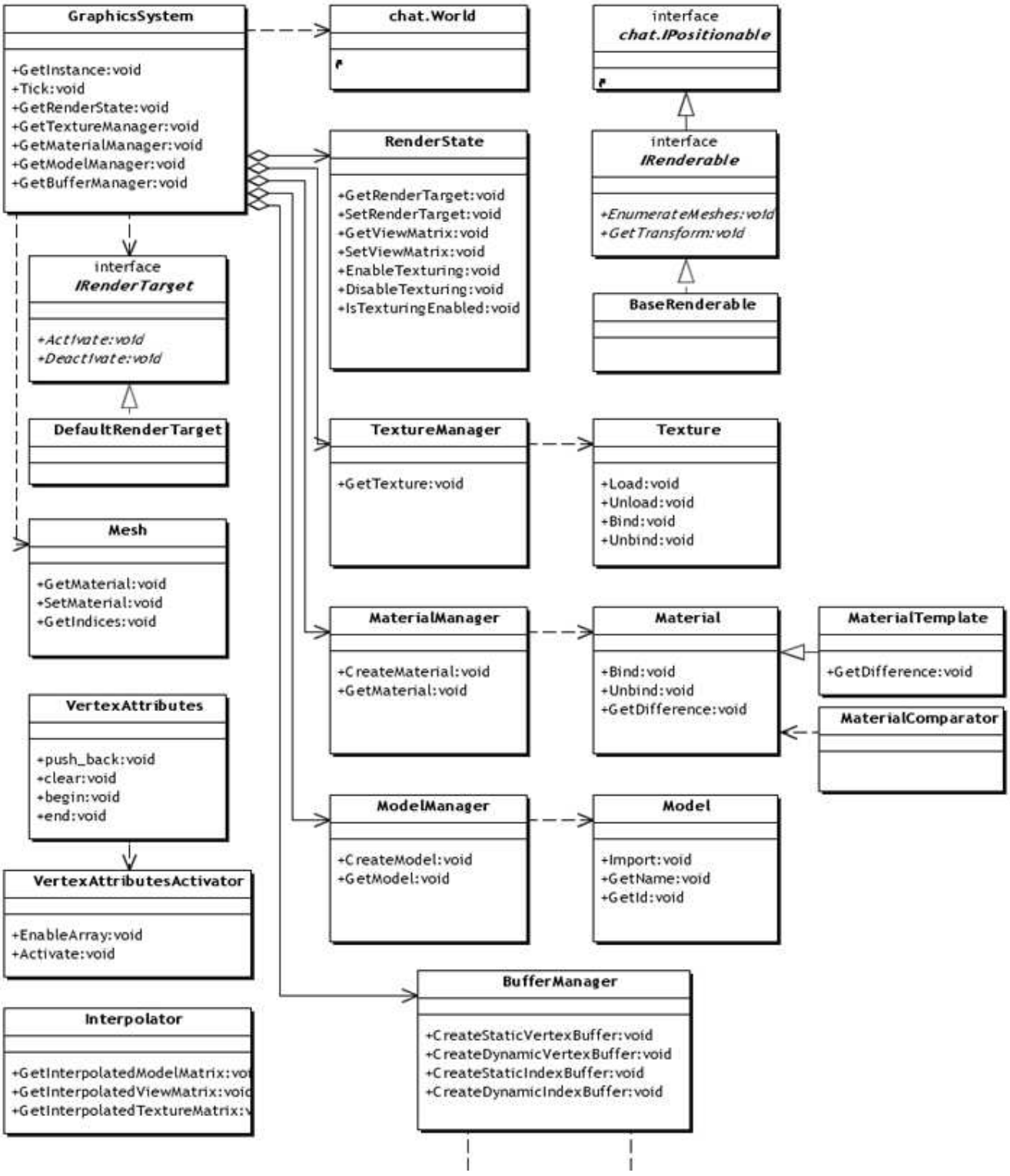
El interfaz **IRenderTarget** describe un recurso que puede ser el destino de un render. El destino más común, y el que es representado por la clase **DefaultRenderTarget**, es el contexto de una ventana. El diseño permite acomodar otro tipo de destinos, como puede ser un *Pbuffer* [25] (buffer de renderizado no visible), o múltiples destinos simultáneos para hacer uso de MRT[10].

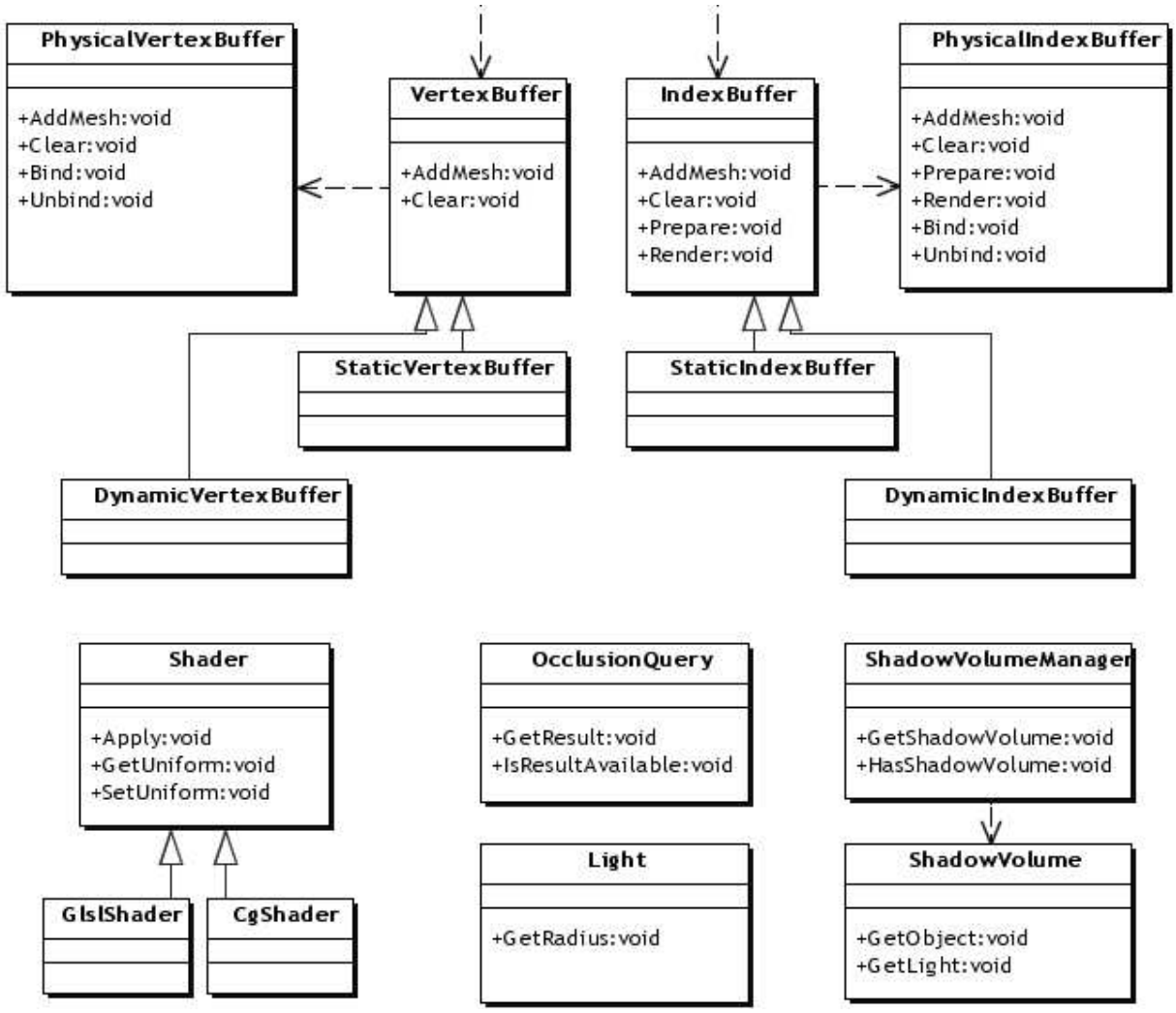
La clase **RenderState** representa el estado de renderizado en cada momento. Da acceso a las distintas matrices de transformación y proporciona información sobre qué partes del pipeline gráfico están activadas.

Todos los objetos que tengan algún tipo de representación gráfica han de implementar el interfaz **IRenderable**, o heredar de la clase **BaseRenderable**, que proporciona una implementación por defecto de todos los métodos del interfaz. IRenderable proporciona dos métodos imprescindibles para representar un objeto en la escena: la lista de mallas que lo componen y la matriz de transformación del objeto que lo pasa de coordenadas locales a coordenadas de mundo.

Las mallas están representadas por la clase **Mesh**. Una malla no es más que un conjunto de primitivas, normalmente triángulos sueltos, que comparten material, matriz de transformación y buffer de vértices (ver más adelante). Una malla permite enviar a la tarjeta gráfica toda la geometría que la compone sin tener que enviar los polígonos de forma individual y sin tener que hacer ningún cambio de estado.

[10] Multiple Render Targets


GraphicsSystem
+GetInstance:void
+Tick:void
+GetRenderState:void
+GetTextureManager:void
+GetMaterialManager:void
+GetModelManager:void
+GetBufferManager:void
chat.World
interface
chat.IPositionable
RenderState
+GetRenderTarget:void
+SetRenderTarget:void
+GetViewMatrix:void
+SetViewMatrix:void
+EnableTexturing:void
+DisableTexturing:void
+IsTexturingEnabled:void
interface
IRenderable
+EnumerateMeshes:void
+GetTransform:void
BaseRenderable
interface
IRenderTarget
+Activate:void
+Deactivate:void
DefaultRenderTarget
TextureManager
+GetTexture:void
Texture
+Load:void
+Unload:void
+Bind:void
+Unbind:void
Mesh
+GetMaterial:void
+SetMaterial:void
+GetIndices:void
MaterialManager
+CreateMaterial:void
+GetMaterial:void
Material
+Bind:void
+Unbind:void
+GetDifference:void
MaterialTemplate
+GetDifference:void
MaterialComparator
VertexAttributes
+push_back:void
+clear:void
+begin:void
+end:void
ModelManager
+CreateModel:void
+GetModel:void
Model
+Import:void
+GetName:void
+GetId:void
VertexAttributesActivator
+EnableArray:void
+Activate:void
BufferManager
+CreateStaticVertexBuffer:void
+CreateDynamicVertexBuffer:void
+CreateStaticIndexBuffer:void
+CreateDynamicIndexBuffer:void
Interpolator
+GetInterpolatedModelMatrix:vo
+GetInterpolatedViewMatrix:voi
+GetInterpolatedTextureMatrix:



Las clases **VertexBuffer** e **IndexBuffer** representan los buffers de vértices y de índices, respectivamente. El uso de buffers permite hacer un uso más eficiente de los recursos de la tarjeta. El uso de este tipo de buffers se detalla más adelante.

La clase **Model** representa la geometría de un objeto que no forma parte del escenario. Los objetos de la clase Model son compartidos por todas las entidades que tengan el mismo modelo, aunque el aspecto de cada una puede variar dependiendo del material y la transformación que le sea aplicada.

Por otro lado, la clase **Interpolator** es una clase que contiene solamente métodos estáticos (también llamado *monostate*) y que es usada durante el ciclo de renderizado para conocer la posición interpolada y otras propiedades de los objetos de la simulación. Dado que el ciclo de renderizado se puede ejecutar varias veces antes de que se ejecute el siguiente ciclo de lógica es necesario llevar a cabo una interpolación de las posiciones y demás propiedades visibles de los objetos para que la animación sea fluida. Si no se llevara a cabo interpolación, el efecto sería el de un render con framerate igual a la frecuencia del ciclo de lógica.

Finalmente, existen una serie de clases auxiliares de menor importancia en el módulo Graphics. La clase **Light**, por ejemplo, representa una fuente de luz activa en el escenario. Dependiendo del modelo de iluminación elegido las propiedades de las distintas luces afectarán de una manera u otra al aspecto de los objetos que se encuentren en su radio de acción. Por

otro lado, la clase **OcclusionQuery** representa una *occlusion query* [27], o test de visibilidad de un objeto (para más información referirse al apartado de visibilidad más adelante).

La clase **Shader** encapsula un *shader*, o programa que es ejecutado por la tarjeta gráfica durante la fase de geometría (vertex shader) o la fase de rasterizado (fragment shader). Los shaders permiten configurar partes del pipeline gráfico y conseguir efectos que de otro modo serían difíciles o imposibles de conseguir. Existen una serie de lenguajes en los que pueden ser programados los shaders. El diseño del módulo permite utilizar shaders escritos en distintos lenguajes, como pueden ser los lenguajes de alto nivel Cg [28][29] y GLSL[11] [30][31].

Para concluir, el soporte de sombras dinámicas se proporciona a través de las clases **ShadowVolume** y **ShadowVolumeManager**. Estas clases se encargan de llevar el control de los volúmenes de sombra generados por los distintos objetos con respecto a las fuentes de luz que les afectan, intentando en la medida de lo posible el reusar los volúmenes de sombra ya calculados para los siguientes renders.

[11] A fecha del presente documento, el soporte de shaders escritos en GLSL proporcionado por los últimos drivers de NVIDIA y ATI es aún preliminar e incompleto.

2.2.5 - MÓDULO LOGIC

2.2.5.1 - Objetivos

El diseño del módulo Logic se guió por dos objetivos principales. El primero es conseguir independencia respecto de la velocidad de la plataforma haciendo una separación clara entre la ejecución del código correspondiente a la lógica y del correspondiente al rendering (ver capítulo anterior). Esta independencia es muy importante porque es la base de la sincronización entre clientes.

El segundo objetivo principal es implementar una sincronización que haga posible el uso de la aplicación en redes de alta latencia reduciendo en la medida de lo posible los efectos visuales producidos por el *lag*[12]. Este objetivo influyó de manera importante en el diseño, como se explica a continuación.

2.2.5.2 – Descripción del diseño

El pilar principal del módulo Logic es la clase **Simulation**. Esta clase es la encargada de llevar el control de los eventos de la simulación y de decidir en qué momento debe ser ejecutado cada uno. Cuando se ejecuta un comando, ya sea de entrada de usuario o recibido del servidor, se genera un evento de simulación que es añadido a la lista para ser ejecutado en un tiempo determinado, pero no se modifican los estados de las entidades que participan en la simulación. Es cuando se ejecuta el evento de simulación cuando finalmente se actualizan los estados de éstas.

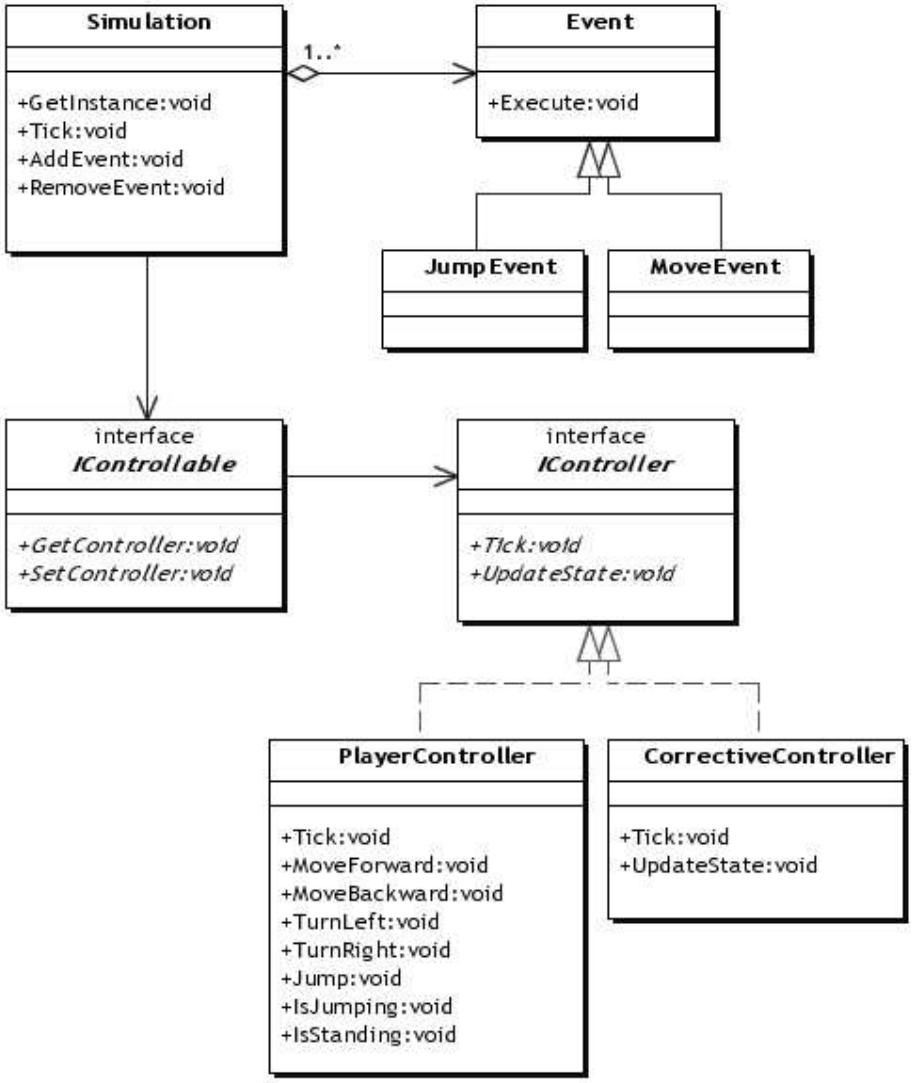
Dado que la aplicación se puede estar ejecutando en una red como Internet, es posible que algunos eventos enviados por el servidor sufran un retraso indefinido y no lleguen a tiempo a los clientes. Si un evento llega a un cliente en un tiempo posterior al tiempo en que tendría que haber sido ejecutado entonces es necesario volver a simular lo ocurrido desde el tiempo marcado por el evento, ya que la simulación no contó con toda la información necesaria y posiblemente el estado de las entidades no sea el correcto.

Si estos eventos retrasados representan cambios en el movimiento de los objetos entonces es necesario reducir el impacto visual que produciría el reposicionamiento inmediato de éstos a sus posiciones reales. Para evitar efectos visuales debido al retardo en la llegada de eventos los clientes harán *predicciones* sobre las posiciones de los objetos en movimiento (normalmente los demás personajes), llevando a cabo una extrapolación que tenga en cuenta las últimas posiciones del objeto y su trayectoria. Debido a que las nuevas posiciones se están calculando a través de predicciones, es posible que cuando finalmente llegue el evento correspondiente éste indique una posición del objeto distinta de la estimada.

Cuando se detecta una diferencia entre las posiciones real y estimada se lleva a cabo una corrección progresiva de la trayectoria del objeto, de tal manera que durante los siguientes pasos de simulación su posición se vaya corrigiendo hasta coincidir con su posición real.

El sistema de sincronización comentado en el que se lleva a cabo una re-simulación de los últimos pasos cuando se recibe un evento retrasado se conoce como **time-warp synchronization** [51][52][53]. La predicción de las posiciones de los objetos en movimiento y su posterior correción si resulta haber sido incorrecta se conoce como **dead reckoning** [32][33].

[12] Intervalo de tiempo entre el envío de un mensaje al servidor y la llegada de su respuesta



La otra parte importante del módulo Logic es lo que se conoce como *controladores*. Los controladores son clases que se asocian a entidades de la simulación y que se encargan de llevar a cabo la actualización del estado de éstas en cada paso de simulación, teniendo en cuenta los distintos eventos que se produzcan sobre ellas. Un tipo especial de controlador es el CorrectiveController, que es la clase que implementa la parte de corrección de trayectoria del sistema de dead reckoning.

2.2.6 - MÓDULO NET

2.2.6.1 – Breve descripción

Esta sección trata sobre el módulo de red que da soporte al chat de la aplicación. Este módulo de red consta de dos partes bien diferenciadas: cliente y servidor. A continuación se discute sobre la implementación de ambos, seguridad y datos que se transfieren.

2.2.6.2 – Sobre la arquitectura

En un chat las dos partes bien diferenciadas tienen como elemento principal las dos partes de un socket.

El cliente lanza un socket (anzuelo) sobre un puerto de la máquina servidor.

El servidor, por su parte, tiene como elemento principal un serversocket, que sería el elemento que agarra los anzuelos formando la conexión. Un único serversocket es capaz de agarrar muchos sockets, que deben ser tratados en hilos distintos.

Cliente

Una implementación tradicional del cliente [50], sería proporcionar un hilo para la recepción de mensajes desde el servidor, de esta forma la aplicación tendría dos hilos asíncronos. Esta implementación se encuentra con dos problemas: la posible recepción de las respuestas fuera de orden y la concurrencia entre ambos hilos.

Estas cuestiones fueron solventadas mediante:

- Recepción fuera de tiempo: Desligaremos entonces las peticiones de las respuestas, haciendo que las respuestas del servidor sean acatadas como órdenes independientemente de la petición. Usando el protocolo TCP además nos aseguraremos de que los paquetes lleguen y lo hagan en el orden en el que el servidor los mandó.
- Concurrencia entre hilos: tradicionalmente esto se solventó mediante secciones críticas, es decir, que sólo un hilo podía ejecutar cierto código a la vez. En programación orientada a objetos se usaron los métodos monitores (synchonized)

Nuestra arquitectura de cliente sólo debe tener un hilo, puesto que varios hilos complicarían la lógica y la estabilidad del cliente. Para ello, nuestro cliente se aprovecha del buffer asociado al socket para que la entrada por él espere hasta que sea solicitada. Esta solicitud la hace la lógica del cliente al módulo de red de forma asíncrona y espera que la llamada sea no bloqueante(Ver 2.3.3.3.2). El módulo de red devolverá entonces una Enumeración(Ver sección 2.4) de los comandos[24] formados al analizar sintácticamente la entrada.

Servidor

Por su parte, las implementaciones tradicionales[50] del servidor son análogas a la implementación tradicional del cliente, sólo que para cada cliente conectado al servidor hace falta un hilo que esté escuchando lo que le proviene del socket, lo que hacía del servidor una obra pesada de ingeniería en el campo de la concurrencia.

En los servidores tradicionales, son los hilos los que llaman directamente a los métodos de la lógica, con lo que tendremos varios hilos manipulando las mismas estructuras de datos, lo que puede llegar a producir incoherencias.

Nosotros proponemos una implementación que va más allá de la tradicional. Solucionaremos la concurrencia mediante una lista de comandos. Cada hilo receptor irá introduciendo en una cola entidades que representen la acción que el servidor quiere que se ejecute (comandos[24]), y el hilo de aplicación las irá tomando de la cola y ejecutándolas. De esta forma ambos hilos actúan sobre secciones distintas de código de la aplicación, y sólo comparten como datos una cola, que es en la que hay que vigilar la concurrencia.

2.2.6.3 – Prototipos y evolución del módulo Net

2.2.6.3.1 - Arquitectura de chat con synchronized:

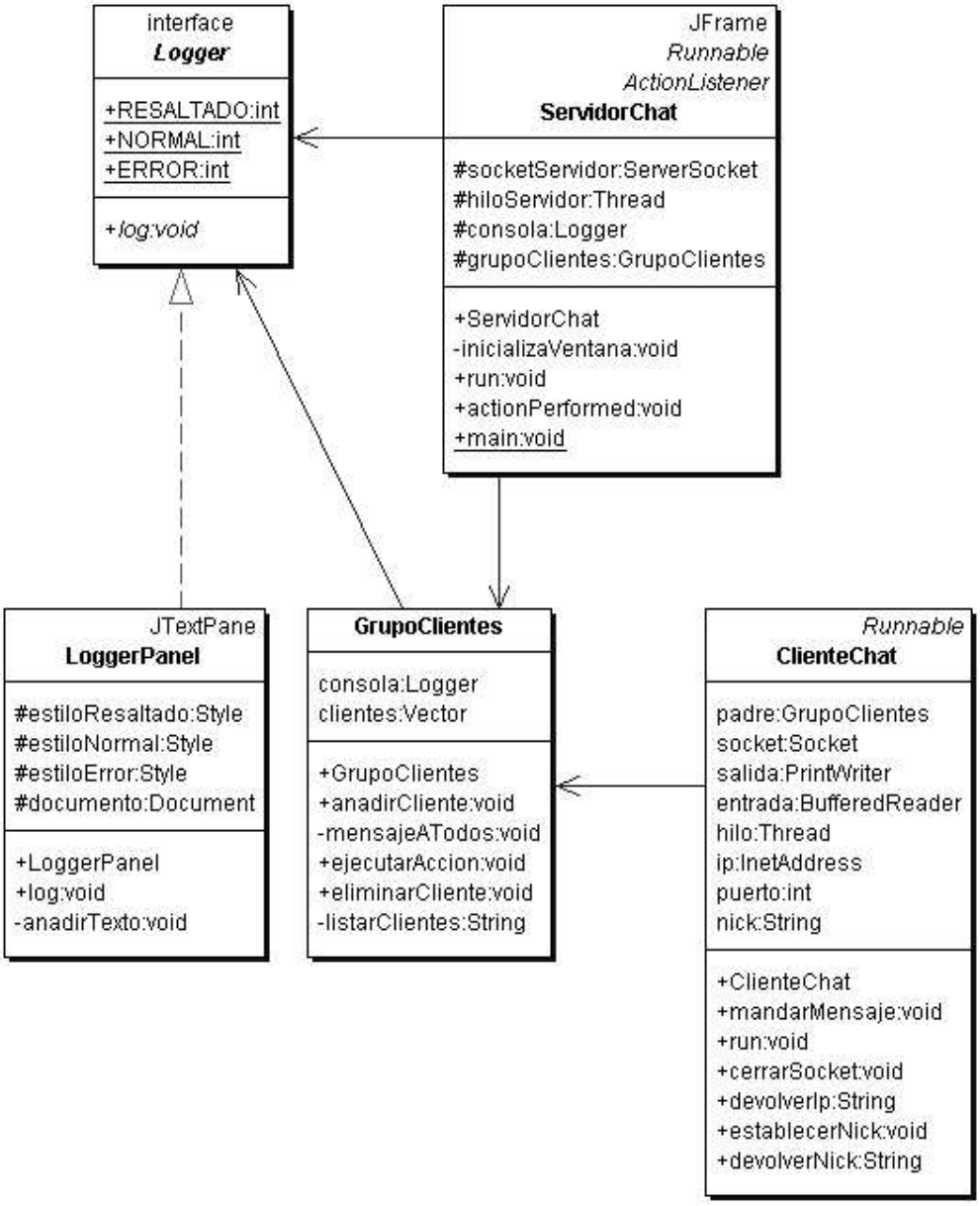
Para estudiar la viabilidad del módulo de red, en primera instancia se implementó un chat sobre java, que usaba los métodos synchronized para gestionar la concurrencia.

Este servidor vimos que no sólo era complicado de entender, si no que también era complicado de extender, y con gran tendencia a la inestabilidad cuando se modificaba el código. Realmente uno no podía considerar todos los casos problemáticos con la cantidad de hilos que se estaban ejecutando por toda la aplicación.

Puesto que el servidor usa multihilo, debíamos asignar a cada hilo una zona en la que sólo él debiera andar, y planificar estos de acuerdo con ello.

El servidor tenía las siguientes clases:

- ClienteChat: Se trataba de un proxy(desde el punto de vista de diseño de arquitecturas software) hacia el cliente, es decir, una clase que en su interfaz se comportaba como el cliente remoto, y que su implementación era una conexión a dicho cliente por medio de un socket. Había tantos activos como usuarios conectados.
- ServidorChat: La clase principal. Se encargaba de proporcionar la interfaz gráfica y de inicializar todos los componentes
- GrupoClientes: Se refiere a los canales que tenía el chat. Un canal es una agrupación de personas por un tema. Un usuario sólo puede oír las cosas que dicen el resto de usuarios que están en el canal.
- Logger: Interfaz del chat para hacer log(grabar los eventos), pudiendo ser implementada por un escritor a un archivo, a un socket o como en este caso a un panel de swing.
- LoggerPannel: Panel de swing donde se hace log.



En este caso, la aplicación principal contaba con el hilo de swing, para interactuar con la interfaz gráfica, y cada cliente tenía un hilo. Los hilos de cliente ejecutaban llamadas a GrupoCliente.

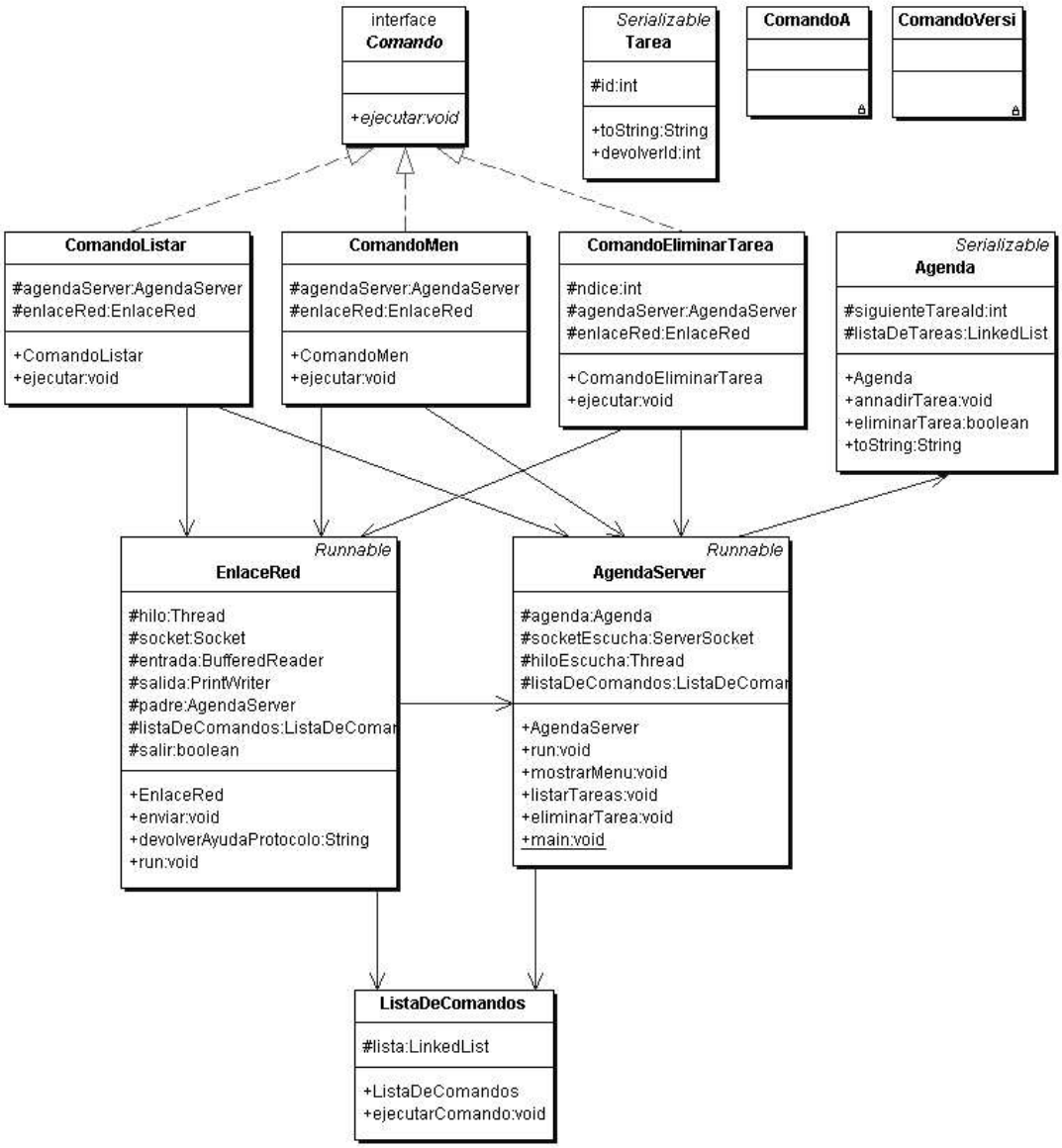
Esto tenía dos inconvenientes, el primero es que cada GrupoCliente debía ser un monitor, y el segundo es que el hilo volvía hacia atrás en la jerarquía, es decir, que salía de Cliente, iba a GrupoClientes y volvía a Cliente, por lo que Cliente también debía ser un monitor. Esto a la larga, en un servidor grande, podía haber degenerado en un interbloqueo.

La solución fue desechada.

2.2.6.3.2 - Arquitectura de chat con cola de mensajes, primer intento:

Para estudiar la viabilidad de la cola de mensajes, se implementó un servidor de agenda rápido sobre J#.NET, que usaba el patrón comando para la ejecución de las órdenes que le venían por la red.

El esquema de diseño es el siguiente:



Las clases del servidor de agenda son las siguientes:

- AgendaServer: El servidor de agenda, inicializaba todas las clases y se mantenía a la escucha de conexiones entrantes, para ir creando las clases que se ocuparían de

atender a los clientes (EnlaceRed). Para ello, esta clase tiene un hilo que sólo se dedica a la escucha, y por supuesto, ese hilo es el único que ejecuta el código de run de AgendaServer, y cuando le llega la conexión, crea un comando de conexión nueva y lo mete a la lista. AgendaServer es también el encargado de la ejecución de los comandos de la lista, por medio de su otro hilo, el de aplicación, que es el que circula por las partes a las de lógica (Funciones de AgendaServer, Agenda, y partes salientes de EnlaceRed).

- EnlaceRed: Cuando un comando de ClienteNuevo le llega a AgendaServer, este lo ejecuta con el hilo de aplicación y crea un nuevo enlace red. Esta clase se encarga de mandar y de recibir cosas de cada socket. Tiene un método de escucha, que se ejecuta sobre un hilo propio, y que al entender una orden desde el cliente, la mete en un comando y a la cola de comandos, donde será ejecutada.
- Agenda: lógica básica de la aplicación. Iterar sobre listas de tareas, sobre todo.
- Tarea: Estructura de datos de la agenda.
- Comando: Patrón comando. Esta estructura encapsula una llamada o varias a una o varias funciones, es decir, un método. De esta forma, se puede retrasar la ejecución de ese método, cancelarlo antes de que ocurra, o incluso (como es el caso) hacer que la ejecute un hilo distinto.
- ListaDeComandos: la cola en la que se metían los comandos provenientes de todos los hilos, para que el principal los tomase y los ejecutase. Es la única zona en la que se cruzan los hilos. Por ello, esta lista es un singleton[24](instancia estática accesible desde cualquier punto del código) y monitor, para dar acceso a todos y que no haya conflictos. Para hacer el servidor eficiente, además, usamos wait y notify de forma que el hilo principal sólo se despertase si tenía comandos pendientes de ejecutar, y se durmiera cuando no los tuviera.

De esta forma, hacíamos que fuera el hilo de aplicación el que ejecutase la mayor parte del código, el resto de hilos, sólo escuchaban a los sockets y transformaban la información que venía en estructuras de datos que encapsulan código, para que el hilo principal pudiese ejecutar dicho código sin problemas.

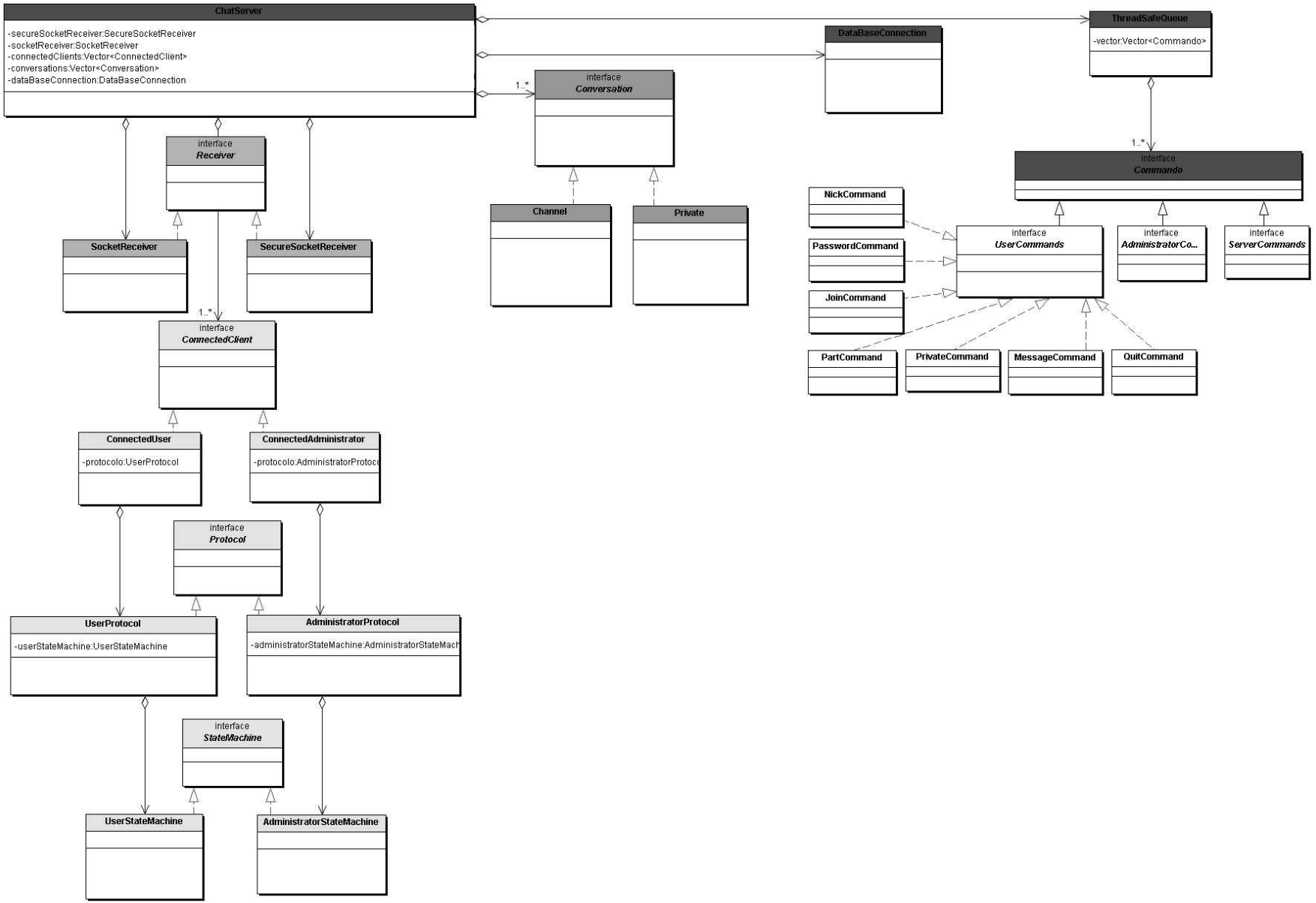
El servidor resultó ser asombrosamente estable, sin caídas, a falta de fuertes pruebas de carga.

La solución de la cola de mensajes fue tomada como núcleo principal del servidor de red.

2.2.6.3.3 - Arquitectura de chat definitivo con cola de mensajes:

Una vez tomada la decisión de usar multihilo con cola de mensajes, decidimos plantear otros objetivos a la arquitectura:

- Soporte de distintos tipos de conexión en/y distintos puertos de escucha.
- Soporte de protocolos distintos para el mismo chat, y compatibilidad hacia abajo.
- Múltiples canales, múltiples privados.
- Los clientes pueden cambiar de canal, pero preservar los privados.
- Soporte para protocolos complejos mediante máquinas de estado.
- Soporte de login mediante una base de datos.
- Aislar las zonas con la misma funcionalidad.
- Aislar las zonas ejecutadas por cierto hilo
- Extensibilidad
- Soporte para nuestras dos alternativas: ICE (parecido a corba) y Sockets.


ChatServer
-secureSocketReceiver:SecureSocketReceiver
-socketReceiver:SocketReceiver
-connectedClients:Vector<ConnectedClient>
-conversations:Vector<Conversation>
-dataBaseConnection:DataBaseConnection
DataBaseConnection
ThreadSafeQueue
-vector:Vector<Commando>
interface
Conversation
1..*
interface
Receiver
interface
Commando
1..*
Channel
Private
NickCommand
PasswordCommand
JoinCommand
interface
UserCommands
interface
AdministratorCo...
interface
ServerCommands
SocketReceiver
SecureSocketReceiver
1..*
interface
ConnectedClient
PartCommand
PrivateCommand
MessageCommand
QuitCommand
ConnectedUser
-protocolo:UserProtocol
ConnectedAdministrator
-protocolo:AdministratorProtocol
interface
Protocol
UserProtocol
-userStateMachine:UserStateMachine
AdministratorProtocol
-administratorStateMachine:AdministratorStateMach
interface
StateMachine
UserStateMachine
AdministratorStateMachine

Para el servidor propusimos entonces el esquema de la página anterior:

En el cual:

- La zona azul representa la zona de aceptación de conexiones desde clientes
- La zona amarilla representa la zona de red con cada cliente, en la cual se envían y se reciben comandos. Estos comandos son traducidos en todos los casos por el protocolo, que es el que asegura el entendimiento entre servidor y cliente. La máquina de estados asegura el buen uso del protocolo.
- La zona verde representa la estructura de datos del chat. Está formado por canales, usuarios, privados...
- La zona roja es la zona de lógica de aplicación, que es la que toma las ordenes y opera sobre las estructuras de datos.

Con respecto a los hilos:

- Cada receptor de conexión tiene su hilo que sólo circula sobre él y mete comandos en la cola.
- Cada ClientConnection tiene su hilo de escucha, que actúa sobre él y sobre el método de traducir de cadena a comando de protocolo. También mete los resultados en la cola.
- Hilo de aplicación: Ejecuta los comandos de la cola y actua sobre el resto de código que no tocan los hilos anteriores.

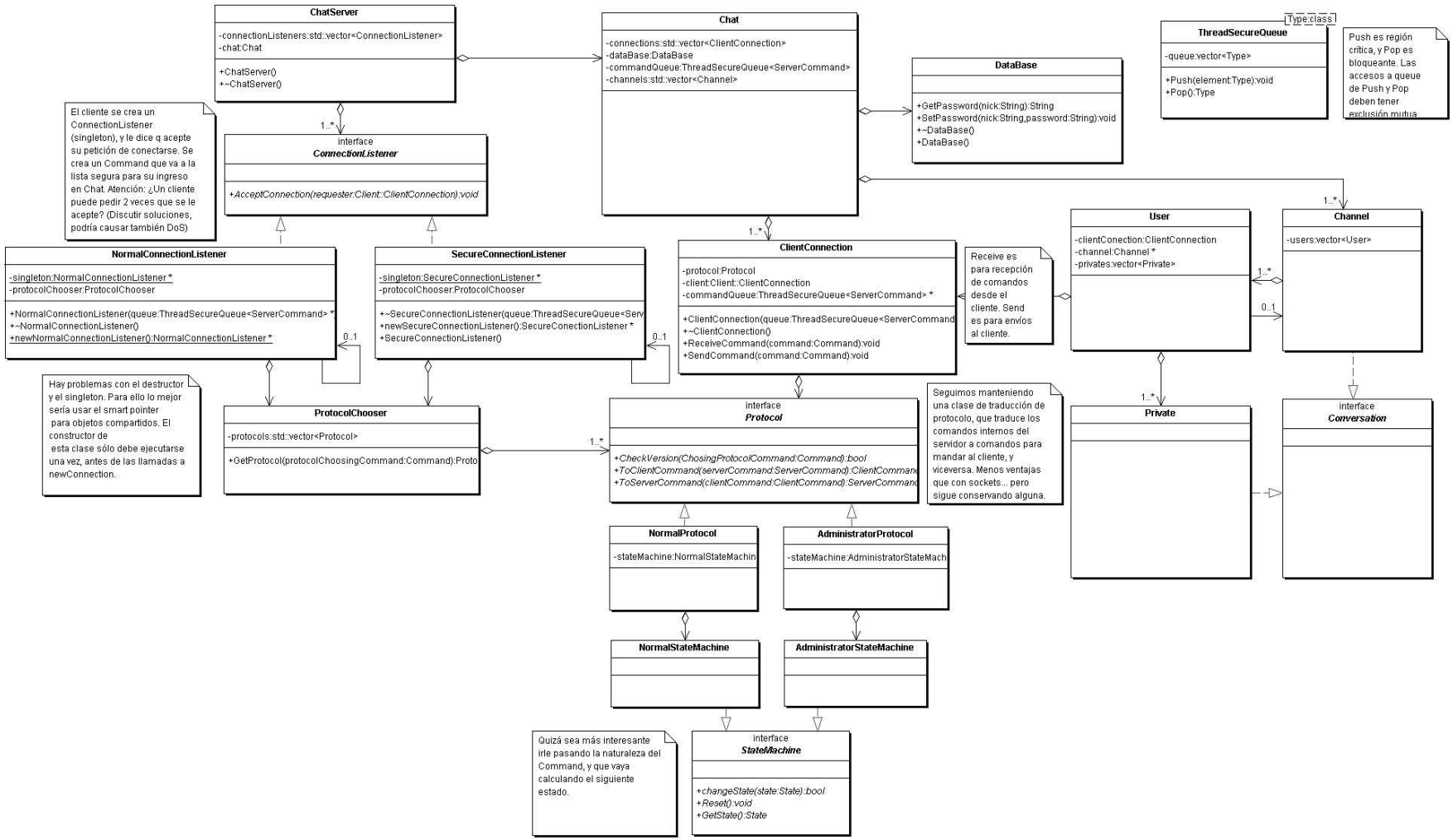
Por lo tanto, los hilos sólo ejecutan el mismo código y actuan sobre datos compartidos en la cola de mensajes.

Una posterior modificación fue la de la página siguiente:

Que desglosa la lógica del chat y se centra más en diseño.

Explicación de cada clase:

- ChatServer: Inicializa el chat y los listeners de conexiones.
- ConnectionListener: Interfaz para las clases que escuchan las conexiones entrantes por un puerto.
- NormalConnectionListener: Escucha las conexiones entrantes mediante un socket y cuando acepta una, mete un comando de NewConnection a la cola de comandos.
- SecureConnectionListener: Igual que NormalConnectionListener, pero usa SSL.
- ProtocolChooser: Determina qué protocolo va a usar un cliente que se ha conectado de todos los protocolos seleccionables.
- Chat: Lógica de chat. Toma los comandos y los ejecuta. Es el poseedor del hilo de aplicación.
- ClientConnection: Manda y recibe mensajes a/desde el cliente por red. Los transforma en comandos mediante protocolo y los mete en la cola de comandos si son entrantes, o los pasa a cadena y envía si son salientes.
- Protocol: Interfaz para protocolo. Transforma de cadenas recibidas a comandos para servidor, y de comandos de servidor a cadenas a enviar.
- NormalProtocol : Implementación de protocolo para clientes normales.
- AdministradorProtocol: Implementación de protocolo para administradores.
- StateMachine: máquina de estados para soportar protocolos complejos. NormalStateMachine y AdministratorStateMachine son las implementaciones correspondientes de esta interfaz para dichos protocolos.
- ThreadSafeQueue: La cola en la que todos los hilos encolan los comandos y que el hilo principal se encargará de sacarlos y ejecutarlos. Es el corazón de nuestra concurrencia.
- DataBase: Proxy de ODBC para poder hacer login.


ChatServer
-connectionListeners:std::vector<ConnectionListener>
-chat:Chat
+ChatServer()
+~ChatServer()
1..*
interface
ConnectionListener
+AcceptConnection(requester:Client::ClientConnection):void
El cliente se crea un ConnectionListener (singleton), y le dice q acepte su petición de conectarse. Se crea un Command que va a la lista segura para su ingreso en Chat. Atención: ¿Un cliente puede pedir 2 veces que se le acepte? (Discutir soluciones, podría causar también DoS)
Chat
-connections:std::vector<ClientConnection>
-dataBase:DataBase
-commandQueue:ThreadSecureQueue<ServerCommand>
-channels:std::vector<Channel>
DataBase
+GetPassword(nick:String):String
+SetPassword(nick:String,password:String):void
+~DataBase()
+DataBase()
Type:class
ThreadSecureQueue
-queue:vector<Type>
+Push(element:Type):void
+Pop():Type
Push es región crítica, y Pop es bloqueante. Las accesos a queue de Push y Pop deben tener exclusión mutua
1..*
User
-clientConection:ClientConnection
-channel:Channel *
-privates:vector<Private>
Channel
-users:vector<User>
1..*
0..1
NormalConnectionListener
-singleton:NormalConnectionListener *
-protocolChooser:ProtocolChooser
+NormalConnectionListener(queue:ThreadSecureQueue<ServerCommand> *
+~NormalConnectionListener()
+newNormalConnectionListener():NormalConnectionListener *
SecureConnectionListener
-singleton:SecureConnectionListener *
-protocolChooser:ProtocolChooser
+~SecureConnectionListener(queue:ThreadSecureQueue<Serv
+newSecureConnectionListener():SecureConectionListener *
+SecureConnectionListener()
ClientConnection
-protocol:Protocol
-client:Client::ClientConnection
-commandQueue:ThreadSecureQueue<ServerCommand> *
+ClientConnection(queue:ThreadSecureQueue<ServerCommand
+~ClientConnection()
+ReceiveCommand(command:Command):void
+SendCommand(command:Command):void
Receive es para recepción de comandos desde el cliente. Send es para envíos al cliente.
Hay problemas con el destructor y el singleton. Para ello lo mejor sería usar el smart pointer para objetos compartidos. El constructor de esta clase sólo debe ejecutarse una vez, antes de las llamadas a newConnection.
ProtocolChooser
-protocols:std::vector<Protocol>
+GetProtocol(protocolChoosingCommand:Command):Proto
interface
Protocol
+CheckVersion(ChosingProtocolCommand:Command):bool
+ToClientCommand(serverCommand:ServerCommand):ClientCommar
+ToServerCommand(clientCommand:ClientCommand):ServerComman
Seguimos manteniendo una clase de traducción de protocolo, que traduce los comandos internos del servidor a comandos para mandar al cliente, y viceversa. Menos ventajas que con sockets... pero sigue conservando alguna.
Private
interface
Conversation
NormalProtocol
-stateMachine:NormalStateMachin
AdministratorProtocol
-stateMachine:AdministratorStateMach
NormalStateMachine
AdministratorStateMachine
Quizá sea más interesante irle pasando la naturaleza del Command, y que vaya calculando el siguiente estado.
interface
StateMachine
+changeState(state:State):bool
+Reset():void
+GetState():State

- Channel: representa a un canal de nuestro chat.
- User: representa a un usuario del chat.
- Private: conversación entre dos usuarios.
- MovementQueue: lista de últimos movimientos
- Movement: movimiento.
- Logger: maneja el fichero o medio de registro del servidor, en el que se escriben los eventos más importantes.

En una versión posterior, se decidió hacer un movimiento más expresivo, de forma que nos pudiéramos deshacer de la cola de movimientos y de su máquina de estados asociada.

2.2.6.3.4 - Arquitectura de cliente de chat:

El módulo de cliente de chat repite en su arquitectura a la zona amarilla del servidor de chat. La estructura es básicamente la misma, pero habida cuenta de que queríamos un cliente de chat monohilo y de que sólo hay una conexión tendremos las siguientes diferencias:

- El protocolo es el inverso (complementario) al del servidor, i.e, los mensajes recibidos y parseados por el servidor son los que se tienen que generar en el cliente y los generados en el servidor los parseados en el cliente. Así, nuestro servidor recibe mensajes "DICO" que son los que manda el cliente, pero el cliente recibe mensajes "DICIT", que son los que manda el servidor.
- La lista de comandos ya no es necesaria puesto que queremos un cliente monohilo. En vez de eso, el hilo pide al módulo todos los comandos llegados desde la última vez que se los pidió. El módulo los devuelve en un Enumeration (Ver sección 3.4)
- Para no bloquear el único hilo, necesitaremos un socket no bloqueante(Ver 2.3.3.3.2).
- Los comandos desde el servidor son semejantes en funcionalidad a los comandos de servidor a cliente en el servidor, y lo mismo con los mensajes para el servidor. Pero sus estructuras son semejantes con sus opuestos. Es decir, los que en el servidor eran comandos puros con execute() ahora son simples eventos portadores de información, y viceversa.

2.2.6.4 – Protocolo de red

2.2.6.4.1 - Introducción

El protocolo es la lengua común que permitirá al servidor hablar con los clientes, y que ambos entiendan el significado de los eventos que se están comunicando. Estos protocolos se pueden ver como una conversación, en la que el cliente indica todas las acciones que está realizando, y el servidor se encarga de comunicarlo al resto o intervenir en caso de que no se pueda realizar aquello que el cliente ha dicho que quería hacer.

El protocolo está basado en la lengua latina:

- Por ser una lengua internacional
- Por ser una lengua con verbos conjugables, a diferencia de las lenguas artificiales como el Esperanto o el Ido.

Ambos puntos se han tomado por las siguientes razones:

- El uso de una lengua internacional facilita el desarrollo de implementaciones alternativas de cliente o servidor por personas de distintas nacionalidades y lenguas.
- El uso de una lengua conjugable nos va a permitir expresar comandos de significado parecido simplemente manteniendo la raíz.

2.2.6.4.2 - Objetivo

¿Qué queremos comunicar?

- Texto de los personajes
- Movimientos de los personajes
- Información de sincronización

2.2.6.4.3 – Las conjugaciones

Los comandos del protocolo vienen expresados en el tiempo presente de indicativo (basado en la forma de infectum). De esta forma, en nuestro protocolo queda totalmente reflejada la naturaleza del protocolo desde el punto de vista de que es una conversación entre servidor y cliente. Leyendo los paquetes que cliente y servidor intercambian por red, una persona podría comprender qué ha pasado y qué se ha dicho sin necesidad de mirar la representación en la interfaz gráfica.

Dependiendo de la persona en la que esté conjugado el verbo significará lo siguiente:

- **Primera persona del singular**: El cliente solicita realizar una acción al servidor
- **Segunda persona del singular**: El servidor fuerza al cliente a que realice una acción
- **Tercera persona del singular**: El servidor notifica al cliente que otro usuario ha realizado una acción

Los comandos en primera y segunda persona tienen el nick implícito. Esto es así porque como se mandan sobre una conexión en la que están sólo dicho cliente y el servidor, ambos saben quién es quién. De esta forma es el servidor el que se encarga de administrar los nicks, y no hay problema si un cliente mal implementado piensa que tiene un nick que no tiene.

Los comandos en tercera persona van siempre precedidos por el nick de la persona que realiza la acción.

2.2.6.4.3 – Comandos del protocolo

El protocolo que vamos a usar será un protocolo de non-echo, esto quiere decir que un usuario que quiera realizar una acción, mandará un comando al servidor y sólo será respondido si la acción no se puede llevar a cabo. De esta forma nos ahorramos esperas innecesarias del eco(echo) desde el servidor (el lag puede llegar a ser de segundos) y con ello disminuimos el tráfico de red.

Este protocolo es case sensitive, es decir, que distingue entre mayúsculas y minúsculas, tanto para comandos (MOVEO, APPELLO,...) como para nicks(Gonzalo es distinto que GoNzAlO).

Las acciones del protocolo son:

Decir una frase

El cliente manda:

DICO <frase>

El resto de usuarios del canal en el que está reciben:

DICIT <nick> <frase>

Siendo

<nick> el nick del hablante (una palabra)
<frase> la frase que quiere decir

Cambiar de nick

El cliente manda:

APPELLO <nick>

Si el nick es correcto y no está repetido, el resto de usuarios recibe:

APPELLAT <viejo_nick> <nuevo_nick>

Si el nick es en cambio incorrecto o está siendo usado por otro usuario, el servidor pide que vuelva al nick que tenía antes, proporcionado por el servidor:

APPELLAS <nick>

APPELLAS también es usado por el servidor al principio de la conexión para comunicar al cliente el nick que se le ha otorgado por defecto.

Nota: La máquina de estados ha sido desechada del servidor para hacerlo flexible, para ello el servidor fuerza valores iniciales, comunicados debidamente al cliente (nick, posición, canal). El cliente después puede mandar instrucciones para cambiarlos... pero libramos al servidor de lógica innecesaria. También se ha considerado esta opción debido a que el cliente manda siempre datos simples que caben en un comando. Los nicks forzados por el servidor son "UserX" donde la X es un número unívoco para nosotros.

Entrar en un canal

El cliente manda:

INTRO <#canal> <X> <Y> <Z> <ANGxz> <TIEMPO>

El resto de usuarios del canal en el que está reciben:

INTRAT <nick> <X> <Y> <Z> <ANGxz> <TIEMPO>

El cliente recibe (único comando con echo, debido a la información adicional):

INTRAS <#canal> <X> <Y> <Z> <ANGxz> <TIEMPO>

Notas:

- INTRAS también es usado por el servidor al principio de la conexión al meter al cliente en el canal por defecto.
- Este comando sí que necesita espera de echo en el cliente, debido a la información extra que hay que mandar.
- En INTRAT no hace falta el canal puesto que un usuario sólo puede estar en un único canal
- Los parámetros <X> <Y> <Z> <ANGxz> <TIEMPO> actúan igual que en MOVEO y su estado de movimiento parado y sin girar.

Mover a un personaje

El cliente manda:

MOVEO <X> <Y> <Z> <ANGxz> <TRANS> <ROT> <TIEMPO>

El resto de usuarios del canal en el que está reciben:

MOVET <nick> <X> <Y> <Z> <ANGxz> <TRANS> <ROT> <TIEMPO>

Si el movimiento no se puede efectuar, el servidor forzará una parada:

MOVES <X> <Y> <Z> <ANGxz> <TRANS> <ROT> <TIEMPO>

Siendo

<X> <Y> <Z> la posición (floats)

<nick> el nick del que se mueve (una palabra)
<TIEMPO> el tiempo de sincronización (long)
<ANGxy> el ángulo al que mira el personaje.
<TRANS>: Estado de translación:
PRORSUS (adelante)
RETRORSUS (atrás)
QUIETUS (quieto)
<ROT>: Estado de rotación:
LAEVUS (izquierdo)
DEXTER (derecho)
QUIETUS (quieto)

Señal de sincronización con el servidor

El servidor manda:

TEMPUS <TIEMPO>

Siendo
<TIEMPO> la hora en segundos en el servidor

Notas:
- Se manda al principio de la conexión de un cliente al servidor
- Se manda de forma periódica por el servidor

Indicar que un usuario ya se encontraba en un canal

El servidor manda:

STABAT <X> <Y> <Z> <ANGxz> <TRANS> <ROT> <TIEMPO>

Que toma los mismos parámetros que MOVET. Esta instrucción sirve para indicar a un cliente que acaba de entrar qué usuarios estaban ya dentro del canal y dónde. Es en cierto sentido distinto de INTRAT puesto que esto se le manda al cliente que acaba de entrar al canal, y por lo tanto los otros no entran, si no que estaban dentro.

Salir de un canal o salir del chat

Para salir del chat, el cliente manda:
EXEO
Para salir de un canal, el cliente entra en otro:
INTRO <#canal> <X> <Y> <Z> <ANGxz> <TIEMPO>
El servidor manda a los que estaban en el canal del que se sale:
EXIT <nick>

Mandar un mensaje privado

El cliente emisor manda:
SUSURRO <nick_receptor> <frase>
El cliente receptor recibe:
SUSURRAT <nick_emisor> <frase>

Donde <nick_emisor> y <nick_receptor> son los dos usuarios envueltos en el mensaje privado, y <frase> lo que se quiere decir.

Códigos de error

Los posibles códigos de error son:

- 000 : Error interno del servidor
- 001 : El servidor no ha entendido la cadena del protocolo recibida del cliente (protocolo incompatible)

2.3 DISEÑO IMPLEMENTADO

En el presente apartado se comentará toda la parte que se ha implementado de todas las cosas que se propusieron. Está divido en las siguientes partes:

- La escena 3D: que hace referencia al pintado de los sprites.
- El escenario: que se refiere al mundo en el que se mueven los sprites.
- Módulo de red: en el que se trata la comunicación entre los distintos clientes.

Ambas partes han sido desarrolladas prácticamente individualmente y finalmente se ha procedido a su integración en un solo proyecto. La integración se hizo del siguiente modo:

- Mediante la clase Scene se fusionaron el conjunto de sprites y el escenario. Aunque ambas partes se pintan por separado no hay ningún problema en pintar primero una y luego otra

- Mediante la clase Chat3D se fusionó la parte anterior con el módulo de red. Antes de proceder a pintar la escena, se reciben por la red los datos del resto de clientes y se envían los datos del cliente actual al resto de clientes.

Con la fusión de las tres partes se consiguió por fin una implementación de un chat funcionando por red perfectamente y en un escenario en condiciones.

2.3.1 LA ESCENA 3D

Por escena 3d nos referimos a toda la parte gráfica del chat. No obstante, en este apartado se trata todo lo referente a esta parte gráfica pero sin tener en cuenta el escenario, que se llevará a cabo en el siguiente apartado. Es decir, el presente apartado trata sólo el tema referente al pintado de muñecos y sus animaciones, así como sus bocadillos.

Para poder añadir al chat los objetos que se editasen mediante el 3dstudio Max, hacía falta hacer un importador de alguno de los formatos de este programa. Se recurrió a uno de los más fáciles, pero no por ello menos potente, el archivo ASE. La ventaja frente a otros formatos es que es en modo texto lo que facilita mucho su investigación.

Dos apartados surgen al realizar la lectura de este tipo de archivos ASE. El primero, ¿cómo fusionar los vértices de la malla y de la textura? Que se explica en el apartado 2.3.1.2. Y, el segundo, ¿cómo componer dos rotaciones de manera eficiente? Que requiere el uso de cuaterniones vistos en el apartado 2.3.1.3.

Analizando el fichero ASE se vio que dicho formato no exportaba de ninguna forma la matriz de textura que se quería usar para las animaciones de éstas. Por lo tanto, hubo que aumentar dicho exportador, creando así los ficheros ASZ, para lo que hizo falta usar el SDK del 3dstudio Max, capítulo 2.3.1.4.

Una vez que se sabe cómo se van a leer los ficheros hace falta una estructura de datos para almacenar los sprites. Esta estructura es la que se conoce como la clase Scene y se trata en el capítulo 2.3.1.5. Conviene insistir en que a esta clase se le añadió también al final un mundo que se trata en el apartado 2.3.2. Es decir, la Scene de la que se habla aquí sólo se refiere l pintado de sprites.

No olvidemos que se está hablando de sprites en un chat. Por lo tanto, hace falta añadirles los bocadillos para que hablen. Lo primero que se hizo fue implementar un editor de fuentes que se iban a utilizar en el proyecto. Este editor está incluido en el cd y se incluye un

breve manual para su utilización en el apéndice 3. Estas fuentes se utilizarón para escribir texto en los bocadillos de los muñecos del chat. También se utilizó una extensión de openGL conocida como pbuffers. Todo esto está desarrollado en el apartado 2.3.1.6, donde se explica toda la implementación de los bocadillos.

Y, finalmente, se implementó el SpriteChat, la clase que se encarga de manejar un muñeco por el mundo3d. Esta clase maneja tanto su movimiento, como sus animaciones o sus bocadillos.

2.3.1.1 Ficheros ASE (ASCII Export Files)

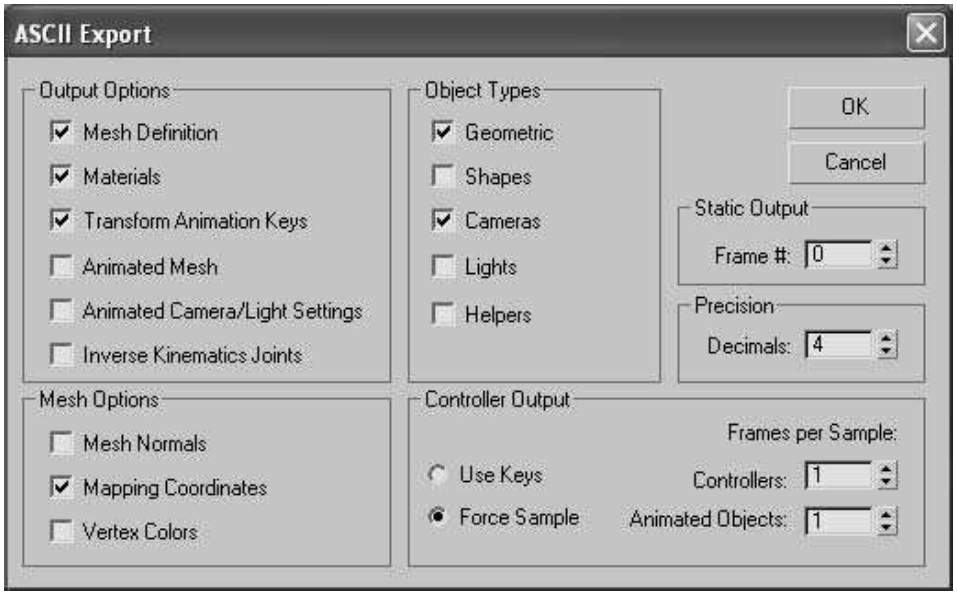
¿Qué es un fichero ASE?

El programa de diseño 3d studio Max contiene una serie de exportadores de varios formatos para que las mallas y animaciones que se definen en él puedan ser utilizados por otras aplicaciones. Uno de estos exportadores es el conocido como ASCII Export que genera ficheros en modo texto con la extensión *.ASE.

Al igual que en el resto de exportadores, en un fichero ASE se exporta una gran cantidad de información sobre la escena que se está editando, pero no toda (solo los ficheros *.max guardan toda la información de la escena). Lo que se va a comentar aquí es la parte que se ha utilizado para el proyecto: información sobre la escena, sobre los materiales, animaciones...

Configuración del exportador

El fichero ASE es más extenso de lo que parece y no se ha pretendido en este proyecto crear un lector de todo tipo de ASES. Todos los objetos utilizados usan la siguiente configuración para el exportador:



Como se ve, se exportan la definición de la malla, de los materiales y de la animación. Admite también los dos tipos de cámara del Max, tanto el free como el target y exporta las coordenadas de textura. Por último, la animación no la lee por KeyFrames (el 3dstudio utiliza muchas interpolaciones y el implementarlas todas ellas podría suponer

un proyecto en si mismo) sino que se leen los samples, esto es, para cada frame, la posición y rotación del objeto en dicho instante.

Partes en las que se divide un fichero ASE

El fichero ASE se divide en tres partes:

- **Parámetros de la escena**: informa de los parámetros de la escena en general: nombre, tiempo que dura, etc.
- **Lista de materiales**: ofrece una lista de los materiales utlizados en la escena con cada uno de sus parámetros. Dentro de un material está también la definición de la textura.
- **Información sobre cada uno de los objetos**. Esta parte a su vez se puede dividir en:
    - **Información sobre el nodo**: indica quién es su padre, y la matriz inicial del objeto
    - **Información de la malla**: indica cómo está formada la malla del objeto, esto es, vértices, caras, normales, vértices y caras de textura...
    - **Animación**: la animación del objeto.

A continuación se irán desmembrando, una por una cada, una de esas partes. Se indicará principalmente la utilidad de los apartados del ASE que se han usado en el importador del proyecto

2.3.1.1.1.- Parámetros de la escena

Un ejemplo de definición de la escena es el siguiente:

```
*SCENE {
    *SCENE_FILENAME "CaratecaAndando.max"
    *SCENE_FIRSTFRAME 0
    *SCENE_LASTFRAME 30
    *SCENE_FRAMESPEED 30
    *SCENE_TICKSPERFRAME 160
    *SCENE_BACKGROUND_STATIC 0.8078     0.8078      0.8078
    *SCENE_AMBIENT_STATIC 0.0431  0.0431      0.0431
}
```

Aparece siempre al principio de cualquier fichero ASE exportado y viene precedido por la palabra "*SCENE". Los parámetros interesantes aquí son:

- "*SCENE_FIRSTFRAME" y "SCENE_LASTFRAME": con la resta de estos dos valores se puede calcular el número total de frames de animación que presenta la escena actual
- "SCENE_TICKSPERFRAME" este valor dividido por 5 es el número de milisegundos que transcurren entre un frame y el siguiente

2.3.1.1.2.- Lista de materiales

En este apartado aparece una lista de materiales numerados. Los materiales pueden estar divididos a su vez en submateriales, siempre que un objeto presente varios de ellos en su malla. En este proyecto no se ha tratado en tema de los submateriales, puesto que no hacían falta.

```
*MATERIAL_LIST {
    *MATERIAL_COUNT 10
```

```
    *MATERIAL 1 { ... }
    .
    .
    .
    *MATERIAL 10 {...}
}
```

La lista de materiales empieza con la palabra "*MATERIAL_LIST". A continuación, entre llaves ({ }) aparece la palabra "*MATERIAL_COUNT" seguida de un entero. Dicho entero indica el número total de materiales utilizados para la escena.

Seguidamente viene la definición de cada material. Un ejemplo de material:

```
*MATERIAL 0 {
          *MATERIAL_NAME "Material #3"
          *MATERIAL_CLASS "Standard"
          *MATERIAL_AMBIENT 0.6941     0.6902     0.7020
          *MATERIAL_DIFFUSE 0.9373     0.9294     0.9608
          *MATERIAL_SPECULAR 1.0000    1.0000     1.0000
          *MATERIAL_SHINE 0.2500
          .
          .
          .
          *MAP_DIFFUSE {
                *MAP_NAME "Map #3"
                *MAP_CLASS "Bitmap"
                *MAP_SUBNO 1
                *MAP_AMOUNT 1.0000
                *BITMAP "C:\Gonzalo\code\lectorsillo\carateca.bmp"
                .
                .
                .
          }
    }
```

Las partes que aparecen punteadas se omiten por carecer de utilidad para el proyecto. Primeramente aparece el nombre del material.
Seguidamente, se pueden leer las tres componentes equivalentes a las de opengl de los materiales: ambiente, difuso y especular. En el caso de que no haya textura el color dado por esas tres componentes será el color del objeto. En caso contrario, se multiplicará el color de la textura por el del material
El apartado "*MAP_DIFFUSE" puede aparecer o no. Si se define en el 3dstudio MAX una textura para la componente difusa aparecerá aquí. La textura para la componente difusa es la que se suele utilizar como textura propiamente dicha. Así pues dentro de este apartado interesa la linea que comienza por la palabra: "*BITMAP", el campo que aparece a continuación es una ruta donde se encontraría situada la textura.
Nota: se recomienda ignorar la ruta de la textura y leer sólo el nombre (en este caso "carateca.bmp") puesto que de ese modo se puede obligar a que la textura se encuentre siempre en el mismo directorio en el que se encuentra el fichero ASE.

2.3.1.1.3.- Información sobre cada uno de los objetos

El siguiente campo es el que se podría denominar corazón del ASE, pues en él se encuentra la información más importante. El lector del proyecto admite además de objetos3d la lectura de cámaras. La diferencia entre ambas es poca: una cámara no presenta definición de malla, ni material y además presenta un campo adicional denominado "*CAMERA_SETTINGS"

en el que aparecen los parámetro de near, far y focus utilizados por opengl para definir la matriz de projección (GL_PROJECTION). Además, en el caso de una cámara de tipo target, aparece también la información del target (el target es un punto al que la cámara siempre mira) que se debe ignorar puesto que al realizar la lectura de animaciones por samples, el 3dstudio se encarga de exportarnos la cámara mirando siempre a dicho target.

La definición de un objeto3d está identificada por la palabra "*GEOMOBJECT" y a continuación se abren llaves para definir los parámetros de turno:

```
*GEOMOBJECT {
    *NODE_NAME "Paquete"
    *NODE_PARENT "PantalonDerecha"
    *NODE_TM {
          // Información del nodo aquí
    }
    *MESH {
          *TIMEVALUE 0
          *MESH_NUMVERTEX 17
          *MESH_NUMFACES 24
          *MESH_VERTEX_LIST {
                // Información de los vértices
          }
          *MESH_FACE_LIST {
                // Información de las caras
          }
          *MESH_NUMTVERTEX 51
          *MESH_TVERTLIST {
                // Información de vértices de textura
          }
          *MESH_NUMTVFACES 24
          *MESH_TFACELIST {
                // Información de caras de textura
          }
    }
    .
    .
    .
    *TM_ANIMATION {
          *NODE_NAME "Paquete"
          *CONTROL_POS_TRACK {
                // Información sobre los parámetros de la animación
          }
    }
    *MATERIAL_REF 0
}
```

Esta es la forma que presenta una definición de un objeto. Nótese que el material utilizado por el objeto es un parámetro que aparece al final, y es un dato importante que se debe leer. La razón por la que no aparece dentro del campo "*NODE_TM" es porque como ya se ha dicho, una cámara no presenta material.

También aparecen fuera, el nombre del objeto y el nombre de su padre (el nombre del padre puede no aparecer, indicando que el nodo actual es un nodo raíz). Es importante leer este segundo parámetro para respetar la jerarquía entre objetos (si no se hace, es mejor ir olvidándose de la animación). El nombre del nodo, si no se lee aquí, aparece también dentro de cada campo("*NODE_TM", "*MESH" y "*ANIMATION"), al inicio.

Para el caso de las cámaras se tiene:

```
*CAMERAOBJECT {
```

```
*NODE_NAME "Camera01"
*CAMERA_TYPE Target
*NODE_TM {
      *NODE_NAME "Camera01"
      .
      .
      .
}
*NODE_TM {
      *NODE_NAME "Camera01.Target"
      .
      .
      .
}
*CAMERA_SETTINGS {
      *TIMEVALUE 0
      *CAMERA_NEAR 2821.0000
      *CAMERA_FAR 6379.0000
      *CAMERA_FOV 0.2676
      *CAMERA_TDIST 3993.1372
}
*TM_ANIMATION {
      // igual que en GEOMOBJECT
}
```

No hay muchas diferencias en el caso de que haya una cámara: en vez de "*GEOMOBJECT" la palabra que lo identifica es "*CAMERAOBJECT". Puede aparecer dos veces la definición de "*NODE_TM" (una de ellas es la del target que se debe ignorar como ya se ha indicado) y presenta un campo nuevo, llamado "*CAMERA_SETTINGS" con los parámetros solicitados por la función gluPerspective (el campo FOV hay que convertirlo a grados, pues aparece en radianes)

Como se indicó anteriormente la definición de un objeto se divide en:

- Información del nodo
- Información de la malla (en las cámaras no aparece)=, y
- Animación
    - o De posición
    - o De rotación

2.3.1.1.3.1.- Información sobre el nodo

Un ejemplo:

```
*NODE_TM {
      *NODE_NAME "MusloDerecho"
      .
      .
      .
      *TM_ROW0 -0.9998  -0.0130     0.0117
      *TM_ROW1 -0.0000  -0.6691     -0.7431
      *TM_ROW2 -0.0175  0.7430      -0.6690
      *TM_ROW3 -8.1719  5.4965      78.1651
      .
      .
      .
}
```

Se encuentra uno en esta sección con una de las cosas más curiosas de este formato. Los campos de "*TM_ROW0", "*TM_ROW1", "*TM_ROW2" y "*TM_ROW3", que definen una matriz de orientación por filas de un objeto 3d. Sin embargo ocurren tres cosas:

- La matriz puede presentar un escalado negativo que hay que corregir.
- La matriz está dada respecto al mundo, no respecto al padre, por lo tanto hay que realizar la conversión .
- Los valores de las componentes 'y' y 'z' están intercambiados entre sí, y además la componente y tiene el signo cambiado

Afortunadamente, las tres cosas presentan una solución.

• Para el **escalado negativo** (detectado cuando el determinante de la matriz es menor de cero) se debe postmultiplicar por una matriz de escalado que invierta dicho escalado, esto es, la identidad con las diagonales con signo negativo:

```
        | -1.0f,  0.0f,   0.0f, 0.0f |
        |  0.0f, -1.0f,   0.0f, 0.0f |
        |  0.0f,  0.0f,  -1.0f, 0.0f |
|  0.0f,  0.0f,   0.0f, 1.0f |
```

Es bastante aconsejable antes de seguir transformando la matriz el almacenar la matriz tal como está en algún sitio durante la lectura del ASE, pues es así como se usará tanto para convertir sus propios vértices y animaciones, como para la conversión de las matrices de los hijos.

• Para **convertir la matriz de mundo a coordenadas locales** (esto es, respecto al padre) se debe postmultiplicar la matriz por la matriz del padre que aparecía en el ASE invertida, o sea, la matriz sin convertir, tan sólo con el escalado negativo citado anteriormente. El motivo de todo esto son unos simples cálculos matemáticos:

> Sea *M* la matriz del padre, *mm* la del hijo en coordenadas de mundo y *ml* la del hijo en locales. La fórmula que da la matriz de mundo a partir de las locales es la siguiente:
>
> $$ml * M = mm$$
>
> El proceso inverso se realiza multiplicando por la inversa a ambos lados, y queda:
>
> $$ml = mm * M^{-1}$$

• Para el **cambio de los valores 'y' y 'z'** se postmultiplicará la matriz que corresponda a los nodos raíz por la siguiente:

```
        | 1.0f, 0.0f,  0.0f, 0.0f |
        | 0.0f, 0.0f, -1.0f, 0.0f |
        | 0.0f, 1.0f,  0.0f, 0.0f |
| 0.0f, 0.0f,  0.0f, 1.0f |
```

Haciendo algo tan sencillo se obtiene un sistema de coordenadas equivalente al habitual de trabajo con opengl. Como se ve, también se cambia el signo de la componente 'y' (que aparece como 'z' en la matriz dada).

Por último decir que los tres cambios sobre la matriz deben realizarse en el orden citado, y que el segundo y el tercero son excluyentes entre sí. La conversión a locales se debe realizar, lógicamente, cuando el nodo no sea un nodo raíz (esto es, tenga un

padre) mientras que el intercambio de la 'y' y la 'z' se debe realizar sólo en los nodos raíz.

2.3.1.1.3.2.- Información sobre la malla

En este apartado aparece la definición de malla de un objeto. La malla de un objeto esta dividida en 4 partes bien diferenciadas:

- Vértices
- Caras
- Vértices de textura
- Caras de textura

• Vértices

La definición de los vértices viene dentro del campo "*MESH_VERTEX_LIST". Un ejemplo:

```
*MESH_VERTEX_LIST {
                *MESH_VERTEX    0 -7.1746     23.8509     59.9578
                *MESH_VERTEX    1 -10.5053    23.8134     59.9916
                *MESH_VERTEX    2 -7.1746     25.5226     61.8145
                .
                .
                .
        }
```

Se puede apreciar que cada vértice consta de un índice y de tres valores que corresponden a las componentes x, y, z. Al igual que ocurría con la matriz del objeto, dichos vértices están dados en coordenadas de mundo (quizá para que cuando uno sólo quiera leer la escena sin más, pues simplemente lea estos valores) y hay que convertirlos a locales. Además, del mismo modo que ocurría con la matriz, los valores de 'y' y 'z' están intercambiados y la 'y' de signo contrario.

El cambio a locales de cada vértice se realiza postmultiplicándolo (o sea, el vértice colocado como una matriz fila) por la matriz inicial del objeto (la que aparecía en el ASE y no la transformada a locales) invertida.

Este hecho, de nuevo, proviene de unos simples cálculos matemáticos. Sea vm el vértice en coordenadas de mundo, sea vl el vértice en coordenadas locales y sea M la matriz del objeto (en mundo). La forma de obtener vm es:

$$vm = vl * M$$

Mutiplicando por la inversa de M a ambos lados se tiene

$$vm * M^{-1} = vl$$

Podría uno plantearse el hecho de por qué se está utilizando M en coordenadas de mundo y no en coordenadas locales. Es evidente que las coordenadas de mundo no se obtienen multiplicando el vértice por la matriz en locales, sino que hay que hacerlo en mundo; ésta es la razón.

• Caras

La lectura de caras es mucho más sencilla que la de vértices, pues están dadas como índices que hacen referencia al array de caras (cada cara consta de tres índices, que hacen referencia al índice del vértice) . Un ejemplo de cara:

```
*MESH_FACE_LIST {
                *MESH_FACE 0: A: 3 B: 2 C: 0 AB: 0 BC: 1 CA: 1
*MESH_SMOOTHING 1     *MESH_MTLID 1
                *MESH_FACE    1:    A:    0 B:    1 C:    3 AB:    0
BC:    1 CA:    1      *MESH_SMOOTHING 1      *MESH_MTLID 1
                .
                .
                .
          }
```

Como se puede apreciar, cada cara comienza por un índice (el índice de la cara) y a continuación vienen 3 campos que son los más interesantes: A, B y C, que representan una cara dada en sentido contrario a las agujas del reloj (es la opción por defecto de openGL así que no hay que realizar ningún cambio).

El resto de valores de cada cara se puede ignorar, puesto que hacen referencia a submateriales y otros aspectos que caen fuera de los ámbitos del proyecto.

Notas importantes:

- No se garantiza en todas las caras las existencia del campo "*MESH_SMOOTHING" por lo que no se recomienda el pensar que una cara ocupa siempre el mismo número de palabras
- Una vez leídas las caras sería un buen momento para calcular las normales de la malla, en el caso de que no se vayan a leer del ASE, cosa nada aconsejable si se utiliza, como es habitual, una normal por vértice y no por cara.

• Vértices de textura

Hay que advertir que este tipo de vértices no aparece siempre. Un objeto que tenga vértices de textura es porque, lógicamente, tiene una material con una textura. Un ejemplo típico de una lista de este tipo de vértices podría ser el siguiente:

```
*MESH_TVERTLIST {
                *MESH_TVERT 0     0.7822      0.5599      1.2793
                *MESH_TVERT 1     0.7961      0.5599      1.2793
                *MESH_TVERT 2     0.7702      0.5599      1.0403
                .
                .
                .
          }
```

Nótese que dichos vértices siempre presentan coordenadas con valores comprendidos entre 0 y 1, al igual que en openGL. Curiosamente, aquí no aparecen intercambiados los valores de 'y' y de 'z' sino que se deben leer tal cual.

La última componente, la z o w, puede ser ignorada si sólo se usan texturas 2D

• Caras de textura

Al igual que ocurre con los vértices de textura, las caras sólo aparecen si el objeto tiene asignado un material con textura. Un ejemplo:

```
*MESH_TFACELIST {
```

```
                *MESH_TFACE 0     8     0     2
                *MESH_TFACE 1     0     9     1
                *MESH_TFACE 2     10    4     5
                .
                .
                .
        }
```

Cada cara, consta de cuatro índices. El primero es el índice de la cara y los tres siguientes se corresponden con los índices de vértices de textura.

Nota importante: el número de vértices no tiene por qué coincidir con el número de vértices de textura, pero el número de caras debe ser exactamente el mismo, y, de hecho, la cara número x se corresponde con la cara número x de textura. Para poder trabajar en openGL con arrays de vértices se deben unificar los vértices de malla con los de textura, de modo que los arrays de caras acaben siendo exactamente los mismos, duplicando para ello vértices de uno u otro array cuando sea necesario. Se comenta cómo realizar todo esto en la sección 2.3.1.2.

2.3.1.1.3.3.- Animación

La parte de la animación suele ser la más compleja de todas, y la que da más quebraderos de cabeza. Sin embargo, si se ha respetado la jerarquía y se han convertido las matrices de mundo a local, como se podrá ver, no es tan complicado. La animación presenta el siguiente esquema:

```
*TM_ANIMATION {
          *NODE_NAME "PiernaDerecha"
          *CONTROL_POS_TRACK {
                //Samples de animación de posiciones
          }
          *CONTROL_ROT_TRACK {
                //Samples de animación de rotaciones
          }

}
```

El nombre puede ser ignorado perfectamente, puesto que ya se habrá leído en las secciones anteriores. Es útil sin embargo que aparezca aquí, para un ASE limitado en el que sólo se quieran exportar animaciones para añadirlas a una malla previamente leída por separado.

Como es de prever, el campo "*CONTROL_POS_TRACK" contiene los samples referentes a la posición, y "*CONTROL_ROT_TRACK" los referentes a la rotación.

• Animación de posición

La animación de posiciones viene dada como sigue:

```
*CONTROL_POS_TRACK {
          *CONTROL_POS_SAMPLE 0   -1.5218     -0.1379     82.2868
          *CONTROL_POS_SAMPLE 160 -1.5218     -0.1379     82.3208
          *CONTROL_POS_SAMPLE 320 -1.5218     -0.1379     81.3139
          *CONTROL_POS_SAMPLE 480 -1.5218     -0.1379     78.7847
          .
          .
          .
          }
```

Cada sample (que se corresponde con un frame de animación) viene definido por 4 valores. El primer valor es un entero que da los ticks que transcurren entre un frame y otro, o en este caso, el tick al que corresponde el frame. Antes se dijo ya, pero un tick son 5 milisegundos. Conviene realizar dicha conversión puesto que los ticks son una medida que utiliza el 3dstudio simplemente.

Los tres valores siguientes dan la posición del objeto en un frame determinado. De nuevo, dicha posición no está dada en coordenadas locales y hay que convertirla. Para ello, se postmultipica cada posición (como si fuera un vértice) por la inversa de una de las dos matrices siguientes:

- **Si es un nodo raíz**: la matriz tal cual se leyó del ASE pero con el escalado negativo eliminado (se comentó anteriormente que se debía almacenar esa matriz pues sería útil, ya se ve aquí por qué).
- **Si no es nodo raíz**: la matriz en ese caso es la matriz en coordenadas locales.

La verdad, es que uno puede pensar que lo anterior es erróneo y que se ha escrito incorrectamente, pero no, es así. Esta aclaración viene por el hecho de que los vértices se convertían con la matriz en coordenadas de mundo y aquí se hace con la matriz local. Los implementadores del exportador tendrían alguna razón para hacerlo así, y solo ellos deben saber el por qué de este pequeño lío.

Notas importantes:

- No se debe pensar que cada sample se separa siempre del anterior por una unidad de tiempo fija (160 ticks en este caso). Si dos samples coinciden, el ASE no lo escribirá y se debe suponer que es el mismo que el anterior. Es muy importante leer la unidad de tiempo para que no halla problemas posteriormente.
- Si aparece un frame con unidad de tiempo 0, se debe ignorar. El frame 0, que siempre debe existir debe ser el que dé la posición (0,0,0)

• Animación de rotación

Muy similar a la anterior en el apartado de los ticks. Una animación de rotación presenta el siguiente aspecto:

```
*CONTROL_ROT_TRACK {
    *CONTROL_ROT_SAMPLE 0   -0.9998  -0.0196 -0.0087  0.8379
    *CONTROL_ROT_SAMPLE 160  1.0000   0.0000   0.0000  0.0298
    *CONTROL_ROT_SAMPLE 320  1.0000   0.0000  -0.0000  0.0838
    *CONTROL_ROT_SAMPLE 480  1.0000   0.0000  -0.0000  0.1296
          .
          .
          .
          }
```

El primer entero de cada sample se corresponde con el tick correspondiente. A continuación le siguen 4 números decimales: el primero se corresponde con un ángulo y los tres siguientes definen un vector. Queda definido de este modo una rotación del tipo ángulo eje como las que utiliza openGL.

Pero en esta ocasión hay tres problemas:

- Los vectores de rotación están dados de nuevo en coordenadas de mundo.
- Una rotación viene dada respecto a la anterior (¿¿?¿, aquí si que parece que lo hicieron para fastidiar, porque con la animación de posiciones no pasa esto).
- El ángulo viene dado en radianes y en sentido inverso.

De nuevo, con un poco de paciencia, los tres problemas pueden ser solucionados del siguiente modo:

• Para convertir los vectores a coordenada locales, se debe postmultiplicar (como si fuera un vector) por la inversa de una de las dos matrices siguientes:

- **Si es un nodo raíz**: la matriz tal cual se leyó del ASE pero con el escalado negativo eliminado.
- **Si no es nodo raíz**: la matriz en ese caso es la matriz en coordenadas locales.

Esta transformación es similar a la que se realizó para la animación de posiciones.

• Para que una rotación venga dada respecto al frame 0 y no respecto al anterior, lo que se debe hacer es justamente, multiplicar cada rotación por su anterior (por su anterior se entiende la anterior respecto al frame 0. Esto es fácil de hacer si se empieza haciendo por el frame 0 y cada una se va multiplicando por la anterior)

La forma de más sencilla de multiplicar (o componer) dos rotaciones entre sí consiste en realizar un paso a cuaternión de ambas y multiplicarlos entre sí. Finalmente el cuaternión resultado se debe pasar a ángulo eje. Para más información sobre los cuaterniones y estas operaciones, consultar el apartado 2.3.1.3.

• La conversión de radianes a grados es un paso muy sencillo. Sea angle el ángulo dado en radianes:

```
angle= -1.0f * ((angle * 360.0f) / (2.0f *3.141592653f));
```

Se multiplica por –1, puesto que como ya se dijo, presenta el sentido opuesto

Nota: para realizar correctamente la conversión de las rotaciones es necesario hacerlas en este orden

- Cada rotación se convierte a locales y luego se multiplica por la anterior.
- Una vez se han realizado dichas operaciones con todos los frames, convertir de radianes a grados.

El motivo de hacer las cosas así es porque los cuaterniones trabajan con los ángulos dados en radianes, por lo que una conversión al acabar con un frame nos llevaría a volver a convertir a radianes para poder multiplicar por la siguiente, lo cual no es muy eficiente.

2.3.1.2 Fusión de los vértices y caras de malla con los vértices y caras de textura

El problema

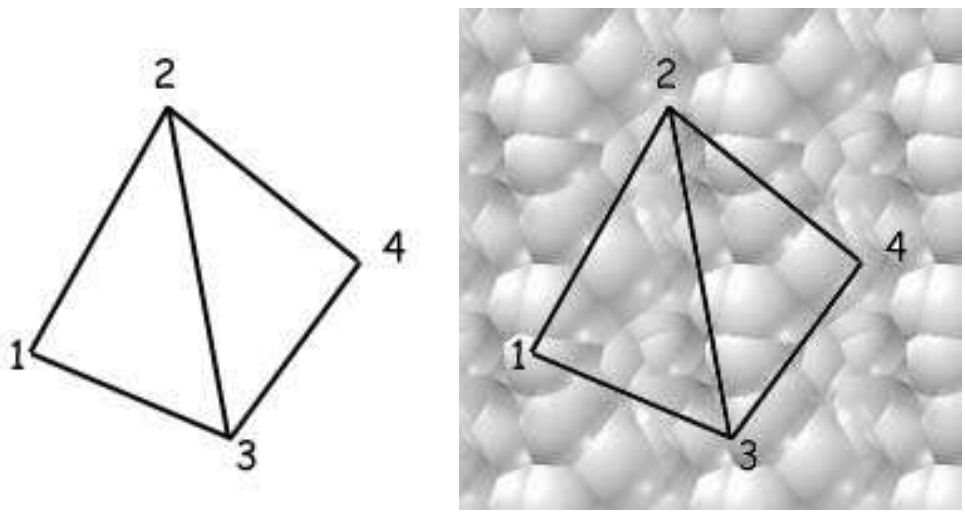
A la hora de utilizar los vertex buffers de opengl surge un problema que se debe solucionar. Cuando se define una malla, ésta presenta una serie de vértices agrupados en caras. Si esa malla tiene una textura, presentará también una lista de caras y vértices de textura.

Los arrays de caras y de caras de textura se corresponden, de modo que la cara 4 de malla tiene sus coordenadas de textura localizadas en la cara 4 de textura. Sin embargo, supongamos que dichas caras son como siguen:

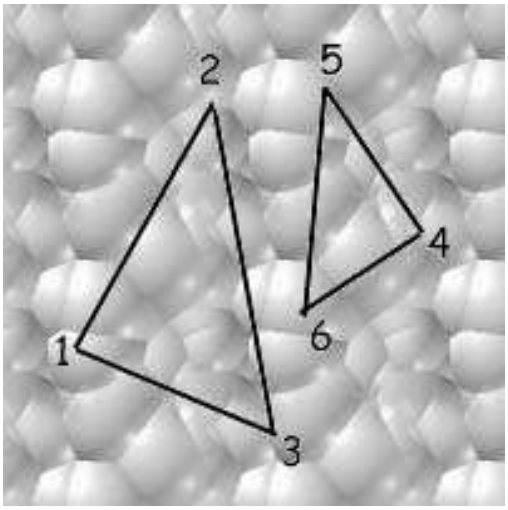
Cara de malla:	3	4	5
Cara de textura:	1	7	9

Efectivamente, los índices no coinciden. La cara de malla hace referencia a los índices 3, 4, y 5 del array de vértices de malla, mientras que la cara de textura hace referencia a los índices 1, 7 y 9 del array de vértices de textura. Sin embargo, opengl obliga, a la hora de utilizar vertex buffers, a que estos índices coincidan.

¿Por qué dos caras no coinciden? Es decir, cuál es la causa por la que puede pasar lo anterior. Aunque el número de caras sea el mismo, el número de vértices de ambos arrays no tiene por qué serlo. Se puede observar mejor en el siguiente dibujo:



Bien, ésta es una posible manera de asignar los vértices de textura a la malla definida en la imagen de la izquierda, sin embargo no es la única. Se podría hacer lo siguiente:



En este caso, sigue habiendo dos caras, pero como se ve, se han duplicado los vértices 2 y 3, dando lugar a los vértices 5 y 6. Esto no es nada raro, de hecho es bastante habitual cuando se manejan con mallas un poco más complejas, pues se trata de aprovechar al máximo una textura, dando lugar a vértices duplicados.

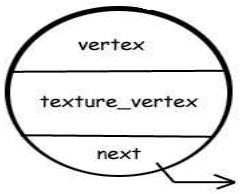
Solución

Afortunadamente, como todo problema, éste tiene su solución. La solución consistirá en ir duplicando los vértices de la malla cada vez que se detecte que se ha duplicado uno de textura. Igualmente, se debe duplicar un vértice de textura, cuando se haya hecho lo propio con uno de malla.

Estructura de datos necesaria

Pero en vez de hacer las cosas de una manera salvaje con un algoritmo que podría ser de coste cuadrático (para cada vértice de textura, comprobar con todos y cada uno de los de malla) se va a tratar de dar una solución más eficiente. Para ello hace falta una estructura de datos como la siguiente:

```
struct FullVertex
{
    int vertex;
    int texture_vertex;
    int next;
}
```



Con esta estructura se deberá generar un vector del tamaño del array de vértices y situando en cada posicicón del vector el vértice de malla correspondiente, y el resto se inicia a –1 (por ejemplo, para indicar que está vacío, o apuntando a null, puesto que no son punteros, sino índices).

- **vertex** es el índice de un vértice de la malla
- **texture_vertex** es el índice de vértice de textura que se corresponde con ese vértice de malla. Ambos coindicen en una de las caras
- **next** indica, en el caso que el vértice de malla esté duplicado, la posición de dicha duplicación en el vector. La razón de que aparezca este campo es precisamente para hacer más eficientes las búsquedas de un vértice duplicado por el vector

Algoritmo

Finalmente, el algoritmo para la duplicación de vértices de malla es el siguiente.

Para cada una de las caras del array de caras hacer:
Para cada uno de los tres vértices de dicha cara hacer:
Se comprueba si el vértice de malla tiene ya asociado uno de textura (texture_vertex distinto de –1). Dos casos:

∇ Si no tiene asociado ningún vértice de textura:
Se le asocia el vértice de textura actual

∇ Si tiene asociado un vértice de textura:
- Si el vértice de textura es igual al actual, no se hace nada
- En caso contrario hay que duplicar el vértice de malla. Para ello:
  - Se crea un nuevo nodo, full_vertex
  - Se pone el vértice de malla a duplicar en el nodo
  - Se pone el vértice de textura actual
  - Se añade el nuevo nodo al vector
  - Se cambia el campo next de la posición donde estaba el vértice duplicado apuntando al nuevo nodo (final del vector)

Una posible implementación en C++ quedaría como sigue:

```
//inicializo el vector de FullVertex con los vértices de la malla
```

```
    boost::shared_ptr< FullVertex > vert;
    for(unsigned int i = 0; i < vertices.size(); i++)
    {
        vert.reset(new FullVertex(i));
        full_vertex.push_back(vert);
    }

    // Rellenamos el vector FullVertex, duplicando vértices de malla cuando sea
    // necesario
    // Para ello hacemos un recorido por los arrays de caras (ambos miden lo mismo,
    // lógicamente)
    for(unsigned int i = 0; i < faces.size() ; i++)
    {
        //Recorro cada vértice de la cara
        for(int j = 0; j < 3; j++)
        {
            int actual_vertex = faces[i].Get(j);
            int actual_tvertex = texture_faces[i].Get(j);
            //Caso 1: no hay vértice de textura asociado a ese vértice de la malla
            if( full_vertex[actual_vertex]->texture_vertex == -1 )
            {
                //le asignamos el vértice actual (el j)
                full_vertex[actual_vertex]->texture_vertex = actual_tvertex;
            }
            //Caso 2: ya hay un vértice de textura asociado con ese vértice.
            else
            {
                // Si es el propio vértice de textura no hago nada, está bien así
                // De este modo evitamos un duplicamiento inútil
                if(full_vertex[actual_vertex]->texture_vertex != actual_tvertex)
                {
                    //Busco donde añadir el vértice duplicado (actualizar next)
                    int m = actual_vertex;
                    while(full_vertex[m]->next != -1)
                    {
                        m = full_vertex[m]->next;
                    }

                    boost::shared_ptr< FullVertex >new_v (new FullVertex
(actual_vertex));
                    new_v->texture_vertex = actual_tvertex;
                    full_vertex.push_back(new_v);
                    full_vertex[m]->next = (int)full_vertex.size() - 1;

                    //Hacemos los cambios en el array de caras
                    faces[i].Set(j, (int)full_vertex.size() - 1);
                }
            }
        }
    }
```

2.3.1.3 Cuaterniones y su utilización en el proyecto

Antes de comenzar el presente apartado quizá convenga aclarar que un cuaternión es mucho más que lo que se va a decir aquí, y que dichos elementos son muy útiles y muy utilizados para cálculos de transformaciones similares a los que se hacen con una matriz. La principal ventaja que poseen es que además de ocupar menos espacio que una matriz, son muy fáciles de interpolar. No es ese el uso que se le ha dado.

Básicamente podríamos definir un cuaternión como una forma de representar una matriz de transformación que carece de escalado. La diferencia principal es que una matriz de transformación es de 3x3 (o sea 9 componentes) mientras que los cuaterniones solo constan de 4 elementos. También es mucho menos costoso operar con cuaterniones que con matrices.

Utilidad

El uso que se la ha dado es simplemente para poder multiplicar (sería mejor decir componer) dos rotaciones ángulos-eje entre sí. Es decir, dada una matriz, se puede multiplicar por varios ángulos-eje (en opengl con glRotatef), ¿cuál sería la rotación equivalente a todas ellas?

Básicamente se han utilizado tres operaciones que trabajan con cuaterniones:

- Cambio de ángulo-eje a cuaternión
- Multiplicación de cuaterniones
- Cambio de cuaternión a ángulo-eje

Para más información sobre los cuaterniones se puede consultar [34].

2.3.1.3.1.- Cambio de ángulo eje a cuaternión

Sea el ángulo eje formado por el ángulo *angle* y el vector *vector:*

Sea half = 0.5f * angle;
Sea cs = cosf(half);
Sea sn = sinf(half);

Entonces, el cuaternión (angle, vector(x, y, z)) viene dado por:

vector.x = (sn * vector.x);
vector.y = (sn * vector.y);
vector.z =(sn * vector.z);
angle = cs;

Es muy importante que el vector se encuentre normalizado, si no, la conversión estará mal hecha. Aunque se supone que así es, pues en un ángulo eje el vector debe estarlo.

2.3.1.3.2.- Multiplicación de cuaterniones

Sean los cuaterniones (t, X) y (t', X') donde X y X' son vectores y t y t' son escalares. La multiplicación de ambos se define como:

$$(t, \vec{X})\left(t', \vec{X'}\right) = \left(t\,t' - \vec{X}.\vec{X'},\ t\,\vec{X'} + \vec{X}\,t' + \vec{X} \times \vec{X'}\right)$$

donde . es el producto escalar y x es el producto vectorial.

Nota: un cuaternión se puede representar por 4 componentes pero también se puede ver como dos, al igual que un ángulo eje. En ese caso se entiende que el vector X está formado por los tres últimos elementos del cuaternión.

2.3.1.3.3.- Cambio de cuaternión a ángulo eje

Es el paso inverso de lo indicado en el paso 1. Sean angle y vector de nuevo:

```
angle = acosf(angle) * 2;
Sea sn = 1 / sinf(angle);

vector.x = vector.x * sn;
vector.y = vector.y * sn;
vector.z = vector.z * sn;
```

En el caso de valores de sn muy próximos al cero, es aconsejable devolver un ángulo eje nulo, para evitar la división por cero.

2.3.1.4 Ampliando el exportador de ASES : el sdk del 3dstudio Max

El SDK (software development kit)

El 3dstudio max es un programa de diseño 3d muy utilizado por los desarrolladores de videojuegos o de aplicaciones 3d en general. Es por ello que los desarrolladores decidieron incorporarle una manera de que los programadores que lo utilicen puedan incorporarle todo tipo de añadidos. Para ello, con cada versión del programa sale a su vez lo que se conoce como un sdk, un conjunto de librerías y cabeceras para poder desarrollar aplicaciones a modo de "pluggins" para el programa.

Las utilidades de este tipo de pluggins son inmensas y es por ello que existen infinidad de páginas en internet con aplicaciones desarrolladas por multitud de usuarios en todo el mundo. Se pueden realizar modificadores propios para el programa (de hecho algunos de los que vienen por defecto son pluggins), exportadores, importadores... etc.

Dichos pluggins poseen el formato de una librería de enlace dinámico (dll) corriente de Windows, pero para distinguirlas, la extensión que tienen es *.dle. Para añadirlas al programa simplemente hay que incluirlas en el directorio "pluggins" del programa y al arrancar la incluirá automáticamente.

No hay mucha documentación sobre el SDK del Max, al menos gratuita. Pero dicho software viene acompañado de [35] y [36]. [35] es un documento muy teórico y más bien es una guía de referencia. [36] es un FAQ en el que hay una gran cantidad de ejemplos, muy aconsejables.

Ampliando el exportador de ASES

Una vez que se sabe como funcionan los ficheros ASE uno puede estar satisfecho o bien puede querer exportar más datos que dicho fichero no facilita. Como ya se indicó en el apartado que describe el funcionamiento de estos ficheros, el exportador de ASES no facilita todos los datos de la escena del Max, sino sólo unos cuantos. En el caso de este proyecto se necesitaba exportar de algún modo animación de textura, cosa que en principio no nos facilita. Para ello, precisamente, se decidió recurrir al SDK.

Tipos de animación de textura

Bien, pues no se puede exportar algo sin saber como funciona. Se quiere exportar animación de textura, ¿sólo hay una manera de hacerlo?. Pues no, hay muchas. Por ejemplo:

- El propio fichero ASE tiene una opción para exportar en cada frame toda la malla del objeto, útil para cuando en vez de animar por matrices se anima vértice a vértice (aunque sin duda el tamaño del fichero crece de una manera increíble para mallas complejas). Dentro de la malla aparece la definición de las coordenadas de textura, para cada frame. Esta primera idea se desechó desde un principio, por la incomodidad de lectura del fichero y por el tamaño exagerado de los mismos.

- Se puede, por ejemplo, acceder a la animación del "gizmo" del objeto. El "gizmo" es un objeto 3d (un plano, un cubo, una esfera...) que el programa genera cuando se usa el

modificador UVW Map y que luego utiliza para mapear los objetos. Si se rota, traslada, etc; dicho gizmo, las coordenadas varían.

- Accediendo al modificador UVW Xform que no permite rotaciones de ningún tipo, pero permite avanzar en los ejes de textura u, v y w y también escalarlos (con el u, v, w offset).

Para este proyecto no se requería ningún tipo de rotaciones para las texturas, así que lo más sencillo es la última opción, y por eso precisamente es por la que se optó.

El nuevo exportador: ASZ

A continuación se indican los pasos que se siguieron para generar el exportador de ASZ, que básicamente son ficheros de tipo ASE pero que incluyen un tipo de animación de textura en al apartado "*TM_ANIMATION" mediante el modificador UVW Xform.

No se comenzará un pluggin desde cero, sino que, como se ha dicho, se retocará el exportador de ASES. El código de dicho exportador se distribuye con el propio sdk, en la carpeta "samples\impexp\asciiexp". El proyecto de dicha carpeta está generado con el visual c++ 6.0, así que si se está utilzando dicho compilador, simplemente se puede coger tal cual está y ampliarlo, que es lo que se hará aquí.

2.3.1.4.1.- Incluyendo en el proyecto los ficheros necesarios.

Al proyecto indicado hay que añadirle los siguientes ficheros:

- Ficheros del modificador UVW Xform: en la carpeta "samples\modifiers\" aparecen todos los modificadores de los que hace uso el programa. En este caso nos hacen falta los ficheros **uvwxform.h y uvwxform.cpp**

- Ficheros incluidos por dicho modificador: **mods.h** y **mods.cpp** que a su vez incluyen a **modsres.h**

Se recomienda no hacer un #include al directorio del maxsdk sino copiar dichos ficheros a la carpeta del proyecto e incluirlos desde allí. De este modo si cambia la ubicación del sdk en el disco duro no habrá problemas para seguir compilando.

2.3.1.4.2.-Generando un ID para el exportador

Lo primero que se debe realizar antes de comenzar a programar un pluggin para el max es generarle un ID que lo identifique. Este ID debe ser único, si no, el programa al arrancar nos avisará de que hay algún tipo de conflicto, no cargando el pluggin. Para generar un ID para el pluggin, se incluye en la carpeta del maxsdk (en el directorio "help") un programa llamado gencid.exe.

Entonces, en el asciexp.cpp se debe buscar la siguiente línea:

```
// Class ID. These must be unique and randomly generated!!
// If you use this as a sample project, this is the first thing
// you should change!
#define ASCIIEXP_CLASS_ID  Class_ID(0x85548e0b, 0x4a26450c)
```

Y sustituir dicha ID por la generada anteriormente. Conviene además acceder al archivo de recursos asciiexp.rc y modificar los valores de la String Table para no confundir luego un exportador con otro (esto no es obligatorio hacerlo, pero si no se hace, se verán dos exportadores idénticos a primera vista y no habrá manera de saber cuál es el nuevo)

2.3.1.4.3.- Accediendo y exportando el modificador UVW Xform

Bueno, pues todo parece indicar que la parte que exporta toda la animación de un objeto se encuentra en el fichero animout.cpp, así que será este el fichero que hay que retocar.

Investigando un poco dicho archivo se pueden ver las siguientes funciones y lo que hacen:

- ExportAnimKeys : es la función principal y la que escribe la animación en el fichero
- CheckForAnimation: Comprueba si hay algún tipo de animación, devolviendo true o false en cada caso
- DumpPosSample: escribe los samples de la animación de posiciones (la posición en cada frame)
- DumpRotSample: lo mismo que el anterior pero para las rotaciones
- DumpScaleSample: igual, pero para los escalados (en el presente proyecto la animación de escalado, por ejemplo, no está soportada)

El resto de funciones carecen de interés, para lo que se va a hacer a continuación. Son funciones para la exportación a modo de keyframes en vez de samples, que cae fuera de los ámbitos del proyecto.

Lo primero que se hará es una función llamada "CheckForUVWXForm" que comprueba si un objeto presenta este tipo de modificador o no. Para ello se le debe pasar el Inode (nos lo da la función ExportAnimKeys) y comprobar si en su lista de modificadores tiene presente el UVW Xform mediante llamadas sucesivas a la función "GetModifier" y comprobando si el modificador devuelto coincide con éste. Se tiene lo siguiente:

```
BOOL ZaloAsciiExp::CheckForUVWXForm(INode* node)
{
    BOOL hasUVWXF = false;

    IDerivedObject *pDerivedObject = (IDerivedObject *)node->GetObjectRef();
    if (pDerivedObject != NULL)
    {
        for (int i = 0 ; i < pDerivedObject->NumModifiers() ; i++)
        {
            Modifier *pMod = pDerivedObject->GetModifier(i);

            if (pMod->ClassID() == Class_ID(UVW_XFORM_CLASS_ID, 0))
            {
                hasUVWXF = true;
            }
        }
    }

    return hasUVWXF;

}
```

La siguiente función a realizar es la que se encarga de escribir en el fichero ASZ la animación de dicho modificador. Presenta el siguiente aspecto:

```
void ZaloAsciiExp::DumpUVWXFormSample(INode* node, int indentLevel)
```

```
{
    IDerivedObject *pDerivedObject = (IDerivedObject *)node->GetObjectRef();
    if (pDerivedObject != NULL)
    {
        for (int i = 0 ; i < pDerivedObject->NumModifiers() ; i++)
        {
            Modifier *pMod = pDerivedObject->GetModifier(i);

            if (pMod->ClassID() == Class_ID(UVW_XFORM_CLASS_ID, 0))
            {
                //El modificador ha sido localizado, lo exportamos
                TSTR indent = GetIndent(indentLevel);

                fprintf(pStream,"%s\t\t%s{\n", indent.data(), "*CONTROL_TXT_TRACK");

                TimeValue start = ip->GetAnimRange().Start();
                TimeValue end = ip->GetAnimRange().End();
                TimeValue t;
                int delta = GetTicksPerFrame() * GetKeyFrameStep();

                UVWXFormMod *uvw = (UVWXFormMod*)pMod;

                float u, v, w;
                float old_u, old_v, old_w;
                for (t=start; t<=end; t+=delta)
                {
                    old_u = u; old_v = v; old_w = w;
                    uvw->pblock->GetValue(PB_UOFFSET, t, u, FOREVER);
                    uvw->pblock->GetValue(PB_VOFFSET, t, v, FOREVER);
                    uvw->pblock->GetValue(PB_WOFFSET, t, w, FOREVER);

                    if(t == start || old_u != u || old_v != v || old_w != w)
                    {
                        fprintf(pStream, "%s\t\t\t%s %d\t %f\t%f\t%f \n",
                            indent.data(),
                            "*UVW_XFORM",
                            t,
                            u, v, w);
                    }
                }

                fprintf(pStream,"%s\t\t}\n", indent.data());
            }
        }
    }
}
```

La primera parte de esta función es muy similar a la anterior, se busca mediante la función GetModifier(i) el modificador UVW Xform, y una vez se encuentra, se realiza su exportación.

Para que la escritura del fichero sea coherente con el resto de cosas exportadas por el ASE, se ha imitado a las funciones DumpPosSample y DumpRotSample. De ese modo, se usa la función GetIndent(), que da el nivel de indentación actual (número de tabulaciones en el margen izquierdo) y se ha decidido iniciar el campo de la animación de textura mediante la palabra "*CONTROL_TXT_TRACK" y encerrar entre llaves la animación.

Del mismo modo, se cogen los valores para el tiempo, exactamente como está en dichas funciones, y para cada unidad de tiempo se crea una línea como la siguiente:

```
*UVW_XFORM 0     0.000000   0.000000    0.000000
```

El primer valor es el del tiempo, que se obtiene como se ha dicho. Los tres siguientes corresponden a la u, v y w de la animación de textura en ese frame. Indican cuánto se han trasladado los vértices de textura en dicho frame y eso es fácilmente implementable en openGL mediante la matriz de textura, y realizando una traslación.

La forma de acceder a esos tres valores es, una vez que se tiene el modificador UVWXform:

uvw->pblock->GetValue(PB_UOFFSET, t, u, FOREVER);

Esto está sacado de la función `LocalValidity` de uvwxform.cpp donde se hace algo muy similar. Si se consulta la ayuda del max sdk, se ve que la función GetValue de un IParamBlock recibe los siguiente parámetros:

int i
Dentro de la estructura ParamBlockDesc, el índice al elemento que se quiere acceder. En este caso, se accede a PB_UOFFSET, PB_VOFFSET, PB_WOFFSET.

TimeValue t
El instante para el que se quieren consultar los datos

float v
La variable donde almacenar los datos, es un parámetro de salida

Interval &ivalid
El intervalo a actualizar

Una vez se tienen estas dos funciones, simplemente hay que modificar un par de lineas en la función `ExportAnimKeys:`

- Sustituir:

```
else if (CheckForAnimation(node, bPosAnim, bRotAnim, bScaleAnim)
```

por esto otro, donde se comprueba además si hay animación de textura:

```
else    if(CheckForAnimation(node,    bPosAnim,    bRotAnim,    bScaleAnim)        ||
CheckForUVWXForm(node) )
```

- Incluir:

```
DumpUVWXFormSample(node, indentLevel);
```

después de `DumpScaleSample`, una vez que ya se han exportado las traslaciones, rotaciones y demás.

Con todo lo realizado, sólo queda compilar. Entonces se generará un *.dle que habrá que incluir en el directorio pluggins del 3dstudio max. Si se ha hecho todo correctamente, el programa detectará sólo el nuevo pluggin, y en la opción de exportar se encontrará un nuevo tipo de fichero llamado *.ASZ

2.3.1.5. La escena 3d

En el presente capítulo se explicará cómo funciona lo que es el núcleo de la parte gráfica del chat3d. Se verá cómo está organizada una escena (cámaras, metaobjetos... etc) Se hará primero una breve explicación de cada clase, para pasar después a comentar como se interrelacionan éstas entre sí.

Se comentan a continuación las clases de las que se hace uso mediante los componentes de la escena.

- **Matrix**

Representa una matriz de transformación 4x4. Posee todas las funciones para trasladar, rotar... etc. Una matriz asociada a un objeto define su orientación y su posición en el espacio 3d.

• **Vertex**

Un vértice es un punto en el espacio. Los vértices se agrupan de tres en tres formando caras (triángulos) de una malla.

• **Vector**

Un vector consta de módulo , dirección y sentido. Su representación es idéntica a la de un vértice. La clase vector posee funciones para normalizar, producto escalar, vectorial... etc.

• **Face**

Como ya se indicó, las caras de este proyecto son triángulos. Una cara está definida, por tanto, con tres vértices en sentido antihorario.

• **Textura**

La textura es una imagen que se pega sobre un objeto 3d para hacerlo más realista. No se ha implementado en el proyecto ninguna clase para leer formatos de textura, sino que se ha utilizado la librería devIL, que lee una amplia variedad de formatos, siendo por tanto muy recomendable. Ver [44].

• **Material**

El material define las propiedades ambiente, difusa y especular de la superficie de un objeto. Además de esto, indica también que textura tiene asociada, en caso de tenerla.

• **IAnimation**

Se profundizará después en esta clase. Digamos que la clase IAnimation lo que hace es, de acuerdo al tiempo transcurrido, transfomar las matrices de modelado y textura del objeto, permitiendo así tener un mundo muy animado.

Las clases principales de la escena 3d son las siguientes:

• **Scene**

Es la clase principal, de la que cuelgan el resto de clases. Básicamente posee dos métodos fundamentales: mover y pintar. Mediante el método Move se invoca a todos y cada uno de los objetos para que realicen su animación jerárquicamente y, mediante el método Paint se pintarán todos de manera ordenada.

• **MetaObject3D**

Un metaobjeto, como su propio nombre indica, agrupa a un conjunto de objetos (Object3D) en forma de árbol. Cada metaobjeto posee un método "mover", mediante el cual provoca al movimiento de todos sus hijos, pero no posee un método pintar ya que los metaobjetos se pueden considerar como nodos vacíos (no poseen una malla ni materiales).

• **Object3D**

Un objeto es la parte más pequeña en la que se puede dividir una escena. Se podría decir que una escena está compuesta por un montón de objetos relacionados jerárquicamente entre sí. Un objeto puede ser, por ejemplo, un pie de un personaje, o una mano. Los objetos, como tales, poseen malla y materiales, es decir, se pintan en pantalla. Cada objeto además posee una animación.

2.3.1.5.1.- La clase Object3D

Un objeto 3d define su malla a través de los siguientes componentes:

- Un vector de vértices : definido mediante la clase std::vector. Es importante saber por qué se ha utilizado esta estructura de datos. En el proyecto se han utilizado vertex buffers, por lo que es importante que toda la información de los vértices aparezca seguida en memoria. Esto no es posible con otras estructuras como por ejemplo la clase std::list. La ventaja principal que tiene la clase std::vector frente a los arrays convencionales es que se conoce en todo momento su tamaño, evitando así el tener una variable como podría ser num_vertex, o alguna similar.

- Un vector de normales: donde cada normal se refiera a la normal al vértice que se encuentra en el vector de vértices en la misma posición. Dichas normales, deben estar normalizadas, como su propio nombre indica

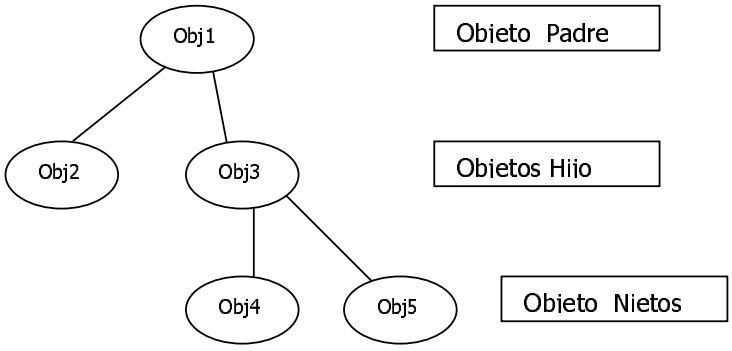
- Un vector de caras: donde cada cara está definida por tres enteros. Cada entero indica un índice del vector de vértices. Esta es otra de las razones por las que se usa la clase vector. Con la clase std::list el accedo aleatorio es lineal, por lo que acceder, por ejemplo al vértice 23, puede ser algo costoso

- Un vector de vértices de textura: similar al de vértices, pero refiriéndose a las coordenadas de textura. Es importante el hecho de que estos dos vectores deben referirse en las mismas posiciones a los mismos vértices, puesto que si no no se podrían usar vertex buffers. Para conseguir este resultado, ver el algoritmo de fusionado de malla y textura

- Un vector de caras de textura: igual que el de caras de la malla, pero refiriéndose a las caras de textura. El vector de caras de textura y de caras de malla acaba siendo el mismo, tras la fusión de ambas, por lo que no es necesario realmente.

Además, todo objeto debe tener asociado un material, que define sus propiedades de iluminación y la textura que tiene asociada, si es que tiene dicha textura. En el caso de que el material no tenga asociada una textura, el objeto no poseerá vértices de textura. Además, un objeto con textura puede tener canal alpha o no, dependiendo de las propiedades de ésta.

Cada objeto se encuentra situado en un árbol de objetos, es decir, que cada obeto debe implementar la clase NodeTree, que se explicará más adelante. De este modo los objetos se relacionan entre si, formando un árbol, del siguiente modo:



Las coordenadas de cada objeto en la jerarquía vienen dadas repecto a su padre. De esto modo, por ejemplo, las del objeto 5, vendrían dadas respecto a las del objeto 3, que a su vez vienen dadas respecto a las del 1.

2.3.1.5.1.1.- La animación de un Object3D

Todo Object3D lleva asociada una animación del tipo IAnimation, definida mediante el siguiente interfaz:

```
class IAnimation
{

public:
    virtual void SetMatrix(const Matrix &m) = 0;

public:
    virtual const Matrix& GetMatrix()const = 0;

public:
    virtual ~IAnimation(){};

public:
    virtual void Move(DWORD time, Matrix &mat, Matrix &texture_mat) = 0;
};
```

Es decir, que toda animación deberá poseer una matriz y la función Move, que dado el tiempo transcurrido desde que comenzó la animación y las matrices de animación y textura, transforme éstas. En el presente proyecto se definió, siguiendo esta interfaz, la clase AnimationPosRot que, como su nombre indica, define transformaciones de posición y rotación.

2.3.1.5.2 - La clase AnimationPosRot

Los atributos de esta clase son:

- ticks: el número de milisegundos que transcurren entre un frame y otro
- num_frames: el número de frames que tiene la animación
- matrix_ini: la matriz que indica la orientación y posición en el frame 0
- positions: es un std::vector que en cada frame indica la posición en la que debe encontrarse un objeto que posea esta animación
- rotations: es un std::vector que en cada frame indica la rotación que sufre un objeto (en ángulo eje, como openGL) respecto a matriz_ini (se carga dicha matriz y se rota tanto como indique la rotación correspondiente)
- texture_positions: indica en cada frame, cuanto hay que trasladar la matriz de textura del objeto.

Nótese que se han implementado posiciones y rotaciones, pero no escalado. No hay ninguna razón en especial, simplemente que los escalados no se iban a utilizar, pero se podrían añadir sin problemas, siguiendo la interfaz. También se puede observar que no hay rotaciones en la matriz de textura, por la misma razón: no se iban a utilizar. Además, la rotación en texturas es un poco complicada de exportar desde el 3dstudio Max ya que no hay ningún modificador que la realice, no como para el caso de la traslación (mediante UVW Xform).

La animación funciona pues del siguiente modo: en un determinado momento, un objeto llama a su animación pasándole sus matrices y el tiempo transcurrido desde el frame 0. La animación busca, para cada tipo de transformación (rotación y traslación, y traslación de textura), si posee animación de dicho tipo. Si la posee, la aplica, transformando las matrices.

Cada frame, de los que se han venido hablando hasta ahora, está definido mediante la interfaz Animation. Definida básicamente con: nºde frame y propiedades del frame:

```
class Animation
{
protected:
    int frame;

… //Accesores y destructor
};
```

Según el frame sea de rotación o de posición vendrá definido mediante AniPos o AniRot repectivamente:

```
class AniPos : public Animation
{
private:

    /**Vértice que indica la posición en el frame actual*/
    Vertex pos;

... //Métodos accesores, contructor y destructor
}
```

Como se ve, en cada frame un vértice nos indica la posición del objeto.

```
class AniRot : public Animation
{
private:

    /**El ángulo de rotación*/
    float angle;

    /**El vector*/
    Vector vector;

... //Métodos accesores, contructor y destructor

}
```

Aquí, una rotación ángulo-eje está definida por un ángulo y un vector.

El acceso a cada uno de los frames se hace mediante una búsqueda binaria, ya que los vectores de frames están ordenados. Para eso mismo, se definen las funciones de Find y FindR (no tiene mucho sentido el copiar aquí dicho algoritmo, de sobra conocido en el mundo de la programación).

2.3.1.5.3.- La clase Camera

Las cámaras son metaobjetos. Pero se distinguen de estos porque tienen los siguientes atributos:

- camera_near: indica la distancia del plano cercano.

- camera_far: indica la distancia del plano lejano.

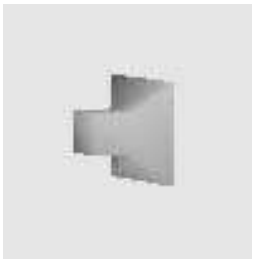
- camera_fov: indica el ángulo del foco de la cámara.

- camera_aspect: es una variable estática. Indica la relación ancho/alto. Debe ser ajustada cada vez que se haga un cambio en el tamaño de la ventana.

Además las cámaras poseen los siguientes métodos:

- void ChangeGLProyection():
  Mediante esta función se cambia la matriz de proyección de openGL para que se ajuste a los parámetros de la cámara. Básicamente es una llamada a gluPerspective con los valores de sus atributos.

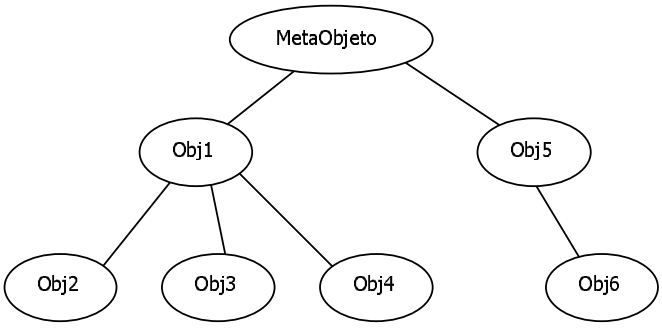
- void SetView(Matrix &model_matrix)
  Sitúa la vista en la cámara. O lo que es lo mismo, sitúa en model_matriz la inversa de su matriz (recuérdese que una cámara es un Objec3D y que por tanto tiene una matriz de orientación).



A la hora de mover y pintar una cámara estas se comportan del mismo modo que los Object3D. De hecho, en este proyecto, se les ha definido una malla por defecto para poder verlas cuando no son la cámara con la que se está viendo la escena.

2.3.1.5.4.- La clase MetaObject3D

Un metaobjeto define ciertas propiedades para todos los objetos que están por debajo de él en el árbol de objetos. El aspecto de un metaobjeto es el de un árbol, donde el propio metaobjeto es el propio nodo raíz y los hijos son nodos internos:



Las propiedades que afectan a todos los hijos que se encuentran debajo de un metaobjeto son las siguientes:

- La matriz: al igual que los Object3D un metaobjeto tiene una matriz, por lo que las posiciones de los hijos estarán dadas respecto a esta matriz

- animations_number: indica el número de animaciones que poseen todos los objetos situados por debajo. Cada animación es del tipo AnimationPosRot, como ya se indicó antes

- ani_actual: indica la animación en curso. Todos los objetos de un metaobjeto se deben encontrar realizando la misma animación (el mismo número de animación). Basta con cambiar esta variable para que cambie en todos los hijos (ejemplo: una animación puede ser andando, y otra corriendo)

- std::vector< int > animation_length: este vector indica en cada posición lo que dura la animación que se encuentra en la posición correspondiente

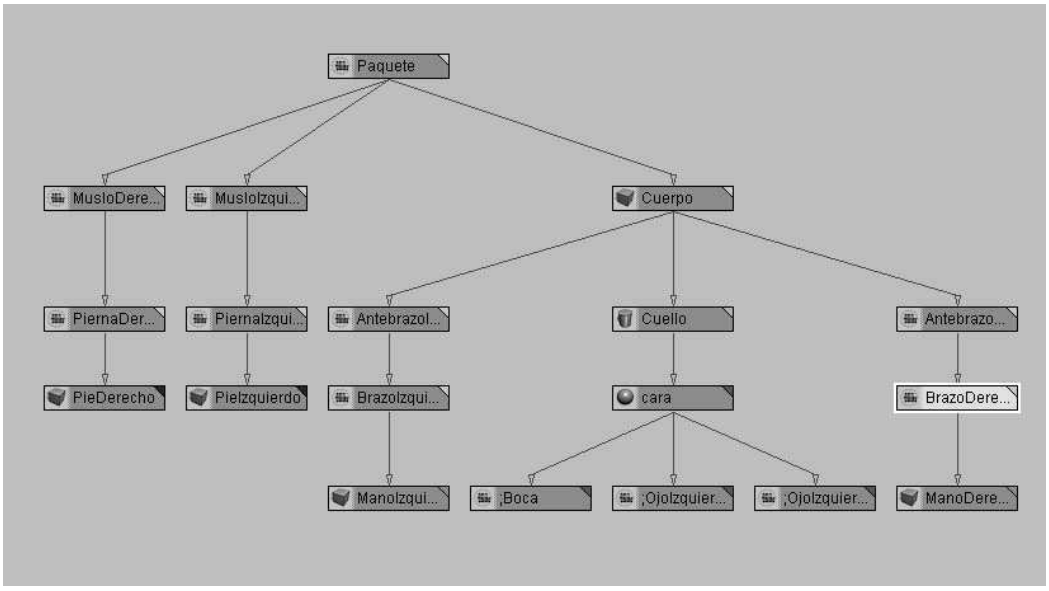
- time : indica el tiempo transcurrido desde que se hizo un reset a la animación (desde que se ejecutó el frame 0 de la presente animación). De este modo, es fácil calcular cuál debe ser el frame actual (si time es 3000 el frame será el que toca en ese tiempo)

Un metaobjeto manda mover a todos sus hijos mediante la siguiente función:

```
virtual void Move(DWORD time, const Matrix &model_matrix, int ani_actual = 0)
```

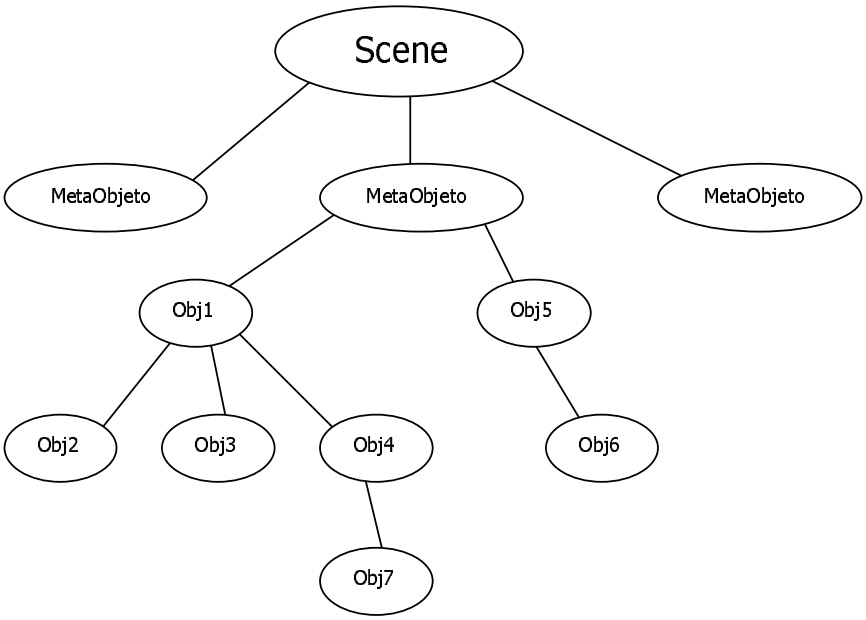
Aquí, time es el tiempo transcurrido desde el último frame, por lo que se sumará a time local y se pasará ese tiempo a los hijos. La matriz model_matriz es la matriz de modelado actual (cámara) y ani_actual la animación que se quiere que se ejecute.

Un ejemplo de un metaobjeto es el muñeco utilizado para el chat:



2.3.1.5.5.- La clase Scene

La clase Scene está compuesta por metaobjetos. La visión global de una escena, por tanto, es la de un árbol en dónde, en el primer nivel se encuentra el nodo con la escena en sí. En el segundo nivel estarían todos los metaobjetos, y del segundo nivel en adelante se encuentran los Object3D



Los dos métodos principales de la clase Scene son: pintar y mover

• El método Move(DWORD time)

Este método mueve toda la escena el tiempo indicado por el parámetro (en milisegundos). Para ello, va llamando al método paint de cada metaobjeto. Estos a su vez llaman a los de cada objeto y estos, del mismo modo, a sus objetos hijos. De este modo se acaban moviendo todos los elementos de la escena.

• El método Paint()

Manda pintar a los objetos de una manera ordenada. Para comprender mejor esta función hay que echar un vistazo a las listas de Object3d definidas en la clase Scene:

- material list: es una lista de materiales. Cada material contiene una referencia a todos los objetos que lo están utilizando, de este modo, se pueden pintar todos los objetos que tienen un mismo material seguidos, sin hacer ningún cambio. La lista de materiales tiene tres partes distinguidas:
    - Materiales sin ningún tipo de textura: este tipo de materiales sólo definen las componentes ambiente, difusa y especular y para pintar los objetos que los utilizan. No debe estar activado ni el pintado con textura ni el blending.
    - Materiales con textura sin canal alfa: los materiales de este tipo poseen textura, pero dicha textura no tiena canal alfa, por lo que a la hora de pintar es bastante aconsejable no tener activado el blending (eficiencia).
    - Materiales con textura con alfa: este tipo de material posee textura, y además su textura tiene canal alfa, por lo que el blending debe estar activado.
- objects with alpha: contiene todos los objetos que poseen un material cuya textura tiene alfa. A continuación se explicará el por qué de esta lista.
- camera list: una lista con todas las cámaras de la escena. Toda cámara de la escena que se quiera seleccionar debe aparecer en esta lista. No basta con que esté definida en la jerarquía de objetos. El parseador de ASES que se ha hecho para el proyecto se encarga de que esto sea así, por lo que no hay que preocuparse.

Bien, pues a continuación se puede ver cómo es el algoritmo de pintado de toda la escena:

1. En primer lugar se coge la matriz de la cámara que esté seleccionada en ese momento (si la hay, si no admitimos que la matriz de la cámara es la identidad) y se invierte. De este modo se obtiene la matriz de modelado por la que se irán multiplicando los objetos de la escena.
2. A continuación, se procede al pintado de los objetos que no tienen textura. Para ello se accede a la lista de materiales y se van seleccionando uno a uno, pintando todos los objetos que los utilizan. Como se ve, de este modo el cambio de materiales es mínimo.

3. Lo siguiente es activar las texturas en la tarjeta y se pintan todos los objetos que contienen textura, pero sin canal alfa. Para ello, se procede como en el caso anterior. Los materiales con textura están situados a continuación de los que no la tienen.

4. El paso siguiente es el pintado de los objetos con canal alfa. Nótese que estos objetos deben pintarse al final, para que el blending quede en condiciones, y, además, estos objetos entre sí hay que pintarlos desde atrás hacia delante. Es decir, los objetos que se encuentren más lejos de la cámara deben pintarse antes. Por esta razón, este tipo de objetos no se puede pintar como los anteriores, seleccionando un material y pintando todos los que lo usan. En cada pintada debemos:

    - Ordenar dichos objetos según su profundidad
    - Pintarlos.

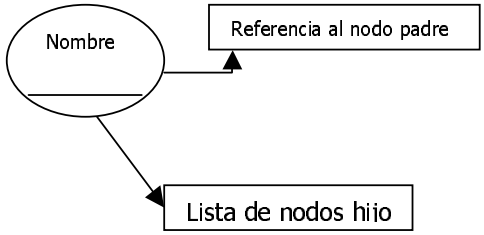
    Puesto que el uso de objetos con blending es bastante costoso y la ordenación de los objetos de atrás hacia delante no soluciona de todo el problema del blending en algunas partes, no es aconsejable abusar de este tipo de objetos

Queda pendiente comentar por qué los en la lista de materiales se almacenan también aquellos con textura con canal alfa, ya que, como se ha visto, el pintado de este tipo de objetos se hace sin tener en cuenta la lista de materiales. La razón es simplemente, porque en algún lugar se debe almacenar una referencia a dichos materiales. Al usar smart pointers, si no se hiciera esto así, se perdería la referencia con este tipo de materiales, eliminándolos de la memoria y provocando un completo caos.

Un ejemplo de Scene es por ejemplo la siguiente ventana, donde están definidos 3 metaobjetos: los dos muñecos del chat y el mundo. Los árboles no son metaobjetos, sino que son objetos en jerarquía con el mundo

2.3.1.5.6.- La clase NodeTree

Como ya se ha visto, la estructura que forman los objetos entre sí, con los metaobjetos y la escena, es la de un árbol. En la librería estándar de C++ los árboles no aparecen definidos, por lo que se han tenido que implementar. Básicamente la estructura del nodo de un árbol es:




```
class NodeTree
{
protected:

    std::list< boost::shared_ptr< NodeTree > > sons;

    std::string name;

    NodeTree *parent;

            ... //resto de funciones
     }
```

- sons: es una lista de los nodos hijo. Todos los nodos de esta lista son hermanos entre sí
- name: el nombre del nodo, para, como se verá a continuación, poder realizar búsquedas
- parent: una referencia al nodo padre, de este modo se pueden realizar también recorridos ascendentes

De este modo, un árbol completo, estará formado por un conjunto de nodos enlazados unos con otros mediante la lista de hijos. El nodo raíz tendrá una referencia a null en parent y los nodos hoja tendrán la lista de hijos vacía.

Uno de los métodos más importantes en un árbol es el método:

```
boost::shared_ptr< NodeTree > Find(const std::string &node_name)const;
```

Que se encarga de buscar por todos los hijos que se encuentran por debajo del nodo, uno cuyo nombre sea el indicado por parámetro. La búsqueda realizada es en profundidad.

2.3.1.5.7.- La clase InodeMovable

Los nodos del árbol de nuestra escena no son del tipo NodeTree simplemente, sino que implementan el siguiente interfaz:

```
class INodeMovable : public NodeTree
{
public:
    virtual ~INodeMovable(){}
```

```
public:
   virtual const Matrix& GetActualMatrix()const = 0;
   virtual void Move(DWORD time, const Matrix &model_matrix, int ani_actual) = 0;
   virtual void MoveHierarchy(DWORD time, const Matrix &model_matrix, int ani_actual)
= 0;
};
```

Es decir, que todos ellos poseen la función para realizar un Move, y una matriz. De este modo la estructura del árbol queda más definida. Tanto los Object3D como los MetaObject implementan esta interfaz. La clase Scene, que se considera el nodo raíz, no la implementa. La razón de esto es que la clase Scene no posee ninguna matriz.

2.3.1.6. Los bocadillos

En el siguiente capítulo se tratará todo el tema referente a los bocadillos del chat. Se comenzará comentando cómo introducir texto en una ventana de opengl (recuérdese que esto en principio no es posible) para, a continuación comentar qué son los pbuffers y para que se han usado. Finalmente, se explicará con todas estas nociones como se implementaron los bocadillos.

La idea

¿Quién no ha leído alguna vez un cómic? Esperamos que nadie responda "yo" a esta pregunta. Cuando se meditó acerca del chat hubo una cosa en la que todos estuvimos de acuerdo en un principio: las frases que decían los muñecos debían ir en un bocadillo estilo cómic, donde las letras fuesen apareciendo una a una para que diese la sensación de que el muñeco que las pronuncia estuviera hablando. Finalmente se puede apreciar que esto se ha conseguido satisfactoriamente.

Además de salir las letras, un efecto curioso que tiene este chat3d es el de lo que se conoce como emoticonos. Los emoticonos, también llamados iconos textuales, han venido siendo utilizados en la escritura para ser el equivalente a los gestos en una conversación hablada. Han sido utilizados en los chats desde que estos aparecieron. Pero claro, al disponer aquí de un mundo en tres dimensiones con muñequitos que hablan, se podía aprovechar el tema para convertir los emoticonos en gestos del muñeco. Así se hizo; de este modo, cuando un bocadillo detecta la escritura de un emoticono pone al muñeco correspondiente en la animación gestual adecuada

2.3.1.6.1.- Textos en opengl

Como bien saben los programadores de opengl o cualquier otro entorno gráfico, en una ventana no se pueden utilizar las funciones de salida estándar para escribir texto. Esto lleva a muchas personas a pensar que es imposible escribir en una ventana, y eso no es así. Lo que pasa es que hay que echarle un poco de imaginación.

Antes de seguir, comentar que, efectivamente, la librería glut [43] posee funciones para escribir en una ventana. No obstante dichas funciones son un poco limitadas ya que solo dejan elegir entre unas pocas fuentes y no parece que sea una función del todo eficiente (por experiencia). Además, mucha gente tiene problemas con esa función porque no es tan mágica como algunos piensan, y hay que desactivar estados de la tarjeta y otras complicaciones que no vienen al caso. Desde aquí desaconsejamos por completo el uso de dichas librerías salvo para hacer algún tipo de prueba rápida.

Bien, ¿entonces cómo escribir textos en una ventana? La solución es fácil. Supóngase que cada carácter de una frase es un cuadrado (es decir, un polígono de los de opengl de 4

vértices). Escribiendo una serie de cuadrados seguidos donde cada uno sea una letra se tiene una frase, y escribiendo varias frases de este tipo se tienen líneas. Así de sencillo. Existen otras formas de renderizar texto en una ventana, pero según [42] y diversos foros de internet, esta es la más aconsejable.

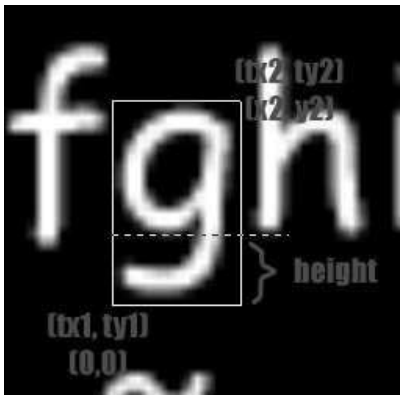
Evidentemente, para definir los caracteres no se hace a mano. Se utiliza el editor de textos que se ha implementado exclusivamente para el proyecto, donde todos los caracteres usan la misma textura, pero cada carácter define un cuadrado en la textura que engloba a su letra correspondiente. Un ejemplo:

2.3.1.6.1.1.-La clase GLCharacter

Como el propio nombre indica, esta clase define una letra. Una letra viene definida por los sigientes atributos:

- x1, y1: definen el ancho y el alto de la letra respectivamente
- tx1, ty1, tx2, ty2: definen las coordenadas de textura del carácter
- height: indica la altura de la letra.

Estos conceptos se aprecian más claramente en la siguiente captura del editor de fuentes:



En este ejemplo se aprecia como estaría definida la letra g. El resto de caracteres ASCII se definen del mismo modo. Hacer esto a mano es poco cómodo y poco extensible si se quieren hacer varias fuentes con diferentes texturas, es por eso que se hizo el editor de fuentes.

El método más interesante de la clase es:

```
float Paint(float x, float y, float size = 1.0f);
```

Este método pinta un carácter tomando como extremo inferior izquierdo el indicado en x, y. Pinta la letra con el tamaño indicado (por defecto lo pinta a su tamaño natural). Lo más importante es el float que devuelve, que indica cuánto ha ocupado la letra de ancho. De este modo se sabe que la siguiente letra deberá situarse en x + el ancho devuelto. Este valor se

calcula en la propia función, también si el tamaño de la letra no es 1, por lo que nos libra de muchos problemas.

2.3.1.6.1.2.- La clase GLFONT

Esta clase define la forma más sencilla de pintar textos en opengl. Tiene definido un array de 256 posiciones, una por cada carácter del código ASCII, de modo que en cada posición se almacena una referencia a un GLCharacter. Por defecto todos las posiciones apuntan al GLCharacter de ' ' que es obligatorio que esté siempre. De este modo cuando se use una letra que no se haya definido se escribirá un espacio

En esta clase se tiene la textura común a todos los caracteres. Una textura de este tipo es, por ejemplo, la siguiente:

Con todo esto, se puede pasar a examinar detenidamente la función

```
void Write(const std::string &text, float x = 0, float y = 0);
```

Recibe por parámetros una frase en formato string y la posición donde pintar el texto. La forma de actuar es la siguiente: va recorriendo el string, y por cada carácter manda pintar al GLCharacter asociado y va avanzando en la x, como se mostró en el apartado anterior. De este modo cuando termina la función Write aparece en pantalla la frase escrita.

Se puede considerar entonces a esta clase como una línea donde se escriben letras: se pueden proyectar en dos dimensiones sólo, en tres creando bonitos efectos, animar las letras, aumentar los tamaños con las funciones de opengl, etc. La verdad es que una vez implementado algo tan sencillo, uno se da cuenta de las grandes posibilidades de una clase como ésta.

Esta implementación de textos que se ha visto es la más sencilla de todas. Ahora bien, ¿qué ocurre si una línea ocupa más que el ancho de la pantalla? Pues que evidentemente, no salta a la siguiente, lo mismo que si encuentra un carácter de \n. Todas estas cosas hay que programarlas. En el proyecto se programó una escritura en un rectángulo cíclico. Este tipo de escritura está definido en la clase SquareTimerText.

2.3.1.6.2.- La clase SquareTimerText

Como se ha dicho, la clase SquareTimerText define el tipo de escritura en un cuadrado cíclico. ¿Y esto qué es? Pues con ese término nos referimos a que cuando el texto llegue a una determinada altura (la altura del cuadrado) se vuelva a pintar por arriba. Es decir, si se va a pintar en un rectángulo de altura 100 y se elige la altura 150, se deberá pintar en la altura 50.

Y no sólo esto. Las letras en este tipo de escritura se van pintando poco a poco. Digamos que van apareciendo dando así la sensación de que se están escribiendo en ese momento, o de que se están pronunciando.

Pero además, esta clase debe reconocer los emoticonos. Por lo tanto, cuando reconozca un emoticono escrito en la frase que esté procesando en ese momento, debe indicarle al muñeco asociado que cambie a la animación correspondiente.

La implementación más cómoda para una clase que realiza tantas cosas es mediante una máquina de estados. Los estados definidos son:

- NOTHING: no está procesando ningún texto y por tanto no tiene nada que hacer. Si llega un texto pasará al estado WRITING

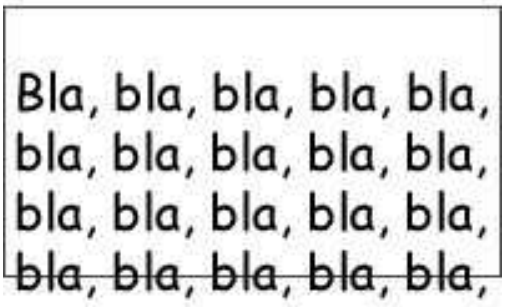
- WRITING: se está escribiendo un texto actualmente. En el caso de que se detecte fin de línea se pasa a END_OF_LINE. En el caso de que se detete un emoticono se pasa a EMOTICONING. Y, por último, en el caso de que la frase que se está procesando llegue a su fin, se pasa al estado NOTHING.

- END_OF_LINE: se ha localizado un fin de línea. Es decir, la palabra que se quiere escribir no cabe en la línea actual. Se está en espera que de que el bocadillo asociado termine la animación de salto de línea (más adelante se trata la clase de los bocadillos)

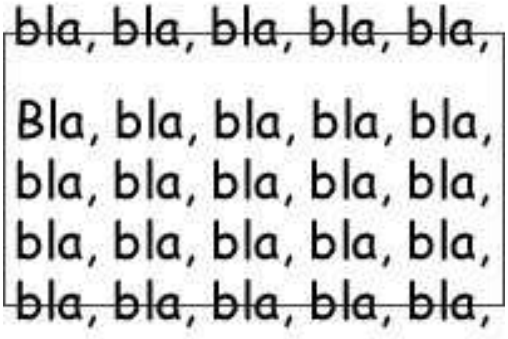
- EMOTICONNING: se ha localizado un emoticono y está en espera de que la animación del muñeco termine

La escritura en un SquareTimerText es distinta de la forma de pintar habitual en openGL. En este caso no se borra en cada frame todo lo que se halla pintado anteriormente y se pinta todo de nuevo, sino que se continúa con lo que se pintó en el frame anterior. El motivo de esto es porque está pensado para utilizarse con los pbuffers, que se explicarán a continuación.

Para simular el efecto cíclico no basta con localizar la posición y hacer módulo la altura del cuadrado. Hay un caso especial que hay que tratar, que es el caso en el que una línea no cabe por completo, bien en la parte superior o bien en la inferior. Esto se aprecia mejor en los siguientes dibujos:



Este ejemplo de la izquierda estaría mal hecho, porque se está perdiendo una parte de la línea de abajo. Lo que se debe hacer es pintar de nuevo la línea por arriba en la posición módulo h (la altura del cuadrado) de modo que por arriba aparezca el trozo de línea que se está cortando por abajo



La forma correcta de hacerlo es la que se muestra aquí a la izquierda. De esto modo la escritura es completamente cíclica. El motivo de querer esto es, que el texto se va a utilizar como textura cíclica, y puede que la parte superior y la inferior caigan en medio del polígono. Si esto

ocurre, no se notará, porque la parte de abajo encaja perfectamente con la de arriba

Ya se ha comentado la clase que se encarga de la escritura en un bocadillo. A continuación se explicará la clase Ballon, que es el bocadillo en si. Pero antes de eso, conviene conocer una extensión de opengl que se ha utilizado para su implementación y que son los pbuffers. Los pbuffers se han encapsulado dentro de una clase que hereda de textura y que se llama DrawableTexture

2.3.1.6.4.- La clase DrawableTexture:

Esta clase implementa un tipo de textura muy especial. Como su propio nombre indica, en una textura de este tipo se puede pintar. Es decir, igual que se renderiza en una ventana de opengl, esta clase permite que se pinte sobre una textura. Todo ello se hace por hardware, aprovechando la potencia de las tarjetas. Para ello se utilizan los pbuffers, que no están definidos en la API de opengl pero que son una extensión del tipo ARB por lo que se puede considerar que en un futuro aparecerá, o, al menos, muchas tarjetas del mercado actual las tienen implementadas.

Pasemos a ver los pbuffers, ¿qué son?, ¿cómo se crean y destruyen? y ¿cómo utilizarlos?

2.3.1.6.4.1- Los PBUFFERS

• Iniciando las extensiones

Para información sobre las extensiones de opengl y su utilización, se recomienda consultar [37]

Los pbuffers son una extensión de opengl que permiten el renderizado en una textura. Hay mucha gente que teme a las extensiones de opengl por varias razones:

- La primera de ellas es que son bastante engorrosas de utilizar, hay que reconocerlo.

- La segunda es que, como se ha dicho, no pertenecen a la API, y por tanto puede haber tarjetas que no las soporten. En especial hay que tener cuidado con las de un fabricante en particular, como puede ser Nvidia o Ati. No es aconsejable su uso para aplicaciones que se quiera que funcionen en varias tarjetas.

Lo segundo es algo que no se puede solucionar. Se debe avaluar a quién va destinada la aplicación que se está desarrollando y comprobar si trae cuenta o no su uso. A veces es mejor evitarlas e intentar suplirlas con la API.

Frente a lo primero, francamente, es aconsejable hacerse con una librería que ahorre el tener que ir definiendo los punteros a las diferentes funciones, buscándolas... etc. En nuestro caso se ha utilizado glew. Basta una llamada para que se inicialice, y captura todas las extensiones que soporta la tarjeta:

```
GLEnum glewInit();
```

En el caso de que no se pueda inicializar, devuelve un error que habría que analizar. También posee funciones para comprobar si la tarjeta del usuario soporta una determinada extensión.. etc. En el caso de los pbuffers se requieren las siguientes extensiones:

- WGL_ARB_extensions_string [40]
- WGL_ARB_render_texture [40] [38]
- WGL_ARB_pbuffer [25] [39]
- WGL_ARB_pixel_format [41]

• Creación de un pbuffer

Lo primero que se debe hacer es crear un objeto de textura mediante el procedimiento habitual, pero no se definirá el tamaño de ésta, ni el puntero a los bits. Es decir, se debe hacer algo como lo siguiente:

```
void DrawableTexture::CreateTextureObject()
{
    glGenTextures(1, &textId);
    glBindTexture(GL_TEXTURE_2D, textId);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
}
```

Como se ve, no hay ninguna llamada a GLTexImage2D, ni la debe haber.

A continuación, se puede crear el pbuffer (al haber hecho glBindTexture estará asociado al objeto de textura anterior). A continuación, hay que seguir los siguientes pasos:

1. Obtener el contexto del dispositivo: para ello se debe llamar a
   HDC hdc = wglGetCurrentDC();

2. Elegir el formato de píxel mediante wglChoosePixelFormat(). Antes de ello hay que definir dos listas de atributo/valor que deben acabar en 0/0. Dichos atributos aperecen especificados perfectamente en la documentación de la extensión WGL_ARB_pixel_format. Aquí se han utilizado los siguientes:

```
iattributes[2*niattribs  ] = WGL_DRAW_TO_PBUFFER_ARB;
iattributes[2*niattribs + 1] = GL_TRUE;
niattribs++;
iattributes[2*niattribs  ] = WGL_BIND_TO_TEXTURE_RGB_ARB;
iattributes[2*niattribs + 1] = GL_TRUE;
niattribs++;
iattributes[2*niattribs  ] = WGL_DEPTH_BITS_ARB;
iattributes[2*niattribs+1] = 24;
niattribs++;
iattributes[2*niattribs  ] = WGL_RED_BITS_ARB;
iattributes[2*niattribs+1] = 8;
niattribs++;
iattributes[2*niattribs  ] = WGL_GREEN_BITS_ARB;
iattributes[2*niattribs+1] = 8;
niattribs++;
iattributes[2*niattribs  ] = WGL_BLUE_BITS_ARB;
iattributes[2*niattribs+1] = 8;
niattribs++;

iattribs[0] = WGL_DRAW_TO_PBUFFER_ARB;
iattribs[1] = true;
iattribs[2] = WGL_DEPTH_BITS_ARB;
iattribs[3] = 24;
iattribs[4] = WGL_COLOR_BITS_ARB;
iattribs[5] = 24;
iattribs[6] = WGL_RED_BITS_ARB;
iattribs[7] = 8;
iattribs[8] = WGL_GREEN_BITS_ARB;
iattribs[9] = 8;
iattribs[10] = WGL_BLUE_BITS_ARB;
iattribs[11] = 8;
iattribs[12] = WGL_ALPHA_BITS_ARB;
iattribs[13] = 0;
```

```
iattribs[14] = WGL_SUPPORT_OPENGL_ARB;
iattribs[15] = true;
iattribs[16] = WGL_ACCELERATION_ARB;
iattribs[17] = WGL_FULL_ACCELERATION_ARB;
iattribs[18] = WGL_DOUBLE_BUFFER_ARB;
iattribs[19] = false;
iattribs[20] = WGL_BIND_TO_TEXTURE_RGB_ARB;
iattribs[21] = true;
iattribs[22] = 0;
iattribs[23] = 0;
```

Una vez definidos los atributos, se puede crear el formato de píxel para el pbuffer. Para ello se llama a:

```
wglChoosePixelFormatARB( hdc, iattribs, fattributes, max_num, pformat, &nformats )
```

donde:

- hdc es el contexto que se obtuvo en el paso 1.
- iattribs, fattributes son los atributos definidos.
- max_num es el máximo número de formatos compatibles con los atributos que se quiere que devuelva la función.
- pformat es un array de max_num donde se almacenan los formatos devueltos por la función
- nformats es el número de formatos compatibles con los atributos que se pasaron de entrada

3. Crear el pbuffer. Para ello, se debe definir de nuevo una lista de atributo / valor que acabe en 0, al igual que antes. Los distintos valores de dichos valores están recogidos en la documentación de la extensión de los pbuffers. En el proyecto se definieron los siguientes:

```
pattributes[0] = WGL_TEXTURE_FORMAT_ARB;
pattributes[1] = WGL_TEXTURE_RGB_ARB;
pattributes[2] = WGL_TEXTURE_TARGET_ARB;
pattributes[3] = WGL_TEXTURE_2D_ARB;
pattributes[4] = WGL_PBUFFER_LARGEST_ARB;
pattributes[5] = true;
pattributes[6] = WGL_MIPMAP_TEXTURE_ARB;
pattributes[7] = true;
pattributes[8] = 0;
pattributes[9] = 0;
```

Hecho esto se llama a:

```
HPBUFFER wglCreatePbufferARB(hdc, pformat[0], width, height, pattributes);
```

donde:

- hdc es el contexto obtenido en el paso 1
- pformat es el formato de píxel obtenido en el paso 2
- width, height son las dimensiones del pbuffer
- pattributes es la lista de atributos definida

Si todo va bien, nos devolverá el pbuffer

4. Se debe obtener el contexto de dispositivo del pbuffer mediante

```
pbuffer_device_context = wglGetPbufferDCARB(pbuffer);
```

5. Por ultimo se debe crear un contexto de opengl para el pbuffer con la siguiente función:

```
    pbuffer_GL_context = wglCreateContext(pbuffer_device_context);
```

• **Usando los pbuffers**

Para poder pintar en un pbuffer, el algoritmo a seguir es el siguiente

```
wglMakeCurrent(pbuffer_device_context, pbuffer_GL_context);

    //Comandos de opengl para pintar en el pbuffer

wglMakeCurrent(hwindc, hwinglrc);
```

donde hwindc, hwinglrc son el handle de ventana y el contexto de dispositivo de la ventana actual.

Para utilizar un pbuffer a modo de textura se usa:

```
glBindTexture(GL_TEXTURE_2D, textId);
wglBindTexImageARB(pbuffer, WGL_FRONT_LEFT_ARB);
```

• **Eliminando lo pbuffers**

Cuando ya no se vayan a seguir utilizando los pbuffers, deben ser eliminados de la memoria para que no ocupen espacio. Esto se consigue liberando el contexto del pbuffer, el contexto de opengl del pbuffer y el pbuffer

```
wglDeleteContext(pbuffer_GL_context);
wglReleasePbufferDCARB(pbuffer, pbuffer_device_context);
wglDestroyPbufferARB(pbuffer);
```

Bien, una vez que se ha comentado cómo funcionan los pbuffers se comprenderá mucho mejor la clase DrawableTexture, ya que básicamente tiene un pbuffer que utiliza para pintar, y, al ser una textura, puede ser utilizada como las texturas normales y corrientes. La diferencia con la clase Texture es:

- Sobrecarga la operación Activate: para no hacer simplemente un glBindTexture, sino llamar también a wglBindTexture
- Define dos funciones para manejar el pintado en la textura:
    - void ActiveDraw(): activa el contexto del pbuffer de modo que todos los comandos que se ejecuten tras esa llamada lo hagan en el pbuffer y no en la ventana
    - void DisableDraw(): activa el contexto de la ventana y, por lo tanto se desactiva el del pbuffer.

De este modo, para pintar en una textura, se llama primero a ActiveDraw(), se pinta y se desactiva con DisableDraw(). La forma de activarla para texturizar es igual que con cualquier textura, se llama a Activate y desde ese momento queda seleccionada.

2.3.1.6.5.- La clase Ballon

Y, por fin, la clase Ballon (bocadillo). Todo lo definido anteriormente se utiliza en esta clase. Así, el bocadillo presenta los siguientes atributos, entre otros:

- boost::shared_ptr< SquareTimerText > square_timer_text: que se utiliza para pintar en el pbuffer el texto que sale en los bocadillos

- boost::shared_ptr< DrawableTexture> pbuffer: es la textura del bocadillo. Se podría decir que el bocadillo es un rectángulo que se texturiza con el pbuffer. No es exactamente así ya que los bocadillos presentan un marco alrededor, para que queden más bonitos. Este marco no tiene textura.

- boost::weak_ptr< Object3d > border: es el marco del que se hablaba antes. Como los bocadillos son Object3D, un border o marco está colocado como hijo del bocadillo

- boost::weak_ptr< MetaObject3d > meta_object: es un puntero al metaobjeto del que cuelga el bocadillo, es decir, al muñeco del chat. Este puntero hace falta para cambiar el estado de las animaciones del personaje

- std::list< std::string > phrases_list: una lista de frases. En el SquareTimerText solo puede haber una, así que, mientras se está mostrando, el resto se apilan aquí.

El comportamiento de un bocadillo está definido también como una máquina de estados. Ya que es la mejor manera de hacer las cosas ordenadamente. Los estados definidos son:

- NOTHING: en este estado el bocadillo ni se pinta. La lista de frases a procesar está vacía y sólo cambia de estado si llega una frase. En ese caso pasa al estado APPEARING

- APPEARING: estado en el que el bocadillo está apareciendo. Este estado quedó sin uso al final, pero se diseñó para hacer alguna animación para que el bocata salga. Por ejemplo podía salir de la cabeza del muñeco e ir agrandándose hasta aparecer

- WRITING: el SquareTimerText está procesando una frase. De este estado se puede salir por 3 razones:

    - Se acaba la frase, y se pasa al estado DISAPPEARING
    - El SquareTimerText detecta fin de linea, se pasa al estado WAITINGEOL
    - El SquareTimerText detecta un emoticono, se le indica al metaobjeto que cambie a la animación del emoticono y se pasa a WAITING_ANIMATION

- WAITING_ANIMATION: en este estado se está esperando que el metaobjeto termine de hacer la animación, cuando acaba se vuelve al estado WRITING

- WAITING_EOL: en este estado se realiza una animación en la matriz de textura del objeto para que se cause la sensación de que el texto asciende. Cuando se asciende lo suficiente para que quepa otra linea se vuelve a WRITING

- DISAPPEARING: cuando acaba de mostrarse el texto de un bocadillo, se espera 5 segundos antes de borrarlo, para que de tiempo a leer lo último que se escribió. Si hay más frases en la lista de frases se pasa al estado APPEARING; si no hay más, se pasa a NOTHING

También merece la pena comentar el hecho de que los bocadillos heredan de la clase BildBoard, de modo que siempre aparecen de frente a la cámara, ya que si no serían bastante incómodos de leer. La clase BildBoard hereda de la clase Object3D, y por lo tanto los bocadillos son también Object3D, por lo que su comportamiento en la clase escena es igual que el de

cualquier otro objeto. Es decir, pueden tener hijos, padres, etc. Sobrecargan la función Move para realizar el comportamiento de la máquina de estados.

Para usar un bocadillo solo hay que añadirle frases y ya se encarga el sólo de ir procesándolas. Es tan sencillo como hacer AddPhrase(const std::string &phr) cuantas veces sea necesario, y los textos irán apareciendo en el bocadillo. Del mismo modo, se puede uno despreocupar de los emoticonos puesto que son ellos mismos los que los gestionan, cambiando las animaciones del muñeco asociado.

2.3.1.7. La clase SpriteChat

Un objeto de la clase SpriteChat es cada uno de los monigotes que se mueven por el mundo. Básicamente son metaobjetos, del tipo de los que se explicaron en la sección de la escena, pero presentan alguna diferencia con estos y añaden cosas nuevas.

Para empezar, una peculiaridad de este metaobjeto es que las animaciones de todos los nodos hijo que tiene por debajo no están coordinadas entre si. Es decir, en un metaobjeto todos los nodos hijos deben estar en el mismo número de animación, pero aquí esto no es así. La razón de esto es porque mientras que la parte inferior del muñeco solo tiene dos animaciones (parado y andando), la parte superior presenta 17 (parado, andando, hablando y los emoticonos).

La cuestión es que cuando un muñeco anda o está parado siga ejecutando el emoticono en curso, pero si no está haciendo ningún emoticono, entonces la animación de la parte superior se hace corresponder con la de andando o parado. Lo mismo ocurre con la animación de hablar, y, así, un muñeco puede ir hablando mientras anda. Si no se hubiera hecho así esto, los muñecos tendrían que pararse para poder hablar y para poder hacer gestos, lo cual es un poco incómodo

Otra peculiaridad frente a la clase metaobjeto, es que en ésta si se utiliza la matriz que tiene definida, para hacer movimientos dentro del mundo (en los metaobjetos normales esta matriz es siempre la identidad). Estos movimientos son de dos tipos:

- Interpolables: para movimientos por red. Se van dando diferentes posiciones y se debe interpolar entre ellas
- Movimiento frame a frame: en cada frame se accede a la matriz del metaobjeto y se modifica

Los movimientos interpolables están pensados para establecer una conexión por red. De este modo, cada cierto tiempo irán llegando posiciones del muñeco y queda bastante brusco el no interpolar entre ellas y pintarlas directamente (se probó primero sin interpolación y el resultado era horrendo). La interpolación definida es para los elementos definidos en la struct Position:

- x, y, z: que definen la posición del muñeco
- angle: que define la rotación en el eje y (nótese que en el muñeco no hay rotaciones en el resto de ejes por lo que la submatriz 3x3 será siempre una matriz de rotación en el eje de las y)

Estos atributos no están definidos directamente en la clase en la clase matriz, pero son fácilmente accesibles. Los tres primeros se sacan directamente de la matriz. El ángulo es más complicado de sacar, pero si se tiene en cuenta que la submatriz 3x3 es siempre de rotación en el eje y, se puede sacar del siguiente modo:

```
if (matrix.Get(0,2) < 0)
{
    pos.angle = (180.0f / (float)M_PI) * (acos(matrix.Get(0,0)));
}
else
{
    pos.angle = (180.0f / (float)M_PI) * (acos(matrix.Get(0,0)));
    pos.angle = 360.0f - pos.angle;
}
```

La posición de la matriz (0,2) es el seno del ángulo y la (0,0) es el coseno. Jugando con esos valores se obtiene lo que se quería.

Lo que se hace cuando llega una nueva Position desde el servidor es colocarla como la posición de destino, para que el SpriteChat vaya interpolando en cada frame hasta que la posición sea igual a la de destino.

La interpolación del ángulo y la de la posición se hacen por separado. A continuación se detallan los algoritmos:

- Para interpolación de ángulos
    - Si la diferencia entre los angulos actual y destino es menor que la cantidad de ángulo que se puede avanzar en el presente frame (velocidad de giro * tiempo transcurrido):
        - Colocar como ángulo el ángulo de destino
    - Si no:
        - Avanzar en el sentido donde haya menos distancia al ángulo destino (horario o antihorario) la cantidad velocidad de giro * tiempo (para

obtener la dirección de giro se hace angulo destino – actual y si es negativo se gira en sentido horario y si no en antihorario)

- Para interpolación de posiciones
  - Si la distancia al punto de destino es menor que lo que se puede avanzar en este frame (velocidad del muñeco * tiempo)
    - Colocar como posición la posición destino
  - Si no:
    - Avanzar en dirección hasta el punto de destino la cantidad velocidad * tiempo (para avanzar en una dirección, se calcula el vector origen-destino, se normaliza y se multiplica por lo que se quiera avanzar)

En cuanto al otro tipo de movimiento. Hay un par de funciones para su implementación, que son:

```
const Matrix& GetMatrix()const;
void SetMatrix(const Matrix &m);
```

Estas dos funciones permiten modifcar la matriz del muñeco de manera ajena a él, por lo que se delega el movimiento a otra clase. Esta clase es la que se ha llamado GlutSpriteController y se verá a continuación.

Por último comentar que los SpriteChat definen el método:

```
void Say(const std::string &str);
```

mediante el cual se le envía una frase al bocadillo del muñeco, y como ya se vio, este bocadillo se encargará de escribir la frase, provocar los gestos para los emoticonos... etc

Hay dos clases que se relacionan muy directamente con SpriteChat: la clase SpriteChatReader, que se encarga de la lectura de sus archivos y la clase GlutSprirteController, que se encarga de manejarlo mediante captura del teclado. Veamos qué hacen exactamente.

2.3.1.7.1. La clase SpriteChatReader

Esta clase hereda directamente de AseParser. Define ya la ruta donde se encuentra el fichero *.ASZ por lo que no tiene parámetros. Es una clase muy pequeña que básicamente hace lo siguiente:

- Lee la malla del objeto del directorio ./chatin/chatin-parado.ASZ
- Añade una cámara al muñeco.
- Crea un objeto de tipo Ballon y lo añade a la jerarquía del objeto creado (añadiendo también su lista de materiales).
- Lee las animaciones de las que va a hacer uso el muñeco y se las añade.

2.3.1.7.2.La clase GlutSpriteController

Esta clase permite manejar el SpriteChat mediante eventos de teclado. Para ello hace uso de la librería glut (de ahí su nombre). Utiliza las funciones definidas en el espacio de nombres llamado KeyListener que básicamente permiten conocer:

- Si una tecla está pulsada o no en un determinado momento
- Un buffer con los últimos caracteres introducidos

La utilidad de ambas funciones es casi obvia. La primera se utiliza para saber si alguna de las teclas de dirección (que son las que controlan al muñeco) se ha pulsado y actuar en

consecuencia. La segunda permite conocer una frase escrita por el usuario y poder mandársela al SpriteChat para que la diga.

Todas estas acciones se realizan en la función Move. Esta función nos devuelve un objeto de tipo std::pair< bool , bool > que indica si, en este frame el muñeco se ha movido (primera componente) o si ha dicho alguna frase (segunda componente). Es también en esta función donde se realiza la detección de colisiones con el mundo.

2.3.2 – EL ESCENARIO

La primera aproximación que se implementó para el escenario fue un escenario simple mediante mapa de alturas. Este tipo de escenario proporcionaba una manera muy simple de hacer la detección de colisiones, como se explica en el siguiente apartado. El inconveniente era que los escenarios eran excesivamente simples y no correspondían con el tipo de escenarios que se había considerado en un principio. Por estas razones se procedió a incluir la parte del diseño inicial correspondiente a la representación del mundo (ver apartados 2.2.2 y 2.2.4) y dar soporte a escenarios arbitrarios de cientos de miles de polígonos.

La implementación de los mapas de alturas se detalla a continuación.

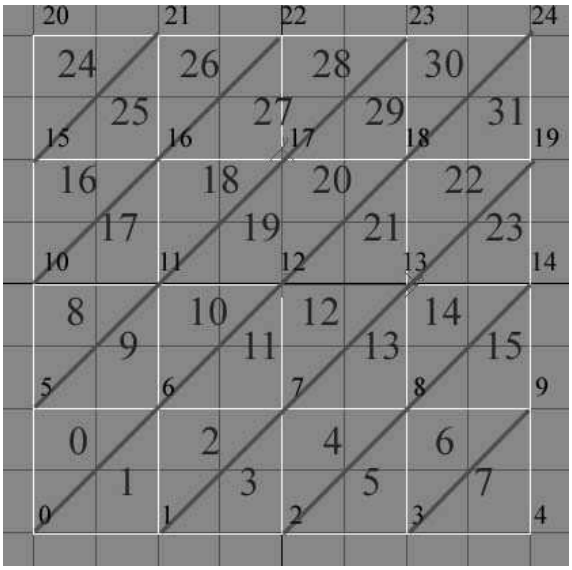
2.3.2.1 – La clase ChatWorld

Una de las técnicas de detección de colisiones con el escenario que se ha implementado es la que presenta la clase ChatWorld. Esta técnica se conoce como mapa de alturas, y permite conocer para cualquier posición (x,z) la y(altura) asociada. A continuación se verá cómo.

La clase ChatWorld hereda directamente de MetaObject3D, por lo tanto, posee todas las ventajas de este tipo de objetos. De hecho su comportamiento a la hora de pintarse es idéntico. La diferencia más notable es que tiene la siguiente función:

```
float GetHeight(float x, float z);
```

Que como se ha dicho permite calcular la altura. Para calcular la altura se debe buscar dentro del metaobjeto al objeto que tiene por nombre "suelo". Este objeto define una malla de dimensión width x height (variables locales del ChatWorld) y presenta div_w y div_h divisiones en horizontal y en vertical respectivamente. Además, debe estar centrado en (0,0). Visto desde arriba es como una cuadrícula, donde la numeración de los vértices y las caras es del siguiente modo:



En este caso en concreto, para una malla de 4 x 4. La ventaja que tiene esta malla es que en todo momento se puede conocer sobre que polígono se encuentra el muñeco en el mapa, es

decir, qué 3 vértices definen el plano sobre el que debería aparecer encima. El algoritmo para la localización de la altura sigue 3 pasos:

1. Localizar el cuadrado sobre el que se encuentra el punto (x,z). Esto se consigue mediante las siguientes líneas de código

```
//localizo los 4 vértices que forman el cuadrado donde entra el punto x,y
    int v1, v2, v3, v4;
    float sq_width = width / div_w;
    float sq_height = height / div_h;
    int x_num = (x  + width/2.0f)/ sq_width;
    int y_num = ((-z) + height/2.0f)/ sq_height;

    v1 = ((div_h + 1) * y_num) + x_num;
    v2 = v1 + 1;
    v3 = v1 + div_w + 1;
    v4 = v3 + 1;
```

El cuadrado se encuentra localizado en los vértices que tienen los números v1, v2, v3 y v4. Llamemos a estos vértices vert1, vert2, vert3 y vert4

2. Una vez localizado el cuadrado hay que localizar el polígono exacto sobre el que se encuentra la x y la z (por polígono nos referimos a un triángulo; y nótese que un cuadrado está formado por 2 triángulos). En un cuadrado, para conocer si nos encontramos por encima de la diagonal que lo atraviesa o por debajo, basta con comprobar si la la coordenada horizontal es superior a la vertical. En la división superior la coordenada vertical es siempre mayor, mientras que en la inferior es al contrario. En la diagonal ambas coordenadas son equivalentes

3. El último paso es calcular la altura dados los 3 vértices que definen el triángulo. Para ello se utiliza la función

```
float ChatWorld::GetYFromPlane(Vertex v1, Vertex v2, Vertex v3, float x, float z)
```

que calcula la ecuación del plano definida por v1, v2 y v3. Para ello, realiza el producto vectorial de v1v2 x v1v3 obteniendo así la normal al plano. Con la normal N(n1, n2, n3) y un punto P(p1, p2, p3) se calcula el valor D de la ecuación del plano del siguiente modo

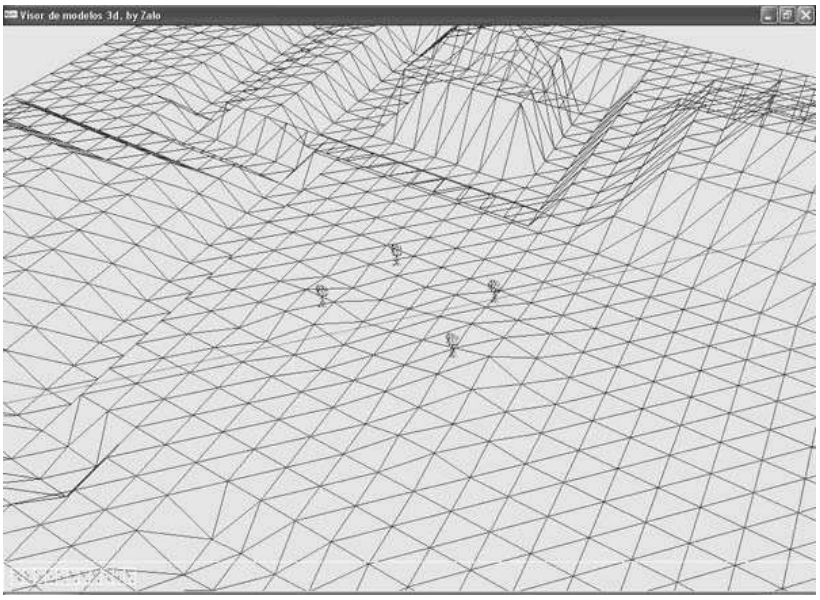
$$n1*p1 + n2*p2 + n3*p3 + D = 0 \quad => \quad D = -(n1*p1 + n2*p2 + n3*p3)$$

De este modo se obtiene la ecuación Ax + By + Cz + D = 0, y sustituyendo x y z queda solo la variable y por despejar

$$Y = (Ax + Cz + D) / -B$$

La ventaja de haber definido todo esto así es que el mapa puede ser perfectamente editado en un programa de diseño 3d como es el 3dstudio Max y luego exportarlo. Una cosa que se debe tener en cuenta cuando se esté editando la malla es que se debe respetar lo que

se ha dicho aquí. Es decir, la malla deber ser de width x height y presentar el número de divisiones adecuado. Y, lo más importante, cuando se edite dicha malla, sólo se debe modificar la componente 'y' de los vértices.



2.3.3 - MÓDULO RED

2.3.3.1 - Introducción

Debido a los cambios en la especificación (Ver sección 1.3), la implementación de la red se adapta a los nuevos requisitos, y no implementa alguna de las funcionalidades en principio pensadas.

El cliente actual de chat:

- Usa un único canal
- No admite privados
- No tiene modo administrador
- Sólo usa un protocolo
- Usa un modelo de movimiento basado en posiciones y no en estados
- No detecta colisiones
- No permite registro permanente de usuario (poder reservar un nick y asignarle contraseña)

Además, esta sección también trata de las tecnologías utilizadas en dicha implementación.

2.3.3.2 Cambios en el protocolo

El protocolo actual se diferencia del especificado en:

- INTRO, INTRAS, INTRAT no mandan mensaje de posición, sino que presuponen (0,0,0) con ángulo 0.
- MOVEO, MOVES, MOVET tienen todos los campos, pero el cliente no usa TRANS, ROT ni TEMPUS.
- El cliente no usa el comando TEMPUS, aunque el paquete se envía al iniciar la conexión (no de forma periódica).
- MOVET en este momento podría simular el movimiento de todos, para no tener que esperar a los paquetes, pero viene mal para el uso del protocolo actual (interpolación lineal podría dar lugar a comportamientos anómalos visualmente al intentar corregir los problemas de lag) y necesitaría la información de TEMPUS, no usada por el cliente.
- MOVES necesitaría el conocimiento del mapa por parte del servidor, por ello no se ha implementado.
- SUSURRO Y SUSURRAT no están implementados, al no haber privados en el cliente.
- INTRO no está implementado, al haber un solo canal en el cliente.
- STABAT es temporalmente sustituido por INTRAT + MOVEO, lo que da un resultado similar en el cliente

2.3.3.3 Cambios en el diseño

Servidor

Las siguientes clases han sido excluidas de la implementación:

- ProtocolChooser: Puesto que sólo tenemos un protocolo, no hace falta comprobar qué protocolo usa el cliente.
- AdministratorProtocol: La figura del administrador ha desaparecido.
- SecureConnectionListener: No escuchamos conexiones mediante sockets SSL.

- StateMachine, NormalStateMachine, AdministratorStateMachine: El protocolo es lo suficientemente simple para no necesitar máquina de estados.
- DataBase: No leemos información adicional de una base de datos.
- Private: No implementamos conversaciones privadas

Las siguientes clases han sido incluidas:

- StringTokenizer: Analiza léxicamente una cadena descomponiéndola en unidades simples como palabras y enteros.
- XMLReader: Utilidad para cargar la configuración(puerto por el que escucha) escrita en un documento jerárquico XML (Ver sección 3.2)

Las siguientes clases han sido modificadas:

- Logger: Fue implementada para mostrar los eventos por consola, distinguiendo tipos de eventos. En el diseño no se especificaba la salida que logger podía tener.
- Chat: La lógica ha sido modificada para crear un canal por defecto e introducir a todos los nuevos usuarios en él.
- User: Listen llama a ListenFar, donde debería de haberse medido la distancia entre el emisor y receptor del mensaje y comprobado si está cerca de él para poder oírlo.
- NormalProtocol: Se adapta a las correcciones al protocolo (Ver 2.3.2)
- Movement: El método refresh de simulación de movimiento en servidor no ha sido implementado. Este método servía para que el servidor pudiera actualizar las posiciones basándose en suposiciones, librando de esta forma a los clientes de realizar los cálculos. Como no usamos el campo TEMPUS del protocolo, los cálculos no son posibles.

Cliente

El cliente, por su simplicidad, sólo muestra cambios en la implementación de la clase NormalProtocol para adaptarse al protocolo modificado (Ver 2.3.2).

Al igual que el servidor, carece de máquina de estados para protocolo.

También lee la configuración de la dirección y el puerto del servidor al que se tiene que conectar de un XML, mediante XMLReader (Ver 3.2).

2.3.3.4 - DETALLES DE IMPLEMENTACIÓN

Esta sección está destinada a explicar las tecnologías usadas en el módulo de red

2.3.3.4.1 - OPENTOP

2.3.3.4.1.1 -Introducción

Para nuestro servidor, necesitábamos una biblioteca de hilos. Cada cliente debía ejecutar un hilo para la escucha, además de los hilos de aceptación de conexión. Además, estos hilos debían proporcionar mecanismos básicos para controlar el acceso a las secciones críticas.

Por otro lado tampoco teníamos una biblioteca de sockets para comunicarnos por red.

Tras la búsqueda de, por un lado, una biblioteca de hilos, y por otro lado una biblioteca de sockets que nos satisficiera para nuestro servidor, optamos por usar opentop[45].

Opentop es una biblioteca con doble licencia. Por un lado es libre(GPL) para aplicaciones no comerciales (libres, académicas o de uso personal no distribuible). Por otro lado, requiere royalties para uso comercial.

Opentop nace bajo la idea de implementar la api java 1.1 para C++ (llamado también java core), que tiene los paquetes básicos lang, io, util, net y awt. Esto no fue del todo posible por la naturaleza de c++, ya que uno tiene que usar macros de preprocesador cuando quiere hacer un sinchronized, o usar smart pointers y un monitor de memoria para simular el recolector de basura.

2.3.3.4.1.2 - Smart Pointers en Opentop

Un smartpointer es una estructura de datos que se usa de forma estática (en contraposición a un puntero), pero que en su interior contiene un puntero.

Cuando un smartpointer es pasado como parámetro, se invoca al constructor copia, pero éste tiene sobrecargado el método para llevar la cuenta del número de ejemplares.

Cada ejemplar decrementa el contador una vez que sale de su scope, por medio del destructor. El que al decrementar deja el contador a 0 es el encargado de eliminar el puntero de la memoria.

Esto hace que no tengamos que preocuparnos de hacer delete a la memoria dinámica.

Opentop tiene esta estructura de datos implementada como una plantilla:

RefPtr<Clase>

Para usarlo, hay que declarar de forma estática (en el sentido de no ser un puntero) un SystemMonitor en la función main, y éste se encarga de la gestión de memoria.

Los smartpointers se comportan como punteros, puesto que tienen -> sobrecargado. Si un smart pointer falla, lanza un assert.

2.3.3.4.1.3 - Hilos en Opentop

Los hilos de opentop se crean por medio de la clase Thread. Las dos estrategias fundamentales para la creación de un hilo son:

- Hacer un objeto que descienda directamente de Thread.
- Asignar al thread un objeto que implemente Runnable.

Los métodos con los que cuentan los hilos de opentop son:

- Thread(): Construye un hilo (usado para clases que heredan de Thread)
- Thread(Runnable): Construye un hilo asociado a un objeto runnable.
- Start() : Comienza a ejecutar el hilo en el método run del objeto que hereda del hilo o del runnable, dependiendo de cual de las opciones se ha elegido.
- Yield() : Cede el control al planificador de hilos, para que éste replanifique.
- Sleep(milisegundos): Duerme el hilo (detiene su ejecución) durante un número de milisegundos. Mientras tanto, el planificador cede el control a los otros.
- Join(): Espera juntarse con otro hilo.

2.3.3.4.1.4 - Monitores en Opentop

Un monitor es una clase de la que queremos que en cada instancia sólo pueda haber un hilo ejecutando a la vez. Estas clases son útiles para proteger la integridad de los datos que contienen o a los que acceden.

Para realizar un Monitor en Opentop, la clase que queremos que sea monitor debe extender a la clase Monitor, y en cada método hay que poner una directiva para protegerlo.

OT_SYNCHRONIZED

Que equivale a los métodos synchronized de java.

Un ejemplo de uso es:

```
class ActiveCounter : public Monitor {
    public:
        ActiveCounter() : m_count(0) {}

        void updateBy(long diff) {
            OT_SYNCHRONIZED  // creates a locked scope
            m_count+=diff;
            notifyAll();      // notifies all waiting threads
        }

        long getValue() const {
            OT_SYNCHRONIZED  // creates a locked scope
            return m_count;
        }

    private:
        long m_count;
    };
```

La clase ThreadSafeQueue, que se encarga de almacenar los comandos, es un monitor. De esta forma, preservamos a la estructura de datos de las incoherencias que pueden causar varios hilos que ejecutan el mismo código.

2.3.3.4.1.5 - Sockets en Opentop

Un socket es una conexión por red a un puerto.

Podemos distinguir en una conexión dos tipos de socket:
- Serversocket: acepta conexiones
- Socket: Tiene un flujo de entrada y salida para enviar y recibir datos.

El servidor declara un serversocket y pone un hilo para las aceptaciones de éste. Aceptar una conexión es un método bloqueante, y dicho hilo queda parado hasta que llega una conexión entrante. En ese momento el hilo continúa y la llamada a accept devuelve un RefPtr al socket que nos comunica al cliente que se ha conectado.

Por el lado del cliente, éste sólo tiene que declarar un socket e intentar la conexión:

Métodos importantes de serversocket:

- ServerSocket(puerto): Se pone a recibir conexiones por el puerto indicado.
- Accept(): Espera a una conexión entrante (llamada bloqueante) y devuelve el socket para comunicarnos con dicha conexión.

Métodos importantes de socket:

- Socket(servidor, puerto): Intenta conectarse a una máquina por el puerto indicado.
- GetInputStream(): Devuelve un RefPtr al flujo de entrada.
- GetOutputStream(): Devuelve un RefPtr al flujo de salida.
- Close(): Cierra la conexión.

Normalmente el flujo de entrada y el flujo de salida se suelen encapsular en Readers y Writers provistos de buffer, ya que los streams de por sí nos ofrecen sólo la posibilidad de trabajar a bajo nivel (buffers de bytes y número de bytes recibidos).

2.3.3.4.1.6 – Otras cosas de Opentop

Como hemos dicho, usamos otras clases de opentop::io para recubrir los streams(flujos). De esta forma, podemos usar los flujos básicos de entrada y salida de datos del socket como flujos más sofisticados. Esto nos permite trabajar con cadenas, enteros y otros tipos de datos en vez de tener que utilizar arrays de bytes.

También usamos en el logger del servidor (una clase para registrar los eventos de servidor, que podría ser implementada para escribir en fichero de texto, en xml, en impresora, mandar por red... pero que usamos para mostrar por consola) varias cosas:

- Flujos de salida
- Consola ofrecida por opentop
- Conversión de tipos (entero <-> cadena)

Podríamos haber usado opentop además para todos los smartpointers, y para la lectura de xml(Ver 3.2), pero decidimos optar por otras tecnologías más específicas.

2.3.3.4.2 - SOCKET NO BLOQUEANTE IMPLEMENTADO CON WINSOCK

2.3.3.4.2.1 – Introducción

Dado que no queríamos usar opentop en el cliente, ya que utiliza sus propios smart pointers y ante el deseo de que el cliente fuera monohilo (el debug multihilo es muchísimo más costoso), decidimos crear nuestro propio socket.

¿Por qué no cogerlo de una biblioteca? Porque las bibliotecas no suelen implementar llamadas no bloqueantes. Estas son de bajo nivel, y por lo tanto nosotros tuvimos que usar una biblioteca de bajo nivel para implementar el nuestro.

Puesto que el cliente se ejecuta en Windows, tomamos la biblioteca standard de Microsoft winsock[46], que hace llamadas directas a la API del sistema operativo.

Winsock fue la implementación de bajo nivel que Microsoft hizo de los sockets para los programadores de Windows, y que está integrada en el sistema operativo. Los sockets Winsock están basados en los sockets de la distribución Unix de Berkeley BSD.

2.3.3.4.2.2 – ¿Qué es un socket?

Un socket es un punto terminal de una conexión. Normalmente los sockets intercambian datos dentro de un mismo dominio, aun cuando existan varios. Cada socket en uso tiene un tipo determinado y un proceso asociado.

Normalmente, el usuario dispone de dos tipos de sockets, que determina el protocolo que se va a usar:

- el *stream socket* que garantiza un flujo de datos bidireccional, seguro, secuenciado y sin duplicados, carente de límites de registro
- el *datagram socket* que también es bidireccional, pero no promete ser secuenciado,seguro o sin duplicidades.

En nuestro chat, usamos un stream socket. Este socket, que está en el módulo de red de cada cliente de red, se conecta con el serversocket de opentop que tenemos en el servidor. El serversocket del servidor, al recibir una conexión, genera un socket en el servidor. El nuevo socket en el servidor y el socket de cliente comparten una misma conexión, y tienen dos canales (implementando full duplex) de forma que en cada uno un extremo manda y el otro escucha.

2.3.3.4.2.3 – Funciones más importantes de Winsock

Las funciones más importantes son:

- WSAStartup: Se ocupa de la inicialización de winsock para nuestra aplicación. Hay que comunicarle que versión de la API vamos a usar. Ésta puede ser 1.1 o 2.0.

- gethostbyaddr: Localiza una máquina si le damos su ip de la forma "xxx.xxx.xxx.xxx", y la almacena en una estructura de datos manejable y que precisaremos en otras llamadas a winsock.

- WSAGetLastError: Devuelve el código del último error que se ha producido al hacer una petición a winsock.

- socket: Devuelve un handler de socket. En esta función podemos indicarle si usar TCP o UDP y otras cosas relacionadas con la comunicación en la capa de transporte.

- connect: Hace que el socket se conecte a una máquina dada por un puerto dado.

- ioctlsocket: Nos permite definir propiedades sobre un socket. En nuestro caso la usamos para decir que receive sea no bloqueante.

- send: Manda un buffer de caracteres a través del socket

- receive: recibe un buffer de caracteres a través del socket

2.3.3.4.2.4 – Otras funciones de Winsock

Prototipo	Definición
socket ()	Crea una terminación para una comunicación y retorna un descriptor de socket.
accept ()	Conexión reconocida y asociada con socket recién creado.
bind ()	Asigna un socket a una dirección de red determinada.
closesocket ()	Elimina un descriptor de socket de la tabla de referencia de objetos por proceso.
Conté ()	Inicia la conexión del socket especificado.
Getpeername ()	Recibe el nombre del par conectado al descriptor de socket especificado.
Getsockname ()	Recibe el nombre actual de un socket.
Getsockopt ()	Recibe las opciones asociadas a un descriptor de un socket.
htonl ()	Convierte una cantidad de 32 bits del orden de los bytes del anfitrión al orden de bytes de la red.
htons ()	Convierte una cantidad de 16 bits del orden de los bytes del anfitrión al orden de bytes de la red.
inet_addr ()	Convierte una cadena de caracteres representando un número en la notación estándar de Internet a un valor de dirección Internet.
inet_ntoa ()	Convierte un valor de dirección Internet a una cadena ASCII en notación "." .
Ioctlsocket ()	Proporciona un control para los descriptores.
listen ()	Indica que el socket esta disponible para recibir peticiones, crea una cola conexiones.
recv ()	Recibe datos de un socket conectado.
recvfrom ()	Recibe datos de un socket conectado o desconectado.
select ()	Multiplexado E/S zirconio.
send ()	Envia datos a un socket conectado.
Sendto ()	Envía datos a un socket conectado o desconectado.
Setsockopt ()	Almacena las opciones asociadas con descriptor de un socket especificado.
Shutdown ()	Finaliza parte de una conexión full-duplex.

2.3.3.4.2.5 – ¿Qué es el bloqueo?

Como bloqueo hemos de entender el hecho de que el ordenador se mantenga a la espera de un determinado proceso sin que pueda realizar ninguna otra tarea mientras tanto.

En un sistema multitarea o multithead (Windows 95 ó NT) no existe ningún problema al respecto, ya que la CPU es la encargada de asignar los tiempos de ejecución. Sin embargo, en

Windows 3.x es sumamente fácil que el sistema se quede bloqueado esperando la ejecución de un *recv()*.

La solución elaborada y definitiva consiste en crear dos modos de trabajo: con bloqueo permitido y sin bloqueo. La especificación Winsock 1.x especifica exactamente lo que debe ocurrir cuando se entra en un bloqueo: mientras se espera que se realize la petición, Windows llama constantemente a una función que, en particular, se encarga de procesar los mensajes de Windows.

El usuario puede redireccionar las llamadas a una función propia utilizando **WSASetBlockingHook**. Mientras se está en modo de espera, cualquier intento de iniciar otra operación de red, aunque sea usando un socket diferente, causa un error **WSAEinProgress**, lo que significa que hay una operación en progreso

Por defecto cualquier *socket* creado con la función *Socket* opera en modo de bloqueo, aunque mediante funciones como **WSAAsyncSelect** se puede cambiar el modo de operación de forma que no bloquee.

En esta situación, cualquier operación sobre ese *socket* devuelve inmediatamente el control al programa, bien con el resultado esperado o bien con un código de error. Este último, no obstante, suele ser **WSAEWouldBlock**, que significa literalmente que para completar la petición se ha tenido que esperar, y por tanto entrar en modo de bloqueo.

En estos casos la operación debe reintentarse hasta que se consiga el resultado satisfactorio o se obtenga un error de tipo "definitivo" (servidor no encontrado, conexión denegada,etc).

Cuando se trabaja de esta manera, se suele decir que se está trabajando en modo síncrono, ya que el programa pide una operación y comprueba continuamente el estado de esa operación hasta que termina.

A pesar de que los modos síncrono y de bloqueo pueden ser más o menos convenientes en aplicaciones sencillas, la forma habitual de trabajar es en modo asíncrono. En este modo, el programa realiza la petición y se le devuelve el control inmediatamente. Cuando la operación se completa u ocurre algún evento especial, el propio Windows se lo notifica al programa mediante un mensaje.

A través de la función *WSAAsyncSelect* la aplicación indica a Windows qué eventos desea que se le comuniquen, así como el número de mensaje que quiere que se genere.

En nuestro wrapper nosotros hemos indicado a winsock que nuestro bloque no es bloqueante por medio de las opciones del socket, con ioctlsocket(), al que indicamos como opcion de entrada/salida que no bloquee en la recepción. Send entonces devolverá el número de bytes que ha podido leer hasta el momento y nos lo dejará en un buffer. Nosotros iremos recopilando la informaración de ese buffer en cada pasada, de forma que al no haber espera para recepción, la aplicación puede seguir pintando y actualizandose. Seremos nosotros los que decidamos que un paquete se ha acabado al leer un final de linea, devolviendo el buffer acumulado en cada pasada.

2.3.3.4.2.6 – Nuestro Wrapper

Para aislar la tecnología y darle un uso orientado a objetos, decidimos realizar una clase que inicializa y libera winsock de forma transparente y usa la biblioteca de bajo nivel ocultándola tras llamadas de alto nivel.

Nuestra clase recubridora de socket tiene la siguiente interfaz:

- Socket() : La constructora, crea un socket desconectado. Si no hay más ejemplares de sockets, es decir, si es el primero, inicializa winsock.
- bool Connect(const std::string& server, int port): Se conecta a una máquina cuya dirección se le pasa en el parámetro server con formato "xxx.xxx.xxx.xxx", y por el puerto indicado en el parámetro port.
- bool Disconnect(): Desconecta el socket de la máquina remota.
- bool Send(std::string message): Manda una cadena a través del socket.
- const std::string Receive(): Recibe una cadena de forma no bloqueante. Si los caracteres recibidos no terminan en un delimitador de cadena, la función devuelve "" y guarda los caracteres para que en las llamadas siguientes vaya formándose hasta que esté completa para ser devuelta. Sólo devuelve cadenas de una en una.
- ~Socket(): Destruye el socket, y finaliza winsock si era el único socket que quedaba.
- const std::string& GetLastError(): Todas las funciones anteriores devuelven un bool. Para saber qué error ha ocurrido, cuando alguna nos devuelve false llamaremos a esta función que dará un mensaje descriptivo.

2.3.3.4.2.7 – Ejemplo de uso

```
// ConsoleClient.cpp : Telnet simple para probar con netcat.
//

#include "stdafx.h"
#include "Socket.h"

int _tmain(int argc, _TCHAR* argv[])
{
	Socket my_socket;
	std::string message_out;
	std::string message_in;
	my_socket.Connect("127.0.0.1",1981);
	std::cout << my_socket.GetLastError();
	while (true){
		std::cin >> message_out;
		my_socket.Send(message_out);
		message_in = my_socket.Receive();
		if (message_in != ""){
			std::cout << message_in;
			std::cout.flush();
		}
	}
	my_socket.Disconnect();
	return 0;
}
```

3 – APÉNDICES

3.1 - ESTILO DE CÓDIGO

Esta sección describe el estilo que se usará en el código de las cabeceras, y en la medida de lo posible de las implementaciones, para que los nombres de los interfaces y los métodos públicos de éstos sean lo más predecibles posibles y la aplicación en conjunto sea consistente.

La siguiente guía de estilo pretende dotar de la mayor legibilidad posible al código de la aplicación, sacrificando la concisión que es posible obtener en C++ por un nombrado de variables y métodos más descriptivo, un sangrado generoso y una disposición de los bloques más espaciada que permita distinguir claramente las distintas unidades funcionales.

Se empezará por fijar una forma estándar de nombrado de ficheros:

3.1.1 - Nombres de ficheros

La extensión usada para los ficheros de cabecera será `.h`, mientras que la de los ficheros de implementación será `.cpp`

Los nombres de los ficheros coincidirán en nombre y estilo con el del componente que declaran o implementan (p.ej. `ISoundSystem.h`, `XmlParser.cpp`). Serán lo más descriptivos posibles, sobrepasando los 8 caracteres si es necesario. El nombre sin extensión de una cabecera coincidirá con el de su implementación, si la hubiera (p.ej. `XmlParser.h` y `XmlParser.cpp`).

3.1.2 - Árbol de directorios

En el directorio raíz de la aplicación sólo se encontrarán los ficheros de proyecto y otros ficheros como el de licencia, documentación de última hora, etc.

Bajo el directorio raíz habrá un subdirectorio para cada módulo de la aplicación (en este caso *módulo* se refiere a una unidad que es compilada por separado, como por ejemplo un ejecutable o una biblioteca). Bajo el directorio de cada módulo habrá tres subdirectorios:

- `\src` – Contendrá los ficheros `.cpp`
- `\include` – Contendrá los ficheros `.h`
- `\doc` – Contendrá la documentación asociada al módulo

Las cabeceras comunes a todos los módulos se encontrarán bajo el directorio raíz, en el subdirectorio `\include`.

3.1.3 - Tabulaciones

Las tabulaciones serán de 4 espacios, y se sustituirán por blancos, de tal manera que la disposición del código no varíe en plataformas con distinto tamaño del carácter de tabulación.

```
int main(int argc, char **argv)
{
    ...

    return 0;
}
```

3.1.4 - Ficheros de cabecera

3.1.4.1 - Los ficheros `.h` tendrán, como primera línea de código, la directiva de preprocesador `#pragma once`, para evitar inclusiones múltiples.

```
// ejemplo.h – Cabecera de ejemplo

#pragma once

...
definiciones
...

/* EOF */
```

3.1.4.2 - No se usará la sentencia `using namespace` en un fichero de cabecera, ya que tendría efecto no sólo en esa cabecera sino en todos los ficheros donde ésta se incluya. Si es necesario referirse a un identificador que se encuentra en otro espacio de nombres tendrá que especificarse el nombre completo del identificador:

```
// ejemplo2.h - Cabecera de ejemplo II

#pragma once

#include <vector>
#include <string>

class Clase
{
public:
    void AddName(const std::string & name);
private:
    std::vector<std::string> m_Names;
};

/* EOF */
```

3.1.4.3 - La primera línea de código de todo fichero `nombre.cpp` será `#include "nombre.h"`, donde `nombre.h` es el fichero de cabecera asociado.

3.1.5 - Comentarios

Aparte de los comentarios de línea (`// comentario`) y de bloque (`/* comentario */`), se usarán comentarios al estilo JavaDoc para generar la documentación.

```
// Comentario de línea
/*
...
Comentario de bloque
...
*/
```

```
/**
Comentario estilo JavaDoc.

@param x Descripción de parámetro
@return Descripción de valor de retorno
*/
int f(int x)
{
    ...
}
```

3.1.6 - Bloques

3.1.6.1 - Las llaves que delimitan un bloque estarán alineadas, a la altura de la cabecera del bloque, y se encontrarán en una línea aparte:

```
int main(int argc, char **argv)
{
    if (argc > 1)
    {
        // Es conveniente definir las variables de bucle
        // en el sitio.
        for (int i = 1; i < argc; ++i)
        {
            parse_arg(argv[i]);
        }
    }
    else if (argc == 1)
    {
        show_help();
    }
    else
    {
        // No puede darse
    }

    while (condition)
    {
        int x = ...;

        switch (x)
        {
        case 0: ...;
            break;
        case 1: ...;
            break;
        default: ...;
        }
    }

    ret = 1;

    // Condición complicada
    if
    (
        argc == 1 ||
        (argc > 1 && argv != 0) ||
        (argc > 1 && argv == 0 && getenv("PATH") != 0)
    )
```

```
    {
        ret = 0;
    }

    return ret;
}
```

3.1.6.2 - Los cuerpos de las sentencias condicionales estarán siempre delimitados por llaves, aunque sólo contengan una instrucción; de esta manera se evita que el desarrollador olvide añadir las llaves al introducir una nueva instrucción. La única excepción permitida será cuando la sentencia condicional completa, incluyendo la instrucción que contiene, sea corta y se pueda escribir en una sola línea sin sacrificar la claridad del código:

```
// OK - Sentencia corta en misma línea
if (x > 0 && y > 0) ++i;

// No - Sentencia larga en misma línea
if (x > 0 && x < n && y > 0 && y < n) std::cout << "[X = " << x <<
", Y = " << y << "]" << std::endl;

// No - Sentencia larga en distinta línea pero sin llaves
if (x > 0 && y > 0)
    std::cout << "[X = " << x << ", Y = " << y << "]" << std::endl;

// No - Condición complicada e instrucción en misma línea
if
(
    x > 0 && x < n &&
    y > 0 && y < n &&
    x + y == 2 * n
) std::cout << x << y << std::endl;

// OK - Sentencia larga en distinta línea con llaves
if (x > 0 && x < n && y > 0 && y < n)
{
    std::cout << "[X = " << x << ", Y = " << y << "]" << std::endl;
}

// OK - Condición complicada e instrucción entre llaves
if
(
    x > 0 && x < n &&
    y > 0 && y < n &&
    x + y == 2 * n
)
{
    std::cout << x << y << std::endl;
}
```

3.1.7 - Uso de espacios

3.1.7.1 - En las llamadas a funciones no habrá espacio entre los paréntesis y los argumentos, pero sí habrá un espacio tras una coma. Si el número de argumentos de la llamada es elevado se separará cada argumento en una línea, de tal manera que se mejore la legibilidad y se puedan añadir comentarios explicatorios para cada argumento, si es necesario:

```
int m = std::max(a, b)

HWND handle = ::CreateWindowEx(
        0,
        "WindowClass",
        "Test Window",
        WS_OVERLAPPED,
        CW_USEDEFAULT,    // Posición por defecto
        CW_USEDEFAULT,    // Posición por defecto
        400,
        600,
        0,
        hmenu,
        ::GetModuleHandle(0),
        0);
```

3.1.7.2 - Se dejará un espacio entre los operadores artiméticos y lógicos binarios, pero ningún espacio entre un operador unario y su argumento:

```
if (!b && c == i + 2)
{
    ...
}
```

3.1.7.3 - Los identificadores de acceso de las clases estarán alineados con el bloque de la clase:

```
class Window
{
public:
    ...
protected:
    ...
private:
    ...
};
```

3.1.8 - Nombres

3.1.8.1 - Los nombres de las variables y funciones libres estarán en minúsculas, y usarán el guión bajo como separador de palabras:

```
int f(int a, int b);
void show_help();

float value;
std::string server_name;
std::vector active_connections;
```

No se usará la notación húngara[13] para decorar los nombres de variables con su tipo. Hay varias razones por las que no es recomendable usar esta notación, pero la más importante es que si se cambia el tipo de la variable habría que cambiar su identificador:

```
int iAge;        // No
char * lpszName; // No
float fValue;    // No
```

[13] http://en.wikipedia.org/wiki/Hungarian_notation

3.1.8.2 - En cuanto a los nombres de clases y de métodos, cada una de las palabras que los componen empezarán por mayúscula, incluyendo la primera. No se antepondrá C u otro prefijo al nombre de las clases, a no ser que definan un interfaz puro, en cuyo caso tendrán el prefijo I (p.ej: `ISoundSystem`).

Los nombres de las clases deberían ser sustantivos, mientras que los de los métodos deberían ser verbos.

```
class IServer
{
public:
    virtual void Start() = 0;
};

class Server : public IServer
{
public:
    virtual void Start();
    Client * AcceptConnection();
};

class MainClientWindow : public Window
{
public:
    void Show();
    void Hide();
    bool SetTitle(const std::string & new_title);
};
```

Si algún acrónimo forma parte del identificador de clase se pondrá su primera letra en mayúscula y el resto en minúsculas:

```
class IrcClient {};
class HttpRequest {};
class XmlParser {};
```

3.1.8.3 - Los nombres de los `typedef` tendrán el mismo formato que el de las clases pero tendrán el sufijo `Type`:

```
typedef unsigned int DwordType;
typedef std::vector<int> IntVectorType;
typedef std::map<int, std::string> IntStrMapType;
```

3.1.8.4 - Los nombres de constantes estarán en mayúsculas y con guiones bajos como separadores. No se usará `#define` para definir las constantes. Se usará `enum` para las constantes numéricas y static const para las constantes no numéricas.

```
enum { RED_COLOR = 1, BLUE_COLOR = 2, GREEN_COLOR = 3 };
static const char * DEFAULT_NAME = "Test";
```

3.1.8.5 - Para denotar un puntero nulo se usará la constante `0` en vez de `NULL`. La razón es que `NULL` se implementa como una macro del preprocesador, y es necesario incluir la cabecera `<stddef.h>` para poder usarla.

```
Server * pserver = GetCurrentServer();
if (pserver != 0) { ... }
```

3.1.8.6 - Los nombres de los atributos de una clase tendrán el prefijo `m_` y estarán seguidos de su identificador con el mismo estilo que los nombres de clases y de métodos. El prefijo es importante, ya que permite distinguir claramente cuándo se está accediendo a un atributo de la clase, en contraste con acceder a una variable local o un parámetro, y además evita tener que pensar nombres alternativos para los parámetros de los constructores y de los mutadores:

```
class Entity
{
private:
    std::string m_Name;
    std::map<int, Entity> m_SubEntities;
    unsigned int m_IsDynamicEntity;
};
```

3.1.8.7 - Los nombres de las variables globales tendrán el prefijo `g_`:

```
int g_Counter;
Server g_CurrentServer;
```

3.1.9 - Clases

3.1.9.1 - En la definición de las clases se separarán los métodos en grupos (constructores, accesores, etc). También se separarán claramente los métodos de los demás miembros de la clase (atributos, `typedef`, etc), aunque sea necesario reiterar el identificador de acceso:

```
class Server : public ServerBase
{
public:
    // Métodos públicos

    // Constructores
    Server();
    Server(const Server & rhs);
    // Accesores
    Address GetAddress() const;
    void SetAddress(const Address & addr);

    // Destructor
    virtual ~Server();
public:
    // Otros miembros públicos
    typedef std::string NameType;
    enum { ENABLED, DISABLED };
protected:
    // Métodos protegidos
private:
    // Métodos privados
    Client AcceptConnection();
    void CloseConnection(const Client & client);
private:
    // Otros miembros privados
    std::vector<Client> connected_clients;
    NameType name;

    enum { X, Y, Z };
};
```

3.1.9.2 - No se definirán métodos en la declaración de la clase a no ser que se quiera expresamente que sean `inline`:

```
class Client
{
public:
    // No
    std::string GetName()
    {
        return name;
    }

    // Sí
    void SetName(const std::string & name);
private:
    std::string name;
};
```

3.1.9.3 - Si una clase tiene métodos virtuales, su destructor será obligatoriamente virtual. Si no fuera así, el uso de `delete` sobre un puntero a una clase base de ésta daría lugar a comportamiento indefinido (es decir, es un error de programación):

```
class IClient
{
public:
    virtual ~IClient();
};

class Client1 : public IClient
{
public:
    virtual ~Client1();
};

class Client2 : public IClient
{
public:
    ~Client2();
};

IClient * c;
Client1 * c1 = new Client1();
Client2 * c2 = new Client2();

delete c1; // Ok
delete c2; // Ok

c = new Client1();
delete c;  // Ok

c = new Client2();
delete c;  // Error
```

3.1.10 - Casting

Para la coerción de tipos (casting) se usarán los operadores de casting de C++ en lugar del operador de casting de C. De esta manera se evitan muchos de los problemas que conlleva este último, como son por ejemplo las conversiones no seguras o la dificultad de encontrar los lugares donde se utiliza con una herramienta tipo `grep`:

```
int * p = (int *)malloc(sizeof(int)); // No
int * q = new int();          // Ok

int i;
float f;
A * a;
const A * ca;
B * b;
D1 * d1 = new D1();
D2 * d2 = new D2();

// static_cast - Usado para hacer una conversión que sólo necesite
// comprobación estática de tipos (p.ej: conversión entre dos tipos
// integrales, hacia arriba en una jerarquía de clases, etc)
i = static_cast<int>(f);
B * b = static_cast<B *>(d1);

// dynamic_cast - Usado para hacer una conversión hacia abajo en
// una jerarquía de clases (puede fallar).
d1 = dynamic_cast<D1 *>(b);

// reinterpret_cast - Usado para hacer conversiones entre dos tipos
// sin relación alguna. Es el más parecido al operador clásico de C
d2 = reinterpret_cast<D2 *>(d1);

// const_cast - Usado para quitar o añadir los cualificadores const
// y volatile
a = const_cast<A *>(ca);
```

3.1.11 - Parámetros de funciones y métodos

Siempre que sea posible los parámetros de una función o método serán de paso por referencia constante (`const &`), excepto cuando deban ser punteros o tipos básicos. De este modo se evita el paso por valor y la copia potencialmente costosa que esto conlleva.

```
class Client
{
public:
    void SetName(const std::string & new_name);
    void AddServer(int id, Server * server)
};
```

3.2 - XML Y XERCES

3.2.1 - HISTORIA DE XML

XML(eXtensible Markup Language)[54] es un lenguaje de marcas, creado en 1998. Aunque a priori pueda parecer una tecnología inmadura, es una tecnología probada y robusta ya que está basada en SGML, cuyas implementaciones datan de los primeros 80.

El lenguaje de marcas más extendido es HTML. Se podría decir que HTML es un subconjunto de XML, ya que en HTML las marcas estás predefinidas y asociadas a una semántica determinada.

Esto es la mayor virtud y debilidad de XML. En XML las marcas no estás predefinidas, sino que son inventadas por el usuario. Ello nos permite una flexibilidad incomparable para expresar cualquier realidad como datos estructurados, siendo este su gran punto a favor. Por contra... dicha flexibilidad hace que a la hora de leer los documentos, el lector deba conocer a priori la estructura de los datos y su semántica, para poder darle un significado correcto.

Para atajar esto, en los últimos tiempos se han establecido ciertos protocolos que se implementan sobre XML para poder tener semánticas estándares para intercambio de información de determinados documentos, como SOAP[47].

En definitiva, XML es un metalenguaje, es decir, un descriptor de lenguajes.

3.2.2 - ETIQUETAS EN XML

Como ya hemos mencionado, XML es un lenguaje de etiquetas.

3.2.2.1 - Etiqueta de declaración

Un documento XML comienza con la etiqueta como primera línea:

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

Esta etiqueta indica:

- Que es un documento XML
- La versión del estándar XML que cumple. En este punto no hay mucho problema por el momento, porque sólo existe la 0.1. De todas formas, este campo es obligatorio ponerlo.
- El alfabeto de caracteres en el que está escrito (encoding). Se puede poner cualquiera, y depende del parser el entender o no la codificación. Por defecto es UTF-8, aunque podrían ponerse otras, como UTF-16, US-ASCII, ISO-8859-1, etc. No es obligatorio salvo que sea un documento externo a otro principal.
- Si es o no dependiente de un DTD (standalone="yes" si no depende de un DTD). En principio no hay porqué ponerlo, porque luego se indica el DTD si se necesita.

Un DTD (Document Type Definition) es un documento que sirve para establecer ciertas reglas sobre documentos XML. De esta forma podemos crear un lenguaje o protocolo en XML y plasmarlo en forma de reglas en un DTD para después poder validar que el XML está bien formado en el lenguaje que hemos descrito.

Nosotros no vamos a usar DTD.

3.2.2.2 - Etiquetas de comentario

Después de la etiqueta <?xml ...> el resto de etiquetas siguientes son definidas por el usuario, excepto las etiquetas de comentario, de la forma:

<?--Aquí viene el comentario -->

Estas etiquetas son ignoradas al parsear.

3.2.2.3 - Etiquetas de usuario

El poder de XML reside como hemos dicho en que el usuario puede definir las propias etiquetas del lenguaje. Para ello debemos tener en cuenta que:

- Los documentos XML son case sensitive, esto es, en ellos se diferencia las mayúsculas de las minúsculas. Por ello <ATRIBUTO> sería una etiqueta diferente a <atributo>.
- Además todos los espacios y retornos de carro se tienen en cuenta (dentro de las etiquetas, en los elementos).
- Hay algunos caracteres especiales reservados, que forman parte de la sintáxis de XML: <, >, &, " y '. En su lugar cuando queramos representarlos deberemos usar las entidades <, >, &, " y ' respectivamente.
- Los valores de los atributos de todas las etiquetas deben ir siempre entrecomillados. Son válidas las dobles comillas (") y la comilla simple (').
- Todas las etiquetas deben estar metidas dentro de una, a la que se le llama raíz.

Las etiquetas de usuario sirven para contener elementos. Es decir, una etiqueta es como un contenedor de un espacio de nombres. Según se anidan etiquetas, éstas van formando un nombre completo para cada rama del árbol que su estructura forma. Las etiquetas no sólo pueden contener nodos, sino información en forma de cadena. Es a ésta información a la que se le llama elemento.

```
<etiquetaA>
        <etiquetaB>
                Información
        </etiquetaB>
</etiquetaA>
```

En este caso, etiquetaA y etiquetaB son etiquetas. 'Información' es en cambio un elemento, al que uno se puede referir como etiquetaA.etiquetaB (nombre completo).

Hay dos tipos de elementos: los vacíos y los no vacíos. Hay varias consideraciones importantes a tener en cuenta al respecto:

- Toda etiqueta no vacía debe tener una etiqueta de cerrado: <etiqueta> debe estar seguida de </etiqueta>. Esto se hace para evitar la aberración (en el buen sentido de la palabra) a la que habían llegado todos los navegadores HTML de permitir que las etiquetas no se cerraran, lo que deja los elementos sujetos a posibles errores de interpretación.
- Todos los elementos deben estar perfectamente anidados: no es válido poner:

```
<atributo><nombre>Puerto</atributo></nombre>
```

y sí lo es sin embargo:

```
<atributo><nombre>Puerto</nombre></atributo>
```

- Los elementos vacios son aquellos que no tienen contenido dentro del documento. Un ejemplo en HTML son las imágenes. La sintáxis correcta para estos elementos implica que la etiqueta tenga siempre esta forma: <etiqueta/>.

Por último, cabe reseñar que las etiquetas de usuario pueden tener atributos definidos por éste:

<mascota animal="perro" años="8">Blacky</mascota>

Como vemos, XML permite una libertad absoluta a la hora de definir datos. Es el usuario el que escoge la forma de estructurarlos y el que tiene que tener en mente su organización y semántica a la hora de parsearlo.

3.2.3 - COMPLEMENTOS A XML: DTD Y XSL

3.2.3.1 - DTD:

Un DTD (Document Type Definition) es una definición de los elementos que puede incluir un documento XML, de la forma en que deben hacerlo (qué elementos van dentro de otros), y los atributos que se les puede dar. Es en definitiva un sustituto más simple de una sintaxis BNF que permite decir si el documento pertenece o no al tipo de documentos que queremos parsear.

3.2.3.2 - XSL:

Un XSL (eXtensible Stylesheet Language) se puede decir que es una vista de interfaz gráfica de XML. Un XSL permite a un navegador mostrar un XML de forma amigable, como si fuera una elaborada página web con CSS (Cascade Style Sheets).

De esta forma, XML implementa en cierta forma el patrón modelo-vista-controlador, separando cada aspecto en un documento distinto.

3.2.4 - ANALIZANDO UN XML: XERCES

Una vez hemos construido el XML, necesitamos un analizador para que nuestros programas puedan tomar la información que el documento contiene.

Dado que la sintaxis básica de XML es fija, construir analizadores manuales de XML es perder el tiempo. Existen analizadores genéricos que nos permite continuar con lo importante del código y no dedicarnos a la parte engorrosa, tediosa y fuente de posibles fallos que es el análisis sintáctico.

Hay varios paquetes disponibles para ello. Nosotros usaremos xerces[48], del proyecto Apache (Yakarta). Lo hemos elegido por su carácter libre (Apache License) y por su respeto con los estándares.

Aún así, todos los paquetes se parecen mucho, ya que implementan 2 API estándares para parsear el documento:

- DOM: Document Object Model (DOM) toma el XML y genera un árbol en memoria con toda la información. Este árbol se puede recorrer, modificar, hacer búsquedas sobre él, y todo con una API bien definida. El problema que tiene DOM es que el árbol permanece entero en memoria, cosa que muchas veces no nos hace falta y nos ocupa inútilmente el recurso. Ideal para documentos cortos y documentos que se quieran modificar.

- SAX: SAX toma el XML y cada vez que lee una etiqueta llama a una función callback para que esta sea procesada por el usuario. De esta forma, no existe árbol en memoria, pero a cambio, la modificación con SAX no es posible. Ideal para documentos largos o documentos de sólo lectura.

  Nosotros usaremos SAX.

3.2.5 PARSEANDO CON SAX

Como hemos dicho, SAX[49] es un analizador basado en eventos, y cuando éstos ocurren, se llama a funciones callback.

Xerces nos proporciona una clase base que ya tiene definidas dichas funciones, pero no implementadas (la implementación depende del XML, y por lo tanto asunto del usuario de Xerces).

Por lo tanto, lo primero que debemos hacer es definir una clase Handler que extienda dicha clase base:

```
#include <xercesc/sax/HandlerBase.hpp>

class MySAXHandler : public HandlerBase {
public:
    void startElement(const XMLCh* const, AttributeList&);
    void fatalError(const SAXParseException&);
};
```

La función startElement es llamada cuando se ha leído una etiqueta y sus parámetros son el nombre de la etiqueta y sus atributos.

La función fatalError es llamada cuando ha ocurrido un error irrecuperable al analizar.

Para comprender mejor su funcionamiento, podemos definir como clase de ejemplo una que muestre por consola el XML:

```
#include "MySAXHandler.hpp"
#include <iostream.h>

MySAXHandler::MySAXHandler()
{
}

MySAXHandler::startElement(const XMLCh* const name,
                           AttributeList& attributes)
{
    char* message = XMLString::transcode(name);
    cout << "I saw element: "<< message << endl;
    XMLString::release(&message);
}

MySAXHandler::fatalError(const SAXParseException& exception)
{
    char* message = XMLString::transcode(exception.getMessage());
    cout << "Fatal Error: " << message
         << " at line: " << exception.getLineNumber()
         << endl;
}
```

```
      #include <xercesc/parsers/SAXParser.hpp>
          #include <xercesc/sax/HandlerBase.hpp>
          #include <xercesc/util/XMLString.hpp>

          int main (int argc, char* args[]) {

              try {
                  XMLPlatformUtils::Initialize();
              }
              catch (const XMLException& toCatch) {
                  char*                    message                    =
XMLString::transcode(toCatch.getMessage());
                  cout << "Error during initialization! :\n"
                       << message << "\n";
                  XMLString::release(&message);
                  return 1;
              }

              char* xmlFile = "x1.xml";
              SAXParser* parser = new SAXParser();
              parser->setDoValidation(true);    // optional.
              parser->setDoNamespaces(true);    // optional

              DocumentHandler* docHandler = new HandlerBase();
              ErrorHandler* errHandler = (ErrorHandler*) docHandler;
              parser->setDocumentHandler(docHandler);
              parser->setErrorHandler(errHandler);

              try {
                  parser->parse(xmlFile);
              }
              catch (const XMLException& toCatch) {
                  char*                    message                    =
XMLString::transcode(toCatch.getMessage());
                  cout << "Exception message is: \n"
                       << message << "\n";
                  XMLString::release(&message);
                  return -1;
              }
              catch (const SAXParseException& toCatch) {
                  char*                    message                    =
XMLString::transcode(toCatch.getMessage());
                  cout << "Exception message is: \n"
                       << message << "\n";
                  XMLString::release(&message);
                  return -1;
              }
              catch (...) {
                  cout << "Unexpected Exception \n" ;
                  return -1;
              }

              delete parser;
              delete docHandler;
```

En este ejemplo, creamos un SaxParser y una instancia de nuestro Handler. Le decimos al Parser que utilice nuestro Handler para tratar los eventos. Como vemos, la clase Parser tiene métodos para validar automáticamente los XML partiendo de un DTD, pero ésta es sólo una de las muchas posibilidades que nos ofrece el parser SAX.
Para una información más detallada sobre la API de Xerces, consultar http://xml.apache.org/xerces-c/

3.2.6 - USO DE LA CLASE XMLREADER

La clase XmlReader fue en principio ideada para la lectura de ficheros xml simples de configuración mediante SAX.

Según fue avanzando el proyecto, se nos ocurrió agregarle funcionalidad extra para que pudiera leer ficheros xml complejos, y que por el contrario permitiese su consulta de forma fácil y cómoda.

Esto nos supuso bastante problema, puesto que el adaptar la lectura simple que hacíamos sobre nuestros ficheros pequeños (al que habíamos dado un enfoque SAX), habría requerido quizá un enfoque DOM desde el principio.

Los problemas surgieron primero en los diferentes grados de profundidad y en la búsqueda de propiedades, pero cuando encontramos el mayor escollo fue al parsear ficheros con etiquetas iguales al mismo nivel, dado que teníamos que tener constancia y poder comunicarle al usuario cuantos elementos iguales tenía y de que forma referirse a ello.

Para almacenar todo lo que leíamos utilizamos una hash, haciendo que la búsqueda sea más eficiente que DOM. Las claves de la hash son las rutas absolutas, de la siguiente forma:

```
<raíz>
   <servidor>
      <cosecha>
         <punto x = ″5″/>
         <punto x = ″10″/>
      <cosecha/>
      <punto x = “20″/>
      <pan>
            bueno
      <pan/>
   <servidor/>
<raíz/>

Da lugar a las entradas:
servidor.cosecha.punto[0]@x cuyo valor es “5”
servidor.cosecha.punto[1]@x cuyo valor es “10”
servidor.cosecha.punto@x cuyo valor es “20”
servidor.cosecha.pan cuyo valor es “bueno”
```

La raíz se eliminó por razones de comodidad.

Como vemos, un punto(.) marca el camino entre etiquetas mostrando la profundidad, y en última instancia la arroba marca la propiedad que queremos sacar. Los corchetes desambiguar entre elementos iguales como si fuera un array.

Es decir, hemos implementado a mano la herramienta XPath sobre árboles DOM.

3.2.7 - INTERFAZ PÚBLICA DE LA CLASE XMLREADER

Interfaz pública:

- XmlReader(): Crea un XmlReader vacío.
- bool Open(std::string archivo): Manda parsear un archivo a XmlReader. Si hay algún fallo, devolverá false.
- void Close(): Libera los recursos.
- int GetCount(std::string ruta): Devuelve el número de elementos del array que coincide con esa ruta. (Para hacer esto tuvimos que insertar el número de elementos de todo en la hash y después postprocesarlo para quedarnos con los significativos)
- template <typedef T> boost::lexical_cast<T> GetProperty(std::string ruta): Damos el nombre completo de la propiedad (la ruta) y nos devuelve su valor en el tipo que queramos.

El XmlReader es un componente reutilizable para cualquier proyecto de cualquier clase.

Ahora mismo la clase está siendo usada en el servidor y en el cliente para obtener la configuración de la red.

La clase XMLReader está siendo también utilizada por el módulo de mundo. Este módulo tiene XML diseñados para:

1. Describir los materiales de los escenarios y la posición y orientación de todos los objetos
2. Describir los materiales de cada modelo de objeto
3. Definir las propiedades de los distintos materiales

3.3. MANUAL DE USUARIO DEL EDITOR DE FUENTES

Se adjunta en este apartado un pequeño manual de usuario para el editor de las fuentes utilizadas en el proyecto.

Como se indicó en el apartado 2.3.1.6.1. los textos utilizados en el proyecto se definen mediante cuadrados sobre una textura de letras. Cada cuadrado equivale a una letra. Para definir los cuadrados para cada letra, se utiliza el presente editor de fuentes.

3.3.1.- Interfaz gráfica:

La interfaz que presenta el editor es muy sencilla. No tiene menú de modo que todas las acciones se realizan con los iconos que aparecen en la barra de iconos:

 **Abrir una fuente**: abre una fuente ya creada anteriormente para seguir editándola. Las extensiones de las fuentes son *.fnz.

 **Salvar la fuente actual**: salva la fuente actual en el directorio indicado.

 **Seleccionar la textura para la fuente**: selecciona una textura para la fuente. Se admiten todos los formatos soportados por la librería devIL, para más información consultar http://openil.sourceforge.net/features.php.

 **Mover el punto de vista**: desplaza el punto de vista con el movimiento del ratón.

 **Hacer un zoom**: Hace un zoom sobre la vista actual con el movimiento del ratón. Se puede realizar un zoom también con la rueda del ratón.

 **Borrar un carácter**: Borra el carácter actual de la lista de caracteres. Nota: el carácter de espacio es obligatorio e imborrable.

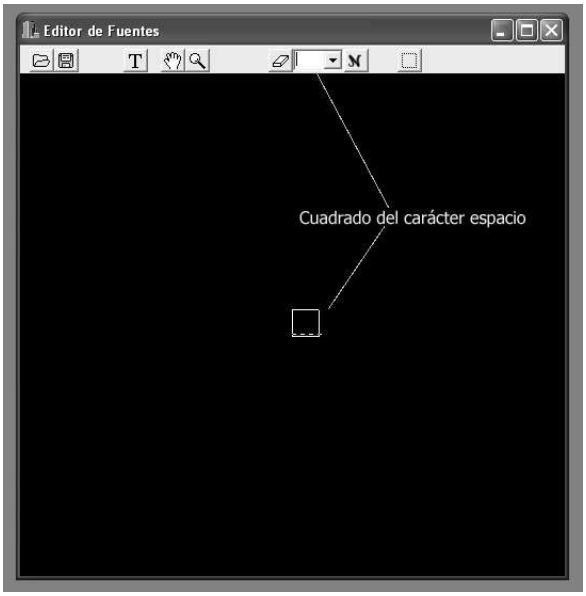
 **Crear un nuevo carácter**: crea un nuevo rectángulo para el carácter nuevo.

 **Modificar un carácter**: modica el carácter actual.

3.3.2.- Creando una fuente

Se adjunta en este apartado los pasos necesarios para la creación de una fuente.

Tras iniciar el editor, aparece con una fuente nueva lista para ser editada. Como se puede observar el carácter de espacio ' ' aparece ya creado y habrá que editarlo posteriormente:



Lo primero que se hará será seleccionar una textura para la fuente. Para ello se pincha sobre el botón T y se selecciona la textura de la fuente que se quiere editar. Tras esto, la textura aparecerá visible en la pantalla y lista para ser editada

Como se puede apreciar, no es necesario editar los 256 caracteres del código ASCII para cada fuente. En la pantalla anterior se escogió una fuente que tiene números, mayúsculas y minúsculas.

El paso siguiente es editar un carácter. Como por defecto aparece el carácter de espacio ' ' se comenzará por ese. Este carácter es el que se usa para separar unas letras de

otras, por lo que habrá que elegir un hueco vacío de la textura. En este caso dicho hueco se encuentra en la esquina superior izquierda. Pinchando sobre el botón de editar carácter y situando el cursor del ratón sobre el rectángulo de la imagen, se puede apreciar como dicho cursor cambia, mostrando la acción que puede realizar en ese instante:

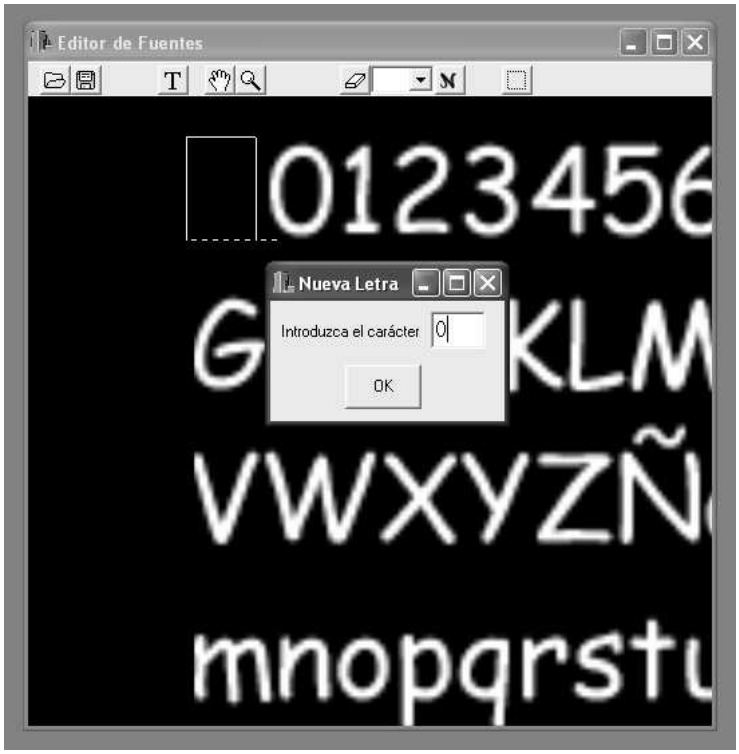
Alargar horizontalmente. El cursor se pone así cuando nos situamos en los lados izquierdo o derecho, y permite modificarlos.

Alargar verticalmente. Igual que el horizontal, pero para los lados superior e inferior. También sirve para desplazar la barra de altura de la letra.

Mover: Cuando se sitúa el cursor dentro del cuadrado adquiere este aspecto. Permite desplazar el cuadrado sin deformarlo.

Con todas esas operaciones se puede colocar el carácter de espacio, del siguiente modo:

Una vez se tiene dicho carácter colocado, es el momento de pasar al siguiente. Para ello, hay que pinchar sobre el botón de nuevo carácter y aparecerá un cuadro de diálogo solicitando el carácter que se quiere añadir, por ejemplo, el del número 0.



Al introducir el carácter pulsando OK se puede apreciar que el carácter que aparece seleccionado es el 0

lo que quiere decir que ya desde ese momento se está editando dicho carácter. También conviene apreciar que al crear un carácter nuevo, el cuadrado de inicio con el que se empieza es una copia del de la letra que se tenía seleccionada. Esto se puede aprovechar para hacer que dos caracteres sean exactamente iguales. Por ejemplo una letra minúscula y su correspondiente mayúscula, en el caso de una fuente donde mayúsculas y minúsculas coincidan.

Procediendo como en el caso anterior para el carácter de espacio ' ', se declara el carácter 0 y del mismo modo el resto de caracteres.

Hay ciertos caracteres que no están a la misma altura que el resto. Por ejemplo, véase la letra 'q' :

La forma correcta de declarar este tipo de caracteres no es la que aparece en esa imagen sino que se debe desplazar la barra de altura para indicarle de algún modo cuál es la línea sobre la que debe apoyarse:

Lo mismo ocurriría con la 'p', el carácter de la coma ',' y similares.

En el caso de que haya una equivocación al declarar una letra, basta seleccionarla y pinchar sobre el botón  y será automáticamente eliminada. Si se quiere volver a declarar se procederá como ya se indicó.

Una vez finalizada la fuente, se pincha sobre el botón salvar  y se eligirá donde se quiere almacenar. Importante: la textura utilizada para la fuente debe encontrarse en el mismo directorio que el fichero *.fnz creado con el editor. Si no, la próxima vez que se abra dicha fuente el editor no podrá localizar la textura .

3.4 MANUAL DE USUARIO DE LA APLICACIÓN

Se adjunta a continuación un pequeño manual para poder disfrutar del proyecto realizado. El proyecto son dos ejecutables. El primero de ellos es el servidor, que debe ser lanzado en alguna máquina para poder conectar los clientes. El otro es el cliente.

Los requisitos para poder disfrutar del chat son:

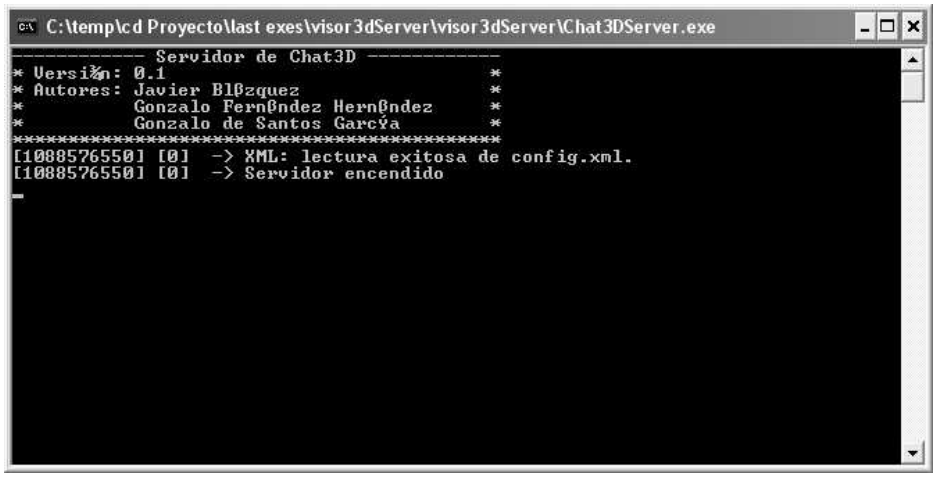
- Pentium 3 700 Mhz o superior.
- Tarjeta gráfica que soporte opengl 1.5 (mínimo ATI Radeon 9500 ó NVIDIA GeForce FX) con 64MB de memoria de video.
- Sistema operativo Windows 2000 o posterior.
- Conexión de ADSL para disfrutar del chat por internet. Cuanto más rápida mejor, sobretodo para el caso del servidor.

3.4.1 El servidor

Se debe modificar el archivo config.xml que se encuentra en el mismo directorio que el servidor. Se deben configurar tres apartados:

- El puerto por el que se quieren recibir los paquetes de los clientes: para modificarlo, se accede a la línea <port>...</port> y se escribe el puerto deseado.
- El mapa: para elegir el mapa donde se va a chatear, modificar la linea <map>...</map> escribiendo uno de los directorios de la carpeta maps.
- Posición inicial de los clientes: Se puede indicar el sitio donde se quiera que aparezcan los clientes que se vayan conectando al chat.. Para ello, modificar la línea <position x="-20" y="4" z="0" /> por los valores deseados (en este caso aparecerían en la posición –20, 4, 0)

Configurado el servidor, se puede arrancar ejecutando Chat3Dserver.exe. Si todo ha ido bien se verá una pantalla como la siguiente:



3.4.2. El cliente

Para lanzar el cliente se procede igual que en el caso del servidor. Lo primero que se debe hacer es editar el fichero config.xml. Para ello, se modifican los siguiente campos:

- La ip del servidor: se debe colocar en la línea <server>...</server>
- El puerto: este puerto es el que se le indicó al servidor. Se debe colocar en la línea <port>...</port>
- El mapa donde se va a chatear: que debe ser el mismo que el que se indicó en el servidor. Se indica en la línea <map>...</map>
- La posición de comienzo: que es la misma que se indicó en el servidor, y se escribe del mismo modo

A continuación se debe ejecutar chat3D.exe. Si todo va bien, aparecerá una ventana como la siguiente:

Desde ese momento, se ha establecido la conexión y se podrá pasear y hablar con el resto de clientes. Los controles para el muñeco son:

- Las flechas de dirección para moverse por el mundo.
- Cualquiera de los caracteres del teclado para escribir textos. Una vez escrito un texto, se puede enviar con la tecla intro. En ese momento aparecerá un bocadillo en la cabeza del muñeco y comenzará a decir lo que se haya escrito.
- Dentro del texto se pueden introducir los emoticonos para que nuestro personaje realice las animaciones deseadas. Estos emoticonos van siempre entre corchetes [...], y son los siguientes:

Emoticono	Acción
[XD]	Chatín se ríe
[¬¬]	Chatín muestra un gesto de desprecio
[O_O]	Chatín se queda alucinado
[E)]	Chatín se rasca la nuca
[>0]	Chatín se cabrea
[;)]	Chatín hace un guiño
[9_9]	Chatín suspira
[@_@]	Chatín se marea
[#_#]	Chatín pone cara de bueno
[._.]	Chatín no sabe que decir
[XDO]	Chatín intenta contener su carcajada
[U_U]	Chatín se cruza de brazos y asiente
[:_(]	Chatín se pone a llorar
[(O)_(O)]	Chatín se queda realmente impresionado
[:P]	Chatín hace un gesto burlón

4 - CONCLUSIONES

Finalmente se ha logrado la tan ansiada implementación del chat3d. Se ha conseguido crear un protocolo suficientemente bueno para que la conexión por red sea jugable, se han realizado las detecciones de colisiones mediante bsp y se han conseguido unos muñecos animados con la suficiente vida. Las tres cosas juntas dan lugar a un bonito y original programa, a nuestro parecer.

Hay muchas partes de este proyecto que se podrían aprovechar por separado. La red permite la transmisión de las coordenadas de los muñecos y se podría reutilizar para realizar un chat en 2d, por ejemplo. Parte de este chat 2d, por cierto, se llegó a implementar. En cuanto a los escenarios, la detección de colisiones se podría realizar no sólo con los muñecos del chat, ni únicamente con los mundos que vienen con el chat; sino que se podrían usar otros escenarios. Del mismo modo, los muñecos se pueden utilizar para realizar cualquier otro tipo de juegos donde se requiera animación, o quizá para leer bonitas presentaciones del 3dstudio Max.

El grupo que ha realizado el presente proyecto está bastante contento con el resultado aunque se han quedado muchas cosas en el tintero. Una parte con la que nos sentimos realmente contentos es con la integración final, pues, aunque cada uno ha llevado su parte por separado, se han conseguido juntar todas al final, dando como resultado un chat muy completo. Una pena que se queden sin realizar cosas como la personalización de los muñecos mediante avatares, detección de colisiones con los objetos del escenario y una implementación de red que sea más robusta frente al lag.

Pensamos que muchas de las partes que se han implementado podrían servir de punto de partida para futuros proyectos. Gran parte del código fuente está perfectamente documentado, por lo que navegar en él no debería ser demasiado complicado. A nosotros se nos ha acabado el tiempo, pero si alguien desea tirar de aquí, francamente nos sentiríamos muy contentos.

5 - BIBLIOGRAFÍA

[1] Akenine-Möller, Haines, *Real-Time Rendering Second Edition*, A K Peters Ltd., http://www.realtimerendering.com/, 2002

[2] Cohen, Lin, Manocha, Ponamgi, "I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scaled Environments", *Proceedings of the 1995 ACM International 3D Graphics Conference*, pp. 189-196, http://haptic.mech.nwu.edu/library/cohenj/acm95/, 1995

[3] Zachmann, "Rapid Collision Detection by Dynamically Aligned DOP-Trees", Fraunhofer Institute for Computer Graphics, http://web.informatik.uni-bonn.de/II/ag-klein/people/zach/papers/vrais98.html, 1998

[4] Gottschalk, Lin, Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection", *Proceedings of ACM Siggraph*, pp. 171–180, 1996.

[5] Eberly, *Dynamic Collision Detection using Oriented Bounding Boxes*, http://www.magic-software.com/Documentation/DynamicCollisionDetection.pdf, 1999

[6] Meyers, "Item 31: Making functions virtual with respect to more than one object", *More Effective C++*, Addison-Wesley, 1996

[7] Alexandrescu, *Modern C++ Design*, http://www.moderncppdesign.com/, Addison-Wesley, 2001

[8] Eberly, *Wild Magic v2.3 Library*, http://www.magic-software.com/SourceCode.html, 2004

[9] Eberly, *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufmann Publishers, 2000

[10] Watt, Policarpo, *3D Games: Real-time Rendering and Software Technology – Volume One*, Addison-Wesley, 2001

[11] Barequet, Chazelle, Guibas, Mitchell, Tal, "BOXTREE: A Hierarchical Representation for Surfaces in 3D", 1996

[12] de Berg, van Kreveld, Overmars, Schwarzkopf, *Computational Geometry by Example*, 1995

[13] Catmull, Edwin, "Computer Display of Curved Surfaces", *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures*, pp. 11-17, 1975.

[14] Everitt, "Interactive Order-Independent Transparency", *NVIDIA OpenGL Applications Engineering*, http://developer.nvidia.com/object/Interactive_Order_Transparency.html, 2001

[15] Wade, "Binary Space Partitioning Trees FAQ", http://www.faqs.org/faqs/graphics/bsptree-faq/, 1995

[16] Naylor, "A Tutorial on Binary Space Partitioning Trees", Spatial Labs Inc.

[17] Ranta-Eskola, "Binary Space Partioning Trees and Polygon Removal in Real Time 3D Rendering", Computing Science Department, Uppsala University, 2001

[18] James, "Binary Space Partitioning for Accelerated Hidden Surface Removal and Rendering of Static Environments", University of East Anglia, 1999

[19] Luebke, Georges, "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets", Department of Computer Science, University of North Carolina, http://www.cs.virginia.edu/~luebke/publications/portals.html, 1995

[20] de Berg, de Groot, Overmars, "Perfect Binary Space Partitions", Technical Report RUU-CS-93-23, Department of Computer Science, Utrecht University, 1993

[21] Beltrán, del Cerro, Cuenca, García, "Motor gráfico para el desarrollo de videojuegos", Facultad de Informática, Universidad Complutense de Madrid, 2003

[22] Cassen, Subramanian, Michalewicz, "Near-Optimal Construction of Partitioning Trees by Evolutionary Techniques", Department of Computer Science, University of North Carolina, 1995

[23] *IEEE 754: Standard for Binary Floating-Point Arithmetic*, IEEE, 1985

[24] Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[25] Kirkland, Podder, Urquhart, "OpenGL ARB Extension #11: WGL_ARB_pbuffer", http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pbuffer.txt, 2000

[26] Arévalo, "Main Loop with Fixed Time Steps", http://www.iguanademos.com/Jare/docs/FixLoop.htm, 2001

[27] Craighead, Ginsburg, "OpenGL ARB Extension #29: GL_ARB_occlusion_query", http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt, 2003
[28] "Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics v1.2", ftp://download.nvidia.com/developer/cg/Cg_1.2.1/Docs/Cg_Toolkit.pdf, 2004
[29] Fernando, Kilgard, *Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, 2003
[30] Kessenich, Baldwin, Rost, *The OpenGL Shading Language Specification v1.10*, http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf, 2004
[31] Rost, *OpenGL Shading Language*, Addison-Wesley, 2004
[32] Cronin, Kurc, Filstrup, Jamin, "An Efficient Synchronization Mechanism for Mirrored Game Architectures (Extended Version)", Electrical Engineering and Computer Science Department, University of Michigan, 2003
[33] Mauve, "Consistency in Continuous Distributed Interactive Media", University of Mannheim, 1999
[34] Tomas Möller, Eric Haines, *Real-Time Rendering*, A. K. Peters, 1999
[35] *3ds Max Plug-In Software Development Kit* , version 5.1, 27-Noviembre-2002 (se incluye con la sdk del 3dstudio Max)
[36] *Public Sparks Message Archive*, 21-Noviembre-2002 (se incluye con la sdk del 3dstudio Max)
[37] Mark J. Kilgard, *All About opengl Extensions, including specifications for some significant opengl extensions*, NVIDIA Corporation, http://developer.nvidia.com/attach/6296
[38] Chris Wynn, *OpenGL Render-to-Texture*, NVIDIA Corporation, http://developer.nvidia.com/attach/6725
[39] Chris Wynn, *Using P-Buffers for Off-Screen Rendering in OpenGL*, NVIDIA Corporation, 08/09/2001, http://developer.nvidia.com/attach/6533
[40] Mark J. Kilgard, *All Nvidia OpenGL Extensions*, http://www.nvidia.com/dev_content/nvopenglspecs/nvOpenGLspecs.pdf (para consultar alguna extensión en concreto se puede acceder también a http://developer.nvidia.com/object/nvidia_opengl_specs.html)
[41] Paula Womack, *WGL_ARB_pixel_format,* Author Revision: 1.0, March 22, 2000, http://www.nvidia.com/dev_content/nvopenglspecs/WGL_ARB_pixel_format.txt
[42] Steve Baker, *Fast Text in OpenGL*, http://sjbaker.org/steve/omniv/opengl_text.html
[43] Mark J. Kilgard, *The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3*, Silicon Graphics Inc, http://www.opengl.org/resources/libraries/glut/spec3/spec3.html
[44] James Joplin, Denton Woods, DevIL API Reference, http://openil.sourceforge.net/docs/function_index.htm
[45] Excel Technology, "Opentop 1.3 Reference Manual", http://www.elcel.com/docs/opentop/1.3/API/ , 2003
[46] Microsoft Corporation, "Winsock Reference: Windows Sockets 2.0", http://msdn.microsoft.com/library/en-us/winsock/winsock/winsock_reference.asp , 2004
[47] Nilo Mitra, "SOAP Version 1.2 Part 0: Primer, W3C Recommendation, 24th June 2003", http://www.w3.org/TR/2003/REC-soap12-part0-20030624/ , 2003
[48] The Apache Software Foundation, "Xerces C++ Documentation", http://xml.apache.org/xerces-c/apiDocs/index.html , 2004
[49] David Brownell, "SAX2", O'Reilly , 2002
[50] Joel Fan, Eric Ries, Calin Tenitchi, "Juegos en java", Anaya Multimedia, 1998
[51] Mauve, "How to Keep a Dead Man from Shooting", University of Mannheim
[52] Cronin, Filstrup, Jamin, "Cheat-Proofing Dead Reckoned Multiplayer Games", Electrical Engineering and Computer Science Department, University of Michigan
[53] Smed, Kaukoranta, Hakonen, "A Review of Networking and Multiplayer Computer Games", Department of Computer Science, University of Turku, 2002
[54] François Yergeau, John Cowan, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, "*XML 1.1*, W3C Recommendation, 4th February 2004", http://www.w3.org/TR/2004/REC-xml11-20040204/ , 2004

Los autores del presente proyecto - Javier Blázquez de Miguel, Gonzalo de Santos García y Gonzalo Fernández Hernández – autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, el código, la documentación y los ejecutables desarrollados para este proyecto.

Madrid a 8 de Julio de 2004

Firmado: