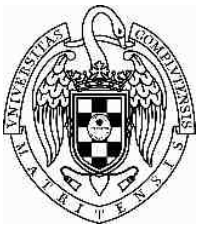


Sistemas Informáticos
Curso 2003-04

Simulador de Cooperativismo robótico utilizando técnicas de juegos

Sandra Sánchez Arribas
Santiago Beca Moreno
Juan Ignacio Plaza Alonso

Dirigido por:
Prof. Segundo Esteban San Román
Dpto. Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

INDICE

I. Resumen

Se ha desarrollado un Simulador para cooperativismo robótico utilizando técnicas de juegos. El comportamiento dinámico de los robots se simula remotamente en nodos diferentes al nodo simulador, de tal forma que los nodos-robot se comunican con el nodo-simulador y con el resto de los nodos-robot a través de la red.

El nodo-simulador debe comunicar a los nodos-robots si han sufrido alguna colisión para que éstos puedan modificar su dinámica. A su vez, los nodos-robots comunican al nodo-simulador cualquier cambio en su estado.

Por la similitud de esta idea con los famosos juegos de acción multi-jugador (por ejemplo Quake, Doom, etc), aprovecharemos técnicas y herramientas utilizadas en ellos:

1. El potencial de OpenGL para implementar la representación gráfica de la simulación.
2. El potencial de comunicación de Direct-Play para realizar la comunicación entre el nodo-simulador y los nodos-robots, y entre nodo-robot y nodo-robot.
3. Técnicas de detección de colisiones utilizadas en los juegos para comunicar a los nodos-robots si han colisionado.

Resumen en inglés

A robotic cooperative simulator has been developed using game techniques. The robots' dynamic behaviour has been remotely simulated in different nodes to the simulator node, in such way that the robots communicate with each other and with the simulator-node through the network.

The simulator-node must communicate the robots if they have suffered a collision so they can change their dynamic. At the same time, robots must communicate any change in their state to the simulator-node.

Due to the similarity with multiplayer games (ie. Quake, Doom, etc), we have taken advantage of the techniques and tools used in them:

1. OpenGl potential to implement the simulation graphical representation.
2. DirectPlay communication potential to establish the communication between the simulator-node and the robots and among the robots.
3. Collision detection techniques used in these games to communicate the robots if a collision has occurred.

II. Especificación general de requisitos

Tras estudiar el alcance del proyecto y se ha procedido a su división dos fases de implementación claramente diferenciadas:

1. Prototipo de la arquitectura cliente/ servidor
2. Integración del juego con la arquitectura

Se detalla a continuación la especificación de cada fase.

Prototipo de la arquitectura Cliente / Servidor

La finalidad del prototipo es asegurar el conocimiento de las nuevas tecnologías que se van a aplicar en el simulador: DirectPlay de DirectX

Se construirá una aplicación 'Servidor' que gestione la comunicación entre los clientes proporcionando los datos que le sean solicitados, bien información de los otros clientes, bien información del propio servidor.

La aplicación 'Cliente' dispondrá de opciones para realizar la búsqueda de servidores activos y para unirse a uno de ellos.

Una vez establecida la comunicación ente ambas aplicaciones, los clientes ofrecerán la posibilidad de iniciarse en el juego.

El funcionamiento de un cliente dentro del juego es completamente independiente del resto de los clientes. Disponen de una inteligencia artificial compuesta por una serie de estados y los clientes obtienen la información necesaria para cambiar de estado vía servidor. Esta información va encapsulada en mensajes soportados por DirectPlay.

Los tipos de datos que manejan tanto el cliente como el servidor se elegirán teniendo en cuenta los datos que maneja el juego, para facilitar así la integración posterior.

Integración del juego con la arquitectura

El juego dispone de una estructura de datos que es la que contiene los objetos de la aplicación, esto es personajes, terreno, etc. Y usa esta estructura para hacer las tareas de visualización y los cálculos necesarios para la detección de colisiones.

Esta etapa de integración incluye varias fases:

1. Lo primero es hacer que los clientes se puedan tratar como un objeto más dentro de la estructura del juego, esto es, se puedan añadir a la estructura, se puedan visualizar y ejecutar la detección de colisiones y, finalmente, se puedan eliminar de la estructura cuando el cliente 'muera'.

2. Es necesario encontrar una fórmula eficaz y eficiente para que la estructura de datos del juego pueda detectar los cambios que se producen en los distintos clientes. Cambios de posición o estado de la inteligencia artificial.

3. Encontrar la forma de gestionar una ventana OpenGl desde la aplicación Servidor para hacer la visualización desde aquí. Seria deseable que la ventana incluya un menú con opciones para cambiar la cámara y así poder hacer la visualización desde distintos ángulos.

III. Fase de documentación en nuevas tecnologías

El objetivo de esta fase es dominar las técnicas y herramientas que se emplean actualmente en el desarrollo de juegos de acción multi-jugador (por ejemplo Quake o Doom)

A. DirectPlay

El API de Microsoft DirectPlay en un componente de Microsoft DirectX que proporciona las herramientas para el desarrollo de aplicaciones multi-jugador tales como juegos o chats. La versión sobre la que nos hemos documentado y que hemos utilizado es la que está incluida en Microsoft DirectX 8.0 para C/C++.

Una característica común a este tipo de aplicaciones es que, en todas ellas, hay al menos dos usuarios independientes donde cada uno de los cuales tiene la aplicación 'cliente' en su ordenador. Además aplicación debe permitir a los usuarios la comunicación entre ellos a través de un a red.

Lo que DirectPlay proporciona al desarrollador de aplicaciones multi-jugador, es abstracción de los detalles del tipo de red que está por debajo de la aplicación, para poder concentrar así sus esfuerzos en otros aspectos de la programación.

Los cuatro aspectos más importantes en el uso de DirectPlay son:

1. Creación y gestión de sesiones
2. Topologías para el paso de mensajes dentro de una sesión.
3. Comunicación en red.
4. Comunicación con los objetos DirectPlay.
5. Soporte de sesiones lobby.
6. Soporte para la comunicación por voz.

1. Creación y gestión de sesiones:

Se habla de la sesión de un juego para referirse a una instancia particular del mismo. Consta de dos o más usuarios que juegan simultáneamente y cada uno de ellos con la aplicación instalada en su ordenador.

El primer paso en la creación de una sesión es reunir a un grupo de usuarios. Para ello hay dos maneras de hacerlo: a través de una aplicación 'lobby' que este corriendo en un servidor remoto o bien comunicando los usuarios individuales con el resto. Más tarde se discutirá el uso de sesiones lobby.

Una vez que la sesión ha sido creada y lanzada, el juego empieza. Nuevo jugadores pueden unirse a la sesión y también pueden ser eliminados de la misma los jugadores ya existentes.

En toda aplicación multi-jugador, cada jugador debe estar sincronizado con el resto de los jugadores que están en la misma sesión. Esto implica un flujo de mensajes a y desde cada jugador. Por ejemplo, cada vez que un jugador cambia de posición se debe enviar un mensaje para actualizar la posición de este jugador en el resto de las aplicaciones cliente. La esencia de DirectPlay es que una parte de su API se encarga de proporcionar un soporte eficiente y flexible del paso de mensajes entre los ordenadores de una misma sesión.

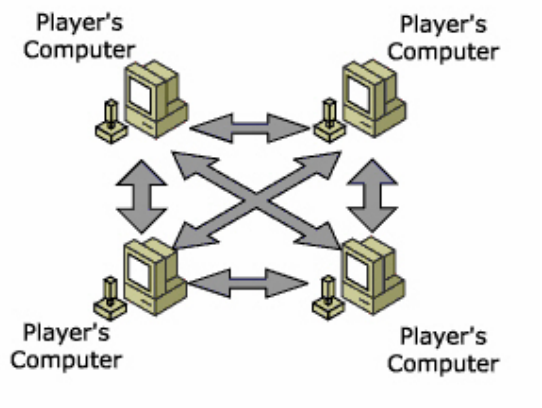
2. Topología para el paso de mensajes dentro de una sesión.

Hay dos formas básicas de estructurar la topología del paso de mensajes de una sesión: arquitectura punto a punto y cliente/ servidor. Ambas topologías tienen ventajas y desventajas, así pues es tarea del diseñador evaluar cual de las dos es mas apropiada para una aplicación concreta.

a. Arquitectura punto a punto

En esta topología cada usuario se comunica directamente con el resto de los usuarios. Por ejemplo, cuando un usuario cambia su posición, éste tiene que enviar un mensaje a cada uno de los usuarios de la sesión.

Esquemáticamente, una topología punto a punto para cuatro usuarios tiene la siguiente apariencia:



En este tipo de arquitectura la sesión se establece normalmente a través de un cliente lobby que reside en ordenador del usuario. Esta comunicación puede hacerse desde un cliente lobby a otro directamente o bien usar el cliente lobby como un link a un servidor lobby que corre en un servidor remoto.

Si se usa la segunda aproximación las únicas responsabilidades del servidor remoto serán mantener actualizados los miembros de la sesión y permitir la entrada en la sesión a nuevos usuarios.

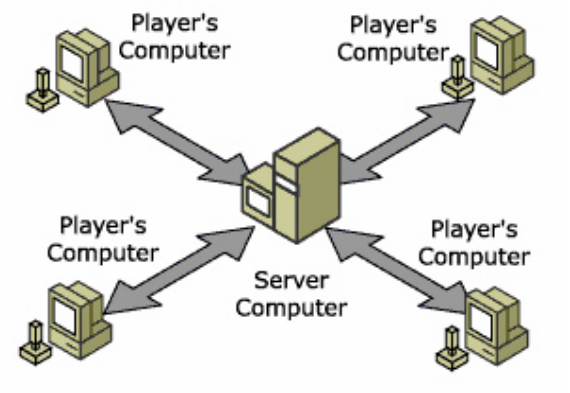
Dentro de esta arquitectura hay que designar a uno de los usuarios como host del juego. El host es el encargado de gestionar los detalles logísticos como por ejemplo añadir nuevos usuarios a la sesión que está en marcha.

La ventaja principal de la arquitectura punto a punto es la simplicidad pero juegan en su contra la difícil escalabilidad y la posible saturación de la red ante un elevado tráfico de mensajes.

b. Arquitectura cliente/servidor

En esta topología todos los usuarios están conectados a un servidor centralizado y realizan las comunicaciones a través del mismo.

Esquemáticamente, una topología cliente/servidor para cuatro usuarios tiene la siguiente apariencia:



El servidor es el responsable de pasar la información de un usuario a otro. Por ejemplo, cuando un usuario cambia de posición, le envía un mensaje al servidor y es el servidor quien se encarga de mandar mensajes al resto de los usuarios. Además de esto, el servidor tiene otras responsabilidades como ser el host del juego y realizar otras operaciones para mantener el 'universo del juego' que está en el servidor.

Los clientes son, esencialmente, responsables de manejar su propia interfaz de usuario.

En este tipo de arquitectura la sesión se establece de forma similar a como se hace en la topología punto a punto. Un cliente lobby que reside en ordenador del usuario actúa de link con un servidor lobby que reside normalmente en el mismo ordenador que el servidor.

La escalabilidad de este tipo de arquitectura es bastante mayor que la de la arquitectura punto a punto. Jugadores adicionales solo causan un incremento lineal en el trafico del paso de mensajes. Esta característica hace que las aplicaciones con gran numero de usuarios usen esta arquitectura.

El inconveniente más destacable de la arquitectura cliente/servidor es que una vez terminada la aplicación necesitará una serie de servicios de soporte y mantenimiento para el servidor y el software asociado junto con la red que gestiona el paso de mensajes.

3. Comunicación en red

Como ya se dijo anteriormente, la primera funcionalidad que pretende cubrir DirectPlay es la de proporcionar un soporte eficiente y flexible de la gestión del paso de mensajes que permita aislar la aplicación del software y hardware de red subyacente. DirectPlay soporta comunicación sobre TPC/IP, IPX, MODEM y "serial links".

a. Protocolo de transporte de DirectPlay

El protocolo ofrece una serie de características que han sido adaptadas a las necesidades de las aplicaciones multi-jugador. Estas características incluyen:

- Entrega de mensajes con o sin confirmación. Los mensajes con confirmación se pueden reenviar hasta que lo reciba el destinatario.
- Entrega de mensajes secuencial y no secuencial. En el modo secuencial los mensajes se pasan en el mismo orden e que sean enviados.
- Fragmentación de mensajes si el tamaño del mensaje excede la capacidad de la red. Este proceso es automático.
- Control de la congestión.
- Posibilidad de cambiar la prioridad de un mensaje
- Establecimiento de un 'timeout' para el envío de un mensaje. Si pasado un este tiempo se elimina de la cola de mensajes, sin importar si se ha podido enviar o no.

b. Direcciones DirectPlay

Con el objetivo de poder enviar mensajes, cada participante en la aplicación muti-jugador debe tener asignada una dirección única. Las direcciones pueden referirse al ordenador donde está corriendo la aplicación (device address) o bien al ordenador con el cual tu aplicación necesita comunicarse (host address).

Las direcciones DirectPlay tienen la forma de una cadena que sigue un formato general:

x-directplay:/[cadena de datos]

La cadena de datos contiene varios elementos que especifican toda la información necesaria para la comunicación, por ejemplo el proveedor de servicios, el nombre del host, el puerto, etc.

4. Comunicación con los objetos DirectPlay

DirectPlay consiste básicamente en una colección de objetos COM (Component Object Model). Cada objeto presenta uno a más interfaces que permiten al programador tener control de varios aspectos de DirectPlay. Por ejemplo el objeto peer (CLSID_DirectPlay8Peer) se usa para manejar juegos basados en arquitectura peer-to-peer.

La comunicación con los objetos DirectPlay se hace a través de los métodos implementados en sus interfaces. Para enviar datos, por ejemplo, a otro usuario en una sesión peer-to-peer hay que enviar un mensaje llamando a la función IDirectPlay8Peer::SentTo y DirectPlay se encarga de hacer llegar el mensaje a su destino.

La forma en que DirectPlay se comunica con la aplicación es a través de las llamadas funciones 'callback'. La aplicación tendrá que implementar la función callback y durante la inicialización se pasara a DirectPlay un puntero a dicha función. Cuando DirectPlay necesita comunicarse con la aplicación hará una llamada a la función callback con dos argumentos clave de información: Un identificador del tipo de mensaje que se está enviando y un puntero a un bloque(normalmente a una estructura) que proporciona los detalles necesarios. Siguiendo con el ejemplo anterior, cuando el mensaje enviado anteriormente llega a su destino, la callback de la aplicación destino recibirá un mensaje con identificador DPNMSGID_RECEIVE acompañado a su vez de una estructura que contiene los datos que se enviaron.

5. Sesiones Cliente/Servidor en detalle

Cliente/Servidor ha sido la topología usada para el desarrollo del proyecto. Es por ello por lo que dedicamos esta sección a profundizar un poco más. Como ya se dijo anteriormente una sesión Cliente/Servidor está formada por una colección de jugadores o clientes conectados a través de un servidor central y donde cada jugador no dispone de información a cerca del resto de jugadores.

La implementación de una sesión Cliente/Servidor consta, como su propio nombre indica de dos aplicaciones perfectamente separadas: las aplicaciones Cliente y Servidor.

La aplicación Servidor corre en un servidor remoto y como mínimo sirve de central para el paso de mensajes y es el host del juego. Debe además poder recibir y manejar todos los mensajes procedentes de los clientes y debe a su vez poder enviar el mensaje apropiado en respuesta al recibido. Cualquier transferencia de datos entre dos clientes debe ser manejada por el servidor.

La aplicación Cliente corre en la máquina de cada cliente y su principal función es implementar el interfaz de usuario y mantener el estado del jugador sincronizado con el servidor.

a. Inicialización de la sesión Cliente/Servidor

En el servidor:

La sesión puede ser lanzada directamente por el servidor o bien usando un lobby. La manera más directa de hacerlo es implantando un a aplicación lobbyable. Esta aproximación ofrece una forma de lanzar el servidor y de soportar la comunicación entre el servidor y el lobby durante el transcurso de la sesión. Se discutirán en mas detalle los lobbies de DirectPlay en secciones posteriores.

La otra opción es lanzar la sesión directamente desde el servidor, esto es, es el propio servidor el que se anuncia como disponible y espera a que se le conecten los clientes. Esta ha sido la forma de lanzar la sesión en el proyecto.

Una vez que la sesión ha sido lanzada se debe inicializar a si mismo llamando al método IdirectPlay8Server::Initialize cuyo principal propósito, es proporcionar a DirectPlay un puntero al gestor de mensajes callback. La implementación en esencia es:

```
// objeto servidor DirectPlay
IDirectPlay8Server* dpServer       = NULL;

//Crear IDirectPlay8Server
if( FAILED( hr = CoCreateInstance( CLSID_DirectPlay8Server,
NULL,
                          CLSCTX_INPROC_SERVER,
                          IID_IDirectPlay8Server,
                          (LPVOID*)&dpServer)))

//Inicializar IDirectPlay8Server
if( FAILED( hr = dpServer->Initialize( NULL,
GestorMensajesDirectPlay, 0 ) ) )
        return DXTRACE_ERR( TEXT("Initialize"), hr );
hr = CoCreateInstance( CLSID_DirectPlay8Address, NULL,
                          CLSCTX_ALL, IID_IDirectPlay8Address,
                          (LPVOID*) &pDP8DirLocal );
```

En el cliente:

En esta parte de la aplicación se debe crear e inicializar el objeto cliente (CLSID_DirectPlay8Client). La implementación es:

```
//Declaración del objeto servidor
IDirectPlay8Server* dpServer       = NULL;

// Creamos el  IDirectPlay8Client
if( FAILED( hr = CoCreateInstance( CLSID_DirectPlay8Client,
                          NULL,
                          CLSCTX_INPROC_SERVER,
                          IID_IDirectPlay8Client,
                          (LPVOID*) &g_pDPClient ) ) )

        return DXTRACE_ERR( TEXT("CoCreateInstance"), hr );

//Inicializar IDirectPlay8Client
if( FAILED( hr = g_pDPClient->Initialize( NULL,
DirectPlayMessageHandler,0)))
        return DXTRACE_ERR( TEXT("Initialize"), hr );
```

Este objeto es la clave principal para la comunicación entre DirectPlay y el servidor.

b. Seleccionar un proveedor de servicios para el cliente

En nuestro caso el proveedor de servicios es TCP/IP. Lo primero que haremos es inicializar la ListBox del formulario para mostrar los servidores disponibles y preparar el proveedor de servicios para poder usarlo. Para esto último hay que hacer una llamada al método IDirectPlay8Client::GetSPCaps pasándole como argumento el proveedor de servicios deseado, en este caso: CLSID_DP8SP_TCPIP. Este método recibe como respuesta un puntero a la

estructura de datos DPN_SP_CAPS que contiene información del proveedor de servicios. La implementación es:

```
DPN_SP_CAPS dpspCaps;
ZeroMemory( &dpspCaps, sizeof(DPN_SP_CAPS) );
dpspCaps.dwSize = sizeof(DPN_SP_CAPS);
m_pDPClient->GetSPCaps( &CLSID_DP8SP_TCPIP, &dpspCaps, 0 );
```

Entre la información recibida en la estructura esta dwDefaultEnumRetryInterval que especifica el intervalo por defecto para volver a probar.

```
m_dwEnumHostExpireInterval =
                    dpspCaps.dwDefaultEnumRetryInterval * 3;
```

c. Enumerar las sesiones del juego que estén disponibles

El siguiente paso es enumerar las sesiones que hay abiertas en una máquina sabiendo su dirección IP y mostrarlas en la ListBox. Esto se hace llamando al método IDIrectPlay8Client::EnumHosts :

```
hr = m_pDPClient->EnumHosts( &dpnAppDesc, pDP8AddressHost,
                        pDP8AddressLocal, NULL,
                        0, INFINITE, 0, INFINITE, NULL,
                        &m_hEnumAsyncOp, 0 );
```

Los parámetros mas importantes de este método se definen a continuación. El primer parámetro es un puntero a la estructura DPN_APPLICATION_des que especifica el host para hacer la enumeración.

pDP8AddressHost es un puntero que especifica la dirección del ordenador que es el host de la aplicación, es decir, la dirección DirectPlay del servidor.

pDP8AddressLocal es también un puntero a una dirección DirectPlay que especifica el proveedor de servicios y las opciones de la máquina local para que se usan al hacer la enumeración.

d. Conectarse a una sesión Cliente/Servidor

Todos los clientes deben explícitamente conectarse al host para participar en la sesión. La conexión establece al cliente como un miembro de la sesión y proporciona al host la información necesaria para mantener la comunicación con el cliente. Por otra parte el hots tiene la opción de aceptar o rechazar la conexión realizada por un cliente.

Veamos por separado lo que ocurre el las aplicaciones cliente y servidor:

En el servidor :

Cuando un cliente intenta unirse a una sesión, el host recibe un mensaje de conexión (DPN_MSGID_INDICATE_CONNECT). Para acepar al jugador en la sesión se devolverá S_OK (cualquier otro valor que se devuelva significa que la conexión se ha rechazado) y el cliente recibirá un mensaje DPN_MSGID_CONNECT_COMPLETE con el contenido de la respuesta. Si el jugador se ha añadido con éxito a la sesión, todos los clientes y el propio servidor recibirán un mensaje (DPN_MSGID_CREATE_PLAYER) con el identificador del nuevo jugador, es decir su DPNID que lo distingue unívocamente del resto de los objetos.

En el cliente:

Para que un cliente pueda conectarse a una sesión, debe tener la dirección del host de la sesión. Si la aplicación realizó la conexión a través de un cliente lobby se puede obtener la dirección haciendo una llamada al método IDirectPlay8LobbiedApplication::GetConnectionSettings. En otro caso, si el

proveedor de servicios usado es IO o IPX se verán las sesiones disponibles llamando a IDirectPlay8Client::EnumHost como se dijo anteriormente. La información devuelta por la enumeración incluye la dirección del host de cada una de las sesiones. Finalmente para realizar la conexión se llama al método IDirectPlay8Client::Connect con el host seleccionado. El gestor de mensajes debe entoncesrecibir una mensaje PDN_MSGID_CONNECT_COMPLETE que contendrá la respuesta del host. Si en host aceptó la conexión el campo hResultCode del la estructura asocuada será S_OK, sino será DPNERR_HOSTREJECTEDCONNECTION.

e. Gestión de la sesión Cliente/Servidor

El servidor, como host, es responsable de gestionar el curso de la sesión. Los detalles de esta gestión dependerán de la implementación concreta de la aplicación, pero entre sus responsabilidades mínimas están:

- Mantener un registro con los miembros de la sesión y sus direcciones de red. DirectPlay maneja parte de esta tarea pero la aplicación servidor necesita normalmente manejar mas información de los jugadores que la que proporciona DirectPlay.
- Decidir si un nuevo jugador será añadido a una sesión.
- Proporcionar a los nuevos usuarios el estado de la sesión en el momento de la conexión.

El servidor también cuenta con la posibilidad de eliminar a un jugador de la sesión llamando al método IDirectPlay8Server::DestroyClient.

f. Transcurso de la sesión Cliente/Servidor

En DirectPlay los mensajes sin esencialmente bloques que datos relacionados con el juego que se envían desde los clientes al servidor y viceversa. DirectPlay no especifica nada a cerca del contenido o el formato del bloque de datos, si no que solamente proporciona los mecanismo necesarios para la transmisión de los datos.

Una vez que el juego ha empezado, cada cliente enviará un flujo constante de mensajes al servidor durante el transcurso del juego. El principal propósito de estos mensajes es mantener el estado del juego sincronizado, así cada cliente mostrará el mismo interfaz de usuario.

Para muchos juegos, especialmente para los rápidos, la gestión de los mensajes debe tratarse con mucho cuidado. DirectPlay acelera los mensajes salientes hasta la máxima velocidad a la que pueden ser gestionados por el objetivo. El paso eficiente de mensajes se discute en el apartado 'Aspectos básicos de la comunicación'.

El servidor utiliza el método IDirectPlay8Server::SendTo para la transmisión de datos a algún cliente dentro de la sesión. El mensaje se puede mandar a un cliente concreto o bien a un grupo de ellos según se especifique en el primer parámetro del método. Véase el siguiente fragmento de código como ejemplo:

```
if (dpnidObjetivo==NULL){
        dpServer->SendTo( DPNID_ALL_PLAYERS_GROUP, &bufferDesc, 1,
                0, NULL, &hAsync, DPNSEND_NOLOOPBACK );
        }
        else {
        dpServer->SendTo( dpnidObjetivo, &bufferDesc, 1,
                0, NULL, &hAsync, DPNSEND_NOLOOPBACK );
        }
```

dpnidObjetivo es del tipo DPNID y si su valor es nulo el mensaje se envía a todos los clientes que están conectados a la sesión (DPNID_ALL_PLAYERS_GROUP),

en otro caso al cliente con la DPNID especificada. Después de hacer una llamada a este método el cliente recibirá un mensaje de tipo DPN_MSGID_RECIEVE para indicarle que el mensaje se ha procesado.

Por otra parte, el cliente usa un método muy parecido para enviar mensajes al servidor. Hace uso del método IDirectPlay8Client::Send. En este caso no es necesario especificar el destinatario del mensaje ya que los clientes solo pueden comunicarse con el servidor. Después de la llamada a este método, el servidor recibirá un mensaje de tipo DPN_MSGID_RECIEVE al igual que ocurre en el cliente.

DirectPlay ofrece además la posibilidad de organizar los jugadores en grupos. Estos sirven esencialmente para simplificar el sistema de mensajes. Para crear un grupo hay que usar el método IDirectPlay8Server::CreateGroup y el gestor de mensajes recibirá mensaje de tipo DPN_MSGID_CREATE_GROUP con los detalles, incluido el identificador del grupo que se usará para enviar mensajes al grupo. Una vez que el grupo se ha creado enviar mensajes al mismo es tan fácil como llamara al método IDirectPlay8Server::SendTo especificando como primer parámetro el identificador del grupo.

Existen a su vez métodos para borrar clientes de un grupo: IDirectPlay8Server::RemovePlayerFromGroup y finalmente cuando el grupo no se necesita nunca más se puede destruir con IDirectPlay8Server::DestroyGroup.

g. Abandonar la sesión y finalizar

Un cliente puede abandonar fácilmente la sesión llamando al método IDirectPlay8Client::Close y el servidor será notificado de dicha eliminación mediante la recepción de un mensaje de tipo DPN_MSGID_DESTROY_PLAYER.

El servidor por su parte para finalizar la sesión llamará al método IDIrectPlay8Server::Close y hace que los clientes sean notificados de esta finalización mediante la recepción del mensaje DPN_MSGID_TERMINATE_SESSION. Una vez hecho esto el servidor irá recibiendo sucesivos mensajes del tipo DPN_MSGID_DESTROY_PLAYER procedentes de cada uno de los clientes conectados a la sesión. Una vez que la llamada a Close regresa la aplicación se puede cerrar de forma segura.

B. Baño de Sangre

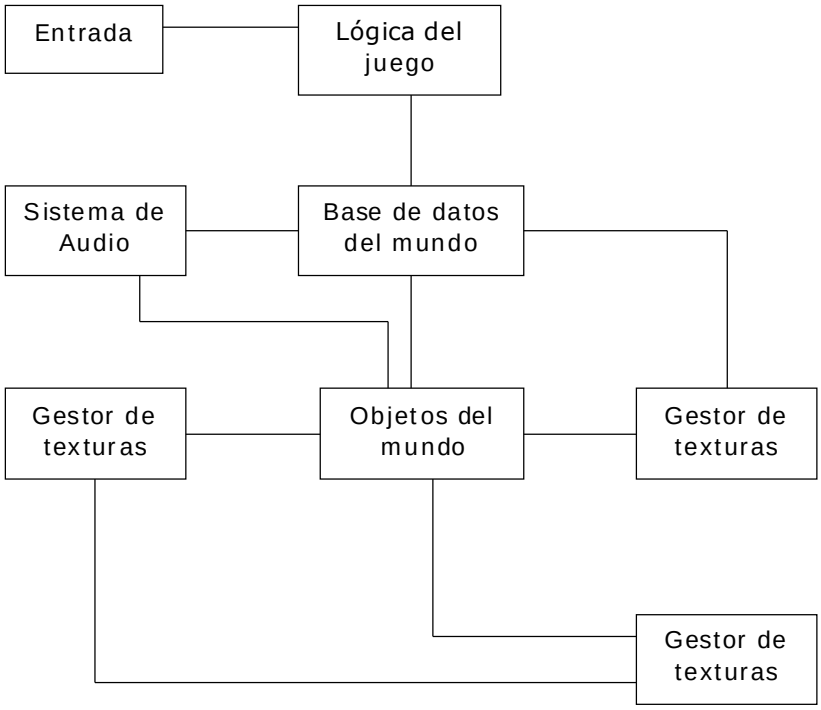
En esta sección se discutirán solamente los aspectos más relevantes juego. Aspectos de diseño que han afectado a al ampliación del juego para convertirlo en multi-jugador.

Veamos por separado los siguientes puntos:

- El motor gráfico
- El ciclo de juego
- La cámara y el mundo
- El sistema de partículas

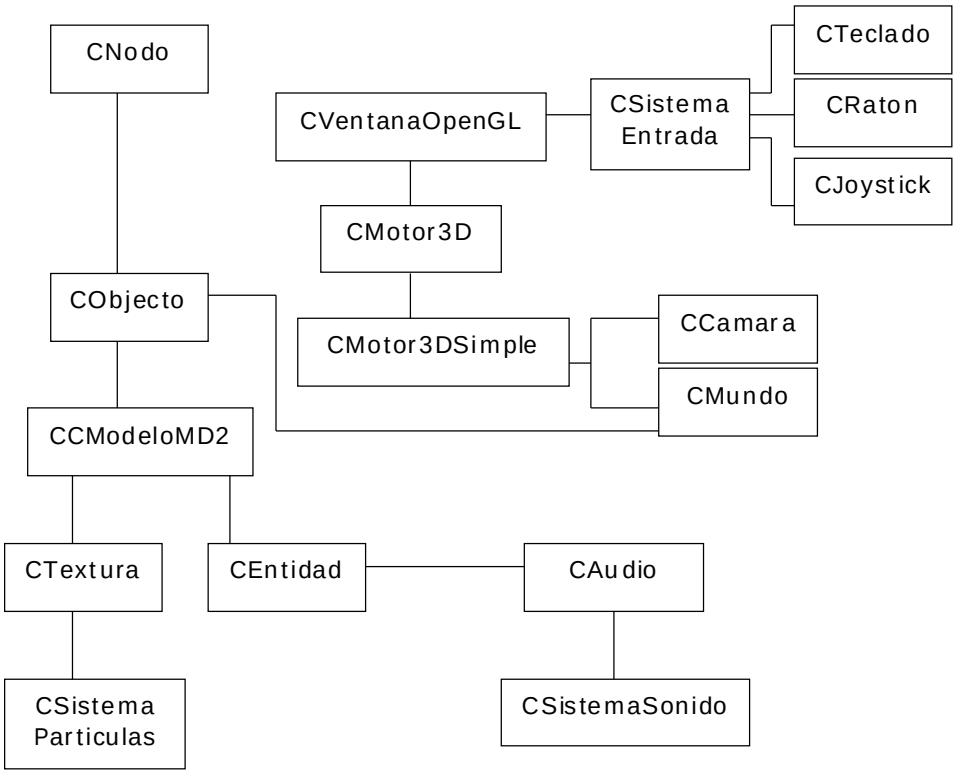
1. El motor gráfico

El motor gráfico se implementa en la clase CMotor3Dsimple, clase que hereda a su vez de CMotor3D. La estructura del motor se puede descomponer en distintos sistemas como indica la siguiente figura:



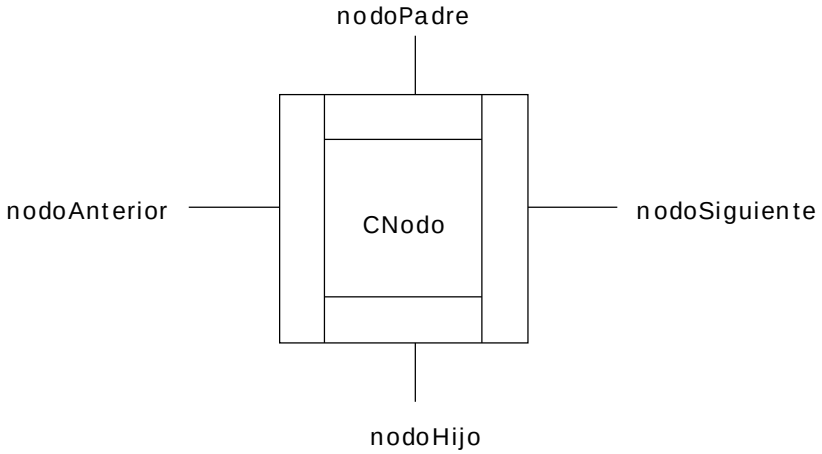
Esencialmente, el motor recibe su entrada a través del subsistema Entrada, entonces envía un mensaje la Lógica del Juego, que tratará pertinentemente el mensaje y ejecutará un ciclo de juego. Durante el ciclo de juego la Lógica responde la mensaje, realiza cualquier cálculo físico necesario en los objetos del juego, hace la detección de colisiones y responde cargando o destruyendo objetos, moviendo la cámara por el mundo y haciendo sonar el sistema de audio.

Si entramos más en detalle podemos desglosar el motor gráfico en las siguientes clases:

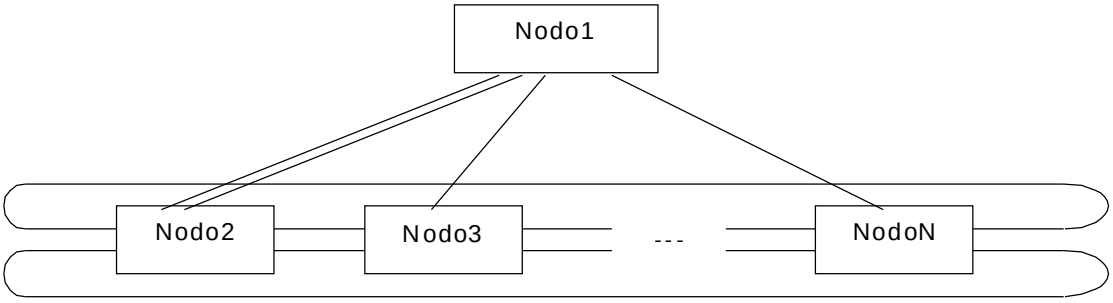


Vamos a detenernos en explicar como se gestionan los objetos desde el motor a través de las clases CNodo y CObjecto.

La clase CNodo representa un único nodo de una lista cíclica doblemente enlazada, que a su vez forma parte de la estructura de un árbol. Esta es la estructura usada para construir la jerarquía de objetos del mundo. Como se puede ver en la siguiente figura CNodo mantiene enlaces al nodo padre, nodo hijo y a los nodos anterior y posterior.

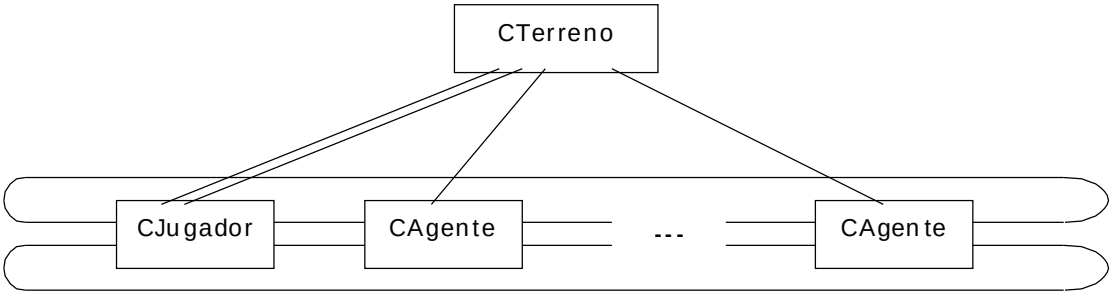


Uniendo varios de los nodos de la forma que se dijo antes se obtiene una estructura similar a la siguiente:



La clase CNodo incluye métodos para añadir y eliminar nodos sin importar si el nodo tiene padre o hijos.

La clase CObjecto representa la mínima entidad del motor gráfico y de esta clase hereda CNodo. CObjeto nos permite representar en el motor cualquier cosa que se pueda dibujar, moverse, colisionar e incluso 'pensar'. Precisamente por que CObjeto hereda de CNodo, juntando unos objetos con otros se puede crear una jerarquía de objetos que facilite la gestión de los mismos en el motor. En la adaptación del Baño de Sangre a nuestras necesidades la estructura que emplea el motor para gestionar los objetos del juego es:



El verdadero poder de la clase CObjeto se hace mas evidente cuando hay que renderizar, mover o detectar colisiones en un objeto. La clase incluye los siguientes métodos en la parte pública:

```
void Preparar();

void Animar(float intervaloTiempo);

void ProcesaColision(CObjeto* objetoColision);

void Dibujar(CCamara* camara);
```

y además dispone de un conjunto de funciones virtuales en la parte protegida:

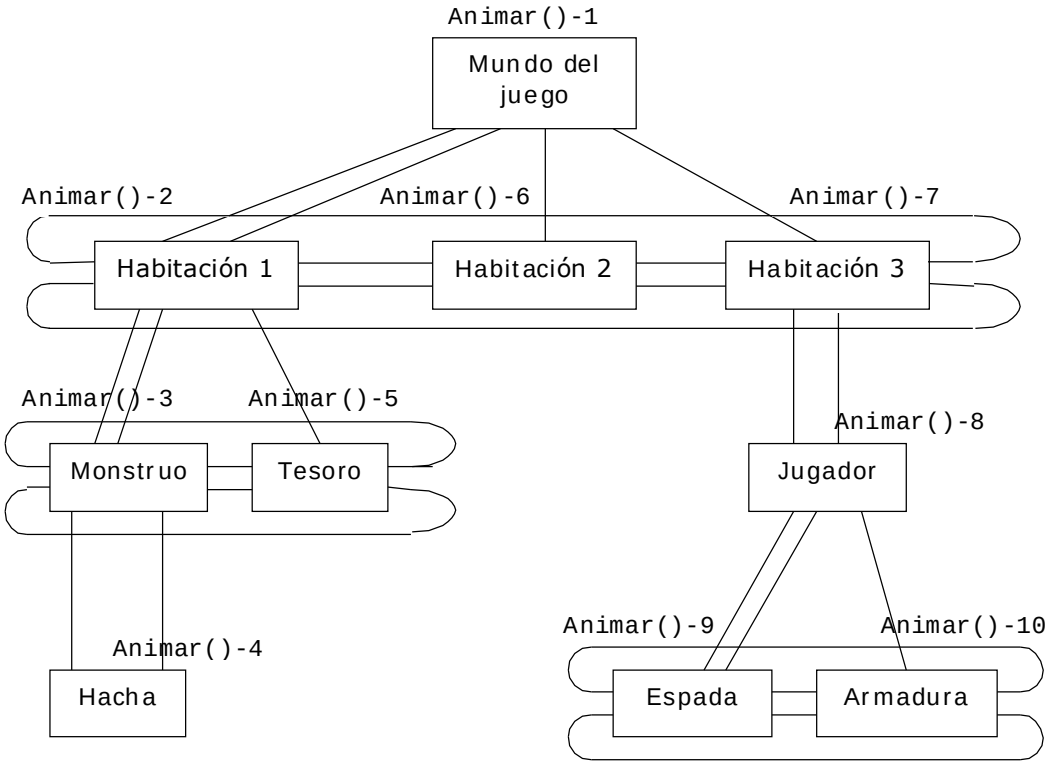
```
virtual void AlAnimar(float intervaloTiempo);

virtual void AlDibujar(CCamara* camara);

virtual void AlColisionar(CObjeto* objetoColision);

virtual void AlPreparar();
```

Normalmente, en Animar()se hacen los cálculos físicos que necesiten los objetos, en Preparar() se ejecuta la inteligencia artificial y los posibles cambios de estado en los objetos, en ProcesaColision() se hace la detección de colisiones de los objetos y en Dibujar().

Porque AlPreparar(), AlAnimar(), AlColisionar() y AlDibujar() son funciones virtuales, necesitan ser redefinidas con el código a la medida para cada objeto hereda de CObjeto. Si tenemos, por ejemplo una clase CEspada que hereda de CObjeto tendrá una implementación del método AlDibujar() diferente que otro objeto de la clase CArmadura (que también hereda de CObjeto).

El uso de funciones virtuales permite una fácil expansión del motor simplemente especializando los métodos de las clases derivadas de CObjeto.

En cada una de las funciones públicas hay una llamada al correspondiente método protegido virtual Al*(), método implementado en cada tipo de objeto particular. Así por ejemplo, la función Animar, anima primero el objeto actual y luego anima todos los objetos hijos y después a sus hermanos haciendo llamadas recursivas a la función Animar. Véase gráficamente el proceso de animación para un árbol de objetos ejemplo:

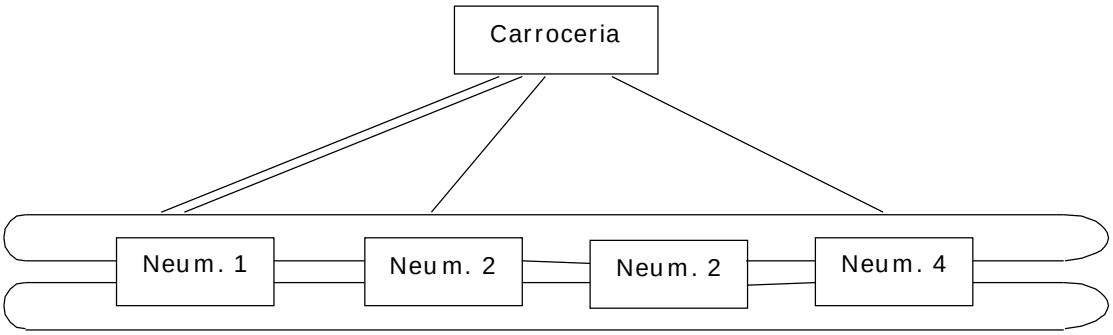


En esta jerarquía de objetos, es sistema de coordenadas de un objeto hijo son relativas a las de su padre. Para dibujar, por ejemplo, un coche y sus cuatro neumáticos una posible solución es crear la carrocería del coche como un objeto y adjuntar cada uno de los cuatro neumáticos a la carrocería. Todo lo que se necesita es especificar la localización de los objetos en relación a la carrocería del coche y la jerarquía de CObjeto se encargará de renderizar la escena completa. Una posible implementación para este ejemplo es la siguiente:

```
CObjeto *carrocería;
CObjeto *neumatico1, *neumatico2, *neumatico3, *neumatico4;
…
```

```
neumatico1->AttachTo(carrocería);
neumatico1->AttachTo(carrocería);
neumatico1->AttachTo(carrocería);
neumatico1->AttachTo(carrocería);
…
carrocería->Dibujar(…);
```

El código anterior produce como resultado la siguiente jerarquía de objetos:



En este punto, entendemos que la estructura de datos usada queda suficientemente explicada y su uso justificado. Ç

Volviendo al diagrama de clases del Baño de Sangre vemos que en lo más alto de la jerarquía, por encima de Motor3D se encuentra la ventana de OpenGL. CVentanaOpenGl, como su nombre indica, crea una ventana de OpenGl y gestiona los mensajes de Windows que se reciben del sistema operativo.

La clase CMotor3DSimple contiene el bucle principal de los mensajes de Windows, es decir, el bucle del juego, además proporciona los métodos necesarios para manejar cualquier entrada que se reciba del Sistema de Entrada. Durante cada paso del bucle, se ejecuta un ciclo de juego y se comprueba si se ha recibido un mensaje para salir de la aplicación. La función del ciclo de juego recibe como parámetro la cantidad de tiempo que ha pasado desde el último ciclo. Este tiempo se usará en todo el motor gráfico para realizar los cálculos físicos y otras funciones basadas en el tiempo.

2. El ciclo de juego

Se puede decir en términos generales que un ciclo de juego conlleva los siguientes pasos:

1. Recoger la entrada
2. Mover el jugador
3. Ejecutar la inteligencia artificial
4. Realizar los cálculos físicos
5. Ejecutar los sonidos
6. Renderizar la escena

Sin embargo, el ciclo de juego del motor simple del Baño de Sangre no es tan claro debido a la jerarquía de objetos que maneja el juego.

El ciclo de Juego debe llamar los métodos de CObjeto Preparar(), Animar() y Dibujar() como se muestra en el siguiente fragmento de código.

```
//-------------------------------------------------------------
// Nombre: CicloJuego(float intervaloTiempo)
// Desc:   Ejecución del ciclo de juego
//-------------------------------------------------------------
void CMotor3D::CicloJuego(float intervaloTiempo) {

      //obtener la cámara.
      CCamara* camara = this->AlObtenerCamara();

      //obtener el mundo.
      CMundo* mundo = this->AlObtenerMundo();

      if(this->usarDirectInput)
            this->CompruebaEntrada(intervaloTiempo);

      //inicializa OpenGL.
      this->AlPreparar();

      //mueve/orienta la cámara.
      camara->Animar(intervaloTiempo);

      //mueve/orienta los objetos.
      mundo->Animar(intervaloTiempo);

      //prepara objetos y comprueba colisiones.
      mundo->Preparar();

      //dibujar objetos.
      mundo->Dibujar(camara);

      //intercambiar búffers.
      SwapBuffers(this->hDC);
}
```

Por debajo de CMotor3D, en la jerarquía de clases, encontramos CMotor3DSimple. Esta clase contiene métodos de respuesta a eventos del teclado como por ejemplo

```
      void AlPresionarTecla(int nVirtKey);
```

Además, contiene la cámara del motor gráfico y los objetos del mundo. Veamos en mas detalle las clases CCamara y CMundo a continuación.

3. La cámara y el mundo

La cámara:

Como es de esperar CCamara define el sistema de visión del motor gráfico y es responsable de determinar como se ven el mundo y sus objetos.

En el Baño de Sangre, la cámara en muy fácil de controlar ya que la mayor parte del tiempo se maneja a través de la entrada recibida por teclado.

La función más importante de CCamara es Animar() que se usa para mover y orientar la cámara. La cámara lleva asociados una velocidad y una aceleración para controlar el movimiento. La orientación se controla a través de los ángulos de giñada y cabeceo, que representan la rotación respecto de los ejes y y x respectivamente. Además se dispone del vector lookAt que representa el vector

hacia donde apunta la cámara. Si por ejemlo la cámara apunta directamente hacia abajo (parte negativa del eje z) entonces el vector lookAt será (0.0,0.0,-1.0).

Se lista a continuación el codigo de la función Animar()

```
void CCamara::Animar(float intervaloTiempo) {

      if((this->anguloRotacionEjeY >= 360.0f) ||
         (this->anguloRotacionEjeY <= -360.0f))
            this->anguloRotacionEjeY = 0.0f;

      if(this->anguloRotacionEjeX > 60.0f)
            this->anguloRotacionEjeX = 60.0f;
      if(this->anguloRotacionEjeX < -60.0f)
            this->anguloRotacionEjeX = -60.0f;

      //calcula funciones trigonométricas.
      float cosenoAnguloRotacionEjeY = cosf(GRADaRAD(this->
            anguloRotacionEjeY));
      float senoAnguloRotacionEjeY = sinf(GRADaRAD(this->
            anguloRotacionEjeY));
      float senoAnguloRotacionEjeX = sinf(GRADaRAD(this->
            anguloRotacionEjeX));

      //la celeridad es la componente z de la velocidad. Indica
      el movimiento adelante/atrás.
      float celeridad = this->velocidad.z * intervaloTiempo;

      //la componente x de la velocidad indica el movimiento
      izquierda/derecha.
      float celeridad2 = this->velocidad.x * intervaloTiempo;

      //límite de celeridad.
      if(celeridad > 15.0)
        celeridad = 15.0;
      if(celeridad2 > 15.0)
        celeridad2 = 15.0;
      if(celeridad < -15.0)
          celeridad = -15.0;
      if(celeridad2 < -15.0)
          celeridad2 = -15.0;

      //rozamiento.
      if(this->velocidad.DevuelveLongitud() > 0.0) {
      CVector temp = this->velocidad.Negacion();
            this->aceleracion = temp.ProductoPorEscalar(1.5);
      }

      CVector temp2 = this->
            aceleracion.ProductoPorEscalar(intervaloTiempo);
      this->velocidad = this->velocidad.Sumar(temp2);

      //calcula la nueva posición de la cámara.
      this->posicion.x = this->posicion.x + cosf(GRADaRAD(this->
            anguloRotacionEjeY + 90.0)) * celeridad2;
      this->posicion.z = this->posicion.z + sinf(GRADaRAD(this->
            anguloRotacionEjeY + 90.0)) * celeridad2;
      this->posicion.x = this->posicion.x +
            cosenoAnguloRotacionEjeY * celeridad;
```

```
this->posicion.z = this->posicion.z +
      senoAnguloRotacionEjeY * celeridad;

//calcula hacia dónde mira la cámara basándose en la nueva
posición.
this->mirada.x = this->posicion.x +
      cosenoAnguloRotacionEjeY;
this->mirada.y = this->posicion.y +
      senoAnguloRotacionEjeX;
this->mirada.z = this->posicion.z +
      senoAnguloRotacionEjeY;

//establece la nueva posición y orientación de la cámara.
gluLookAt(this->posicion.x, this->posicion.y,
      this->posicion.z, this->mirada.x, this->mirada.y,
      this-> mirada.z, 0.0, 1.0, 0.0);
}
```

Como se puede observar, al final del método se establece la nueva posición y orientación de la cámara a traves de la función de OpenGl gluLookAt(). Como se dijo en el apartado del ciclo de juego, la cámara es el primer objeto que usa la función Animar(), la razón es la llamada a gluLookAt(). Antes de empezar a dibujar el mundo, se necesita primeramente establecer el punto de vista de la cámara así OpenGl sabe qué dibujar y como hacerlo.

El mundo:

Como se vio anteriormente, la función CicloJuego() de CMotro3D usa la clase CMundo para interactuar con todos los objetos del juego a través de las funciones Preparar(), Animar() y Dibujar(). Normalmente, CMundo almacena los distintos niveles del objetos del juego así como el sistema de audio CSistemaAudio. El Baño de Sangre incluye en esta clase un objeto de la clase CTerreno que representa el mundo que los jugadores explorarán y es la raíz de la jerarquía de objetos como se vio en uno de los primeros diagramas.

4. El sistema de partículas

La clase CSistemaParticulas se usa para producir algunos efectos como explosiones, humo, fuego o niebla. El principal efecto de CExplosion, por ejemplo, deriva de CSistemaParticulas. Las funciones virtuales que proporciona para Actualizar(), Renderizar() y InicializarSistema() deben rescribirse en CExplosión, incluyendo los cálculos necesarios para conseguir el efecto deseado.

C. Open Gl para el entorno gráfico

La representación gráfica del terreno donde se desarrolla el juego se ha generado con un algoritmo de mapa de altura normalizado. Primero se establecen unas dimensiones para el terreno, que formarán un cuadrado. Se dan unos valores iniciales de altura a las cuatro esquinas, se calcula el centro del cuadrado y se hace la media de altura del centro respecto con las cuatro esquinas. Al punto central, se le asigna esa altura media calculada más un valor aleatorio que hará que ese punto sea más alto o más bajo del ya calculado. Con ese valor aleatorio que se suma, se puede controlar la rugosidad que tendrá el terreno generado.

Después de generar todas las alturas para todos los puntos, el terreno se normaliza, de forma que las alturas tengan un valor entre 0 y 1. El terreno se ha formado mediante cuadrados divididos en diagonal. Para darle un aspecto más realista se puede aplicar un filtro de erosión. A la hora de dibujar el terreno se hace con triángulos concatenados entre sí (GL_TRIANGLE_STRIP) y se aplica una textura sobre toda la superficie del terreno.

Cuando un personaje se mueve por el terreno, no lo hace de vértice en vértice de cada casilla que lo forma. Por eso, para calcular la altura del terreno en un punto determinado, se localiza la casilla en el que cae ese punto, y se hace la media de las alturas de los cuatro vértices que lo forman. Luego se modifica esa altura ligeramente teniendo en cuenta la proyección de la cámara

Aspecto del terreno de la aplicación

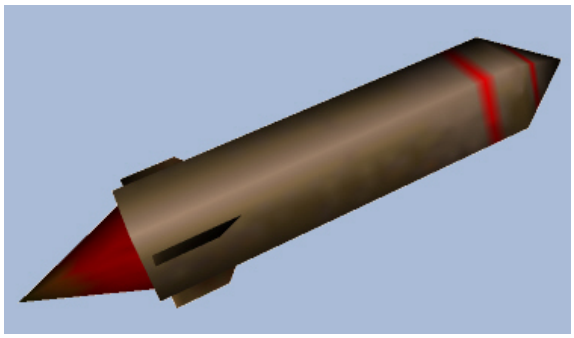
Los personajes del juego están en formato MD2. Este formato fue creado por idSoftware® para su uso en el juego Quake2™. Un archivo md2 contiene la geometría del modelo, información de los frames de animación, nombres de los archivos de skins (pieles), coordenadas de texturas.

Para la animación del personaje, el archivo md2 tiene marcados un número de frames con el personaje en las distintas posiciones que forman la animación. Por ejemplo, la secuencia de disparo de un modelo, está formada por 4 frames. Para generar una animación fluida del personaje, se hace una interpolación entre un frame y el siguiente de una misma animación. Así se

Animación de disparo de un modelo md2

Los personajes md2 incluidos en la aplicación han sido descargados de internet con derecho de los creadores a que sean usados libremente. No obstante, en la aplicación se incluyen todos los archivos que vienen acompañando a los modelos y texturas.

Los disparos son también modelos md2. Tienen el mismo comportamiento que un modelo de un personaje, excepto que en este caso, el modelo sólo tiene dos frames de animación, que afecta a la longitud de la estela de fuego que deja el cohete.



Modelo md2 del cohete

Para las explosiones de los cohetes se ha utilizado un sistema de partículas. Un sistema de partículas se encarga de crear pequeñas partículas a las que se les aplican distintas leyes físicas para producir efectos como explosiones, lluvia o fuego. En nuestra aplicación, al producirse el impacto de un cohete sobre cualquier superficie éste produce una explosión. Una explosión cercana a la cámara se representa con 200 partículas, mientras que si la explosión se encuentra lejos de la cámara, se reduce a 50 partículas, ya que apenas se aprecia diferencia y optimizamos el consumo de recursos.

Proceso de explosión de un cohete

D. Teoria de A gentes

Durante los últimos años la tendencia general de los desarrolladores de software ha dado un giro abismal. Si antes la tendencia era la construcción de software que desempeñaba una gran variedad de tareas y que se implementaba con millones de líneas de código, ahora se tiende al desarrollo de programas modulares donde cada módulo define y desempeña una tarea muy concreta. En este sentido los agentes software son la última innovación. Generalmente hablando, se puede decir que un agente software cubre los siguientes aspectos:

- Proporciona uno o más servicios a otros agentes para que puedan usarlos bajo unas condiciones especificadas
- Incluye la descripción de los servicios que ofrece. Estos servicios pueden ser accedidos y entendidos por otros agentes.
- Tiene la habilidad de funcionar autónomamente, sin requerir instrucciones precisas de un humano
- Dispone de habilidad para interactuar con otros agentes.

Nótese que no todos los agentes software deben presentar las características anteriores, sin embargo cualquier paradigma de programación basado en agentes debe poder crear agentes con alguna o todas las propiedades anteriores.

Por otro lado, la plataforma que soporta la interacción multi-agente debe proporcionar un servicio de 'paginas amarillas' para permitir que los agentes se encuentren y vean los servicios que cada tipo de agente ofrece.

Algunas de estas ideas en las que se basa la teoría de agentes han sido usadas en el desarrollo del proyecto. Así los clientes del juego se ajustan bastante a este paradigma de programación. Cuando los clientes se conectan al servidor, se crea un agente correspondiente a dicho cliente. El agente dispone de una inteligencia artificial que le permite actuar libremente sin la necesidad de una interacción humana y el servidor mantiene unas páginas amarillas con la información relevante de cada agente conectado. Cuando un agente necesita interactuar con otro, hace usa de la información almacenada en las páginas amarillas. Así mismo cuando un agente modifica su estado, las paginas amarillas del servidor deben ser notificadas de este cambio. Esta es la forma de mantener la coherencia y la sincronización entre todas las partes conectadas al servidor. Otros objetos del juego han sido diseñados siguiendo esta filosofía, como por ejemplo los disparos.

IV. Especificación detallada

A. Intelige ncia artificial de los agentes

La Inteligencia Artificial define el comportamiento de los agentes y como éstos van a interactuar con su entorno.

Para describir la IA hemos diseñado una máquina de estados, donde cada estado representa una forma de comportamiento. Dependiendo del estado en el que se encuentre el agente, actuará de una manera u otra, interactuando con su entorno de distintas formas. Así logramos implantar en los agentes una "personalidad", de fácil modificación y sobretodo de fácil ampliación. Nos basta con modificar alguno de los estados de la máquina de estados para cambiar el comportamiento y la dinámica del agente, o con añadirle más estados para complicar la interacción del agente con su entorno.

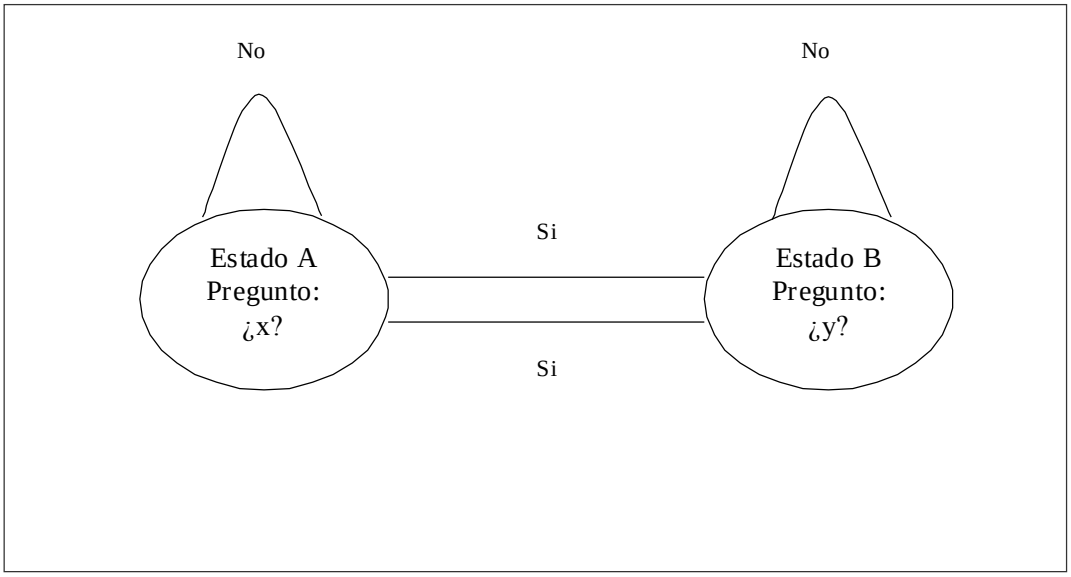
Al estar la IA implantada en el agente, y al estar utilizando una topología cliente-servidor en nuestra estructura del sistema, nuestra IA va estrechamente ligada al paso de mensajes que se produce entre un cliente y su servidor.

Por lo tanto no sólo es importante definir una buena máquina de estados para una óptima IA, sino que es igual de importante o más, realizar una buena gestión del paso de mensajes que se produce entre el cliente y el servidor. No nos sirve de nada diseñar una máquina de estados compleja con un elevado número de estados y una complicada relación entre los mismos que describa un comportamiento perfecto de un agente, si luego la comunicación con el servidor es lenta o ineficaz.

Los mensajes representan las preguntas que el cliente realiza al servidor, o las respuestas del servidor a una pregunta del cliente.

Estas preguntas y sus respuestas definen la transición de estados de la máquina de estados.

En el diagrama siguiente se observa que estando en el estado A, el agente realiza una pregunta al servidor, si la respuesta a esta pregunta es si, el agente realiza la transición A->B. En cambio, si la respuesta es no, el agente permanece en el mismo estado. En el estado B podríamos definir un comportamiento análogo.

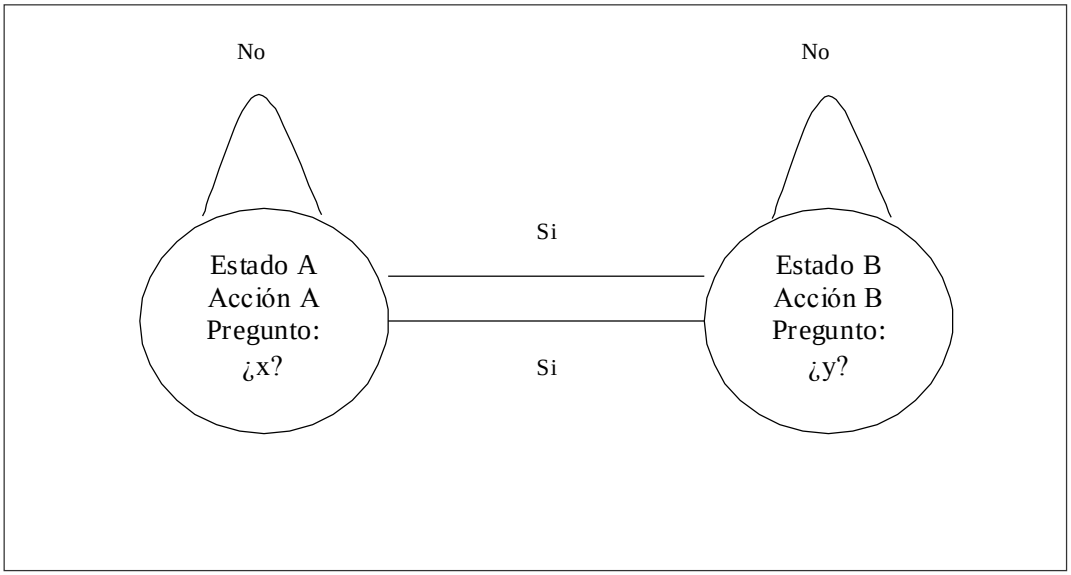


Para conseguir una comunicación fluida entre un cliente y su servidor, es necesario que las preguntas que el cliente realice al servidor sean de fácil y rápida gestión. Es decir, que el tiempo de respuesta del servidor sea lo más bajo posible. Teniendo en cuenta, que al servidor le llegan preguntas de un número elevado de clientes, y que cada cliente le puede realizar un número elevado de preguntas, si tardase mucho en responder a esas preguntas que le realizan los clientes, la comunicación global del sistema sería un caos. Y se podría dar el caso de que la respuesta que le de el servidor a un cliente no sea la correcta ya que esta repuesta podría depender de otra información que le debiera llegar de otro cliente, y como esta información no le ha llegado debido a la saturación de la red, el servidor responde basándose en información no actualizada y dando al cliente información incorrecta pudiendo inferir en una transición de estado no deseada.

Pero no toda la IA se basa en el paso de mensajes que simbolizan preguntas que el cliente realiza al servidor, y las respuestas del mismo, de cara a la transición de estados en la máquina de estados. También en estos estados hay que realizar acciones. Si yo en un estado sólo pregunto al servidor y en base a la respuesta que me llega del servidor paso o no a otro estado, en definitiva lo único que estoy haciendo es pasar de un estado a otro, y no estoy definiendo ningún comportamiento del agente, que es lo se pretende realizar en el diseño de la IA.

Es en estas acciones por tanto donde realmente se define el comportamiento del agente.

A la hora de diseñar la máquina de estados tenemos que identificar claramente las acciones que queremos que pueda realizar nuestro agente, y asignarle un estado a cada acción. Así mantenemos una filosofía de construcción de la máquina de estados muy sencilla, y nos permite construir de manera muy intuitiva una IA de relativa complejidad. En el diagrama siguiente observamos que ahora un estado ya no sólo realiza una pregunta al servidor, sino que también realiza una acción.

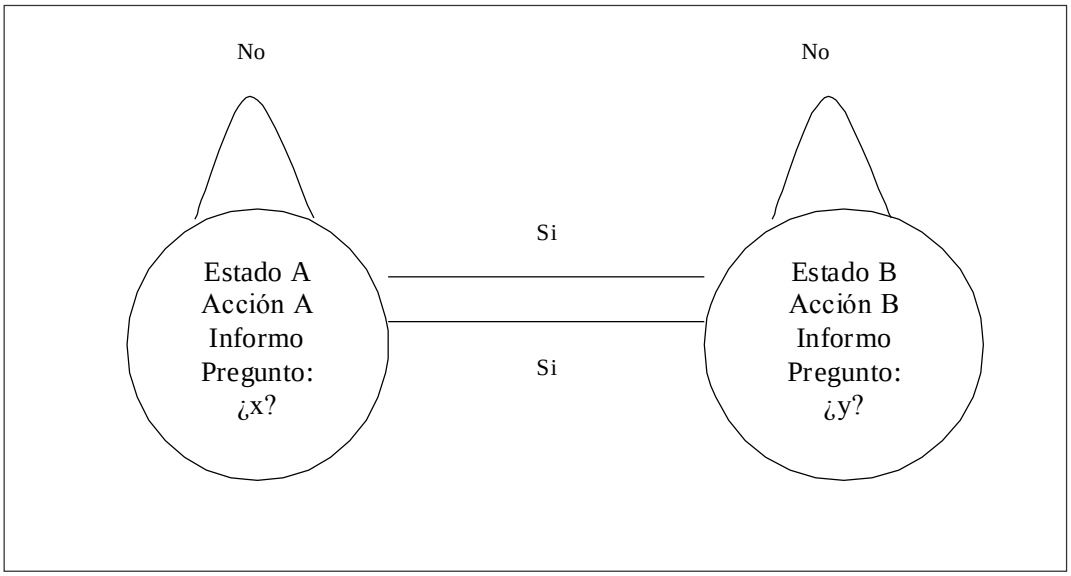


Como vemos en el diagrama la acción se realiza antes que la pregunta. Esto tiene que ser así puesto que la finalidad del estado de la máquina de estados, es ejecutar su acción, ya sea moverse, disparar, etc… y si la pregunta estuviera antes que la acción, podría darse el caso de obtenerse la respuesta antes de realizar la acción y producirse consecuentemente un cambio de estado sin haberla realizado. Nos evitamos este posible problema, colocando la acción antes que la pregunta.

Aún nos quedaría un último paso para finalizar la descripción de un estado de la máquina de estados. Realizando la acción por si sola, y después la pregunta el cliente seguiría desligado del servidor. El servidor seguiría sin tener información acerca de los clientes que están conectados a él. Sólo recibe preguntas, pero no dispone de información para responderlas. Ya que los clientes no informan al servidor del resultado de sus acciones. Si un cliente representase a un robot, y el robot se encuentra en un estado cuya acción es moverse, pero este robot no informa al servidor de sus movimientos, es como si para el servidor el robot no se moviese. Y para cualquier otro cliente que preguntase sobre la posición del robot, el servidor no estaría en disposición de responder. Con lo cual, el cliente debe informar al servidor del resultado de sus acciones, para que el servidor las almacene en una base de conocimiento. Y es de esta base de conocimiento de la cual se sirve para responder a las preguntas que los clientes le realizan.

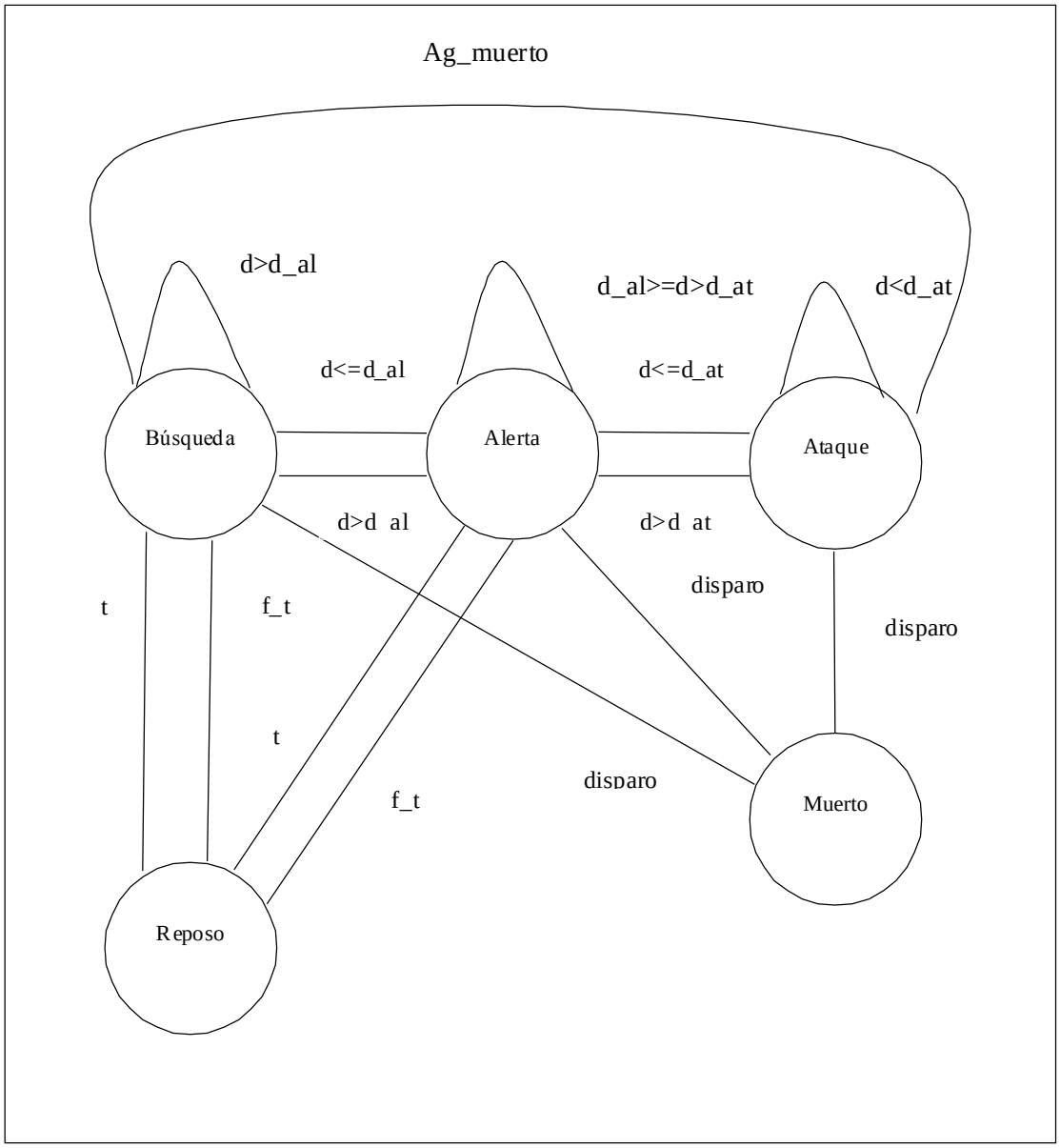
La forma de informar al servidor sobre las acciones de los clientes, es mediante mensajes. Estos mensajes no simbolizan preguntas, puesto que no requieren respuesta. Simplemente informan de las acciones de los clientes.

El diagrama definitivo que representa la estructura de la IA que hemos desarrollado en nuestro proyecto es el siguiente:



1. Máquina de estados del Agente Amigo y Enemigo

Como vemos en el diagrama consta de 5 estados, estado de Búsqueda, Reposo, Alerta, Ataque y muerto. La relación entre los mismos es la que se muestra a continuación:



donde:

d: distancia de un miembro de otro equipo.
d_al: distancia de alerta.
d_at: distancia de ataque.

t: tarea[1].
f_t: fin de tarea.
disparo: recibe disparo.
Ag_muerto: el agente objetivo está muerto.

2. Definición de los estados

Búsqueda:

En este estado el agente se mueve aleatoriamente por el terreno. Por lo tanto la acción que se realiza es la de un movimiento aleatorio. Más adelante se explicará cómo se realiza este movimiento aleatorio. Tras realizar la acción se le envía al servidor un mensaje de actualización de posición del agente, para que el servidor la actualice en la base de datos. En este estado se le pregunta al servidor acerca de los posibles agentes del otro equipo que están dentro del rango de visión de este agente, este mensaje se explica en la sección de mensajes del sistema. Si esta distancia es menor que la distancia prefijada para pasar al estado de alerta, se pasa. En otro caso se permanece en estado de búsqueda.

Alerta:

En el estado de alerta, el movimiento del agente ya no es aleatorio, sino que se mueve hacia el agente del otro equipo que se ha detectado que está dentro del perímetro de alerta. Por lo tanto la acción que se realiza es la de un movimiento controlado. Tras este movimiento se le envía al servidor un mensaje de actualización de posición para que el servidor actualice la posición del agente en la base de datos. Después se pregunta al servidor por la posición del agente que hemos detectado que está dentro del perímetro de alerta. En caso de que esté lo suficientemente cerca como para pasar al estado de ataque, es decir que la distancia que los separe sea menor que la distancia prefijada para pasar al estado de ataque, se realiza la transición al estado de ataque. En otro caso se permanece en estado de alerta.

Ataque:

En este estado el agente ataca al agente del otro equipo que lleva persiguiendo, y que ahora se encuentra dentro del perímetro de ataque. Por lo tanto la acción que se realiza en el estado de ataque no es de un movimiento, sino que es un disparo. Se dispara al agente objetivo. El disparo consiste en un mensaje que se le envía al servidor indicándole quien es el que efectúa el disparo, desde dónde y a dónde se está disparando. Este mensaje se explica detalladamente en la sección de mensajes del sistema. Tras enviar el mensaje de disparo, se realiza una pregunta al servidor sobre la posición del agente a quien estoy atacando. En caso de que la distancia que separa a este agente de su objetivo sea mayor que la distancia prefijada de ataque, se pasa automáticamente al estado de alerta. En otro caso permanezco en este estado. Pero estando en este estado además, nos interesa otra información que nos llega del servidor. Nos interesa saber si el agente objetivo está muerto o no. Cuando un agente muere el servidor informa a todos los clientes que puedan estar conectados. Es en el estado de ataque donde esta información que recibimos del servidor nos puede interesar. Si

[1] Consiste en un intervalo de tiempo en el que el cliente está ocupado procesando información que le llega del servidor, ya sea una lista de agentes del equipo contrario, o recalculando su trayectoria tras una colisión. Es necesario entrar en este estado para cesar el envío de preguntas durante este periodo de proceso.

nuestro agente objetivo está muerto pasamos automáticamente al estado de búsqueda.

Muerto:

En este estado el agente no realiza ninguna acción, ni ninguna pregunta.

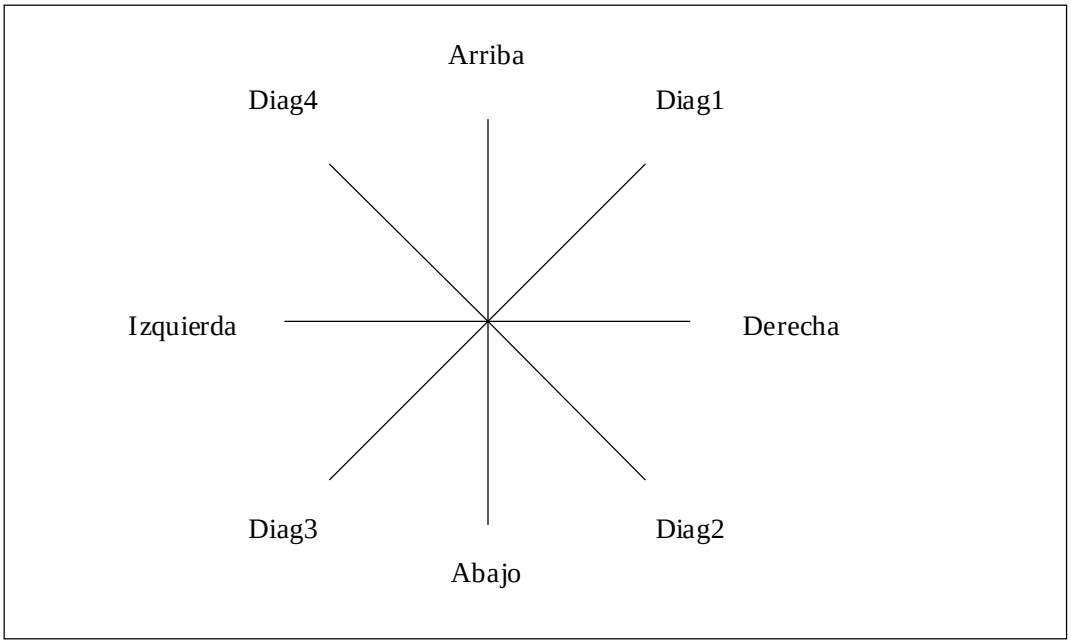
Reposo:

En este estado el agente no realiza ninguna acción, ni ninguna pregunta.

3. El movimiento aleatorio :

En el estado de búsqueda el agente realiza un movimiento aleatorio a la hora de desplazarse por el terreno. Ahora explicaremos cómo hacemos para que el movimiento aleatorio sea óptimo.

Lo primero que definimos son 8 direcciones tal como se muestran en el diagrama. Las elegimos así para cubrir todo el rango de giro. Es sobre estas direcciones donde realizaremos una selección aleatoria.



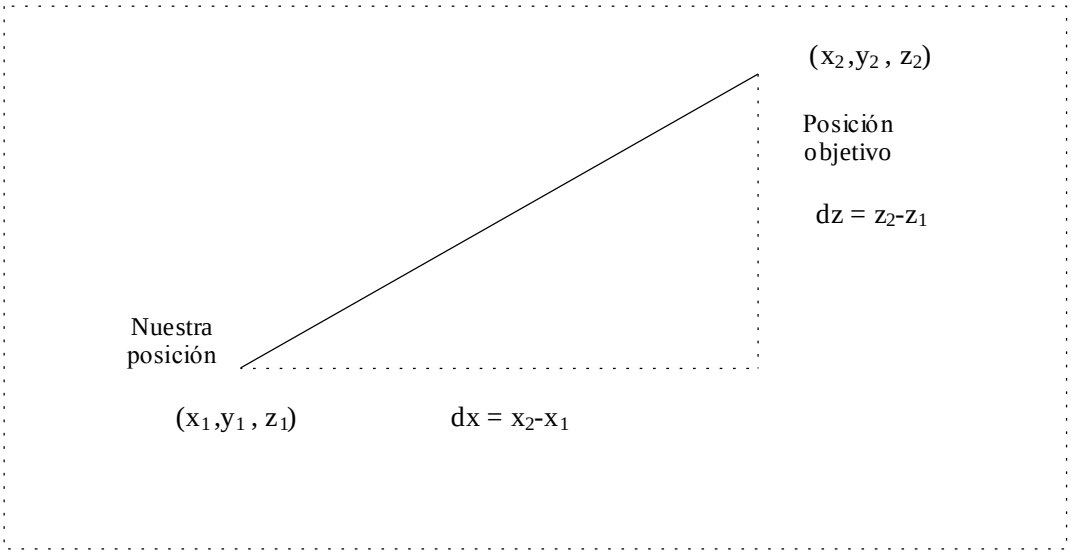
Para evitar que el agente tome giros bruscos, se tiene en cuenta la dirección en la que viaja. Teniendo en cuenta esta dirección, el agente sólo puede decidir cambiar a alguna de las 2 direcciones contiguas. Es decir, si el agente esta viajando en la dirección Arriba, sólo puede cambiar a Diag1 o a Diag4. Con esto conseguimos un movimiento bastante armónico del agente, con giros suaves y nítidas trayectorias.

Además para que no haya muchos cambios de trayectoria, ya que eso perjudicaría al objetivo del estado de búsqueda, que no es más que buscar agentes del equipo contrario, y eliminarlos. Sólo se producen estos cambios de trayectoria en un 10% de las decisiones de cambio. Así el agente recorre y bate más terreno, aumentando las posibilidades de encontrarse con un agente del equipo contrario, y eliminarlo.

Todas las direcciones están normalizadas, así avanzamos lo mismo independientemente de la dirección en la que viajemos. La velocidad del agente la controlamos por medio de una variable. Multiplicando el valor de esta variable por el módulo del vector dirección, que en todas las direcciones al estar normalizado vale 1, obtenemos el vector, que nos indicaría el avance a realizar. Sumamos este avance a la posición actual del agente, y así calculamos la nueva posición del agente. Y es esta posición la que le mandamos por medio del mensaje de actualizar posición al servidor para que actualice su base de datos. Más adelante mostramos el método que realiza estos cálculos, y ahí se puede observar detalladamente lo anteriormente descrito.

4. El movimiento de persecución

En la situación de estar en el estado de Alerta, para calcular la dirección se tiene en cuenta la posición del agente que se ha seleccionado como objetivo. En la gráfica se muestra claramente cómo se realizan estos cálculos.



Obtenemos los valores dx y dz. Y con ellos ya tendríamos el vector de dirección hacia nuestro objetivo. Normalizamos el vector para garantizar que el avance sea controlado. Una vez obtenido procedemos de igual manera que antes.

El código donde realizamos el cambio de estado, y procesamos la Inteligencia Artificial calculando las nuevas posiciones, lo mostramos a continuación.

```
//-----------------------------------------------------------------------
// Nombre: cambiaEstado()
// Desc:   Método que realiza la transición de estados
//-----------------------------------------------------------------------
void IA::cambiaEstado()
{
    if(datos.vida < 0)
       datos.estado = EST_MUERTO;
     else
```

```
{
  switch(datos.estado)
  {
        case EST_BUSQUEDA:
        {
           int pos = 0;
           while (pos<MAX_JUGADORES)
            {
              if((lista_otros_jugadores[pos].equipo==!datos.equipo)
               &&
               (lista_otros_jugadores[pos].datos_validos))
               {
                    int distanciaJug =
                           distancia(lista_otros_jugadores[pos]);

                    lista_otros_jugadores[pos].distancia =
                           distanciaJug;

                   if(distanciaJug < DISTANCIA_ALERTA)
                    {
                       datos.estado = EST_ALERTA;
                       datos.indiceJugadorObjetivo = pos;
                      datos.dpnidJugadorObjetivo =
                           lista_otros_jugadores[pos].dpnidPlayer;
                       MensajeActualizaEstado();
                       break;
                    }
                }
                pos++;
                }
           }break;
         case EST_ALERTA:
         {
            int distanciaJug = distancia
                (lista_otros_jugadores[datos.indiceJugadorObjetivo])
                ;
            lista_otros_jugadores[datos.indiceJugadorObjetivo].
                distancia = distanciaJug;
            if(distanciaJug <DISTANCIA_ATAQUE)
            {
                     datos.estado = EST_ATAQUE;
                     contadorDisparo = 0;
                     MensajeActualizaEstado();
             }
             else if (distanciaJug > DISTANCIA_ALERTA)
             {
                   datos.estado = EST_BUSQUEDA;
                   MensajeActualizaEstado();
             }
         }break;


        case EST_ATAQUE:
        {
           if (
                lista_otros_jugadores[datos.indiceJugadorObjetivo
                ].muerto)
            {
               datos.estado = EST_BUSQUEDA;
               MensajeActualizaEstado();
            }
```

```
               else if(( distancia(
                    lista_otros_jugadores[datos.indiceJugadorObjetivo])
                    > DISTANCIA_ATAQUE))
                 {
                     datos.estado = EST_ALERTA;
                     MensajeActualizaEstado();
                  }
            }break;
               }
      }
     }

//---------------------------------------------------------------------
// Nombre: ProcesaIA()
// Desc:   Algoritmo de la IA
//---------------------------------------------------------------------
void IA::ProcesaIA()
{
      int opcion;
       switch (datos.estado)
            {
            case EST_BUSQUEDA:
             {
                  opcion = rand()%10;
                  if (opcion<=8)
                  {
                        //Seguimos por donde vamos
                        datos.posicion.x +=
                                    incremento*datos.direccion.x;
                        datos.posicion.y +=
                                    incremento*datos.direccion.y;
                        datos.posicion.z +=
                                    incremento*datos.direccion.z;

                        MensajeActualizaPosicion();
                        MensajePideAlrededor();
                  }
                  else
                  {
                    switch(dirMov)
                    {
                      case ARRIBA:
                     {
                       opcion = rand()%2;
                       if(opcion == 0)
                      {
                        dirMov = DIAG1;
                        datos.direccion.x = 1;
                        datos.direccion.z = 1;
                        datos.direccion.y = 0;
                        datos.direccion =
                           datos.direccion.DevuelveVectorUnitario();
                      }



                     else
                                    {
                        dirMov = DIAG4;
                        datos.direccion.x = -1;
                        datos.direccion.z = 1;
```

```
            datos.direccion.y = 0;
            datos.direccion =
               datos.direccion.DevuelveVectorUnitario();
      }
      datos.posicion.x += incremento*datos.direccion.x;
      datos.posicion.y += incremento*datos.direccion.y;
      datos.posicion.z += incremento*datos.direccion.z;
       }break;
case DIAG1:{
       opcion = rand()%2;
       if(opcion == 0)
       {
      dirMov = DECHA;
      datos.direccion.x = 1;
      datos.direccion.z = 0;
      datos.direccion.y = 0;
       }
      else
     {
      dirMov = ARRIBA;
      datos.direccion.x = 0;
      datos.direccion.z = 1;
      datos.direccion.y = 0;
     }
    datos.posicion.x += incremento*datos.direccion.x;
    datos.posicion.y += incremento*datos.direccion.y;
    datos.posicion.z += incremento*datos.direccion.z;
  }break;
case DECHA:{
    opcion = rand()%2;
    if(opcion == 0)
    {
      dirMov = DIAG2;
      datos.direccion.x = 1;
      datos.direccion.z = -1;
      datos.direccion.y = 0;
      datos.direccion =
            datos.direccion.DevuelveVectorUnitario();
    }
    else
    {
      dirMov = DIAG1;
      datos.direccion.x = 1;
      datos.direccion.z = 1;
      datos.direccion.y = 0;
      datos.direccion =
          datos.direccion.DevuelveVectorUnitario();
    }
  datos.posicion.x += incremento*datos.direccion.x;
  datos.posicion.y += incremento*datos.direccion.y;
  datos.posicion.z += incremento*datos.direccion.z;
}break;
 case DIAG2:{
opcion = rand()%2;
if(opcion == 0)
{
      dirMov = ABAJO;
      datos.direccion.x = 0;
      datos.direccion.z = -1;
      datos.direccion.y = 0;
        }
```

```
       else
       {
      dirMov = DECHA;
      datos.direccion.x = 1;
      datos.direccion.z = 0;
      datos.direccion.y = 0;
       }
     datos.posicion.x += incremento*datos.direccion.x;
     datos.posicion.y += incremento*datos.direccion.y;
     datos.posicion.z += incremento*datos.direccion.z;
 }break;
case ABAJO:{
     opcion = rand()%2;
     if(opcion == 0)
     {
      dirMov = DIAG3;
      datos.direccion.x = -1;
      datos.direccion.z = -1;
      datos.direccion.y = 0;
      datos.direccion =
            datos.direccion.DevuelveVectorUnitario();
    }
   else
   {
      dirMov = DIAG2;
      datos.direccion.x = 1;
      datos.direccion.z = -1;
      datos.direccion.y = 0;
      datos.direccion =
            datos.direccion.DevuelveVectorUnitario();
   }
datos.posicion.x += incremento*datos.direccion.x;
datos.posicion.y += incremento*datos.direccion.y;
datos.posicion.z += incremento*datos.direccion.z;
}break;
case DIAG3:{
   opcion = rand()%2;
   if(opcion == 0)
   {
      dirMov = IZDA;
      datos.direccion.x = -1;
      datos.direccion.z = 0;
      datos.direccion.y = 0;
    }
   else
   {
      dirMov = ABAJO;
      datos.direccion.x = 0;
      datos.direccion.z = -1;
      datos.direccion.y = 0;
           }
datos.posicion.x += incremento*datos.direccion.x;
datos.posicion.y += incremento*datos.direccion.y;
datos.posicion.z += incremento*datos.direccion.z;
}break;
case IZDA:{
   opcion = rand()%2;
   if(opcion == 0)
   {
      dirMov = DIAG4;
      datos.direccion.x = -1;
```

```
                datos.direccion.z = 1;
                datos.direccion.y = 0;
                datos.direccion =
                      datos.direccion.DevuelveVectorUnitario();
                }
               else
               {
                dirMov = DIAG3;
                datos.direccion.x = -1;
                datos.direccion.z = -1;
                datos.direccion.y = 0;
                datos.direccion =
                      datos.direccion.DevuelveVectorUnitario();
               }
              datos.posicion.x += incremento*datos.direccion.x;
              datos.posicion.y += incremento*datos.direccion.y;
              datos.posicion.z += incremento*datos.direccion.z;
           }break;
           case DIAG4:{
              opcion = rand()%2;
              if(opcion == 0)
             {
                 dirMov = ARRIBA;
                 datos.direccion.x = 0;
                 datos.direccion.z = 1;
                 datos.direccion.y =0;
             }
             else
             {
                 dirMov = IZDA;
                 datos.direccion.x = -1;
                 datos.direccion.z = 0;
                 datos.direccion.y = 0;
             }
            datos.posicion.x += incremento*datos.direccion.x;
            datos.posicion.y += incremento*datos.direccion.y;
            datos.posicion.z += incremento*datos.direccion.z;
           }break;
           MensajeActualizaPosicion();
           MensajePideAlrededor();
      }//fin dirMov
   }//fin  else
}break;
 case EST_ALERTA:{

      datos.direccion.x=
                  lista_otros_jugadores[datos.indiceJugadorObjetivo].
                  posicion.x- datos.posicion.x ;
      datos.direccion.z=
                  lista_otros_jugadores[datos.indiceJugadorObjetivo].
                  posicion.z -datos.posicion.z ;
      datos.direccion = datos.direccion.DevuelveVectorUnitario();
      datos.posicion.x += incremento*datos.direccion.x;
      datos.posicion.z += incremento*datos.direccion.z;
      MensajeActualizaPosicion();
      MensajePosEnemigo();
}break;
case EST_ATAQUE:{
      if((contadorDisparo == 0))
      {
      MensajeDisparo();
```

```
      }
      if (contadorDisparo == 10)    contadorDisparo = -1;
      contadorDisparo++;
      MensajePosEnemigo();
   }break;
   case EST_MUERTO: //MensajeMuerto();
    break;
   }
}
```

B. Descripción de los mensajes de juego

Tanto el cliente como el servidor se intercambian periódicamente mensajes para mantener sincronizado el estado de la partida. Los clientes, no poseen conocimiento de las dimensiones del mundo en que están jugando ni de los jugadores que están en la partida. Por este motivo, además de enviar su nueva posición al servidor al moverse, los clientes tienen que solicitar los datos que necesiten para poder tomar sus propias decisiones de actuación (inteligencia artificial). Esto hace que la mayor parte de tráfico sea en dirección cliente-servidor.

Toda esta comunicación se realiza a través de mensajes de juego que se han especificado de forma que incluyan la información necesaria en cada momento, y sólo la mínima información necesaria, para optimizar el tráfico de mensajes. A continuación se explica en detalle todos los tipos de mensajes que usa la aplicación:

- DPN_MSGID_CREATE_PLAYER

  Este mensaje es propio de Direct Play. Lo envía el cliente al servidor cuando quiere unirse a una partida y le envía sólo el nombre del jugador. Si el servidor acepta al nuevo jugador, crea un dpnid para éste y lo inserta en la tabla de datos donde se gestionarán todos los datos de los jugadores. El dpnid (Direct Play Number ID) es un identificador numérico para identificar y diferenciar a los jugadores que están jugando.

- MSJ_SER_SET_AGENTE_ID

```
struct MSJ_SER_SET_AGENTE_ID: public MENSAJE_GENERICO
{
DPNID dpnidPlayer;            // dpnid del jugador
};
```

  Mensaje del servidor al cliente en respuesta al mensaje DPN_MSGID_CREATE_PLAYER, para indicarle el dpnid que tendrá el jugador en la partida. En este momento, el jugador ya está registrado en el servidor como un jugador más.

- MSJ_CLI_TIPO_AGENTE

```
struct MSJ_CLI_TIPO_AGENTE : public MENSAJE_GENERICO
{
    DWORD dpnidPlayer;              // dpnid del jugador
    TipoAgente Agente;              // bando del agente
    int rangoVision;
};
```

  Mensaje del cliente al servidor para indicar qué tipo de agente es el que se ha conectado y el rango de visión que tiene este tipo de agente. Se envía en respuesta al mensaje MSJ_SER_SET_AGENTE_ID.

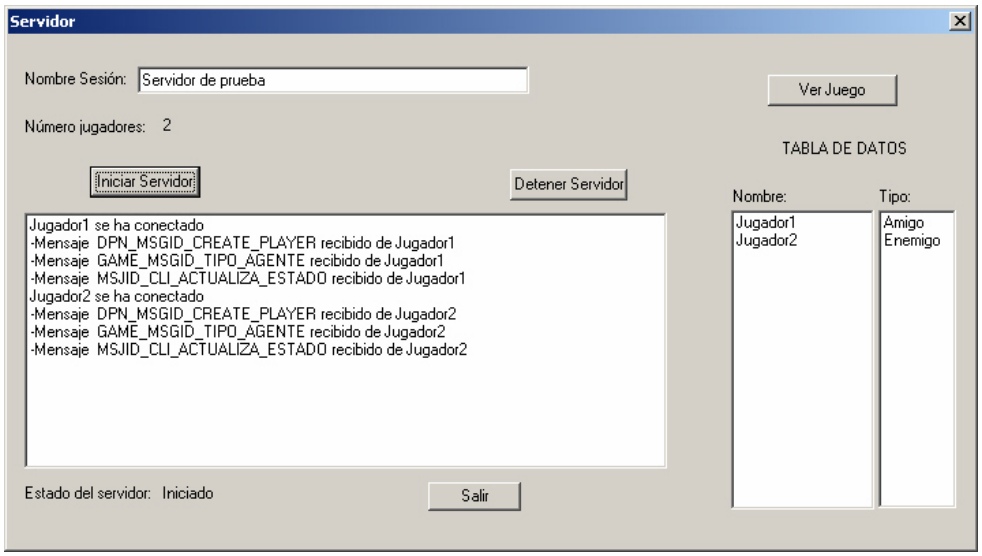
- MSJ_CLI_ACTUALIZA_ESTADO

```
struct MSJ_CLI_ACTUALIZA_ESTADO: public MENSAJE_GENERICO
{
DPNID dpnidPlayer;             // dpnid del jugador
ESTADO_JUGADOR estado;
};
```

Mensaje del cliente al servidor para indicar en qué estado se encuentra el jugador: si esté en estado de reposo, de búsqueda, de alerta, de ataque o si está muerto.

En este momento, el cliente (jugador) ya está conectado con el servidor y está preparado para comenzar a jugar.

Éste es el estado del servidor cuando se han conectado dos agentes: uno de tipo Amigo y otro de tipo Enemigo. Cuando en el servidor se presione el botón "Ver juego", se les indicará a los jugadores conectados de cual va a ser su posición inicial en el juego y de que ya ha comenzado la partida, de forma que éstos ya pueden empezar a moverse y a actuar de acuerdo a su inteligencia artificial. Al mismo tiempo, se abrirá una ventana OpenGL donde se pueda visualizar el transcurso de la partida.



- MSJ_SER_ENVIA_POS_INICIAL

```
struct MSJ_SER_ENVIA_POS_INICIAL : public MENSAJE_GENERICO
{
CVector pos;
};
```

Este mensaje se envía a cada jugador que esté conectado al servidor cuando se inicia una partida. Se le envía la posición inicial donde comenzará a jugar el jugador.

- MSJ_SER_ENVIA_SENAL_START

```
struct MSJ_SER_ENVIA_SENAL_START : public MENSAJE_GENERICO
{
};
```

Mensaje que envía el servidor a cada cliente para indicar que la partida ha comenzado. El mensaje no contiene ningún dato, ya que los clientes sólo necesitan identificar el tipo de mensaje como "mensaje de comienzo de partida". Al recibirlo, los clientes activan su inteligencia artificial, que será la

que se encargue de la toma de decisiones del jugador a partir de los datos que tiene el cliente del jugador al que representa y de los datos del su entorno que le envíe el servidor.

Ahora ya hay uno o varios jugadores conectados al servidor y la partida ha comenzado. Los jugadores irán moviéndose por el campo de juego e irán indicándole al servidor su nueva posición. Dado que los clientes no tienen conocimiento del terreno donde se están moviendo, el servidor aceptará sus movimientos siempre que no se produzca una colisión en el punto donde el jugador se ha movido. Si el servidor detecta una colisión, avisará al jugador indicado diciéndole el punto donde colisionó y la dirección en la que se estaba moviendo. Ya será cuestión del cliente la decisión de la nueva dirección hacia la que moverse para evitar ese obstáculo. Todo este proceso se hace mediante los siguientes mensajes:

- MSJ_CLI_ACTUALIZA_POS

```
struct MSJ_CLI_ACTUALIZA_POS : public MENSAJE_GENERICO
{
    DPNID dpnidPlayer;            // dpnid del jugador
    CVector pos;
    CVector dir;
};
```

Mensaje en el que el cliente le indica al servidor la nueva posición hacia la que se ha movido y la dirección que está siguiendo. La dirección la necesita saber el servidor para que el modelo que representa al jugador en el servidor se dibuje orientado hacia la dirección del movimiento. Los clientes no esperan confirmación del servidor de que la posición es válida. Sólo recibirán mensajes en respuesta a éste cuando se produzca una colisión en el punto al que se han intentado mover.

- MSJ_SER_ENVIA_ULTIMA_POS_ALCANZADA

```
struct MSJ_SER_ENVIA_ULTIMA_POS_ALCANZADA : public
MENSAJE_GENERICO
{
    DPNID dpnidPlayer;        // dpnid del jugador
    CVector pos;              // Ultima posición válida
    CVector dir;   // Dirección en la que iba el agente
};
```

Mensaje del servidor al cliente indicándole la última posición recibida de éste para indicarle que se ha producido una colisión en ese punto. El mensaje contiene la posición de la colisión, y la dirección en la que se estaba moviendo el jugador al colisionar. La inteligencia artificial reconocerá la colisión, y decidirá una nueva dirección hacia la que moverse para evitar ese obstáculo.

A partir de ahora, el intercambio de mensajes entre el cliente servidor serán en su mayor parte mensajes para enviar la nueva posición a la que un jugador se ha movido y respuestas del servidor para informar de colisiones.

- MSJ_CLI_PIDE_ALREDEDOR

```
struct MSJ_CLI_PIDE_ALREDEDOR : public MENSAJE_GENERICO
{
    DPNID dpnidPlayer;        // dpnid del jugador
    CVector pos;              //posicion hacia donde mira
```

```
    CVector direccion;
};
```

En este mensaje, el cliente le pide al servidor información sobre los jugadores enemigos que tiene a su alrededor. Se le envía la posición donde está situado el jugador y la dirección hacia la que mira. El servidor tiene almacenado un valor del rango de visión de cada tipo de agente. De esta forma, al recibir la posición donde está situado el jugador, puede localizar sólo los enemigos que se encuentran dentro de ese rango de visión. La dirección hacia donde mira el jugador es útil para establecer el ángulo de visión. No sería muy real, que un agente pudiera localizar a otro que está situado a su espalda.

- MSJ_SER_SITUACION_ALREDEDOR

```
struct MSJ_SER_SITUACION_ALREDEDOR : public
MENSAJE_GENERICO
{
INFO_JUGADOR jugadorAlrededor;
bool ultimo;
};
```

Este mensaje es la respuesta del servidor ante el mensaje MSJ_CLI_PIDE_ALREDEDOR. En él se incluye información de los jugadores que están dentro del rango de visión del agente que pidió los datos. El campo "ultimo" se usa para indicar que no se va a enviar información de más jugadores. La información que se envía es de este tipo:

```
struct INFO_JUGADOR
{
    DPNID dpnidJugador;
    TCHAR nombreJugador[MAX_NOMBRE_JUGADOR];
    CVector posicion;
    CVector velocidad;
    CVector direccion;
    TipoAgente tipoAgente;
    ESTADO_JUGADOR estado;
    int rangoVision;
};
```

Esta información es la que se recibe del resto de jugadores cercanos al que lo solicitó. De esta forma, el cliente podrá saber los datos de la gente que tiene alrededor y decidir cómo actuar. Se podía haber enviado sólo la información del enemigo más cercano, pero se optó por hacerlo de esta forma por si en una futura ampliación se modificara la inteligencia artificial del jugador. Disponiendo de todos estos datos, el cliente podría alejarse de los enemigos más cercanos si éstos estuvieran agrupados, y podría elegir como objetivo a otro enemigo que se encontrara aislado.

- MSJ_CLI_POS_ENEMIGO

```
struct MSJ_CLI_POS_ENEMIGO: public MENSAJE_GENERICO
{
DPNID dpnidPlayer;  // dpnid del que envia el mensaje
DPNID dpnidEnemigo;     //enemigo por el que se pregunta

};
```

Con este mensaje, el cliente le pide al servidor la posición de un enemigo en concreto. Aquel que tiene el dpnid "dpnidEnemigo". Una vez que un cliente fija a un enemigo como objetivo, dejará de enviar mensajes MSJ_CLI_PIDE_ALREDEDOR, y enviará mensajes de este tipo. Esto reduce el tráfico de mensajes, porque ahora el cliente en vez de esperar muchos mensajes indicándole qué es lo que tiene alrededor, esperará sólo un mensaje de este tipo, para poder tener localizado a su enemigo y poder ir a atacarle.

- MSJ_SER_ENVIA_POS_ENEMIGO

```
struct MSJ_SER_ENVIA_POS_ENEMIGO : public MENSAJE_GENERICO
{
CVector pos;
};
```

Este mensaje es la respuesta al mensaje MSJ_CLI_POS_ENEMIGO. Como información, contiene la posición del enemigo que el cliente tiene fijado como objetivo.

- MSJ_SER_INICIALIZA_ESTADO

```
struct MSJ_SER_INICIALIZA_ESTADO : public MENSAJE_GENERICO
{
ESTADO_JUGADOR estado;
};
```

Este mensaje lo envía el servidor a todos los jugadores conectados para que cambien su estado al que indique el contenido del mensaje. Realmente su uso es para inicializar los estados de los jugadores al comenzar una nueva partida. De esta forma, se envía este mensaje indicando a todos los jugadores que pasen al estado de búsqueda, con lo que los jugadores que estaban muertos "reviven".

- MSJ_CLI_DISPARO

```
struct MSJ_CLI_DISPARO : public MENSAJE_GENERICO
{
      DPNID dpnidPlayer;    // dpnid del que dispara
CVector pos;          // posicion origen del disparo
CVector posEnemigo;   // posicion a la que disparo
CVector direccion;    // dirección del disparo
float distancia;      //distancia del enemigo
};
```

Mensaje que envía el cliente al servidor para indicarle que ha efectuado un disparo. Cuando el servidor recibe un mensaje de este tipo, crea un AgenteDisparo con los datos contenidos en el mensaje: su propio dpnid, la posición inicial, la dirección del disparo, la posición del enemigo hacia el que dispara y la distancia hasta él. El contenido del mensaje incluye el dpnid del jugador que dispara, que es necesario para asignárselo al AgenteDisparo que se crea. Así cuando el AgenteDisparo choque contra otro jugador, podremos saber quién fuen el que disparó.

- MSJ_SER_JUGADOR_MUERTO

```
struct MSJ_SER_JUGADOR_MUERTO : public MENSAJE_GENERICO
{
```

```
    DPNID dpnidPlayer;        // dpnid del jugador muerto
};
```

Mensaje que envía el servidor a todos los jugadores conectados indicándoles el dpnid del jugador que ha muerto.

1. Flujo de mensajes

El cliente y el servidor se están intercambiando mensajes continuamente. En esta sección se explica el flujo de mensajes que se produce durante una partida.

Cliente		**Servidor**
DPN_MSGID_CREATE_PLAYER	———	Recibo el nombre del jugador
Me asignan un dpnid	———	MSJ_SER_SET_AGENTE_ID
MSJ_CLI_TIPO_AGENTE	———	Indico que tipo de agente soy, y mi rango de visión
MSJ_CLI_ACTUALIZA_ESTADO	———	Indico el estado en el que me encuentro: inicialmente en estado de búsqueda.

Ahora la conexión entre cliente y servidor está establecida. El cliente está a la espera para comenzar a jugar.

Recibo la posición inicial donde empezaré a jugar	———	MSJ_SER_ENVIA_POS_INICIAL
Señal de comienzo de partida	———	MSJ_SER_ENVIA_SENAL_START

Inicialmente, el cliente no dispone de datos para poder tomar una decisión de movimiento, así que solicita al servidor que le informe de qué es lo que tiene alrededor para así saber cómo actuar.

MSJ_CLI_PIDE_ALREDEDOR	———	Recibo solicitud de información alrededor de la posición indicada
Recibo N mensajes de este tipo con información de agentes cercanos a la posición pedida	———	MSJ_SER_SITUACION_ALREDEDOR

Los clientes ahora tienen información de su entorno, y pueden moverse según lo que tengan a su alrededor y las decisiones que tome su inteligencia artificial. Cuando el jugador encuentre a un enemigo cercano, se centrará en él y le perseguirá hasta que sea eliminado de la partida, por él mismo o por otro jugador.

MSJ_CLI_ACTUALIZA_POS	———	Recibo la nueva posición de un jugador
Recibo la posición a la que he intentado moverme pero he chocado con algo	———	MSJ_SER_ENVIA_ULTIMA_POS_ALCANZADA
MSJ_CLI_POS_ENEMIGO	———	Recibo solicitud de la posición de un enemigo concreto
Recibo la posición actualizada del enemigo al que persigo	———	MSJ_SER_ENVIA_POS_ENEMIGO

MSJ_CLI_DISPARO	_____	Recibo información de un disparo de un jugador desde un punto hacia otro punto en concreto
Recibo información de que un jugador ha muerto	_____	MSJ_SER_JUGADOR_MUERTO

C. Comunicación Cliente – Servidor - Motor gráfico

Como se ha comentado anteriormente, entre el cliente y el servidor debe existir una comunicación fluida. Además el servidor debe disponer de los medios necesarios para registrar todos los datos relevantes de los clientes que están conectados a él. Al mencionar la teoría de agentes se habló de forma abstracta de la necesidad de mantener en el servidor unas 'páginas amarillas', pues bien, la implementación de esta idea se ha hecho en la clase CTablaDatos cuyo funcionamiento se describe a continuación.

El objetivo principal de esta clase es mantener sincronizados los estados de los agentes que están conectados al servidor. Cuando un jugador se conecta, en el servidor se debe registrar la información relevante del mismo y cualquier cambio que se produzca también se debe notificar en la tabla de datos.

La información que el servidor necesita de cada cliente está almacenada en un registro de tipo INFO_JUGADOR que consta de los siguientes campos:

```
struct INFO_JUGADOR
{
    DPNID dpnidJugador;                          // DPNID del
jugador
    TCHAR nombreJugador[MAX_NOMBRE_JUGADOR];    // Nombre del
jugador
      CVector posicion;
      CVector velocidad;
      CVector direccion;
      TipoAgente tipoAgente;
      ESTADO_JUGADOR estado;
      int rangoVision;
};
```

Veamos en detalle cada uno de los campos:

- dpnidJugador: En el identificador único de cada cliente. Lo proporciona DirectPlay.
- nombreJugador: Este campo tiene un objetivo meramente informativo. Se usa para hacer mas amigable la representación del interfaz gráfica de usuario.
- posicion y velocidad: son vectores de tres elementos. Esta información es de vital importancia para mantener la sincronización de los distintos agentes, entre ellos y con el motor gráfico. Los agentes solicitan esta información de los demás agentes para poder ejecutar su inteligencia artificial, así cada vez que algún agente varía su posición o velocidad, se recibirá un mensaje en el servidor y este debe actualizar el valor del dato en la tabla de datos para que las peticiones de otros agentes reciban datos actualizados.
- tipoAgente: El juego dispone de dos tipos de agentes (excluyendo el agente disparo). Son el agente *'Amigo'* y *'Enemigo'*. Se distinguen básicamente en el tipo de malla que se usa para la representación.
- rangoVision: Este campo especifica el alcance de visión que tienen los agentes. Se necesita en la inteligencia artificial.

Toda esta información de cada cliente se guarda en una tabla, tabla que se ha implementado como una 'hash table'[2] y se accede a sus filas mediante al identificador único de cada agente, es decir, su DPNID.

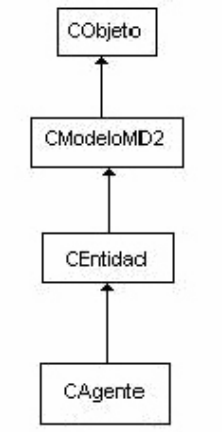
Las hash tables se implantan en Microsoft Visual C++ en la clase CMap. La definición del objeto tabla es:

```
CMap <DPNID, DPNID ,INFO_JUGADOR, INFO_JUGADOR> tabla;
```

Los datos que guarda la tabla son del tipo INFO_JUGADOR y la clave de acceso es del tipo DPNID. La tabla se mantiene como una variable global en el servidor y el acceso a ella, tanto para lectura como para modificación de sus datos, está controlada por el uso de secciones críticas para asegurar la consistencia de la información.

Por otro lado, existe otro elemento en el proyecto que tiene que ver los cambios que tienen lugar en la parte del cliente. Es el motor gráfico. La arquitectura de la aplicación incluye los mecanismos necesarios para la integración del Cliente/Servidor con el Baño de Sangre y poder obtener así una representación gráfica de lo que ocurre en los clientes.

La clase que hace de nexo de unión es CAgente. Esta clase hereda a través de una jerarquía de la clase CObjeto usada en el Baño de Sangre.



Por cada cliente que se conecta al servidor se crea un objeto de tipo CAgente:

```
CAgente* agente = new CAgente(posicion,velocidad,
                  cliente.dpnidJugador,cliente.tipoAgente,tabla,&a
                  ccesoTabla);
```

Entre los parámetros de la constructora está *'tabla':* puntero a la variable global del servidor que es del tipo CTablaDatos. Sirve para poder realizar el acceso a la tabla de datos desde el motor gráfico.

'dpnidJugador' sirve para identificar al agente de igual forma que el cliente correspondiente y además es la clave para acceder a la tabla de datos.

'&accesoTabla' implementa la sección crítica.

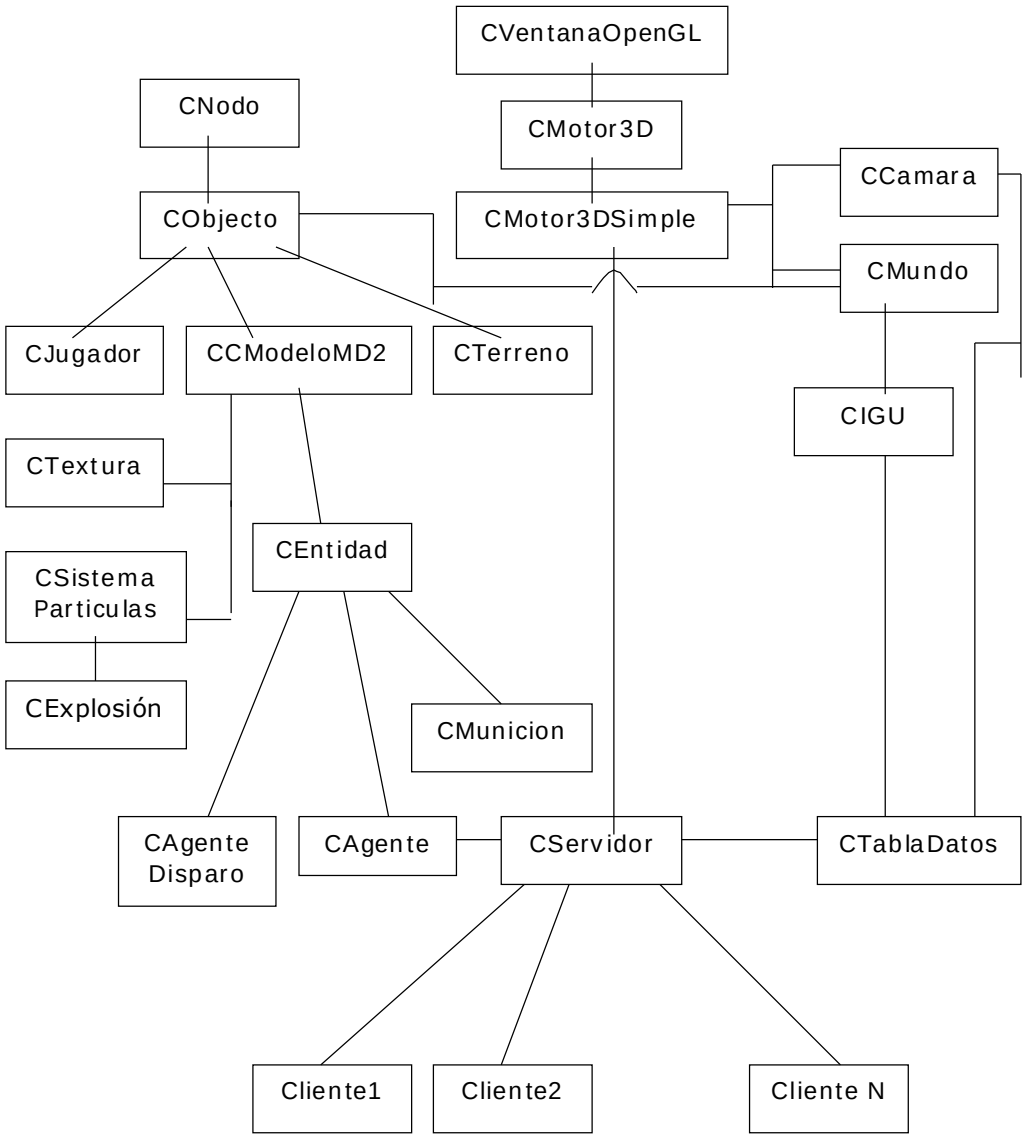
Una vez creado el agente correspondiente a un cliente del juego, hay que unirlo a la estructura que maneja el juego para que se pueda tratar de forma análoga a como lo hace el Baño de Sangre.

[2] Estructura de datos que guarda un conjunto de datos cada uno de ellos junto con un a clave. La clave permite el acceso a los datos en un tiempo constante.

CAgente incluye métodos de representación gráfica:

```
void AlAnimar(float intervaloTiempo);
void AlColisionar(CObjeto* objetoColision);
void AlPreparar();
void ProcesaColision(CObjeto* objetoColision) ;
```

El diagrama de clases del servidor junto con las clases del juego queda de la siguiente forma:



D. Los Agentes

Los agentes son las estructuras de datos que creamos en tiempo real, que utiliza el motor gráfico para visualizar los clientes y obtener una representación gráfica del funcionamiento y dinámica de los mismos.

Estos objetos se crean a medida que van conectándose los clientes al servidor, por lo que hay tantos objetos de tipo agente como clientes conectados al servidor.

Al crearse se añaden a la estructura de datos del motor gráfico, que está formada además del terreno, y del jugador por defecto del sistema. Estos agentes van estrechamente ligados a los clientes, ya que son una representación software en el servidor de los clientes. Tienen la misma información que los clientes, ya que estos cuando intervienen en el ciclo de juego actualizan su información con la información de la base de datos del servidor sobre el cliente que representa.

Se puede ver en el diagrama de clases la jerarquía de clases y las herencias de la clase CAgente. Hemos mantenido la misma estructura que existía en el Baño de Sangre, para facilitar la integración de nuestro código y funcionalidades. El ciclo de juego se mantiene similar al del Baño de Sangre, salvo una pequeña modificación en el orden. En el baño de sangre, como se menciona en la sección que explica su diseño, el ciclo de juego es el siguiente[3]:

1.- Inicialización de comandos Opengl. Como la habilitación de texturas, inicialización de matrices, etc, etc…

2.- Se comprueban posibles colisiones en el mundo. En caso de colisión se procesa su respuesta.

3.- Se mueve y orienta la cámara.

4.- Se animan los objetos. Dependiendo de lo que sean harán una cosa u otra.

5.- Se dibujan los objetos del mundo.

Nuestro ciclo de juego es el siguiente:

1.- Inicialización de comandos Opengl. Como la habilitación de texturas, inicialización de matrices, etc, etc…

2.- Se mueve y orienta la cámara.

3.- Se animan los objetos. Dependiendo de lo que sean harán una cosa u otra.

4.- Se comprueban posibles colisiones en el mundo. En caso de colisión se procesa su respuesta.

5.- Se dibujan los objetos del mundo.

Se puede apreciar que el orden es significativamente distinto. Aunque realizamos las mismas tareas.

El cambio tiene una sencilla razón, hemos creído conveniente que la colisión de los objetos se realizase justo antes que su dibujo, porque así controlamos que los objetos no se salgan del mundo de juego. Es decir, si mantuviésemos el mismo orden, y procesáramos la colisión, después la animación y después dibujásemos, en caso de colisionar tendríamos que mandar un mensaje al cliente de que éste ha

[3] La explicación se realizará por pasos y de palabra en lugar de introducir pseudocódigo, para una mejor comprensión.

colisionado. El cliente tendría que actuar en consecuencia y responderle al servidor, mediante otro mensaje. Mientras tanto en el servidor el ciclo de juego sigue su curso, es decir, animamos el agente, que no es más que leer de la base de datos la posición nueva del agente, una posición que no está actualizada puedo que el cliente la envió antes de saber que había colisionado, y acto seguido se dibujaría. Dando como resultado que el cliente está en una posición equivocada. Y sabiendo que el agente está constantemente colisionando con el terreno puesto que el terreno es irregular, sería un caos.

La solución que hemos tomado es la de cambiar el orden, como ya hemos mencionado anteriormente. Así, lo primero que hacemos es animar el agente. Es decir, leemos la posición de la base de datos. Comprobamos que después si se realizan colisiones. Tenemos una colisión segura con el terreno. Informamos al cliente en caso de colisión, enviándole el mensaje correspondiente, que hemos explicado en la sección de mensajes del juego. Pero como sabemos que al cliente no le da tiempo a responder antes de dibujar, lo que hacemos es dibujar al cliente en una situación correcta, sabemos los márgenes y la altura del terreno en el punto de colisión, no dejamos que la posición del cliente tome valores equívocos.

Otro aspecto importante de los agentes es su eliminación. Los agentes se eliminan cuando llegan al estado de muerto. Cuando el servidor detecta que un agente ha muerto, lo elimina de la estructura y además informa al cliente correspondiente que en la partida en curso ha muerto. Eso no significa que se pierde la conexión con el cliente. La conexión permanece para futuras partidas.

Cuando tras una colisión, o porque la partida se cierra, se detecta que el agente debe morir, se le cambia al estado de muerto y además se activa una variable interna que indica si el agente debe ser eliminado de la estructura o no. Cuando esta variable estamuerto se activa, en el ciclo de juego, y en concreto en el método animar de la clase CObjeto se detecta y se elimina el agente de la estructura. Es importante reseñar, que tras decidirse que un agente está muerto tras una colisión por ejemplo de un agente disparo, hay un periodo en el que el agente aún no es eliminado de la estructura, puesto que ese tiempo se dedica a ejecutarse la animación correspondiente de la muerte del agente.

E. El Agente Disparo

El agente disparo es distinto al agente descrito anteriormente. El agente disparo al contrario que el agente anterior, no tiene un representación en forma de cliente. No hay un cliente *manejándole.* Si que es un cliente el que indica al servidor que debe crearse un agente disparo, pero se desentiende de él inmediatamente tras la petición.

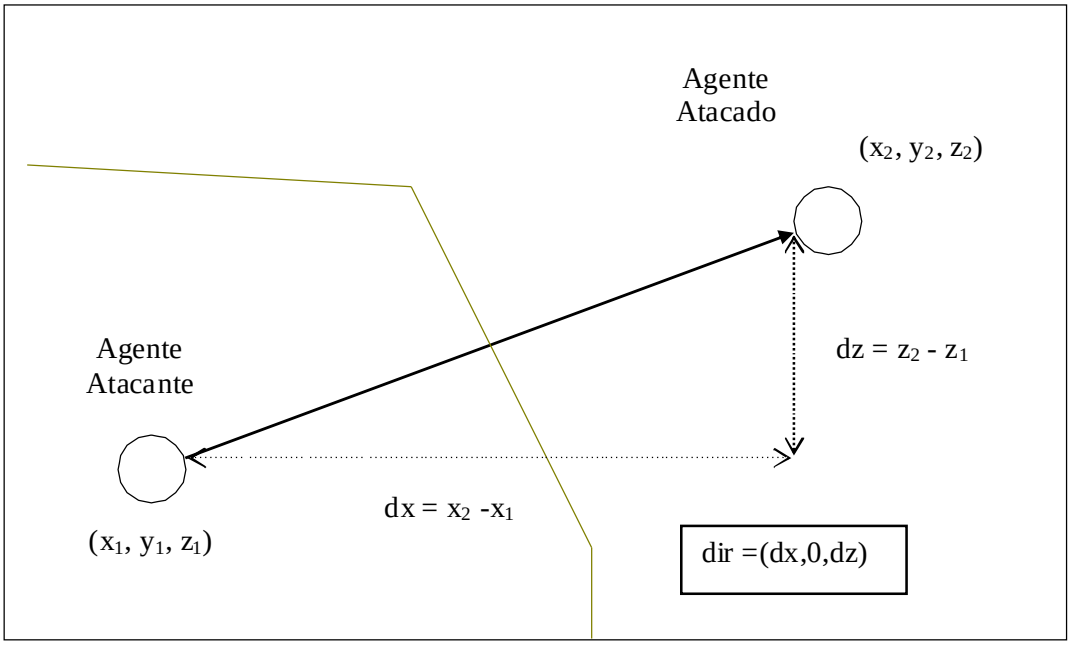
El agente disparo, al no tener esta representación exterior, no es introducido en la base de datos. Pero si que es introducido en la estructura de datos del juego.

Al crear un disparo e introducirlo en la estructura de datos, tenemos que crearlo con la suficiente información para que este pueda llevar a cabo su cometido, sin necesidad de ninguna guía exterior. Por lo que le pasamos la posición y DPNID del agente que dispara, la posición del agente al que queremos disparar. Con la DPNID del agente que dispara otorgamos al disparo con una identificación única, que nos servirá a la hora de la colisión de ese disparo con algún agente, saber quien le ha matado.

Con la posición inicial, y final podemos calcular la dirección del disparo, y obteniendo las alturas de esos 2 puntos en terreno, también podemos calcular el ángulo de inclinación de la trayectoria del disparo. Con toda esta información además de la velocidad que queramos que tenga el disparo, el objeto agente disparo es totalmente autosuficiente.

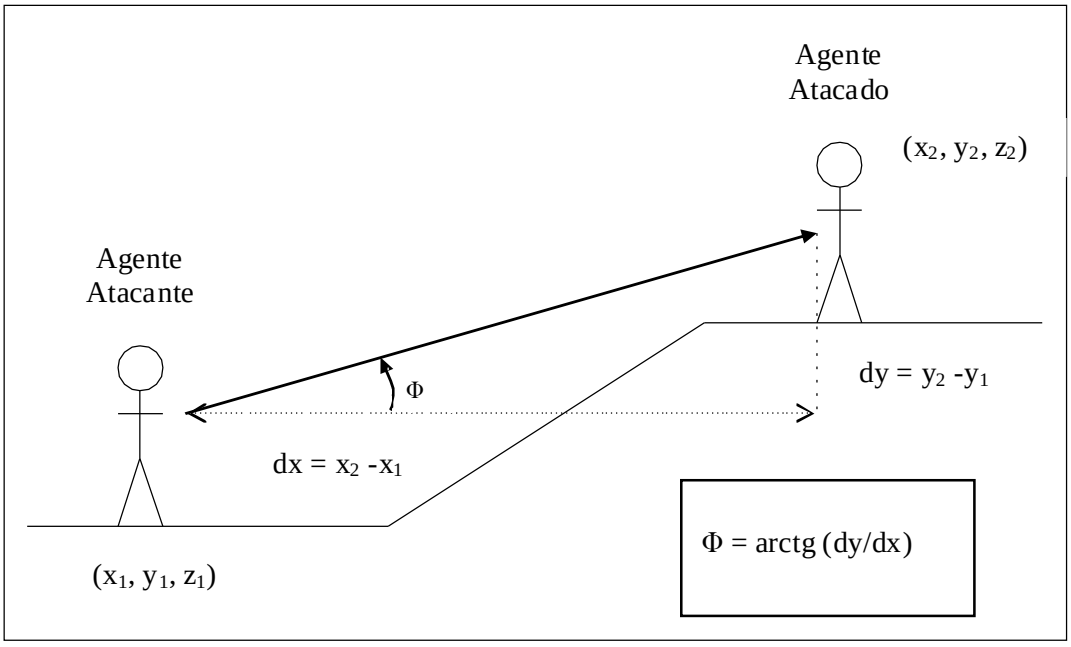
En el diagrama siguiente mostramos cómo se realizan estos cálculo.

En este diagrama mostramos el cálculo de la dirección:



El vector de dirección hay que normalizarlo para que el avance sea controlado.

En este diagrama mostramos el cálculo de la inclinación:



Sabiendo por tanto la dirección y el ángulo de inclinación, lo que queda es calcular la posición siguiente en función de la posición anterior. Esto se realiza en el método AlAnimar de la clase CAgenteDisparo:

```
void CAgenteDisparo::AlAnimar(float intervaloTiempo)
{

        this->posicion->x += Velocidad* this->direccionMov->x;
        this->posicion->y += Velocidad *sinf(this->anguloInclinacion);
        this->posicion->z += Velocidad *this->direccionMov->z;

        if(this->esExplosion)
                this->explosion->Actualizar(intervaloTiempo);

        if(!this->esExplosion) {
                if(this->distanciaViajada >= 500000.0f)
        {
                  this->esExplosion = true;
                  this->velocidad->modifica(0.0, 0.0, 0.0);
                  this->explosion = new CExplosion(50, this->posicion, 0.1,
                                                   this->texturaExplosion->texID);
                }
        }
}
```

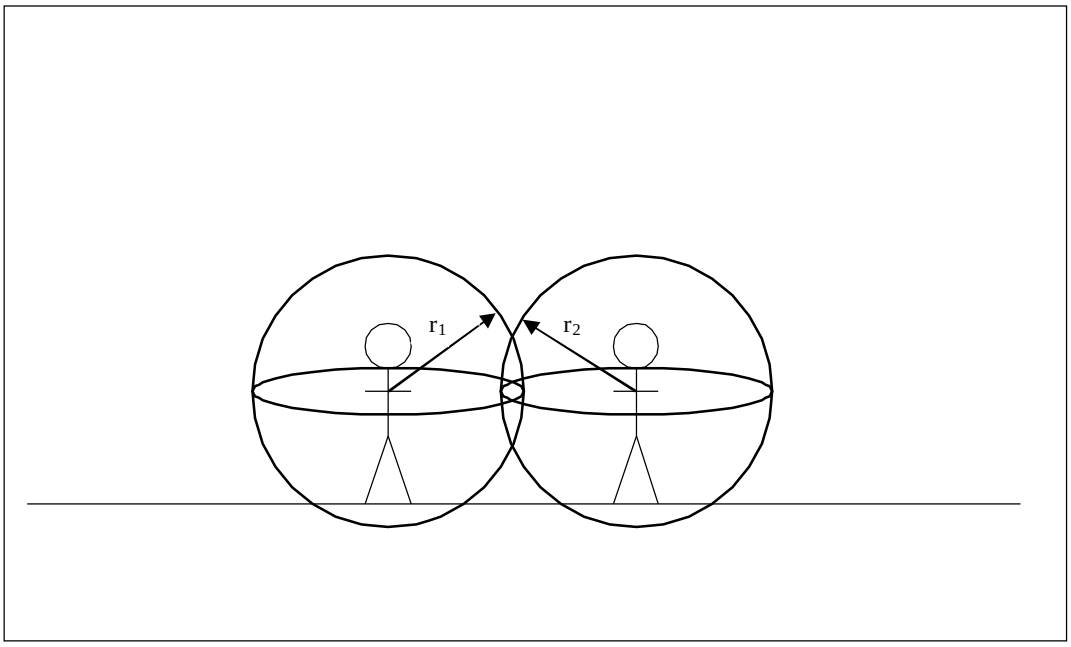
Observamos que al animar el agente disparo, depende de si este está explotando o no. Si no está explotando entonces seguimos calculando en cada ciclo la siguiente posición de la trayectoria. En caso de estar explotando se ejecuta el sistema de partículas. Este sistema se comenta en la sección del baño de sangre.

Lo último que queda mencionar es la eliminación del disparo. El disparo al igual que el agente, hereda de CObjeto. Así que una vez que se ha ejecutado la animación de la explosión se activa la variable estamuerto y se elimina de la misma manera que los agentes.

F. Sistema de Colisiones

El sistema de colisiones es un apartado muy importante dentro del motor gráfico. Es muy importante detectar cuando 2 objetos del sistema colisionan, y también qué hacer cuando 2 objetos colisionan.

Para la detección de la colisión utilizamos una esfera de colisión. Es decir, englobamos al objeto dentro de una esfera imaginaria, y si la esfera de algún otro objeto la toca, inferimos que se produce una colisión. En el diagrama se muestra cómo funciona esta técnica:



Observamos que las 2 esferas se están tocando. Entonces aunque los 2 muñecos no se están tocando, al producirse este choque de esferas, nosotros decidimos que ya se ha producido la colisión. Esta técnica es muy sencilla de utilizar a la vez que es bastante efectiva. Únicamente es necesario definir una esfera de un tamaño acorde al objeto que engloba.

También es importante decidir qué se hace a la hora de colisionar con ciertos objetos. Cuando un agente colisiona con un agente disparo, como ya hemos comentado anteriormente, se inician los procesos de animación de la muerte para el agente y de la explosión para el agente disparo. Pero además en el caso del agente, se le envía un mensaje al cliente como respuesta a esta colisión informándole que está muerto.

Pero cuando un agente colisiona con el terreno, sobretodo con los márgenes del terreno[4], lo que es la longitud y la anchura, o colisiona con otro agente, al cliente se le envía un mensaje indicándole la posición donde se ha producido la colisión y la dirección en la que iba al producirse la colisión.

[4] Cuando el agente colisiona con el terreno en cuanto a la altura, no se le notifica al cliente mediante un mensaje puesto que esto saturaría la red. El cliente siempre colisiona con el terreno en ese aspecto. En la descripción de nuestro ciclo de juego se explica la acción que se toma en este caso.

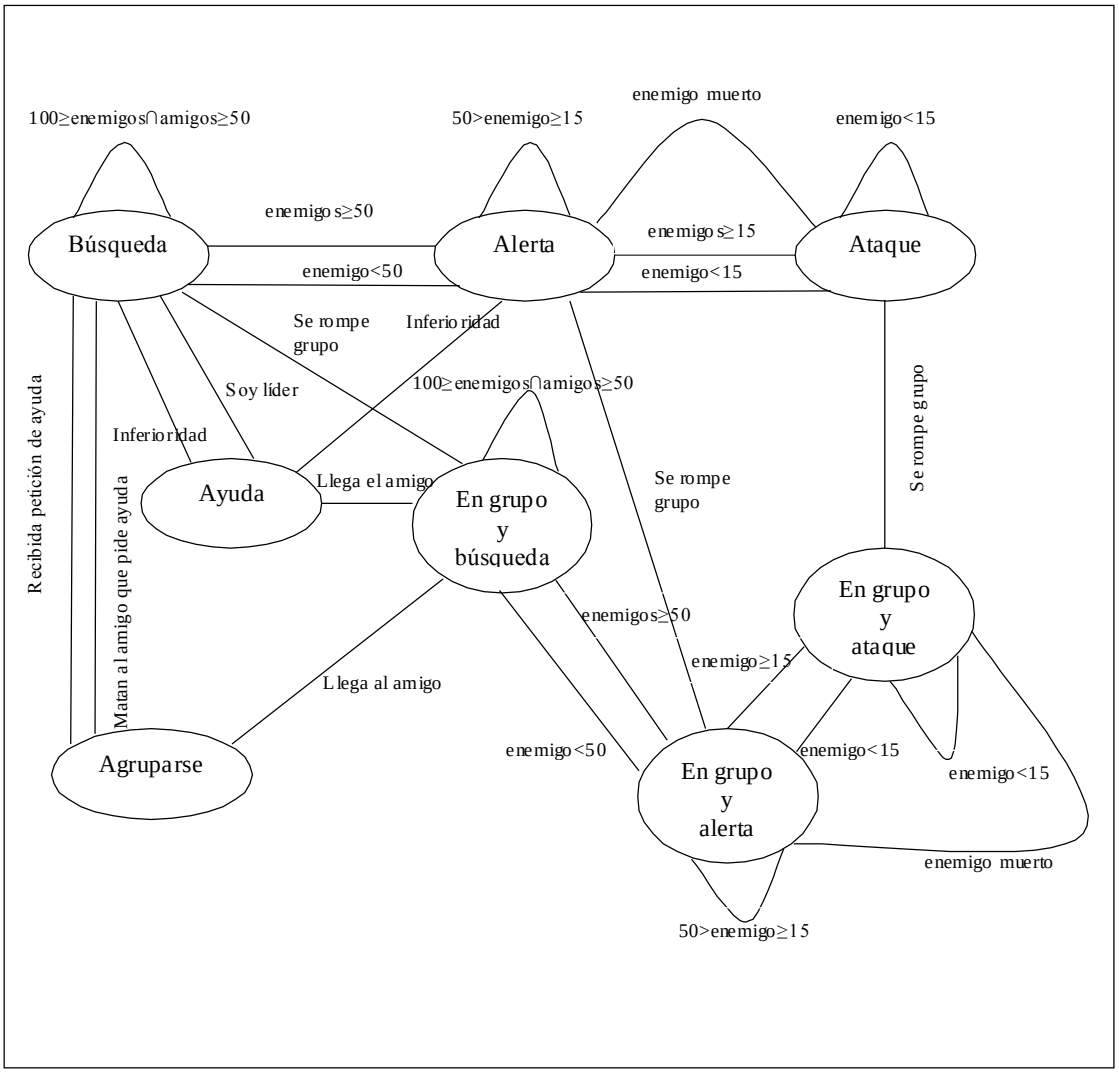
El cliente ante este mensaje lo que hace es recalcular una nueva dirección de movimiento y por tanto recalcular una nueva posición actual. Lo hace de 2 maneras distintas. Una manera es invertir completamente la dirección de movimiento y en base a esta recalcular la posición actual. Y la otra es calcular de manera aleatoria una nueva posición de movimiento, comprobar que no coincide con la dirección de movimiento en la que iba, y volver a calcular la posición actual. Alterna de manera aleatoria entre estas 2 opciones.

V. Posibles ampliaciones

A. Ampliación de la IA

Como ya dijimos en la sección donde se explica el funcionamiento y diseño de la IA, esta gracias a la forma en la que está construida es de muy fácil extensión. Sólo es necesario definir nuevos estados y añadirlos de forma conveniente a los estados ya diseñados.

Como también dijimos en la sección de la IA, esta sirve para describir el comportamiento que nosotros queremos que tengan los agentes. Como ampliación proponemos un comportamiento en grupo que ya habíamos contemplado al inicio del desarrollo del proyecto. Ahora los agentes a pesar de estar divididos en grupos, no muestran ningún tipo de cooperación entre ellos. Así que diseñamos una máquina de estados que representa este comportamiento. La máquina se muestra en el diagrama a continuación:



Para no complicar innecesariamente el diagrama hemos omitido los estados de Reposo y Muerto. Pero desde cualquier estado puedo morir y desde la mayoría de los estados puedo entrar en reposo temporalmente.

Ahora pasamos a describir el funcionamiento de los estados más característicos de esta máquina de estados:
Búsqueda:

En este estado el agente se mueve aleatoriamente por el terreno. Por lo tanto la acción que se realiza es la de un movimiento aleatorio. Más adelante se explicará cómo se realiza este movimiento aleatorio. Tras realizar la acción se le envía al servidor un mensaje de actualización de posición del agente, para que el servidor la actualice en la base de datos. En este estado se le pregunta al servidor acerca de los posibles agentes del otro equipo que están dentro del rango de visión de este agente. Si encuentra una situación de inferioridad en cuanto a número de agentes del equipo contrario, con respecto al número de agentes del mismo equipo, pasa al estado de Ayuda. Si encuentra a un agente del equipo contrario que a una distancia menor que la distancia prefijada para pasar al estado de alerta, se pasa. Si se recibe un mensaje de petición de ayuda y el solicitante se encuentra dentro de un perímetro predefinido para la asistencia de ayuda, se pasa al estado de agruparse. En otro caso se permanece en estado de búsqueda.

Alerta:

En el estado de alerta, el movimiento del agente ya no es aleatorio, sino que se mueve hacia el agente del otro equipo que se ha detectado que está dentro del perímetro de alerta. Por lo tanto la acción que se realiza es la de un movimiento controlado. Tras este movimiento se le envía al servidor un mensaje de actualización de posición para que el servidor actualice la posición del agente en la base de datos. Después se pregunta al servidor por la posición del agente que hemos detectado que está dentro del perímetro de alerta además de posibles agentes dentro del rango de visión. Si se encuentra en situación de inferioridad pasa al estado de Ayuda. En caso de que el agente objetivo esté lo suficientemente cerca como para pasar al estado de ataque, es decir que la distancia que los separe sea menor que la distancia prefijada para pasar al estado de ataque, se realiza la transición al estado de ataque. En otro caso se permanece en estado de alerta

Ataque:

En este estado el agente ataca al agente del otro equipo que lleva persiguiendo, y que ahora se encuentra dentro del perímetro de ataque. Por lo tanto la acción que se realiza en el estado de ataque no es de un movimiento, sino que es un disparo. Se dispara al agente objetivo. El disparo consiste en un mensaje que se le envía al servidor indicándole quien es el que efectúa el disparo, desde dónde y a dónde se está disparando. Tras enviar el mensaje de disparo, se realiza una pregunta al servidor sobre la posición del agente a quien estoy atacando. En caso de que la distancia que separa a este agente de su objetivo sea mayor que la distancia prefijada de ataque, se pasa automáticamente al estado de alerta. En otro caso permanezco en este estado. Pero estando en este estado además, nos interesa otra información que nos llega del servidor. Nos interesa saber si el agente objetivo está muerto o no. Cuando un agente muere el servidor informa a todos los clientes que puedan estar conectados. Es en el estado de ataque donde esta información que recibimos del servidor nos puede interesar. Si nuestro agente objetivo está muerto pasamos automáticamente al estado de búsqueda.

Ayuda:

En este estado el movimiento es de evasión. Se intenta hallar una trayectoria por la cual el agente pueda escapar de la situación de inferioridad. La acción consistiría en huir. El agente mandaría un mensaje de actualización de

posición al servidor. También se mandaría un mensaje de ayuda a todos los miembros de su mismo equipo. Se mantiene en este estado hasta que lleguen refuerzos o hasta que pase un tiempo razonable. Cuando llegan refuerzos, pasa al estado Búsqueda como líder de un grupo.

Agruparse:

En este estado la acción es similar a la que se realiza en el estado de Alerta, pero ahora como objetivo está la posición del agente que ha solicitado ayuda. Solicita constantemente la posición del agente que solicita ayuda y cuando está suficientemente cerca, esta distancia esta predefinida, para formar un grupo, pasa al estado En grupo y búsqueda.

En grupo y búsqueda:

El agente actúa de la misma manera que en el estado Búsqueda, excepto al desplazarse, que ya no sería una acción de movimiento aleatorio sino que siempre se mantiene una misma posición relativa respecto al líder del grupo. Se solicitaría constantemente la posición y orientación del líder y en base a esta información se calcularía la posición de este agente. Con esto se consigue un movimiento en grupo y en formación donde el líder decidiría donde moverse y el resto le seguiría. Si se recibe un mensaje de disolución rompe el grupo, se pasa automáticamente al estado de Búsqueda.

En grupo y alerta:

El agente actúa de la misma manera que en el estado Alerta, excepto al desplazarse, que lo hará siguiendo al líder. No se tiene que compartir el mismo objetivo que el líder. Si hubiese más de un agente del equipo contrario dentro del perímetro de estado de alerta se seleccionaría al azar entre alguno de ellos. Si se rompe el grupo, pasará al estado Alerta.

En grupo y Ataque:

El agente actúa de la misma manera que en el estado Ataque. El ataque se realiza sobre el objetivo seleccionado por cada agente en caso de que el objetivo estuviese dentro del perímetro de ataque. Si se rompe el grupo, pasará al estado Ataque.

VI. Aplicaciones del proyecto

El simulador de cooperativismo robótico tiene varias aplicaciones, la más intuitiva es la simulación de agentes. Estos agentes pueden representar multitud de funcionalidades reales. Algunos ejemplos son:

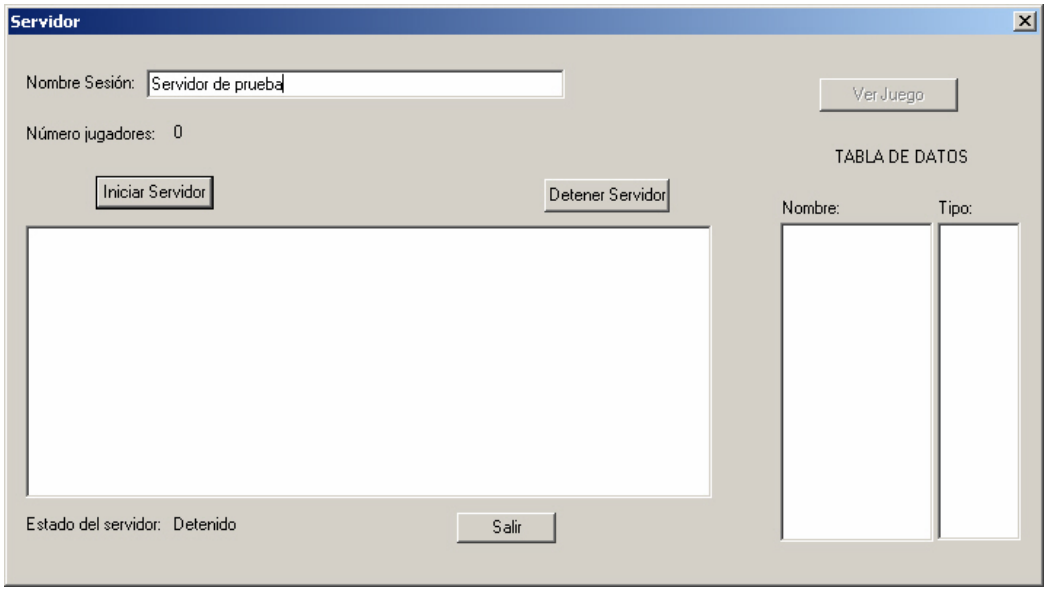
1. Simulación del funcionamiento de un GPS para el seguimiento de un avión.
2. Simulación de estrategias de combate.
3. Control y monitorización:
    a. de los trenes de una red de metro
    b. de una planta de almacenamiento y distribución de mercancías
4. Desarrollo de juegos
5. Simuladores en general

VII. Manual de usuario

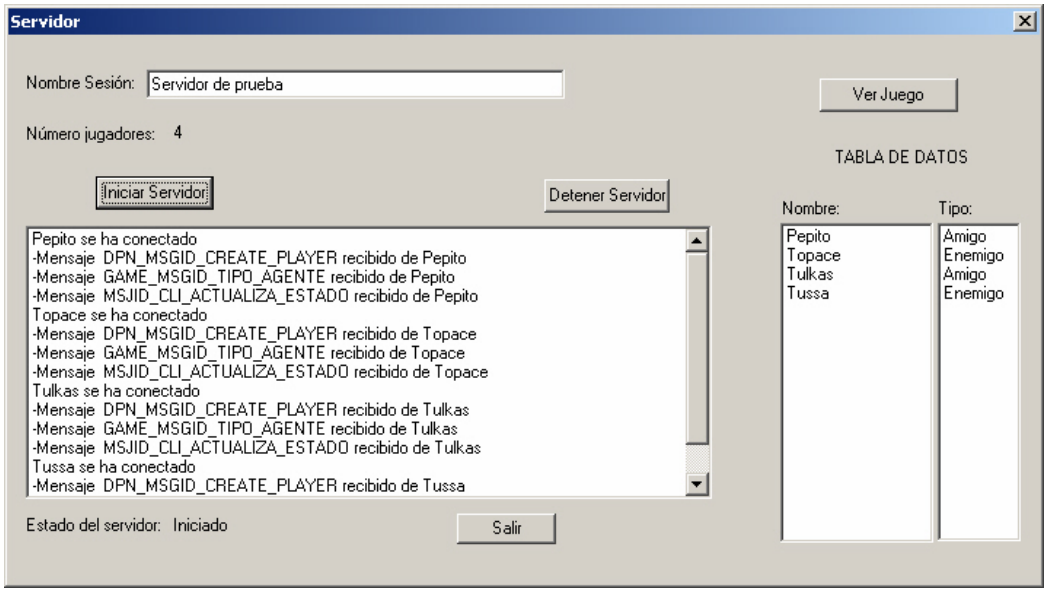
A. Servidor

El formulario principal del servidor consta de las siguientes partes:

- Nombre Sesión: campo para introducir el nombre que queramos darle a la partida.
- Número jugadores: nos indica cuantos jugadores hay conectado a nuestra partida.
- Iniciar Servidor: arranca el servidor para que empiece a aceptar jugadores.
- Ver Juego: una vez iniciado el servidor, este botón se habilita, y al pulsarlo abre una ventana OpenGL con la representación gráfica de la partida.
- Detener servidor: detiene el servidor.
- Tabla de Datos: nos indica los nombres de cada jugador conectado al servidor y el equipo al que pertenecen: amigos o enemigos.
- Ventana de mensajes: es donde se muestran los mensajes más importantes recibidos de los jugadores.
- Salir: botón que cierra la aplicación.



Para iniciar una partida, introducimos el nombre de partida que queramos y pulsamos "Iniciar Servidor". En este momento, el servidor está listo para recibir conexiones de clientes (jugadores). Una vez iniciado el servidor, independientemente de si hay jugadores conectados o no, podemos abrir la pantalla de visualización del juego (botón "Ver Juego"). Para volver a la pantalla anterior pulsaremos la tecla "Esc". La partida se para cuando la ventana OpenGL se cierra y se reanuda una nueva al volver a abrir la ventana de visualización



Servidor arrancado con 4 jugadores conectados

Al pulsar el botón "Ver Juego" abriremos la ventana OpenGL de visualización de la partida. En la pantalla de visualización aparece en la esquina superior derecha el número de agentes de cada clase que quedan con vida en la partida. También aparecen tres menús desplegables, que se seleccionan con las teclas 1, 2 y 3. Para movernos por las subcategorías, podemos hacerlo con las teclas de flecha arriba y flecha abajo, y para seleccionar la opción indicada pulsaremos la tecla "Enter". Con la tecla "Espacio" se deshace la selección.

El menú "Cambiar Cámara" (tecla 1) permite colocar la cámara en una de las 4 posiciones sobre el terreno que están predefinidas.

Visualización de la partida desde la cámara 2

El menú "Seleccionar Jugador" (tecla 2) despliega un submenú con la lista de jugadores vivos que siguen en la partida. Ahora si seleccionamos un jugador del submenú, la cámara cambiará y se situará a la espalda de dicho jugador, siguiendo sus movimientos. A continuación podemos ver una captura de pantalla con la cámara en modo subjetivo siguiendo al jugador Tussa.

El menú "Estadísticas" (tecla 3) nos muestra las estadísticas acumuladas de muertes producidas (llamadas Frags en los juegos online) y recibidas de cada jugador durante todas las partidas jugadas.

Visualización de las estadísticas

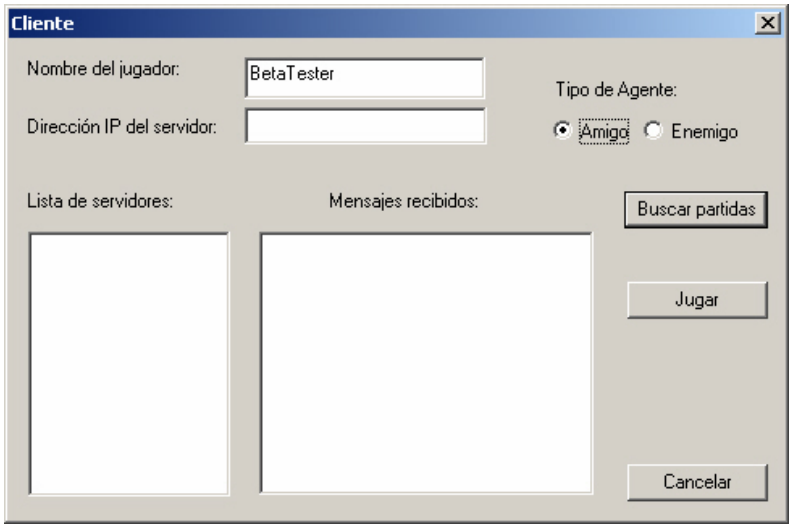
Con la tecla "J" desactivamos el modo de visualización y pasamos al modo de jugador. Como jugador, podremos desplazarnos por el terrenos con las teclas A, W, S y D, orientar la cámara con el ratón, y disparar con el botón izquierdo del ratón. El punto de mira donde apunta el jugador es la cruz roja que aparece en el centro de la pantalla.

Modo jugador activado

B. Cliente

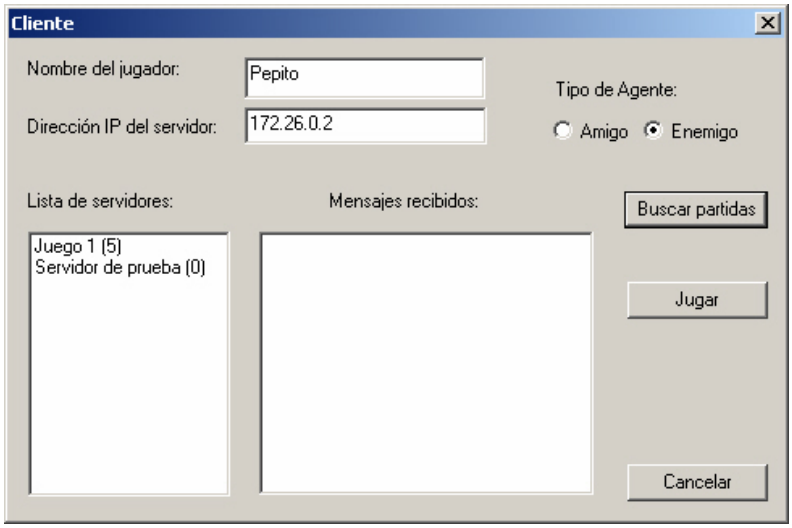
Al iniciar el cliente, podemos ver las siguientes partes del formulario:

- Nombre del jugador: introduciremos el nombre con el que queremos que se nos identifique en la partida. Inicialmente aparece "Beta Tester" por defecto.
- Tipo de Agente: seleccionaremos a que bando queremos pertenecer: amigos o enemigos.
- Dirección IP del servidor: dirección IP de la máquina que hará de servidor de la partida. Si se va a jugar en red local, se puede dejar en blanco este campo.
- Buscar partidas: este botón buscará partidas en funcionamiento en la dirección IP especificada, o en la red local si no se ha especificado IP.
- Jugar: botón para unirse a la partida seleccionada en la tabla Lista de servidores.
- Mensajes recibidos: aquí irán apareciendo los distintos mensajes recibidos desde el servidor.
- Cancelar: botón para salir de la aplicación.

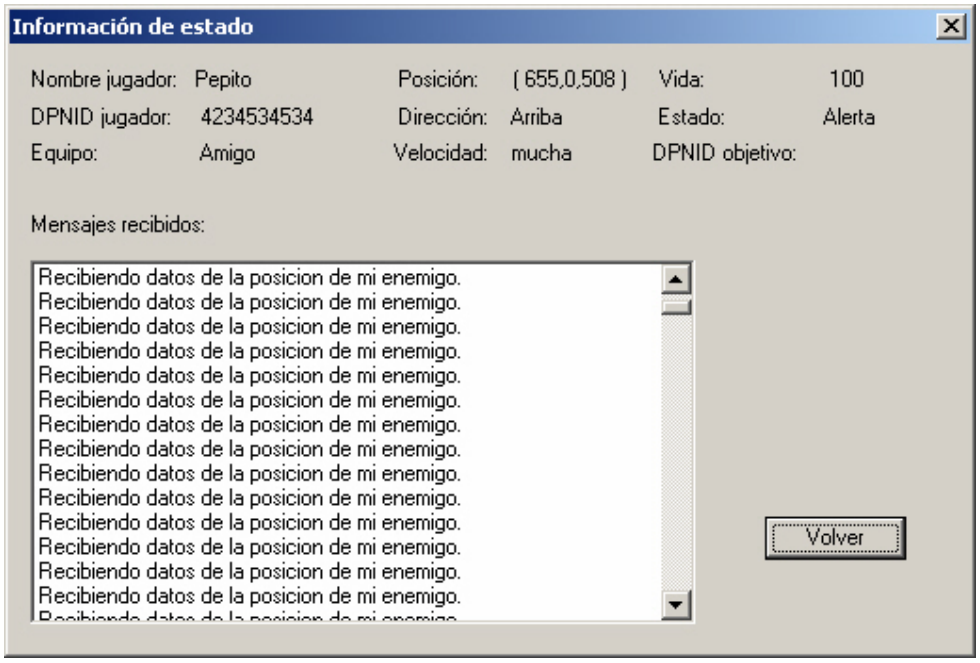


Para unirse a una nueva partida, introduciremos un nombre con el que identificarnos, seleccionamos el tipo de agente que deseemos ser y la dirección IP del servidor. A continuación deberemos pulsar el botón "Buscar partidas". En la lista de servidores, nos aparecerán los servidores activos y entre paréntesis el número de personas que están actualmente jugando en el servidor.

En esta captura podemos ver como el jugador Pepito ha encontrado dos servidores, uno vacío y otro con 5 jugadores, en la IP 172.26.0.2.



Ahora seleccionamos en la lista de servidores aquél al que queremos unirnos y pulsamos el botón "Jugar". Se abrirá una nueva pantalla de información de estado del jugador, donde podemos ver los valores de diversos parámetros del jugador, como su posición, la vida que le queda o en que estado se encuentra.



Para salir de la partida, sólo hay que pulsar el botón "Volver", que nos lleva a la pantalla anterior, y aquí pulsar "Cancelar".

VIII. Glosario

Agente: entidad que tiene la habilidad de funcionar autónomamente, sin requerir instrucciones precisas de un humano y de interactuar con otros agentes.

Cliente: aplicación remota que representa a un agente.

Inteligencia artificial: define el comportamiento de los agentes y como éstos van a interactuar con su entorno.

Servidor: aplicación local que ofrece servicios para la comunicación del los clientes.

Sesión: se habla de la sesión de un juego para referirse a una instancia particular del mismo. Consta de dos o más usuarios que juegan simultáneamente.

Sistema de partículas: es una colección de elementos individuales (partículas) que tienen propiedades individuales como la velocidad, el color y que actúan de forma independiente entre ellas.

IX. Bibliografía

Libros:

K. Hawkins, D. Astle; OpenGL Game Programming

Edward Angel; Interactive Computer Graphics : a top-down approach with OpenGL

I. Parberry; Learn Computer Game Programming with DirectX 7.0

A. Watt, F. Policarpo; 3D Games : real-time rendering and software technology

T. Polack; Focus on 3D Terrain Programming

V.S Subrahmanian, P. Bonatti; Agent Systems

Páginas web:

Desarrollo de juegos con OpenGL:

http://nehe.gamedev.net/

http://gamedev.net/

http://www.gametutorials.com/Tutorials/OpenGL/OpenGL_Pg1.htm

Modelos MD2:

http://www.planetquake.com/polycount/

http://www.quake2.com/modeling/

DirectX 8.0 for C/C++ ; DirectPlay :

http://msdn.microsoft.com/

X. Cesión de los derechos sobre el proyecto

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado".

Santiago Beca Juan Ignacio Plaza Sandra Sánchez