
Francisco Andrés Sanchís Pérez
Juan Rubén Segovia Vilchez
Joaquín Vegara García

# Índice de contenido

# I. ANÁLISIS

# *1. Ámbito del juego*

El proyecto consistirá en la realización de un juego de estrategia 3D en tiempo real (tipo warcraft o Age of Empires) utilizando la librería AllegroGL, junto con openGL. La finalidad del juego es derrotar a los equipos enemigos mediante la destrucción de todas sus unidades usando tu propio ejército y ayudado de los recursos del juego.

El entorno del juego estará ambientado en la época actual, por lo tanto se asumirá la tecnología desarrollada hasta la fecha. Además se dispondrá de unidades civiles que se ocupen de las tareas de mantenimiento y administración de los recursos que existan en el mapa.

## *1.1 Perfilado del juego*

Se trata de un juego de estrategia de partidas cortas (entendiendose por esto que no es un civilization o un heroes of Might and Magic). La duración deseada para una partida es de 15-30 minutos. El motivo de esto es que queremos dotar al juego de un toque de acción (por ejemplo permitiendo a los jugadores poseer cualquier unidad y convertir el juego en un shoot'em up). Es por esto que consideramos necesario que la dinamica del juego sea acorde a nuestra intención, proporcionando partidas rapidas y "sin descanso".

En base a esto podemos marcar una serie de objetivos en cuanto a la dinamica del juego:

- Las unidades deberán crearse en poco tiempo. Esto favorece las partidas con muchos ataques de proporciones limitadas, en lugar de partidas de "preparación de un gran ejercito y aplaste" en las que no tendría utilidad alguna la caracteristica shoot'em up y además se perdería velocidad.
- Las unidades debén morir rapidamente. Cuanto más aguante tengan las unidades la demografía aumentará más. Cuantas mas unidades existan, más fuerza ganará el caracter estrategico y pausado, y menos participará la acción y el shoot'em up.
- Las unidades no deben morir muy rapidamente. Si muriesen muy rapidamente, no se favorecería en absoluto la característica shoot'em up, puesto que el tiempo de vida medio de una unidad sería tan bajo que el juego estaría lleno de interrupciones debido a que la unidad que estaba controlando el jugador ha muerto, rompiendo la inmersión en el juego.
- De los dos puntos anteriores se deduce que la resistencia de las unidades no puede tender hacia ningún extremo, por lo que una vez tengamos el juego hecho, habrá que jugar mucho hasta ajustar el nivel de resistencia para lograr una experiencia óptima.
- Habrá recursos en el mapa por los cuales los jugadores deberán pelear. En nuestro caso habrán yacimientos de petroleo. La recompensa por controlarlos será tener mas dinero para comprar más unidades. Es importante esta faceta puesto que añade puntos estrategicos al mapa, reforzando de esta manera la faceta de estrategia del juego. Para reforzar esto, se deberá construir un edificio sobre el yacimiento (un pozo de extracción), el cuál será facilmente destruible, para así obligar a los jugadores a estar pendientes de defenderlos (a defender el punto estratégico)
- Después estarán los puntos estrategicos de segundo orden, es decir, aquellos que no producen un beneficio concreto, sino abstracto. Estarán definidos por la topología del terreno: zonas elevadas, zonas de paso, etc. Su control le podrá proveer al jugador de ventajas, como ventaja en los combates, o ventaja de movilidad, o significando desventaja a los rivales por falta de movilidad, etc.
- También para reforzar la faceta estrategica (y por que resulta mas realista) se incluyen tiempos de espera para completar las cosas (unidades, construccion de edificios, desarrollo de tecnologias), de modo que sea necesaría una buena capacidad de planificación.

Partiendo de las premisas anteriores, se pueden empezar a concretar más aspectos del juego:

- Una buena forma de conseguir armonizar que las unidades mueran rapido, y que no lo hagan, como se veía antes, puede ser haciendo auntoregeneración. Es decir, las unidades no destacarán especialmente por su resistencia, pero aquellas que sobrevivan a una batalla, podrán retirarse a sanar sus heridas. Para hacer esto, continuamente se subirá el nivel de vida de todas las unidades, pero de forma muy lenta. Como se acaba de decir, las unidades no tendrán mucha resistencia, y expuestas a un combate, el lento sanar pasará por completo desapercibido antes las significativas heridas mortales que reciban las undiades. Por supuesto, sólo sanarán las unidades humanas, no los vehiculos.
- Los vehiculos podrán ser reparados. Una buena idea es que sean reparados por la misma unidad que construye los edificios. La unidad que construye los edificios es el ingeniero. Se le añadiá a los ingenieros la habilidad de reparar vehiculos, puesto que si no, esta unidad dejaría de tener utilidad una vez construida la base. Además puesto que los ingenieros tendrán muy poca resistencia y ataque nulo, llevarlos al combate para reparar los vehiculos en "caliente" dota al juego de nuevas características estratégicas.
- Los ingenieros también podrían tener alguna utilidad en la tarea de recolección de recursos (extracción de petroleo), de modo que se vean ampliadas las demandas de este tipo de unidad.
- En cuanto a la forma de financiación, la mayoría de los juegos de estrategía aplican la dinámica de pagar sólo al construir o reclutar una unidad. Es decir, sin coste de manutención. El warcraft aplica una simplificación del coste de manutención, haciendo que los ingresos de la mina disminuyan de velocidad superados ciertos umbrales de población, pero con cierto matíz muy interesante: Aunque nosotros recibimos menos oro por viaje del peón, en realidad la mina de oro sigue vaciandose a la misma velocidad!, por lo que si tenemos una población alta agotaremos nuestras minas de oro con menor eficiencia, pero si tenemos una población baja, nos estamos arriesgando a ser conquistados... Para nuestro juego sería interesante contar con algún sistema de financiación con posibilidades estrategicas como el anterior, con la manutención reflejada (lo que ayudaría a reducir el tamaño de ejercitos tal y como deseamos) y en el que preferiblemente participen los ingenieros.
- El sistema de financiación al que hemos llegado consiste en constuir pozos de extracción en yacimientos de petroleo, pero estos no producen beneficio por si solos, sino que se pueden meter en el pozo hasta cinco ingenieros. El pozo producirá beneficios por cada ingeniero que haya trabajando en él. El pozo tendrá una opción para sobre-explotar el yacimiento, produciendo dinero de forma más rapida, pero consumiendo el yacimiento de forma más rapida aún. Por ejemplo, un pozo sobre-explotado podría producir beneficio a un ritmo del 150%, pero consumir las reservas a un ritmo del 200%. Además, un pozo sobre-explotado no puede volverse al estado normal.
- En cuanto a la impsibilidad de dejar de sobre-explotar un pozo, es conveniente que se pueda hacer si se destruye el pozo y se vuelve a construir otro pozo de extracción sobre el yacimiento. Esto es debido a que si no un jugador podría sobreexplotar los yacimientos cercanos del otro jugador (en acciones rapidas, después sería seguramente expulsado), de forma que tras ser explusado el jugador rival no podría explonar eficientemente ya sus yacimientos. Esto podría representar el hecho de que la sobreexplotación supone un deterioro en el pozo de extracción y no en el propio yacimiento, un deterioro tal que no puede ser arreglado con "chapuzas", sino derribando y volviendo a construir de nuevo el pozo de extracción.

## *2. Sector al que va dirigido el juego*

Este juego va dirigido al público en general. No tiene una violencia excesiva que lo haga no apto para menores, aunque por su caracter bélico no se recomienda para menores de 7 años.

En concreto va dirigido para gente que "suele" jugar a video juegos. Esto supone el grueso del mercado, y afrontar este sector es una de las formas más directas de obtener beneficios.

Existen otros sectores de éxito potencial, por ejemplo, el sid meiers' pirates ha obtenido un gran exito, sobre todo en lo que se podría denominar "sector ampliado" puesto que es apto para jugadores no habituales, e incluso, personas con pobre dominio informático. Sin embargo, el grueso del comercio se mueve entre los jugadores habituales, un sector que se podría caracterizar por el hecho de ser personas con un buen nivel de informática a nivel usuario, jovenes, con predilección por géneros de acción o "violentos" aunque incluyendo juegos de genero como los juegos de rol o de estrategia.

Combinando en un mismo juego (y esperemos que de forma acertada) el reto de un juego de rol, y la adictividad e inmersión de un juego de accion shoot'em up, esperamos hacer este juego especialmente atractivo para dicho sector de jugadores habituales.

Es por ello que se ha hecho especial enfasis en lograr introducir accion al juego, pero sin perder el nivel estratégico.

# *3. Requerimientos*

## *3.1 Requerimientos hardware*

### 3.1.1 Requerimientos hardware recomendados

Pentium IV 2000MHz o equivalente, tarjeta gráfica ATI Radeon 9200, 256 RAM, ratón, teclado, monitor 1024x768, altavoces, tarjeta de sonido.

### 3.1.2 Requerimientos hardware mínimos

Pentium IV 1300MHz o equivalente, tarjeta gráfica compatible openGL, ratón, teclado, monitor1024x768.

## *3.2 Requerimientos software*

Windows 98, Me, XP, drivers actualizados de la tarjeta gráfica,

## *4. Herramientas a utilizar*

Dev-C++ para compilar el proyecto.

Allegro y AllegroGL para soporte de rutinas de bajo nivel. (Graficos, sonido, input)

Lib3ds y Cal3D para la carga y visualización de modelos y modelos animados de 3D Studio MAX.

3D Studio MAX para modelización de unidades, edificios, etc.

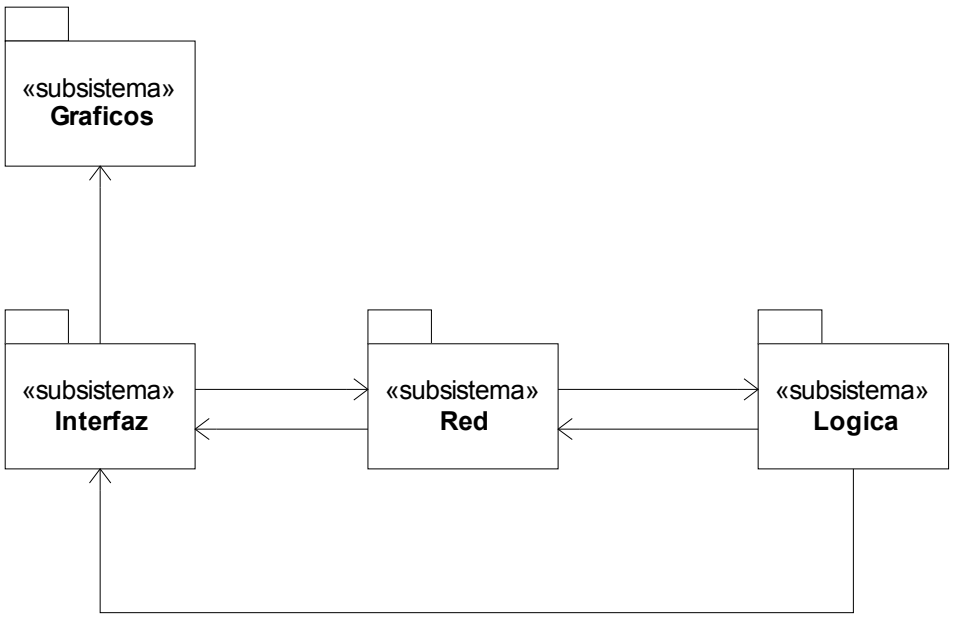
Gimp y Photoshop para el diseño de las texturas e imágenes.

# II. DISEÑO

# *5. Arquitectura del juego*

## *5.1 División modular*

El proyecto se divide en varios módulos que realizan tareas diferentes. Se tiene el módulo de graficos, el módulo de interfaz, el módulo de lógica y el módulo de red. En el código, cada módulo se define en un namespace propio, y los ficheros de dicho modulo empiezan todos por una letra minúscula que identifica el módulo. Además cada módulo proporciona una clase llamada Api para su interrelación con el resto de módulos.




En el esquema anterior se observa que el modulo de red hace de comunicador entre el módulo de lógica y el módulo de interfaz. Aunque se verá más adelante es conveniente destacar ahora que el módulo de red no implica una conexión de red, sino que sirve de comunicador abstracto entre lógica e interfaz, permitiendo que las diferencias entre una partida local, y una partida de red queden generalizadas por completo dentro de dicho módulo sin tener que modificar el codigo externo.

El módulo de gráficos proporciona una Api amigable para la creación y manipulación de elementos 2D de GUI, los cuales utiliza el módulo de interfaz. Asimismo proporciona métodos para la visualización 3D como el control de la camara, carga del terreno, modelos, y visualización de objetos que son igualmente utilizados por el módulo de interfaz para visualizar el área de juego.

El módulo de interfaz, muestra los menús del juego, y en su momento, según se inicie una partida local, o en red, inicializará el sistema de red inidicándole si va a ser una partida local o en red. Además, si va a ser una partida local, o se va a ser el servidor de una partida en red, se inicializa también el módulo de lógica.

Es el módulo de lógica el que guarda información sobre los jugadores existentes en la partida, y todas sus unidades, simulando el mundo, y enviando notificaciones de cambios al módulo de interfaz para que se encargue de su visualización.

Toda la comunicación Interfaz – Lógica se realiza a traves del módulo de Red, salvo unas pocas excepciones, en las que el módulo de lógica accede directamente a la Api del módulo de Interfaz, para realizar ciertas consultas (consultas de configuración del juego, o cosas como pedir que se calcule si un disparo impacta a alguna unidad), son en definitiva todas aquellas consultas que la Logica tenga que realizar a la Interfaz.

## *6. Graficos*

Este módulo proporciona una api sencilla e intuitiva al modulo de interfaz simplificandole sus tareas, de modo que el módulo de interfaz se pueda centrar en la "lógica subyacente a la interfaz" mientras este módulo se encarga de las rutinas de visualización de bajo nivel (allegro y opengl) Incluye también algo de control de sonidos, pero es tan poca la fracción del total que esto supone que el nombre que de forma más intuitiva representa a este paquete es "Graficos" y por eso se ha utilizado en lugar de algún otro nombre más general como pudiera ser "Media".

Para este módulo se tiene una interfaz representada por la clase Api. Es decir, es la clase dominante del paquete, ya que todas las interactuaciones con el módulo de lógica se realizan a través de esta clase.

### *6.1. Imagen*

Las imagenes son uno de los elementos básicos del módulo de graficos, representa una imagen que puede ser pintada en la pantalla. No obstante, la api del módulo graficos no permite pintar imagenes en pantalla directamente, sólo cargar y liberar imagenes.

Se pueden crear imagenes de varios tipos (varios constructores). La forma más típica será pasando el path del archivo a cargar, aunque también se puede utilizar un constructor que recibe una Fuente y un string, y crea una imagen con el texto escrito con dicha fuente. Existe además otro constructor parecido a este ultimo que además recibe tres parámetros r,g,b indicando el color con el que pintará el texto.

Cuando se crea una imagen, (bitmap de allegro) inmediatamente se convierte en una textura de openGL y se libera el bitmap que se había creado. El unsigned int que representa la textura de allegro es guardado en el objeto Imagen. Antes de destruir el bitmap se guarda también el ancho y alto del bitmap en el objeto Imagen (puesto que una vez convertido a textura esta información se pierde) y es utilizado posteriormente cuando se pinta la imagen en pantalla poder pintarla con el tamaño original.

Es a nivel de imagen donde se guarda si una imagen se debe pintar en modo máscara o no. Es decir, si el color rosa debe ser tenido en cuenta como transparente o no. Esto afecta unicamente cuando la Imagen se pintada con el método Pintar(), y hará que se llame al método Blit o BlitMascara de la Api. Cuando una imagen se pinta en modo máscara, lo que se hace es activar el GL_ALPHA_TEST, para que se ignoren aquellos pixeles cuya componente alpha sea menor que 0.5. La textura de openGL de las imagenes es creada de tal forma que la componente alpha de los puntos rosas vale 0.0, y del resto vale 1.0.

Para pintar la imagen se llama al método Pintar(int x, int y), que pintará la imagen en las coordenadas x, y de pantalla (teniendo en cuenta si se tiene que pintar en modo máscara o no con el estado almacenado en la propia Imagen). También hay disponible el método PintarTransparente(int x, int y, int alpha) que pinta la imagen con el nivel de opacidad indicado en la variable alpha (Aunque este método no utiliza GL_ALPHA_TEST, utilizar GL_BLEND, por lo que aquellos puntos con alpha 0.0, es decir, que eran de color rosa, no se pintaran en pantalla, por lo que no se puede tener puntos de coor rosa puro en imagenes que vayan a ser pintadas de este modo).

Aunque se acaba de describir los métodos que pintan una imagen no está pensado para que sean llamados desde fuera del módulo de graficos (de hecho ningun objeto sale de la capa de abstraccion que supone la API, pasandose al exterior únicamente handlers; puesto que la clase Api no proporciona ningún método para pintar imagenes esto no puede ser realizado en condiciones normales). Si un usuario crease por su cuenta un objeto Imagen y decidiese llamar al método Pintar el resultado no está definido, por lo que no debería seguirse esta práctica, lo más probable es que ni siquiera se viese la imagen en pantalla, puesto que no estaría establecida la proyección y matriz de vista de modelos apropiadas.

### *6.2 Fuente*

Esta clase representa una fuente de texto. Tiene un constructor que recibe un nombre de fichero del que cargar la fuente de letras. El constructor por defecto crea un objeto Fuente que encapsula la fuente por defecto de allegro.

Esta fuente por defecto es la única que se utiliza en el juego. Más adelante ya le pondremos fuentes de "diseño".

La forma en que los textos son llevados desde la fuente a pantalla es la siguiente. Se crea una fuente (o se dispone de una Fuente), se pasa la Fuente y el string al constructor de Imagen, con lo que se crea un Bitmap de allegro, y se convierte dicho bitmap en una textura de openGL tal y como se describia en la sección de Imagenes.

Para permitir estas acciones sobre la fuente de allegro encapsulada por el objeto Fuente, se provee de un método GetFont() que devuelve dicha fuente de allegro para ser utilizada por el constructor de la clase Imagen.

De momento el juego utiliza por entero la fuente por defecto, el cambio de dicha fuente por otras más vistosas es algo que queda pendiente para el futuro.

### *6.3 Entidad2D*

Esta clase es la clase base de todas las entidades 2D pintadas en la pantalla (sprites, text boxs, botones, etc), almacena las coordenadas de la entidad, en especial la coordenada z, que indica que objetos 2D se pintan encima de otros. Tiene un control automático de todas las entidades2D creadas, y provee de un método estático para pintarlas todas, por orden de coordenada z. Para ello recorre una lista dinámica ordenada por z, que contiene un puntero a cada Entidad2D creada, y llama al método Pintar de cada Entidad2D. Cada clase derivada redefine el método pintar.

El control de Entidades 2D existentes se lleva a cabo mediante una lista enlzada de la STL. Esta lista es estatica de la clase Entidad2D, y en el constructor de Entidad2D se inserta el puntero this en la lista, respetando el orden por z. Para ello las clases Entidad2D definen el operador >= que es el que se utiliza para la inserción ordenada en la lista, y que lo que realiza es la operación >= sobre las variables z de cada objeto.

En el destructor de Entidad 2D se elimina el puntero a la unidad de la lista.

Una entidad 2D puede ser desactivada, la entidad sigue existiendo y almacenando sus valores, pero no es pintada en pantalla. En cualquier momento puede volver a ser activada.

La api del módulo gráfico no permite la creación directa de Entidades 2D.

Sería interesante añadirle un metodo para Actualizar todas las entidades 2D, es decir, separando la lógica de la visualización en las entidades 2D. (Por ejemplo la clase boton realiza el control de eventos en el propio metodo Pintar, esto trae la desventaja de que los objetos no se actualizan si no están activados, o la necesidad de pintar los graficos a la misma frecuencia que se actualiza la interfaz del juego). Por estos motivos en breve será separada la lógica de la visualización en las entidades 2D.

### *6.4 Segmento : Entidad2D*

Un objeto Segmento representa como su nombre indica un segmento, una porcion de recta. Es decir, una linea recta con principio y final. Un segmento es derivado de Entidad2D, y las coordenadas x, y se corresponden con las coordenadas de inicio del segmento. El final del segmento viene definido por los valores finX y finY.

Un segmento puede tener cualquier color RGBA, aunque no está soportado que cada vertice del segmento tenga un color distinto, es posible que sea añadido en breve.

Cuando es pintado un Segmento (por ser una entidad2D tiene un metodo redefinido Pintar() ), lo que se hace es pintar una linea con opengl utilizando el modo GL_LINE.

Además los objeto Segmentos pueden pasar una etapa de clipping. El clipping permitido es rectangular y se utiliza una simplificacion de un algoritmo basado en zonas. Para cada punto se comprueba si se encuentra abajo de la ventana de clip, arriba de la ventana de clip, a la izquierda de la ventana de clip, o a la derecha de la ventana de clip (pudiendo darse 0, 1 o 2 casos para cada vertice). Si los dos vertices estan fuera de la ventana en la misma zona (algun test ha dado positivo para ambos vertices) el segmento no se pinta. Si los dos vertices estan dentro de la ventana (ningun test ha dado positivo) el segmento se pinta tal cual. Si uno de los dos vertices está fuera de la ventan y el otro dentro, se calcula la intersección con la ventana y se pinta el segmento truncado. En este algoritmo se ha obviado el caso en que estando los dos vertices fuera de la ventana (en zonas distintas), si que hay un fragmento interior visible en la ventana. Se hará próximamente, aunque temporalmente no se ha incluido por el hecho de que para los casos en los que se utilizan los segmentos no ha sido necesario contemplar este caso.

El grosor de las lineas pintadas con esta clase es de una unidad. Esto podrá cambiar en breve para convertirse en un valor parametrizable. Las posiciones del segmento, tanto el vertice inicial como el final, pueden ser cambiados en cualquier momento. El clipping no se puede asignar en tiempo de construcción, sino con una llamada posterior a SetClipping(int x0, int y0, int x1, int y1). No es posible desactivar el clipping sobre un segmento una vez ha sido activado, aunque es probable que sea añadido en un futuro.

Los segmentos han sido utilizados para dibujar el área visible en el minimapa del juego.

### *6.5 Rectangulo : Entidad2D*

Esta clase representa un Rectangulo que es pintado en pantalla. Como entidad2D tiene los parametros x,y que representan la coordenada superior izquierda del rectangulo. Tiene además los parámetros ancho,alto que indican las dimensiones del rectangulo (Los parámetros ancho y/o alto pueden tener valores negativos, pintandose entonces el rectangulo hacia la izquierda y/o arriba)

Un rectangulo puede tener cualquier color RGBA. Además se puede cambiar el color de un rectangulo en cualquier momento. También se pueden cambiar las dimensiones del rectangulo y, puesto que una Entidad2D permite cambiar sus coordenadas, también puede el rectangulo al utilizar dichas coordenadas como punto de origen.

Un rectangulo puede ser pintado con o sin relleno, y esto viene indicado por la variable booleana relleno, que se le es pasada en el constructor y que se puede cambair en cualquier momento con el método SetRelleno(bool).

Un rectangulo es pintado como un QUAD. Si el rectangulo no debe ser rellenado, antes de dicha operacion se cambiara el modo de renderizado de poligonos con glPolygonMode (GL_FRONT_AND_BACK, GL_LINE). Además se ha establecido el ancho de las lineas a 2 unidades. Esto en el futuro podría cambiar y convertirse en un valor parametrizable.

Los rectangulos han sido utilizados para mostrar el color de cada jugador en la pantalla de configuración de la partida.

También se utiliza un rectangulo con el relleno desactivado para pintar el rectangulo de selección de unidades.

### *6.6 BarraVida : Entidad2D*

Esta clase representa una barra de vida. Una barra de vida basicamente es un rectangulo con una altura fijada, y un ancho variable. Además el color de la barra de vida cambia automáticamente según cual sea la relación entre el ancho actual y el ancho máximo.

Al construir una barra de vida se le pasa el alto, ancho, y borde. El borde indica cuantas unidades se deben dejar a cada borde sin pintar el rectangulo. En realidad debajo de la barra de vida se pinta un rectangulo negro con transparencia, que sera visible en la medida que haya más o menos borde.

En cualquier momento se puede asignar un porcentaje a la barra de vida, significando que se debe pintar solo dicho porcentaje del ancho máximo de la barra.

El color de la barra de vida depende del porcentaje, siendo roja para un 0%, amarillo para un 50% y verde para el 100%, haciendo una interpolacion para los valores intermedios.

La barra de vida es pintada con dos quads de opengl, lo que logra un buen efecto con una carga mínima.

La barra de vida se utiliza para mostrar la vida de las unidades cuando el raton se pone encima de ellas en el area de juego.

También se pintan barras de vida debajo de los retratos de unidades en el area de unidades seleccionadas.

Es utilizada también al empezar el juego, en la pantalla de carga, para mostrar el progreso de carga.

### *6.7 Sprite : Entidad2D*

Un sprite es una entidad2D al que en el momento de construirlo se le pasa un puntero a una Imagen. El sprite, redefine el método Pintar de Entidad2D para hacer que aparezca en pantalla la imagen con la que ha sido construido en las coordenadas en las que se encuentra el sprite.

La api del módulo gráfico permite la creación y manipulación de sprites, tales como modificar su posición, o activar o desactivar el sprite.

A diferencia de una imagen, un sprite representa una imagen que se encuentra en pantalla. Puede crearse más de un sprite para una misma imagen y diferenciarse por tanto en las coordenadas en las que son pintados.

Un sprite no posee mayor lógica que la instanciar una imagen, y encargarse de pintarla en cada frame en la posición del sprite. Para cualquier modificación de la imagen, como definirla imagen con mascara, se deber realizar directamente sobre la imagen.

Un sprite no realiza una copia de la imagen, sino que pinta la imagen con la que ha sido creado en cada frame, es por esto que si se destruye la imagen y no se destruye un sprite que la utiliza se producirá un error no controlado.

Puesto que no se genera copia de la imagen origen, si se modifica una imagen instanciada por varios sprites, la modificación afectará a todos los sprites.

Un sprite desactivado deja de pintarse en pantalla hasta que es activado de nuevo, conservando todas sus propiedades. Además si se modifica el estado de un sprite mientras permanece desactivado (por ejemplo se mueve de sitio), dichos cambios serán visibles cuando se vuelva a activar el sprite.

Los sprites son utilizados en todas las partes del juego para poner imagenes estaticas en pantalla. Apenas se ha utilizado la posibilidad de desplazamiento de sprites (que en un juego 2D por ejemplo habría sido clave)

### *6.8 Boton : Entidad2D*

Un boton es una entidad2D que se crea a partir de una imagen del boton, una imagen de selección, una fuente de texto, y un texto. El boton pinta la imagen de fondo, y el texto escrito con la fuente de texto pasada en el centro de la imagen. Cuando el raton pasa por encima de la imagen, pinta ademas la imagen de selección al 50% de opacidad. Si además de esto se pulsa el boton del raton se eleva el nivel de opacidad al 75%.

Hay que resaltar que el texto del boton es generado por el propio boton, creando una imagen a partir de la fuente que recibe y el texto. No se puede pasar directamente la imagen de texto al boton, debe hacerse de esta manera. El boton se encarga de destruir dicha imagen de texto en su destructor (puesto que es un recurso que ha creado él). El resto de imagenes que recibe deberán ser liberadas por el usuario.

Al igual que el sprite, el boton no controla la forma en que se pintan las imagenes (modo mascara o no), estas características deben ser definidas directamente sobre las imagenes que componen el boton. Sin embargo, el boton utiliza el metodo PintarTransparente sobre la imagen de selección, por lo que tal y como se comentaba en la sección de Imagen, no podrá contener puntos de color rosa puro (ya que serán tratados como transparentes por el blending).

Se le puede indicar al boton que avise de evento de click, pasandole un puntero a bool, que el propio boton pondrá a true cuando se produzca un click, y a false cuando pase el evento. El evento no finaliza cuando se libera el raton, sino en el ciclo siguiente al de producirse el click.

También se le puede indicar que avise del evento MouseOver, en el que el boton pondrá a true un puntero a bool recibido cuando el raton pase por encima del botón. En este caso el bool permanecerá a true mientras el raton permanezca encima del boton, y se pondrá a false cuando el ratón abandone el area del boton.

Se debe tener en cuenta que el area del boton es calculada tomando el area de la imagen de fondo, sin tener en cuenta areas transparentes, es decir, puede que no sea muy fino al detectar colisiones con botones no rectangulares.

Los botones son utilizados ampliamente en el juego. Los menus están plagados de botones. Tanto los habituales botones de navegación en la parte izquierda, como los botones para la configuración de las opciones de video.

Se utilizan objetos Boton para los botones de acciones de unidad. Los retratos pequeños de unidad son botones. Incluso el retrato grande de unidad es un boton también.

El minimapa esta implementado como un boton también.

En la barra superior hay un boton para volver al menu.

### *6.9 BarraProgreso : Entidad2D*

Esta clase representa la típica barra de progreso. Se crea pasandole al constructor una imagen de fondo, una imagen de progreso, el tamaño del borde, y la separación.

La imagen de progreso es el "cuadro" que va a ser pintado repetidamente hasta rellenar la imagen de fondo.

El borde es la distancia que se deja en cada margen para pintar las imagenes de progreso. La separación es la distancia que se dejará entre cada imagen de progreso. Estos valores pueden ser 0, o incluso tener un valor negativo.

Para indicar cuantas imagenes de progreso pintar se llama al metodo SetPorcentaje, el cual calculará cuantas imagenes enteras caben en dicho porcentaje. Es decir, las imagenes de progreso no se cortan, o se pintan 4, o se pintan 5, pero nunca 4'7.

Las barras de progreso se utilizan en el juego, para mostrar el progreso en la construcción de alguna unidad, o en el desarrollo de alguna tecnología.

### *6.10 TextBox : Entidad2D*

Un textbox es una entidad2D, que tiene una imagen de fondo, y una fuente de texto. El textbox muestra un texto encima de la imagen de fondo, y permite la activación de dicho textbox con el raton y la modificación del texto que aparece leyendo el teclado.

La api proporciona además de los habituales, metodos para establecer y obtener el texto actualmente mostrado en un textbox.

Para pintar el texto, se crea una imagen con la fuente del textbox y el nuevo texto. Cada vez que el texto es cambiado se destruye la imagen anterior y se vuelve a crear otra con el nuevo texto. Aunque se crean objetos Imagen para la generación del texto, no se crean objetos sprite para pintarlos, sino que esto es hecho directamente por el TextBox.

El usuario puede establecer el texto en cualquier momento. El jugador también puede modificar el contenido del textbox en cualquier momento del juego.

El caracter de cursor es el simbolo de subrayado "_", y es mostrado cuando se esta en modo de edición. También es mostrado cuando el raton pasa por encima del TextBox (ocultandose de nuevo cuando el raton sale del area del TextBox).

De momento no está soportado que el TextBox reciba el "focus" directamente, sin el hecho de que el ratón haga click en el. Esto es necesario en algunas partes del juego y será añadido en breve.

Existen ciertos bugs en el TextBox por los cuales es posible que deje de recibir modificaciones del texto, o que no se actualize el texto visible aunque internamente sí que lo esté haciendo. Será solucionado en breve.

Los textbox se utilizan para que el jugador introduzca su nombre, o la ip de la partida a la que se quiere conectar.

También se utilizan como elemengo GUI de entrada de mensajes para el chat, tanto en la ventana de configuración de partida como dentro del juego. Dentro del juego el textbox permanece desactivado hasta que se pulsa enter, es ahi donde sería conveniente hacer un focus, para que el jugador no tenga que perder tiempo clickando en el textbox, ya que es evidente, que si ha pulsado enter, es porque desea escribir en ese momento.

### *6.11. ChatBox*

El ChatBox representa un TextBox multilinea, de solo lectura, en el que se pueden realizar operaciones de adición de lineas de texto.

Cada linea añadida puede tener un color distinto, lo que es utilizado para mostrar los mensajes de cada jugador de su color.

Es creado con una imagen de fondo, una fuente y las dimensiones del chatbox. La imagen de fondo puede ser NULL, en cuyo caso no se pintará imagen de fondo.

Cada linea que se muestre lleva asociado el número de frames que ha sido mosrada, una vez supere un umbra, empará a desaparecer en un efecto de alpha fade off.

El ChatBox se utiliza para mostrar los mensajes de chat en la pantalla de configuración de partida.

El ChatBox se utiliza en el juego (sin imagen de fondo) para mostrar los mensajes de chat.

### *6.12. ListBox : Entidad2D*

Un listbox permite la selección entre varias opciones textuales. Se crea a partir de una imagen de fondo, uma imagen de opcion seleccionada, y una imagen de seleccion, además de una fuente de texto que se utilizará para escribir las opciones. La imagen de opcion seleccionada se pinta debajo del texto seleccionado, y la imagen de seleccion se pinta al 50% de opacidad encima del texto que se seleccionaría si se pulsara el boton del raton.

La api proporciona metodos para añadir nuevas opciones y para obtener el número y texto de la opción seleccionada. Además se puede pedir que informe del evento de opción cambiada pasando un puntero a bool. También se le puede pasar un puntero a int para que mantenga actualizado el indice de la opción seleccionada.

El valor de indice de opcion seleccionada se mantiene siempre actualizado, sin embargo, el bool que recibe el vento de opción cambiada es reseteado a false en el ciclo siguiente a producirse.

Actualmente no está implementado pero se permitirá hacer el desplazamiento del area visible de listbox con una barra de scroll en la parte derecha del listbox. Esto se hará en un futuro, probablemente no muy cercano.

El listbox se utiliza unicamente para la selección del mapa que se va a jugar.

### *6.13. Modelo*

Un modelo es la analogía 3D de una imagen. Es decir, es aquello que puede ser pintado en pantalla. Se puede crear un modelo pasandole un string con el nombre del fichero que contiene la información del modelo. También se puede crear un modelo pasandole un identificador de CallList de openGl.

La api del módulo de gráficos permite crear un modelo a partir de un fichero 3ds, o crear un bitmap 3D, o quad, para poder pintarlo en 3D.

Los detalles de implmentación del modelo se detallan ampliamente en la sección de implementación.

La clase Modelo deriva de BaseModelo. Esta pensado para que hayan clases derivadas de BaseModelo tales como Modelo, ModeloAnimado, etc, cada una optimizada para la carga y renderizado de diferentes tipos de modelos. Puesto que actualmente sólo estan implmentados modelos estaticos, solo existe la clase Modelo, y ni si quiera se hace uso de su clase base Modelo, esto debería cambiar en un futuro cercando, cuando la carga de modelos dinámicos permita establecer las generalidades entre ambas clases.

Los modelos se crean para cada unidad del juego.

También se crea un modelo de tipo quad para cada equipo con el anillo de seleccion con su color.

### *6.14. Entidad3D*

Una Entidad3D es la clase base de todos los objetos 3D que son pintados en la pantalla. Almacena la posición de la entidad, así como parámetros de control como si está activada (si se debe pintar o no en la pantalla), si es transparente, o si es siempre visible. El hecho de saber si una entidad 3D es transparente o siempre visible a nivel de entidad 3D es necesario ya que las entidades 3D deben pintarse primero las que no son ni transparentes ni siempre visibles, después las transparentes pero no siempre visibles, y por último, las siempre visibles con el DEPTH_TEST desactivado.

La clase Entidad3D proporciona un método PintarTodos igual que hacía Entidad2D, peró resulta de mayor utilidad el método PintarTodosEn que recibe las coordenadas de un rectangulo en el plano y=0, y realiza un clip de todos aquellos objetos que no sean interiores a dicho rectangulo tras aplicacar una proyección con Base:=(y=0) y Direccion:=(x=0, z=0), es decir, todos aquellos objetos cuyas coordenadas x,z estén dentro del rectangulo especificado.

El método anterior es util por su sencillez, sin embargo pronto se quedó pequeño para los requerimientos del juego. Hacía falta algún método más efectivo de clipping, para lo cual se incluyo en las clases métodos estaticos para controlar un clipping utilizando el select buffer de opengl aplicado sobre bounding boxs. Esto se detallará ampliamente en la sección de implementación. Una vez se han preparado las entidades 3D con la información sobre el clipping, se puden pintar solo aquellas que sean visibles llamando al método PintarTodosClipping().

Es interesante para el método de clipping el método estatico PintarTodosBoundingBox, que va llamando al método PintarBoundingBox de todas la entidades3D, de modo se puedan enviar los bounding boxs de forma eficiente.

Está planeado volver a modificar el algoritmo de clipping para que no incluya llamadas opengl, y comprobar si se gana velocidad de esta forma. Además, puesto que las implementaciones opengl son libres en gran parte, utilizarlo para una seccion tan usada como el clipping podría traer problemas de ejecución. Es muy probable que en breve se implemente un algoritmo usando gluProject aplicado sobre los 8 vertices del bounding box, que tan buenos resultados ha dado en el clipping del terreno.

### *6.15. Objeto3D*

Un objeto 3D representa un modelo que está en algún sitio (es la analogía 3D de sprite), se construye pasandole un modelo, que será el que pintará. Redefine el método pintar de la entidad 3D para adecuar la matriz MODELVIEW de forma que el objeto se situe en las coordenadas y orientación correcta.

Recibe además un nombre de opengl que será el que cargará en la lista de nombres antes de pintar el modelo (y sobre todo antes de pintar el bounding box del modelo) que sirve para las operaciones de selección de listas de opengl. Cada objeto tiene un nombre distinto, que equivale a su posición en el array de objetos del Api.

Se puede modificar en cualquier momento su orientación en el eje Y y cambiar su escala. Estas transformaciones serán realizada antes de llamar al método Pintar del modelo, y la matriz modelview será recuperada tras la llamada.

Como entidad 3D tiene métodos para indicar si es transparente y/o siempre visible, lo que afectará al momento en que será llamado por el método PintarTodos (o derivados) de Entidad3D.

También tiene interés la posibilidad de indicar si el objeto es seleccionable o no, un objeto que no sea seleccionable no será tenido en cuenta en las selecciones de listas de opengl, por lo tanto, cuando se clica sobre una unidad, la funcion concreta devolverá unicamente el handler de dicha unidad, sin tener en cuenta si en el camino del ratón había también un arbol de decorado, al cual obviamente el jugador no desea dar ninguna orden.

Se pude indicar si el objeto es bloqueante o no. Esto tiene efecto en la colocación de edificios, de tal forma que se podrá colocar un edificio en una zona (al margen de la topología del terreno) unicamente si no colisiona con ningún objeto bloqueante. Por ejemplo, las unidades no son bloqueantes y por tanto se puede construir edificios encima de ellas (se ha de suponer que las unidades se apartaran). Para esto la clase Objeto provee de un método Colision que recibe el puntero a otro objeto y determina si los bounding boxs de ambos objetos colisionan.

Se puede asignar una mascara de escritura al objeto, lo cual hace que el objeto sea pintado con una máscara de opengl de escritura (Las mascaras de escritura especifican que canales del framebuffer se pueden actualizar: rojo, verde, azul, alpha). Esto se utiliza cuando se está situando un edificio nuevo, y en la posicion actual no se permite su construcción, entonces se pone una máscara que solo permite actualizar los canales rojo y alpha, pintando del edificio unicamente su componente roja.

Se puede obtener la altura de un objeto (se devuelve la altura del bounding box del modelo al que está enlazado el objeto), esto se utiliza para saber a que altura pintar la barra de vida de la unidad.

Se puede obtener el radio del objeto (se devuelve el radio del bounding box), esto se utiliza para saber que tamaño debe tener el anillo de selección.

Por supuesto tiene los metodos Pintar y PintarBoundingBox redefinidos de Entidad3D.

Los objetos se utilizan para las unidades del juego y son creados en el constructor de cada unidad, con el modelo ya cargada al principio de la partida para ese tipo de unidad y color.

También se usan objetos para mover los anillos de selección de cada unidad por el mapa al mismo tiempo que la unidad. Toda unidad tiene un anillo de selección, que activa cuando debe ser mostrado (está seleccionada) y desactiva cuando no.

### *6.16. Terreno*

La clase terreno soporta la creación del terreno de juego, y metodos de consulta como obtener la altura de una posición.

Al crear el terreno se debe especificar si se desea crear un terreno multitexturizado o no. Esto es así puesto que la multitexturización no está soportada por gran cantidad de tarjetas gráficas (sobre todo nVidia) además de suponer un consumo amplio de la tarjeta que podría afectar a los modelos más antiguos. Por este motivo es posible desactivar el multitexturizado del terreno en el menú de configuración de video.

Se puede solicitar la altura discreta (directamente obtenida del mapa de alturas) para una coordenada discreta (en el sistema de coordenadas del mapa de alturas) con el método GetAltura (int x, int y).

Se puede asi mismo obtener la altura de un punto concreto del mundo con el método GetAlturaSuave(float x, float z). Este método realiza la intersección de la linea vertical que pasa por el punto (x,0,z) con los triangulos que conforman el terreno para obtener la altura exacta de dicho punto.

Se puede indicar al terreno que "ha llegado el invierno" o que "ha acabado el invierno" con el método SetInvierno(bool), el terreno entonces hara una transición de estación. Dicha transición funciona tanto con el terreno multitexturizado como con el terreno sin multitextura.

El método Pintar de terreno debe ser llamado explicitamente cada ciclo, puesto que no se trata de una entidad3D. Por ello es llamado en el propio método Pintar de la Api.

La generación y renderizado del terreno se discute en amplitud en la sección de implementación.

### *6.17. Sonido*

Esta clase muy sencilla se encarga de la reproducción de sonidos. Los sonidos representan aquello que puede ser reproducido en los altavoces.

Se crea pasandole el nombre del fichero a cargar.

Existen dos formas de reproducir un sonido, con el método Play() y PlayLoop(). El primero reproduce el sonido una vez y el segundo lo reproduce ciclicamente de forma infinita.

De momento no se puede dentener un sonido, ni aunque se este reproduciendo de forma cíclica. Para ello en breve se dispondrá de un método Stop.

Es probable que en le futuro se permita instanciacion de sonidos, en lugar de reproducir directamente los sonidos. Esto permitira tener el mismo sonido reproduciendose varias veces a la vez sin necesidad de cargarlo más de una vez en memoria.

Sin embargo todos estos cambios se verán pospuestos hasta que se realize la migración del código de sonido a openAL o DirectSound9. Estamos aún evaluando que librería elegir.

## *6.18. Api*

Es la clase dominante del modulo y sirve de capa de abstraccion del mismo. Permite la creación de objetos del módulo.

Jamás se devuelve un objeto a fuera del módulo. En lugar de esto se devuelve un handler al tipo de objeto. Los handler son números enteros. Ese handler es el que se tendrá que pasar como parámetro en todas las interactuaciones con el objeto que representa.

No se va a repetir ahora la explicación de las clases del módulo, sin embargo, la clase Api proporciona una serie de funcionalidades extra aparte de la interactuación con objetos del módulo que se describirán aqui.

### 6.18.1 Bliteo de imagenes

La api proporciona algunos métodos para bliteo de imagenes 2D. Existen varios métodos que permiten el bliteo de imagenes en diferentes modalidades. Se tiene el blit normal, el blit con máscara, o el blit con transparencia (la imagen se pinta con un nivel global de opacidad).

Para más información consultar la seccion de Imagen.

Puesto que sólo la clase Imagen hace uso de estos métodos (y el resto de objetos utilizan objetos de clase Imagen para pintar en pantalla), es probable que en el futuro estos métodos se eliminen y se introduzca el codigo de bliteo en los propios métodos Pintar de la clase Imagen.

### 6.18.2 CallLists

Una gran cantidad de objetos utilizan callLists para el renderizado. Para gestionar esto, en lugar de usar directamente los métodos de openGL se ha usado una serie de métodos de la clase Api que realizan dicha gestion.

Esto se ha hecho así ya que inicialemente los nombres de objetos se correspondian con el identificador de lista de opengl, y estos debían corresponder con los handlers de objetos, por lo que debía reservarse los nombres de listas del 1..n, siendo n el máximo de objetos permitidos. El resto de objetos (todos menos Objeto) deben llamar al método ReservarCallListAlta() que devuelve un callList libre superior a n.

Esto ha dejado de ser necesario, puesto que no es necesario que los objetos compartan nombre e identificador de lista, por lo que es probable que se deje de utilizar, al menos ReservarCallListAlta.

Si se mantendrá sin embargo ReservarCallList puesto que nos permite tener un control de la eficiencia del algoritmo que con opengl no se puede tener debido a que todo lo relacionado con las listas de render es de implmentación libre.

### 6.18.3 Clipping

El método CalcularClippingRect() se encarga de determinar que objetos entran en el area de visión de la cámara y por tanto serán pintados en el frame.

Lo primero que hace es resetear el clipping, llamando a ResetClippingRegs(), esto hace que el valor de clipping de todas las entidades 3D se ponga a false.

Después renderiza los bounding boxs de todas las entidades 3D llamando a Entidad3D::PintarTodosBoundingBox(). Esto se hace en modo GL_SELECT, por lo que tras esto se dedica a determinar que objetos han aparecido en pantalla y a poner su clipping reg a true.

Una vez se ha calculado el clipping, la funcion Entidad3D::PintarTodosClipping() solo tiene que comprobar el clipping reg de cada entidad 3D para decidir si pinta o no el objeto.

### 6.18.4 Camara

La api proporciona además metodos para el posicionamiento de la camara, es decir, para la transformación de la matriz de modelview.

Existen dos conjuntos de instrucciones para esto, la api antigua, con SetPosicionCamara y SetInclinaciónCamara; y la moderna, con funciones como ResetCamara, EscalarCamara, RotarCamara, TrasladarCamara.

Se ha cambiado al nuevo sistema, en el cual como se puede observar las funciones encapsulan transformaciones directas de la matriz de modelview debido a que la api antiguo resulto ser poco potente y flexible.

Sin embargo sigue siendo necesario que se utilicen los dos métodos puesto que hay partes del módulo de gráficos que aún utilizan los valores almacenados por la api antigua para trabajar. Estas partes seran migradas al nuevo sistema progresivamente y se espera que en breve se pueda eliminar el antiguo.

La camara es establecida por la clase Interfaz::Camara, que se encarga de situar la camara en el modo de vista aérea.

La camara también es establecida por el método ActualizarPrimeraPersona de las unidad seleccionada, cuando se está en el modo de 1ª persona.

### 6.18.5 Ratón

El api proporciona unas pocas y sencillas funciones para el manejo del raton. Se puede establecer la imagen del raton con el método SetImagenRaton, el cual recibe el handler a una imagen que será pintada continaumente en las coordenadas que ocupe el ratón.

Esto no crea una copia de la imagen, y mientras el ratón permanezca enlazado a dicha imagen, liberar la imagen provocará un fallo no controlado.

También se dispone del método MostrarRaton y OcultarRaton, para poder decidir cuando mostrar u ocultar el raton.

El ratón es pintado al final de todo (después del resto de entidades 2D), por lo que siempre es pintado por encima de todo elemento.

El ratón es establecido en la inicialización de la clase Interfaz::Menu.

En breve se hará que el ratón sea desactivado cuando se entre en 1ª persona, y vuelto a activar cuando se salga de la 1ª persona.

### 6.18.6 Iluminación

Se proveen un par de métodos para el cambio de la iluminación de la escena. SetIluminaciónDifusa y SetIluminaciónAmbiente. Ambos metodos afectan a la componente difusa y ambiental de la luz 0. La luz ambiente global no es afectada por este método.

En estos métodos se especifica, además de los cuatro valores de iluminación deseados, el número de frames que se desea que tarde el efecto. Entonces la api realizara una interpolación de los valores de iluminación desde los valores actuales a los nuevos, tardando el número de frames especificados.

Esto es utilizado por el cambio dia / noche. Durante el día la iluminación ambiental es {0.3, 0.3, 0.3, 1.0} y la difusa {0.8, 0.8, 0.8, 1.0}, mientras que por la noche la iluminación ambiental es {0.0, 0.0, 0.0, 1.0} y la difusa es {0.3, 0.3, 0.7, 1.0}.

## *7. Interfaz*

Dentro de este espacio de nombres se encuentran las clases que gestionan la interacción del usuario con el juego, por lo tanto es la encargada de manejar los periféricos para comunicar las acciones del usuario al módulo gráfico y al módulo de la lógica del juego.

Dentro de este conjunto de clases podemos distinguir: Interfaz de menús, interfaz para las unidades, e interfaz para el juego. Además de una serie de clases adicionales de soporte.

La clase dominante del modulo es la clase Api (Interfaz::Api). Todas las notificaciones de la lógica (directas y provenientes de la Red) serán llamadas a métodos de la clase Api.

## *7.1 Interfaz de menús*

### 7.1.1 Menu

Esta es la clase base para todos los menús del juego. El constructor recibe un puntero al menú padre y cinco strings con el texto que debe poner en cada uno de los cinco botones de navegacióndel menú.

Si se pasa la cadena vacía "" como nombre de un boton, dicho botón no se muestra. Los botones son botones del módulo de graficos. Se utiliza el evento OnClick para controlar los clicks sobre los botones.

La clase incorpora su lógica en el método Actualizar, este método se encarga de comprobar los bools de eventos de clicks, y llamar a uno de los metodos ClickBoton1() ... ClickBoton5() según corresponda. Estos métodos pueden ser redefinidos por clases derivadas para añadir la funcionalidad de cada botón de forma sencilla.

Es importante en el caso de redefinir alguno de los métodos ClickBotoni() que en el método se llame al método de la clase base, puesto que en el se hacen ciertas operaciones necesarias para un funcionamiento coherente.

También se puede redefinir el método Actualizar en caso de ser necesario alguna funcionalidad extra. De nueve es necesario que en dicho método redefinido se llame al método Actualizar de la clase base puesto que si no dejaria de funcionar la lógica de base del menú (a menos que eso sea lo que se pretenda).

Cuando un menu deja de verse (Generalmente porque se ha abierto algún submenu, o menú posterior) se llama al método Desactivar, que desactiva todos los elementos de pantalla del menú. En caso de que en la clase derivada se hayan creado nuevos elementos, se deberá redefinir el método Desactivar para poder desactivar dichos elementos. Después por supuesto habrá que llamar al método Desactivar de la clase base para que desactive los elementos comunes. Exactamente lo mismo ocurre con el método Activar.

Cuando se pulsa uno de los botones, es muy probable que sea para abrir un nuevo menu. En ese caso el menú actual no se cierra, sino que se desactiva (Se deja de visualizar), se crea el menu hijo (pasandole el puntero al menu actual) y se indica a la Api que el menú actual ha cambiado. Se debe avisar en todo momento a la Api de los cambios del menú actual, puesto que es la api la que llama al método Actualizar del menú actual.

Otra posibilidad cuando se pulsa uno de los botones es que sea para cerrar el menú actual y volver al menú padre. En este caso lo que se hace no es destruir ni desactivar el menu actual. Simplemente se llama al método Continuar() del menu padre (Recordemos que todo menu recibe en el constructor un puntero al menú padre que lo ha creado). Es el menú padre el que en el método continuar desactiva el menú hijo, lo destruye, se activa a si mismo, y notifica a la Api que el menú actual es ahora él.

Puesto que cuando un menu llama al método Continuar del menú padre, va a ser destruido antes de regresar de dicho método, es importante que justo tras esa llamada haya un return.

#### 7.1.2. Menu principal : Menu

Este menú es mostrado nada más entrar al juego, y al finalizar cualquier partida. Las opciones que tiene son "Crear partida", "Unirse a partida", "Configuración", "Creditos", "Salir". No define ningún elemento adicional a los soportados por la clase base Menu, por lo que no necesita redefinir Actualizar, Activar ni Desactivar.

Sin embargo las dos primeras opciones tiene un poco de logica asociada. Cuando se crea una partida, o se va a unir a una partida, lo primero que sale es el menu para introducir el nombre del jugador. Una vez que el jugador ha introducido nombre, se regresa al menu principal, el cual en lugar de volver a activarse a si mismo como si de un regreso normal fuese, lo que hace es abrir el menú de selección de mapa o de introducir ip según inicialmente se haya pulsado en Crear partida o Unirse a partida.

El boton Configuración abre el menú de configuración. El botón de Salir finaliza la ejecución del juego.

El botón de Creditos no hace nada de momento, dentro de poco se creará el menú de creditos donde mostrar los participantes del proyecto.

#### 7.1.3. MenuConfiguración : Menu

Se trata de un menú muy simple en el que únicamente se muestran las opciones "Video", "Audio", "Juego" y "Volver". De estas únicamente funcionan Video, que abre el menú de configuración de video, y volver que cierra el menú regresando al menú anterior.

Las opciones Audio y Juego serán añadidas conforme vayan haciendo falta para parametrizar aspectos del juego.

#### 7.1.4. MenuConfiguraciónVideo : Menu

Este menú permite configurar ciertos aspectos de la apariencia del juego. Muestra unicamente la opción "Volver", pero en el constructor de la clase derivada se crean tres botones para cambiar la configuración de multitextura, dia/noche, y estaciones climáticas.

Estos botones utilizan el método SetFijo para quedarse marcados si hay alguna opcion seleccionada. Se controlan los enventos OnClick en el metodo Actualizar de la clase derivada, y en caso de producirse, se modifica el estado de la propiedad en en la configuración del juego, y se cambia el estado fijo del boton al nuevo estado que refleja la configuración.

#### 7.1.5. MenuNombre : Menu

Este menú define los botones "Aceptar" y "Volver", en el constructor define un nuevo elemento, un textbox para poder introducir el nombre del jugador. Inicialmente carga ese textbox con el nombre del jugador que refleja la configuración (por lo que recuerda el ultimo nombre puesto mientras no se salga del juego).

No necesita redefinir el método Actualizar puesto que la accion de logica sobre el nombre la lleva a cabo en el metodo ClickBoton4 (correspondiente a "Aceptar") donde guarda el string del textbox como nombre de jugador en la configuracion y cierra el menú (llama a menuPadre->Continuar() ).

Si en lugar de eso se pulsa el boton volver, se cierra el menú sin actualizar la configuración del juego con el nuevo nombre.

El menú padre (El menú principal) necesita saber si el jugador ha pulsado en aceptar o en volver, ya que si se ha pulsado en volver no se deberá seguir mostrando el menú de seleccion de mapa o de introducción de IP, por lo cual, el MenuNombre tiene un metodo Aceptado() que devuelve true unicamente si se ha pulsado el boton Aceptar.

#### 7.1.6. MenuMapa : Menu

Este menú presenta algo más de complejidad. Muestra los botones "Partida local", "Partida multijugador" y "Volver". Obviamente el resultado de pulsar el botón volver será cerrar este menú y volver al menu principal. Si se pulsa el botón de partida local creara una nueva partida local, y si se pulsa el botón de partida en red creará una nueva partida en red.

Además el menú mapa crea algunos elementos propios. En primer lugar lee los mapas disponibles y rellena un vector de elementos InfoMapa * , despues muestra en un ListBox del módulo de graficos los nombres de dichos mapas.

En el método actualizar de la clase derivada se comprueba el evento OnSelectedItemChanged, y cuando se produce, se cambia la imagen de preview del mapa que se estaba mostrando.

Cuando se crea una partida (con cualquiera de las dos formas) se pasa a la api la ruta al mapa seleccionado.

Es tarea de los métodos ClickBoton3 y ClickBoton4 (partida local y partida en red) el inicializar la logica Logica::Api::Inicializar(), y el inicializar la red: Red::Api::Inicializar ( Red::Api::MODO_LOCAL ) y Red::Api::Inicializar( Red::Api::MODO_SERVIDOR ) según se este creando una partida local o en red.

Tanto la lógica como la red son finalizados ( Finalizar() ) en el método continuar de este menú.

#### 7.1.7 MenuUnirse : Menu

Este menú presenta los botones "Unirse" y "Volver". El boton volver cierra el menú y regresa al menú padre (presumiblemente el menú principal). El boton unirse guarda la ip en la configuración e inicializa la red en el modo cliente. Esto intentará realizar una conexión con la ip especificada, y en caso de conseguirlo cargará el menú de configuración de partida.

Si se produce algún error de conexión finalizará la ejecución del juego, mostrando un mensaje de cual ha sido el error de conexión.

En breve se incluirá tratamiento de errores de conexión dentro del juego, es decir, mostrando el error dentro del juego y regresando al menu principal, o quedandose en el menú de unirse a partida.

#### 7.1.8. MenuPartida : Menu

Este es el menú con más logica asociada del juego. Se encarga de permitir a los jugadores elegir un color y chatear antes de empezar la partida.

Los botones comunes que se muestran son "Empezar partida" y "Volver".

Se muestra además el nombre de cada jugador que haya conectado a la partida. Estos nombres son añadidos en el método NuevoJugador, que será llamado por la Api cuando la llegue una notificación de la Red de que se ha unido un nuevo jugador. (Debe considerarse que las notificaciones de la Logica llegarán siempre a traves de la Red, y es la Red la que se inicializa en modo de red o local)

Se muestra el color actual de cada jugador, para cada jugador creado se crea un rectangulo para mostrar su color. El color de un jugador es cambiado en el metodo Actualizar, comprobando el color del jugador n-esimo, donde n es la posicion que ocupa el jugador.

Se muestra un boton debajo del rectangulo de color del jugador propio para poder cambiar el color. La posicion del boton es actualizada en el metodo Actualizar comprobando el indice del jugador propio con una llamada a Jugador::GetJugadorPropio().

Cuando un jugador abandona la partida, la notificación llegara a la Interfaz::Api que llamará al método AbandonoJugador de este menú, y que hará que se destruyan el sprite del nombre, y el rectangulo de color.

Si llega un mensaje de chat a la Interfaz::Api, el mensaje será pasado al método MensajeChat de este menú para se visualizado en el ChatBox.

Actualmente a los clientes de una partida en red también les sale el botón Empezar partida, si lo pulsan se iniciará su lógica local, provocando un malfuncionamiento general. Habría que hacer que no se mosrase dicho botón a los clientes.

### *7.2 Interfaz de las unidades*

Para representar la interfaz de las unidades se ha diseñado una jerarquía de clases de unidades en las que las unidades son agrupadas según sea parecido su comportamiento en la interfaz (unidades que comparten botones, características de selección, etc).

La clase base es Unidad, el siguiente nievel de especialización es UnidadSeleccionable, que serán aquellas uniddades que puedan ser seleccionadas. Después se dividen en UnidadUniSeleccionable (por ejemplo solo se puede tener un edificio seleccionado al a vez) o UnidadMultiSeleccionable (el resto de edificios).

Dentro de las unidades MultiSeleccionables encontramos otra gran grupo que es el de UnidadAgresiva y que engloba a todas las unidades que comparten la posibilidad de atacar a unidades enemigas.

Algo que abunda en el codigo de estas clases son las llamadas descendentes en métodos estaticos, por ejemplo, si se llama a Unidad::Inicializar, este se encarga de llamar al método Inicializar de todas las clases derivadas de él. Esta solución no me parece del todo apropiada, sobre todo del abuso que se llega a hacer de ella, y me parece conveniente pensar si se podría haber encontrado otra solución.

### 7.2.1 Unidad

Esta es la clase base para las unidades en la interfaz.En primer lugar tiene los métodos estáticos Inicializar, Cerrar y Reset. El método Inicializar se llama al empezar el juego, y el Reset al acabar una partida. Al método Cerrar se llamará al finalizar la ejecución del juego, aunque aún no se hace. Estos métodos estaticos propagan el flujo hacia clases derivadas.

Se puede obtener y establecer el valor del handler de unidad, es decir, su posición en el vector de unidades, además de ser el valor que debe utilizarse para las notificaciones a la lógica.

También tiene métodos para cambiar y obtener su posición y angulo.

Se define el método abstracto int GetTipo(), este método será definido en las hojas para devolver el tipo de unidad de que se trata. Los tipos de unidad se definen en Red::Protocolo y son los mismos para todos lo módulos del juego.

Proporciona un método que devuelve el número de jugador al que pertenece la unidad. Es interesante observar que se define como un método que siempre devuelve -1. Esto es así porque todas las unidades que pertenezcan a algún jugador serán seleccionables. Las unidades no seleccionables serán en su totalidad elementos de "nadie" o de gaia, que tiene el código -1.

Tiene un método SetPorcentajeConstruido, que aunque no tendrá efecto a nivel de unidad, si lo tendrá en UnidadUniSeleccionable, el motivo de ponerlo aquí es debido a que la Api guarda un vector de punteros a Unidad, y si recibe un comando de la logica de actualizar un porcentaje de construcción, tendría que comprobar que el handler de unidad recibido se corresponde con una unidad uni seleccionable. Resulta más sencillo definir este método en la clase unidad que no haga nada, y redefinirlo para las unidades uni seleccionables.

Por ultimo, el método estatico ActualizarUnidades se encarga de actualizar las undiades que esten seleccionadas, por lo que la llamada será propagada únicamente si hay alguna unidad seleccionada. Y el método estatico DestruirUnidades es llamado cuando finaliza la partida desde el método Reset, para destruir todas las unidades que se hubiesen creado.

#### 7.2.2. UnidadSeleccionable : Unidad

Esta clase sirve de base para todas las unidades seleccionables del juego. Tiene los métodos Inicializar, Cerrar y Reset que se han comentado para la clase Unidad.

Además tiene el método ActualizarUnidadesSeleccionables, que se encarga de actulizar la logica de las posibles unidades que haya seleccionadas. Este metodo recibe el flujo de ActulizarUnidades de la clase Unidad.

Tiene un método para buscar una unidad por su handler de objeto. Este método es utilizado al hacer una seleccion opengl, y obtener los nombres de objetos seleccionables que hay bajo el raton, entonces se pasan dichos nombres (equivalentes a handlers de objetos) a este método para que devuelva punteros a unidades.

La logica de seleccion de undiades sin embargo no se encuentra aquí sino en las clases derivadas UndiadUniSeleccionable y UnidadMultiSeleccionable, que son las que controlan dicha lógica de forma distinta segun sea el caso. Por lo tanto, en ActualizarUnidadesSeleccionables se llamara o bien a ActualizarUnidadesUniSeleccionables o a ActualizarUnidadesMultiSeleccionables, según sea la selección actual, para lo cual en esta clase se guarda el estado sobre si la selección actual consta de unidades uni seleccionables o multi seleccionables, para propagar el flujo en la dirección correcta.

Hay veces en que el comportamiento de la interfaz (que ocurre cuando se clicka a algun elemento) cambia del comportamiento actual debido a que se está a mitad de hacer una opcion. Por ejemplo, se ha pulsado el boton atacar, por lo que el siguiente click izquierdo sobre el mapa significará que se desea atacar dicha zona, y no que se desea seleccionar alguna unidad que haya ahí. Es en esta clase donde se guarda cual es la accion actual que se esta desarrollando para las unidades seleccionadas. Puede tener el valor AccionNinguna que significa que la interfaz se comporta de forma normal.

Hay un par de métodos que dan soporte a las hot keys generales (H y . (punto)), estas són BuscarCuartelGeneral y BuscarIngeniero. Ambas buscan el siguiente cuartel general o ingeniero del jugador y lo seleccionan. Para ello se debe guardar donde se quedó la ultima búsqueda y empezar por esa posición en la nueva búsqueda ya que si no siempre se encontraría el mismo elemento.

Los objetos UnidadSeleccionable tiene un conjunto de métodos para el control de la primera persona, estos son EntrarPrimeraPersona y SalirPrimeraPersona, que se encargan de preparar o limpiar los cambios necesarios. Y ActualizarPrimeraPersona, que se encarga del control de la interfaz de 1ª persona. Es en ese método donde por ejemplo se actualiza la camara cuando se está en primera persona.

#### 7.2.3. UnidadUniSeleccionable : UnidadSeleccionable

Esta clase es base para todas las undiades que son uniseleccionables (principalmente edificios).

Tiene los métodos Inicializar, Cerrar y Reset que se han comentado en la clase Undiad.

También tiene el método ActualizarUnidadesUniSeleccionables que comprueba si hay alguna unidad seleccionada y llama a su método Actualizar().

Tiene un método estático SeleccionarUnidad con el cual podemos seleccionar una unidad uni seleccionable. Este método llamara al método Seleccionar de la undiad recibida.

El método DeseleccionarTodo hace que se deseleccione la unidad seleccionada actual si es que había alguna.

#### 7.2.4. UnidadMultiSeleccionable : UnidadSeleccionable

Esta es la clase base de las unidades multiseleccionables, y presenta algunas semejanzas con la clase UnidadUniSeleccionable, pero otras muchas diferencias.

Por supuesto tiene los métodos Inicializar, Cerrar, Reset que se comentan en la clase Unidad.

Ahora no se dispone unicamente de un método SeleccionarUnidad, sino que se tiene DeseleccionarTodo, AnyadirUnidad y QuitarUnidad.

Además la selección actual se puede asignar a un grupo (con control+[0-9]) y seleccionarlo en cualquier momento pulsando el número del grupo.

Los botones y demás logica de interfaz de las unidades multi seleccionables no es dirigida por los objetos seleccionados, ya que es posible que se tengan dos unidades (cual de las dos debería dirigir la interfaz) no es una solución valida. Lo que se hace es que se controla en la propia clase UnidadMultiSeleccionable, en el metodo ActualizarUnidadesMultiSeleccionables. Por el hecho de haber unidades multiseleccionables seleccionadas ya se muestran unos botones.

Después se comprueba si todas las unidades son agresivas, en cuyo caso se llama a UnidadAgresiva::ActualizarUnidadesAgresivas para que pinte los botones característicos de dichas unidades. Despues (en ActualizarUnidadesMultiSeleccionables) se comprueba si además todas las unidades son del mismo tipo. Si es así se llama al método Actualizar de la primera unidad seleccionada, pero no se ejecutará ahí el control de la interfaz, sino que todas las unidades multiseleccionables tienen en su método Actualizar una llamada a un método estatico ActualizarIngeniero, ActualizarSoldado, etc, de modo que así desde la lógica se puede dirigir el flujo a cualquier tipo de unidad de formas sencilla, y una vez allí se envia a un método estatico que permita un control optimo y libre de errores.

#### 7.2.5. UnidadAgresiva

Esta clase sirve de base a aquellas unidades que pueden atacar. Tiene los típicos métodos Inicializar, Cerrar y Reset.

También tiene el método ActualizarUnidadesAgresivas que es llamado cuando todas las unidades seleccionadas son agresivas.

Cada unidad tiene información de estado, donde guarda cual es su modo actual (agresivo, defensivo, mantener terreno o pasivo) para así poder mostrar seleccionado el botón correspondiente cuando sea seleccionado.

### *7.3 Panel*

Son las clases encargadas de gestionar el panel con el que interaccionamos con las unidades del juego y el mapa. Además permite el acceso rápido a acciones usando métodos abreviados del teclado. Se compone de las siguientes clases:

1. ***InterfazPanel*** (*iInterfazPanel.cpp* y *iInterfazPanel.h*).

Crea el panel mediante composición de las siguientes clases proporcionando a su vez un interfaz para interaccionar con ellas.

2. ***IntMinimapa*** (*iIntMinimapa.cpp* y *iIntMinimapa.h*).

Implementa un minimapa que permite un movimiento más rápido por el mapa haciendo click izquierdo sobre él. También permite que una unidad se desplace a un punto marcado en el minimapa mediante click derecho. sobre él podemos ver representadas todas las unidades de la partida mediante cuadros del color del equipo seleccionado tanto del jugador actual como de su contrincante.

3. ***IntSeleccion*** (*iIntSeleccion.cpp* y *iIntSeleccion.h*).

Implementa el interfaz encargado de mostrar la selección múltiple. Al seleccionar varias unidades se mostrará una lista de éstas en el panel pudiéndolas seleccionar desde éste mismo.

Es posible la seleccion de una unidad del conjunto seleccionado simplemente pulsando sobre su imagen de *preview* en el panel. Si esa pulsacion la hacemos teniendo presionada la tecla *Shift* lo que haremos es eliminarla del conjunto seleccionado.

Modos de seleccion:

1. Selección Normal: Mediante un recuadro de selección.
2. Selección Acumulativa: Presionando la tecla *Shift* y seleccionado nuevas unidades éstas se agregan a la selección anterior.

También se permite asignar un número a cada grupo de selección mediante el cual podemos realizar la seleccion rápida de ese grupo sin más que pulsar el número asignado mediante Ctrl + num.

4. ***IntUnidad*** (*iIntUnidad.cpp* y *iIntUnidad.h*).

Esta clase se encarga de mostrar la unidad seleccionada mediante selección simple. Cuando se realice una selección múltiple también aparecerá una unidad seleccionada por defecto que puede ser la primera de la selección.

5. ***IntAcciones*** (*iIntAcciones.cpp* y *iIntAcciones.h*).

Gestiona las acciones disponibles para la unidad seleccionada mostrándolas mediante botones. Cuando se produzca una selección múltiple sólo se mostrarán las acciones comunes del conjunto de unidades seleccionadas. También es posible activar estas acciones mediante atajos de teclado (hotkeys). Al pasar el ratón sobre uno de estos botones de accion aparecera una descripción de la misma a modo de tool tip.

- ***IntBarraSuperior*** (iI*ntBarraSuperior.cpp y iIntBarraSuperior.h)*.

  *La barra superior nos proporciona infomación sobre el estado actual del juego como puede ser cantidad de dinero disponible y número de unidades actuales y máximas. Tambíen dispone de un botón mediante el cual se puede acceder al menú del juego.*

### *7.4. Camara*

La cámara está implementada en la clase *Camara* definida en los archivos *iCamara.cpp* y *iCamara.h*. Esta clase realiza el movimiento de la cámara mediante un movimiento uniformemente acelerado en cada dimensión, hasta la mitad del recorrido, momento en el cual el movimiento es decelerado para alcanzar el punto final suavemente. Por esto, para un movimiento sencillo, la función de posición respecto al tiempo tiene la forma de medio periodo de una función sinusoidal.

Puesto que se puede cambiar la posición deseada antes de alcanzar la posición deseada anterior, en realidad lo que se hace es empezar a desacelerar el movimiento a partir de cuando se calcula que actuando así la cámara va a llegar a la posición deseada, para esto partiendo de la ecuación del movimiento uniformemente acelerado, se tiene que cumplir que la ecuación tenga una o dos soluciones, es decir, el discriminante (argumento de la raíz cuadrada) sea mayor o igual a 0, ya que no tiene sentido físico la solución compleja.

El discriminante es *(vel^2)-2*acel*(posIni-posFinal)*.

La inclinación se trata con el mismo procedimiento.

### *7.5 Configuracion*

La clase *Configuración* está implementada en los archivos *iConfiguracion.cpp* y *iConfiguracion.h* y nos proporciona unos métodos para poder configurar los parámetros de la aplicación como pueden ser el directorio principal, el nombre del jugador o la dirección *ip* del servidor.

### *7.6 Información del mapa*

La clase *iInfoMapa* definida en los archivos *iInfoMapa.cpp* y *iInfoMapa.h* proporciona las herramientas necesarias para obtener la información de un mapa descrito en un archivo *.map*. Estos archivos contienen la siguiente información:

- Nombre del mapa.
- Directorio del mapa.
- Número máximo de jugadores.
- Descripción del mapa.

### *7.7 Interfaz del juego*

La clase *Juego* esta definida en los archivos *iJuego.cpp* y *iJuego.h*. Esta clase hereda de la clase *Menu* para poder volver al menú principal de forma transparente sólo con establecer como menú actual el menú padre desde donde se entró. Esta forma de diseño también nos permitirá en un futuro incluir un menú que sea útil durante el juego.

La clase *Juego* es la encargada de crear la cámara, cargar el terreno, cargar el panel, crear unidades y todos los elementos visualizables en el mapa. También explora los periféricos de entrada para poder interactuar con el juego.

### *7.8 Api*

Proporciona por un lado métodos de inicialización llamado directamente por *main.cpp* para inicializar todos los aspectos visuales del juego. También proporciona unos cuantos métodos de consulta que permiten obtener información sobre el módulo de interfaz a clases de la propia interfaz y de la lógica. Esta clase ofrece también métodos para recibir notificaciones del cambio del estado del mundo.

## 8 Logica

Este módulo se encarga de la actualización del Juego, esto comprende mantener la información referente a cada unidad lógica del juego, las unidades se representan mediante una jerarquía de clases que agrupan sus atributos comunes, para cada unidad puede existir una unidad de Inteligencia Artificial que controle su interacción con el resto de unidades e implemente su comportamiento. Para la interacción de este módulo con el resto del juego así como la interacción entre partes de este módulo se proporciona un API, que se implementa como un interfaz de funciones útiles accesibles estáticamente por todos los módulos y que encapsulan los datos internos de la aplicación.

### 8.1 Api

La API de la lógica esta definida en los ficheros lAPI.h y lAPI.cpp, se define mediante una clase con métodos y datos estáticos ya que no se van a instanciar objetos de la API, define funciones para diversas partes:

- *Abstracción de jugadores:*

Esta parte define un mecanismo para poder enviar comandos a la lógica desde un jugador abstracto, que sirva en un futuro para Humano o IA.

Mediante las funciones EnviarComando, EnviarParametro, FinComando, se enviaran comandos, que serán interpretadas por las unidades como acciones a realizar (El diseño actual del paso de comandos es temporal ya que cambiará cuando se implemente el Protocolo)

- *Inicialización:*

Aquí se definen las funciones necesarias que se llamarán cuando la lógica comience a funcionar, actualmente Inicializar() prepara la lógica y todos los datos para comenzar a crear unidades y EmpezarPartida() inicializará además parámetros de red.

- Gestión unidades:

Se han implementado dos métodos, Crearunidad y Destruirunidad, que se ocupan de instanciar y guardar los objetos en la Lógica, para que esa unidad empiece a pintarse o deje de hacerlo, es necesario notificar al módulo gráfico de ello.

- Interacción con unidades:

Desde dentro de una unidad hace falta comunicarse con el resto de unidades del juego, para ellos se proporcionan unas funciones que permitirán obtener o cambiar datos de éstas.

Será necesario saber la posición o las dimensiones de otras unidades, modificar la vida de los enemigos, y obtener cualquier otra característica de la unidad.

También se proporcionan funciones útiles, como son buscar unidades en un radio o detectar zonas no accesibles en el mapa.

- Actualizar:

Método encargado de recoger los comandos enviados a las unidades (temporal) y de recorrer las unidades para actualizarlas una a una.

### 8.2 Subsistemas

Se han definido unos subsistemas que cada unidad podrá poseer según sus características:

- Máquina de estados:

Define el comportamiento de la unidad, en cada estado se podrán enviar órdenes al sistema motor o mensajes a otras unidades del juego ( "Te he dado" , "Recibes experiencia" ).

- Sistema sensorial:

Se actualizará periódicamente recogiendo información del mundo utilizando las capacidades sensoriales de la unidad a la que pertenece, por lo tanto cada unidad recogerá más o menos

información dependiendo de sus características físicas, un soldado puede ver en un ángulo y a una distancia determinada. Cada sistema sensorial será dueño de una memoria a corto plazo en la que se almacenará esta información.

- Sistema motor:

Este sistema se encarga de todo lo relacionado con el movimiento inteligente de la unidad, se puede definir distintos comportamientos, como perseguir, huir, esconderse, evitar obstáculos, alcanzar destino.

- Memoria a corto plazo:

Esté sistema será básicamente estático, es un almacén de datos recientes de enemigos percibidos en el contexto de la unidad actual, se almacenará la posición, el estado (visible, no visible), el ultimo momento en que se vió y el tiempo en el que se guardo el registro, para poder descartar datos obsoletos tras pasado un periodo corto de tiempo.

- Sistema de Objetivos:

Este sistema utilizará los datos de la memoria para elegir un objetivo, de las unidades que haya se elegirá la que éste dentro del FOV del soldado y de las de dentro la más cercana, si no existen objetivos que cumplan estos requisitos y hay objetivos desaparecidos (que en un instante anterior eran visibles y en el actual no) se utilizará éste como objetivo potencial y dependiendo de las características de la unidad, se actuará de una determinada forma (perseguirlo, apuntarlo, etc..) .

- Sistema de armas:

Básicamente este sistema guarda las armas de cada unidad, y mantiene la seleccionada actualmente, tendrá métodos abstractos para poder disparar independientemente del arma, aunque con las restricciones de la misma, se puede hacer una gestión automática del arma para unidades que lo requieran, y que no tenga el usuario que elegir el arma que quiere utilizar, cada arma se encarga de crear los proyectiles que necesite al actualizar el sistema de armas, se actualizará el estado de las que hayan sido disparadas.

### *8.3. Unidades*

Jerarquía de clases de unidades, definen los parámetros de éstas y se asocian en algunos casos con una IA para comportamientos que no dependen del jugador.

- Unidad:

Clase base de todas las unidades, define propiedades como posición, velocidad, ángulo, bando, raza, equipo, y métodos para acceder y modificar éstas propiedades.

Métodos para asociar y quitar la IA de una unidad, crea una instancia del tipo que se pase por parámetro y la asocia.

Métodos para establecer y obtener las acciones a realizar por la IA, setNuevaAcción, getAccionActual

Método virtual Actualizar que será implementado por las unidades para las que sea necesario.

Método ActualizaIAUnidad que encapsula el método Actualizar de la IA.

- UnidadSoldado:

Clase que deriva de unidad y que representa a un soldado, define nuevas propiedades características de este objeto como la experiencia.

El constructor de unidades soldado la asigna a una IA mediante el método de la clase base asociaIA y establece su posición inicial y su velocidad.

Método actualizar, llama a intervalos regulares a ActualizarIAUnidad de la clase base para que actualice la IA y basándose en los resultados devueltos de velocidad y ángulo, calcula una nueva posición teniendo en cuenta características físicas de la unidad, por ejemplo, si la unidad tiene poca vida la velocidad devuelta por la IA podrá reducirse.

- Unidad Recurso:

Unidad sin IA que almacena un valor, y representa una fuente de algún recurso del juego que será explotado a lo largo del transcurso del mismo.

### *8.4. IA's de unidad*

Parte encargada de controlar los comportamientos de las unidades, las clases que se asocien a alguna unidad implementarán un autómata de estados finitos en el método principal Actualizar, que será el que realice transiciones de estados, y en definitiva el que modifique el comportamiento de la unidad.

- IAUnidad:

Es la clase base de las IA's de unidad. Define parámetros básicos necesarios por toda la jerarquía, como son, estado, acción, velocidad, ángulo, o unidad, que será la referencia a la unidad que controlan.

Método setNuevaAcción, será el encargado de cambiar la acción actual.

Cada IA define e implementa unas acciones concretas.

- IAUnidadMóvil:

Es una IA que está pensada para servir a IA's derivadas, implementando para ello acciones primitivas que tengan que ver con el movimiento como PasoAndar, PasoCorrer, PasoParar , que avanzan un paso el movimiento de la unidad, también se implementan eventos de movimiento, como eventoHaLlegado o eventoColision, que serán utilizados por los autómatas de las IA's derivadas para cambiar de estado.

Método setNuevaAcción, se redefine en esta clase para que tenga en cuenta las acciones ANDAR,CORRER y PARAR.

Métodos setParamsMovil, se utilizan para pasar los parámetros de las acciones con movimiento a las IA's (temporal)

- IAUnidadOfensiva:

Clase que implementa las funciones auxiliares, acciones primitivas y eventos necesarios para el combate.

Métodos auxiliares:

AuxBuscarMejorEnemigo: Intenta buscar el enemigo más cercano y más alineado con la unidad actual (que requiera menos giro) en un radio determinado (zona audible + zona visible), si lo encuentra devuelve el identificador de la unidad, sino -1

AuxHayObstaculos: Comprueba si entre la unidad actual y entre la que se pasa como parámetro hay obstáculos de mapa o de unidades grandes, como edificios.

AuxHayEdificios: Busca edificios en la línea que se pasa como parámetro.

AuxEsVisiblePorUnidad: Comprueba si no hay obstáculos (utilizando funciones anteriores) y si además la distancia y orientación de la unidad permite visualizarla.

Hay eventos que se deberán gestionar como,  visibilidad de enemigos, cercanía, saber si ha muerto, si la unidad está lejos de la base o si tiene poca vida (No están todos implementados).

Las acciones primitivas definidas en la IA ofensiva, son PasoDisparar que dispara un arma de fuego y calcula un daño al enemigo y PasoAtacar, que calcula un daño cuerpo a cuerpo.

El método Actualizar implementa un autómata.

La jerarquía de las IA's aún está en fase de diseño y es posible que cambie en un futuro, se ha pensado en utilizar herencia múltiple, para por ejemplo definir IA's de unidades que tengan movimiento y sean ofensivas (soldados), y poder distinguir entre otras que sólo tienen movimiento (ingeniero) o que sólo son ofensivas (torretas de vigilancia)

## *9. Util*

En esta clase hemos métido aquellas clases que eran utiles para todas la partes del juego. El tipo de clases que la conforman se pueden dividir en dos grupos: destinadas al control del tiempo, y tipos de datos (vectores).

- FrecEv:

Esta clase se utiliza para medir frecuencias de ejecución, se define pasando al constructor un porcentaje de ejecución (de 0 a 1) y se llama al método comprobar() que sólo devolverá true ese porcentaje de veces, se utiliza para controlar las ejecuciones de algunos eventos de IA que pueden ser muy costosos, como búsqueda de caminos o de unidades

- FrecReal:

Se basa en la misma idea que la anterior, pero esta vez lo que se mide son frecuencias en tiempo real, es decir, se define pasando un número de veces por segundo, y cuando se llame al método comprobar() éste sólo devolverá true a intervalos regulares. Se utilizará por ejemplo para controlar e independizar las animaciones y la velocidad del juego de la CPU en la que se ejecuta.

- Vector2D:

Clase que define e implementa un vector en dos dimensiones y las operaciones básicas sobre él, como sumas, restas, producto escalar y módulo.

- Vector3D:

Análoga a la versión 2D.

# III. DESARROLLO

## *10. Seguimiento de objetivos*

Leyenda:

Objetivo cumplido.

Objetivo en progreso.

Objetivo retrasado.

Objetivo cambiado.

### *10.1. Hito 1*

Fecha de revisión: 06-04-05.

Porcentaje a realizar: 30% (30% Realizado)

#### 10.1.1 Graficos

1. Carga de recursos (modelos, texturas, bitmaps, etc).
   - La tarea fundamental aquí, y la que más horas llevará es la carga de modelos 3D. La carga del resto de recursos resulta trivial con las funciones que proporciona Allegro. (15 horas)
   - Carga de bitmaps, fuentes y sonidos (1 hora)
2. Creación de elementos de GUI utilizables de forma sencilla por el modulo de interfaz. (10 horas)
3. Instanciación de unidades. (30 min)
4. Carga y visualización del mapa. (2 horas)
5. Control de la cámara. (2 horas)

#### 10.1.2 Interfaz

1. Soporte de desplazamiento por el mapa. (1 hora)
2. Inclinación de la vista. (30 min)
3. Realizar menús (5 horas)
4. Interacción con unidades a traves de botones de la interfaz
   - Interacciones con el terreno: (3 horas)
     Movimiento agresivo
     Mover
   - Interacciones con otras unidades: (4 horas)
     A: Atacar unidad
   - Ordenes inmediatas:
     Detenerse
5. Interacción básica con ratón y teclado.
   - Desplazamiento del mapa con el raton (30 min)
   - Búsqueda de unidades con teclas estableclidas (H -> HeadQuarters) (3 horas)
     H: Cuartel general
     . (punto) : Siguiente unidad
   - Asignación de ordenes con teclas establecidas (A -> Atacar) (5 horas)
     A: Atacar
     D: Detenerse
     M: Mover
   - Selección de unidades (2 horas)
   - Selección multiple de unidades (con la tecla shift) (5 horas)
     Este punto presenta una mayor complejidad puesto que hay que adaptar la

interfaz, de modo que las ordenes que se den, sean dadas a todas las unidades (todas las unidades que puedan realizar dicha orden)

#### 10.1.3 Logica

1. Crear jerarquía básica de clases para las unidades (soldado, edificio, etc). (5 horas)
2. Crear jerarquía básica de clases para la Inteligencia Artificial de las distintas unidades (guerrero, misil, recolector, etc). (5 horas)
3. Implementación de un protocolo básico de comunicación con el módulo de GUI y el Gráfico. (5 horas)
4. Implementar algunas órdenes básicas. (2 horas)
5. Implementar comportamientos básicos (andar, pararse, crear y destruir unidades). (10 horas)

### *10.2 Hito 2*

Fecha de revisión: 19-05-05.

Porcentaje a realizar: 50% (80% Realizado)

#### 10.2.1 Graficos

1. Finalizar el motor gráfico.
   - Carga de modelos con texturas (20 horas)
   - Carga de modelos con compuestos (20 horas)
   - Carga de modelos con partes de color de bando (5 horas)
   - Texturizacion del agua (1 hora)
   - Ligero movimiento del agua (30 min)
2. Soporte de sonido. (3 horas)

#### 10.2.2 Interfaz

- Soporte completo del interfaz.
  - Finalizar botones de ordenes (10 horas)
  - Interfaz inteligente (boton derecho del raton) (10 horas)
  - Retratos de unidades e interaccion con estos (15 horas)
  - Movimiento suave de camara (5 horas)
- Implementación de un mapa reducido en la ventana de comandos para acceder rápidamente a las distintas zonas de éste. (3 horas)

#### 10.2.3 Logica

1. Completar el protocolo de interacción con los módulos GUI y Gráfico. (10 horas)
2. Completar jerarquía de clases para las unidades y para la IA de las unidades. (20 horas)
3. Completar implementación del comportamiento mediante IA. (20 horas)

### *10.3 Entrega*

Fecha de revisión: Última clase práctica.

Porcentaje a realizar: 20% restante

6. Disponibilidad de varios escenarios. (15 horas)
7. Completar el modelado de las distintas unidades. (40 horas)
8. Finalizacion de menús (5 horas)
9. Depurar errores (10-15 horas con un 75% de fiabilidad, 5-25 horas con un 90%)

### *10.4 Optativos*

1. Leer y guardar partidas
2. Posibilidad de entrar en modo de primera persona (mediante un desplazamiento de cámara)
3. Realización de distintos efectos de luces.
4. Efectos especiales (fuego, tormentas, lasers, sistemas de partículas)
5. Variación de las texturas del mapa según el transcurso del juego (suelo quemado tras un combate, etc.)
6. GUI amigable, mejorada con facilidades al usuario.
7. Posibilidad de jugar en red.
8. Posibilidad de que las unidades se muevan por terreno inclinado
9. Animación de unidades
10. Posibilidad de que las unidades disparen en una direccion distinta de la que caminan (que los tanques giren la torreta para disparar)
11. Implementación de IA de grupo sencilla que asigne prioridades a zonas y de ordendes sencillas a las unidades (moverse a dichas zonas).
12. Selección múltiple de unidades mediante un cuadro de selección.
13. Creación de grupos de unidades (Ctrl+[0..9])
14. Unidades aéreas (helicóptero) movimiento igual que el resto de unidades, solo que el modelo se situa con la coordenada Y valiendo una cierta cantidad más que el suelo. (Antes de hacer esto sería conveniente hacer el apartado de unidades animadas, ya que resultaría un poco sorprenderte ver volar un helicóptero con las aspas paradas...)
15. Indicar la salud de las unidades en el terreno de alguna forma (barra de vida, o color de la marca de selección)
16. Indicar la salud de las unidades en el retrato de selección.
17. Mostrar progreso de tarea de edificio en el area de retratos (construcción, investigación, etc) una barra de progreso.
18. Menu dentro de juego (pausa, opciones de video, juego, etc)
19. Menu de opciones de juego, video y/o sonido. (Es decir, posibilidad de configurar ciertos aspectos del juego, como la calidad de las texturas o la velocidad de scroll)
20. Secuencia de muerte de unidades (explosiones para edificios y vehiculos, elevación del alma para personas...) algo en lugar de hacer desaparecer directamente la unidad.
21. Secuencia de construcción de edificios (hacerlo opaco gradualmente, o hacerlo crecer en el eje Y, o truncarlo en el eje Y cada vez a mas altura...) algun efecto en lugar de hacer aparecer al edificio de repente en cuanto llega el constructor. Durante este tiempo el constructor permanecera junto al edificio (construyendolo claro, no admirando el bonito efecto que hemos hecho..., asi que disponer de animaciones para las unidades ayudaría a distinguir este hecho)
22. Búsqueda de camino para las unidades.
23. Cambio del estado de respuesta de la unidad (estado ofensivo, defensivo, mantener terreno, o pasivo) Si no se hace este optativo las unidades tendrán un unido diagrama de actuación que correspondera al del estado ofensivo (perseguir cualquier unidad que vean hasta donde sea)
24. Depurar en mayor medida los errores de modo que incluso sea posible jugar mas de una partida sin salir del juego.
25. Incluir recursos al juego, necesarios para construir cosas.
26. Interfaz del juego en primera persona.
27. Realización y control de modelos complementarios (disparos, misiles, granadas, etc)
28. Optimizacion del opengl usando glInterleavedArray
29. Variación del tono de la escena simulando dia/noche (incluyendo varios tonos según hora del dia.
30. Visualización del area visible en el minimapa.
31. Poder mover unidades a zonas pinchando en el minimapa.

32. Incluir un chat en el juego
33. Poder cambiar de color en la pantalla de configuracion de partida
34. Incluir un chat en la pantalla de configuracion de partida
35. Cambio de estaciones: verano/invierno.
36. Centrar la unidad seleccionada pinchando en su retrato grande.

# 11. Reevaluación de objetivos

## 11.1 Reevaluación de objetivos tras el hito 2

Comparando los hitos planificados para el hito 2 con los alcanzados se observa que se han cumplido la mayoría. Además si se han dejado de cumplir ciertos objetivos ha sido debido a que se ha considerado que aportaba poca funcionalidad, mientras que habían otros objetivos no contemplados que requerían una atención más urgente. Por ejemplo, no tenia sentido finalizar la carga de modelos con texturas, si antes no se solucionaba el problema que había con la iluminación que hacía que todo se viera blanco o negro. Ahora que ya hemos conseguido que se visualize correctamente los modelos según sus valores de material, podemos afrontar la carga de modelos con texturas.

Si bien en cuanto a cumplimiento de objetivos nos sentimos plenamente satisfechos, no lo estamos tanto en cuanto al plazo de cumplimiento del hito 2, es decir, el nivel de desarrollo que se ha tenido en cuenta para compararlo con el hito 2 se toma dos semanas después de la fecha establecida para el segundo hito. ¿Por qué hemos retrasado el hito y evaluamos ahora los motivos que nos han llevado a retrasarlo, en lugar de entregar en la fecha establecida y analizar los motivos que nos hubiesen llevado a no cumplir los objetivos? Bueno, hemos considerado que el problema no estuvo en una mala planificación de las tareas del hito, sino en una mala evaluación del tiempo del que íbamos a disponer para el segundo hito.

Es decir, hemos retrasado la entrega del segundo hito hasta el momento en el que le hemos dedicado la cantidad de tiempo proporcional que inicialmente habíamos estimado, para de este modo poder tener una referencia util de cara a planificar los objetivos del tercer hito (entrega).

Finalmente sólo queda analizar los motivos que nos han llevado a errar en la previsión del tiempo del que ibamos a disponer para el segundo hito. Todos los miembros del grupo coincidimos en que la falta de tiempo disponible ha sido debido a otras asignaturas. La conclusión que se extrae de esto es que la planificación falló al no evaluar correctamente los riesgos externos al proyecto.

Aprendiendo de nuestros errores esto será tenido en cuenta para el tercer hito, si bien el resto de asignaturas ya no supone un riesgo a tener en cuenta, seremos especialemente cuidadosos en tratar de manera urgente el módulo de lógica, debido a que Juan Rubén, el encargado de dicho módulo está esperando ser adminitido en una empresa para trabajar, lo cual de ser así sin lugar a dudas perjudicaría el desarrollo de dicho módulo.

Ya desde una perspectiva global, y centrandonos en el cumplimiento de objetivos según tiempo invertido, podemos afirmar que estamos cumpliendo casi la totalidad de objetivos propuestos. Un único apunte sobre este tema sería el hecho de que si bien se están cumpliendo objetivos, el proyecto esta acumulando inestabilidad, nos planteamos por tanto un nuevo objetivo consistente en reparar todos los fallos que se producen durante el juego. Este objetivo incluye el caso de jugar una partida, y salir del juego sin que se produzca un error, dejando como optativo el lograr que una vez acabada una partida se pueda empezar otra sin que se produzcan errores.

En cuanto a partes optativas que cumplir, existirían dos grandes areas que cubrir, por un lado, incrementar los componentes del juego (mas niveles, mas unidades, etc) ya que de momento solo se cubren 2 edificios, 2 unidades y 2 niveles. Por otro lado, mejorar el gameplay, o las posibilidades de juego. Consideramos que el objetivo debe ser maximizar el resultado final, y que para ello la mejor forma de hacerlo partiendo del estado actual es trabajar ambos apartados, en lugar de centrarse en una de las areas descritas. Por tanto, el tiempo del que dispongamos para desarrollar partes opcionales en este tercer hito será dedicado tanto a mejorar las características de juego, como a añadir componentes al juego.

# 12. Implementación

## 12.1 Carga y renderizado de modelos

Inicialmente tuvimos problemas con los modelos, ya que tardaban mucho en renderizarse. Cargamos los modelos de ficheros .3ds, y utilizando el algorimto de carga y renderizado del ejemplo de la libreria lib3ds, lo que se hacía era generar una callList para cada nodo, y después cuando se van a renderizar, se multiplica la matriz modelview por la matriz de transformación del nodo.

Tras hacer varios perfiles se vio que el hecho de abrir y cerrar (glBegin, glEnd) el render relentizaba notablemente a este. Además, el hecho de separar las callLists en nodos y aplicar la matriz de transformación en tiempo de render puede ser necesario para modelos animados, pero una completa pérdida de tiempo para modelos estaticos.

Por ello se optimizó el algoritmo de cara a modelos estaticos. Durante la creación de las listas, se carga en la matriz modelview las diferentes transformaciones, cada vez que se lee un vertice se obtiene su valor transformado multiplicandolo nosotros mismos por los valores de la matriz modelview. Esto nos permite obtener los valores finales que va a tener cada vertice (valido solo para modelos estaticos claro).

Además, para realizar una única llamada a glBegin/glEnd, lo que se hizo fue utilizar glDrawArrays. Creamos una interleaved element en la que se guarda temporalmente para cada vertice sus coordenadas (ya calculadas), junto con sus coordenadas de mapeo de textura, su color y su normal. Despues la llamada a glDrawArrays es encapsulada en una callList con lo que se genera una copia de los datos del interleaved element y este se puede utilizar para otras cosas, es por esto que interleaved array es estatico.

Además, en nuestra intencion de pintarlo de una vez, vimos que era necesario texturizar los modelos con una única textura, afortunadamente pudimos utilizar la opcion render to texture del 3ds max que casi lo hacía todo solo. Lo único que habría que cambiar las características de la iluminación del render puesto que en la textura hay zonas iluminadas y zons con sombras, cuando este debe ser hecho en tiempo real segun la orientación del objeto.

Considero muy notable la mejora obtenida con estas modificaciones, que nos ha permitido pasar de 15.000 triangulos en pantalla a casi 50.000.

## *12.2 Terreno*

### 12.2.1. Terreno redondeado

Para lograr la forma redondeada del terreno, lo que se ha hecho ha sido calcular la normal de cada vertice, sumando las normales de los 8 triangulos a los que pertence dicho vertice, y normalizando de nuevo, de este modo, y activando el sombreado SMOOTH se obtiene dicho resultado.

### 12.2.2. Calculo de altura de un punto continuo

Para calcular la altura del terreno en un punto x-z determinado, es necesario primero determinar en que quad de la malla cae, después determinar en cual de los dos triangulos del quad cae, para esto se lleva las coordenadas a coordenadas relativas al vertice superior izquierdo del quad, y basta con comprobar que si x < z o no para saber en que quad cae.

Una vez se tiene eso se obtiene la ecuación del triangulo sobre el que esta el punto a partir de la normal del mismo. La normal se obtiene multiplicando (v2-v1)*(v3-v1), siendo v1, v2, v3 los vertices del triangulo, (la multiplicación debe ser vectorial). El vector resultado se normaliza y se tiene la normal del triangulo.

Con dicha normal (A, B, C) se sabe que la ecuación del triangulo es Ax + By + Cz = D, se dan valores a x, y, z los de algun vertice, y así se obtiene el valor de D.

Con todas las constantes determinadas, es posible sustituir x-z en la ecuación y obtener el valor de y.

### 12.2.3 Algoritmo de multitextura

Al generar el terreno se genera un conjunto de listas de opengl para cada region del terreno. Las regiones son generalmente regiones de 10x10 quads, sobre las que se aplicará clipping.

No se genera una única lista por cada region porque esto significaría estar cambiando de textura continuamente para pintar las texturas de cada región. Esto sería muy lento. En lugar de esto se genera una lista por cada region y textura. Ademas esas listas son generadas con glDrawArrays, usando una interleaved element array, en la que hay información de opacidad en el color (se usa glColorMaterial).

De este modo, lo único que se tiene que hacer al pintar es, calcular clipping, guardando en una array que regiones se pintaran. Pintar la primera capa (textura) en la todas las regiones visibles. Pintar la segunda capa, la tercera, etc.

Es necesario para el algoritmo de multitextura (un algoritmo multipass) que la tarjeta lo soporte, gran cantidad de tarjetas nVidia pueden tener problemas con esto, o ser necesario bajar drivers actualizados. Sobre una tarjeta nVidia que hemos podido probar, esto no ha funcionado hasta instalar los drivers de este mismo mes. Con todas las tarjetas ATI que se ha probado ha funcionado sin problema (mas o menos rapido está claro, pero ha funcionado)

Lo que se hace, es asignar a cada vertice de la malla, un nivel de opacidad dependiente de su altura y/o angulo respecto a la horizontal. Por ejemplo, de base se pinta siempre las rocas, y encima se pinta la hierba, peor esta es más transparente cuanto más inclinado el terreno, hasta que desparece. (En realidad no se pinta siempre la capa de rocas, el algoritmo está optimizado para no pintar triangulos que vayan a tener encima una capa 100% opaca).