1. ¿Qué es CrisolEngine?.

CrisoEngine es una aplicación o motor (a partir de ahora se mencionará usando el término *Crisol* y engine indistintamente) que sirve para ejecutar juegos de rol por ordenador. *Crisol* no es un juego, sino un soporte para que un diseñador de juegos pueda llevar a la práctica sus propias ideas de juego. Se entenderá por diseñador de juegos, cualquier persona que quiera realizar un juego con *Crisol*.

Para lograr que *Crisol* sea realidad como aplicación, se han tenido que conseguir tres objetivos fundamentales. El primero es el de dejar que el diseñador pueda definir por completo el contenido y aspecto con el que *Crisol* funcionará. Cuando hablamos de contenido y aspecto, nos referimos al diseño de la interfaz, de los personajes de juego, de los gráficos de los decorados, de la música, del diseño de niveles, etc. Como segundo objetivo, *Crisol* ofrece al diseñador la posibilidad de especificar el comportamiento lógico ante los diversos eventos internos. En otras palabras, utilizando un sencillo lenguaje script embebido, el diseñador podrá dar "inteligencia" al motor, decidiendo qué será lo que pase. El último objetivo de todos, y quizás el más importante, es el de ofrecer una comunicación aplicación – diseñador lo más sencilla posible, pues la idea es que *Crisol* sea una herramienta destinada a cualquier tipo de usuario con unos mínimos conocimientos de informática.

El nombre de *Crisol* proviene, precisamente, de que el motor pretende demostrar que con él se pueden hacer un crisol de juegos diferentes gracias a la posibilidad de modificar el aspecto del mismo y al uso de un lenguaje script sencillo.

1.1. Introducción a los motores de juegos.

Los videojuegos comerciales, popularizados de forma masiva con la llegada de las consolas, tienen todos una arquitectura muy parecida basada en un principio elemental que consiste en separar el juego en tres partes fundamentales: la aplicación que ejecuta el juego, y que vamos a llamar a partir de ahora *motor* o *engine* indistintamente los archivos de datos, que contienen la información gráfica, de sonido y de diseño de niveles, y la parte lógica, que da coherencia a todo y hace que lo que veamos sea un juego [Saltzman'99]. De esta forma, una compañía que se dedique a realizar juegos y ya tenga desarrollada una tecnología base para su motor o aplicación, sólo tendrá que centrarse, para cada nuevo producto, en las partes destinada a los datos y a la información lógica. Carece de sentido que una empresa tenga que reescribir el código de todo el motor si va a sacar, por ejemplo, una expansión o la segunda parte de un juego que utilizó ese mismo motor (debemos de tener en cuenta, además, que si un juego alcanza éxito, el mercado demandará una segunda parte o expansiones en el tiempo más breve posible).

Esta separación descrita se ha hecho imprescindible en nuestros días ya que en la actualidad, realizar un juego con perspectivas internacionales supone estar trabajando por un periodo de 2 a 3 años tanto en el contenido como en tecnología (que será el motor propiamente dicho). Es más, son muchas las empresas que se centran en el desarrollo de juegos pero teniendo como objetivo igual de importante el licenciar el motor, a partir del cual construyen sus productos a otras empresas desarrolladoras. Estas otras compañías, utilizando ese motor que han licenciado, podrán hacer el juego que quieran según las propias posibilidades del engine que hayan licenciado y de las modificaciones o mejoras que introduzcan.

Para lograr crear juegos completamente distintos utilizando motores idénticos, se recurre a los lenguajes scripts (procesadores de lenguajes) o bien a librerías dinámicas, DLL [Huebner'97]. En ambos casos, lo que se pretende es asociar código con el fin de que cuando pase un determinado evento en el juego el diseñador pueda decidir qué será lo que suceda. Por ejemplo, un evento en un juego pudiera ser que el jugador pise una determinada zona de la pantalla o decida manipular un objeto del escenario. En cualquiera de estos casos, el motor generaría un evento (el de pisar, el de manipular, etc) que a su vez tendría asociado un fichero script o bien una porción de código en una DLL (llamada a una función contenida en la DLL). Así, el diseñador podría decidir que si el protagonista pisa una determinada posición éste pierda, se ponga a volar, se transforme, etc. Las posibilidades son infinitas y, lo más

importante de todo, la lógica no está “incrustada” en el código fuente de la aplicación o motor, por lo que el guión de ejecución del juego se podrá cambiar desde el exterior.

En la actualidad, el medio más usado para dar coherencia al motor de juego es el uso de lenguajes script. Sus ventajas más importantes radican en que son lenguajes generalmente muy sencillos de aprender y permiten que los diseñadores del juego (que no tienen porque saber nada de programación) puedan asimilarlos sin dificultad. Son portables de forma directa en cuanto el motor esté disponible para otro sistema operativo distinto (ya que es el motor quien interpreta el lenguaje script, que permanece inalterado, a través de su máquina virtual), no hay que recompilar el código fuente del motor para ver los cambios realizados en los scripts (por así decirlo, los cambios son vistos de manera casi instantánea), cualquier error fatal introducido por el diseñador puede “dominarse” sin que el motor se cuelgue, etc. Por el contrario, las DLL necesitan que el diseñador tenga conocimientos de programación importantes ya que se trabaja en el mismo lenguaje con el que se programó el motor, teniéndo éste acceso a las instrucciones de la aplicación de forma nativa. Estos factores hacen que la depuración sea más complicada y el proceso de creación más lento, pues los errores en el código fuente suministrado por una DLL podrían bloquear más fácilmente el motor. La ventaja más clara es la velocidad y aquí, en el mundo de los juegos, eso es algo muy importante. De todas formas, la tendencia viene apuntando al uso de los lenguajes script pues sus ventajas superan ampliamente a las de las DLL. La mantenibilidad y la reutilización de código ha pasado a ser más importante que la velocidad, pues los equipos tienden a poseer un hardware más potente y el software aumenta complejidad en cada nuevo desarrollo.

Cabe destacar que el uso de un lenguajes script procesados por intérpretes puros ha sido siempre descartado debido a su gran lentitud a la hora de procesarse. Un juego es una aplicación totalmente interactiva, que necesita dar siempre respuestas instantáneas a los deseos del jugador, por lo que la velocidad ha ocupado siempre un apartado fundamental en este tipo de productos. Por todo esto, desarrollar un lenguaje script supone desarrollar un compilador que lo pase a código intermedio y, a su vez, una máquina virtual embebida en el motor que se encargue de procesar ese código intermedio generado. Las últimas tendencias van dirigidas a la utilización de lenguajes script hechos por terceros que ahorren tiempo de desarrollo a la empresa (crear un lenguaje script, por sencillo que éste sea, es una tarea muy laboriosa). Algunos de estos lenguajes son el *Lua* que es un lenguaje script muy pequeño y sencillo y está siendo usado en un número cada vez mayor de juegos comerciales (http://www.lua.org/), el *Java* [Huebner’99], usando *Java Native Interface*, también está siendo muy utilizado (http://java.sun.com/docs/books/tutorial/native1.1/concepts/index.html) o el también muy popular *Phyton* (http://www.python.org/). Algunos motores de juegos muy conocidos han desarrollado auténticas maravillas de lenguajes script como el caso del motor que da soporte al juego *Unreal* (http://unreal.epicgames.com) el cual cuenta con un lenguaje script llamado *UnrealScript* lleno de aspectos interesantes (http://unreal.epicgames.com/UnrealScript.htm).

Como lenguaje de programación utilizado para la construcción de los motores de juegos, el C++ es el rey indiscutible. Aún en nuestros días se utiliza en cierta medida el C pero lo normal viene siendo, debido a la alta complejidad de los proyectos actuales, acudir al C++ [Rollings’00].

En la actualidad, el desarrollo de juegos se ha convertido en una de las partes más apasionantes e importantes en el desarrollo de software. Ha pasado de convertirse en algo creado por una sola persona en su casa a ser un sector que mueve miles de millones de doláres todos los años y que necesita de gente muy cualificada para poder llevar a cabo una producción. Los equipos encargados de desarrollar un juego están formados por un mínimo de 10 a 15 personas mientras que los destinados a copar las listas de ventas a nivel internacional, cuentan con equipos de entre 15 y 50 (o incluso más). Debemos de tener en cuenta que la creación de un juego reúne una gran cantidad de campos artísticos y técnicos. Desde la programación y diseño de arquitecturas de software y utilidades de apoyo, al diseño gráfico (dibujo), de niveles, música, guión, testeo y un largo etcétera [Rollings’00].

Así pues, los motores de juegos son aplicaciones carentes de inteligencia, que necesitan de un código de apoyo que les diga cómo procesar los datos y responder a los eventos que el jugador desencadene. En cierta forma, es como la metáfora del ordenador. Un ordenador en sí no es más que un conjunto de piezas de hardware que no pueden servir al usuario por si solas

y que necesitan de un programa, en este caso un sistema operativo, que los gestione y los ofrezca al usuario mediante una interfaz adecuada. Con un motor de juegos pasa lo mismo, el motor no sabe hacer juegos por si sólo, necesita de un código creado por el diseñador que sirva para dar sentido a lo que el motor mantiene (los archivos de datos) y a lo que en él sucede (los eventos desencadenados por el usuario).

1.2. Descripción general de CrisolEngine como motor de juegos.

CrisolEngine sigue los principios generales comentados sobre los motores de juegos. *Crisol* está orientado a la creación de juegos de rol, por lo que una parte importante de su funcionalidad será la de definir aspectos concretos de este tipo de juegos, desde las reglas hasta la representación del escenario (naturalmente, *Crisol* se basa en ciertos juegos comerciales que han creado un determinado estilo muy popular, en concreto, en los que cuentan con un sistema de representación isometrica, que permite, utilizando un sistema de representación 2D, añadir un aspecto tridimensional al universo de juego visualizable).

Uno de los aspectos más importantes que se intentó alcanzar, fue el de ofrecer suficiente libertad al diseñador como para que éste no se tuviera que ceñir a aspectos predefinido por el programador. Esta premisa se llevo tanto al campo visual como al campo lógico, es decir, el motor permite que todas las interfaces, cursores, botones, textos, etc, puedan ser modificados por el diseñador con el fin de que éste pueda crear un estilo único en sus juegos (no es lo mismo crear un juego de rol basado en el futuro que en la época medieval o en cualquier otra situación). Por otro lado, utiliza un sistema de reglas que permite a diferentes diseñadores crear entidades y sistemas de juegos completamente diferentes, siempre acordes a lo que realmente quieran desarrollar. Naturalmente, toda esta libertad esta basada en ciertas normas pues sería completamente imposible crear algo que diera libertad absoluta.

Para permitir que el diseñador pueda adaptar el aspecto del motor a sus necesidades, así como diseñar los niveles o áreas, *Crisol* utilizará un lenguaje sentencial destinado únicamente a definir configuraciones y contenido (básicamente, se asociaron valores a variables internas de configuración). El lenguaje script verdaderamente importante será el utilizado para vincular sucesos a los distintos eventos que se producirán en el universo de juego. Para dar soporte a dicho lenguaje, la aplicación poseerá una máquina virtual en su interior de tal forma que procesará código intermedio previamente creado a partir de un compilador que, a su vez, compilo un fichero script. Este lenguaje script se denominará *CrisolScript*.

Para dar soporte al motor existirán diversas herramientas de apoyo. Estas herramientas serán en línea (no gozarán de una interfaz visual) y estarán destinadas a compilar el código script (herramienta *CSCompiler*), chequear los archivos de datos pasando algunos de ellos a formato binario (*CrisolBuilder*) y empaquetar todos los archivos de datos (menos los de sonido) en un único fichero para facilitar la distribución de los juegos y no permitir la modificación o borrado de los datos (*CPAKTool*).

2. Tecnologías empleadas.

Para la construcción de *Crisol* se han utilizado diversas tecnologías. A continuación se enumerarán cada una de ellas y se comentará qué responsabilidad han jugado en la construcción del engine.

2.1. Uso del lenguaje de programación C++ y entorno VisualStudio.

Toda la aplicación se halla construida utilizando el lenguaje de programación C++ [Stroustrup'97]. El motor está construido para el sistema operativo Windows utilizando la metodología orientada a objetos [Boer'00a] pero sin utilizar la biblioteca de clases MFC (Microsoft Fundation Classes). El uso del lenguaje de programación C++ ha sido completamente necesario debido a que es el mejor en la creación de motores de juegos pues cuenta con un amplísimo soporte de documentación tanto en libros impresos como en Internet y su velocidad de ejecución es superior al de otros lenguajes como el *Java* o *Delphi*.

El entorno de desarrollo utilizado ha sido el VisualStudio, en concreto, la versión 6 del mismo. Este entorno posee un compilador de C++ para Win32 que es el que mejores ejecutables genera para Windows. Durante el desarrollo de *Crisol* se ha utilizado el citado compilador y la interfaz visual disponible en VisualStudio para facilitar la programación y la gestión de los proyectos en C++.

2.2. Uso de la biblioteca estándar STL de C++ (Standard Template Library).

El uso de esta biblioteca se tornó imprescindible. En un principio se pensó (e incluso se llevo a la práctica durante un periodo de tiempo) en la programación de clases que manejaran estructuras de datos (listas, árboles, tablas hash, etc). Más tarde se comprendió que el tiempo invertido en la creación y la depuración de estas clases iba a ser lo suficientemente largo como para desviarnos del verdadero objetivo, que era la construcción de un motor de juegos [Rollings'00]. Además, las clases que dieran soporte a esas estructuras de datos jamás iban a poder competir con lo que las STL ofrecían, tanto en rapidez como en prestaciones [Meyers'96] y [Meyers'97].

Así pues, al comienzo del desarrollo de *Crisol* se pasó a utilizar una porción importante de las ventajas que ofrecen las STL [Boer'00b]. Para empezar, todas las estructuras de datos basadas en listas, pilas, árboles, tablas hash, conjuntos, pilas, colas y ordenación se realizaron utilizando la biblioteca estándar. El resultado fue un incremento considerable de la velocidad y lo más importante, mantenibilidad. Al trabajar con la seguridad de no tener que pensar en implementar estructuras de datos cuya única finalidad pudiera dar soporte a la verdadera aplicación que se quería construir, se pudo avanzar mucho más rápidamente en la construcción de la aplicación en sí. Pese a todo, la utilización de estas clases no fue aleatoria, sino que se tuvo que seguir aplicando conocimientos de estructuras de datos para saber qué tipo de estructura convenía en cada momento [Meyers'01].

2.3. Uso de las librerías DirectX de Microsoft.

La creación de un motor de juegos necesita trabajar de forma rápida con los gráficos, la entrada y el sonido. Para el desarrollo de *Crisol* se han utilizado las librerías DirectX. En concreto la versión 7.1a. Para hacer funcionar *Crisol* bastará con tener una versión igual o superior a la comentada. Las librerías DirectX fueron creadas por Microsoft para que la creación de juegos en su sistema operativo Windows. Gracias a ellas, el tratamiento de los gráficos, el sonido, la entrada, las comunicaciones, etc, es lo suficientemente rápido como para que se puedan ejecutar juegos bajo Windows. Antes de las DirectX existía una librería multimedia pero que hacía aún inviable el poder desarrollar juegos "serios" en esta plataforma. Con la llegada de las DirectX todo cambio y los desarrolladores fueron dejando paulatinamente el sistema operativo MS-DOS para pasarse definitivamente a Windows [Bargen'98].

Las librerías DirectX poseen varias componentes cada una de las cuales están orientada a una misión. Si bien, en cada nueva versión de DirectX estas sufren notables cambios e incluso pueden llegar a desaparecer. Todos estos cambios están supeditados a la evolución de la

tecnología en el momento de ser desarrollada una nueva versión de DirectX (por ejemplo, en la versión 7, el tratamiento de los gráficos se utilizaba con DirectDraw o Direct3D dependiendo de si se deseaba, o no, utilizar las capacidades gráficas de las tarjetas 3D. En la versión 8 de DirectX, estos dos componentes pasaron a fusionarse, existiendo sólo DirectGraphics que trabaja directamente sobre tarjetas 3D).

En *Crisol* se hace uso de las componentes Direct3D, para el manejo de gráficos, DirectSound, para el manejo de sonidos WAV, DirectMusic, para el manejo de archivos musicales MIDI, y DirectInput para el tratamiento de la entrada de una forma rápida (cuando hablamos de entrada, hacemos referencia al manejo del dispositivo de teclado y de ratón). Para que *Crisol* funcione, será absolutamente necesario que el ordenador que lo ejecute posea de una tarjeta de sonido y una tarjeta de vídeo con soporte 3D (como se comentará en el apartado siguiente).

Como las librerías DirectX están programadas en C, se optó por crear unas clases que las envolvieran, haciendo mucho más fácil el trabajo con las mismas desde el punto de vista de la programación [Root'99]. Estas clases luego fueron compiladas como una librería estática y enlazadas a la aplicación. Gracias al uso de esta librería de clases, el paso de una versión a otra de DirectX es mucho menos traumático para el código general pues los cambios a realizar se centralizan en unas clases y no en el propio código de *Crisol*.

2.4. Representación gráfica usando dispositivos 3D.

En las fechas por las que comenzó a desarrollarse el proyecto, las tarjetas 3D estaban ya completamente popularizadas, por lo que el sistema de vídeo de los ordenadores ya trataban de forma mucho más eficiente todo aquello que estuviera representado internamente con polígonos. Además, con la llegada de las tarjetas 3D la realización de efectos especiales (iluminación, transparencias, escalado, etc) se pueden realizar de una forma realmente rápida pues éstas aceleran por hardware la de los algoritmos [Wolverson'00].

Crisol iba a ser un motor que representara el universo de juego en 2D, por lo que en principio no iba a poder aprovecharse de las ventajas de las tarjetas 3D. Finalmente, se encontró una forma de seguir representando el motor en 2D pero utilizando las capacidades gráficas de las tarjetas 3D. Así, aunque *Crisol* es una aplicación que muestra el universo de juego a través de elementos prerenderizados (los tradicionales sprites), en contraposición a lo que sería dibujar en tiempo real (3D), usa el componente Direct3D de las DirectX para hacer uso de las tarjetas 3D (en el apartado de diseño se comentará cómo se resolvió este problema). Naturalmente, el uso de Direct3D obliga a disponer de una tarjeta gráfica 3D compatible con DirectX (cosa que, a fecha de hoy, poseen la práctica totalidad de ordenadores PC).

2.5. Patrones de diseño.

La utilización de patrones de diseño fue una constante en la construcción de la aplicación pues había que dar soluciones elegantes, desde la orientación a objetos, a problemas planteados desde el punto de vista del análisis (más adelante, se comentarán dichos problemas en la sección de análisis). En el desarrollo de arquitecturas para motores de videojuegos, no es extraño que se desarrollon patrones específicos [Salkin'98] si bien, en *Crisol* el 99% de ellos corresponderán a casos universalmente conocidos.

La totalidad de patrones "populares" han sido sacados de las referencias [Gamma'95] y [Rollings'00] sin embargo, una de los mas utilizados en las clases de *Crisol* y sus herramientas ha sido el tomado de la referencia [Arévalo'01], donde se implementa un protocolo de inicialización y finalización para las clases muy interesante.

Los patrones de diseño "populares" utilizados se citarán a continuación:

2.5.1. Patrón Singleton.

Crisol es una aplicación formada por un determinado número de subsistemas diferentes, estos subsistemas existen en la aplicación encarnados por una sola instancia. El objetivo del patrón

singlenton es, precisamente, asegurar que sólo exista una instancia asociada una determinada clase por lo tanto, todos los subsistemas del motor utilizan este patrón.

2.5.2. Patrón Facade.

Gracias a las clases facade se pudo reducir la complejidad de trabajo con los distintos subsistemas. Estas clases coordinan los componentes que forman parte de un subsistema. En realidad, un subsistema en *Crisol* es una clase singlenton que actúa de facade con respecto a los componentes (que a su vez serán otras clases) que lo forman.

Las clases construidas para envolver las librerías DirectX, también actuarán de facade con respecto a estas. Quizás este sea el ejemplo más claro de utilización de este patrón de diseño en el motor, pues con la utilización de estas clases, el trabajo con DirectX (formado por un número considerable de funciones y estructuras) se hace realmente sencillo en el motor.

2.5.3. Patrón Bridge.

Todas las clases que en *Crisol* pueden ser solicitadas por diferentes instancias, utilizarán este patrón. Con el, toda clase sensible de ser solicitará heredará una interfaz. La interfaz no será más que una estructura con métodos virtuales puros. De esta forma, el cliente de una clase solicitará dicho interfaz y no se tendrá que preocupar de más (se usarán estructuras porque por propia definición, éstas poseen siempre sus métodos públicos). Será la instancia que herede de dicha interfaz la que añada el código para los métodos virtuales. Con este interfaz, aunque se pueda perder algo de velocidad al utilizar métodos virtuales puros, las ventajas de organización y mantenibilidad de código serán muy elevadas. No hay peligro de llamada a métodos que deban de permanecer ocultos para un cliente y se logra una separación completamente nítida entre la interfaz de una clase y la verdadera implementación de la misma. Todos los subsistemas, heredan de interfaces.

Cabe puntualizar que en todo el código de *Crisol* el uso de herencia múltiple se ha evitado, pues es una fuente habitual de problemas de diseño. Sin embargo, las clases en *Crisol* pueden heredar una o varias interfaces. En tales casos no se considerará la existencia de una herencia múltiple, sino la herencia de interfaces para que se puedan usar sólo ciertas funcionalidades de una clase.

2.5.4. Patrón Observer.

Este patrón permitirá la comunicación de cualquier tipo de cambio en una clase a todas aquellas instancias que actúen de observadoras a la misma. Este patrón en una aplicación como *Crisol*, que es altamente interactiva, ha sido imprescindible. Así, por ejemplo, cualquier modificación en el jugador es inmediatamente notificado a los paneles de interfaz informativos.

Todos los clientes observadores de una clase siempre serán recogidos a través de interfaces (patrón bridge) de tal forma que cuando ocurra un cambio se notifique de forma homogénea a los mismos.

2.5.5. Patrón Command.

Este patrón se utilizó para encapsular la lógica en la que las distintas entidades del universo de juego (principalmente las criaturas) resuelven ordenes. Por así decirlo, todas las acciones concretas en el motor son comandos contenidos en una cola, que pasan a ejecutarse de forma cíclica.

2.5.6. Patrón Iterator.

Este patrón, ampliamente representado en las STL, se utilizó para “navegar” entre las distintas clases contenedoras, esto es, aquellas que contienen colecciones de objetos. Resulto extremadamente útil para recorrer las entidades existentes en un determinado área de juego.

2.6. Procesadores de lenguaje.

Probablemente la tecnología más importante sea la referida a los procesadores de lenguaje [Cueva'98] y [Aho'98]. En este sentido, se implementaron dos tipos de lenguajes. Por un lado uno relativo a la configuración y diseño y otro, el más importante, el relativo al lenguaje script que sirva para implementar la respuesta que el diseñador desee dar a los distintos eventos que se produzcan en el juego y al que se le denominó *CrisolScript*.

Para la implementación de *CrisolScript* fueron utilizados dos utilidades. Por un lado *Flex* para transformar el código fuente a tokens con los que ir reconociendo la gramática de *CrisolScript* y por otro *Bison* para procesar los tokens detectados por *Flex* con el objetivo de validarlos en la gramática del lenguaje [Aaby'96]. El compilador, construido a partir de los ficheros generados por estas utilidades, se programó en lenguaje C puro por motivos de compatibilidad, ya que tanto *Flex* como *Bison* producen ficheros en ese lenguaje. Como ya se comentó, el resto de elementos del proyecto fueron realizado en C++ con metodología orientada a objetos.

La utilización de *Flex* y *Bison* se debió a motivos de ahorro de tiempo en el desarrollo del compilador. De no utilizarlos, hubiera llevado un tiempo considerable la creación del compilador (sobre todo con respecto a lo que ofrece *Bison*) [Huebner'97].

Con el compilador de *CrisolScript*, se genera un archivo binario de código intermedio (muy inspirado en la máquina virtual de *Java*) que posteriormente es interpretado por una máquina virtual alojada en el motor [Vennc].

Las características de *CrisolScript* serán enumeradas en la sección de análisis, pues todas ellas están orientadas a ofrecer unas comodidades básicas de cara a la creación de scripts para los eventos del motor.

3. Análisis.

En esta sección se realizará el planteamiento del problema y un posterior refinamiento del mismo para que al final sepamos exactamente qué es lo que queremos hacer.

3.1. Planteamiento general del problema.

Se pretende construir una aplicación capaz de ejecutar juegos de rol. Estos juegos habrán sido diseñados utilizando una serie de herramientas que producirán archivos con datos. Estos archivos serán interpretados y manipulados por *Crisol* con el fin de conocer cómo deberá de representar los juegos diseñados y cómo deberá de actuar según sean las acciones del jugador. Así pues, el motor deberá de ofrecer una doble personalidad; para los diseñadores será una herramienta con la que construir juegos de rol y para los jugadores un videojuego de rol.

Crisol trabajará con el concepto de universo de juego. Un juego estará compuesto por un universo y todo universo estará dividido en diferentes áreas (al menos deberá de existir un área). Al trabajar con el concepto de universo de juego, *Crisol* llevará en todo momento la cuenta horaria, mostrando visualmente los efectos de la noche (oscuridad) y día (claridad). El diseñador también tendrá la oportunidad de crear áreas en donde la luz ambiente permenzca fija (independientemente de la hora en la que se halle el universo de juego).

En todo momento el jugador deberá de hallarse en un área. En cada área, el jugador podrá interactuar con diferentes entidades. Estas entidades podrán ser criaturas (el jugador también se considerará una criatura), ítems (objetos transportables y/o equipables), objetos de escenario y paredes o puertas (estas entidades deberá de mostrarse transparentes siempre que puedan obstaculizar la visión del jugador). Todas las acciones que se realicen en las distintas áreas deberán de recordarse, de tal forma que cuando el jugador regrese los cambios que hubiera realizado permanezcan (por ejemplo, transportar ítems de un área a otra). Las entidades podrán tener un valor luminoso asociado y ser dibujadas a diferente altura sobre el suelo (celda) en la que se encuentren (por defecto la altura será 0). También se podrá asociar, de forma optativa, un componente sombra que variará en intensidad según la hora del día.

Toda las áreas se construirán gráficamente a partir de celdas o tiles. Para que una celda del área sea válida (se pueda visitar) deberá de tener asociada una celda de suelo. Las celdas de techo serán opcionales y de existir, no deberán de dibujarse si con ello tapan al jugador cuando éste se halle por debajo de ellas. Las celdas del suelo podrán tener una elevación variable de tal forma que cuando una entidad se coloque sobre ellas, su altitud con respecto a 0 varíe según la celda. También se podrá asociar a las celdas del suelo sonidos según su tipo, de tal forma que cuando una criatura ande sobre ellas se oigan pasos acordes al tipo de suelo que se pisa. Con respecto a los techos, cada uno de ellos podrá colocarse a la altura que se desee.

A la hora de crear los juegos el diseñador deberá de poder modificar el aspecto del motor de forma completa. Esto incluirá poder cambiar la posición, color y aspecto de todos los elementos de interfaz. El diseñador también podrá definir a su gusto las reglas que estipulen y coordinen el funcionamiento lógico del motor.

El diseñador contará con un lenguaje script muy sencillo que le permitirá asociar, a los eventos que se produzcan en el motor, los sucesos que el crea oportunos con el fin de poder dar sentido al juego que quiera crear.

Con respecto a las reglas, se deberá de desarrollar un sistema que, sin estar ceñido a ningún tipo de juego, permita adaptarse a los diversos estándares y posibilidades existentes en los productos comerciales (es decir, no existirá un sistema de magia ni tampoco las criaturas tendrán atributos como peso o dinero pero el sistema de reglas de *Crisol* deberá de poder ser capaz de adaptarse para que el diseñador pueda crear dichas características).

Desde el punto de vista del jugador que ejecute un juego creado para *Crisol*, éste verá una aplicación, un juego. Como tal, el jugador podrá realizar una serie de acciones que estarán determinadas por el estado en el que se haya situado a *Crisol*. Para pasar al estado inicial se

deberá de arrancar la aplicación. Desde ahí se podrá comenzar un nuevo juego, recuperar una partida previamente guardada, ver los créditos o bien retornar a Windows saliendo del motor. En caso de querer comenzar una nueva partida, se pasará al estado de creación del personaje y una vez creado, se comenzará a ejecutar el juego propiamente dicho. Desde el juego se podrá también recuperar partidas o bien guardar la actual. Lógicamente, también se podrá salir del juego para retornar a la pantalla principal.

Al tratarse de un motor destinado a juegos de rol, el jugador tendrá la posibilidad de crear el personaje que le representará durante el juego. Las posibilidades en la creación del jugador habrán sido estipuladas por el diseñador (en cuanto a la posibilidad de personalizar la criatura a crear como jugador. Tanto en aspecto como en atributos).

Durante el juego el usuario (jugador) podrá mover al personaje por el escenario que se le presente e interactuar con los diferentes elementos del universo de juego, teniendo la posibilidad de entablar conversación, observar, manipular, examinar objetos contenedores (esto es, que contengan a ítems, como por ejemplo un baúl), utilizar ítems sobre otras entidades, equipar ítems, gestionar el inventario o entrar y huir de combate. El resto de acciones destinadas a controlar su huida de combate, observar su inventario, sus atributos básicos, utilizar lo que lleve equipado u obtener información que el juego (el diseñador) le notifique serán gestionados por una barra de interfaz que se localizará en la parte inferior de pantalla.

Siempre que el jugador entre en combate, el universo de juego pasará a un modo por turnos de tal forma que en cada turno habrá una sola criatura que pueda actuar. Para poder realizar acciones en ese turno, la criatura en cuestión poseerá una serie de puntos de combate. Una vez gastados, se pasará a la siguiente criatura. Cuando todas las criaturas declaradas como enemigas del jugador sean destruidas o bien el jugador huya, el combate terminará y se retornará de nuevo al sistema en tiempo real.

Para que una criatura pueda combatir ésta siempre deberá de pertenecer a uno de los dos posibles bandos de combate existentes. El primero de ellos estará asociado al jugador, es decir, todas las criaturas que se hallen en dicho bando serán amigas del jugador. El segundo será el reservado para los enemigos del jugador. El combate siempre empezará cuando en cada uno de los bandos se halle al menos una criatura. El jugador siempre se hallará en su propio bando.

Una criatura será aceptada en un bando sí y sólo sí una o más criaturas ya alineadas en un bando se hallan dentro de su rango de acción. El rango de acción de una criatura indicará la capacidad de la misma a la hora de detectar criaturas a su alrededor. Siempre que una criatura entre en el rango de otra ésta última deberá de ser avisada. Lo mismo ocurrirá cuando suceda lo inverso, esto es, una criatura que se hallaba en el rango de otra abandone el mismo.

El motor deberá de ser capaz de gestionar efectos especiales sobre las entidades (animaciones que duren el tiempo que tarden en reproducirse) de tal forma que se puedan implementar efectos clásicos como explosiones, magias, etc.

Durante el juego se podrán crear secuencias cut-scene que supondrán mostrar la visión del área de juego como si fuera una pantalla de cine y donde el personaje no podrá ser controlado por el jugador (todas las entidades deberán de ser manejadas vía script).

Se deberá de implementar un sistema de gestión de presentaciones y muestra de archivos gráficos para enseñar los créditos del juego, pantallas estáticas, secuencias de pantallas para narrar historias con animaciones y sonidos, etc.

El jugador podrá guardar la partida actual o recuperar una previamente guardada. A la hora de guardar, el jugador dispondrá de tantas oportunidades como espacio libre tenga en su disco duro. Desde la sección destinada a guardar partidas, también podrá borrar aquellas que ya no le interese mantener en el disco duro.

El jugador tendrá la oportunidad de pausar el juego por si desea ausentarse.

La representación gráfica será en 2D con perspectiva isométrica para dar sensación de profundidad (falso aspecto 3D en los gráficos). Se esperará un uso de las tarjetas 3D para acelerar el trabajo con los gráficos.

Crisol deberá de poder utilizarse para crear juegos a distintas resoluciones: 640x480, 800x600 y 1024x768.

Con respecto a la reproducción de sonido, el motor deberá de poder reproducir sonidos WAV y MIDI. Los sonidos WAV podrán estar asociados a las animaciones que se muestren, al sonido ambiente del juego, ser reproducidas cuando una entidad desee hablar o bien cuando se desee reproducir un sonido de forma anónima. Los ficheros MIDI, por otro lado, se utilizarán para reproducir música de fondo. Esta música se podrá localizar en la pantalla principal, en la de creación de personaje o bien durante el juego.

Para facilitar la localización de todos los archivos que compongan un juego construido para *Crisol* desde el punto de vista de la distribución y para evitar que los usuarios a los que va destinado un juego puedan modificarlos, se deberá de dar soporte para que el motor pueda ser capaz de trabajar con un sistema de archivos empaquetados.

Por último, *Crisol* deberá de ejecutarse en plataformas Windows.

3.2. Análisis de los requisitos.

En esta sección se realizará un análisis en profundidad de todos y cada uno de los requisitos a tener en cuenta y que fueron planteados de forma general en la sección anterior. Una vez que se lea este apartado, deberá de estar totalmente claro y especificado qué es lo que se quiere hacer.

3.2.1. Requisitos para las entidades.

Todas las áreas diseñadas en *Crisol* poseerán un conjunto de entidades que gestionar. Estas entidades podrán ser criaturas, objetos de escenario, paredes o bien ítems. El jugador se considerará una criatura especial.

Todas la entidades existirán asociadas al área en donde fueron establecidas. El jugador será una excepción ya que podrá moverse por las distintas áreas. Los ítems también serán una excepción ya que si el jugador transporta un ítem podrá llevarle de un área a otra.

Una entidad podrá ser destruida en tiempo de ejecución. Cuando esto ocurra, jamás se podrá volver a tener acceso a la misma.

Toda entidad tendrá asociada un valor de elevación, salvo los ítems, que permitirá situar a estas entidades a una determinada altitud con respecto a la celda en la que se hallen. Esa altitud será la suma de su elevación más la de la propia de la celda en la que se hallen.

Toda entidad podrá tener un valor luminoso asociado.

3.2.1.1. Las entidades de tipo criatura.

Las criaturas serán las entidades que se utilizarán para representar al jugador (la criatura que maneje el usuario del juego) y a los npc's (las criaturas con las que se relacione el jugador).

Se podrán crear de forma dinámica criaturas sobre un determinado área (el área que se halle activo). A la hora de crearlas, se podrá indicar si una vez abandonado el área donde se crearon deberán de destruirse o bien podrán permanecer asociadas al mismo.

Toda criatura será un contenedor pues dispondrá de un inventario donde transportará ítems y de unos slots de equipamiento donde los equipará (naturalmente, sólo a aquellos que sean equipables).

El jugador podrá interactuar con las demás criaturas para observarlas, registrarlas (una vez que estén muertas), dialogar con ellas (siempre que no se esté en modo combate) y utilizar una habilidad (si es que posee alguna).

Cuando una criatura sea registrada por otra, la segunda podrá examinar todos los ítems que la primera transporte.

Las criaturas podrán desplazarse por un área en movimiento normal o bien corriendo.

Los atributos particulares de toda entidad criatura serán los siguientes:

- **Nombre.** El nombre de la criatura. En el caso del jugador, el nombre será introducido por parte del usuario al editar su personaje.

- **Tipo y Clase.** Una criatura poseerá un sólo un tipo y una sola una clase. En ambos casos, los valores irán de forma independiente (a diferencia del resto de entidades).

- **Género.** Una criatura deberá de pertenecer a uno de los dos sexos posibles.

- **Salud.** Atributo impuesto por *Crisol*, pues estará íntimamente ligado con el sistema de combate. La salud tendrá dos valores enteros, el temporal y el base. Se trabajará con el temporal ya que el base servirá para recordar el valor original. Una criatura se considerará muerta si su valor temporal de salud es igual o inferior a 0.

- **Nivel y Experiencia.** Atributos impuestos por *Crisol*. Medirán el nivel de la criatura y la experiencia asociada y acumulada de la misma respectivamente.

- **Puntos de combate.** Atributo impuesto por *Crisol*. Los puntos de combate se utilizará para decidir las acciones que una criatura podrá llevar a cabo durante un enfrentamiento en su turno de combate.

- **Rango**. Toda criatura será capaz de detectar cuando otra criatura se halla cerca de ella o bien cuando una que se hallaba previamente cercana se aleja. El rango servirá para delimitar esta capacidad.

- **Habilidades**. Existirá un conjunto de habilidades definidos en las reglas. Una criatura podrá tener, o no, activa una determinada habilidad. Las habilidades podrán ser de tipo presencial o usable. Las de tipo usable podrán usarse sobre cualquier otra entidad.

- **Síntomas**. Una criatura podrá tener activo, o no, un determinado número de síntomas. Los síntomas servirán para indicar posibles estados sobre las criaturas (por ejemplo, para indicar que se está cansado, hambriento, etc).

- **Atributos extendidos**. Las criaturas podrán desaparecer de una serie de atributos extendidos asociados. Cada atributo extendido tendrá dos valores, el temporal y el base. Se trabajará con el temporal pues el base le servirá para saber el valor original. Estos atributos estarán pensados para completar la definición de las criaturas de cara a hacer extensibles las reglas a los deseos del diseñador (gracias a ellos, se podrán indicar atributos como peso, dinero, suerte, fuerza, etc).

- **Slots de equipamiento**. Existirán dos slots fijos impuestos por *Crisol*, que serán los relativos a los objetos que se podrán llevar en las manos. El resto de slots no tendrán porque existir, pero estarán pensados para representar el equipamiento que se pueda llevar en otras partes del cuerpo. Para que un ítem se pueda equipar en un slot, deberá de haberse especificado en el archivo de reglas con la validación pertinente.

3.2.1.2. Las entidades de tipo ítem.

Los ítems representarán objetos transportables por una entidad contenedora. Los ítems también podrán ser equipados por las criaturas en slots, siempre y cuando esto sea posible.

Los ítems podrán crearse de forma dinámica y nunca serán eliminados al abandonarse el área en donde se creen.

Las criaturas podrán interactuar con los ítems para observarlos, manipularlos, equiparlos y usarlos sobre otra entidad (siempre si se poseen en inventario), recogerlos del escenario y hablar con ellos.

Cuando un ítem sea equipado por una criatura y dicho ítem tenga un valor de luz asociado, dicho valor luminoso se trasladará a la criatura que lo equipó.

Los atributos particulares de toda entidad ítem serán los siguientes:

- **Nombre**. Nombre del ítem.

- **Tipo y Clase**. Todo ítem deberá de tener un tipo y una clase asociada a ese tipo.

- **Atributos globales**. Estos atributos serán los que todo ítem poseerá (independientemente de su tipo y clase). Todo atributo será un valor entero.

- **Atributos locales**. Los atributos locales se definirán para cada familia de tipos existentes, de tal forma que todas las clases de ese tipo los poseerán. Un atributo local será un valor entero.

- **Coste de uso en combate**. Cada tipo de ítem existente podrá ser usado o no en combate. En caso de poderse usar, se le deberá de asociar un coste en los puntos de acción.

- **Slots en los que el ítem podrá equiparse**. Cada tipo de ítem existente podrá, o no, ser equipado en los diferente slots de una criatura.

3.2.1.3. Las entidades de tipo objeto de escenario.

Los objetos de escenario representarán elementos del área que, a diferencia de los ítems, no podrán recogerse y mucho menos equiparse (por ejemplo árboles, piedras, faroles, baúles, etc).

Los objetos de escenario podrán ser contenedores para almacenar ítems.

Las criaturas podrán interactuar con los objetos de escenario observándolos, manipulándolos, hablando con ellos o bien utilizando una habilidad sobre ellos. Al manipularlos, si el objeto de escenario es un contenedor se podrá examinar todos los ítems que posea en su interior.

Los objetos de escenario podrán tener un flag asociado que indique que no puedan ser manipulados, con lo que únicamente formarán parte del escenario de juego.

Los atributos particulares de toda entidad objeto de escenario serán los siguientes:

- **Nombre**. Nombre del objeto de escenario.

- **Tipo y Clase**. Todo objeto de escenario deberá de tener un tipo y una clase asociada a ese tipo.

- **Atributos locales**. Los atributos locales se definirán para cada familia de tipos existentes, de tal forma que todas las clases de ese tipo los poseerán. Un atributo local será un valor entero y permitirá al diseñador personalizar las entidades que vaya diseñando.

- **Flag contenedor**. Este flag indicará si el objeto de escenario puede contener o no ítems.

3.2.1.4. Las entidades de tipo pared.

Las paredes serán unas entidades que servirán para representar todos aquellos elementos del escenario cuya forma general sea el de una pared (paredes, muros, puertas, etc).

Toda pared podrá tener una de las tres orientaciones posibles: sudoeste (lateral izquierda), sudeste (lateral derecha) o frontal (sur).

Una criatura podrá observar, manipular, hablar y usar una habilidad sobre una pared.

Las paredes podrán tener un flag asociado que indique que no puedan ser manipulados, con lo que únicamente formarán parte del escenario de juego.

Siempre que una pared obstaculice al jugador, ésta se hará transparente.

Las paredes podrán tener activo un flag de obstaculización. Cuando tal hecho ocurra, no se podrá franquear a través de ellas. Este flag será imprescindible para poder trabajar con la paredes como si fueran puertas.

Los atributos particulares de toda entidad pared serán los siguientes:

- **Nombre**. Nombre de la pared.

- **Tipo**. Tipo asociado a la pared.

- **Atributos locales**. Atributos locales de la pared. Los atributos serán valores enteros y permitirán al diseñador personalizar las entidades que vaya creando.

- **Orientación**. Orientación de la pared.

3.2.2. Requisitos para la especificación de las reglas.

Crisol no estará gobernado por unas reglas concretas. El único sistema impuesto será el de combate y aquellos atributos que puedan considerarse necesarios para las entidades.

Las reglas se especificarán por secciones. Habrá secciones destinadas a las celdas del suelo, a las distintas entidades (criaturas, ítems, objetos de escenario y paredes), al sistema de combate, al universo de juego y a las distancias mínimas para realizar ciertas acciones.

El trabajo con las reglas será, fundamentalmente, el de asociar identificadores a valores numéricos de tal forma que luego, se pueda trabajar con dichos identificadores en diferentes partes del diseño de juego (el motor conocerá el valor numérico y características de cada uno de los identificadores).

3.2.2.1. Reglas asociadas a los tiles del suelo.

Se deberán de establecer los identificadores asociados a los distintos tipos de tiles existentes, que formarán el suelo de las celdas de un área.

3.2.2.2. Reglas asociadas a las entidades de tipo criatura.

Se deberán establecer todos los identificadores relativos a los tipos y clases de criaturas, a los dos géneros existentes, a los de habilidades y síntomas (teniendo en cuenta que no serán obligatorios y que en el caso de las habilidades, también habrá que incluir si son usables o presenciales y los punto que costarán si son usadas durante un combate) y por último, también se deberán de asociar los identificadores de los atributos extendidos y los slots de equipamiento (al menos se deberán de especificar dos identificadores que corresponderán a la mano izquierda y derecha respectivamente).

3.2.2.3. Reglas asociadas a las entidades de tipo ítem.

Cuando se defina un identificador de tipo, se deberán de identificar también los identificadores de clase asociados, los de atributos locales y la información relativa a los identificadores de los slots de equipamiento de las criaturas en donde se podrá equipar el ítem (si es que es posible equiparlo). Si el ítem puede ser usado en combate, habrá que indicar también el coste en cuanto a puntos de acción.

De forma independiente a los identificadores de tipo, se deberán de establecer los identificadores de atributos globales asociados a los ítems.

3.2.2.4. Reglas asociadas a las entidades de tipo objeto de escenario.

Todos los identificadores de los objetos de escenario estarán supeditados a los distintos identificadores de tipos que se definan. Así, los identificadores de clase, los identificadores de atributos locales y la información sobre si el objeto de escenario es o no contenedor, estarán asociados a un identificador de tipo.

3.2.2.5. Reglas asociadas a las entidades de tipo pared.

Para trabajar con las reglas de pared, deberemos de establecer los identificadores asociados a los distintos tipos. Asociados a los mismos se tendrán que indicar los identificadores de atributos locales. Debemos de recordar que las paredes no tendrán identificadores de tipo clase asociada.

3.2.2.6. Reglas asociadas al combate.

Se deberá de establecer los puntos que costará ejecutar las distintas acciones de interacción a las criaturas cuando éstas se hallen en modo combate. Estas acciones serán relativas a andar, equipar un ítem, desalojar un ítem equipado, usar un ítem de inventario, observar, coger un ítem, manipular una entidad y atacar. Se entenderá por atacar el uso de un ítem establecido en uno de los slots de equipamiento que representen a la mano izquierda o la derecha.

A la hora de iniciar un combate, los participantes en él se irán insertando en una lista. En las reglas se podrá especificar el atributo extendido a partir de cual se ordenará dicha lista. Bastará especificar uno de los identificadores de atributo extendido que se definieron para las criaturas. En caso de no especificar nada, la lista se ordenará según las criaturas soliciten ingresar.

3.2.2.7. Reglas asociadas a las distancias.

Las distancias serán necesarias para saber a partir de qué posición pueden ejecutarse las acciones que procedan. Se deberá de tener en cuenta que una celda representará una unidad de distancia. Según esto, se deberán de especificar las distancias para hablar, para usar las distintas habilidades usables definidas, para usar aquellos ítems con los que se pueda atacar y para establecer la distancia mínima con a la que el jugador podrá huir de una criatura alineada como enemiga. Por defecto, para realizar todas estas acciones, las criaturas deberán de estar en posiciones adyacentes con respecto al objetivo.

3.2.2.8. Reglas asociadas al controlador de juego.

En esta sección únicamente bastará con especificar la cantidad de segundos que supondrán un minuto de juego.

3.2.3. Requisitos sobre la representación del universo de juego.

La representación del escenario de juego será en perspectiva isométrica, es decir, cada porción de terreno será tratada como una baldosa en forma de diamante tal y como se muestra en la figura:

Un tile isométrico, siendo la parte rosada zona invisible (no dibujada).

Un elemento muy importante de la organización del universo de juego, como se verá más adelante, será el área de juego. El motor no podrá llevar a memoria, de una sola vez, toda la extensión del universo así que lo que hará será llevar porciones. A una porción se la llamará área de juego. Existirán unas dimensiones máximas y mínimas para un área, de tal forma que todo área diseñado deberá de tener un tamaño en dicho intervalo.

3.2.3.1. Cámara y actor principal.

El motor isométrico se dibujará según el concepto de cámara – actor principal, pudiendo ser el actor una celda o bien una entidad. El actor principal siempre será, por defecto, el jugador pero el diseñador podrá cambiar el actor principal para realizar escenas narrativas o de cut - scene. Si el actor principal no es el jugador, no podrá dar ordenes a la criatura que se encarga de representarle en el juego.

Cuando una entidad sea el actor principal, el motor isométrico dibujará aquella porción del área que haga que dicha entidad ocupe el tile central (se vea en pantalla).

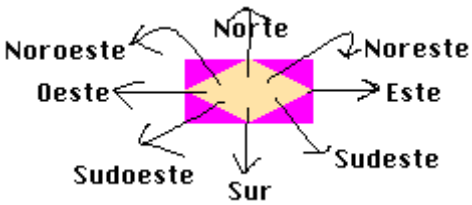
3.2.3.2. Dibujado de celdas y tiles.

El área estará modelado a partir de un motor isométrico y dicho motor isométrico estará construido en base a celdas. En dichas celdas, se hallarán los tiles base (suelo), los tiles superiores (techo) y las entidades de tipo ítem, objetos de escenario, criaturas y paredes.

Para que una celda se pueda visitar y pueda poseer otros elementos sobre ella (entidades y tiles techo) deberá de tener un tile base. En caso de no tener asociado uno, ese hueco aparecerá de color negro. Un área al menos deberá de poseer un tile base para situar al jugador.

Cuando el actor principal esté sobre una celda que tiene techo, acto seguido ese techo no se dibujará, pudiéndose ver lo que está por debajo de dicho techo que no será otra cosa más que el actor principal. Todas las demás celdas que posean un techo sí lo dibujarán siempre y cuando, no pertenezcan a la misma región en la que se halle el techo no dibujado. Para solucionar este problema, a la hora de diseñar un área se podrán crear recintos que formen habitaciones.

Toda las celdas del mapa tendrán asociadas un campo denominado de accesibilidad, de tal forma que se pueda indicar desde qué direcciones un npc puede acceder a ellos. Las posibles direcciones serán norte, sur, este, oeste, noreste, noroeste, sudeste y sudoeste. Por defecto, una celda será accesible desde todas esas direcciones. En la siguiente figura se muestra la organización de la accesibilidad:



Acceso desde un tile a otras posiciones adyacentes.

El orden dibujado de las entidades sobre una celda será: a) tiles base (suelo), b) objetos de escenario, c) ítems, d) criaturas e) paredes y f) tiles superiores (techos).

3.2.4. Requisitos sobre la accesibilidad y posicionamiento de las entidades.

Para que una criatura pueda acceder a una determinada entidad, *Crisol* tendrá en cuenta si es posible llegar a esa entidad con una determinada distancia mínima que leerá de las especificaciones de reglas según sea la acción ha realizar.

Recordando las posibles entidades con las que el jugador podrá interactuar, teníamos a) ítems, b) criaturas, c) objetos de escenario, e) paredes y f) accesorios de pared

Siempre que *Crisol* determine que una accesibilidad es correcta, realizará las siguientes operaciones 1) moverá al jugador al tile desde donde podrá realizarse la interacción y 2) enviará el evento de interación pertinente. En caso de que no se pueda alcanzar la posición deseada para realizar una determinada acción, se indicará al jugador.

3.2.5. Requisitos sobre el sistema de combate.

El sistema de combate será un módulo impuesto por *Crisol*. Se esperará que todo juego desarrollado con este motor incorpore el combate. Las criaturas verán reflejado este hecho en los atributos de salud y puntos de combate.

El combate estará dirigido por turnos, de tal forma que todas las criaturas que participen en un combate tengan asignado un turno. Durante ese turno de combate, podrán realizar todas las acciones que les permitan sus puntos de combate o bien pasar de turno sin realizar ninguna acción.

3.2.5.1. Alineación de combate.

Crisol distinguirá, durante el combate, dos alineaciones. Toda criatura que reciba un turno durante el combate deberá de pertenecer a una de estas dos alineaciones. Si una criatura no recibe un turno de combate será por no estar alineada en combate. Las alineaciones serán:

a) Alineación de amigos del jugador.
b) Alineación de enemigos del jugador.

Por imposición, el jugador siempre se alineará como a), de tal forma que una criatura deberá de alinearse como b) para que sea enemigo del jugador.

Todas aquellas criaturas que, viéndose sobre el terreno de juego, no pertenezcan a una de estas tres alineaciones no realizarán acción alguna durante el combate (no les llegará un turno).

En base a estas dos alineaciones, cuando una criatura reciba un turno de combate, tendrá acceso a dos grupos de criaturas. Toda criatura que esté alineada pertenecerá a uno de estos grupos, menos la criatura que recibe el turno, que no aparecerá en ninguno. Estos serán el conjunto de criaturas amigas y enemigas. De esta forma, una criatura alineada en a) tendrá por enemigos a todas aquellas alineadas en b) y viceversa.

Para que una criatura pueda alinearse será indispensable que dentro de su rango se halle al menos una criatura previamente alineada (independientemente de la alineación de ésta última). Este concepto de rango evitará el que puedan comenzarse combates con criaturas completamente alejadas las unas de las otras.

3.2.5.2. Inicialización de los combates.

El motor pasará al modo combate siempre y cuando exista al menos una criatura en el conjunto de amigos del jugador y en el de enemigos del jugador. Como el jugador siempre pertenecerá al primero, en el momento de que una criatura se halle en el conjunto de enemigos del jugador se iniciará el combate.

3.2.5.3. Establecimiento de turnos de combate.

En el archivo de reglas se podrá especificar el parámetro con el que se creará la lista interna de criaturas que irán poseyendo el turno de combate. Por defecto, el criterio para la elaboración de la lista será el orden en que las criaturas se incorporen al combate. Sin embargo, se podrá indicar el identificador de un atributo extendido para ordenar en base al mismo (por ejemplo, si el diseñador crea un atributo extendido denominado “destreza”, se podría ordenar en base a él).

3.2.5.4. Finalización del combate.

El combate puede finalizar por una de la siguientes causas:

- **El jugador muere**. Es la más elemental. Cuando esto ocurra se generará un evento de muerte sobre la criatura que represente al jugador y el diseñador tendrá la oportunidad de finalizar el juego actual o transportarlo a otra localidad.

- **El jugador desea finalizar el combate**. Esto podrá tener dos interpretaciones. Podrá considerarse como una huida si consideramos que el jugador no comenzó el combate o bien lo comenzó y ha tenido contacto con enemigos (y no ha destruido a todos). Sólo se le permitirá finalizar si no es visto por ninguna criatura enemiga (una criatura que está alineada como enemiga) en el momento de realizar tal acción. La otra interpretación será la de una finalización obligada debido a que en el momento de realizar tal acción no haya criaturas alineadas como enemigos.

- **Cuando no quedan enemigos**. Ocurrirá cuando le llegue el turno al jugador y no queden criaturas alineadas como enemigos.

3.2.5.5. Acciones posibles durante el combate.

Al ser un combate por turnos cada una de las criaturas que estén alineadas poseerán un turno de actuación. En dicho turno podrán realizar las acciones que estimen oportunas siempre y cuando se puedan realizar en combate y se posean suficientes puntos. Las acciones que no se podrán realizar en combate serán las de dialogar con una criatura o examinar una entidad contenedora. El coste de puntos por acción quedará definido en el archivo de reglas.

Cada vez que una criatura realice una acción válida, *Crisol* restará de los puntos de acción de la criatura el coste de la acción realizada. Si los puntos se agotan mientras se realiza tal acción (solo ocurrirá cuando a una criatura se le ordene desplazarse de una celda a otra), *Crisol* pasará de turno automáticamente. Esto mismo ocurrirá cuando se ordene realizar una acción y la cantidad de puntos de combate sea cero. El comportamiento a seguir cuando se ordene realizar una acción para la que no haya suficientes puntos de combate pero éstos sean superiores a cero diferirá de si la criatura es un jugador o un npc. Si es un jugador, se le notificará que no podrá realizar dicha acción pero si es un npc, se pasará de turno automáticamente.

3.2.6. Requisitos sobre el sistema de iluminación y sombras.

Crisol utilizará un sistema de iluminación muy simple (no será capaz de generar sombras en tiempo real) para indicar cómo de iluminado estará un tile y todo lo que sobre él se halle.

Toda entidad podrá tener un factor de iluminación asociado y para cada hora en la que se halle el universo de juego, también se podrá especificar el factor luminoso ambiente. Sobre este último aspecto, añadir que se permitirá diseñar áreas que simulen espacios cerrados con factores de luminosidad constantes, es decir, que el paso del tiempo no les afecte en cuanto a variar el valor de luz ambiente.

Cuando una criatura se halle sobre una celda y posea un factor luminoso afectará a un radio que estará directamente relacionado con la intensidad de la luz que posea. En el caso de los ítems, si estos tienen un factor de luminosidad y son equipados (que no transportados) por una criatura, su factor de luminosidad será tomado como si fuera el de la criatura.

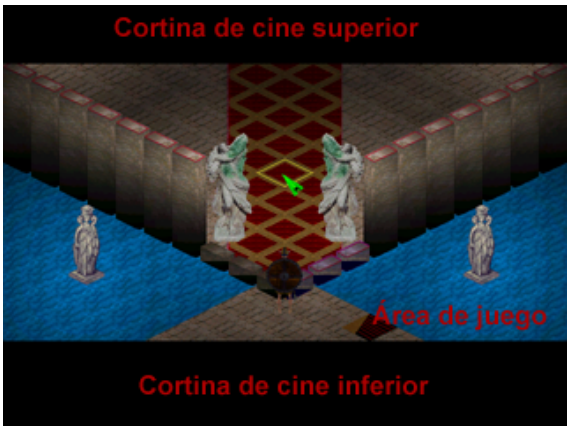
Siempre que se quiera dibujar un elemento de pantalla se tendrá en cuenta el factor de luminosidad del mismo y el factor de oscuridad del ambiente. Restando ambos valores (siempre que el factor de oscuridad sea superior al de luminosidad) se obtendrá un valor indicativo de lo oscuro que se verá dicho objeto en pantalla.

Las sombras no se generarán en tiempo real sino que serán gráficos que se podrán asociar a una entidad de tal forma que sirva para simular tal efecto. Lo que sí que se podrá hacer será el relacionar para cada hora del día una intensidad diferente para la sombra (en cuanto a su transparencia).

3.2.7. Requisitos sobre la creación de escenas automáticas.

Las escenas automáticas se generarán poniendo a *Crisol* en un estado denominado de cut - scene. En dicho estado, el controlador de juego no generará ningún evento relacionado con el control del jugador. Será responsabilidad del diseñador encontrar la forma de entrar y salir de ese estado.

Visualmente, la entrada en estado de cut - scene supondrá que el aspecto se corresponda al de una pantalla de cine (cortina superior e inferior de negro). He aquí una imagen de ejemplo:



Ejemplo donde se pueden distinguir las cortinas de cine superior e infierior.

Durante el estado de cut - scene sólo el diseñador podrá controlar lo que transcurra en el juego pues el usuario (jugador) no podrá dar ordenes al personaje que controla.

3.2.8. Requisitos sobre el sistema para la creación de presentaciones.

Se implementará un sistema que permitirá definir presentaciones basadas en “diapositivas” entendiendo que una diapositiva será una imagen con la posibilidad de tener asociada animaciones, música de fondo y texto. El paso de una diapositiva a otra, se podrá realizar de forma automática (cada x tiempo se pasará de diapositiva) o bien mediante el deseo del usuario.

El diseñador creará las presentaciones que podrá mostrar en cualquier momento durante el juego. El uso de las presentaciones permitirá al diseñador mostrar imágenes o narrar situaciones mediante pantallas gráficas.

Será necesario crear al menos una presentación para mostrar los créditos de juego, que será una opción obligatoria en el menú principal (más adelante se analizarán los menús o estados en los que podrá hallarse el motor).

4.2.9. Requisitos para la creación de efectos especiales.

Los efectos especiales tendrán como objetivo simular efectos clásicos en los juegos tales como explosiones, magias, etc. Su existencia será temporal y no serán más que animaciones que se asociarán a entidades.

Para crear un efecto especial bastará establecer una animación, indicar en qué posición de la entidad deberá e aparecer (en la parte baja, a la misma altura o en la parte superior de la entidad) y durante cuánto tiempo aparecerá (hasta que se quite o bien hasta que pase un determinado número de segundos). Como complemento, se podrá variar por vértice los valores RGB de color.

3.2.10. Requisitos sobre la interfaz de control en juego.

Por interfaz de control en juego entenderemos la forma en la que el jugador podrá interactuar con el escenario, dar ordenes al personaje que controla y gestionar los menús y cuadros de diálogo que se le vayan presentando.

Toda la interfaz será gráfica y controlada con el ratón salvo en los casos en los que sea requerido insertar texto.

3.2.10.1. Control del jugador.

Durante el juego, el usuario (el jugador) dará ordenes al personaje que controle utilizando un cursor gráfico. Estas ordenes podrán dividirse en dos grupos:

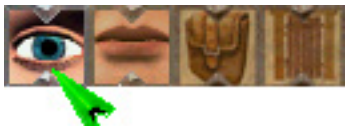
a) Ordenes relacionadas con el movimiento.
b) Ordenes relacionadas con la iteración.

Mediante el botón izquierdo del ratón se ordenará y mediante el botón derecho se podrá cambiar del estado de movimiento al de interacción.

En el caso a), el cursor simplemente señalará celdas. Una pulsación en el botón izquierdo del ratón supondrá que el jugador deberá de moverse a esa celda. Esta orden podrá darse aún cuando el personaje estuviera ejecutando un movimiento previo (en tal caso, el personaje interrumpirá la orden anterior y tendrá sólo en cuenta esta última orden). Naturalmente, en caso de que el jugador haya ordenado un movimiento a una celda a la cual no se puede llegar, el personaje no realizará la orden encomendada. Algo parecido sucederá cuando el jugador señale con el cursor a una celda que posea un objeto de escenario, otra criatura o bien no posea un tile base. En este caso, como será completamente imposible mover hasta ahí al personaje, el cursor podrá cambiar de forma (por ejemplo, podrá mostrarse de color rojo) para indicar de inmediato que esa posición no será alcanzable.

El caso b) será más complejo, pues el cursor deberá de adaptarse dependiendo de la entidad a la que estemos señalando. Si la entidad no es interactuable no pasará nada pero si fuera interactuable, el cursor podrá cambiar de forma para indicar que es posible realizar algún tipo de acción sobre la misma mientras muestra el nombre de la entidad señalada.

Según sea la entidad seleccionada para interactuar las acciones que se podrán realizar sobre la misma variarán. Todas estas acciones serán accesibles mediante un menú flotante que aparecerá nada más pulsar el botón izquierdo del ratón. Dicho menú estará formado por iconos que indicarán una de las acciones posibles a realizar. Un ejemplo sería el siguiente:



Ejemplo de menú flotante

Mover el ratón sobre el menú supondrá seleccionar un icono del menú y pulsar al botón izquierdo ordenar la acción. Como ayuda, siempre que se seleccione un icono para interactuar deberá de aparecer en el cursor el nombre de la acción que represente.

El menú flotante se adaptará según sea el tipo de la entidad con la que interactuar sin embargo, habrá que tener en cuenta que existirán tres opciones adicionales que siempre se añadirán. La primera de ellas será relativa a la posibilidad de usar un ítem sobre la entidad seleccionada, la segunda a la de poder usar una habilidad (si es que el jugador posee habilidades y éstas son usables) y la tercera la de salir del menú flotante para cancelar la interacción.

3.2.10.2. Ventanas y componentes.

Todo cuadro de diálogo o menú que aparezca en *Crisol*, será tratado de forma interna como una ventana virtual. Todas las ventanas, cuando estén activas, serán las dueñas absolutas de la entrada.

Las ventanas virtuales tendrán siempre componentes que serán los que se encarguen de implementar las funcionalidades lógicas de las mismas. Todos los cuadros de diálogos y menús se construirán en base a dichos componentes. Los componentes que podremos distinguir en una ventana serán los siguientes:

- **Imagen**. Componente que servirá para mostrar una imagen.

- **Bloque de texto**. Componente capaz de mostrar un bloque largo de texto por partes, es decir, mostrando sólo una serie de líneas.

- **Línea de texto**. Componente encargado de mostrar una línea de texto y que además será capaz de ser seleccionado actuando como un “botón”.

- **Botón gráfico**. Componente capaz de tener asociado un botón visual con los estados de selección y no selección.

- **Botón con animación asociada**. Tendrá las misma prestaciones que el componente anterior salvo que se representará utilizando una animación externa. Cuando la animación esté asociada, el botón podrá ser seleccionado.

- **De entrada de texto**. Componente para que el usuario (jugador) pueda introducir información.

- **Barra de porcentaje**. Componente para mostrar una información numérica a partir de dos valores: uno temporal y otro base. La barra de porcentaje podrá dibujarse tanto en vertical como en horizontal.

Utilizando todos estos componentes y el concepto de ventana virtual, se construirán todos los menús y cuadros de diálogo que existirán en el motor. El diseñador podrá configurar cada uno de estos componentes, con respecto a la ventana a la que pertenezcan, en posición, color, dimensiones, aspecto, sonido, etc.

3.2.10.3. Interfaz principal de juego.

Esta será la ventana de interfaz más importante durante el juego. Ocupará toda la zona inferior de la pantalla y en ella el usuario podrá obtener información sobre los sucesos que ocurran. Además podrán tener acceso a acciones que, de otra forma, no se podrán llevar a cabo. A modo de esquema general, las opciones e informaciones que figurarán en esta ventana de interfaz serán:

- Acceso al inventario y ficha del jugador
- Consola de juego (donde el diseñador podrá mandar notificaciones).
- Información elemental referida al jugador.
- Acceso a los ítems colocados en los slots principales (manos).
- Acceso a las opciones de combate relacionadas con la huida y el paso de turno.

Ejemplo del interfaz principal de juego usando una distribución de elementos de ejemplo

Dependiente de la interfaz principal de juego encontraremos el cuadro horario que servirá para indicar gráficamente la hora en la que se halla el universo de juego y que podrá seleccionarse con el cursor y los iconos que muestren los síntomas que el jugador tenga activos. Dichos síntomas así como el cuadro de información horario, tendrán un componente de transparencia en el dibujado para que no tapen la porción de área por donde se pueda mover e interactuar el jugador.

Durante el combate el interfaz de juego principal también se encargará de gestionar la información relativa a los puntos de combate que posea una criatura con el turno de tal forma que mediante la muestra de una barra de porcentaje se pueda conocer los puntos de acción a utilizar.

La interacción del jugador pasará a ser de ataque siempre y cuando uno de los slots principales sea seleccionado, ya que *Crisol* entenderá que en ese caso se querrá utilizar un ítem equipado en las manos o bien las mismas manos, para atacar o golpear a otra entidad. Cuando tal hecho ocurra, si se selecciona una criatura para interactuar con ella, *Crisol* mostrará información complementaria sobre la misma (tipo, clase y salud). Para el resto de entidades no mostrará información alguna.

3.2.10.4. Trabajo con las entidades contenedoras.

Se entenderá por contenedor aquella entidad que pueda contener ítems. Podrán ser contenedores los objetos de escenario (un cofre, una mesa, un armario, etc) y lo serán siempre las criaturas (pues todas dispondrán de inventario).

La manipulación de contenedores estará basada en la visualización de una ventana especial con todos los ítems que contenga el contenedor y otra con los ítems que posea el jugador. La idea será que sobre dicha ventana, el jugador pueda intercambiar ítems en un sentido u otro. No se podrá hacer, en la ventana de la entidad con la que se interactúa, operaciones de manipulación. Sólo de observación de los ítems (y esto incluirá los ítems de ambas ventanas) y traslado de un contenedor a otro. Sin embargo, en la ventana asociada al inventario del jugador se podrá realizar la acción extra de tirar un ítem al suelo. Para poder intercambiar el contenido de ambas, bastará con señalar el ítem de una ventana y pulsar al botón izquierdo.

Cuando la ventana de manipulación de contenedores se halle abierta, el jugador no podrá ni moverse ni interactuar sobre el escenario hasta que se cierre

3.2.10.5. Inventario y ficha del jugador.

Esta ventana de información y gestión ocupará toda la pantalla y será accedida desde el interfaz principal de juego. Al activarse el juego se pausará parcialmente. Desde ella se podrá “navegar” por el inventario del jugador, gestionar los slots de equipamiento, usar los ítems que éste transporte sobre sí mismo y ver los valores de sus atributos así como la información complementaria (nombre, nivel, experiencia, puntos de combate y salud). También se mostrarán aquellos síntomas activos y habilidades que se posean.

Para poder equipar un ítem en inventario, bastará con arrastrar un ítem desde el inventario a un slot de equipamiento. *Crisol* se encargará de comprobar las reglas y verificar si dicho ítem puede colocarse en el slot requerido.

3.2.10.6. Interfaz asociado al sistema de conversación.

La conversación siempre estará orientada a que se realice entre el jugador y cualquier entidad que pueble el universo de juego (no se limitará a las criaturas porque eso supondría cortar las posibilidades de diseño). Naturalmente, deberá de ser el diseñador quien cree las conversaciones para cada una de las entidades con las que se pueda hablar.

El interfaz para llevar a cabo la conversación ocupará la misma zona destinada a la interfaz de juego principal. En ella aparecerán los retratos de las dos entidades que estén dialogando (el jugador y la otra entidad) y al lado las líneas de texto que el jugador podrá escoger para hablar.

El interfaz estará preparado para dar soporte a un número grande de opciones de tal forma que si no entran en el área del interfaz de conversación exista la posibilidad de realizar scroll sobre las opciones de diálogo.

3.2.10.7. Interfaz de uso de un ítem sobre entidad.

Este interfaz aparecerá cuando el jugador desee escoger uno de los ítems que lleve en inventario (y que no esté equipado) para usarlo sobre una determinada entidad. También se permitirá cancelar la acción.

3.2.10.8. Cuadros de diálogo.

Existirá una serie de cuadros de diálogo que podrán ser utilizados por el diseñador para obtener información del jugador o bien para notificar algo a éste. Dichos cuadros de diálogo serán:

- **El de selección de texto**. Mostrará unas líneas de texto y el jugador deberá de escoger una de ellas. Se podrán indicar líneas no seleccionables así como la posibilidad de cerrar el cuadro de diálogo sin escoger nada.

- **El de notificación simple**. Mostrará una información escueta al jugador.

- **El de pregunta**. Mostrará una pequeña pregunta al jugador y este podrá escoger si la acepta, la rechaza o cancela tal pregunta.

- **El de muestra de archivos de texto**. Podrá cargar un archivo de texto y mostrarlo en pantalla. Resultará muy útil cuando se quiera ofrecer información detallada sobre algo.

Cuando un cuadro de diálogo esté activo éste será el único medio que recogerá la entrada del jugador.

3.2.11. Requisitos del lenguaje script para la implementar la lógica de juego.

Toda la lógica de los juegos que se creen con *Crisol* será implementada a través de un lenguaje script al que se denominará *CrisolScript*.

El lenguaje deberá de ser sumamente sencillo para que cualquier diseñador sin conocimientos previos de programación pueda manejarse con él. De esta forma, existirán tan sólo tres tipos de datos: los valores numérico (`number`), las cadenas de caracteres (`string`) y los handles a las entidades (`entity`). *CrisolScript* permitirá el trabajo tanto con variables como con constantes de los tipos comentados.

CrisolScript trabaja con una sintaxis similar a lo que pudiera encontrarse en un lenguaje orientado a objetos si bien, se tratará sólo de un maquillaje para implementar de una forma mucho más clara y sencilla el trabajo con ciertos elementos. Existirá un “objeto” llamado `Game`

que se encargará de simular al motor como aplicación, otro llamado `World` que representará al universo de juego y, por último, todas las variables de tipo `entity`, que se manejarán como objetos. Asociados a los mismos existirán una serie de métodos.

CrisolScript tendrá soporte para la declaración de funciones en los cuerpos de los scripts así como la posibilidad de importar funciones globales a los mismos. Este último hecho permitirá trabajar mucho más cómodamente ya que facilitará la reutilización de código.

Los scripts programados con *CrisolScript* se asociarán a los distintos eventos que puedan suceder en el juego. Algunas acciones sólo existirán si existe un script asociado al evento que lo representa (por ejemplo, la acción de observar sólo aparecerá si existe un script asociado).

Los eventos que sucedan podrán ser informativos o bien estar asociados a alguna acción concreta. Por ejemplo, el evento de observar es informativo mientras que el evento de coger un ítem del suelo estará asociado a una acción concreta. Desde los scripts se podrá retornar un valor que indicará si el motor deberá, o no, de realizar la acción por defecto asociada al evento producido. De esta forma, si después de lanzar el script asociado a la acción de coger indicamos que no queremos que se realice la acción por defecto, la criatura no recogerá el ítem del suelo.

Pese a que cada uno de los scripts será tratado de forma individual, existirá un espacio global en donde se podrán definir variables a las que podrán acceder todos los scripts. Esto será muy importante para facilitar la comunicación entre distintos scripts.

Para poder procesar de una forma rápida los scripts se deberá de construir un compilador que los transforme a un formato de código intermedio. Además, el compilador, al que denominaremos *CSCompiler*, realizará un chequeo para que no exista ningún tipo de error en el código fuente. Para procesar dicho código intermedio, el motor dispondrá de una máquina virtual interna.

CrisolScript permitirá el acceso y control a cualquier parte del motor dando la seguridad al diseñador de que no podrá bloquear con sus acciones el motor. Es decir, la máquina virtual que interprete los scripts en el motor, deberá de ser capaz de detectar todos errores lógicos que pudieran existir en el código a procesar e interrumpir el script en ejecución (pero sin finalizar *Crisol*).

3.2.12. Requisitos del lenguaje sentencial para la configuración y diseño.

Para posibilitar la configuración de los interfaces del motor, el diseño de las áreas, la creación de plantillas de animación, especificación de reglas, diseño de presentaciones y efectos especiales así como las posibilidades que tendrán los jugadores para crear sus personajes cuando comiencen a jugar, se usará un lenguaje sentencial. Por sentencial entenderemos un lenguaje en el que únicamente asignaremos valores a una serie de variables. Dichas variables estarán organizadas en secciones de tal forma que sepamos qué es lo que estamos configurando.

Cada uno de los elementos configurables y diseñables con este lenguaje será un archivo de texto con formato "`.cbt`" siendo completamente necesario hacer uso de la documentación de apoyo para saber las variables que habrá que especificar por sección.

Como apoyo existirá una herramienta denominada *CrisolBuilder* que procesará todos estos archivos en busca de errores y creará para algunos de ellos otros equivalentes en formato binario. Esto último será necesario para que luego *Crisol* pueda procesar los datos de una forma mucho más rápida. En concreto, se pasarán a formato binario los archivos de diseño de áreas y los de plantillas de animación ya que serán los que más información deberán de almacenar y procesar. Los archivos binarios poseerán el mismo nombre que el original pero con formato "`.cbb`".

A la hora de definir las entidades, animaciones, presentaciones, etc, se utilizará el concepto de plantilla. Una plantilla no será más que una sección junto a todos los valores para los atributos

de la información que representa. De esta forma, bastará con indicar el nombre de la plantilla (sección) para que *Crisol* localice toda la información relativa. Este concepto tendrá una importancia capital a la hora de diseñar áreas y animaciones pues podremos definir plantillas que luego usaremos en múltiples lugares sin tener que definir todos los datos de nuevo.

3.2.13. Requisitos sobre el uso de un sistema de ficheros empaquetados.

Para facilitar la localización de todos los ficheros de datos, se deberá de implementar un soporte para el trabajo con archivos empaquetados. Un archivo empaquetado será aquel que permita contener en su interior a otros archivos.

Crisol utilizará un archivo empaquetado llamado "`GameData.pak`" en donde el diseñador podrá insertar ficheros gráficos, de diseño de áreas, etc de tal forma que ningún jugador pueda realizar modificaciones.

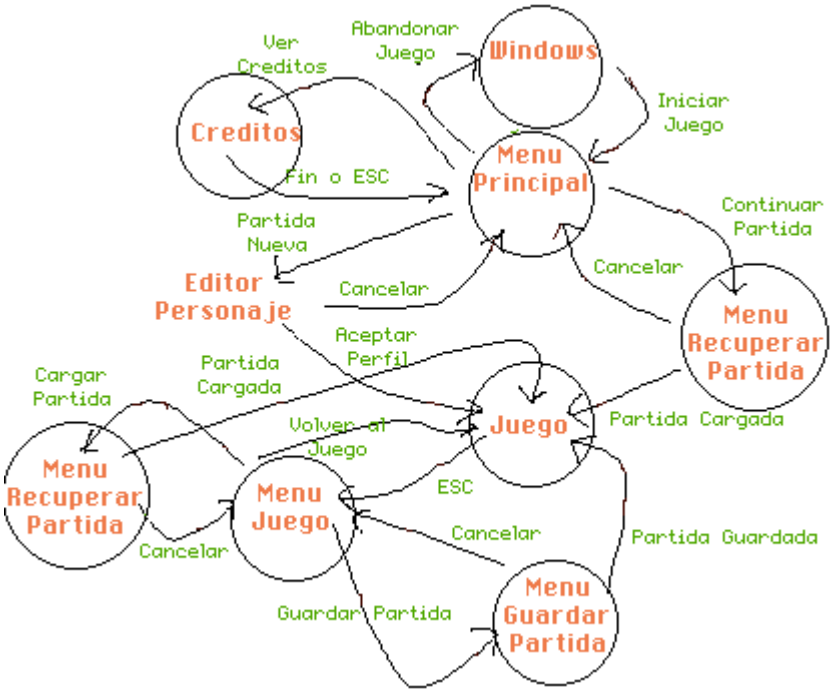
Para la gestión del archivo empaquetado se usará una utilidad llamada *CPAKTool* que tendrá como objetivo permitir al diseñador añadir archivos, borrarlos, conocer los insertados o bien extraerlos.

Crisol podrá trabajar tanto con este archivo como con ficheros individuales. Para lograr este objetivo, siempre que se ordene a *Crisol* buscar un fichero éste mirará primero si existe de forma individual y en caso de no hallarlo buscará en el archivo empaquetado "`GameData.pak`".

Los únicos tipos de ficheros que no se permitirá añadir al empaquetado "`GameData.pak`" serán los de audio (WAV y MIDI) ya que el trabajo con DirectX requiere que dichos ficheros se hallen de forma individual. No ocurrirá así con los ficheros gráficos (tga) y de datos que utilizará *Crisol*.

3.2.14. Requisitos sobre los estados de Crisol como aplicación.

Todo juego implementado con *Crisol* tendrá un mismo grafo de ejecución. Los estados de dicho grafo y las transiciones quedarán especificadas en la siguiente figura:



Grafo de ejecución de Crisol

- **Windows**. Este será el estado inicial y final. Inicial porque será desde donde se arrancará el motor (juego) y final porque será donde se llegue al finalizarlo.

- **Menú Principal**. Será el estado a donde se llegue una vez arrancado el juego. Estará compuesto por un fondo con imagen y una serie de opciones. Las opciones estarán destinadas a permitir el acceso a los estados de "*Comenzar un nuevo juego*", "*Recuperar una partida*", "*Ver créditos*" y "*Abandonar el juego*". Asociado a este menú, el diseñador podrá especificar un fichero musical MIDI de fondo y un texto informativo sobre la versión actual del juego.

- **Ver Créditos**. En esta sección se esperará mostrar una presentación donde se indiquen el autor o autores del juego.

- **Recuperar Partida**. Desde este estado el jugador podrá recuperar una partida previamente guardada. Podrá ser accedido desde el menú principal o bien desde el menú de juego. En esta sección, se mostrarán todas las partidas guardadas. Al seleccionar una de ellas se mostrará la información complementaria que el motor guardó automáticamente al grabar la partida. Siempre se podrá abandonar este estado si no hay ninguna partida para cargar o bien se desea cancelar la acción, volviéndose al estado anterior.

- **Creación del Personaje**. Este estado será obligatorio en caso de querer iniciar una partida nueva. Desde él, el usuario escogerá un perfil de personaje y lo acondicionará según sus preferencias. La pantalla de selección de perfil estará compuesta por un fondo con imagen. Sobre dicho fondo se habilitarán las siguientes zonas:

  - Zona de opciones de cambio de género y siguiente perfil.
  - Zona de introducción de nombre, selección del tipo y clase y muestra del nivel y experiencia.
  - Zona de atributos y puntos disponibles para repartir. Al lado de los atributos existirá una opción para incrementar y decrementar su valor base.
  - Zona de habilidades y coste de las mismas. También se mostrarán los puntos para poder elegirlas
  - Zona de retratos posibles a elegir. Sobre los retratos figurará un botón de pasar a siguiente retrato. El paso se hará de forma cíclica.
  - Zona con la opción de aceptación de perfil editado. Sólo se podrá aceptar cuando todos los datos hayan sido editados correctamente.
  - Zona de cancelación de perfil elegido y vuelta al menú principal..

  En el apartado de reglas se comenta más sobre la creación, por parte del usuario, de una criatura que asuma el rol de jugador. Indicar, por último, que se podrá asociar un fichero musical MIDI de fondo.

- **Juego**. Será el estado que represente el juego en sí. Se podrá llegar recuperando una partida previamente guardada o bien al comenzar un nuevo juego, justo después de editar el personaje que representará al jugador. Desde el juego se podrá grabar una partida, recuperar una previamente guardada o bien retornar al menú principal. Bastará con acudir al estado de menú de juego.

- **Menú de Juego**. Se llegará desde el estado de juego. Estará mostrado como una pequeña ventana con imagen de fondo y las opciones de grabar una partida, cargarla, retornar al juego o bien salir al menú principal. Durante el tiempo que se esté en este estado, el juego se hallará pausado.

- **Guardar Partida**. Desde aquí se podrá guardar la partida actual. Se accederá desde el menú de juego. Para guardar una partida bastará con introducir una descripción general. El motor asociará automáticamente datos complementarios para que a la hora de querer cargarla, se puedan tener más referencias. Es interesante añadir que desde aquí también se podrán borrar partidas previamente guardadas, con el fin de no saturar el espacio en el disco duro.

4. Diseño.

En este apartado daremos una solución a los problemas planteados en la etapa de análisis creando los diagramas de clases para *Crisol*, *CSCompiler*, *CrisolBuilder* y *CPAKTool*.

Para el trabajo con las DirectX se definirá una librería estática ("`.lib`") formada por diferentes clases, cada una de las cuales trabajará con aquellos componentes de DirectX que sean necesarios para construir *Crisol*. Será necesario analizar esta librería de clases antes de proceder a hacer lo propio con *Crisol*.

Las estructuras de datos y el manejo de cadenas serán todas controladas mediante el uso de las STL. En concreto, se utilizarán las clases que den soporte a las cadenas de caracteres, las listas, las colas con y sin prioridad, las pilas, árboles y tablas hash . El uso de estas estructuras de datos también implicará acudir a las clases iteradoras así como a la construcción, en algunos casos (que ya se señalarán), de predicados de búsqueda particulares.

4.1. La librería DX7Wrapper para el trabajo con las DirectX.

Crisol utilizará una librería de clases diseñada para trabajar con los componentes de DirectX que se vayan a utilizar (DirectDraw, Direct3D, DirectInput, DirectSound y DirectMusic) esto beneficiará en los siguientes sentidos [Root'99]:

- Adaptar el motor a una nueva versión de las DirectX no requerirá un cambio sustancial en el código del mismo pues la práctica totalidad de modificaciones se las llevarán las clases que compongan esta librería; a *Crisol* sólo le interesará el interfaz que ofrezcan las mismas.

- Se eliminará la complejidad de uso de estas librerías al encapsularlas en clases y se mejorará la legibilidad del código de *Crisol*.

- Se utilizará un código, el de las librerías, que habrá sido depurado y probado de forma independiente.

La definición de tipos, enumerados, estructuras, etc, que deban de tener un acceso global, se hallarán en espacios de nombres para facilitar su mantenibilidad y evitar la colisión con otros archivos de definición.

El esquema general de la librería de clases estará dividido en el siguiente conjunto de componentes:

- Clases para el trabajo con DirectDraw (`DXDraw`, `DXDDSurface`, `DXDDSurfaceBack`, `DXDDSurfaceScreen`, `DXDDSurfaceTexture`, `DXDDPalette` y `DXDDClipper`).

- Clases para el trabajo con Direct3D (`DXD3D`, `DXD3Device` y `DXD3DQuad`).

- Clases para el trabajo con DirectInput (`DXInput`, `DXDIKeyboard` y `DXDIMouse`).

- Clases para el trabajo con DirectSound (`DXSound`, `DXDSWAVBuffer` y `DXDSWAVSound`).

- Clases para el trabajo con DirectMusic (`DXDMManager` y `DXDMSegment`).

Algunos componentes aparecen por separado pero tienen grandes puntos en común. En concreto, esto ocurrirá con Direct3D que dependerá de DirectDraw, y con DirectMusic, que dependerá de DirectSound.

La versión con la que trabajará la librería será la 7.0a por lo que será necesario que cualquier equipo que ejecute *Crisol* posea dicha versión o superar, en caso contrario no se podrá ejecutar el motor.

4.1.1. La clase DXWrapper.

Todas las clases de cada conjunto de componentes heredarán de `DXWrapper`. Esta clase base proveerá de mecanismos para el rastreo de errores. Por un lado mantendrá el código del último error que se produjo y por otro, una cadena al método donde hallarlo.

4.1.2. Clases para el trabajo con DirectDraw.

El componente DirectDraw será la puerta de acceso al control de los dispositivos gráficos 2D pero también permitirá que se pueda trabajar con las 3D. De hecho, Direct3D hará uso del objeto DirectDraw para poder crearse (tal y como se verá en el siguiente apartado).

El diagrama de clases para el control de DirectDraw será el siguiente.

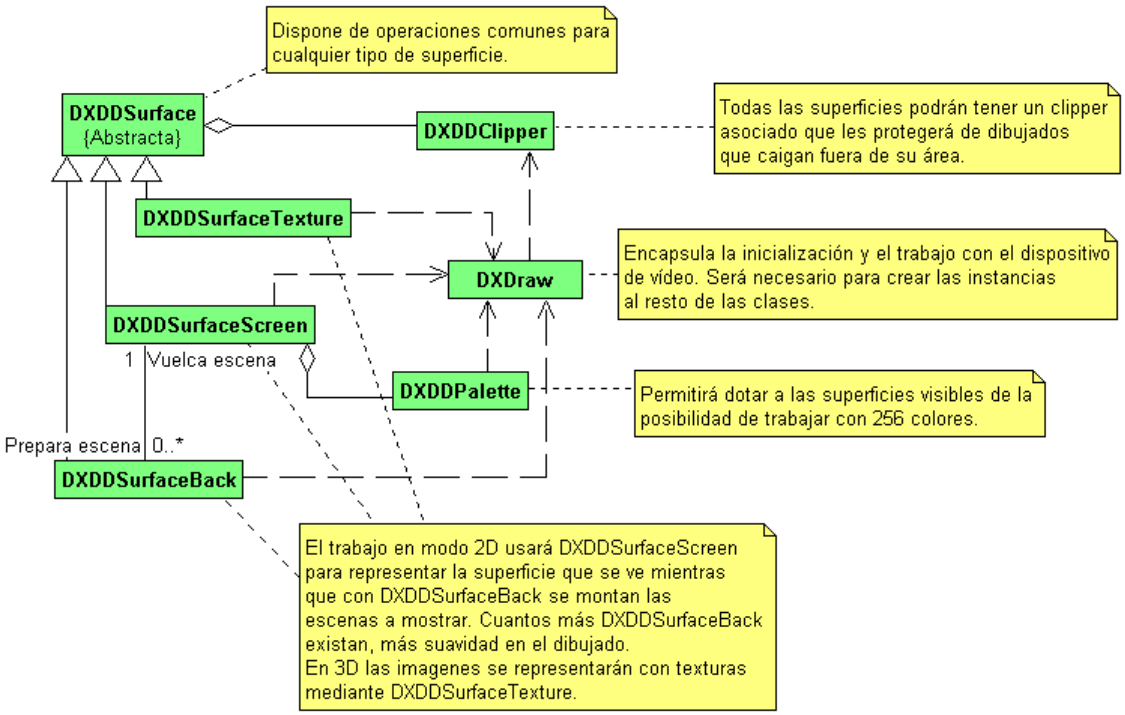


Diagrama del módulo DirectDraw

La clase más importante de todas será `DXDraw` ya que de ella dependerán las restantes para poder ser creadas. `DXDraw` representará el dispositivo gráfico sobre el que trabajar.

Para el trabajo con las superficies gráficas (las zonas de memoria que contendrán las imágenes y texturas) se utilizarán las clases `DXDDSurfaceScreen`, `DXDDSurfaceBack` y `DXDDSurfaceTexture`. Todas ellas heredarán de `DXDDSurface`, que dispondrá de métodos comunes a cualquier tipo de superficie. Dependiendo de si se trabaja en 2D o en 3D, será necesario utilizar unas u otras (en el caso de *Crisol* se trabajará en 3D). Si se trabaja en 2D, la forma de dibujar será poseyendo una superficie principal que representará lo que vemos en pantalla (`DXDDSurfaceScreen`) y utilizando otra para montar la escena que se va a mostrar (`DXDDSurfaceBack`). En el caso de trabajar en 3D, los elementos que dibujemos no serán más que polígonos a los que asociaremos superficies gráficas. Estas superficies se denominarán texturas (`DXDDSurfaceTexture`).

Las clases `DXDDClipper` y `DXDDPalette` estarán principalmente orientadas al trabajo en 2D. Con la primera se permitirá recortar (clippear) aquellos gráficos que se deseen dibujar sobre una superficie evitando errores al acceder a posiciones inválidas de memoria (una superficie no deja de ser una porción de memoria especial donde dibujar). Mediante `DXDDPalette` se permitirá trabajar con gráficos de 256 colores.

4.1.2.1. La clase DXDraw.

Mediante DXDraw se encapsulará el acceso directo a los dispositivos gráficos. Sin esta clase no se podrán crear las instancias a las demás. Ni si quiera se podrá trabajar con Direct3D, como se verá más adelante. Las principales operaciones serán las destinadas a crearse sobre un dispositivo gráfico y obtener las características del mismo.

4.1.2.2. La clase DXDDSurface.

Esta clase abstracta, encapsulará el objeto DirectDrawSurface y ofrecerá métodos complementarios para el trabajo con surfaces primarias (DXDDSurfaceScreen) y secundarias (DXDDSurfaceBack) además de para texturas (DXDDSurfaceTexture) de tal forma que dichas superficies hereden de esta clase.

4.1.2.3. La clase DXDDSurfaceScreen.

Esta superficie se utilizará para representar la pantalla, es decir, mostrará la escena que el jugador verá en un determinado momento. Para poder funcionar podrá tener asociadas cero o más superficies secundarias (DXDDSurfaceBack) que tendrán como objetivo suavizar el proceso de actualización de las imágenes a mostrar.

4.1.2.4. La clase DXDDSurfaceBack.

Este tipo de superficies se denominarán secundarias o no visibles ya que servirán de apoyo a las primarias. Otra importantísima utilidad será la relativa a que permitirán alojar información gráfica auxiliar (que pueda ser luego volcada a una primaria).

Para mejorar el proceso de dibujado, siempre se deberá de utilizar una superficie de este tipo pues la escena a mostrar en pantalla se compondrá primeramente aquí y luego, será volcada a la primaria, evitando efectos de parpadeo que pudieran ocurrir al usar una sóla superficie para crear la escena a mostrar. Cuanto mayor número de estas superficies haya asociadas a una primaria, más suave será el proceso de dibujado. Lo normal será trabajar con una o con dos si bien, habrá que tener en cuenta que el utilizarlas de esta forma consumirá una cantidad de memoria considerable, que se incrementará cuanto más elevada sea la resolución de vídeo utilizada.

4.1.2.5. La clase DXDDSurfaceTexture.

Esta clase encapsulará el trabajo con superficies gráficas destinadas a almacenar texturas. Las texturas se utilizarán únicamente cuando se esté trabajando con el componente Direct3D. Una textura será el elemento gráfico que se asociará a un objeto formado por polígonos.

4.1.2.6. La clase DXDDClipper.

Esta clase se encargará de gestionar que las operaciones de volcado de información gráfica a las superficies quede recortada cuando se intente dibujar fuera de los límites de las mismas. Para ello, la clase se deberá de asociar a las superficies interesadas. De no utilizar este objeto, tales comprobaciones deberán de hacerse manualmente pues de lo contrario, dibujar fuera del área asociada a una superficie sólo traerá errores de memoria.

4.1.2.7. La clase DXDDPalette.

Esta clase permitirá el trabajo con las paletas gráficas cuando el formato de vídeo sea de 8 bits por pixel (256 colores).

4.1.3. Clases para el trabajo con Direct3D

El trabajo con Direct3D dependerá del uso del objeto DirectDraw encapsulado en la clase DXDraw. Además, será en modo 3D cuando se haga uso de las superficies que representen a

las texturas. Será necesario usar una instancia `DXDDSurfaceBack` para montar sobre ella la escena que se vaya dibujando.

Un engine que trabajará con un entorno 3D en tiempo real necesitaría de un módulo gráfico muy complejo y, por tanto, de un mayor trabajo con el componente Direct3D de las DirectX. Como la necesidad de usar Direct3D en *Crisol* radica únicamente en la mejora del aspecto 2D (uso de alpha blending [Simmons'00a], transparencia y traslucencia [Simmons'00c] e iluminación [Pazera'01]), la complejidad de éste será pequeña. El diagrama de clases para el trabajo con Direct3D será el siguiente:

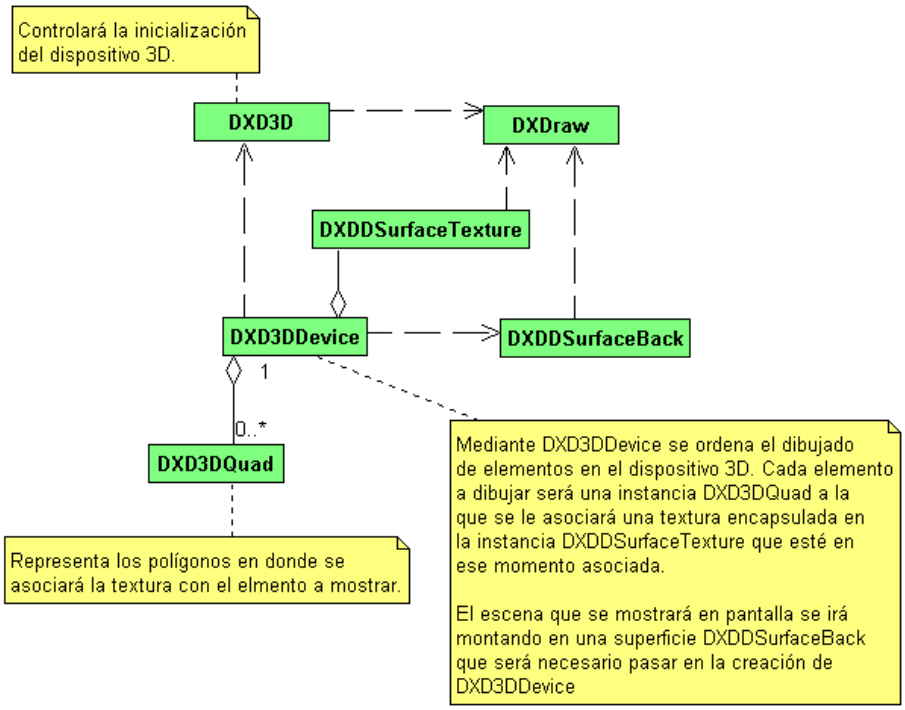


Diagrama del módulo Direct3D.

La clase `DXD3D` representará la inicialización del modo 3D para poder comenzar a trabajar en el dispositivo. Mediante el uso de `DXD3DDevice` se configurará el render y mediante `DXD3DQuad` creamos una representación para los gráficos 2D tradicionales que vayamos a dibujar en formato 3D (dos triángulos rectángulos solapados que formarán un rectángulo al que se le asociará el gráfico o la textura, a dibujar).

4.1.3.1. La clase DXD3D.

Esta clase será muy sencilla pero vital para poder trabajar en 3D pues encapsulará el objeto Direct3D y permitirá que se pueda crear la instancia `DXD3DDevice` para el trabajo directo con el dispositivo gráfico.

La creación de esta clase seguirá dependiendo del objeto DirectDraw por lo que se deberá de pasar una instancia correcta a `DXDraw`.

4.1.3.2. La clase DXD3DDevice.

Esta clase encapsula el Device 3D, esto es, el dispositivo que se encargará de dibujar en 3D sobre la pantalla [Simmons'00b]. Tambien mantendra el viewport que será el área de la pantalla en donde se realizaran las operaciones. El viewport tendrá el comportamiento de un clipper por lo que, al trabajar en 3D, no hará falta usar un objeto `DXDDClipper` para el recorte gráfico.

El dispositivo deberá de tener asociado una superficie `DXDDSurfaceBack` que será donde irá dibujando la escena. Paralelamente, existirá una superficie primaria, `DXDDSurfaceScreen`,

que tendrá asociada esa misma superficie `DXDDSurfaceBack` de tal forma que al hacer la operación de volcado, se realizará el intercambio limpio entre superficies y se mostrará la escena que se estaba dibujando.

Todas las operaciones de dibujado en el dispositivo 3D se deberán de llevar a cabo entre una llamada al método `BeginScene` y una llamada a `EndScene`.

Será muy importante que desde el exterior se llame el mínimo numero de veces a operaciones como `SetTexture`, `Draw`, o las `RT`*X* y `ST`*X*, o se sufrirá un considerable descenso en el rendimiento de dibujado [Simmons'00d]. Se esperará que desde el exterior existan algoritmos que se encarguen de realizar unas llamadas a los métodos comentados de la forma más austera posible. En *Crisol*, como se verá más adelante, estas consideraciones serán tenidas en cuenta por su subsistema gráfico.

4.1.3.3. La clase DXD3DQuad.

Esta clase representara un quad. Un quad sera la unidad con la que esta librería tratará a los objetos que deban de ser enviandos al Device para ser dibujados.

Cada objeto o sprite estará encerrado en un rectangulo (4 vertices). Mediante un quad, definiremos esta figura y la forma en la que las texturas seran asociadas a ella.

El establecimiento de una textura y su configuración en el Device más luego el envio de uno o más objetos `DXD3DQuad` producirán el dibujado de una escena.

4.1.4. Clases para el trabajo con DirectSound.

Las clases que pertenezcan a este componente se encargarán de acceder al dispositivo de audio y de reproducir ficheros de sonido WAV (reproducen sonido digitalizado).

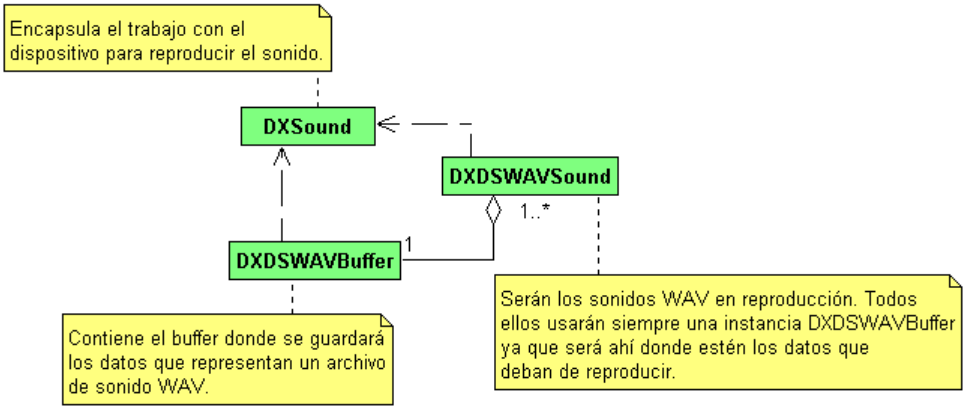


Diagrama del módulo DirectSound.

La clase `DXSound` se encargará de mantener el objeto DirectSound, que permitirá activar el dispositivo de audio. `DXSound` permitirá, también crear el resto de componentes así como los pertenecientes a DirectMusic, como se verá a continuación.

El trabajo con los archivos de sonido WAV se llevará a cabo con las clases `DXDSWAVBuffer` y `DXDSWAVSound`. La primera almacenará los datos de un archivo WAV en buffer y la segunda utilizará a la primera para reproducir esos datos almacenados. De esta forma, se ahorrarán muchos recursos del sistema cuando se deba de utilizar un mismo archivo de sonido en situaciones diferentes.

4.1.4.1. La clase DXSound.

Esta clase encapsulará el trabajo con el objeto DirectSound para el trabajo directo con la tarjeta de sonido. Será una clase fundamental pues desde ella se podrán crear el resto de instancias relativas a este componente y, también, al componente DirectMusic.

4.1.4.2. La clase DXDSWAVBuffer.

Esta clase mantendrá un buffer de sonido WAV. La creacion será a partir de un fichero WAV existente en disco (la clase implementa los metodos necesarios para cargar el fichero WAV y volcarlo en el buffer).

La clase únicamente contendra informacion, por lo que no incluirá métodos para reproducirla. La reproducción se realizará a partir de clases `DXDSWAVSound` que dependerá de ésta. Gracias a esta forma de gestionar el trabajo con el sonido, se ahorrará de forma considerable en recursos.

4.1.4.3. La clase DXDSWAVSound.

Esta clase se encargará de mantener una instancia de sonido que trabaje sobre un buffer (`DXDSWAVBuffer`) previamente creado. Desde aquí se podrá controlar la reproducción del fichero WAV según los flags con los que haya sido creado el buffer asociado.

Podrán existir una o más instancias `DXDSWAVSound` que utilicen un mismo buffer de sonido preservando cada una de ellas su propio estado de reproducción.

Internamente, se mantendrá otro objeto DirectSoundBuffer que será un duplicado del original mantenido por la instancia `DXDSWAVBuffer` pasada.

4.1.5. Clases para el trabajo con DirectMusic.

Para la reproducción de los ficheros de sonido MIDI se utilizará este componente. Estos ficheros están orientados a reproducir música instrumental a diferencia de los WAV. Para trabajar con este componente será necesario hacer uso de DirectSound. El diagrama de clases utilizado será el siguiente:

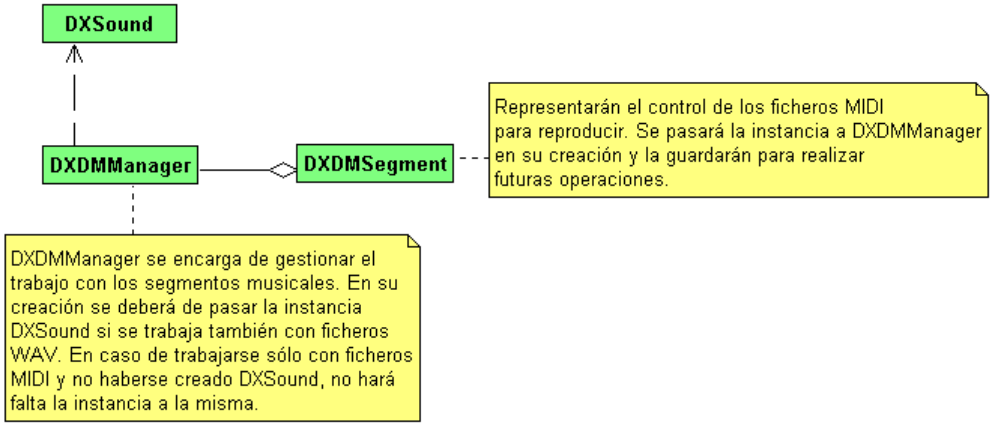


Diagrama del módulo DirectMusic.

La clase `DXDMSegment` servirá para trabajar con los segmentos musicales. Un segmento musical no será más que los datos que representará a un fichero MIDI. Con la clase `DXDMManager` se gestionarán los distintos segmentos o lo que es lo mismo, los distintos archivos MIDI. Para poder crear un segmento será necesario pasar `DXDMManager`.

4.1.5.1. La clase DXDMManager.

Esta clase encapsula el objeto IDirectMusicPerformance, encargado de manejar los segmentos que contendran los sonidos, y el objeto IDirectMusicLoader, encargado de cargar y buscar los ficheros de trabajo.

Todos los segmentos (instancias `DXDMSegment`) deberán de contar con el objeto `DXDMManager` para poderse reproducir.

4.1.5.2. La clase DXDMSegment.

Esta clase que trabajará con los segmentos de DirectMusic encapsulando los objetos IDirectMusicSegment e IDirectMusicSegmentState.

Un segmento vendrá a ser un ínterprete de un fichero musical MIDI de tal forma que sea el encargado de reproducirlos por la tarjeta de sonido.

4.1.6. Clases para el trabajo con DirectInput.

Para el control de la entrada, se utilizará el componente formado por las clases `DXInput`, `DXDIMouse` y `DXDIKeyboard`. La primera de las clases se encargará de mantener el objeto `DirectInput` mientras que las dos últimas trabajarán con el ratón y con el teclado respectivamente. El diagrama utilizado será el siguiente:

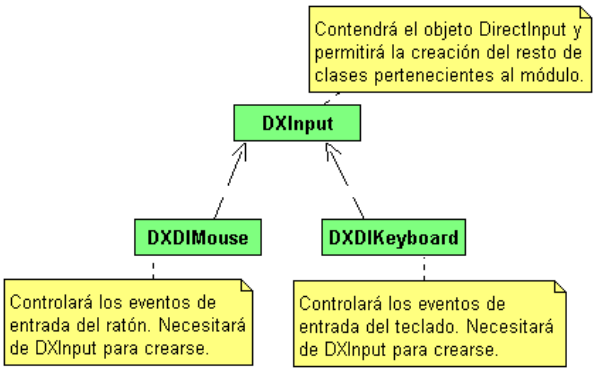


Diagrama del módulo DirectInput.

`DirectInput` tratará al teclado como un componente formado por botones, de tal modo que únicamente podrá ofrecer información sobre el estado de pulsación de los mismos (recién pulsado, pulsando, recién soltado y soltado). La particularidad de utilizar DirectInput para el tratamiento de la entrada por teclado se centrará en la rapidez de respuesta, que será muy superior en comparación al uso de la programación tradicional de Windows, basada en recibir la notificación en la ventana que representa la aplicación.

4.1.6.1. La clase DXInput.

Clase que encapsula la creacion de un objeto DirectInput. Su finalidad es la de servir de punto de apoyo al resto de clases que componen el módulo de entrada destinados al trabajo con el ratón y el teclado (`DXDIMouse` y `DXDIKeyboard`).

4.1.6.2. La clase DXDIMouse.

La instancia a `DXDIMouse` se encargará de gestionar el trabajo con el ratón. Los eventos a los que se atenderá serán los relativos al movimiento del mismo y la pulsación de sus botones. El movimiento podrá ser retrigido a determinadas áreas y la pulsación de los botones será controlada en cuatro niveles: botón recien pulsado, botón pulsado, botón soltado y botón sin utilizar.

4.1.6.3. La clase DXDIKeyboard.

A través de la instancia a `DXDIKeyboard` se controlará la entrada vía teclado. Todas las teclas del teclado serán consideradas como botones de tal forma que se reciban eventos ante la pulsación de las mismas sin tener en cuenta si se trabaja en modo mayúsculas o en modo minúsculas. Para tener un acceso a dicho nivel, habría que acudir a los eventos tradicionales del API de Windows.

4.2. Diseño de las clases y módulos de propósito general.

Por clases de propósito general se entenderán aquellas que se utilizarán en distintos proyectos. Cuando nos referimos a un proyecto, nos referimos al motor *Crisol*, a las utilidades *CPAKTool*, *CrisolBuilder* o al compilador *CSCompiler*. Estas clases serán, por tanto, de carácter general.

Por módulos de propósito general se entenderán aquellos espacios de nombres que contendrán definiciones o funciones compartidos entre distintos proyectos.

4.2.1. Especificación de definiciones generales.

Por especificación de definiciones generales atenderemos al trabajo con un archivo que contendrá los tipos y estructuras de carácter global. Todas las clases deberán de incluir dicho archivo para poder trabajar.

4.2.2. Módulo para el control de asertos.

Para la depuración de los proyectos se trabajará con los asertos. Los asertos permitirán evaluar una expresión de tal forma que si no es verdadera interrumpan inmediatamente la ejecución del programa ofreciendo información del lugar donde ocurrió el problema.

Para el trabajo con los asertos se dispondrá de un espacio de nombres denominado `SYSAssert` y de una macro `ASSERT` que recibirá una expresión a evaluar. En caso de que exista algún problema se interrumpirá la ejecución del programa creando un archivo de texto que contendrá el punto en donde ocurrió el conflicto. Dicho archivo de texto se denominará "`Assert.txt`". Los asertos en *Crisol* sólo estarán presentes en la versión Safe.

A parte de los asertos, *Crisol* dispondrá de un sistema para capturar excepciones lanzadas por Windows [Dawnson'99] y utilizará los servicios de VisualStudio en la localización de memoria no liberada en la finalización ("memory leaks") [Rued'01]

4.2.3. Clase CMemoryPool.

Durante la ejecución de *Crisol* y el resto de utilidades, podrán existir multitud de pequeñas, y no tan pequeñas, instancias creadas en memoria. Cuanto más llamadas `new` y `delete` para crear y destruir instancias dinámicamente, más desfragmentación de memoria habrá. Con esta clase lo que se pretenderá será crear zonas exclusivas de memoria destinadas a guardar instancias a estructuras y clases [Creighton'00] y [Ranck'00].

Al contenerse de forma estática `CMemoryPool` en una clase o estructura, se reservará una zona de memoria (que se podrá variar en tamaño) en donde existirán instancias a dichas clases o estructuras creadas con antelación siendo la operación `new` un simple proceso de obtener la siguiente instancia libre y `delete` el equivalente a liberar de uso a una instancia usada (pero sin borrarla de memoria). Así, se ahorrará un tiempo importante en las operaciones `new` y `delete` bajo demanda (ya que el sistema no tendrá que buscar espacio libre en memoria; dicho espacio libre ya fue buscado y utilizado en la llamada al constructor de `CMemoryPool`) y se evitará de forma notable la desfragmentación de memoria.

La clase tendrá capacidad de variar su tamaño dinámicamente cuando se soliciten más instancias de las que inicialmente fueron preparadas. Cuando esto ocurra, se irá generando una cadena de instancias `CMemoryPool` privada.

Esta clase será utilizada principalmente para alojar estructuras en memoria que a su vez contendrán instancias a clases importantes. No será mostrada en los diagramas UML pues su uso será excesivamente numeroso y sobrecargaría a los mismos.

4.2.4. Clase CCPAKFile.

La clase `CCPAKFile` servirá para trabajar con los archivos empaquetados (archivos CPAK de *CrisolPAcKed*). Será utilizada tanto en el subsistema de ficheros de *Crisol* como en la utilidad *CPAKTool*.

Internamente, `CCPAKFile` mantendrá un árbol en donde cada entrada será el par nombre de fichero, información asociada. La información asociada no será otra cosa que la posición en el fichero "`.pak`" donde cargar la información. Lógicamente no se mantendrán en memoria los datos de los ficheros sino que se ofrecerá una interfaz para que desde el exterior se suministren los buffers donde volcar la información requerida [Towner].

4.2.5. Clases para "parsear" los archivos de datos.

Los archivos de datos serán aquellos que usen el lenguaje sentencial comentado en la etapa de análisis. Las clases para parsear los archivos de datos serán usadas en *Crisol* para aquellos archivos que *CrisolBuilder* no realice una conversión a formato binario.

El diagrama de las clases que intervienen en el proceso de lectura de los archivos de texto con los datos de configuración, diseño, definición de perfiles, etc será el siguiente:

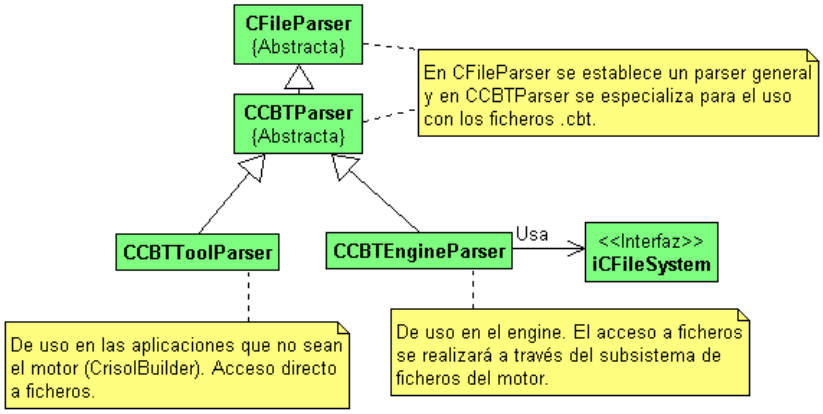


Diagrama del trabajo con los parsers.

La clase `CCBTParser` será base para cualquiera de los dos parsers existentes. El parser `CCBTToolParser` se utilizará para las aplicaciones que no sean *Crisol*, esto es, para la utilidad (tool) *CrisolBuilder* mientras que `CCBTEngineParser` será utilizado por el motor *Crisol*. La necesidad de especializar los parsers se centrará en que el acceso a los datos en el motor no será convencional pues, como se verá más adelante, *Crisol* delegará todo el trabajo a disco a su subsistema de ficheros (de ahí el enlace de uso a `iCFileSystem` por parte de `CCBTEngineParser`).

`CCBTParser` heredará de `CFileParser`. Esta clase establecerá las bases para el reconocimiento general de los archivos de datos; `CCBTParser` se limitará a añadir las peculiaridades propias del lenguaje utilizado, esto es, implementar lecturas concretas de tipos de datos que facilitarán de forma considerable el trabajo de lectura.

Todos los archivos de datos tendrán extensión "`.cbt`" si están en formato texto y, por lo tanto, son reconocibles por estas clases o bien "`.cbb`" si están en formato binario. Para que un fichero esté en formato binario antes, necesariamente, habrá tenido que ser procesado por `CrisolBuilder`.

4.2.5.1. La clase CFileParser.

Esta clase será abstracta y realizará las operaciones base para la lectura de archivos de datos de configuración. Siempre y cuando sigan un formato específico. La estructura de estos archivos deberá de ser, siempre, la siguiente:

```
NombreSeccion
 Variable0 = valor
 Variable2 = valor
 ; ...
 VariableN = valor
FinSeccion
```

Por `NombreSeccion` denotaremos la “etiqueta” que permitirá reunir una serie de variables referentes a la misma. La asignación de valores a dichas variables se basarán en sentencias del tipo:

```
NombreVariable = Valor.
```

Donde `NombreVariable` será el nombre de un identificador (los identificadores no podrán contener espacios) y `Valor` el valor que se asociará a dicha variable. Desde el exterior, se deberá de conocer el nombre de la variable con la que se quiere trabajar y el tipo de valor asociable para leerlo.

Por `FinSección` se entenderá aquel identificador que, una vez alcanzado, indicará que la sección sobre la que se está no contiene más datos. El identificador a establecer aquí no podrá ser igual a ningún nombre de sección ni de variable o de lo contrario ocurrirían efectos laterales durante el procesamiento de los archivos.

4.2.5.2. La clase CCBTParser.

La clase `CCBTParser` supondrá, más que la especialización de `CFileParser`, la extensión de la misma para que se trabaje directamente sobre la lectura de datos complejos y particulares de los archivos de datos, permitiendo así reducir líneas de código asociadas a chequeos de valores.

Esta clase seguirá siendo abstracta pues no especificará la forma en la que los archivos serán leídos.

4.2.5.3. Las clases CCBTToolParser y CCBTEngineParser.

Ambas clases se encargarán de implementar los métodos virtuales puros de la clase `CFileParser`. Estos métodos serán todos protegidos (no van a ser analizados):

```
protected:
  bool OpenFile(const std::string& szFileName);
  word ReadLineFromFile(const dword& udOffset,
                        std::string& szLine);
  dword GetFileSize(void);
  bool IsFileOpen(void);
  void CloseFile(void);
```

Desde ellos se llevará a cabo la gestión de apertura y lectura en los ficheros a procesar mientras que `CCBTToolParser` accederá directamente usando un objeto `fstream` y `CCBTEngineParser` utilizará el subsistema de ficheros de *Crisol* (que se estudiará más adelante) para obtener un handle al archivo y realizar las operaciones con el mismo.

4.2.6. La clase CRulesDataBase para el mantenimiento de las reglas del juego.

La clase `CRulesDataBase` será utilizada tanto en *Crisol* como en *CrisolBuilder* para la carga y gestión del fichero de reglas. Para poder hacer lectura del fichero de reglas se utilizará `CCBTToolParser`, si la ejecución no es en Crisol, y `CCBTEngineParser`, si la ejecución sí es en el motor. En cualquier caso, el trabajo se realizará a través de `CCBTParser`.

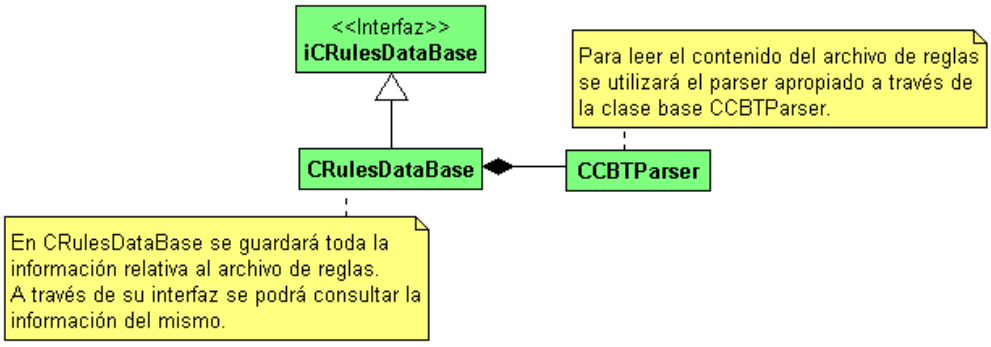


Diagrama de CRulesDataBase.

El mantenimiento de la información se realizará utilizando de forma intensiva árboles que asociarán a los identificadores de tipo de cada entidad la información relativa a los mismos.

Esta clase será fundamental en el motor para poder interpretar las distintas acciones en base a las reglas y para *CrisolBuilder* para verificar que todos los archivos de definición de datos sean coherentes.

4.3. Diseño de CrisolEngine.

Crisol estará dividido en diferentes subsistemas, cada uno de los cuales tendrá una responsabilidad concreta. Todos los subsistemas heredarán de una interfaz que será utilizada por sus clientes [Iqbal'00]. La clase `CCRISOLEngine` será el nucleo de la aplicación *Crisol*, pues actuará de contenedor de todos los subsistemas del motor y filtrará el proceso de inicialización y finalización del mismo.

El acceso a los interfaces de los distintos subsistemas se realizará utilizando una serie de funciones amigas de `CCRISOLEngine`, las cuales se hallarán definidas en un espacio de nombres denominado `SYSEngine` [Iqbal'00]. En ese mismo espacio de nombres también se declararán el resto de funciones de uso general así como las que permitan el acceso a clases de apoyo (`CLogger`, `CMathUtil`, etc).

Como los subsistemas estarán representados por clases que deberán de tener una única instancia, se utilizará el patrón de diseño Singlenton (la clase `CCRISOLEngine` también será Singlenton).

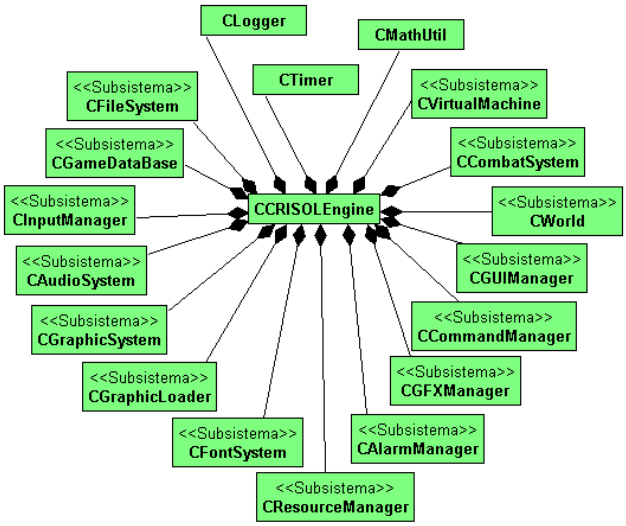


Diagrama general de CCRISOLEngine.

A parte de contener a los distintos subsistemas y clases de apoyo, `CCRISOLEngine` llevará a cabo otras acciones:

- **Inicialización y finalización correcta de los distintos subsistemas**. Existirá un archivo de configuración llamado "`CrisolEngine.ini`" en el que se definirán los parámetros para la inicialización de los distintos subsistemas. La instancia `CCRISOLEngine` se encargará de recoger del archivo la configuración de cada uno, pasando los valores oportunos para su inicialización.

- **La actualización del motor**. Al ser *Crisol* una aplicación de Windows, desde el exterior existirá un bucle que se encargará de realizar sucesivas llamadas a un método de actualización de la clase `CCRISOLEngine`. Éste método, a su vez, trasladará dicha petición a una instancia que representará el estado de juego actual. Habrá un total de cuatro clases distintas para representar los estados de juego. Así, la clase `CGSMainMenu` representará el estado de juego inicial, `CGSCreateCharacter` el estado de juego de creación del personaje, `CGSGameLoading` la de carga de un juego y `CGSInGame` la de estar jugando. Todas ellas heredará de una clase denominada `CGameState` y serán clases amigas a `CCRISOLEngine`. Al ser clases amigas, su método `Update` permitirá coordinar los diferentes componentes del motor para su actualización.

  La actualización del motor se dividirá siempre en tres partes. Por un lado la actualización de IA (Inteligencia Artificial), por otro la gráfica y por último la de sonido. La actualización de IA vendrá determinada por un valor temporal, es decir, se efectuará únicamente cuando hayan transcurrido un determinado número de milisegundos después de la última actualización. En esta actualización se llevarán a cabo las tareas destinadas a obtener información de los dispositivos de entrada, comprobar si hay situación de combate y ejecutar los scripts que pudieran estar solicitados (estas dos últimas comprobaciones, sólo se realizarán cuando se esté en estado de juego). En las actualizaciónes gráficas se dibujará la escena en pantalla y en la de sonido se hará sonar aquellas solicitudes pendientes. Estas dos últimas solicitudes se ejecutarán siempre, es decir, lo más rápido posible.

  Cabe añadir al respecto de las actualizaciónes, que el sistema de comandos estará a medio camino de lo que entendemos por actualización de IA y lo que entendemos por actualización gráfica. En el sistema de comandos siempre se ejecutarán procesos visuales, por lo que deberán de ser atendidos siempre, sin esperar ningún intervalo de tiempo. Sin embargo, deberán de actualizarse "suavemente" (sin saltos bruscos). Para ello, se llevará el control de un valor temporal al que se denominará delta de tiempo. Dicho valor delta, será una medición que indique la cantidad de milisegundos que han transcurrido desde la

última medición. Gracias al valor delta, que estará siempre medido en base a un segundo, se podrán ajustar las actualizaciones lógicas de los comandos [Arévalo'01] y [Olse'00].

- **Gestión del cambio de tareas**. Cuando desde Windows se cambie de *Crisol* a otra aplicación y de otra aplicación a *Crisol*, el motor será capaz de pausarse para posteriormente activarse y restaurar gráficos y sonidos de superficies perdidas (tal y como obliga a hacer la librería DirectX).

4.3.1. Las clases y funciones de apoyo públicas.

Por clases de apoyo entenderemos el trabajo que realizarán `CLogger`, `CTimer` y `CMathUtil`. Estas clases podrán ser utilizadas desde cualquier punto del motor y su interés radicará en ofrecer una serie de prestaciones de utilidad general. Al poderse utilizar desde cualquier punto del motor, todas heredarán de una interfaz propia que será lo que se ofrezca a los clientes que deseen utilizarlas.

4.3.1.1. La clase CLogger como logger del sistema.

El logger del sistema consistirá en el mantenimiento de un archivo al cual se le enviará información con el fin de que sea más fácil poder crear una traza, ver resultados internos o solucionar un error en *Crisol*. Para implementar el logger se utilizará una clase denominada `CLogger` con su propia interfaz para clientes. El diagrama de la clase será muy sencillo:

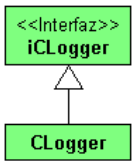


Diagrama de CLogger.

4.3.1.2. La clase CTimer como temporizador del sistema.

El motor utilizará un temporizador con fines muy importantes. Por un lado, servirá para controlar los ciclos de ejecución y conocer el momento en que deberá de actualizar los valores de entrada o ejecutar los scripts asociados a eventos. Por otro, servirá para poder instalar alarmas temporales que gestionen determinadas notificaciones.

Para implementar el temporizador, se utilizar a la clase `CTimer` que tendrá su propia interfaz pues podrá ser utilizado desde subsistema de *Crisol*. El diagrama será muy sencillo:

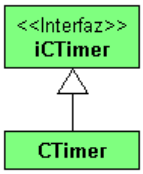


Diagrama de CTimer.

Durante la inicialización, `CTimer` será capaz de comprobar qué temporizadores están disponibles en el sistema y escoger aquel que ofrezca una mayor resolución de medición. El mejor temporizador será siempre el denominado Performance. El problema está en que este temporizador no se halla presente en los equipos más antiguos. En estos casos, la clase se adaptará para hacer uso del temporizador multimedia proporcionado por Windows que posee menos resolución pero se halla disponible siempre.

4.3.1.3. La clase CMathUtil para el trabajo con utilidades matemáticas.

Habrá un conjunto de operaciones matemáticas peculiares del motor y otras de carácter más general. En cualquier caso, todas ellas se localizarán en una misma clase de apoyo para que

no se tengan que reescribir. Esta clase de apoyo se denominará `CMathUtil` y tendrá su propia interfaz. El diagrama será, como en el resto de clases de utilidad, muy sencillo:

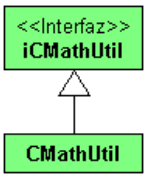


Diagrama de CMathUtil.

4.3.2. El subsistema de ficheros.

El trabajo con ficheros en *Crisol* será completamente filtrado a través de su subsistema de ficheros. En otras palabras, para que desde el interior del motor se pueda trabajar con un fichero antes se deberá de solicitar permiso a este subsistema. Si el permiso es concedido, el subsistema retornará un handle. Mediante ese handle el cliente del fichero podrá realizar las operaciones básicas de lectura y escritura de información. El diagrama del subsistema de ficheros será:

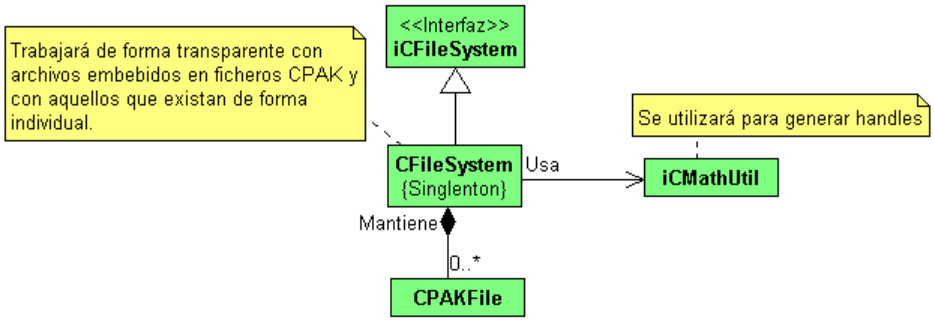


Diagrama del subsistema de ficheros.

El subsistema de ficheros trabajará de forma transparente con respecto a si un fichero se halla de forma individual en el sistema o bien está en un archivo CPAK. Para que el subsistema de ficheros pueda trabajar con los archivos CPAK antes éstos se deberán de haber registrado en él.

Desde el punto de vista de las exigencias de funcionamiento del motor, deberá de existir un único archivo CPAK en donde el diseñador pueda insertar aquellos archivos que considere oportunos (para gestionar este único archivo se utilizará la herramienta *CPAKTool*). Desde el punto de vista del funcionamiento general, el subsistema soportará el trabajo con cero o más archivos CPAK.

Siempre que se desee obtener un handle a un fichero, `CFileSystem` comprobará primero si éste existe de forma individual y solo después de comprobar que no existe, iterará entre los distintos archivos CPAK registrados para comprobar si alguno de ellos lo posee.

El uso de la clase `CMathUtil` se centra únicamente en la generación de handles (que no es más que un valor numéric). Con `CMathUtil` éstos se generarán de forma aleatoria.

4.3.2.1. La clase CFileSystem.

En `CFileSystem` se implementarán todos los métodos definidos en su interfaz. Será mantenida desde `CCRISOLEngine` y los métodos que no se hallen en su interfaz estarán dirigidos a la inicialización, al mantenimiento de la instancia Singlenton y al registro del ficheros CPAK. Desde `CCRISOLEngine` se registrará el único fichero CPAK con el que funcionará el motor. Dicho fichero podrá, o no, estar vacío. En cualquier caso, deberá de existir y el diseñador podrá gestionarlo desde la utilidad *CPAKTool*.

4.3.3. La base de datos del juego.

Este subsistema actuará de Facade para las diferentes clases que mantengan realmente los datos del juego. En primer lugar mantendrá una instancia a `CRulesDataBase` (que devolverá a través de su interfaz `iCRulesDataBase`) y por otro lado, mantendrá la práctica totalidad de los archivos de datos "`.cbt`" y "`.cbb`".

Los archivos "`.cbt`" serán mantenidos usando, para cada uno de ellos, una instancia a un parser `CCBTEngineParser`. Dicho parser deberá de ser solicitado por los clientes indicando la sección (inicial) sobre la que se querrá trabajar. Con esa petición, `CGameDataBase` se encargará de comprobar si la sección existe y devolver el parser.

Los archivos "`.cbb`" serán mantenidos guardando el handle que `CFileSystem` devuelva a `CGameDataBase` cuando el subsistema solicite abrirlos para lectura. Estos mismos handles serán pasados a los clientes que deseen trabajar con los archivos de datos "`.cbb`". El diagrama será:

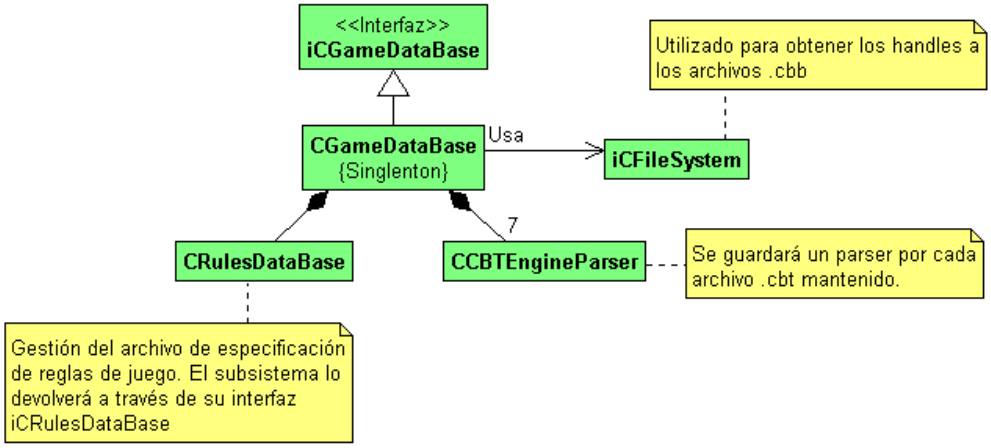


Diagrama de la base de datos del juego.

Los archivos de datos "`.cbt`" mantenidos harán referencia a:

- El archivo de inicialización del motor (este tendrá extensión "`.ini`" pero por lo demás, seguirá las mismas normas que cualquier otro archivo "`.cbt`").
- El archivo de texto estático.
- El archivo de perfiles de jugador.
- El archivo de configuración de interfaces.
- El archivo de perfiles GFX.
- El archivo de perfiles de presentación.
- El archivo de especificación de scripts a eventos globales.

Los archivos de datos "`.cbb`" serán:
- El de plantillas de animación.
- El de perfiles Floor (base de las celdas).
- El de perfiles Wall (paredes).
- El de perfiles SceneObject (objetos de escenario).
- El de perfiles Item (ítems).
- El de perfiles Criatures (criaturas).
- El de perfiles Roof (techos).

Las áreas pese a guardarse en formato "`.cbb`" no serán mantenidas en este subsistema.

4.3.3.1. La clase CGameDataBase.

En `CGameDataBase` se implementarán todos los métodos definidos en la interfaz `iCGameDataBase`. La clase será mantenida desde `CCRISOLEngine`.

Los métodos públicos de `CGameDataBase` no definidos en la interfaz estarán orientados a la inicialización y tratamiento de la instancia Singlenton.

4.3.4. El manager de comandos.

En *Crisol* los comandos se utilizarán para realizar acciones visuales que puedan necesitar más de un ciclo de ejecución. Estas acciones podrán ser la actualización de animaciones, el movimiento de una criatura, el scroll en el motor isométrico, etc.

Cada vez que se desee ejecutar un comando, el cliente que lo posea deberá de utilizar el manager de comandos que estará representado por la clase `CCommandManager`. Este subsistema se encargará de tomar un comando e insertarlo en una cola de prioridad. Dicha prioridad estará regida por el valor temporal con el que el comando fue insertado.

En cada actualización se tomará el tiempo actual y se comenzarán a procesar los comandos que se hallen al comienzo de la cola y así hasta que no quede ninguno o bien hasta que el valor temporal del comando a ejecutar supere al valor temporal leído al comienzo del proceso. Siempre que un comando se halle pausado o bien cuando después de ejecutado no haya finalizado, se volverá a insertar en la cola de comandos pero con el valor temporal (prioridad) actualizado. El diagrama del manager será el siguiente:

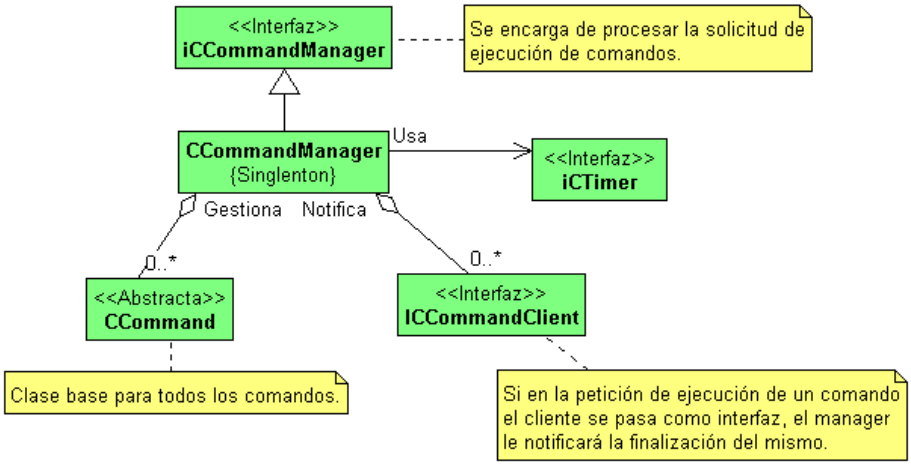


Diagrama del manager de comandos.

Una de las particularidades de `CCommandManager` será la de poder notificar al cliente que insertó el comando para su ejecución. Si el cliente se registró a través de la interfaz `iCCommandClient` entonces `CCommandManager` ejecutará el método de notificación asociado a dicho interfaz.

Los comandos seguirán el patrón Command, esto es, todos heredarán de una clase base que poseerá un método `Execute` a definir por ellos.

4.3.4.1. La clase CCommandManager.

En esta clase se encapsulará todo el trabajo con la gestión de los comandos y se implementarán los métodos definidos en su interfaz. Al ser Singlenton, existirán dos métodos exclusivos para trabajar con la única instancia.

La clase `CCommandManager` será mantenida desde `CCRISOLEngine`. Ésta se encargará, a su vez, de llamar al método `ExecuteCommands` en cada ciclo de ejecución, pasando al mismo el delta de tiempo transcurrido (necesario para la ejecución de los comandos).

4.3.4.2. La interfaz iCCommandClient.

Todas aquellas clases que quieran ser notificadas sobre la ejecución de un comando deberán de heredar de `iCCommandClient`. De esta forma, cuando se desee registrar en `CCommandManager` un comando para su ejecución, las clases que hereden de `iCCommandClient` podrán pasarse para que el manager les notifique la finalización del comando que les acompaña.

La notificación ocurrirá a través del único método de la interfaz que toda clase que herede deberá de implementar. Dicho método será `EndCmdNotify` (consultar el apéndice sobre los métodos de las clases e interfaces).

4.3.5. El cargador de gráficos.

El cargador de gráficos será considerado como un subsistema secundario. Su único objetivo será el de ofrecer una localización para las operaciones de carga de gráficos y volcado de los mismos a superficies de DirectX. En la versión actual de *Crisol*, este subsistema únicamente se encargará de trabajar con ficheros de formato tga, volcándolos luego a superficies `DXDDSurfaceTexture` (ver el apartado sobre la librería de DirectX). El diagrama:

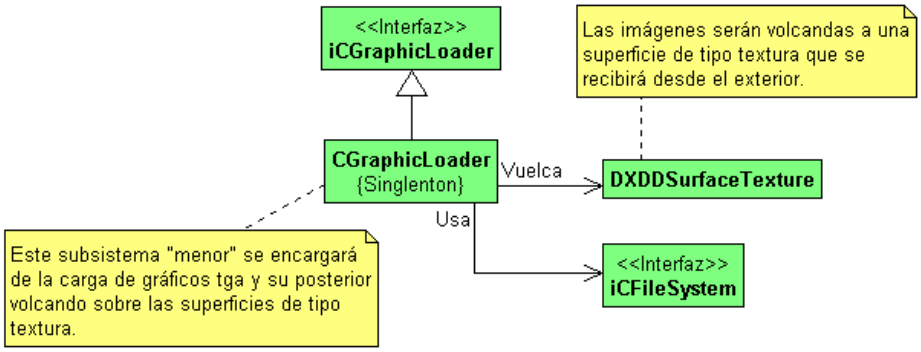


Diagrama del cargador de gráfico.

El cargador de gráficos será muy utilizado por el subsistema de recursos `CResourceManager` en el proceso de registro de imágenes.

4.3.5.1. La clase CGraphicLoader.

La clase `CGraphicLoader` será la que lleve el peso del subsistema, implementando los métodos definidos en la interfaz `iCGraphicLoader`. El proceso de manipulación de las imágenes aquí cargadas estará dividido en tres partes. Por un lado la carga de un gráfico, por otro la obtención de información y, por último, las operaciones sobre ese fichero cargado. Las dos últimas etapas deberán de ser independientes del formato gráfico cargado.

Todas las imágenes cargadas serán mantenidas en memoria hasta que no se descarguen o bien se cargue una imagen distinta a la actual. Interiormente, para evitar que la memoria quede muy desfragmentada, se mantendrá un espacio constante igual al tamaño de la imagen más grande cargada hasta el momento.

Los métodos básicos en `CGraphicLoader` comprenderán la inicialización y finalización así como el mantenimiento de la instancia Singlenton. La clase será mantenida en `CCRISOLEngine`.

4.3.6. El gestor de recursos.

El manejador de recursos, `CResourceManager`, será uno de los subsistemas o managers más importantes de *Crisol*. Gracias a él se realizará una utilización racional de la memoria ocupada

por los recursos de sonido WAV, texturas y configuración de las plantillas de animación [Clark'96]. El diagrama será el siguiente:

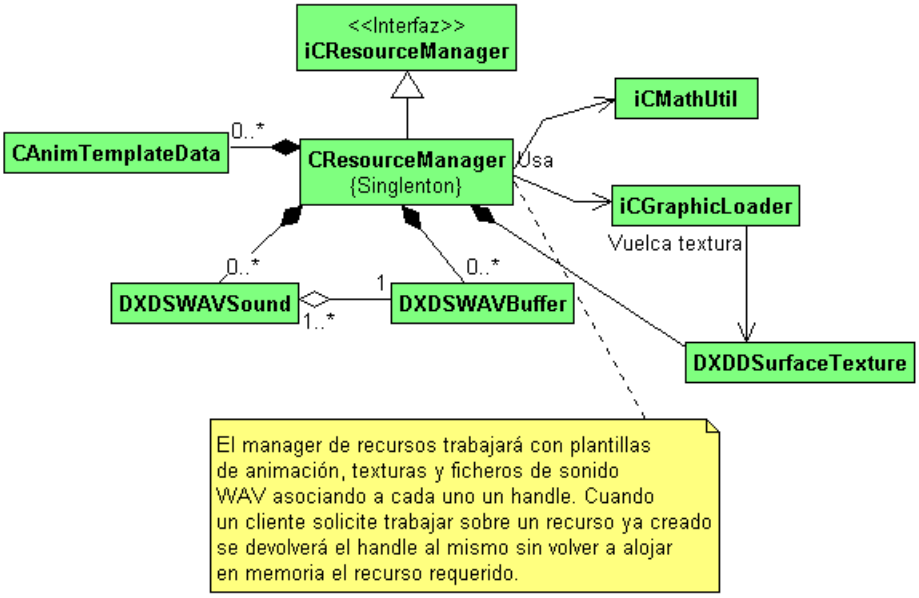


Diagrama del manager de recursos.

Todos los recursos serán creados en el manager y accedidos a través de un handle. Cuando se desee crear un recurso que ya exista, el manager devolverá el handle al recurso ya existente, incrementando un contador de solicitudes al mismo. El contador se decrementará a medida que los clientes vayan liberando el uso de los recursos previamente solicitados. Cuando el contador llegue a cero el recurso quedará finalmente liberado de memoria.

Las texturas (`DXDDSurfaceTexture`) serán siempre creadas en el servidor de recursos y cargadas usando `iCGraphicLoader`.

En el caso de los sonidos WAV, siempre que se quiera cargar un sonido (`DXDSWAVSound`) se llevará a memoria, de forma invisible para el cliente, el buffer asociado (`DXDSWAVBuffer`). Naturalmente, si para el sonido a cargar ya existe un buffer en memoria dicho buffer se reaprovechará.

En el servidor de recursos se guardarán también las instancias a `CAnimTemplateData` mientras exista alguna animación `CAnimTemplate` que la utilice. Como se verá más adelante, toda plantilla de animación tendrá dos representaciones: la parte lógica, encarnada por la clase `CAnimTemplate`, y la configuración de la misma, representada por `CAnimTemplateData`.

4.3.6.1. La clase CResourceManager.

En la clase `CResourceManager` se encapsulará el verdadero trabajo del gestor de recursos. Será mantenida por `CCRISOLEngine` e implementará todos los métodos definidos en su interfaz. Cada recurso dispondrá de una estructura de datos árbol para el mantenimiento de la correspondencia handle - recurso. En el caso de las plantillas de animación no se utilizarán handles sino que las clases `CAnimTemplate` pasarán al servidor el nombre de la plantilla para que éste devuelva la interfaz a la instancia `CAnimTemplateData` con la información de configuración.

Otra de las responsabilidades importantese del servidor será la de restaurar recursos cuando *Crisol* pierda el foco de entrada (recordemos que al trabajar en Windows, el usuario podrá desear cambiar de aplicación). Realmente, el servidor recuperará únicamente los recursos de sonido ya que los recursos gráficos (las superficies) serán recuperadas desde `CGraphicSystem`.

Todos los nodos con la información asociada a los recursos harán uso intensivo de `CMemoryPool` para intentar minimizar la desfragmentación de memoria que seguro se producirá. El trabajo intensivo del servidor se concentrará cuando exista, por un lado, un nuevo estado en el motor y un proceso de carga de áreas de juego y, por otro, en la obtención sistemática de texturas por parte del módulo `CGraphicSystem` para dibujarlas en pantalla. Esto último hará especialmente importante el uso de árboles de búsqueda equilibrados.

Mediante `iCMathUtil` el servidor de recursos creará los handles. Los handles no serán más que valores numéricos de ahí que se utilice `iCMathUtil` para la obtención de valores aleatorios (y que no estén siendo usados).

4.3.7. El manager de alarmas.

El manager de alarmas se utilizará para notificar el paso del tiempo y la finalización en la reproducción de un fichero WAV. Cuando se produzca la condición de notificación el cliente que se hubiera registrado recibirá el aviso. He aquí el diagrama del subsistema:

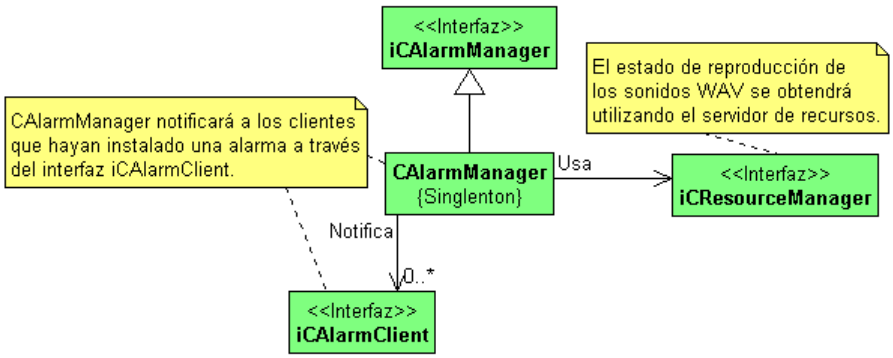


Diagrama del manager de alarmas.

Una clase podrá registrarse en `CAlarmManager` para recibir una notificación siempre y cuando herede la interfaz `iCAlarmClient` e implemente el método de notificación que `CAlarmManager` ejecutará una vez se cumpla la condición de notificación.

Para la comprobación de la finalización de un fichero WAV, `CAlarmManager` usará el handle al mismo. Con él obtendrá la información necesaria usando el servidor de recursos.

4.3.7.1. La clase CAlarmManager.

La clase `CAlarmManager` será el núcleo del subsistema. Implementará todos los métodos definidos en su interfaz `iCAlarmManager` y será mantenida desde `CCRISOLEngine`.

4.3.7.2. La interfaz iCAlarmClient.

Toda clase que registre una alarma, deberá de heredar de esta interfaz pues `CAlarmManager` deberá de tener alguna forma de notificar a sus clientes la finalización del suceso temporal o de reproducción de un fichero WAV.

La interfaz `iCAlarmClient` dipondrá de un sencillo método que los clientes deberán de implementar. Dicho método ser llamará `AlarmNotify` (consultar el apéndice sobre métodos públicos asociados a clases e interfaces).

4.3.8. El subsistema gráfico.

El subsistema gráfico se encargará de recoger las peticiones de dibujado, almacenarlas y, llegado el momento, proceder a realizar el dibujado de las mismas. A la hora de dibujar un elemento este siempre deberá de heredar de una clase `CDrawable` la cual, poseerá su

propia interfaz `iCDrawable`. Mediante dicha interfaz se recogerán los elementos a dibujar y se obtendrá la configuración de los mismos.

El subsistema utilizará un esquema de dibujado basado en zonas. Cada zona poseerá sus propios planos. Por zona entenderemos una sección de dibujado y por plano los diferentes conjuntos en donde organizar los elementos a dibujar en una zona. Primero se dibujarán los elementos que se hallen en los planos de la primer zona, luego ocurrirá lo mismo con la segunda zona y así sucesivamente.

Existirá una zona para las imágenes de fondo, otra para una escena de juego, otra para las interfaces gráficas, otra para los FX y otra para el cursor de interfaz. De esta forma, siempre ocurrirá que el cursor de interfaz se dibujará por encima de cualquier otro elemento que se vea en pantalla. El número de planos variará dependiendo de la zona y seguirán el mismo concepto, esto es, primero se dibujarán los elementos del primer plano luego los de la siguiente y así sucesivamente.

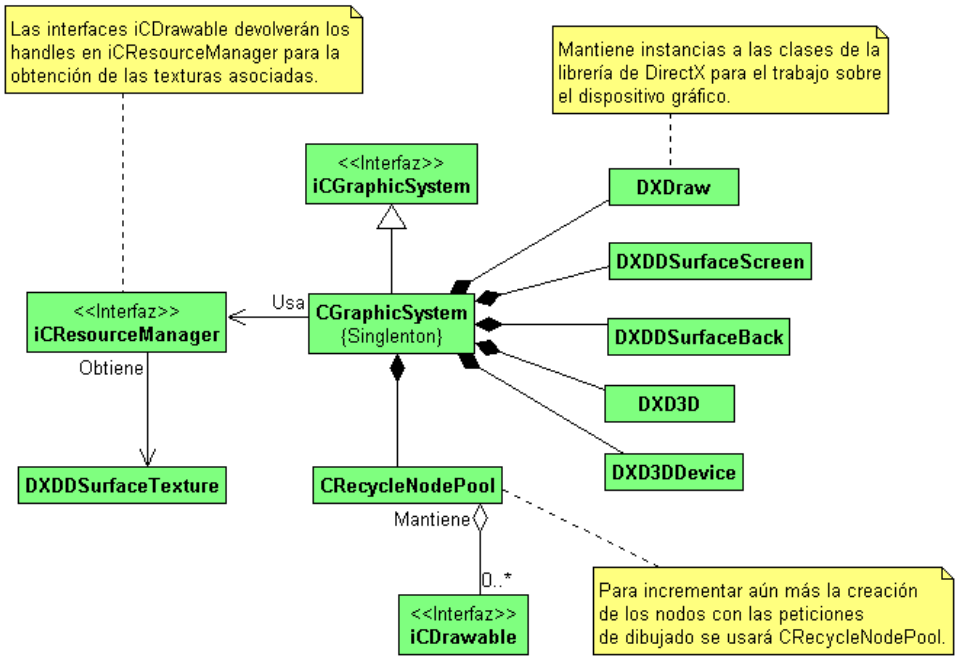


Diagrama del subsistema gráfico.

Desde el subsistema gráfico se controlará, también, el efecto de luminosidad. Para ello, el subsistema de dibujado aplicará, a todos los elementos dibujables que posean un atributo de luz asociado, el llamado factor de oscuridad. El factor de oscuridad no será más que un byte que indicará cómo de oscuro se verán los objetos en pantalla. Si un objeto es dibujado sin su atributo de luz asociado, dicho valor no será tenido en cuenta. En caso contrario, el factor de oscuridad aplicado a un elemento dibujable, se verá contrarestado por el valor luminoso que se pasó junto a él.

A continuación se comentará la clase `CGraphicSystem` y su interfaz para pasar, posteriormente, a analizar la jerarquía de clases dibujables por el subsistema.

4.3.8.1. La clase CGraphicSystem.

En `CGraphicSystem` se mantendrán todas las instancias a las clases de DirectX encargadas de trabajar con los gráficos (`DXDraw`, `DXDDSurfaceScreen`, `DXDDSurfaceBack`, `DXD3D` y `DXD3Ddevice`). Usará `iCResourceManager` para obtener los objetos `DXDDSurfaceTexture` que representarán las imágenes a dibujar. Debemos de tener en cuenta que los objetos `iCDrawable` poseerán un método para obtener el handle de la imagen que tienen asociada. Ese handle se utilizará a su vez para que `iCResourceManager` devuelva la textura que hay que dibujar.

CGraphicSystem implementará mecanismos de optimización para el dibujado. Así, además de utilizar CMemoryPool para agilizar el trabajo con new y delete, usará la clase CRecycleNodePool para reducir completamente el uso de new y delete. Será necesario acudir a todos los mecanismos de optimización pues la actualización de pantalla será el aspecto que más ciclos de ejecución consuma.

Otras medidas serán las relativas a la ordenación, siempre que sea posible, de los dibujados por tipo de textura. Resulta muy caro, en términos de rendimiento, ordenar a la tarjeta gráfica que cambie de textura así como de configuración de dibujado por ello, siempre que se realice una petición de dibujado, CGraphicSystem ordenará las mismas de tal forma que los cambios en la configuración del subsistema gráfico sean los menos posibles.

4.3.8.2. Objetos dibujables en CGraphicSystem.

Todos los objetos que se van a comentar a continuación heredarán de CDrawable (que a su vez poseerá su propia interfaz iCDrawable) de tal forma que sea posible el pasarse a CGraphicSystem para que éste los dibuje obteniendo su configuración de dibujado además del handle al objeto DXDDSurfaceTexture guardado en el servidor de recursos.

Además de los objetos dibujables que hereden de CDrawable, habrá otros que contendrán a éstos y que también serán dibujables sin necesidad de tener que heredar de CDrawable. Este será el caso de CBitmap, CShadow o CFXBack que mantendrán objetos dibujables base. Estas clases también serán comentadas en este apartado.

4.3.8.2.1. La clase CDrawable.

La clase CDrawable, tal y como ya se ha comentado, será heredara por todos aquellos objetos que deseen dibujarse. Será una clase abstracta pues el objetivo principal se centrará en que sus derivadas implementen los métodos concretos para la obtención de información. El diagrama de esta clase será muy sencillo:

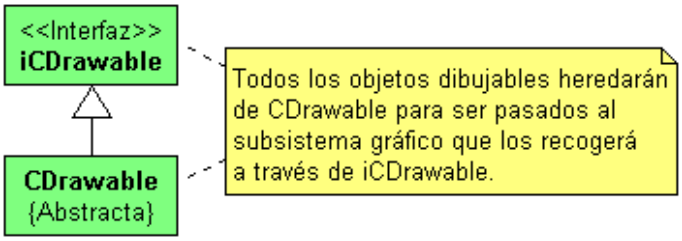


Diagrama de CDrawable.

En el interior de CDrawable se mantendrán los flags de dibujado, un array para el mantenimiento de los colores RGB en cada vértice y otro para hacer lo propio con los valores alpha. La particularidad radicará en que las estructuras de información que se almacenarán en los arrays serán mantenidos desde el exterior y no en el interior de CDrawable. Esto permitirá un ahorro considerable en recursos y una gran versatilidad para crear efectos sobre conjuntos de elementos dibujables. Por defecto dichas estructuras estarán vacías por lo que los objetos se dibujarán en su forma original.

4.3.8.2.2. La clase CCharFont.

Esta clase representará un carácter asociado a una fuente de texto. Tal y como se verá en el apartado destinado al subsistema de fuentes, una fuente de texto estará representada por una instancia a una clase CFont. Dicha instancia mantendrá una textura de 256x256 con todos los caracteres asociados [Fötch'00]. La clase CCharFont servirá para representar a uno de esos caracteres escribibles por lo que en su inicialización se deberá de pasar la fuente a la que estén asociado. A la hora de dibujarse, el subsistema gráfico tomará el handle a la textura de

la fuente a la que está asociado el carácter más luego otra con la región en donde se hallará el gráfico del carácter representado. El diagrama será muy sencillo.

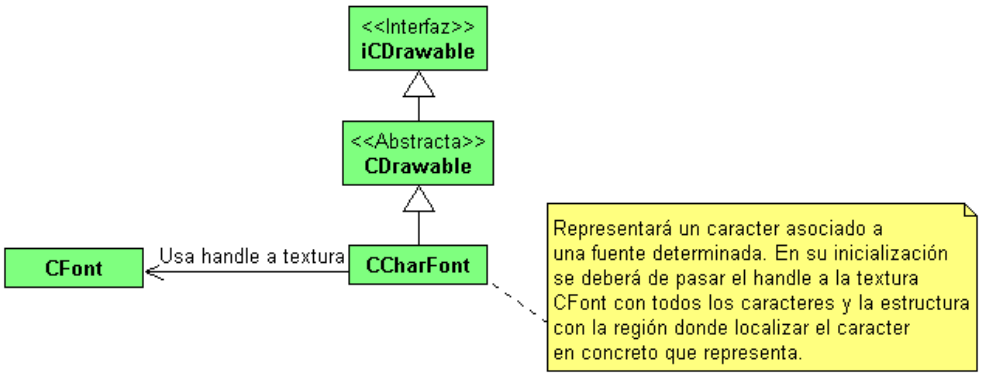


Diagrama de CCharFont.

No tendrá una relación directa con la instancia `CFont` ya que se pasará, el handle a la imagen mantenida por ésta.

4.3.8.2.3. La clase CImage.

Esta clase encapsulará un handle a una textura (a una imagen). El principal uso se encontrará en ser utilizada para la clase `CBitmap` la cual actuará de contenedor de múltiples clases `CImage` para representar una pantalla de dimensiones arbitrarias. El diagrama será el siguiente:

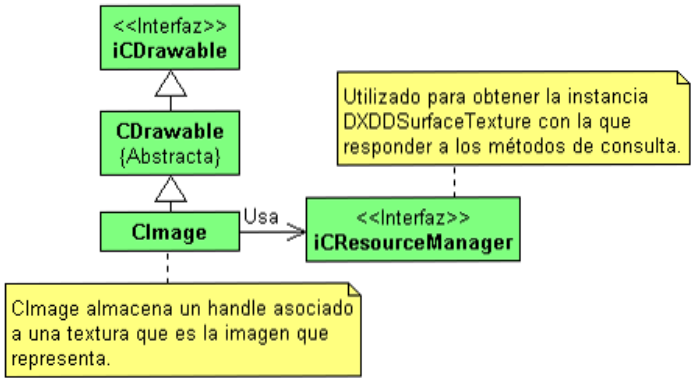


Diagrama de CImage.

Desde las instancias `CFont` y `CShadow` también se utilizarán objetos `CImage`. En el primer caso para contener la textura con todos los caracteres que representarán a la fuente y en el segundo para mantener la imagen con la sombra a asociar a una determinada entidad.

4.3.8.2.4. La clase CBitmap.

La clase `CBitmap` no heredará de `CDrawable` pero será un contenedor de `CImage`. Su objetivo será el de mantener una imagen cuyas dimensiones haga imposible (o posible) tenerla en una sola textura. Esta clase se utilizará, por tanto, para representar imágenes de fondo que ocupen toda la pantalla o parte de ella. La técnica consistirá en dividir la imagen en áreas (instancias a clases `CImage`) de 32x32 y cargar sobre ellas la porción que toque. El diagrama será el siguiente:

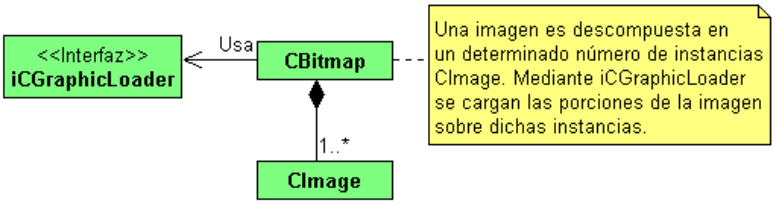


Diagrama de CBitmap.

Naturalmente, todas las acciones de configuración que se realicen sobre la instancia `CBitmap` repercutirá sobre las instancias `CImage` que posea. Del mismo modo, la operación de dibujado no será otra cosa que un recorrido sobre las instancias `CImage` para enviar a las mismas al subsistema gráfico.

La inicialización de la instancia se realizará a partir de un estructura que se declarará como pública en el interior de la clase. Esto evitará tener que pasar muchos parámetros a la instancia.

4.3.8.3.5. La clase CSprite y el control de la animación.

Esta clase servirá para representar elementos gráficos capaces de tener animación asociada y movimiento, esto es, estará completamente orientado a la representación de entidades en el universo de juego aunque también se utilizará para la representación de botones y animaciones estáticas (que no cambien de posición). `CSprite` será, por tanto, la clase más importante para la representación de objetos dibujables en *Crisol*.

La clase `CSprite` no encapsulará la lógica relativa a la animación sino que se basará en el uso de una clase de apoyo llamada `CAnimTemplate` que será la que verdaderamente lleve el peso relativo al trabajo con la animación. `CSprite` contendrá una instancia a `CAnimTemplate` y devolverá al exterior la interfaz a la misma para que se pueda actuar sobre ella. A su vez, `CAnimTemplate` estará asociada a una instancia `CAnimTemplateData` que contendrá sus datos de configuración. Como podrán existir varias instancias a una misma plantilla de animación, los objetos `CAnimTemplateData` serán almacenados en el servidor de recursos `CResourceManager`.

El diagrama completo de `CSprite` será el siguiente:

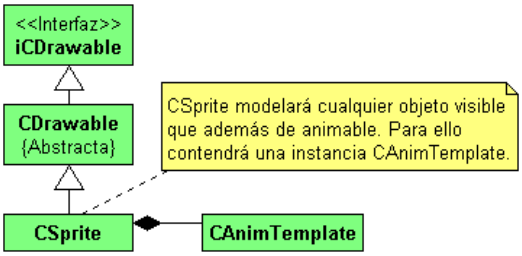


Diagrama de CSprite.

El trabajo interno en `CAnimTemplateData` se basará en el mantenimiento de una tabla de estados de animación. Cada estado contendrá, a su vez, otra entrada para las diferentes orientaciones. Las orientaciones serán necesarias porque al tratarse de un motor isométrico, las distintas entidades podrán manejar hasta 8 orientaciones para una misma animación. En cada una de dichas orientaciones se mantendrá una lista de handles a recursos de tipo textura que serán los frames de la animación. En la clase `CAnimTemplate` se trabajará con toda esa información como base de datos pero además se mantendrán los estados propios de la animación, su frame actual, etc.

El diagrama para las clases `CAnimTemplate` y `CAnimTemplateData` serán los siguientes:

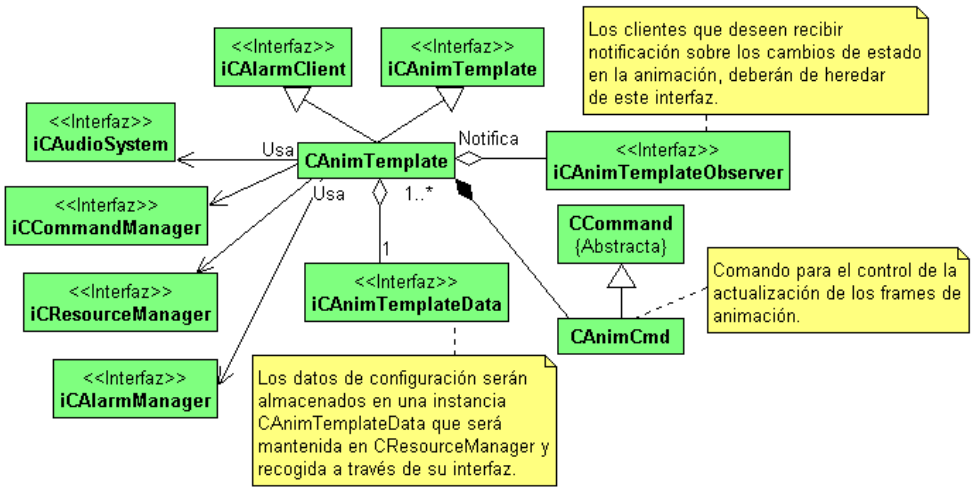


Diagrama de CAnimTemplate y CAnimTemplateData.

Las instancias `CAnimTemplate` podrán tener observadores que reciban notificación cuando la plantilla de animación haga un cambio del estado de animación o produzca un ciclo completo. Todas las clases que quieran ser observadores deberán de heredar de la interfaz `iCAnimTemplateObserver`. El uso de observadores para la animación será muy importante a la hora de coordinar las entidades representables en pantalla cuando se les ordene realizar una acción una vez que hayan completado la animación que la simula.

`CAnimTemplate` trabajará con una instancia `CAnimTemplateData` a través de su interfaz sin embargo, no mostrará al exterior dicha instancia sino que actuará de Facade ofreciendo una interfaz unificada para el acceso a la información brindada por `iCAnimTemplateData` y la suya propia.

4.3.8.2.6. La clase CQuadForm.

Esta clase representará un elemento dibujable que, a diferencia de los anteriores, no tendrá asociado una textura. En su lugar, lo que se dibujará en pantalla será un rectángulo. Dicho rectángulo podrá tener los mismos efectos que cualquier otro objeto dibujable, esto es, color RGB en cada uno de los vértices y valor de transparencia. El diagrama será muy sencillo:

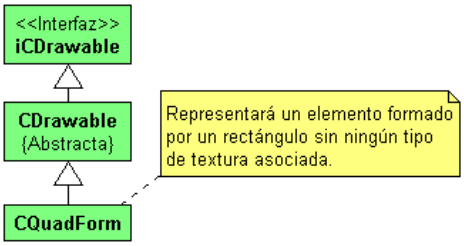


Diagrama de CQuadForm.

Esta clase será utilizara para implementar componentes de interfaz destinados a simular la selección de elementos gráficos, las cortinas de cine asociadas a las cut – scene y para representar los efectos de difuminado en pantalla (usando una clase `CQuadForm` con dimensiones equivalentes a las de la pantalla y controlando su nivel de transparencia).

4.3.8.2.7. La clase CShadow

Esta clase se utilizará para representar la sombra de las entidades. Estará formada por una instancia `CImage`. Como se verá en el apartado destinado al universo de juego, las instancias `CWorldEntity` crearán un objeto `CShadow` si la entidad que representan posee sombra asociada. El diagrama será muy sencillo:

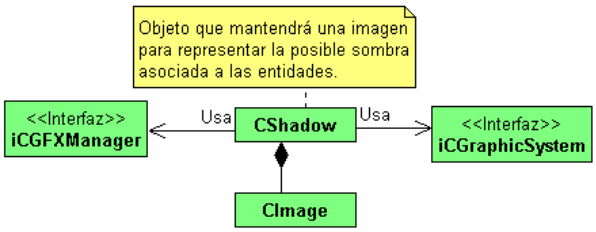


Diagrama de CShadow.

Internamente, la instancia `CShadow` tomará un valor de transparencia del subsistema de efectos especiales. Dicho valor de transparencia cambiará automáticamente de valor según pasen las horas de juego y según haya definido el diseñador en el archivo de configuración de FX. La idea será que, dependiendo de la hora, la intensidad de la sombra varíe dinámicamente.

4.3.8.2.8. La clase CFXBack

Representará una clase que esté compuesta por un objeto `CQuad` configurable desde el exterior en dimensiones, color y valor de transparencia. El uso de la clase irá dirigido a construir fondos para los textos, de tal forma que éstos queden mucho mejor resaltados y no sean confusos con el gráfico que tengan por debajo. Un ejemplo de uso de esta clase podrá encontrarse en la forma en la que el texto es mostrado durante el proceso de habla de una entidad. El diagrama será muy sencillo:

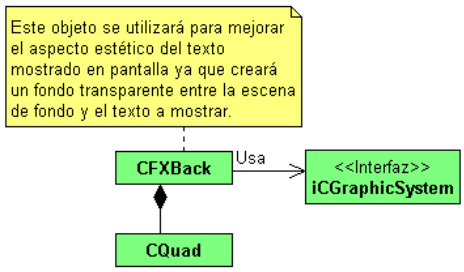


Diagrama de CFXBack.

4.3.9. El manager de entrada.

Este subsistema se encargará de gestionar la entrada y los eventos que sobre la misma se produzcan. En líneas generales, el principal cometido de este subsistema será el mantenimiento de clientes que deseen conocer cuándo un determinado evento de ratón o de teclado se ha producido. Los eventos se dividirán en los relativos al movimiento (sólo para el ratón) y los relativos a la pulsación de un botón (por botón entenderemos tanto una tecla del teclado como un botón del ratón). El diagrama será el siguiente:

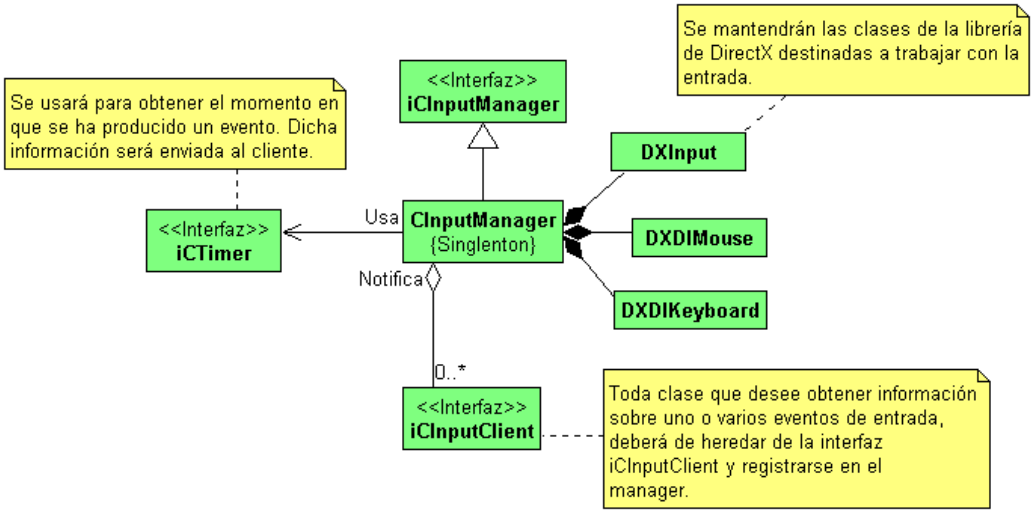


Diagrama del subsistema CInputManager.

4.3.9.1. La clase CInputManager.

En `CInputManager` se mantendrán las clases de la librería de DirectX destinadas al control de la entrada (`DXInput`, `DXDIMouse` y `DXDIKeyboard`) y las instancias de los clientes que deseen conocer el estado de ciertos eventos. Dichos clientes se recogerán a través de la instancia `iCInputClient`, pudiendo atender a uno o más eventos. Cuando dichos eventos de entrada se produzcan serán notificados conjuntamente con el instante de tiempo en que se hubieran producido (de ahí el uso de `iCTimer`).

La clase `CInputManager` será mantenida desde `CCRISOLEngine` y en cada actualización lógica recibirá una orden de actualización interna. En dicha actualización interna ejecutará los métodos `Update` de las clases DirectInput para obtener los últimos cambios que se hayan producido en la entrada (ratón y teclado).

Cuando un evento se produzca, `CInputManager` recorrerá la lista de clientes asociados al mismo notificando, para cada uno de ellos, el suceso. Los clientes deberán de retornar un flag (valor `true` o `false`), de tal modo que el primero que retorne `true` querrá indicar que se queda con el evento que se ha producido sin posibilidad de que llegue al resto de clientes asociados a ese mismo evento. De retornar `false`, el evento podrá ser notificado al cliente que le siga en la lista. Este funcionamiento será muy importante pues permitirá filtrar sucesos entre distintos interfaces. Los clientes serán instalados en `CInputManager` en una lista que hará las veces de una pila pues los últimos clientes que se instalen tendrán preferencia sobre los antiguos (les llegará antes la notificación y, por lo tanto, la posibilidad de quedarse con el evento en exclusiva).

Por último, `CInputManager` será capaz de trabajar con `Win32` para obtener un flujo de entrada normalizado y habilitar así operaciones de escritura de caracteres.

4.3.9.2. La interfaz iCInputClient.

Todas aquellas clases que deseen atender a un determinado evento de entrada deberán de heredar de esta instancia pues `CInputManager` los recogerá a través de la misma. Una vez registrados, `CInputManager` ejecutará el método de notificación de `iCInputClient`, para indicar a los clientes de que su evento de entrada ha sucedido. Tal y como se comentaba en el apartado sobre `CInputManager`, un evento llegará a un cliente siempre y cuando no haya otro cliente pendiente del mismo evento que haya decidido quedarselo.

Un mismo cliente podrá escuchar varios eventos diferentes, bastará con registrarlo múltiples veces para cada uno de ellos.

4.3.10. El subsistema para la gestión de fuentes.

El subsistema tendrá por objetivo mantener todas las fuentes que se hallen instaladas en el sistema así como recoger las peticiones de escritura en pantalla.

Cuando se desee escribir un texto en pantalla se utilizará la fuente que internamente se halle seleccionada. El subsistema podrá trabajar con varios tipos distintos de fuente. Las fuentes se definirán en el archivo de inicialización del motor "`CrisolEngine.ini`" (para más información, consultar el apéndice sobre *CrisolBuilder*). En la versión actual, no se permite definir la fuente con la que el diseñador escribirá información en pantalla por lo que siempre será utilizada la que se haya establecido por defecto. Todas las fuentes deberán de ser TrueType y estar instaladas en el sistema. Si se define una fuente que no existe se usará la fuente por defecto. El diagrama del subsistema será el siguiente:

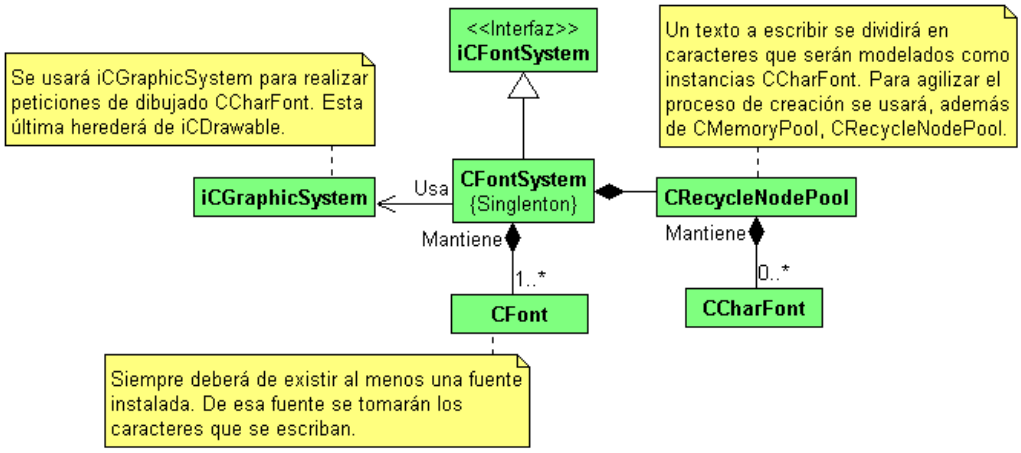


Diagrama del subsistema de gestión de fuentes.

4.3.10.1. La clase CFontSystem.

La clase `CFontSystem` encapsulará el subsistema. Al ordenar a `CFontSystem` que escriba algo en pantalla, internamente descompondrá el texto a escribir en caracteres. Cada carácter será modelado usando una instancia `CCharFont` (clase comentada en el apartado sobre el subsistema gráfico) que, por motivos de rapidez, será mantenida en un nodo (que usará el componente `CMemoryPool`) que se registrará a través de `CRecycleNodePool`. Dicha instancia será entonces enviada a `CGraphicSystem` para que sea dibujada en pantalla. Naturalmente, `CCharFont` heredará de `iCDrawable` (la interfaz con la que `CGraphicSystem` recoge los elementos que dibuja en pantalla).

Todas las fuentes que se registren en el subsistema serán modeladas como instancias `CFont`. Desde `CCRISOLEngine` se registrarán dichas fuentes previa lectura de las mismas desde el fichero de configuración del motor.

En cada ciclo de dibujado en `CGraphicSystem`, se deberá de ejecutar un método en `CFontSystem` destinado a resetear la instancia `CRecycleNodePool` y así tener disponibles, de nuevo, las instancias `CCharFont` para la representación de los caracteres. Dicho método estará disponible a través de su interfaz `iCFontSystem` pero de manera privada. Como `CGraphicSystem` se declarará clase amiga de la interfaz, ésta podrá tener acceso al mismo.

4.3.10.2. La clase CFont.

La clase `CFont` representará una fuente instalada en el subsistema `CFontSystem`. Internamente mantendrá una textura de tamaño 256x256 sobre la que se dibujarán los caracteres de la fuente True Type a la que representan. La representación de la textura con todos los caracteres se realizará a través de una instancia `CImage`. El diagrama será muy sencillo:

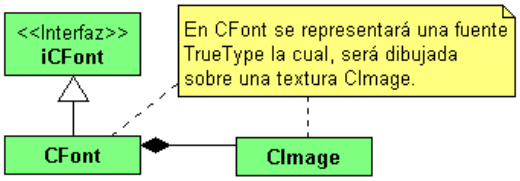


Diagrama de CFont.

Los caracteres de la fuente serán dibujados, a su vez, por instancias `CCharFont` que mantendrán dos tipos de informaciones relativas a una fuente. Por un lado el handle a la textura que mantiene la instancia `CImage` y, por otro, la región en la misma donde localizar el carácter que representan.

`CFont` heredará de un interfaz `iCFont` que será servido a las instancias `CCharFont` para que éstas se puedan inicializar.

4.3.11. El subsistema para la gestión del audio.

Este subsistema será utilizado para almacenar las peticiones de reproducción de sonido WAV y para el mantenimiento del segmento MIDI que podrá hacerse sonar como música de fondo del juego.

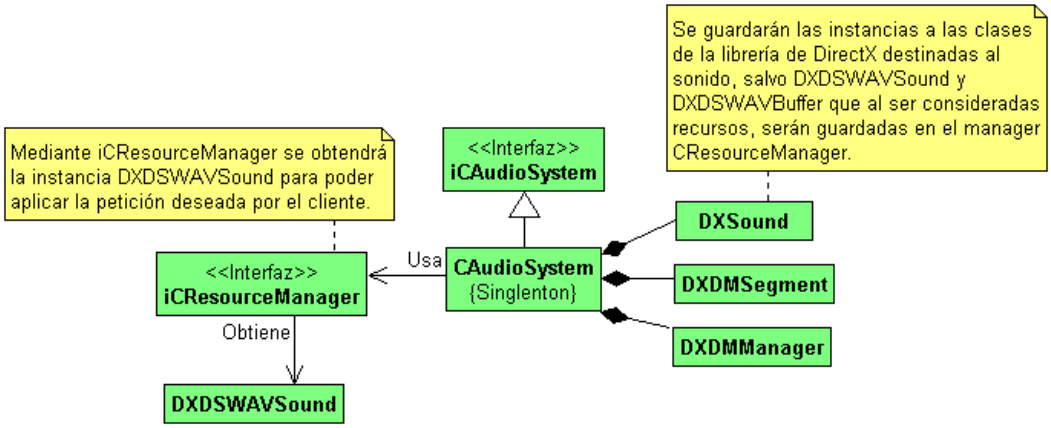


Diagrama del subsistema de audio.

Desde cualquier punto del motor se podrá hacer una operación sobre un fichero WAV con solo pasar al sistema de audio el handle y el código de la acción a realizar. Los ficheros MIDI serán completamente gestionados por el subsistema de audio pues sólo podrá existir uno sonando a la vez. A continuación se explicará con más detalle la clase `CAudioSystem`, que será la encargada de implementar el subsistema de audio.

4.3.11.1. La clase CAudioSystem.

`CAudioSystem` será la clase encargada de implementar la lógica del subsistema de audio. El funcionamiento general será muy similar a la filosofía utilizada para `CGraphicSystem`. Desde el exterior se solicitará realizar operaciones sobre un fichero WAV mediante el uso de su handle (obtenido del servidor de recursos, `CResourceManager`). Esas peticiones (reproducción, pausa, reanudación o finalización) serán almacenadas en una cola. Cuando desde el exterior se actualice el subsistema de audio, éste recorrerá la cola ejecutando las peticiones pendientes.

En lo relativo a los ficheros MIDI, como sólo podrá hacerse sonar un fichero de fondo de cada vez, en el subsistema se mantendrá un segmento que podrá ser cargado con diferentes sonidos MIDI. La petición de reproducción, pausa, reanudación o finalización de un fichero MIDI también será atendida en una cola. Para el control de los archivos MIDI se mantendrá un hilo de muy baja prioridad controlando la notificación de eventos del fichero relativos a la finalización del mismo. Esto será así ya que DirectX funcionará de forma más efectiva mediante el uso de hilos de ejecución para la notificación del estado de un fichero MIDI a través de DirectMusic.

La instancia a la clase `DXSound`, así como todas las referidas a DirectMusic (`DXDMManager` y `DXDMSegment`), serán mantenidas en `CAudioSystem`. Será necesario que al inicializar el subsistema se pase la configuración básica para el trabajo con DirectSound. *Crisol* exijirá que el equipo en donde se ejecute posea tarjeta de sonido.

4.3.12. El gestor de scripts.

Los scripts serán lanzados cuando ocurran determinados eventos durante el juego. De esta forma, en aquellos lugares en donde sucedan eventos a los que se pueda asociar código script,

Crisol comprobará si existe el nombre de un fichero script asociado. Si existe, se pondrá en contacto con el gestor de scripts y solicitará la ejecución del mismo, pasándole los parámetros que hagan falta. El gestor, cuando proceda atender a las solicitudes, se encargará de crear objetos que encapsularán el código script a ejecutar. Las clases que produjeron el evento con el script asociado se convertirán en clientes del mismo y recibirán notificación cuando éste termine de ejecutarse o suceda algún problema y se deba de interrumpir. El diagrama del subsistema gestor de scripts será el siguiente:

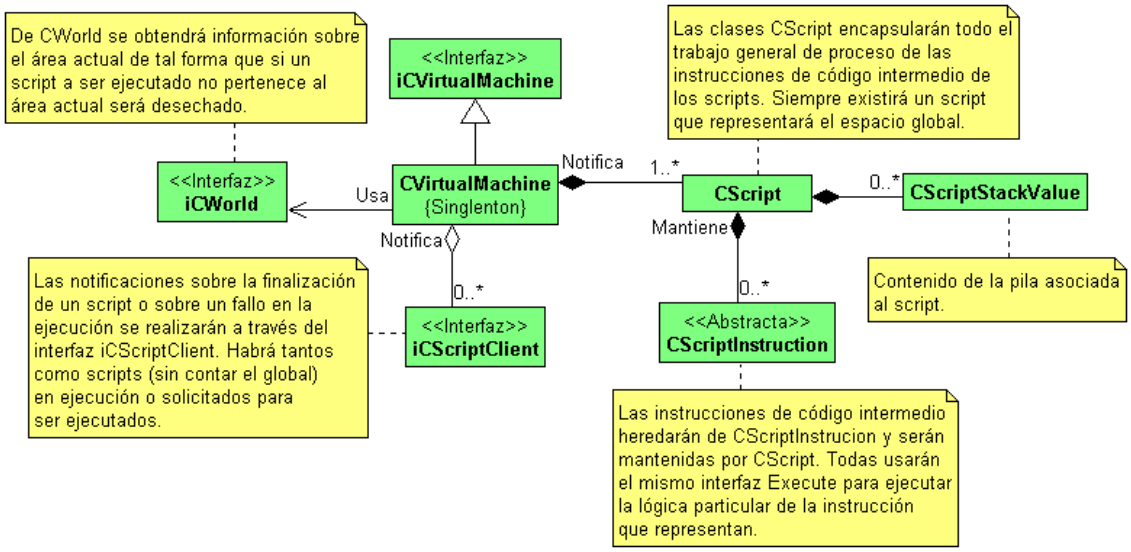


Diagrama del gestor de scripts.

La clase `CVirtualMachine` será el gestor de scripts, el lugar a donde se deberá de acudir para solicitar la ejecución de los mismos, mientras que `CScript` se encargará de representar a un script. En su interior, las instrucciones heredarán de una clase `CScriptInstruction` con un método `Execute` a implementar para llevar a cabo la lógica asociada [Cueva'99]. El número de clases herederas de `CScriptInstruction` será igual al número de instrucciones de código intermedio existentes. Las instancias `CScript` mantendrán una pila en donde se guardarán los datos relativos al trabajo con el código intermedio. Cada elemento de la pila será una instancia `CScriptStackValue`. Por último, los clientes que lancen los scripts heredarán de `iCScriptClient` para recibir las notificaciones sobre la ejecución de los mismos.

4.3.12.1. La clase CVirtualMachine.

Como ya se ha indicado, toda la gestión en la ejecución de scripts será llevada desde la clase `CVirtualMachine` (mantenida en `CCRISOLEngine`). Cuando una clase desee ejecutar un script en primer lugar deberá de heredar de la interfaz `iCScriptClient` pues siempre recibirá notificación cuando el script finalice o bien sea interrumpido ante un error. Por otro lado, deberá de conocer el evento al que pertenece el script a ejecutar. Dicho evento estará reflejado por un método en `CVirtualMachine`, accesible a través de su interfaz, que será al que se deberá de llamar. En dicho método se esperará recibir, además del cliente a través de la interfaz `iCScriptClient`, los parámetros que necesitará el script para comenzar a ser ejecutado.

Los scripts no serán procesados de inmediato sino que serán depositados en una cola de scripts pendientes de ejecución. Desde `CCRISOLEngine` se indicará cuando se deberán de ejecutar los scripts gestionados en `CVirtualMachine` (en cada actualización lógica). En ese momento, `CVirtualMachine` hará dos recorridos. Por un lado iterará por todos los scripts que pudieran estar pausados de ciclos de actualización anteriores. Por cada uno de ellos comprobará si siguen pausados. Si es así, no hará nada con ellos mientras que si están sin la pausa los ejecutará comprobando, al acabar, si han finalizado o siguen en pausa. Si siguen en pausa se devolverán a la lista de scripts pausados (a la lista en la que estaban) mientras que si han finalizado se notificará a su cliente y los descartarán. El segundo recorrido será por las solicitudes de ejecución de scripts. Por cada una de ellas, intentará inicializar una instancia

CScript. Si no hay problemas procedará a ejecutarlo. Si al acabar el script está en pausa lo pondrá en la lista de scripts en pausa y si ha acabado notificará al cliente que solicitó su ejecución. Como se puede observar, el orden de recorrido (primero scripts que estén en pausa y luego solicitudes) será muy importante.

Siempre que se proceda a ejecutar un script en pausa o a procesar una solicitud, se comprobará el identificador de área a la que pertenece el script. Si el script pertenece a un área que ya no se halla activa no será ejecutado. Este hecho será muy importante pues el dieñador, en cualquier momento, podrá cargar un área produciendo que existan solicitudes anteriores (pertenecientes al área sustituido). La única excepción ocurrirá en el trabajo con los scripts asociados al universo de juego y al juego como aplicación, que no se verán afectados por el cambio de área.

En CVirtualMachine se mantendrá una instancia CScript que se corresponderá con el script de ámbito global. El ámbito global simplemente mantendrá variables y constantes accesibles desde cualquier script, por lo que deberá de existir siempre y pasarse a toda instancia CScript que se inicialice.

4.3.12.2. La clase CScript.

Representará un script. Un script estará formando por un espacio de memoria donde realizar sus operaciones y mantener sus variables y por un conjunto de instrucciones que deberá de ejecutar. El espacio de memoria estará representado por una pila encargada de mantener los denominados stack frames del script. Los stack frames representarán un ámbito de ejecución de código (esto es, el código asociado a funciones o al propio cuerpo de un script) y estará formado por un espacio para las variables locales, otro para la realización de operaciones intermedias y otro informativo sobre los stack frames anteriores. Toda esta información se representará por instancias CScriptStackValue y se guardarán en una pila.

Siempre existirá un stack frame activo que será el asociado al cuerpo de la función y se establecerá de forma automática. El resto de los stack frames se asociarán cuando se ejecute el código de funciones locales al script.

Todas las instrucciones de código intermedio serán mantenidas por la instancia y creadas nada más inicializarla. Cada una de ellas será recogida a través de su clase base CScriptInstruction.

Los scripts, tras ejecutar una instrucción, podrán interrumpirse si ha existido algún problema, pausarse si han ejecutado una instrucción que requiere proceso extra (por ejemplo, el movimiento de una entidad) o finalizar si se ha ejecutado la última instrucción de todas.

La instancia CScript heredará de una interfaz iCScript que será pasada a las instancias herederas de CScriptInstruction para que se puedan ejecutar.

4.3.12.3. La clase CScriptStackValue.

Esta clase servirá para representar los valores en la pila. Los valores en la pila podrán ser de tres tipos fundamentales: cadenas de caracteres (string), números reales (float) o bien enteros sin signo (dword). Todos estos valores serán representados en la clase CScriptStackValue que se adaptará al tipo de dato que proceda.

Las instancias CScriptStackValue serán creadas y destruidas por CScript o bien por las instrucciones que ejecute, las cuales serán guardadas como punteros.

La clase estará creada con todos sus operadores sobrecargados para facilitar el trabajo de manipulación de CScriptStackValue como valor de pila.

4.3.12.4. La interfaz iCScriptClient.

Toda clase que tenga la posibilidad de ponerse en contacto con `CVirtualMachine` para la solicitud de ejecución de un script, deberá de heredar de esta interfaz para recibir una notificación cuando el script finalice o se interrumpa.

4.3.12.5. Las instrucciones de código intermedio.

Todas las instrucciones de código intermedio serán mantenidas por las instancias `CScript` y todas heredarán de una clase base denominada `CScriptInstruction`. El número de instrucciones de código intermedio será muy elevado ya que las habrá de dos tipos. Por un lado las instrucciones base y por otro las que representarán llamadas concretas a los métodos de los diferentes subsistemas del motor y que serán las realmente elevadas.

Para gestionar su funcionamiento se usará el patron `Command`, basado en asociar a la clase base un método `Execute` que todas las demás instrucciones rellenarán con la lógica que proceda. El método `Execute` de las instrucciones de código intremedio recibirán la instancia `CScript` a la que pertenecen a través de su interfaz con el fin de poder actuar sobre el mismo (para pausarle, interrumpirle, obtener de él un valor de la pila, depositar resultados, etc).

4.3.13. El subsistema para la gestión del universo de juego.

Este subsistema será el corazón de *Crisol* desde el punto de vista de su importancia a la hora de gestionar el funcionamiento de un juego. Estará formado por tres clases fundamentales; por un lado `CWorld`, que simulará la gestión del universo de juego, por otro lado `CArea`, que mantendrá los datos de la zona de juego en la que se halle el personaje controlado por el jugador y, por último, de `CIsoEngine`, que se encargará de trabajar sobre los datos mantenidos en `CArea` para implementar la lógica de búsqueda de caminos, desplazamiento del mapa, etc. El diagrama general del subsistema será el siguiente:

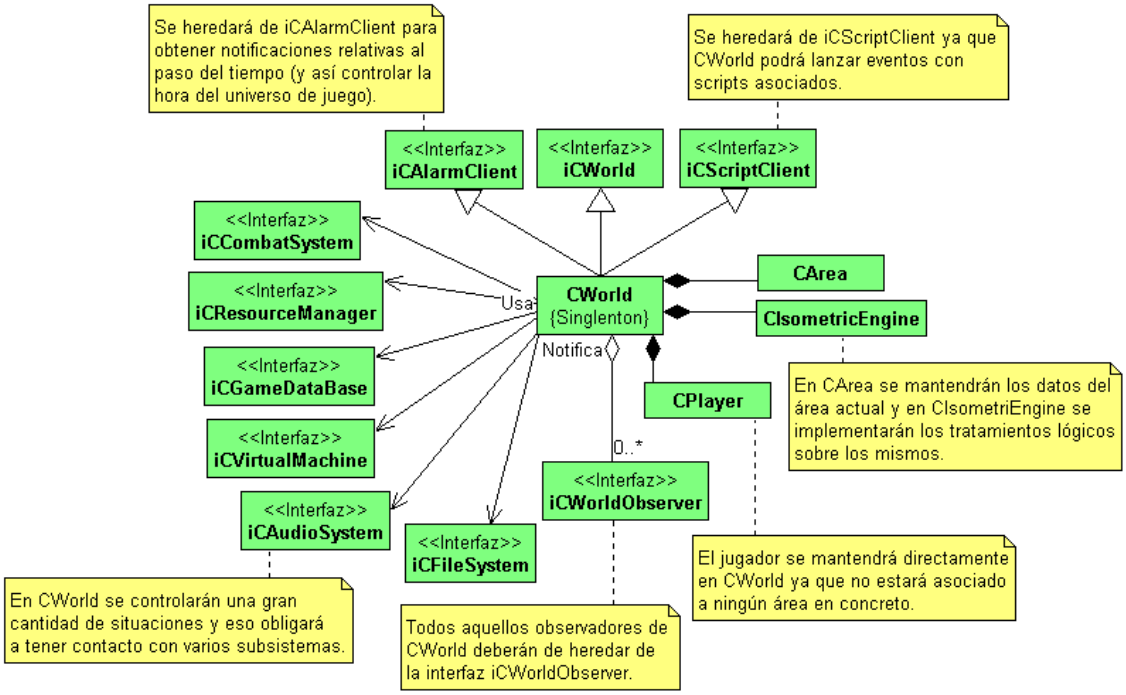


Diagrama del gestor del universo de juego.

Cada una de las tres clases comentadas tendrá un papel largo y complejo en lo referido a la gestión de un juego, a continuación se procederá a comentar con mayor profundidad cada una de ellas.

4.3.13.1. La clase CWorld.

`CWorld` será la clase principal en la gestión del universo de juego. Actuará de Facade ofreciendo una interfaz unificada para las otras dos grandes e importantes clases del subsistema: `CArea` y `CIsoEngine`.

En `CWorld` se mantendrá la instancia al jugador (se verá más sobre la clase `CPlayer` en el apartado siguiente sobre entidades) ya que éste no pertenecerá a ningún área en concreto y deberá de existir ante los diversos cambios de localidad (áreas).

Para la gestión del paso del tiempo, `CWorld` utilizará el subsistema de alarmas, de ahí que herede de `iCAlarmClient` para convertirse en cliente y poder recibir notificaciones. Las notificaciones estarán basadas según las especificaciones de las reglas de juego.

Como `CWorld` podrá lanzar eventos con scripts asociados, heredará de `iCScriptClient` y utilizará la interfaz `iCVirtualMachine` para comunicarse con la máquina virtual y así realizar la solicitud de ejecución de scripts.

`CWorld` mantendrá observadores con el fin de que éstos sepan los cambios generales que puedan suceder en el universo de juego. Todos los observadores deberá de heredar de `iCWorldObserver` y registrarse en `CWorld`.

Las operaciones de cargar y guardar partidas serán llevadas a cabo desde `CWorld` para luego propagarse al resto de subsistemas desde donde se podrán recuperar o guardar el resto de datos, pero `CWorld` actuará de filtro (pues podrá prohibir una petición de guardar partida) y gestor a la hora de realizar esta operación.

Desde `CWorld` también se mantendrá la posibilidad de hacer sonar un fichero WAV de sonido ambiente (que será reproducido de forma cíclica) o bien uno de carácter anónimo (con reproducción lineal).

4.3.13.2. Las entidades.

Antes de pasar a explicar la clase `CArea`, será muy importante tener en cuenta la jerarquía de clases que implementará a las entidades de juego. En la etapa de análisis de *Crisol*, se dejó bien claro que las entidades podrían ser ítems, paredes, objetos de escenario y criaturas, considerando al jugador como una entidad extendida de éstas últimas. Ahora veremos que a estas entidades se las considerará entidades del universo de juego mientras que a los tiles base (suelo) y tiles superiores (techos) se les considerará también entidades, pero simples, esto es, que no podrán ser interactuables en el universo de juego por parte de criatura alguna. El diagrama general de las entidades será el siguiente:

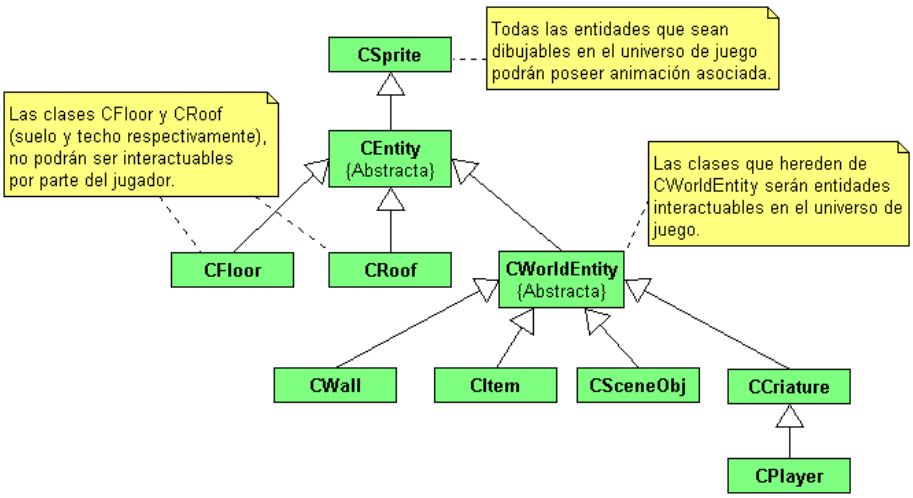


Diagrama general sobre la jerarquía de las entidades.

Todas las entidades heredarán de la clase `CSprite`, por lo que incorporarán la capacidad de ser representables en pantalla con animación así como poderse desplazar sobre la celda a la que estén asignadas (si bien, sólo las criaturas gozarán de tal posibilidad en la práctica).

Las entidades se definirán a partir de archivos de datos. En dichos archivos, se establecerán perfiles o secciones en donde se agruparán conjuntos de datos y especificaciones asociadas a las mismas. Posteriormente, en el archivo de diseño del área, se podrán establecer *instancias* a dichos perfiles para crear una entidad específica. La utilidad *CrisolBuilder* se encargará de pasar los archivos de datos a formato binario con el fin de acelerar el proceso de carga.

Todas las entidades podrán guardar y cargar sus datos a disco con el fin de que el proceso de recuperar un juego previamente guardado o grabarlo sea repartido desde `CWorld` a `CArea` pasando por todas y cada una de las entidades.

Las instancias `CCriature` y `CSceneObj` tendrán la particularidad de que serán (en el caso de `CSceneObj`, esto figurará de forma opcional) contenedores de ítems, es decir, podrán portar ítems en su interior (realmente, handles a ítems ya que las instancias a todas las entidades, salvo el jugador, serán mantenidas en `CArea` exclusivamente). Para ello, dispondrán de instancias a la clase `CItemContainer`. Dicha clase mantendrá ítems de una forma simple y dispondrá de un iterador asociado, `CItemContainerIt`, para facilitar la iteración entre ellos. En el caso de las instancias `CCriature` se mantendrá además un objeto `CEquipmentSlots` para el mantenimiento de los ítems equipados.

Cada clase mostrada en el diagrama anterior tendrá una funcionalidad característica. Todas ellas serán mantenidas en la clase `CArea` (a excepción del jugador, instancia `CPlayer`) y tan sólo el jugador y los ítems (transportados por el jugador) podrán cambiar de área con respecto a la que fueron situados inicialmente.

Todas las entidades, incluido el jugador, tendrán un handle asociado. Los handles no serán más que valores numéricos asignados por `CArea` al inicializarse. Gracias a los handles se podrán obtener instancias concretas a entidades. En el caso particular del trabajo con *CrisolScript*, será fundamental para una comunicación con las instancias almacenadas en el área de juego actual (las variables `entity` de *CrisolScript* representarán realmente handles).

4.3.13.2.1. La clase CEntity.

Esta será la clase base para cualquier tipo de objeto representable en el universo de juego. Será una clase abstracta pues habrá multitud de aspectos que se deban de concretar en las clases derivadas. El diagrama particular será el siguiente:

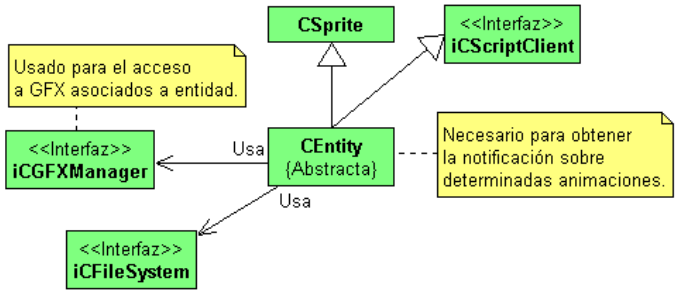


Diagrama de CEntity.

Desde `CEntity` se mantendrán una serie de elementos globales a todas las entidades como el handle, la elevación sobre el terreno y la relación de eventos script activos. También se gestionará el dibujado de los GFX asociados, de ahí el uso de la instancia `iCGFXManager` para el acceso al subsistema de efectos especiales.

El uso de `iCFileSystem` será necesario debido a la posibilidad de recuperar y grabar datos a disco.

Desde el punto de vista de los métodos, en `CEntity` se llevará a cabo el dibujado final de las entidades, se controlará el estado de pausa y la visibilidad de las mismas. El concepto de visibilidad será muy importante para ahorrar recursos. Cuando una entidad no sea visible, el motor isométrico dará instrucciones para que se detenga el comando de animación asociada a la misma o bien bajará el volumen de sus posibles sonidos asociados.

Se heredará de `iCScriptClient` pues será común que las clases que representen las entidades concretas posean eventos asociados, por lo que la mejor posición donde situar la herencia de ésta interfaz cliente será en `CEntity`.

4.3.13.2.2. La clase CWorldEntity.

La clase `CWorldEntity`, clase abstracta, será base para todas aquellas entidades que, siendo reprentables en el universo de juego, podrán ser también interactuables. Así, las clases destinadas a representar los ítems, los objetos de escenario, paredes y criaturas heredarán de `CWorldEntity`. La clase `CWorldEntity` heredará de `CEntity`, también abstracta, y no concretará aspectos de esta, en su lugar, añadirá nuevas extensiones para que las derivadas concreten. El diagrama será el siguiente:

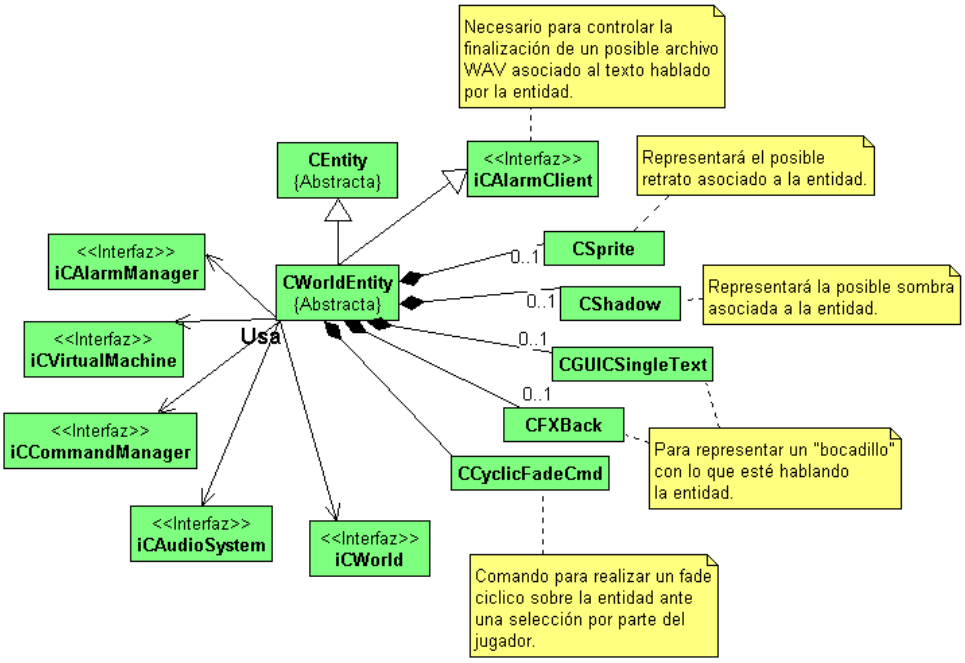


Diagrama de CWorldEntity.

Tal y como se puede observar en el diagrama, desde `CWorldEntity` se controlará un considerable número de situaciones. En primer lugar, la clase mantendrá la información relativa al posible retrato de la entidad que represente, así como una también posible instancia `CShadow` para mantener la sombra de la misma. La sombra variará en intensidad según las especificaciones que existan en subsistema de GFX de forma automática.

Otros aspecto importantes serán el mantenimiento de una instancia `CCyclicFadeCmd`, que se activará cuando el usuario, el jugador, quiera interactuar con la entidad, haciendo que ésta parpadee suavemente en pantalla. No será más que un comando que controlará las estructuras RGB asociadas a la entidad.

El último de los componentes serán en realidad dos; una instancia `CGUICSingleText` y otra `CFXBack`. Dichos objetos se crearán cuando la entidad desee hablar, y servirán para mostrar un “bocadillo” en pantalla con el texto que la entidad esté pronunciando. La posibilidad de reproducir de forma audible el texto, hará que `CWorld` tenga que ser un cliente `iCAlarmClient` para recibir notificación de la finalización del archivo de sonido, pero también se usará como tal para controlar el tiempo que el texto hablado figure en pantalla. Tal y como

se comentó en la etapa de análisis, todas las entidades podrán hablar y es por ello que tal posibilidad se asocie en `CWorldEntity`.

Toda entidad de tipo `CWorldEntity` esperará que las clases derivadas devuelvan su máscara de acceso. La máscara de acceso servirá para determinar si cuando una de estas entidades se halla sobre un tíle, es posible el acceso al mismo en cada una de las diferentes direcciones. Se cumplirá que los ítems siempre permitirán el acceso, mientras que las criaturas y objetos de escenario podrán configurarse al respecto. Las paredes lo permitirán dependiendo de su orientación y de si están, o no, bloqueadas.

Desde la clase se hará uso de un considerable número de interfaces a subsistemas del motor. Por un lado se utilizar `iCAlarmClient` para asociar la notificación en la finalización del posible sonido WAV asociado al habla y para controlar el tiempo que el texto esté en pantalla. Por otro, `iCFileSystem` se utilizará para el tratamiento de las operaciones de almacenamiento y recuperación de datos necesarios en la etapa de guardar y cargar partida. La instancia `iCCommandManager` servirá para solicitar la ejecución del comando de fade cíclico que permitirá implementar el efecto de parpadeo suave sobre la entidad si ésta es seleccionada, mientras que `iCAudioSystem` será utilizada para solicitar la reproducción del posible sonido WAV asociado al habla. Por último, el uso de `iCWorld` se utilizará únicamente para que, durante la llamada al método de finalización, `End`, se pueda solicitar al gestor del universo de juego la desvinculación de cualquier luz asociada a la entidad.

4.3.13.2.3. La clase CFloor.

Será la clase encargada de representar los tiles base, esto es, el suelo de un área. Al heredar de la clase `CEntity`, las entidades que representan el suelo, no podrán ser manipuladas por parte del jugador. El diagrama será muy sencillo:

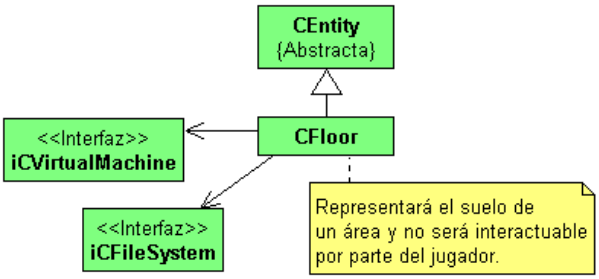


Diagrama de CFloor.

Cada tipo de `CFloor` podrá tener asociado un archivo de sonido WAV activable mediante las pisadas de criaturas en ellas (que serán obtenidos a través de su nombre desde el comando de movimiento `CMoveCmd` en las instancias `CCriature`). El uso de las interfaces `iCVirtualMachine` y `iCFileSystem` serán necesarias para la solicitud de ejecución de los eventos producidos sobre una instancia `CFloor` y para el trabajo en las operaciones de guardar y cargar partidas respectivamente.

Las instancias `CFloor` deberán de existir obligatoriamente sobre una celda para que se puedan situar otro tipo de entidades en la misma. De no estar vinculadas, sobre el terreno del área aparecerá un “hueco” isométrico de color negro.

4.3.13.2.4. La clase CRoof.

Desde esta clase se representarán las entidades asociadas a los tiles superiores, esto es, a los techos. Serán las entidades más sencillas de todas pues, básicamente, su misión será la de ser representables en pantalla. El diagrama será:

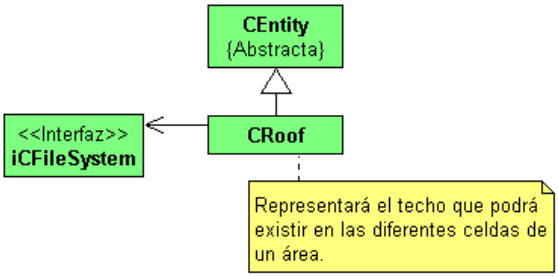


Diagrama de CRoof.

Las entidades que representen los techos no tendrán eventos asociados y únicamente harán uso del subsistema `CFileSystem` para guardar y recuperar la información en las operaciones de cargar y guardar partidas.

Desde el motor isométrico, como se verá más adelante, se realizará una gestión sobre el dibujado de los techos cuando éstos puedan tapar a las entidades que posean la cámara.

4.3.13.2.5. La clase CWall.

Las instancias `CWall` representarán las paredes y puertas dibujables en el universo de juego. Toda pared podrá tener un total de tres orientaciones (sur, sudoeste y sudeste) que será necesario especificar a la hora de crear las instancias de tipo pared en el archivo de diseño de las áreas. El diagrama de las entidades no será muy complejo:

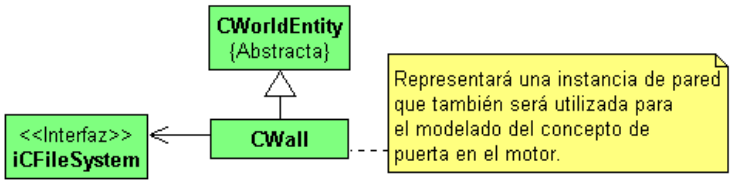


Diagrama de CWall.

Las paredes podrán manipularse para estar bloqueadas o desbloqueadas, esto es, para permitir, o no, el franqueo de los tiles donde sean colocadas las mismas. Desde el motor isométrico se controlará si las paredes llegan a tapar a la entidad que posea la cámara para hacerlas transparentes (tal y como se explicará en profundidad en el apartado del motor isométrico). El uso de `iCFileSystem` estará restringido al proceso de guardar y cargar partida.

4.3.13.2.6. La clase CItem.

Los ítems serán objetos capaces de ser transportados y equipados por las criaturas, además de mantenibles por éstas y por los objetos de escenario que sean contenedores. El diagrama de los ítems será el siguiente:

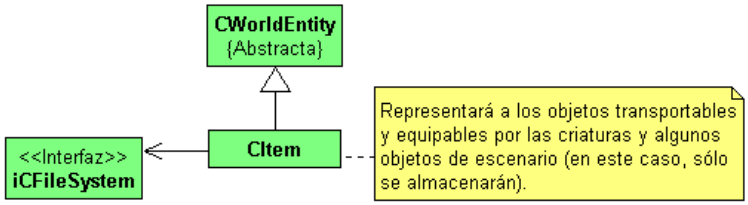


Diagrama de CItem.

Los ítems tendrán siempre asociado un dueño. El dueño estará representado por el handle a la criatura u objeto de escenario que los posea o bien será un handle nulo si éstos se hallan sobre el suelo.

Los ítems podrán ser contenidos a través de dos clases; `CItemContainer` y `CEquipmentSlots`. En ambos casos no se mantendrán las instancias a las clases `CItem` (éstas serán siempre guardadas en `CArea`) sino que mantendrán handles a los mismos.

El uso de la interfaz `iCFileSystem` será debido al trabajo con las operaciones de guardar y cargar partidas.

4.3.13.2.7. La clase CSceneObj.

Los objetos de escenario representarán elementos del área interactuables, o no, pero que no podrán ser transportados ni equipados. Algunos de estos objetos de escenario tendrán la facultad de poder ser contenedores de ítems. El diagrama será sencillo:

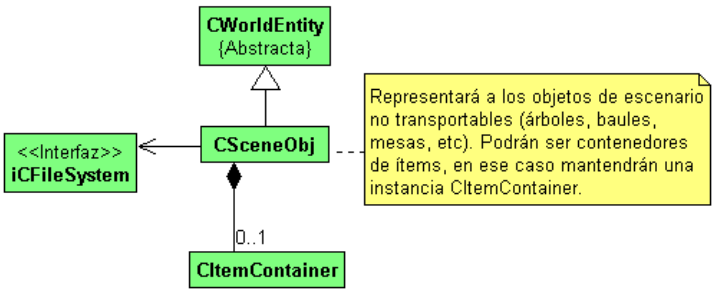


Diagrama de CSceneObj.

El uso de una instancia `CItemContainer` se hará efectivo desde el mismo momento en que el objeto de escenario se cree como contenedor de ítems. La interfaz `iCFileSystem` será utilizada para las operaciones de cargar y guardar partidas.

4.3.13.2.8. La clase CCriature.

La clase `CCriature` será la que se encargue de representar a las entidades más complejas; las criaturas. Las criaturas tendrán la posibilidad de desplazarse por el escenario de juego, poseer un inventario, equipar ítems e interactuar sobre otras entidades. El jugador será una especialidad de las criaturas pero añadirá muy pocas características extra, de hecho, como se verá en el siguiente apartado, únicamente interesará la forma particular que tendrá de inicializarse, que será a partir de la edición que haga el jugador. El diagrama de `CCriature` será el siguiente:

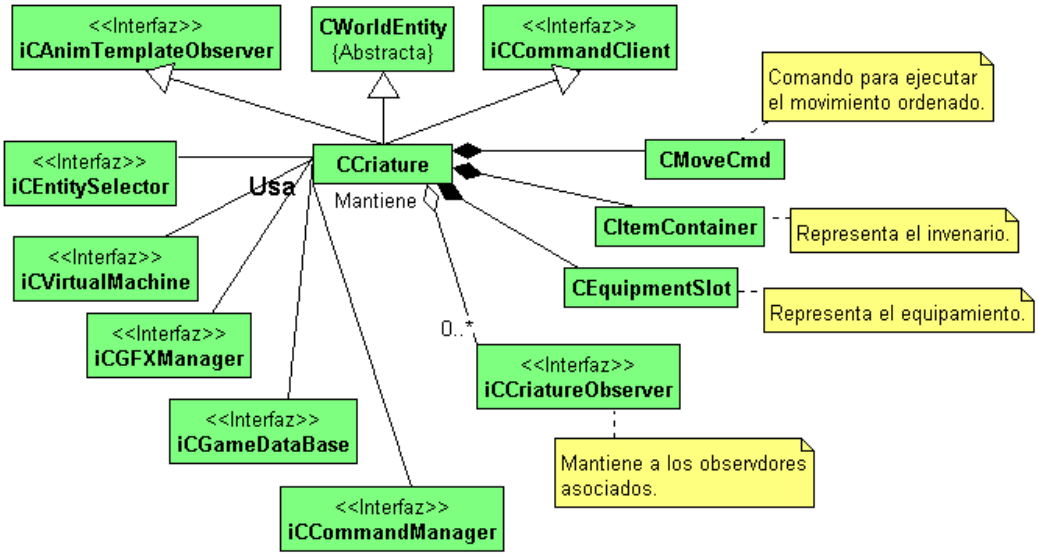


Diagrama de CCriature.

Como se puede observar, la clase `CCriature` representa a una entidad considerablemente compleja. Inicialmente se deberá de tener en cuenta que la criatura deberá de estar pendiente de las animaciones que tenga asociadas para coordinar las acciones ordenadas y la finalización de las animaciones que lo representen. Para lograr esto, heredará de `iCAnimTemplateObserver`.

El movimiento se lleverá a cabo a partir del comando `CMoveCmd`, de ahí la necesidad de usar la interfaz `iCCommandManager` para solicitar su ejecución y la herencia de `iCComandClient` para obtener la notificación de la finalización del mismo. El comando de movimiento, internamente, solicitará al motor isométrico (a través de `CWorld`) que le localice un camino posible entre la posición actual de la criatura y la deseada a alcanzar. En cada ciclo de actualización irá desplazándose por el trayecto calculado (si es que existe trayecto posible). Toda acción ordenada a una criatura que exija antes una aproximación provacará una llamada interna a la orden de caminar.

Una criatura será un contenedor de ítems en dos sentidos. Por un lado mantendrá un inventario de ítems y por otro poseerá un equipamiento, esto es, un conjunto de ítems equipados en las diferentes partes de su cuerpo. Para gestionar los handles de todos estos ítems, `CCriature` dispondrá de una instancia `CItemContainer`, para el inventario, y `CEquipementSlot`, para el equipamiento.

Todas las instancias que deseen ser observadores de las criaturas deberán de heredar de la interfaz `iCCriatureObserver` y registrarse en la clase. Este interfaz será particularmente importante cuando se trabaje con el jugador puesto que la interfaz principal deberá de mostrar de inmediato los cambios en los atributos del mismo.

El uso de un interfaz `iCEntitySelector` se deberá a que, en el método de dibujado, la criatura deberá de comprobar si está combatiendo para obtener su selector de combate. Al obtenerlo, deberá de dibujarlo antes de pasar la responsabilidad a la clase superior, `CWorldEntity`. Esto es así porque el selector de combate siempre se dibujará a los pies de la criatura.

El uso de `iCGFXManager` se deberá a la obtención del componente de transparencia asociado a una criatura que se desee hacer invisible. Este valor se habrá especificado en el archivos de FX y se deberá de tomar de ese subsistema.

4.3.13.2.9. La clase CPlayer.

Esta clase será una simple especialización de `CCriature`. Se usará principalmente para establecer la forma de inicialización de este tipo de criatura, que siempre se hará a partir de la edición de la misma en la interfaz de creación de personajes. Su diagrama será:

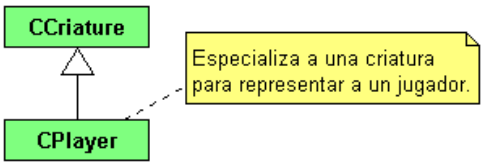


Diagrama de la clase CPlayer.

La instancia `CPlayer` será mantenida en `CWorld` y será la única criatura que pueda andar entre áreas distintas con lo que ello signficará en cuanto al transporte de ítems.

4.3.13.3. La clase CArea.

El universo de juego por el que podrá moverse el personaje controlado por el jugador estará dividido en áreas. Al menos deberá de existir un área. La clase `CArea` se encargará de representar el área en el que actualmente se halle el jugador. En su interior mantendrá todas las entidades de la misma y llevará a cabo todo tipo de actualización de información a medida de que el jugador interactúe sobre ella. El diagrama general será el siguiente:

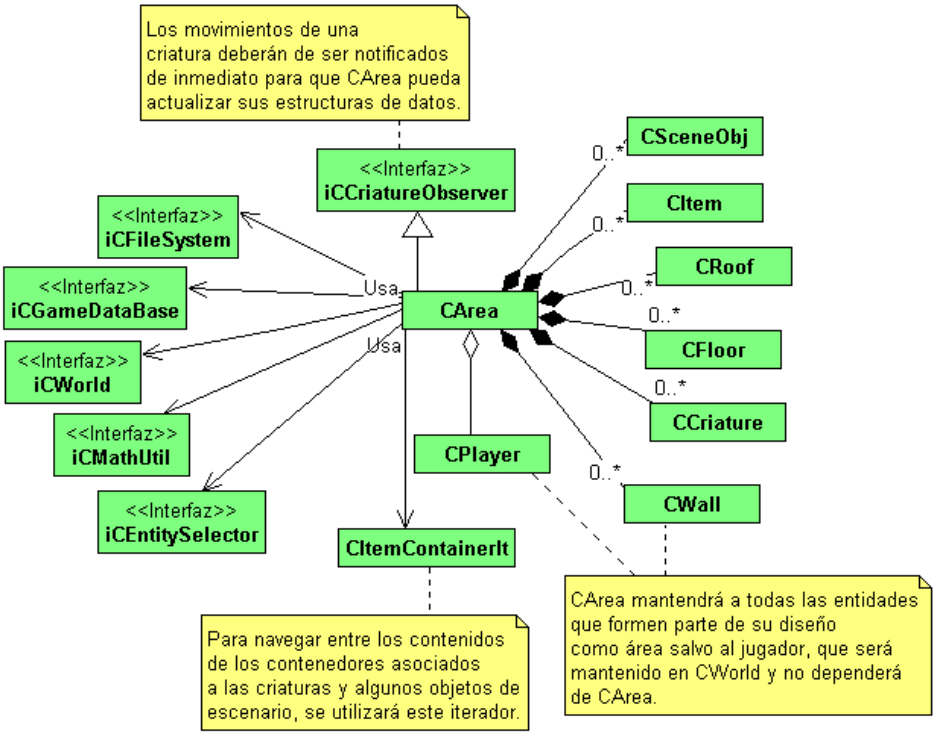


Diagrama de `CArea`.

Las instancias a las entidades `CItem` (ítems), `CCriature` (criaturas que no sean el jugador), `CSceneObj` (objetos de escenario), `CWall` (paredes) y `CRoof` (techo), serán mantenidas en árboles y serán accesibles a través de handles (tal y como se comentó en el apartado sobre entidades). Las instancias `CFloor` (suelos o tíles) se guardarán directamente en las posiciones del array que representen el escenario de juego. El jugador será pasado por `CWorld` cuando el área se active.

Todas las entidades podrán tener asociado un tag (etiqueta). Dicho valor se vinculará en el archivo de diseño del área y permitirá, a través de un identificador de cadena de caracteres, obtener el handle asociado a la entidad que representa. La utilidad de los tag's será relativa al trabajo con `CrisolScript`. Todos los tag's existentes serán mantenidos por `CArea`.

El dibujado de las entidades se realizará siempre en un orden estricto para evitar incoherencias. Cuando en una celda no exista una instancia `CFloor` (se dirá que no tiene contenido), ésta quedará invalidada y se mostrará como un simple "hueco" isométrico de color negro. Será imposible depositar entidad alguna sobre la misma.

La carga de un área se realizará desde un archivo binario. Las entidades también se inicializarán desde este archivo y `CArea` organizará la gestión del mismo y el proceso de obtención de la información. Todas las áreas serán referenciables vía un identificador numérico de tal forma que hablaremos de área 0, 1 o 232. Lo que sí será obligatorio es que los identificadores sean sucesivos (no aleatorios). Dichos identificadores serán utilizados para indicar la carga de determinadas áreas ya que el nombre de los archivos de área se hallarán automáticamente con sólo conocer el valor numérico de su identificador.

La posibilidades que tendrán las criaturas de moverse y cambiar de celda obligará a que `CArea` sea un "observador" de todas y cada una, con el fin de que una criatura solicite, de forma completamente "invisible", el cambio de localidad a `CArea` y ésta pueda actualizar sus estructuras de datos.

Para navegar por el contenido de las entidades contenedoras, `CArea` hará uso de la clase `CItemContainerIt`.

En la operación de carga de un área con el fin de sustituir la actual, la clase se encargará de mantener la información relativa al jugador y la de los ítems que éste transporte. El resto de información será volcada directamente a disco, a un fichero temporal, donde se almacenarán con todos sus estados. Sólo cuando se realice una operación de guardar partida, dichos datos almacenados en los ficheros temporales serán oficialmente guardados. Así mismo, cuando se retorne a un área, los datos serán cargados de los fichero temporales. Todas las operaciones de carga y almacenamiento serán gestionadas por `CArea`, pero el verdadero trabajo se realizará a nivel de instancias a las entidades.

En `CArea` se mantendrá la información relativa a la luz asociada a las entidades. También se gestionarán las actualizaciones de incidencia de la misma cuando una entidad que posee luz la modifique o cambie de posición.

El dibujado de elementos se llevará a cabo por parte de `CArea` pero siendo coordinado por `CIsoEngine`. `CArea` se encargará de dar las ordenes concretas a las diferentes instancias de las celdas que `CIsoEngine` ordene dibujar.

4.3.13.4. El motor isométrico.

Por motor isométrico entenderemos el subsistema, dependiente de `CWorld`, encargado de tratar los datos de `CArea` deacuerdo a una representación en formato isométrico. El motor isométrico estará formado, básicamente, por la clase `CIsoEngine` pero contará también con una instancia `CPathFinder` para el trabajo en el cálculo de caminos entre diversos puntos del área de juego. El diagrama del motor isométrico será el siguiente:

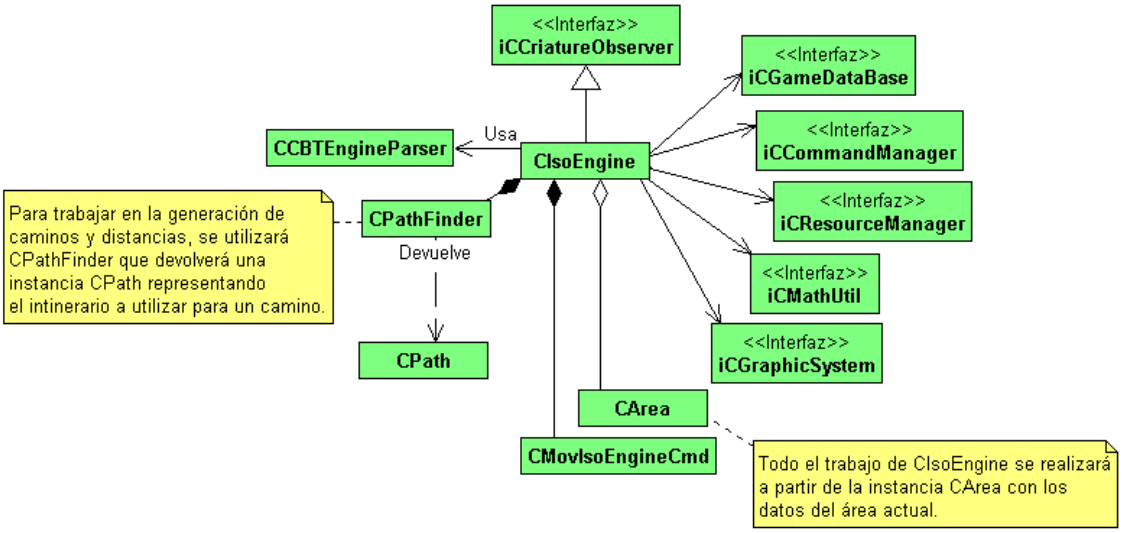


Diagrama del motor isométrico..

4.3.13.4.1 La clase CIsoEngine.

La clase `CIsoEngine` será el centro de trabajo del motor isométrico utilizado por *Crisol*. Necesitará heredar de `iCCriatureObserver` para seguir el rastro de la posible criatura que tenga la cámara asociada (por ejemplo, para comprobar si ha realizado un movimiento y se sitúa sobre una celda con pared con el fin de aplicar un efecto de transparencia sobre la misma).

Para el trabajo con el cálculo de caminos, `CIsoEngine` mantendrá una instancia a la clase `CPathFinder` que, a su vez, devolverá objetos `CPath` que serán los que contengan las trayectorias solicitadas. Estos objetos no serán más que contenedores de las posiciones que una criatura deberá de visitar para llegar desde una posición inicial a un destino. La instancia `CPathFinder` no será accesible directamente ya que `CIsoEngine` se encargará de ofrecer un interfaz unificado para el trabajo en el mismo de forma transparente a su cliente `CWorld`.

Para la gestión de los scrolles posibles en la zona visible del área actual, `CIsoEngine` mantendrá el comando `CMovIsoEngineCmd`. Este comando será el que trabaje con los objetos `CPath`.

4.3.13.4.2 La clase CPathFinder.

Esta clase, definida en el interior de `CIsoEngine`, será la encargada de implementar la lógica relativa a la búsqueda de caminos y distancias. La creación de una clase dedicada por entero a tal tarea se debe a que el algoritmo de búsqueda y los métodos de apoyo que pudiera tener serán mucho más actualizables desde una clase, un componente, a parte. El diagrama de la clase será muy sencillo:

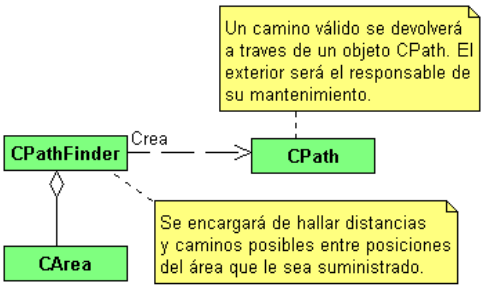


Diagrama de CPathFinder.

El algoritmo en que se basará `CPathFinder` para la localización de los caminos será el A* [Heyes'01] y [Stout'00]. Este algoritmo es el más ampliamente utilizado para el cálculo de trayectorias posibles en juegos que utilizan representaciones de mapa basados en celdas (también se usan derivaciones del mismo en motores con otros sistemas de representación). Es un algoritmo recursivo y heurístico, de tal forma que el éxito del mismo dependerá, en gran medida, en el acierto del heurístico utilizado. Es un algoritmo rápido, pero necesita de la creación de una abundante cantidad de nodos de información por lo que, en en el interior de `CPathFinder`, todos los nodos estarán creados en base a `CMemoryPool`. También se utilizará una instancia `CRecycleMemory` para incrementar aún más la velocidad de proceso de creación y destrucción de los mismos [Rabin'00a] y [Rabin'00b].

4.3.13.4.3 La clase CPath.

La clase `CPath` representará una trayectoria encontrada por `CPathFinder`. La clase no será más que una lista en cuyos nodos existirán las posiciones que se deberán de atravesar desde el origen en donde se solicitó la búsqueda del camino hasta el final. Las instancias `CPath` serán creadas por `CPathFinder` pero no mantenidas, es decir, quedará en responsabilidad del cliente que solictó la búsqueda de un camino el destruirlas. El cliente siempre será el comando de movimiento de criaturas.

4.3.13.5. Clases de apoyo.

Por clases de apoyo entenderemos todas aquellas que se usen dentro del contexto universo de juego y que permita realizar tareas auxiliares pero de carácter muy importante. El grueso de estas clases será el de iteradores y contenedores de handles mantenidos por `CArea`. Todas estas clases ya han sido citadas directa o indirectamente en los apartados anteriores.

4.3.13.5.1. Las clases CItemContainer y CItemContainerIt.

Estas clases servirán para mantener handles a ítems y para iterar por ellos respectivamente. La instancia `CItemContainer` será mantenida por dos clases. Por un lado `CCriature` para simular el inventario que toda criatura posea y, por otro, `CSceneObj` siempre y cuando el objeto de escenario sea contenedor de ítems (por ejemplo, un baúl sería contenedor de ítems pero una farola no). El diagrama de estas clases será el siguiente:

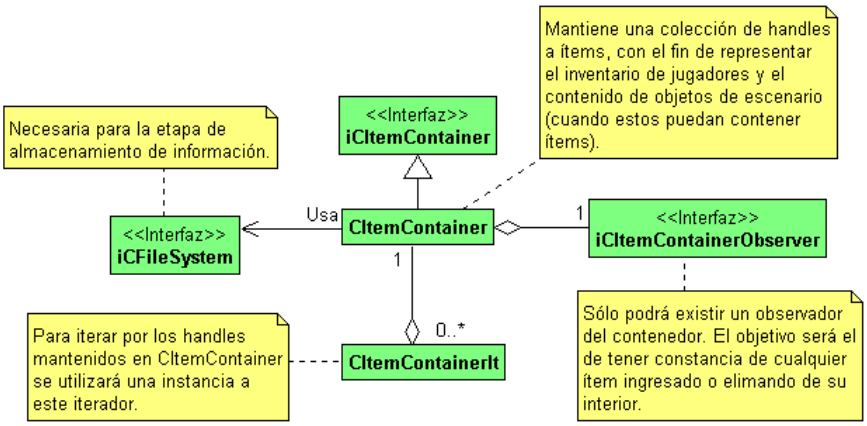


Diagrama de CItemContainer.

La clase `CItemContainer` podrá mantener una instancia a un observador de la misma, de tal forma que cuando se inserte un nuevo ítem o bien cuando se elimine, el observador reciba notificación al respecto. El uso de un observador será vital en el trabajo con las interfaces ya que éstas podrán reflejar gráficamente los sucesos lógicos de una forma elegante y transparente.

El uso de la instancia `iCFileSystem` se deberá a que este contenedor de handles será capaz de recibir la propagación de una petición de almacenamiento de partida, volcando a disco todo el contenido que posea relativo a sus ítems.

La clase `CItemContainerIt` representará un iterador a la clase. Simplemente ofrecerá métodos para iterar por entre cada uno de los handles mantenidos por la instancia.

Siempre que se desaloje un ítem de este contenedor, el mismo irá a parar a la celda sobre la que se halle el que era su dueño. Así, la acción de tirar un ítem por parte de una criatura, no será más que quitar el handle al mismo de este contenedor y depositar el ítem sobre el área de juego (en la misma celda en donde se halle la criatura).

4.3.13.5.2. Las clase CEquipmentSlot.

Esta clase servirá para representar los ítems que una criatura posea equipados en sus slots. En cierta forma, no deja de ser una clase encargada de contener handles a ítems, al igual de lo que ocurría con la anterior, `CItemContainer`. La principal salvedad será que sólo podrá ser poseida por las criaturas ya que sólo estas entidades serán capaces de equiparse ítems. El diagrama será el siguiente:

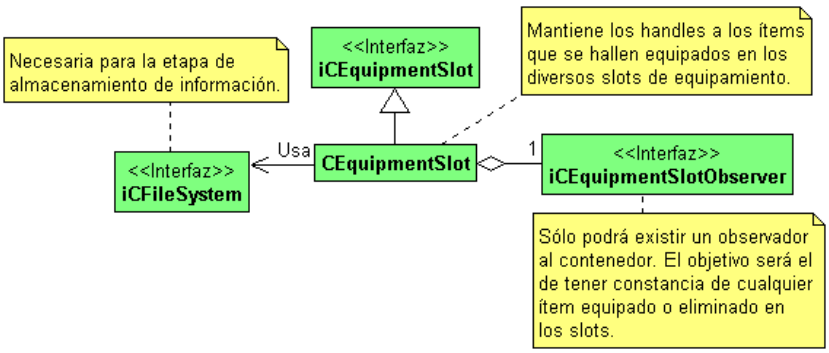


Diagrama de CEquipmentSlot.

Se podrá asociar un único observador a la instancia con el fin de que se puedan detectar de forma inmediata los cambios lógicos sobre la misma. Todo observador deberá de heredar de la interfaz `iCEquipmentSlotObserver`.

Siempre se cumplirá que un handle a ítem que se halle en un inventario, instancia `CItemContainer`, no podrá estar asociado en un slot de equipamiento, instancia `CEquipmentSlot`. Del mismo modo, cuando se desaloje un ítem equipado de esta instancia, éste siempre irá a parar al inventario de la criatura.

4.4.13.5.3. La clase CCellIterator.

Esta clase se utilizará para navegar por entre las entidades que se hallen sobre las celdas del área. La clase será muy utilizada desde el motor isométrico. El iterador ofrecerá handles a las entidades, en lugar de instancias concretas.

4.4.13.5.4. Iteración entre las entidades de un área.

La iteración entre las entidades de un área se realizará con iteradores que hereden de la clase parametrizada `CEntitiesIterator`. En *Crisol* solo hará falta iterar por las criaturas asociadas a un área, por lo que se implementa una clase llamada `CCriatureIterator` que implementará los métodos puros de `CEntitiesIterator`. El diagrama será el siguiente:

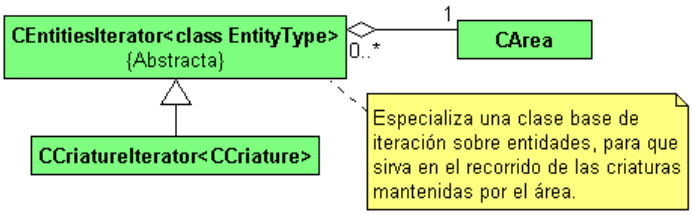


Diagrama CCriaturesIterator.

Como se puede apreciar en el diagrama, `CCriatureIterator` heredará de `CEntitiesIterator` indicando el tipo de entidad que representará. Al igual que otros iteradores, siempre se recogerán los handles a las instancias en lugar de los objetos concretos.

4.3.14. El manager de interfaz.

El manager de interfaz será uno de los subsistemas más grandes y complejos del motor, pues se encargará de gestionar los diferentes interfaces con los que el usuario (el jugador) podrá interactuar con *Crisol*.

Todas las ventanas de interfaz estarán construidas en base a componentes. Por ventana de interfaz entenderemos un menú, una ventana de edición, un cuadro de diálogo, etc y por componentes instancias a clases que representen botones, selectores de texto, imágenes de fondo, etc. De esta forma, una ventana de interfaz será un conjunto de componentes que responderá de una manera particular a los eventos que sobre ellos se produzcan. El diagrama del subsistema será el siguiente:

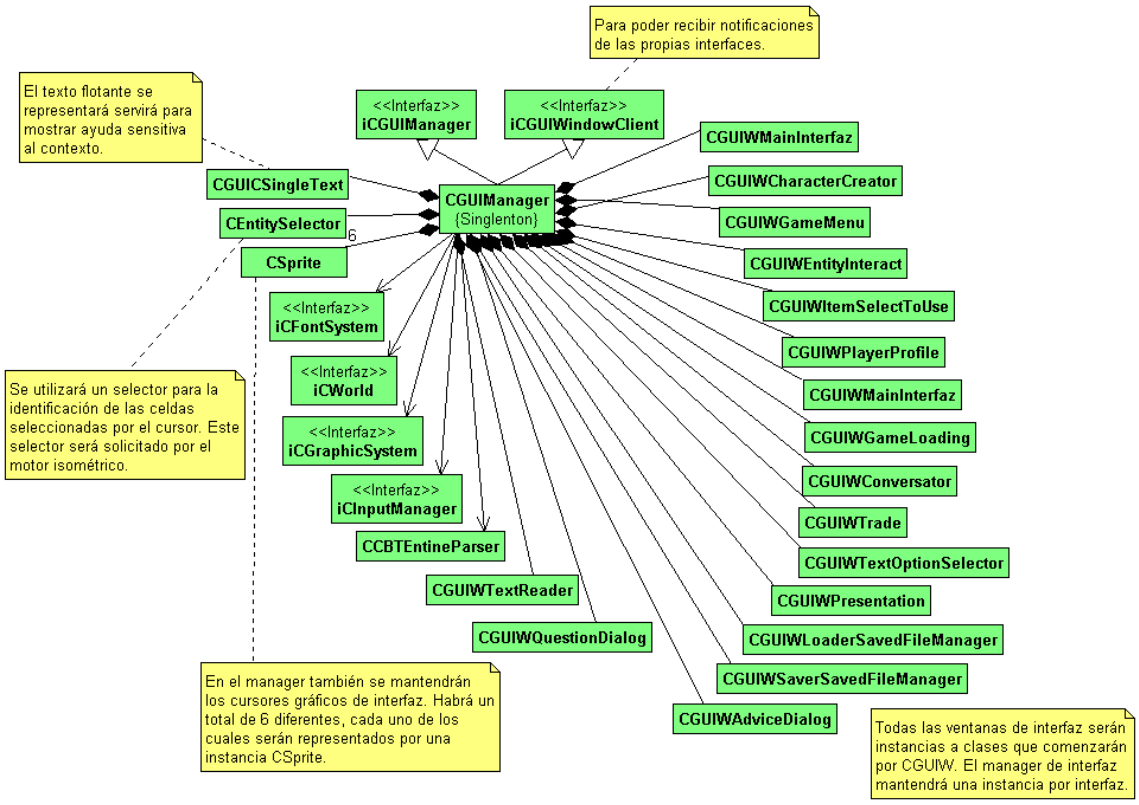


Diagrama del manager de interfaces.

Como se puede observar, la clase `CGUIManager` será el núcleo del interfaz. En ella se mantendrán las instancias de todas las ventanas de interfaz (clases que comienzan por `CGUIW`). A continuación se comentará en mayor profundidad dicha clase para pasar luego a explicar los componentes y, finalmente, las ventanas de interfaz propiamente dichas.

4.3.14.1. La clase CGUIManager.

Tal y como ya se ha comentado, esta clase será el núcleo del subsistema. Desde aquí se mantendrán todas las instancias a las ventanas de interfaz posible. Al actuar de facade, no ofrecerá dichas ventanas de interfaz al exterior, sino que dispondrá de una interfaz unificada para la activación selectiva de las mismas.

Para que el manager permita la activación de una ventana de interfaz, antes se deberán de cumplir unas condiciones elementales en cuanto al estado de ejecución en el que se encuentre el motor. Tales condiciones vendrán definidas por el grafo comentado en la etapa de diseño. En otras palabras, para poder activar una interfaz de juego se deberá de estar dentro del juego, jamás se podrá activar en, por ejemplo, la etapa de creación de personajes. Como excepción, existirán interfaces que podrán ser activadas desde cualquier sitio y podrán ser, además, usadas por el diseñador (el resto se establecerán de forma automática) desde *CrisolScript* para obtener información del usuario. Estas interfaces estarán encarnadas por las clases `CGUIWAdviceDialog`, `CGUIWQuestionDialog`, `CGUIWTextReader` y `CGUIWTextOptionsSelector`, y serán denominadas interfaces secundarias.

En cada ciclo de actualización, `CCRISOLEngine` ordenará un dibujado de las ventanas de interfaz activas en el manager y éste recorrerá una lista interna que mantendrá aquellas ventanas de interfaz que se encuentren activas, delegando en cada una de ellas el dibujado.

Además del mantenimiento y filtrado en la activación de las ventanas de interfaz, éste manager se encargará de trabajar también con los cursores de interfaz. Los cursores de interfaz servirán para representar la posición del ratón y las acciones que se puedan realizar (interacción con las ventanas de interfaz, movimiento del personaje controlado por el jugador, selección de entidades, etc). Todos los cursores serán representados mediante instancias `CSprite`.

El manager también mantendrá el selector de las celdas en las que el jugador podrá moverse. Dicho selector será utilizado por el motor isométrico para pasarselo a `CArea` cuando proceda ser dibujado en pantalla (se usará cuando el personaje controlado por el jugador desee moverse y el usuario, el jugador, seleccione destinos posibles).

El texto flotante será representado por el componente de interfaz `CGUICSingleText` (los componentes de interfaz serán estudiados más adelante). El texto flotante será información que se asociará a la posición en donde se halle el cursor y servirá para ofrecer ayuda breve al usuario de forma inmediata. Será utilizado de forma intensiva en el modo interacción, es decir, cuando el jugador examine las entidades que se hallen sobre el área, pero también en todas aquellas situaciones que requieran información complementaria (por ejemplo, en la ventana de interfaz relativa a la creación del personaje se usará para mostrar los puntos necesarios a la hora de incrementar un atributo o establecer una habilidad).

4.3.14.2. Los componentes de interfaz.

Los componentes de interfaz serán las clases que verdaderamente implementen la construcción de las ventanas de interfaz, pues todas ellas se crearán en base a ellos.

Todos los componentes heredarán de una misma clase abstracta denominada `CGUIComponent`, que incorporará una serie de mecanismos comunes. Los componentes estarán divididos por funcionalidad. Los habrá destinados a mostrar imágenes (`CGUICBitmap` y `CGUICAnimBitmap`), a trabajar con texto (`CGUICBlockText`, `CGUICSingleText`, `CGUICLineText` y `CGUICInput`), con botones gráficos (`CGUICButton`), barras de porcentaje (`CGUICPercentageBar`) y para realizar operaciones de selección sobre instancias `CSprite` exteriores (`CGUICSpriteSelector`). El diagrama general será el siguiente:

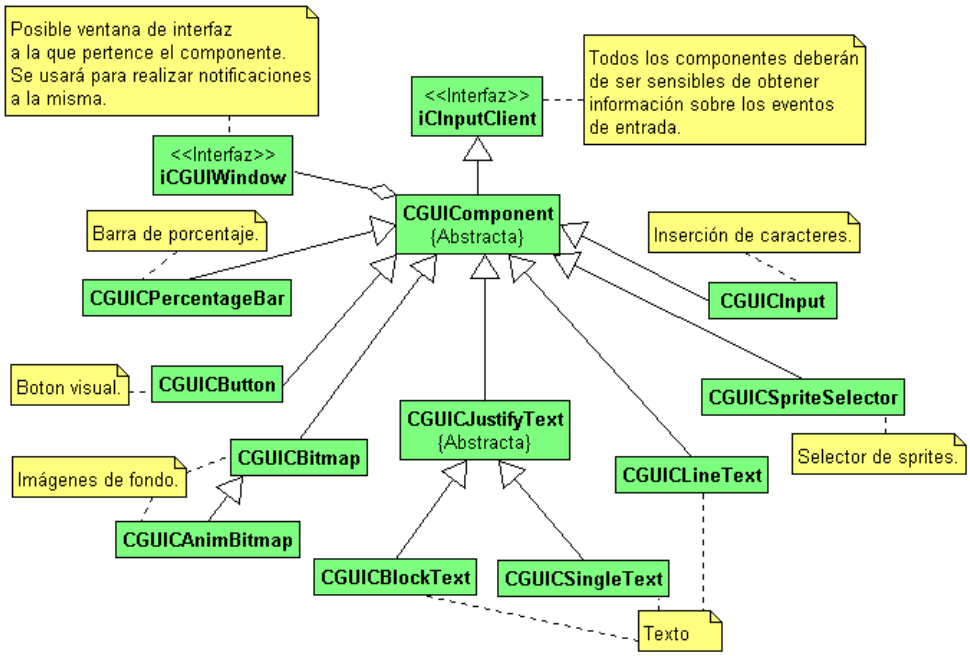


Diagrama de los componentes de interfaz.

Todos los componentes tendrán una inicialización compleja en cuanto a que deberán de conocer su identificador, posibles eventos de entrada a los que atender, valores de color, transparencia, etc. Para evitar trabajar con métodos excesivamente grandes y poco mantenibles, se irá creando una jerarquía de estructuras públicas que partirá de `CGUICComponent` y se irá especializando en cada uno de los componentes particulares. De esta forma, cualquier componente para ser inicializado solicitará una instancia a su estructura de inicialización (definida como pública en su interior).

A continuación se examinarán cada una de las clases mostradas en el diagrama anterior.

4.3.14.2.1. La clase CGUIComponent.

Esta será la clase base de todo componente. Tal y como se muestra en el diagrama, la clase heredará de `iCInputClient` debido a que todo componente podrá recibir eventos de los dispositivos de entrada.

Todos los componentes al inicializarse, propagarán en primer lugar el proceso a esta instancia pasando la estructura con toda la información de inicialización. En esta clase se recogerá dicha inicialización a través de la estructura base, con lo que `CGUIComponent` podrá inicializarse de forma correcta. La estructura base tendrá el siguiente aspecto:

```
struct sGUIComponentInitData {
  // Inicializacion de CGUIComponent
  // Datos
  GUIDefs::GUIIDComponent ID;               // ID de componente
  byte                    ubDrawPlane;      // Plano de dibujado
  SPosition               Position;         // Posicion de dibujado
  iCGUIWindow*            pGUIWindow;       // Ventana asociada
  GUIDefs::GUIEvents      GUIEvents;        // Eventos
  bool                    bClickWithSelect; // ¿Clicks con seleccion?
  // Constructor por defecto
  sGUIComponentInitData(void): ubDrawPlane(0),
                               ID(0),
                               pGUIWindow(NULL),
                               GUIEvents(GUIDefs::GUIC_NO_EVENT),
                               bClickWithSelect(true) { }
};
```

Tal y como se puede observar, cuando a esta clase llega la ejecución del método de inicialización, se provee de un identificador para el componente, de un plano y posición de dibujado, así como como de información sobre la ventana de interfaz a la que estará asociada y los eventos por los que se escuchará. En caso de no pasar ventana de interfaz asociada, el componente no responderá a ningún evento de entrada.

Los posibles eventos a los que podrá atender un componente serán siempre relativos al espacio “físico” que ocupen en pantalla. Para conocer dicho espacio físico, además de la posición pasada en la estructura anterior, se deberá de conocer la anchura y altura del componente. Estos valores serán obtenidos mediante unos métodos virtuales puros que los componentes concretos deberán de implementar. Conociendo el espacio físico ocupado, los eventos que se atenderán serán los relativos al posicionamiento del cursor sobre dicho espacio y la salida del mismo, así como a la pulsación de los botones de ratón (estando el cursor en el espacio físico ocupado por el componente).

Los eventos serán atendidos por los componentes sólo en caso de que tengan asociada una ventana de interfaz a la que notificarlos. De esta forma, cuando suceda un evento de entrada, los componentes avisarán a la ventana de interfaz mediante el pase del código del evento que se ha producido y su propio identificador. Gracias al identificador, las ventanas de interfaz conocerán el componente concreto que ha producido el evento.

Desde el exterior, se podrán activar y desactivar los componentes. La activación y desactivación permitirá anular el dibujado de los mismos así como el envío por parte de éstos de eventos de entrada. Independientemente de la activación y desactivación, también se podrá bloquear la entrada, de tal forma que aunque sean dibujados en pantalla, no notifiquen los posibles eventos que se produzcan en ellos.

4.3.14.2.2. Trabajo con las imágenes de fondo.

Las imágenes de fondo representarán componentes que tendrán asociados una imagen o bien imágenes y animaciones estáticas (aquellas que no podrán cambiar de posición). La utilización fundamental de este tipo de componentes será la de funcionar como fondos de las ventanas de interfaz.

4.3.14.2.2.1. La clase CGUICBitmap.

Esta clase representará el componente de imagen más básico. Únicamente encapsulará una instancia `CBitmap` y atenderá posibles eventos que se sucedan en la región ocupada por ella. En todos los casos de uso en *Crisol*, no se aprovechará las posibilidades de entrada y únicamente se utilizará para que los cuadros de diálogo tengan asociados imágenes de fondo. El diagrama de la clase será muy sencillo:

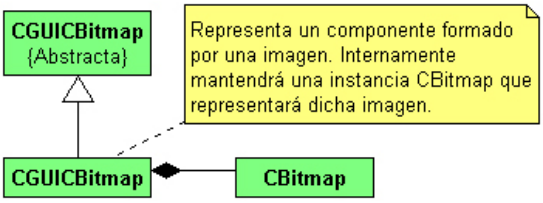


Diagrama de CGUICBitmap.

4.3.14.2.2.2. La clase CGUICAnimBitmap.

Se suscribe todo lo comentado en la clase `CGUICBitmap` pues ésta hereda de la misma. La única novedad es que podrá tener asociada animaciones estáticas. Dichas animaciones serán instancias `CSprite` que se registrarán en una lista. Al recibirse la orden de dibujado del componente, primero se dibujará la imagen asociada y a continuación las animaciones estáticas. Éstas últimas no incorporarán la posibilidad de recibir eventos de entrada. El diagrama de la clase será el siguiente:

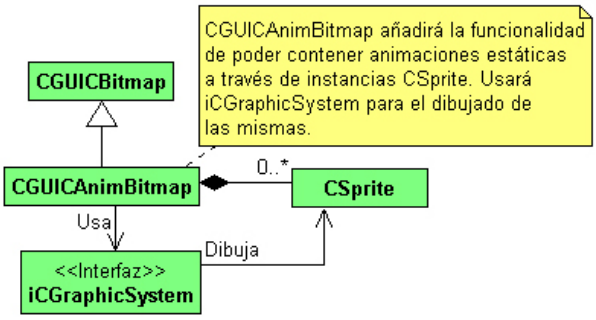


Diagrama de CGUICAnimBitmap.

4.3.14.2.3. La clase CGUICButton.

Estos componentes representarán un botón. Por botón entenderemos una representación gráfica con dos estados; seleccionado y no seleccionado. Internamente, el botón se representará con una instancia `CSprite` que tendrá asociada una plantilla de animación con al menos dos estados. El estado 0 será el utilizado para indicar que el botón se halla no seleccionado y el estado 1 para indicar que está seleccionado. El diagrama será el siguiente:

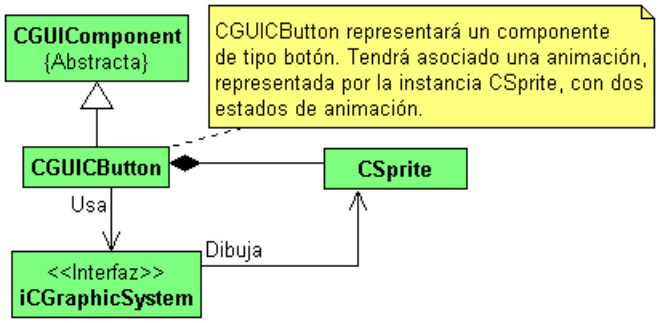


Diagrama de CGUICButton.

La selección / deselección jamás será activada internamente sino que existirán los métodos apropiados para que desde el exterior se establezca según proceda (por ejemplo, cuando una ventana de interfaz notifique un evento de pulsación del botón izquierdo sobre el componente).

4.3.14.2.4. La clase CGUICSpriteSelector.

Este componente será fundamental para el trabajo con las interfaces que permitan gestionar ítems así como las destinadas a mostrar retratos, etc. Básicamente será como un componente `CGUICButton` salvo que la instancia `CSprite` que mantengan será pasada desde el exterior. De esta forma, por ejemplo, se podrán implementar los componentes que muestren lo que el personaje lleve colocado en los slots de equipamiento (se pasarán las instancias `CItem` que serán recogidas a través de su instancia `CSprite`).

Además del `CSprite` pasado desde el exterior, este componente mantendrá dos instancias `CFXBack` (ver el apartado sobre objetos dibujables en el subsistema gráfico) para ayudar a distinguirlo y clarificarlo. Una de esas instancias será usada cuando el componente no esté seleccionado y la otra para cuando sí lo esté.

El diagrama del componente será el siguiente:

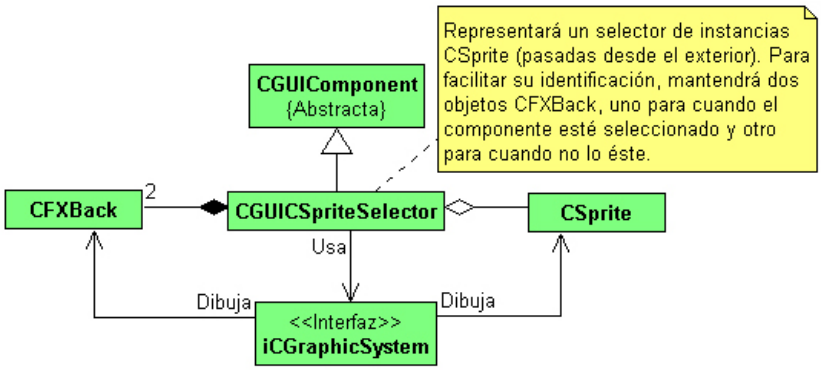


Diagrama de CGUICSpriteSelector.

El componente podrá dibujarse aún sin que exista ningúna instancia `CSprite` asociado al mismo. En este caso, el componente se podrá haber configurado, en su inicialización, para que la selección sea, o no, posible.

4.3.14.2.5. La clase CGUICInput.

Este componente servirá para recoger la entrada del usuario pero cuando éste se ponga a teclear. Trabajará por su cuenta con el manager de entrada para que le sea suministrado los caracteres pulsados por el usuario.

Desde el punto de vista de la configuración, se podrá indicar el color del texto, tanto cuando éste se halle seleccionado como sin seleccionar, así como el número máximo de caracteres que podrán tener las cadenas que escriba el usuario.

Para mejorar el aspecto visual, se incorporará el efecto de cursor intermitente justo después del primer carácter escrito. Su implementación tan sólo requerirá que el componente sea un cliente `iCAlarmClient` para que el subsistem de alarmas le avise cada un determinado número de milisegudos. Esto permitirá que de forma alternativa dibuje el carácter que haga las veces de cursor en pantalla.

El diagrama será el siguiente:

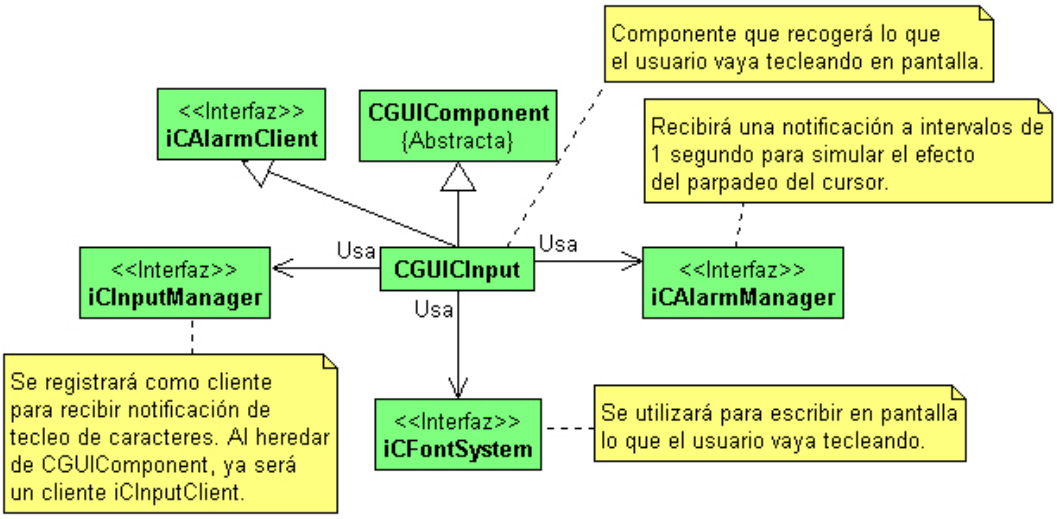


Diagrama de CGUICInput.

El uso del subsistema de fuentes (interfaz `iCFontSystem`) será esencial para escribir en pantalla lo que el usuario vaya tecleando.

El componente será capaz de almacenar lo que el usuario haya tecleado una vez que sea desactivado así como ser capaz de localizar el cursor de escritura después del último carácter cuando vuelva a ser activado.

4.3.14.2.6. La clase CGUICLineText.

Representará una línea simple de texto. A diferencia del texto complejo, no se permitirá formatear el texto mostrado. El uso de este componente será para aquellos casos en los que se quiera señalizar algo de forma escueta y simple, nunca para mostrar cantidades medias / altas de texto en pantalla. El diagrama será muy sencillo:

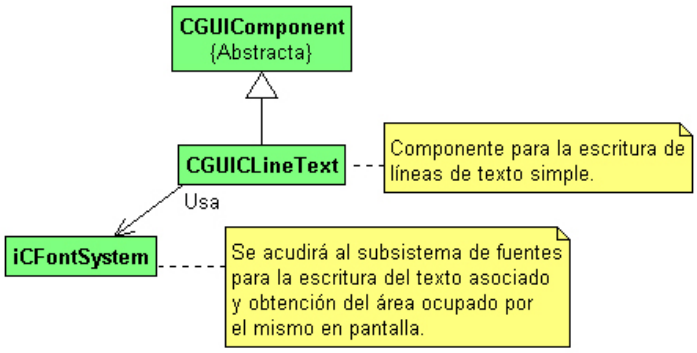


Diagrama de CGUICLineText.

El uso del subsistema de fuentes será esencial para escribir el texto contenido en pantalla por cada orden `Draw` que le llegue al componente desde el manager de interfaz.

4.3.14.2.7. Trabajo con texto complejo.

Por texto complejo se entenderá aquel que pueda ser mostrado en varias líneas diferentes, además de justificado hacia el lado derecho, el izquierdo o bien al centro. Los textos complejos podrán ser mostrados a bloques, es decir, de *n* líneas en *n* líneas, o bien, mostrados de forma completa sin importar el número de líneas que los componen. En el primero de los casos, los textos mostrados a bloques, se utilizará la clase `CGUICBlockText`, mientras que en el segundo caso, los textos mostrados de una sola vez, la clase `CGUICSingleText`. Ambas clases, a su vez, heredarán de `CGUICJustifyText`, base que incorporará, principalmente, métodos destinados al justificado del texto. El diagrama será muy sencillo:

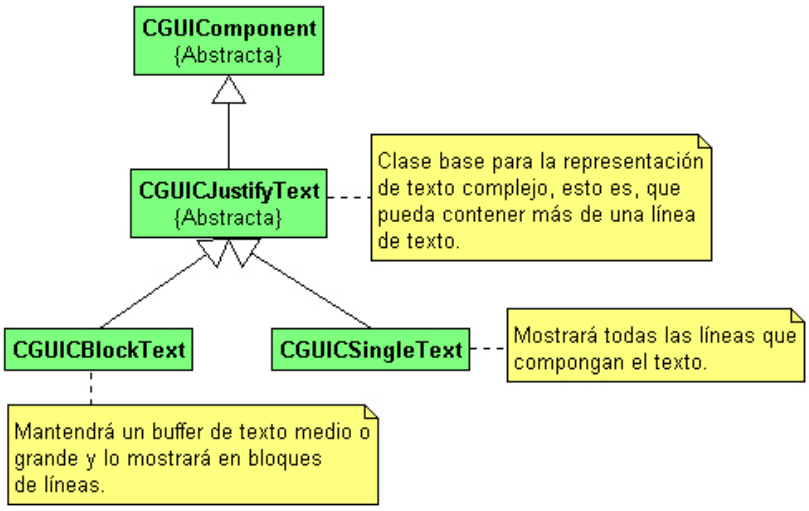


Diagrama de las clases para el tratamiento del texto complejo.

A continuación se analizará cada una de las clases del diagrama.

4.3.14.2.7.1. La clase CGUICJustifyText.

Como ya se comentó en el apartado anterior, `CGUICJustifyText` será la clase base para todos aquellos textos catalogados de complejos. En ella se implementarán los métodos destinados a la justificación de texto que tanta falta harán a las clases `CGUICBlockText` y `CGUICSingleText`. Sin embargo, también se encargará de mantener información común sobre la configuración de los textos que las anteriores clases mantendrán, como el color de selección y deselección, tipo de fuente o valor de transparencia. El diagrama de la clase será muy sencillo:

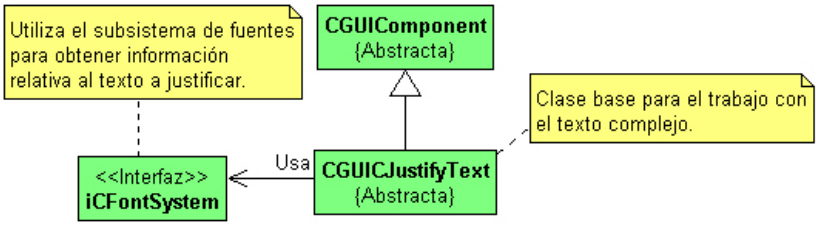


Diagrama de CGUICJustifyText.

El subsistema de fuentes (`iCFontSystem`) será necesario para obtener dimensiones relacionadas con las cadenas que se deban de justificar.

4.3.14.2.7.2. La clase CGUICBlockText.

Esta clase alojará un buffer de texto arbitrariamente grande. El buffer quedará dividido en líneas. Dicha división vendrá establecida por la configuración, pues en ella se especificará el máximo número de pixels que deberá de ocupar cada línea a mostrar. De igual modo, se deberá de indicar el número máximo de líneas que el componente será capaz de dibujar en la pantalla. El diagrama de la clase será el siguiente:

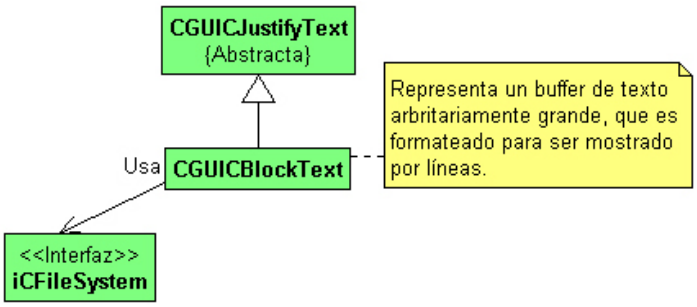


Diagrama de CGUICBlockText.

La consola de juego y la interfaz destinada a la visión de un archivo de texto (`CGUIWTextReader`) utilizarán una instancia a esta clase. Para la navegación por entre el bloque de texto, la clase dispondrá de dos métodos destinados a avanzar y retroceder línea a línea.

El uso del subsistema de fuentes seguirá siendo vital para la escritura de las líneas visibles en pantalla en un momento dado.

4.3.14.2.7.3. La clase CGUICSingleText.

Este componente vendrá a ser una mejora sobre `CGUICLineText`. Al igual que éste, deberá de mostrar en pantalla todo el texto que tiene asociado sin embargo, incorporará la posibilidad de utilizar una línea, o más, y podrá justificarse sobre un área virtual asociada. De esta forma, el texto resultará idóneo, por ejemplo, para mostrar las líneas habladas por una entidad o el texto seleccionable desde un cuadro de diálogo. Además, tendrá la posibilidad de aplicársele efectos especiales de fade, con lo que también será usado para mostrar el texto flotante de ayuda. El diagrama del componente será como sigue:

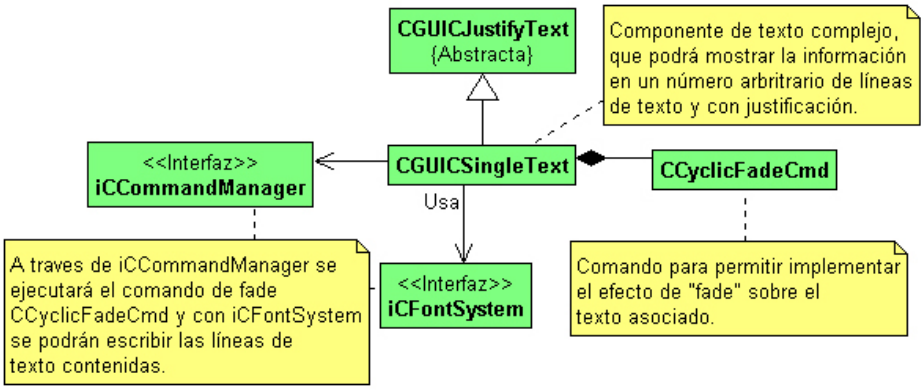


Diagrama de CGUICSingleText.

4.3.14.3. Las ventanas de interfaz.

Las ventanas de interfaz agruparán los componentes comentados en el apartado anterior, con el fin de formar agrupaciones lógicas destinadas a la gestión de un determinado menú, cuadro de diálogo, de selección, etc. Así, por ejemplo, el menú principal que aparece nada más ejecutar *Crisol* no será más que una ventana de interfaz formada por un componente `CGUICAnimBitmap`, para representar la imagen de fondo con posibilidad de tener asociada animación estática más luego una instancia `CGUICButton` por cada botón de opción posible.

El número de ventanas de interfaz será considerablemente elevado, principalmente por la cantidad de ellas que hacen acto de presencia en el estado de juego. Todas las interfaces heredarán de una clase abstracta denominada `CGUIWindow` la cual, a su vez, heredará de una interfaz `iCGUIWindow`. Esta interfaz será la pasada a los componentes anteriormente comentados, para notificar los eventos que sobre ellos se produzcan.

Las ventanas de interfaz estarán divididas en dos grupos fundamentales; las interfaces principales y las secundarias. En las primeras tendremos las que representarán situaciones dentro del grafo de estados de *Crisol*, mientras que en el segundo caso poseeremos cuadros de diálogo y menús de apoyo que podrán ser lanzados en cualquier instante.

A continuación se hará un repaso por todas las interfaces existentes, deteniéndonos antes a comentar las clases base.

4.3.14.3.1. Clases base utilizadas.

La elaboración de las clases que simulen las distintas ventanas de interfaz requerirán, una reutilización en componentes y funcionalidades importante. Es por ello que serán creadas un conjunto de clases base destinadas a tal fin. La más importante de todas será `CGUIWindow` y todas las demás clases que tengan que ver con la creación de una ventana de interfaz, heredará de ella. Después de esta clase existirán otras que permitirán a las ventanas de interfaz trabajar con la gestión de textos como opciones o contenedores de ítems. En todos estos casos, se seguirá heredando de `CGUIWindow`.

A continuación se comentará cada una de estas clases base.

4.3.14.3.1.1. La clase CGUIWindow.

Esta será la clase base de la que hereden todas las ventanas de interfaz. En ella se establecerá el funcionamiento general para las mismas. La clase base dispondrá de una interfaz `iCGUIWindow` que será pasada a los componentes de interfaz para que éstos puedan notificar el suceso de cualquier evento que se haya producido. El diagrama de la clase será el siguiente:

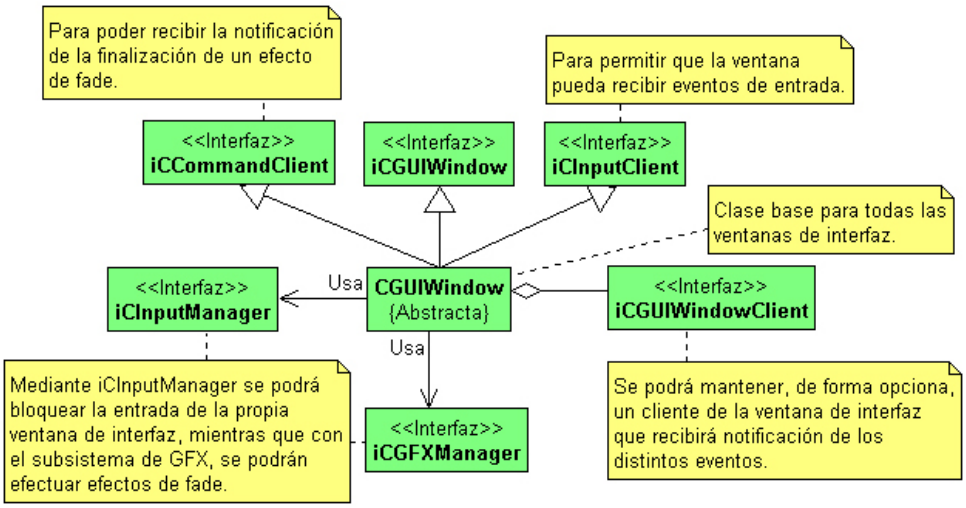


Diagrama de CGUIWindow.

Como se puede observar, la clase también heredará de `iCInputClient` y `iCCommandManager`. La herencia de `iCInputClient` se deberá a que a toda ventana se le dará la oportunidad de poder escuchar eventos de entrada por su cuenta (independientemente de que los componentes que tengan ya lo hagan). Esto será necesario porque algunas interfaces podrán querer utilizar las teclas como atajos (por ejemplo, la salida de un menú mediante la pulsación de `ESC`). Por otro lado, la herencia de `iCCommandManager` se deberá a que desde esta clase se podrá gestionar el trabajo de fade - in y fade - out, necesitándose recibir notificación de cuando cualquiera de los dos posibles fades termine a través de `CGFXManager`.

Toda ventana de interfaz podrá tener clientes asociados, esto es, observadores. Dichos clientes podrán recibir notificación de un determinado evento (por ejemplo, cierre de una ventana de interfaz). Para que una clase pueda instalarse como cliente de una determinada

ventana de interfaz, deberá de heredar de `iCGUIWindowClient` y pasarse en el método de activación de la misma. Una ventana sólo podrá tener asociado un cliente de cada vez.

En cierta forma, una ventana de interfaz también actuará de facade sobre todos los componentes que tenga asociados de ahí que desde `CGUIWindow` se implementen métodos virtuales puros destinados al bloqueo de la entrada y a la activación y desactivación de la misma. Estos métodos supondrán una actuación sobre todos y cada uno de los componentes de la ventana de interfaz (que dispondrán, a su vez, de métodos análogos) y de la propia ventana (por ejemplo, el bloqueo de entrada provocará un bloqueo tanto en los componentes de interfaz como en la propia ventana).

4.3.14.3.1.2. CGUIWBaseItemSelector.

Esta clase encapsulará la lógica para el trabajo con un contenedor de ítems, esto es, una interfaz `iCItemContainer` asociada a una criatura o bien a un objeto de escenario contenedor. El diagrama de la clase será el siguiente:

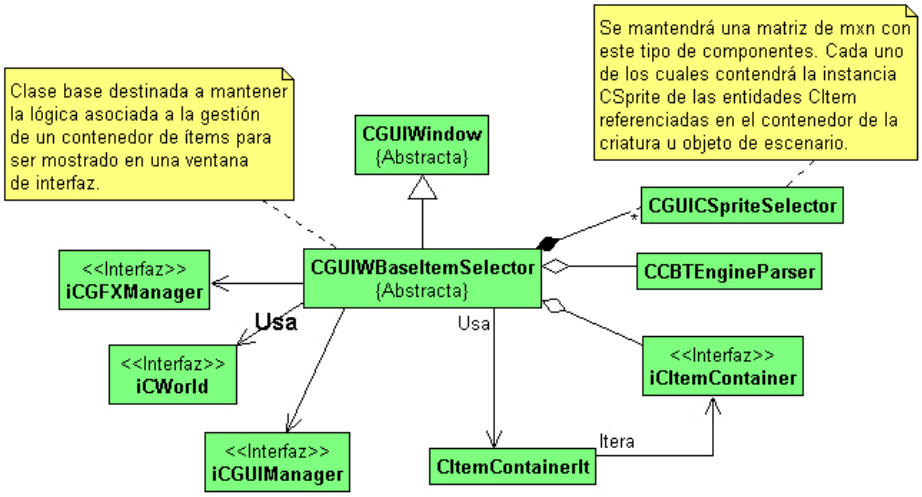


Diagrama de CGUIWBaseItemSelector.

La clase base mantendrán una matriz lógica de *mxn* en cuyos nodos se alojarán componentes `CGUICSpriteSelector`. Dichos componentes tendrán asociados las instancias `CItem` referenciadas en el contenedor `iCItemContainer` que recibió la ventana de interfaz al activarse. Durante el proceso de dibujado, la ventana sólo llevará a pantalla aquel número de filas definido en la matriz que se configuraron como visualizables.

Habrá que tener en cuenta un hecho sumamente importante y es que, si las instancias `CItem` poseen un gráfico de GFX asociado, éste deberá de dibujarse en pantalla, de ahí el uso fundamental de la interfaz `iCGFXManager` en el subsistema de FX.

La configuración de la clase se leerá a través de una instancia `CCBTEngineParser` que será propagada desde las derivadas.

4.3.14.3.1.3. CGUIWBaseWindowContainer.

Esta clase heredará de `CGUIWBaseItemSelector` y servirá para configurar una ventana de interfaz elemental de cara a mostrar la lógica implementada en `CGUIWBaseItemSelector`. En otras palabras, contendrá un componente para la imagen de fondo de la ventana y dos botones para navegar entre el contenido de la misma. El diagrama será el siguiente:

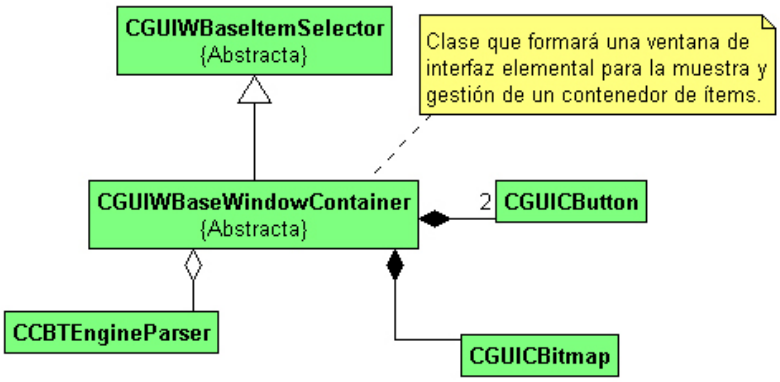


Diagrama de CGUIWBaseWindowContainer.

Como se puede observar desde el diagrama, existirá un componente que representará la imagen de fondo de la ventana de interfaz (`CGUICBitmap`), así como dos botones (`CGUICButton`), uno para subir una fila de ítems hacia arriba y otro para bajarla.

Esta clase será heredada por las ventanas de interfaz encargadas de la selección de un ítem para su uso sobre otra entidad así como las destinadas a mantener los contenedores del jugador y otra entidad para el intercambio.

La configuración se leerá utilizando una instancia `CCBTEngineParser` recibida por propagación de la activación de la interfaz.

4.3.14.3.1.4. CGUIWBaseTextSelector.

Esta clase se encargará de gestionar la lógica de aquellas ventanas de interfaz que tengan que trabajar con líneas de texto sensibles de ser seleccionadas (por ejemplo, la ventana de interfaz destinada a la gestión de los diálogos del jugador). He aquí el diagrama:

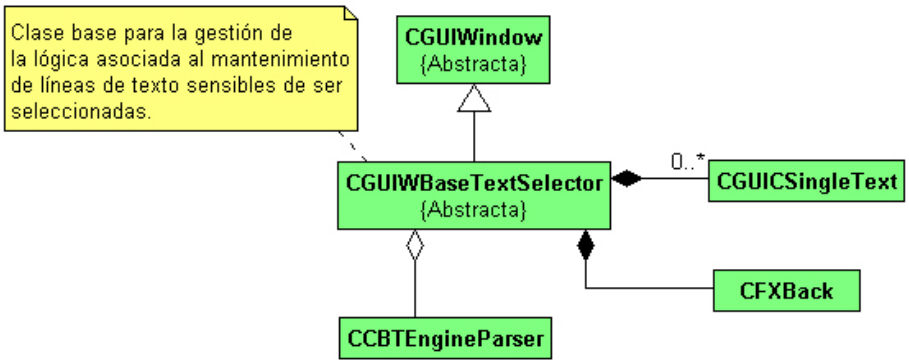


Diagrama CGUIWBaseTextSelector.

Tal y como puede observarse en el diagrama, la clase se encargará de mantener una lista de componentes `CGUICSingleText` igual al número de líneas de texto seleccionables. Para mejorar el aspecto de las líneas de texto en pantalla, se utilizará un componente `CFXBack` que irá por debajo de las mismas.

La configuración de la clase se realizará utilizando una instancia `CCBTEngineParser` que se recibirá por propagación en la activación de las clases derivadas.

4.3.14.3.2. Ventanas de interfaces principal.

Las ventanas de interfaz principal serán aquellas utilizadas para gestionar situaciones concretas del grafo de ejecución de *Crisol*. Siempre se activarán de forma automática o debido a acciones del jugador. A esta categoría pertenecerá el menú de juego, el de edición de

personaje, los destinados a grabar y recuperar partidas y todos los asociados a la ejecución de un juego (interfaz principal, inventario del jugador, registro de contenedores, diálogo, etc).

A continuación se analizará cada una de las clases encargadas de representar los distintos interfaces.

4.3.14.3.2.1. La clase CGUIWMainMenu.

Representará la ventana de interfaz encargada de mantener el menú de principal de juego (el que aparecerá nada más arrancar *Crisol*). Desde ella se podrá comenzar una nueva partida, recuperar una previamente almacenada, observar los créditos de juego y salir. El diagrama será el siguiente:

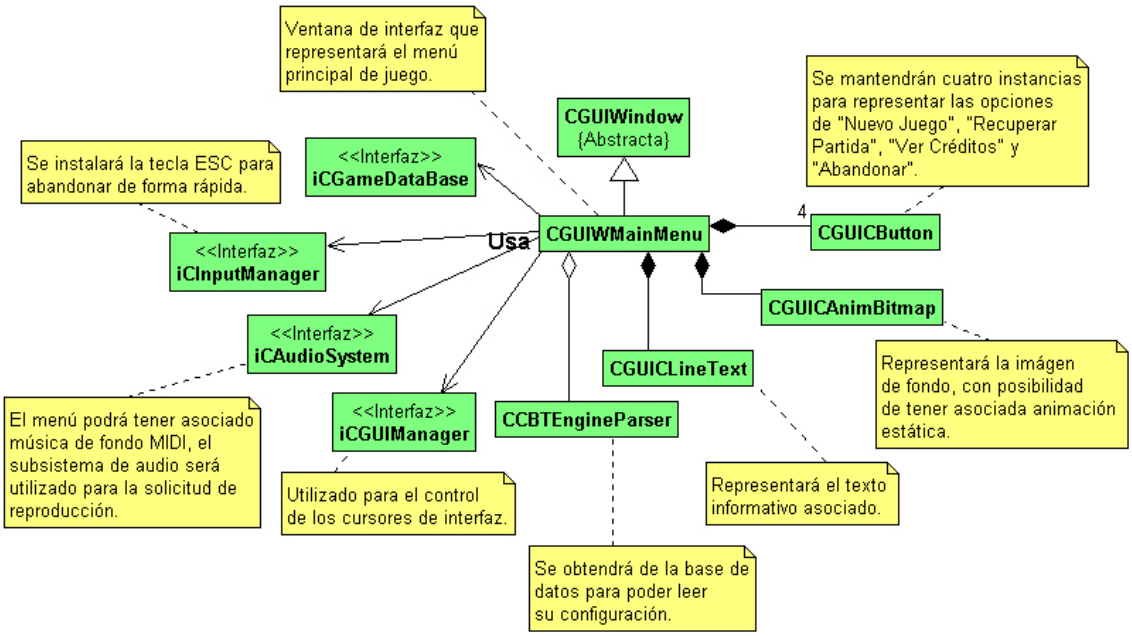


Diagrama de CGUIWMainMenu.

El mantenimiento de las opciones de menú se realizará utilizando cuatro instancias `CGUICButton` que, como se indica en el diagrama, servirán para representar las opciones de "*Nuevo Juego*", "*Recuperar Partida*", "*Ver Créditos*" y "*Abandonar*". La muestra del texto informativo adicional se realizará con un componente de texto simple `CGUICLineText`. Por último, la animación de fondo (que se esperará ocupe toda la pantalla) será encapsulada utilizando una instancia `CGUICAnimBitmap`.

Todas las configuraciones asociadas a los componentes se leerán del archivo de configuración de interfaz a través de una instancia `CCBTEngineParser` obtenida de la base de datos del juego (`iCGameDataBase`).

El uso del subsistema de audio (`iCAudioSystem`) se deberá a que la ventana de interfaz podrá tener asociada una música MIDI de fondo, mientras que la interfaz al manager de entrada (`iCInputManager`) se utilizará para poder instalar el uso de la tecla `ESC` asociada a la salida rápida del juego (será equivalente a pulsar sobre la opción de "*Abandonar*").

Al ser los cursores mantenidos por el manager de interfaz, el uso de `iCGUIManager` se centrará en la activación / ocultación de los mismos ante la elección de determinadas opciones.

4.3.14.3.2.2. La clase CGUIWCharacterCreator.

Desde esta ventana de interfaz se llevará el trabajo asociado a la edición del personaje que será controlado por el jugador. Su activación siempre procederá de la selección, en el menú de

juego (`CGUIWMainMenu`), de la opción de "*Nuevo Juego*". Esta ventana de interfaz trabajará en función de las definiciones que halle en el archivo de configuración del perfil de usuario.

El diagrama será el siguiente:

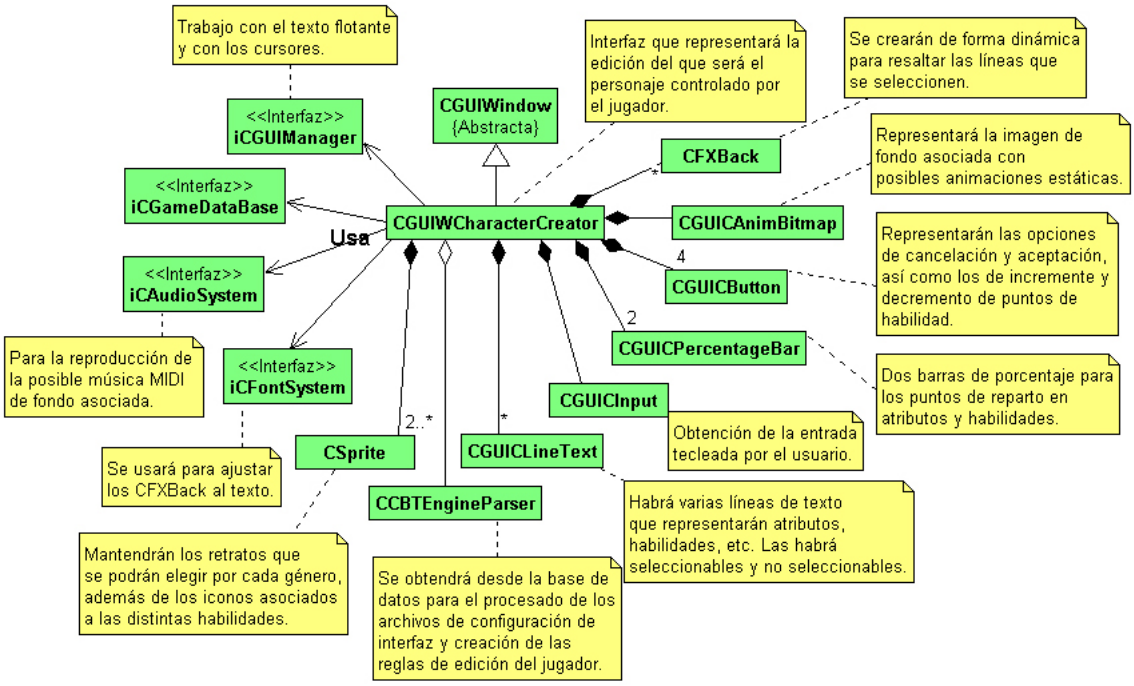


Diagrama de CGUIWCharacterCreator.

Como se puede observar, esta ventana de interfaz será una de las más complejas, con permiso del inventario del jugador (`CGUIWPlayerProfile`) y la interfaz de juego principal (`CGUIWMainInterfaz`). En el diagrama se distinguirán primero los componentes asociados a la ventana.

Por un lado tendremos los botones, que serán en total 4 y estarán destinados a representar la aceptación y cancelación del proceso de edición así como el incremento y decremento en los puntos asociados a las habilidades y atributos. La aceptación sólo se llevará a cabo si el perfil editado de jugador, se ajusta a las especificaciones del archivo de perfiles de jugador (y además, se ha insertado un nombre).

El componente más utilizado de todos será `CGUICLineText`, pues habrá mucha información a mostrar en forma de línea. Además, alguno de estos componentes actuarán como si de botones se tratarán pues se instalarán con el evento de selección y su pulsación supondrá actos como el comienzo de la introducción del nombre del jugador, el cambio de género, de tipo, clase, etc. Para resaltar texto seleccionado se usarán componentes `CFXBack` (el uso del subsistema de fuentes tendrá como objetivo hallar las dimensiones de las cadenas mantenidas en `CGUICLineText` para poder, así, ajustar correctamente los componentes `CFXBack`).

La obtención del nombre del jugador se realizará mediante un componente `CGUICInput` y las barras de porcentaje se utilizarán para representar los puntos de reparto asociados a las habilidades y atributos. La imagen de fondo podrá tener asociada animación estática, por lo que será representada por `CGUICAnimBitmap`.

El uso de instancias `CSprite` servirá para representar los distintos retratos elegibles por parte del usuario, así como para mostrar los iconos asociados a las habilidades.

Las interfaces a subsistemas no comentadas aún serán `iCAudioSystem`, para solicitar la reproducción del posible fichero MIDI asociado e `iCGUIManager` para la gestión de los

cursores y texto flotante de ayuda (se mostrarán los puntos que costará incrementar un atributo o escoger una habilidad)

4.3.14.3.2.3. La clase CGUIWGameLoading.

Esta ventana de interfaz será muy sencilla. Se utilizará siempre que se proceda a llevar a memoria los datos para comenzar una nueva partida o bien para recuperarla. El diagrama será el siguiente:

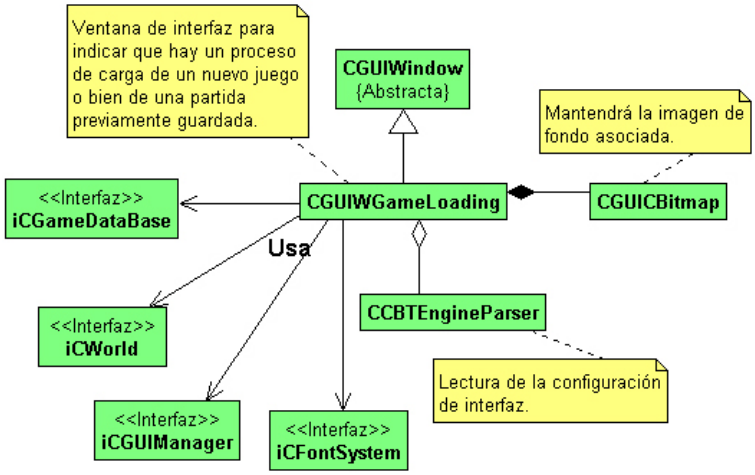


Diagrama de CGUIWGameLoading.

Como se puede observar, únicamente constará de una imagen de fondo sin posibilidad de tener animación estática. Esto es asi porque mientras se proceda a cargar la información, *Crisol* no tendrá la posibilidad de actualizar la pantalla ni el subsistema de comandos.

El trabajo con la base de datos será en dos sentidos. Por un lado para obtener el parser que permita leer la configuración de la interfaz y, por otro, para obtener el texto estático que se mostrará como mensaje de carga encima de la imagen de fondo. Dicho texto se colocará utilizando el subsistema de fuentes `iCFontSystem`.

Naturalmente, el interfaz se deberá de poner en contacto con `CWorld`, a través de `iCWorld`, para indicar que se quiere cargar un juego previamente guardado o bien que se desea comenzar una nueva partida. Cuando el universo de juego finalice el proceso de carga, la ventana de interfaz realizará un fade - out que, una vez acabado, mostrará el universo de juego preparado para que pueda ser jugado.

4.3.14.3.2.4. La clase CGUIWGameMenu.

Esta ventana de interfaz se encargará de representar las opciones generales disponibles durante el juego. Estas opciones serán relativas a la carga de otra partida, a guardar la partida actual, a continuar o bien a abandonar. Su activación supondrá dejar en estado de pausa el juego. El diagrama será el siguiente:

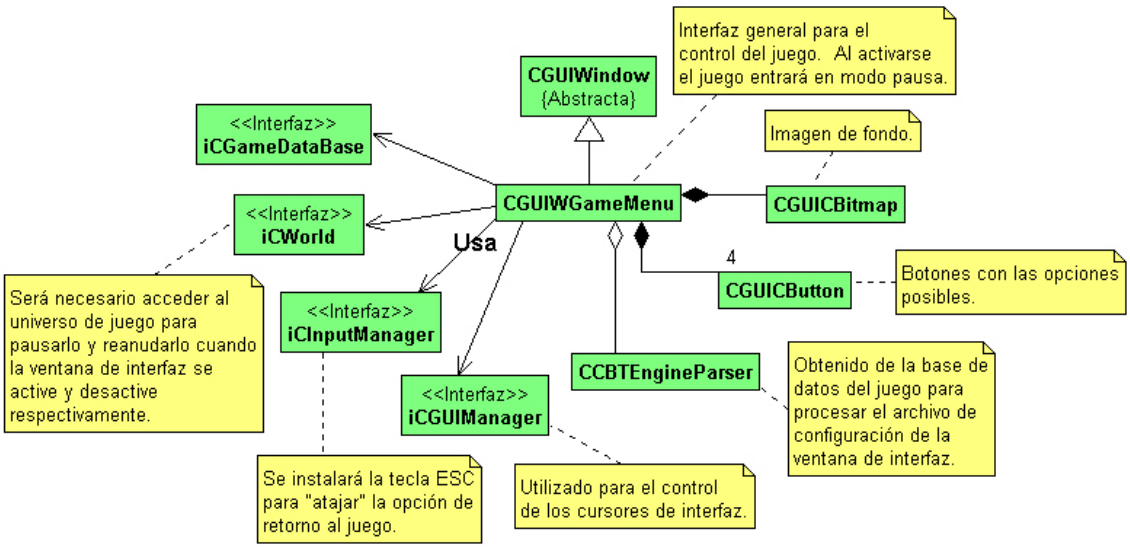


Diagrama de CGUIWGameMenu.

Se utilizarán cuatro componentes `CGUICButton` para representar las opciones posibles, más un componente `CGUICBitmap` para la imagen de fondo. A esta imagen, se le podrá asociar un valor de transparencia desde el archivo de configuración de la ventana de interfaz, de tal manera que se pueda ver parte de la porción del área de juego que ocupe.

Será necesario el acceso al universo de juego (interfaz `iCWorld`) para poder pausar el mismo en cuanto la ventana de interfaz quede activa y reanudarlo cuando se decida retornar al juego.

El uso de la interfaz al manager de entrada se hará para hacer que la ventana de interfaz escuche por el evento de pulsación a la tecla `ESC`. Al pulsar, se simulará la selección de la opción de retorno al juego (como veremos en el apartado destinado a la ventana de interfaz de juego principal, la pulsación desde ésta de la tecla `ESC` producirá que el menú de juego se active).

4.3.14.3.2.5. La clase CGUIWMainInterfaz.

La ventana de interfaz representada por esta clase, será la más importante para el control del sistema de juego. Desde ella, se gestionarán las órdenes del usuario sobre el jugador, se mantendrá la consola de juego, el acceso al inventario del jugador, la visualización de dos atributos y los ítems equipados en los slots principales. También será la encargada de mostrar el panel horario y los síntomas activos en el jugador. El diagrama será como sigue:

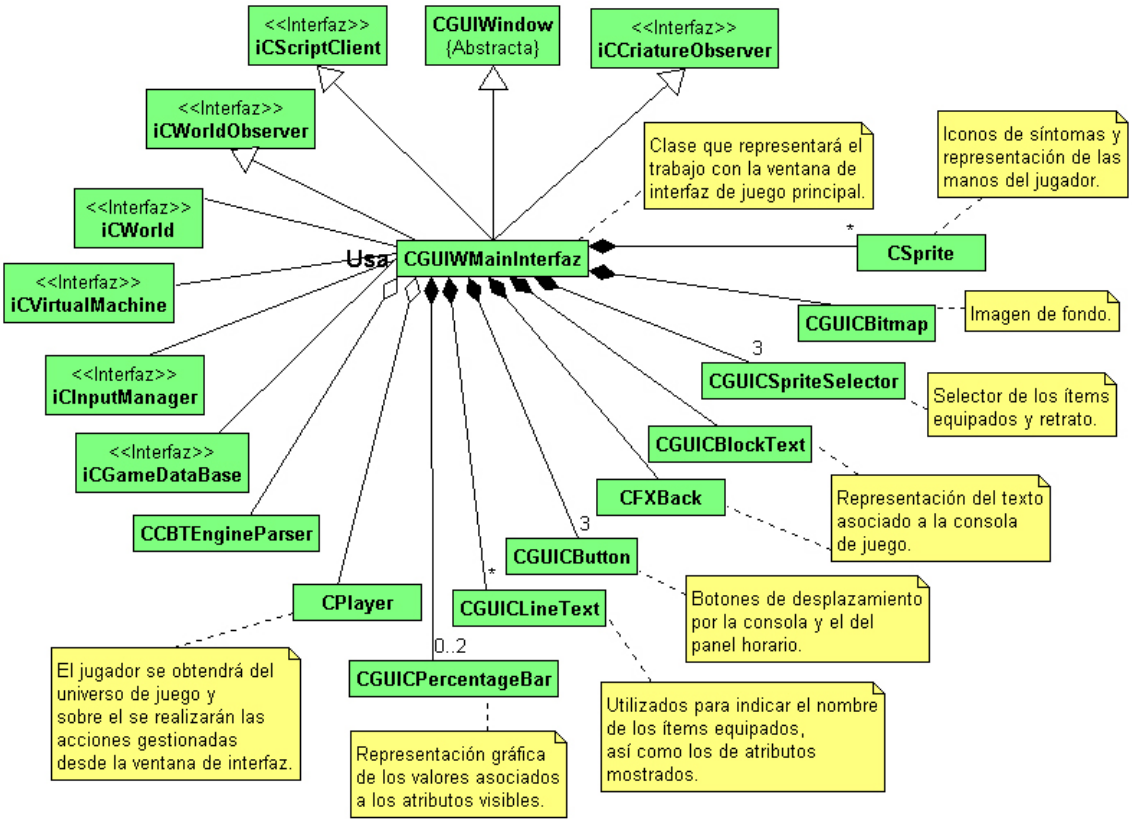


Diagrama de CGUIWMainInterfaz.

Como se puede observar en el diagrama, existirá un número considerable de componentes distintos. En primer lugar tendremos la imagen de fondo representada por `CGUICBitmap`. La imagen tendrá unas características especiales (comentadas en el manual de uso de *CrisolBuilder*) y no podrá tener animación asociada. Sobre dicha imagen irá un consola de juego (representada por `CGUICBlockText`), tres selectores (dos para los ítems equipados en los slots principales y uno para el retrato del jugador), los botones de manejo de la consola (desplazar una linea arriba y otra abajo) y control asociado al combate (pase de turno y opción de huida) y, por último, los componentes relativos a la información de los atributos del jugador (que serán opcionales y, de figurar, contendrán dos líneas de texto y una barra de porcentaje cada uno).

Esta ventana de interfaz tendrá una importancia muy grande a la hora de controlar el jugador y conocer los sucesos que pasen en el universo de juego. El control del jugador se logrará obteniendo la instancia al mismo desde `iCWorld` y registrándose (la ventana de interfaz) como observadora suya (se logrará al heredar de la interfaz `iCCriatureObserver`). Para saber qué pasa en el universo de juego se deberá de ser observador del universo de juego heredando de `iCWorldObserver` y registrándose a través de `iCWorld`.

Los iconos de síntomas serán representados por instancias `CSprite` y aparecerán en cuanto se activen en el jugador.

El panel horario será representado por un componente `CGUICButton`. Cada vez que pase una hora de juego, dicho componente se cargará con la representación gráfica que proceda para la hora cumplida.

Como el interfaz de juego principal podrá producir eventos (serán los asociados al objeto `Game` en *CrisolScript*), heredará de `iCScriptClient` y usará la máquina virtual, `iCVirtualMachine`, para solicitar su ejecución.

El trabajo con el universo de juego con el fin acceder al motor isométrico será intensivo, pues desde esta ventana de interfaz se gestionarán los estados de interacción y movimiento del jugador.

Añadir que, al igual que todas las ventanas de interfaz, se utilizará una instancia `CCBTEngineParser`, obtenida de la base de datos de juego (`iCGameDataBase`), para leer su configuración.

4.3.14.3.2.6. La clase CGUIWEntityInteract.

La ventana de interfaz representada por esta clase estará destinada a la gestión de uno de los menús más importantes; el de interacción con entidades. Será un menú sin localización precisa, pues aparecerá siempre al lado de la entidad con la que se desee interactuar. La configuración de la ventana de interfaz, en cuanto a sus opciones, será dinámica, es decir, según sea la entidad destino y eventos script que tenga asociados así se construirán las opciones del menú. El diagrama de la clase `CGUIWEntityInteract` será el siguiente:

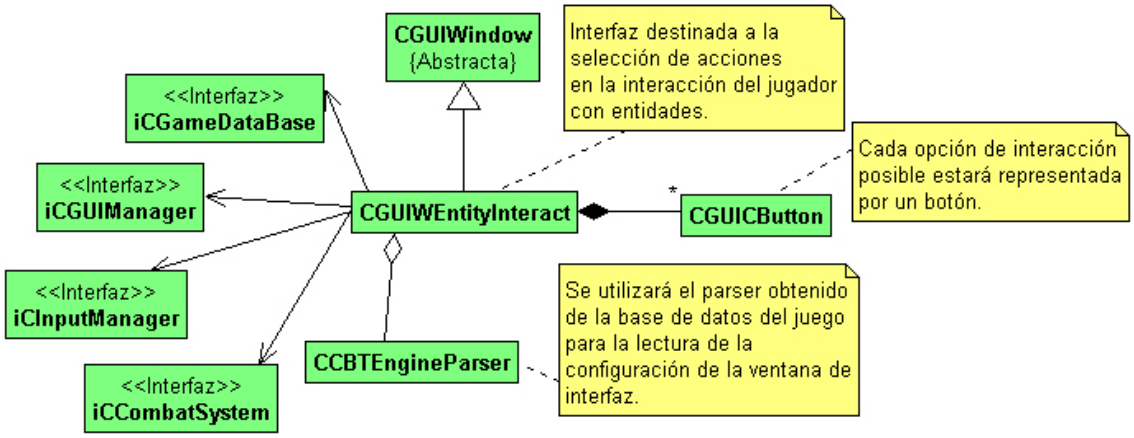


Diagrama de CGUIWEntityInteract.

Internamente `CGUIWEntityInteract` mantendrá un componente de tipo botón (`CGUICButton`) por opción de interacción posible, pero además deberá de tener otros auxiliares destinados a representar el deseo de abandonar el menú y las habilidad a escoger en caso de que se quiera utilizar una.

La ventana de interfaz deberá de tener un acceso al subsistema de combate para conocer si se halla activo algún combate y, en consecuencia, limitar las opciones ofertadas (si el modo combate se halla activo, acciones como el registro de entidades contenedoras de ítems estará prohibido).

Por lo demás, esta ventana de interfaz será lanzada desde el interfaz de juego principal cuando el usuario esté en estado de interacción y pulse el botón izquierdo del ratón sobre la entidad con la que quiera interactuar.

4.3.14.3.2.7. La clase CGUIWItemSelectToUse.

Esta ventana de interfaz se activará siempre que desde el menú de interacción de entidades, el usuario (el jugador) desee utilizar un ítem que se halle en su inventario sobre la entidad seleccionada para interacción. El diagrama será el siguiente:

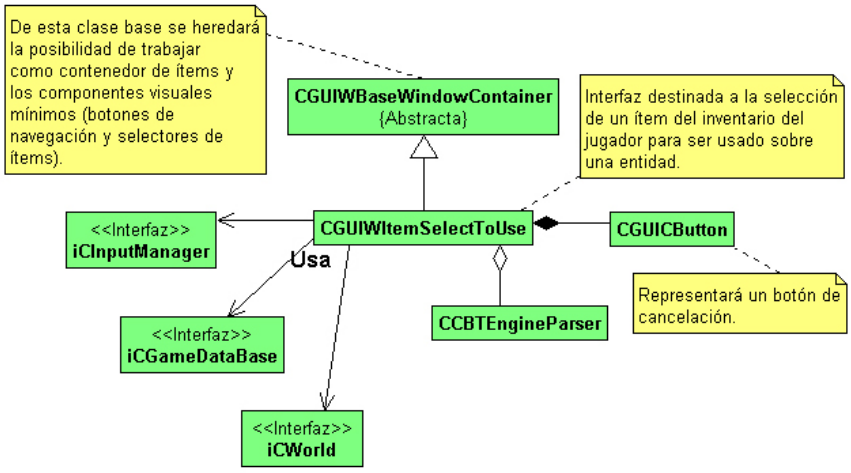


Diagrama de CGUWItemSelectToUse.

Como se puede observar en el diagrama, de la clase `CGUIWBaseWindowContainer` se heredará la apariencia visual base del contenedor, además del funcionamiento lógico para la gestión de los ítems a mostrar en el inventario del jugador. El uso de la interfaz `iCWorld` será necesario para obtener la instancia al ítem escogido para ser usado sobre la entidad destino.

Al igual que el resto de ventanas de interfaz, la configuración visual de la misma se llevará a cabo obteniendo la instancia al parser `CCBTEngineParser` de la base de datos del juego (`iCGameDataBase`) para la lectura del archivo de configuración. El manager de entrada (`iCInputManager`) se utilizará para instalar a la ventana como cliente del evento asociado a la pulsación de la tecla `ESC` para el cierre automático de la misma y cancelación de la acción (representada en la ventana de interfaz por un componente botón, `CGUICButton`).

4.3.14.3.2.8. La clase CGUIWPlayerProfile.

Esta clase se encargará de representar la ventana de interfaz con el inventario del jugador. Probablemente sera la más compleja desde el punto de vista del mantenimiento de componentes, ya que en ella se mostrará el contenido del inventario del jugador, sus slots de equipamiento, los iconos de habilidades, de síntomas, una consola informativa, el retrato del jugador y la información relativa a los distintos atributos. El diagrama UML será el siguiente:

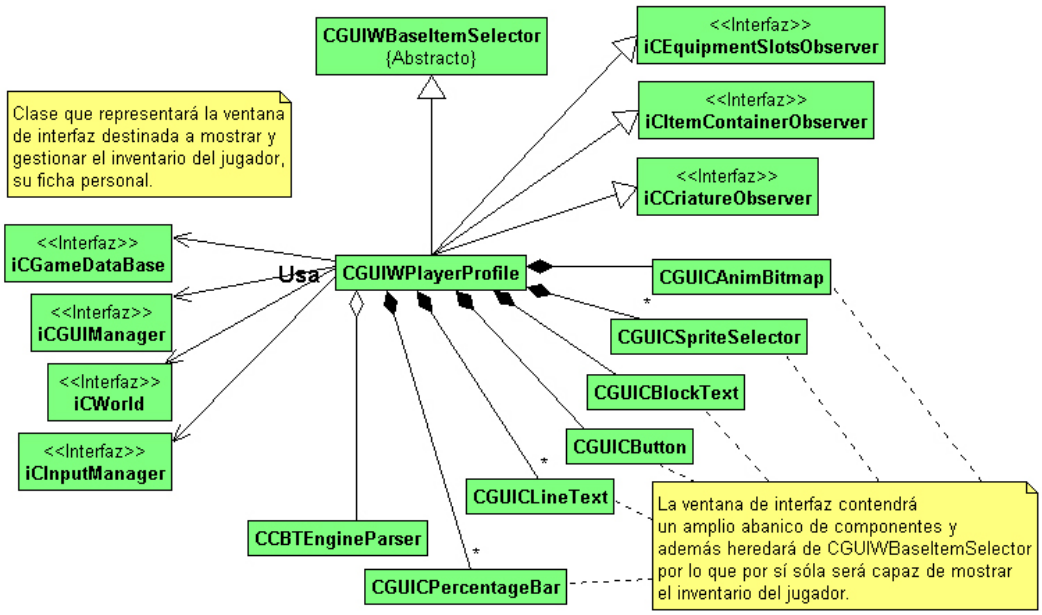


Diagrama CGUIWPlayerProfile.

Como se puede observar, la clase mantendrá un importante conjunto de componentes, sin embargo, se ha de notar que heredará de `CGUIWBaseItemSelector`, lo que permitirá a la clase la gestión lógica de un contenedor de ítems para su posterior muestra en pantalla. Se decidirá esta herencia en lugar de `CGUIWBaseWindowContainer` porque esta ventana de interfaz será capaz de mostrar y ofrecer mucha mayor cantidad de aspectos.

La clase también se instalará como observador del propio jugador para reflejar cualquier cambio que sobre el mismo se produzca, así como para obtener una rápida notificación sobre las acciones que se realicen sobre los slots de equipamiento e inventario del mismo.

Los componentes asociados estarán destinados a representar texto informativo asociado a los atributos, nombres de los ítems equipados, etc. Los slots de equipamiento, así como el retrato del jugador, estarán representados por componentes `CGUIWSpriteSelector`, mientras que la consola asociada se representará utilizando un componente `CGUICBlockText`. Los botones (componentes `CGUICButton`) serán utilizados ampliamente para, por ejemplo, navegar entre los ítems que mantenga el jugador en inventario, en la consola de juego o, incluso, para los síntomas activos (los cuales mostrarán su nombre, cuando se hallen activos, al pasar el cursor del ratón por encima). Añadir, por último, que el componente que represente la imagen de fondo será de tipo `CGUICAnimBitmap`, lo que permitirá añadir animación estática.

La manipulación de un ítem de inventario podrá ser efectiva para observarlo, manipularlo, tirarlo o aplicarle el uso de una habilidad. Para ello, esta interfaz lanzará el menú de interacción (clase `CGUIWEntityInteract`).

El uso de subsistemas será también bastante importante. Mediante `iCWorld` se obtendrá la instacia `CPlayer` a la que esta ventana de interfaz estará dedicada. De `CPlayer` se obtendrá el contenedor de ítems que formará su inventario así como el de slots de equipamiento.

El uso de `iCInputManager` será destinado a gestionar los eventos que produzca el ratón sobre la propia ventana, además de para controlar las acciones que quiera hacer el jugador en su interior (coger un ítem y equiparlo en un slot o utilizarlo sobre el jugador, etc).

4.3.14.3.2.9. La clase CGUIWConversator.

Ventana de interfaz destinada a representar el diálogo entre el jugador y una entidad. Heredará de `CGUIWBaseTextSelector`, por lo que la lógica asociada al mantenimiento de líneas de texto estará resuelta en dicha herencia. El diagrama será el siguiente:

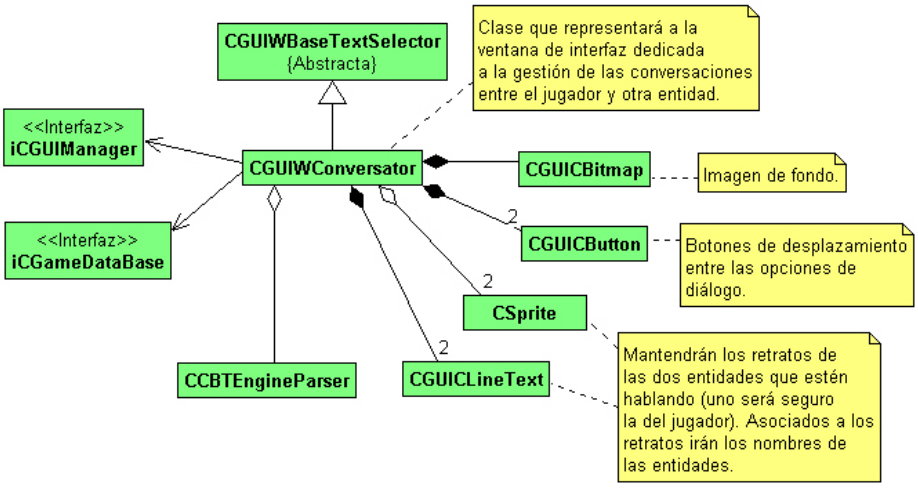


Diagrama de CGUIWConversator.

La ventana de interfaz estará compuesta por todas las líneas de diálogo posibles (mantenido todo en la clase base), dos botones para el desplazamiento por entre las mismas

(CGUICButton), los retratos de las entidades que estén dialogando (CSprite) y, por último, los nombres de las mismas (CGUICLineText).

Siempre que se produzca la elección de una línea de texto, el cliente asociado a la ventana de interfaz recibirá la notificación, la cual incluirá el identificador de la línea de texto seleccionada. Como los diálogos se construirán siempre a partir de código script, la instrucción script que represente la activación de esta ventana de interfaz será el cliente que recibirá las notificaciones.

Por lo demás, el uso del manager de interfaces (iCGUIManager) será necesario para el trabajo con los cursores mientras que la configuración de la ventana se leerá a través de una insancia CCBTEngineParser, obtenida de la base de datos del juego (iCGameDataBase).

4.3.14.3.2.10. La clase CGUIWCombat.

Esta será una de las interfaces más sencillas de todas. Únicamente tendrá como objetivo mostrar los puntos de acción de la criatura que posea el turno de combate, reflejando la variación de los mismos. El diagrama será el siguiente:

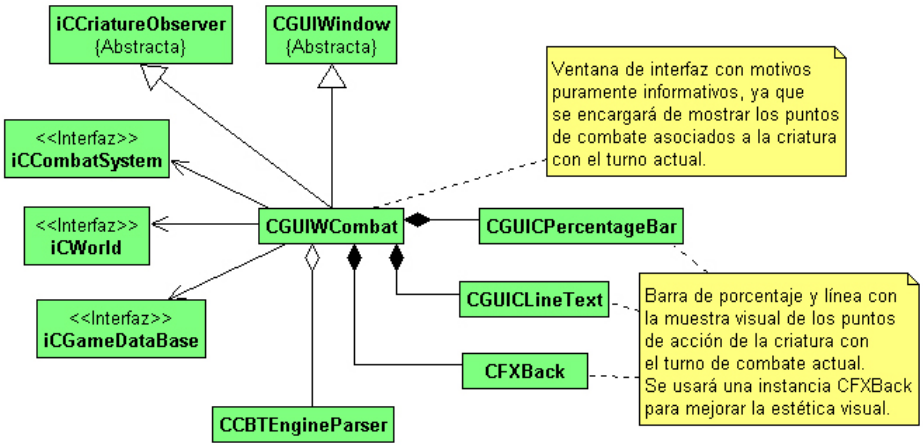


Diagrama de CGUIWCombat.

La información se mostrará mediante el uso de una barra de porcentaje (CGUICPercentageBar) y una línea de texto (CGUICLineText) indicando los puntos actuales sobre los puntos iniciales (*ValoresActuales / ValoresTotales*). El texto informativo se pondrá sobre la barra de porcentaje automáticamente. Para mejorar la estética del proceso de muestra de información, se usará una instancia CFXBack que irá por debajo de la barra de porcentaje y el texto informativo.

El trabajo interno se logrará instalando a la ventana de interfaz como observadora de la criatura recibida vía handle (de ahí la herencia de iCCriatureObserver). De esta forma, cualquier acción realizada recibirá una notificación para que la ventana de interfaz consulte al subsistema de combate y obtenga los puntos actuales de la criatura. Naturalmente, será el subsistema de combate (iCCombatSystem) quien se encargue de activar esta interfaz por cada nueva criatura que posea el turno. El uso de la interfaz al universo de juego (iCWorld) será necesario para obtener las instancias CCriature asociadas a los handles pasados por el subsistema de combate.

Al igual que el resto de ventanas de interfaz, la configuración será leida de un archivo ".cbt" mantenido por la instancia CCBTEngineParser obtenida, a su vez, de la interfaz asociada a la base de datos del juego (iCGameDataBase).

4.3.14.3.2.11. El intercambio de ítems entre dos contenedores

El trabajo de intercambio de ítems con dos contenedores, que sucederá siempre que el jugador registre otra criatura o bien un objeto de escenario contenedor, no será llevado a cabo por una

clase, sino que se crearan dos ventanas de interfaz contenidas en una tercera. La ventana de interfaz encargada de gestionar todo el trabajo que actuará de facade, será `CGUIWTrade`, mientras que `CGUIWPlayerToEntity` y `CGUIWEntityToPlayer` se encargarán de gestionar la ventana con los ítems del jugador y de la entidad con la que éste intercambiará.

A continuación se analizará cada una de estas clases.

4.3.14.3.2.11.1. La clase CGUIWTrade.

Esta clase se encargará de gestionar todo el trabajo entre las dos ventanas. Internamente contendrá una instancia a cada una de ellas y recibirá la notificación de las mismas cuando un suceso ocurra sobre ellas. Así que, además de ser una ventana de interfaz, será también un cliente asociado a las dos ventanas que contendrá. El diagrama será el siguiente:

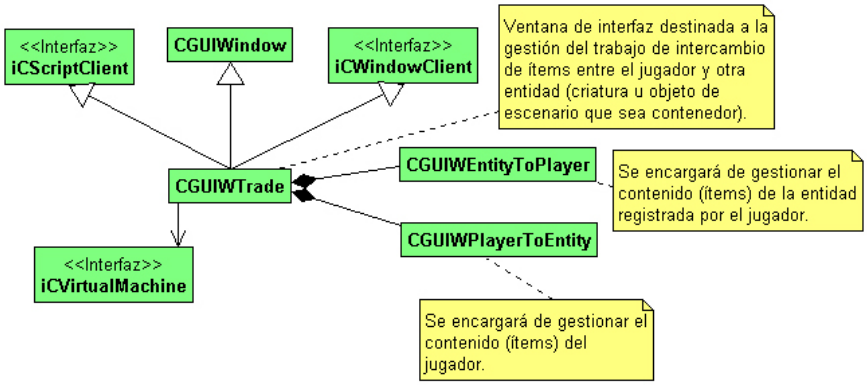


Diagrama de CGUIWTrade.

Como se puede observar, la clase heredará de `iCWindowClient` ya que actuará de cliente de las ventanas de interfaz `CGUIWEntityToInteract` y `CGUIWPlayerToEntity`, con el fin de conocer los distintos sucesos que pasen sobre éstas y actuar en consecuencia. La herencia de `iCScriptClient` se deberá a que podrán producirse eventos relacionados con el cambio de contenedor de un ítem y la ventana de interfaz deberá de solicitar la ejecución el posible archivo script asociado vía `iCVirtualMachine`.

Desde cada una de las ventanas de interfaz destinadas a manejar a los dos contenedores se podrán realizar una serie de acciones. En la del jugador, se podrá realizar la observación de los ítems que este transporte además de poder tirarlos al suelo, mientras que en la de la entidad registrada los ítems sólo podrán observarse. Cabe añadir que la gestión de estas acciones se realizará utilizando el menú representado por `CGUIWEntityInteract`.

Como ya se comentó, esta clase hará las veces de facade y manager de las ventanas de interfaz que se encargarán del control de cada uno de los contenedores, de ahí que no tenga realmente representación visual.

4.3.15.3.2.11.2. La clase CGUIWPlayerToEntity.

Esta será la ventana de interfaz que se encargará de mantener el inventario del jugador. El diagrama será el siguiente:

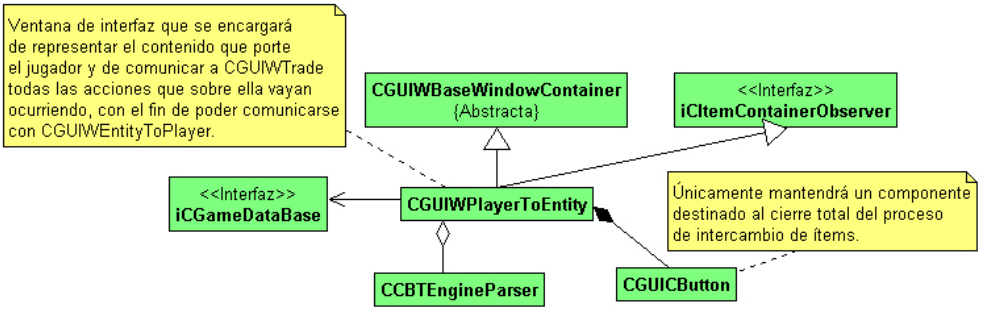


Diagrama de CGUIWPlayerToEntity.

La ventana de interfaz heredará de `CGUIWBaseWindowContainer` para obtener las características base de cualquier ventana encargada de representar el contenido de un contenedor de ítems. Además, actuará de observador del inventario del jugador, con el fin de que cualquier inserción o desalojo en el mismo sea reflejado inmediatamante en pantalla.

El único componente que tendrá asociado será el de un botón (`CGUICButton`) destinado a representar la opción de salida del proceso de intercambio de ítems.

Naturalmente, la ventana de interfaz tomará la instancia `CCBTEngineParser` con la que proceder a leer su configuración.

4.3.15.3.2.11.3. La clase CGUIWEntityToPlayer.

Esta ventana de interfaz será destinada a representar el inventario de la entidad que el jugador esté manipulando. Su diagrama será el siguiente:

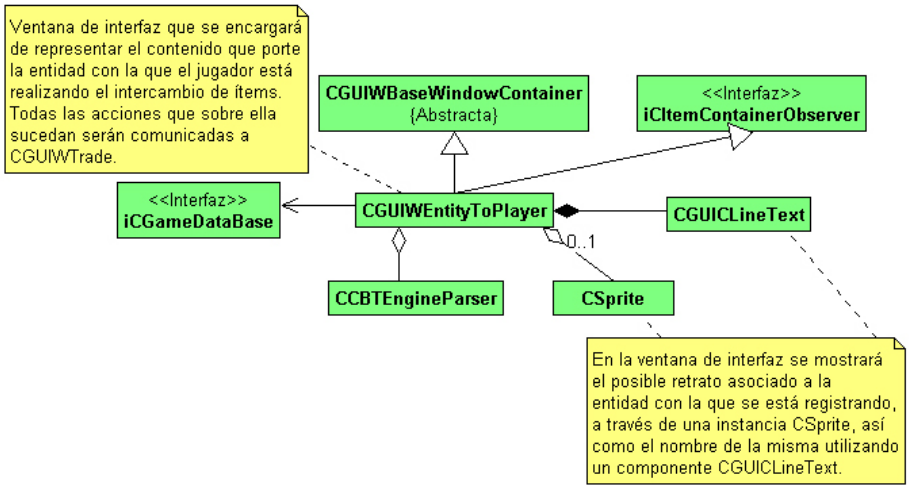


Diagrama de CGUIWEntityToPlayer.

Al igual que la ventana de interfaz destinada a representar el inventario del jugador, ésta heredará de `CGUIWBaseWindowContainer` para tener implementada la forma básica de la ventana de gestión de un contenedor. Además, también heredará de `iCItemContainerObserver` porque se instalará como observador del contenedor de la entidad sobre la que el jugador registre.

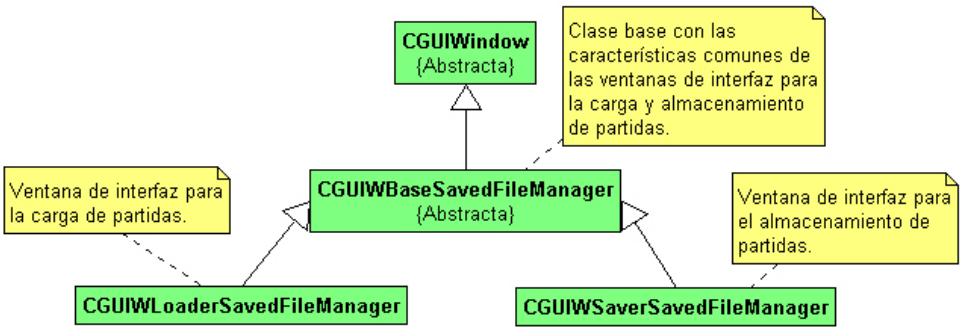
Los componentes que poseerá estarán destinados a mostrar el retrato y el nombre de la criatura a la que pertenece el contendor mostrado. Para ello, se obtendrán la instancia `CSprite` con el posible retrato (recordar que el retrato asociado a una entidad será de carácter optativo) y otra `CGUICLineText` para el nombre.

Todas las acciones que estén destinadas a una comunicación con la otra ventana de interfaz, serán notificadas a `CGUIWTrade`, con el fin de que ésta resuelva el problema lógico.

4.3.14.3.2.12. Interfaces para guardar y cargar partidas.

El proceso de guardar y cargar partidas se realizará a través de dos clases diferentes pero que mantendrán muchas características comunes. Únicamente, desde el punto de vista visual, el proceso de almacenamiento incorporará una novedad sobre el proceso de carga, el relativo a la introducción de la descripción asociada al slot de grabación. El resto de componentes de la ventana de interfaz, así como la gestión de las partidas guardadas será el mismo.

La clase que representará la ventana de interfaz asociada a la grabación será `CGUIWSaverSavedFileManager` y la destinada a la carga de partidas previamente guardadas `CGUIWLoaderSavedFileManager`. Las dos ventanas de interfaz heredarán de `CGUIWBaseSaveFileManager`, que poseerá todas las características comunes de ambos interfaces. He aquí el diagrama:



Jerarquía de clases para la carga y almacenamiento de partidas.

4.3.14.3.2.12.1. La clase CGUIWBaseSaveFileManager.

Esta será la clase base para las destinadas a realizar las operaciones concretas de guardar y salvar partiddas. En su interior mantendrá una lista con todos los nombres de los archivos de partidas guardadas, gestionando la interacción sobre los mismos para mostrar su descripción. Al ser ambas interfaces muy parecidas, mantendrá todos los componentes visuales comunes. He aquí el diagrama:

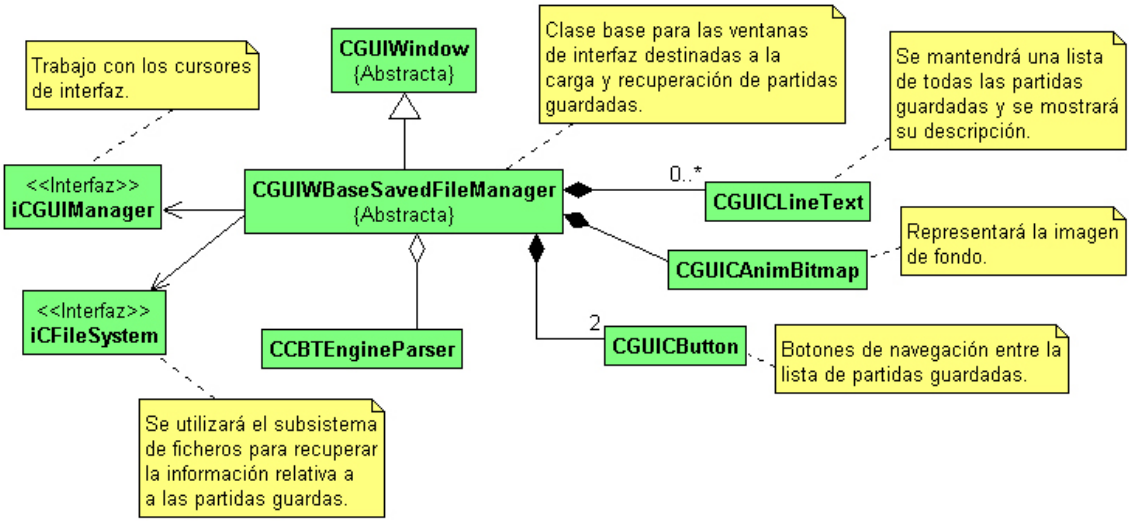


Diagrama de CGUIWBaseSavedFileManager.

Internamente se almacenará una lista con toda la información asociada a los archivos de partidas guardadas existentes. La información que se almacenará será la descripción de las mismas en componentes `CGUICLineText`. Estas componentes serán mostradas cuando el archivo al que pertenezcan sea seleccionado para realizar algún tipo de operación con él. Lógicamente, para obtener la descripción de los archivos se deberá de utilizar el subsistema de ficheros.

Además de estas componentes, la clase base mantendrá dos botones (`CGUICButton`) para la navegación por entre los archivos de partidas guardadas mas luego, uno que representará la imagen de fondo con posibilidad de tener asociada animación estática (`CGUICAnimBitmap`).

Añadir que para la lectura de la configuración asociada a los componentes que mantenga la clase base, el parser (`CCBTEngineParser`) será pasado al propagarse la inicialización desde la clase derivada que proceda (bien la de carga o bien la de almacenamiento de información).

4.3.14.3.2.12.2. La clase CGUIWLoaderSavedFileManager.

Esta clase representará el trabajo con el proceso de carga de partidas previamente guardadas. Será más sencilla que la destinada a la grabación, pues desde aquí se permitirá seleccionar una partida para cargar o bien la salida. Esta ventana de interfaz podrá ser activada desde el menú principal o desde el menú de juego. He aquí el diagrama:

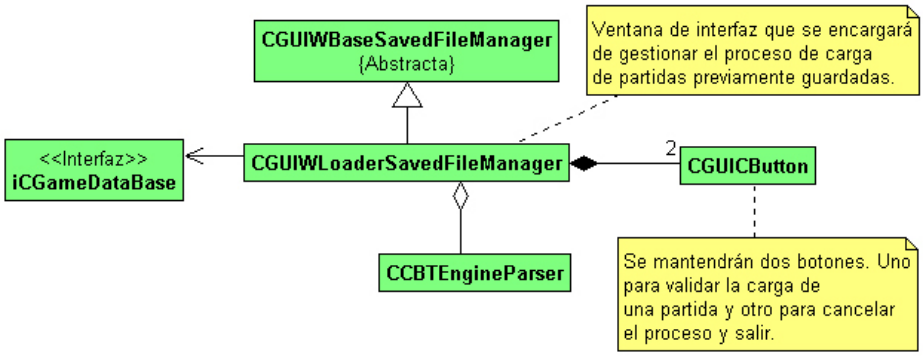


Diagrama de CGUIWLoaderSavedFileManager.

Internamente la ventana de interfaz mantendrá dos componentes de tipo botón (`CGUICButton`) destinados a la validación de la carga de una partida previamente seleccionada y a la cancelación del proceso. La pulsación sobre el botón de carga sólo será posible si existe alguna partida seleccionada.

El uso de la base de datos del juego (`iCGameDataBase`) será para la obtención del parser (`CCBTEngineParser`) con el que cargar la configuración de la ventana de interfaz. Dicho parser será propagado a la clase base para que lea los valores de inicialización asociados a la misma.

La carga de una partida supondrá cambiar de estado al motor (que internamente activará la ventana de interfaz `CGUIWGameLoading`), de ahí que no se use, en esta ventana de interfaz, el universo de juego (`iCWorld`) para llevar a cabo las operaciones lógicas asociadas a la carga de una partida.

4.3.14.3.2.12.3. La clase CGUIWSaverSavedFileManager.

La ventana de interfaz asociada al almacenamiento de una partida será más complejo ya que, además de controlar un componente adicional para la introducción del nombre del archivo, también se dará la oportunidad al usuario de eliminar partidas previamente guardadas, con el fin de ofrecer una ligera gestión de archivos (para no saturar el disco duro del usuario). He aquí el diagrama:

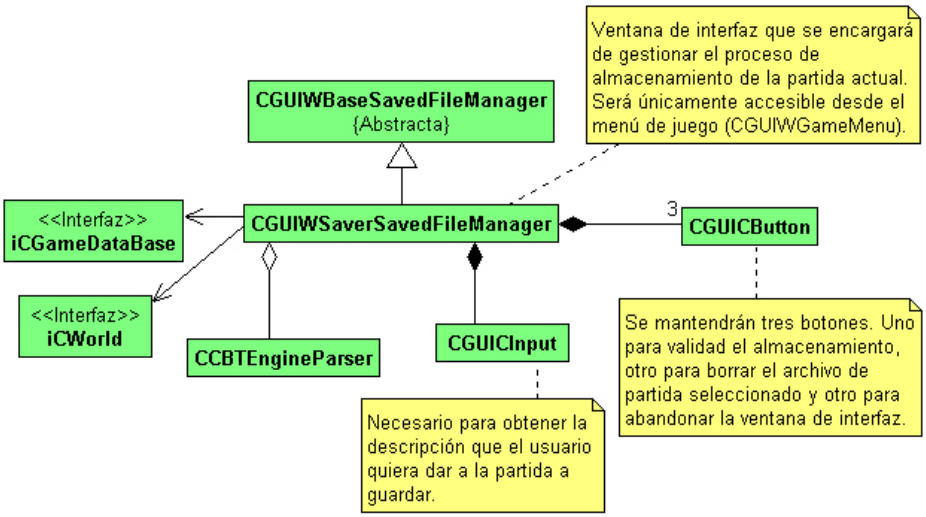


Diagrama de CGUIWSaverSavedFileManager.

Se mantendrán tres componentes de tipo botón (`CGUICButton`) para representar las opciones de almacenamiento, borrado y abandono de la interfaz. El almacenamiento sólo se podrá realizar obteniendo antes una descripción por parte del usuario, de ahí el uso de `CGUICInput`.

El trabajo con el universo de juego será necesario, pues será desde la interfaz `iCWorld` donde se realice la operación concreta de almacenamiento de una partida. Una vez finalizado el proceso, la ventana de interfaz se pondrá en contacto con la clase base para que actualice el listado de archivos de partidas guardadas, ya que será seguro que haya una nueva.

Al contrario que el almacenamiento, la operación lógica asociada al borrado será llevada desde esta misma ventana de interfaz sin embargo, sí que se notificará a la clase base una vez realizada la acción, para que actualice la lista de partidas guardadas pues será seguro que haya una nueva asociada.

Por último indicar que esta ventana de interfaz sólo podrá ser lanzada desde el menú de juego, al contrario que la destinada a la recuperación de partidas, que podía ser lanzada tanto desde el menú principal como desde el menú de juego.

4.3.14.3.2.13. La clase CGUIWPresentation.

La ventana de interfaz encargada de mantener las presentaciones será una de las más complejas en cuanto a las posibilidades que ofrecerá. Las presentaciones no serán más que sucesiones de pantallas, a modo de “diapositivas”, cada una de las cuales podrá tener una considerable cantidad de parámetros de configuración.

Todas las presentaciones se configurarán a partir de plantillas en un archivo. Las presentaciones podrán ser utilizadas para narrar algo o para mostrar un gráfico explicativo. Las presentaciones podrán ser utilizadas durante el juego sin embargo, su uso será obligatorio en la sección de créditos, por lo que al menos deberá de existir una presentación. Toda la información sobre las mismas se podrá hallar en la documentación de usuario sobre *CrisolBuilder*.

El diagrama de la clase `CGUIWPresentation`, encargada de mantener las presentaciones, será el siguiente:

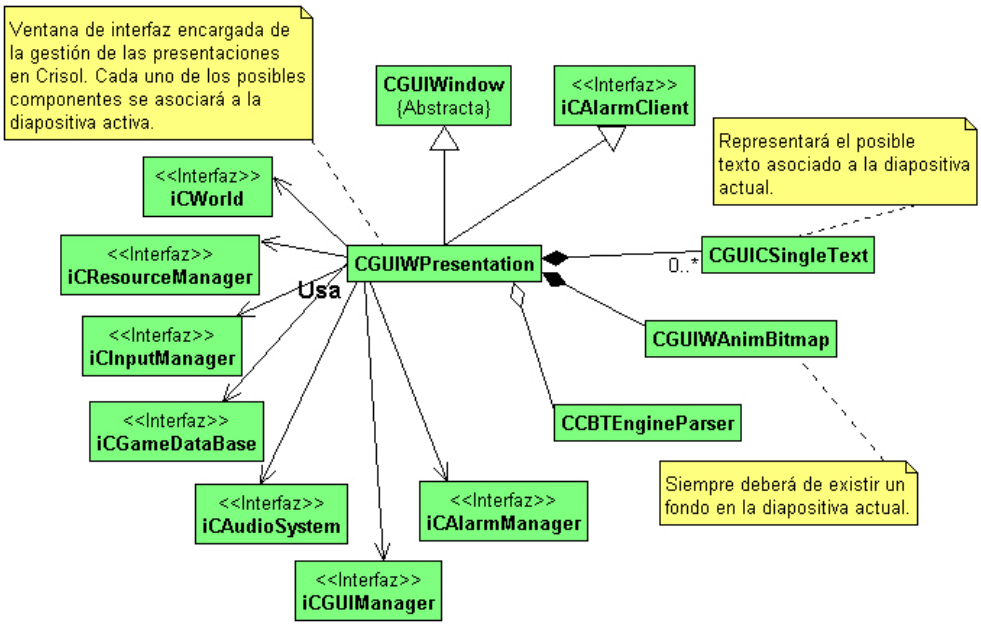


Diagrama de CGUIWPresentation.

En todo momento se mantendrá la información asociada a la presentación actual, ya que cargar todas las diapositivas o pantallas sería un coste excesivo.

Los dos únicos tipos de componentes que mantendrá la ventana de interfaz será relativos a la muestra de posible texto y al mantenimiento de una obligada imagen de fondo. Por otro lado, la información musical será controlada leyendo los nombres de los ficheros y registrando, en el caso de los ficheros WAV, los mismos a través del servidor de recursos (`iCResourceManager`). La reproducción se realizará a través del subsistema de audio (`iCAudioSystem`).

Siempre que una presentación se active el universo de juego (en caso de que se esté jugando) deberá de pausarse. Tal acción será comunicada vía `iCWorld`.

La ventana de interfaz heredará de `iCAlarmClient` para el control del tiempo en el que se podrá pasar las diapositivas de forma automática.

El uso del manager de entrada (`iCInputManager`) será necesario para escuchar los eventos asociados a la pulsación del botón izquierdo del ratón. Por definición (ver manual de usuario de *CrisolBuilder*) la pulsación del botón izquierdo del ratón supondrá el paso automático de una diapositiva a otra.

Por último, la lectura de la configuración de las distintas diapositivas que compongan la presentación se realizará utilizando una instancia `CCBTEngineParser` obtenida a través de la base de datos del juego (`iCGameDataBase`).

4.3.14.3.3. Interfaces secundarios.

Los interfaces secundarios serán aquellos que podrán activarse desde *CrisolScript* con el fin de que el diseñador pueda obtener información diversa del jugador. No serán, por tanto, lanzados por *Crisol* para representar un estado en el grafo de ejecución del mismo y siempre podrán ser activados dando igual el lugar desde donde se lancen.

Las ventanas de interfaz consideradas como interfaces secundarios serán las de lectura de archivos de texto (`CGUIWTextReader`), selección de opciones de texto (`CGUIWTextOptionsSelector`) y cuadros de diálogo de pregunta (`CGUIWQuestionDialog`) e información (`CGUIWAdviceDialog`). A continuación se analizará cada uno de ellos.

4.3.14.3.3.1. La clase CGUIWTextReader.

La finalidad de esta interfaz será la de mostrar un archivo de texto. El diagrama será el siguiente:

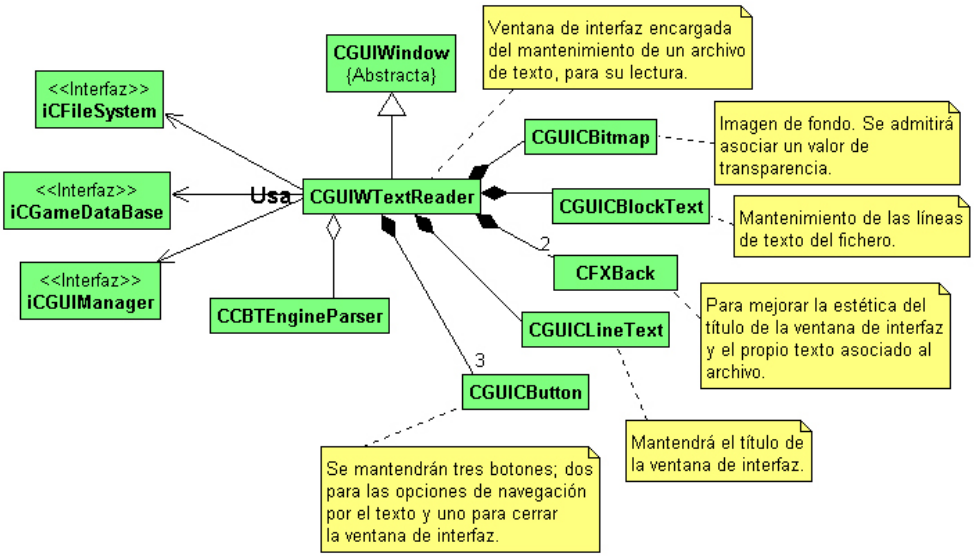


Diagrama de CGUIWTextReader.

La ventana estará compuesta por una imagen de fondo (`CGUICBitmap`), a la que se podrá asociar un valor de transparencia para dejar entrever la información gráfica sobre la que es mostrada, un bloque de texto complejo (`CGUICBlockText`), botones (`CGUICButton`), para abandonar la ventana y navegar entre el texto asociado al archivo, y una línea de texto (`CGUICLineText`), que contendrá el título asociado al fichero de texto que se esté mostrando. El texto del título y del propio archivo, será mostrado sobre un componente `CFXBack` para mejorar su presencia.

Será necesario que la ventana acuda al subsistema de ficheros para la lectura del archivo de texto. Éste se leerá de una sola vez para ser volcado, a continuación, sobre el componente `CGUICBlockText`. El uso de la base de datos del juego se limitará a la obtención de la instancia `CCBTEngineParser` para la lectura de la configuración de la ventana de interfaz. Por último, añadir el uso del manager de interfaz (`iCGUIManager`) con el fin de establecer los cursores.

4.3.14.3.3.2. La clase CGUIWTextOptionsSelector.

Esta ventana de interfaz secundaria se encargará de actuar como un selector de texto, esto es, ofrecerá diferentes opciones en forma de texto de las cuales el jugador, el usuario, podrá seleccionar una. Toda la gestión del texto será llevado desde la clase base `CGUIWBaseTextSelector`, clase que `CGUIWTextOptionSelector` heredará, por lo que su trabajo será, básicamente, el de mantener los componentes de interfaz. He aquí su diagrama:

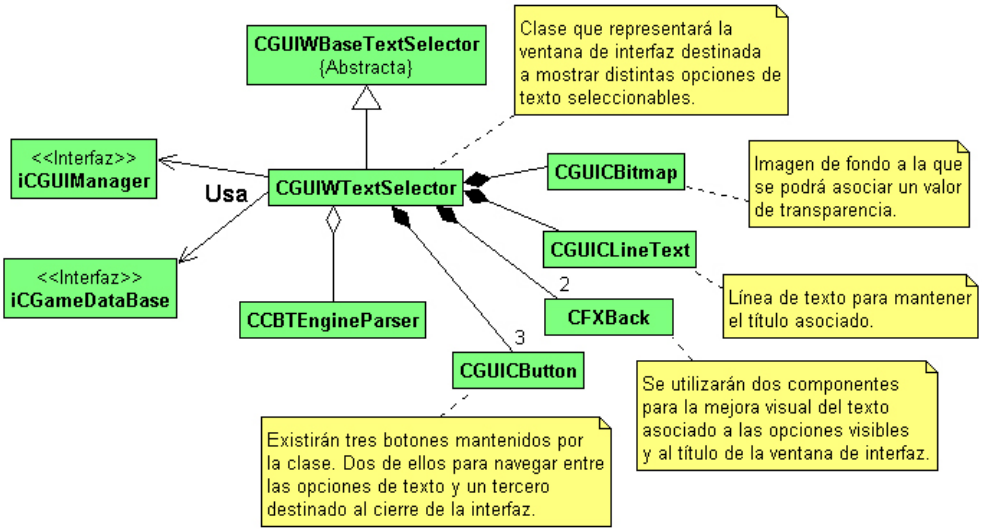


Diagrama de CGUIWTextOptionsSelector.

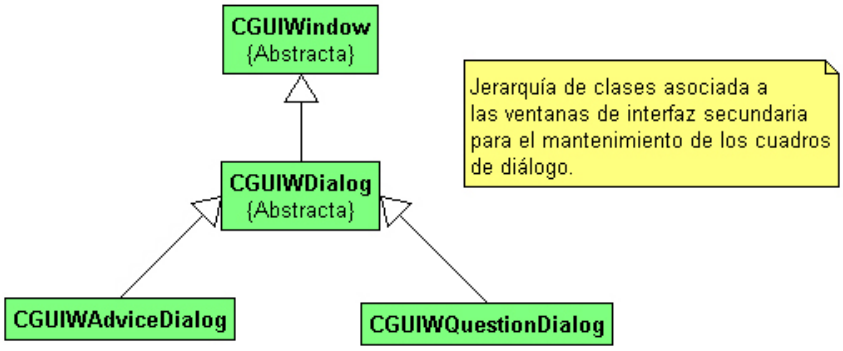
La ventana de interfaz estará compuesta por una imagen de fondo (`CGUICBitmap`), a la que se le podrá añadir cierto valor de transparencia, una línea de texto (`CGUICLineText`), para mostrar el título de la ventana de interfaz, y tres botones (`CGUICButton`), dos para navegar por entre las opciones de texto y un tercero para cerrar la interfaz. A la hora de inicializar se podrá especificar si el cierre de la ventana de interfaz estará permitido o bien si será necesario haber seleccionado una opción.

El acceso a interfaces será el básico para las ventanas de interfaz. Por un lado estará la necesidad de obtener una instancia `CCBTEngineParser` al archivo con la configuración. Esta instancia se obtendrá de la base de datos del juego (`iCGameDataBase`). Por otro lado, será necesario utilizar el manager de interfaz (`iCGUIManager`) para establecer los cursores de interfaz.

4.3.14.3.3.3. Cuadros de diálogos.

Los cuadros de diálogo serán utilizados para mostrar un texto informativo o bien para preguntar algo al usuario. En ambos casos, la estructura de los mismos será idéntica: Tan sólo variará el número de opciones asociadas.

La jerarquía existente asociada a los cuadros de diálogo será la siguiente:



Jerarquía de cuadros de diálogo.

En la clase base `CGUIWDialog` se mantendrán los componentes comunes a los dos cuadros de diálogo existentes, el de aviso, encarnando por la clase `CGUIWAdviceDialog`, y el de pregunta, representado por `CGUIWQuestionDialog`. A continuación se comentará cada una de las clases con detalle.

4.3.14.3.3.3.1. La clase CGUIWDialog.

Esta clase base se utilizará, básicamente, para contener aquellos componentes que sean comunes a cualquiera de los dos cuadros de diálogo. El diagrama será el siguiente:

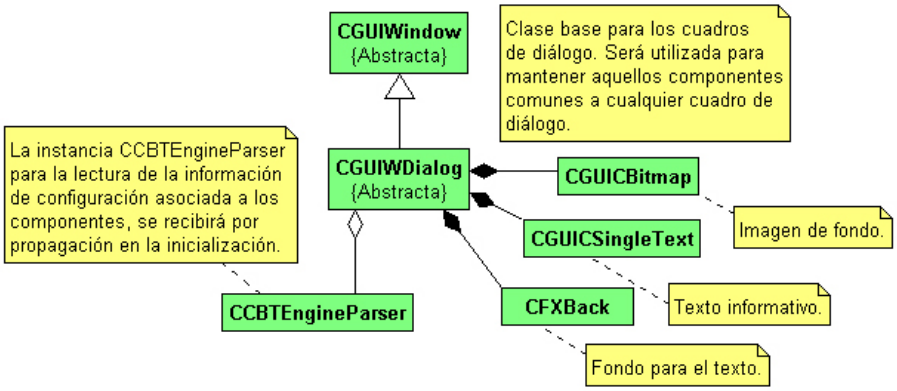


Diagrama de CGUIWDialog.

Como se puede observar, todo cuadro de diálogo poseerá una imagen de fondo sin posibilidad de tener asociada animación estática (`CGUICBitmap`), un texto con la información, pregunta, notificación, etc que se desee mostrar (`CGUICSingleText`), el cual irá a compañado de un componente `CFXBack` para suavizar su dibujado en pantalla.

La instancia `CCBTEngineParser` será recibida en la propagación de la inicialización, comenzada en las clases derivadas del cuadro de diálogo.

4.3.14.3.3.3.2. La clase CGUIWAdviceDialog.

Este cuadro de diálogo se utilizará para notificar algo. Será el más sencillo de los dos cuadros de diálogo. Su diagrama:

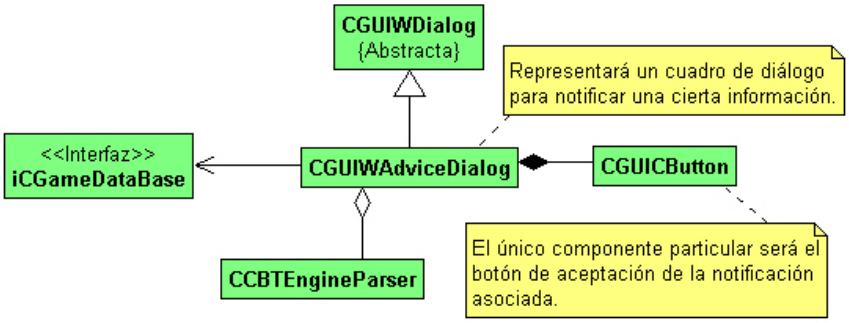


Diagrama de CGUIWAdviceDialog.

Como se puede observar, el diagrama UML es muy elemental. El cuadro de diálogo únicamente poseerá un componente particular, que no será otro que el botón (`CGUICButton`) de aceptación del mensaje que contiene. Cuando el usuario pulse dicho botón, el cuadro se cerrará.

4.3.14.3.3.3.3. La clase CGUIWQuestionDialog.

Esta clase representará el cuadro de diálogo destinado a la realización de preguntas. Para ello, incorporará tres componentes de tipo botón; uno para aceptar lo que se pregunta, otro para rechazarlo y el último para cancelar la pregunta. Este último botón será opcional. El diagrama será como sigue:

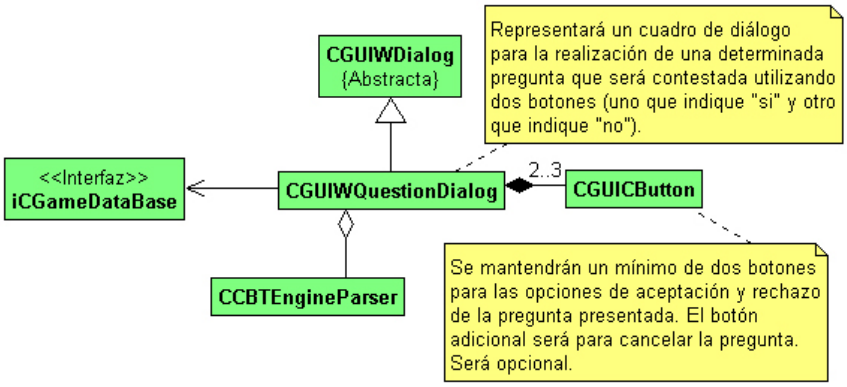


Diagrama de CGUIWQuestionDialog.

Se mantendrán un minimo de dos componentes botón (`CGUICButton`) para las opciones de aceptación y rechazo de la pregunta que contenga el cuadro. El botón adicional será de cancelación y será decisión del diseñador el mostrarlo.

4.3.15. El subsistema de combate.

El trabajo del subsistema de combate será el de comprobar, en cada actualización, si existe al menos una criatura en cada uno de las dos posibles alineaciones de combate. Si esto es así, automáticamente se encargará de iniciar el combate y comenzar a gestionar los turnos del mismo.

El jugador siempre permanecerá en una de las alineaciones (de esto se encargará `CWorld` al inicializar una nueva sesión de juego o bien al entrar en una nueva área) de tal forma que un combate comenzará de inmediato con solo colocar una criatura en el bando opuesto (siempre y cuando en su rango se halle al menos una criatura previamente alineada).

El diagrama del subsistema será el siguiente:

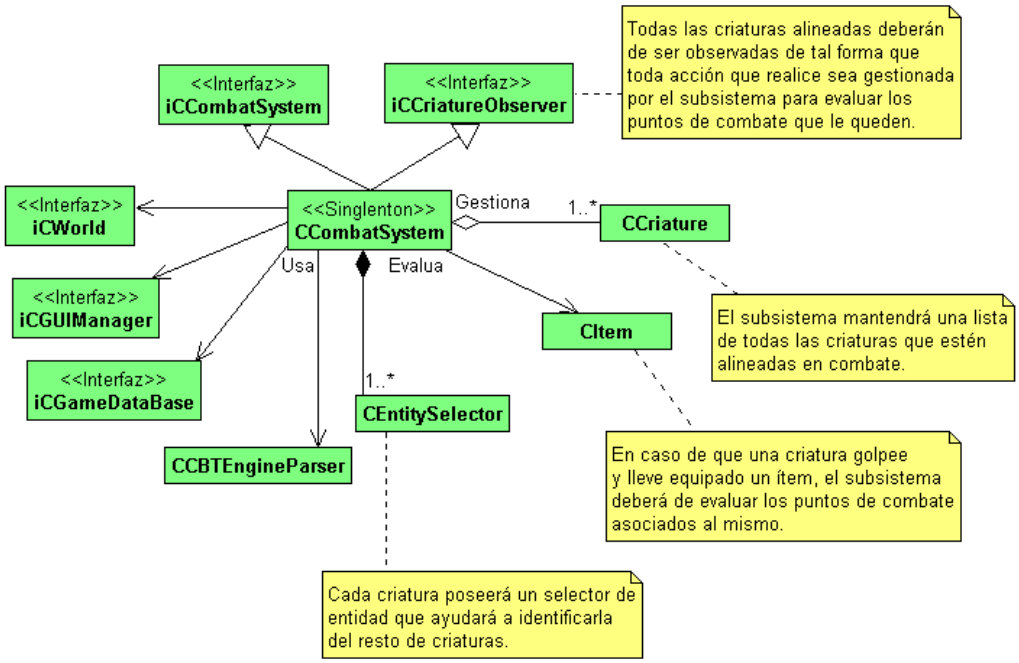


Diagrama del subsistema de combate.

El subsistema de combate se verá representado informativamente desde el gestor de interfaces pues éste permitirá trabajar con la barra de información relativa a los puntos de acción que posee la criatura actual.

4.3.15.1. La clase CCombatSystem.

En `CCombatSystem` se llevará una gestión completa de las criaturas que estén en la lista de combate y muy particularmente de la que posee el turno actual, de tal forma que los puntos de combate de la misma serán mantenidos en `CCombatSystem`. Para poder realizar un rastreo de las acciones que realice la criatura con el turno, `CCombatSystem` se registrará como observador de la misma y recibirá las notificaciones de las acciones que ésta realice, actualizando así el contador de puntos de combate convenientemente.

Todas las criaturas alineadas en combate poseerán una instancia `CEntitySelector` en `CCombatSystem`. Dicha instancia servirá para ayudar a distinguirla cuando le llegue el turno y para identificar a un primer vistazo la alineación de combate de la misma. Los selectores de entidad se leerán de un archivo de datos "`.cbt`" a través de un `CCBTEngineParser` obtenido usando la base de datos del juego (a través de `iCGameDataBase`).

El trabajo con `CItem` será necesario cuando una criatura realice una acción de golpear con un ítem equipado. En ese caso, será necesario obtener la instancia a dicho ítem y hallar los puntos de acción asociados a la misma para actualizar convenientemente los de la criatura.

4.3.16. El manager de GFX.

Este será el módulo encargado de gestionar los efectos visuales del motor. Estos efectos visuales podrán ser considerados de dos tipos. Por un lado, los realizados por las plantillas de GFX (siglas de *Graphics FX*) definidas en un archivo de datos y asociables a entidades. Por otro lado, los que se realizarán de forma directa y que corresponderán al control de la intensidad de las sombras, el mantenimiento de los valores alpha para criaturas que puedan hacerse transparentes así como la realización de los efectos de fundido o fade. El diagrama general vendrá a ser el siguiente:

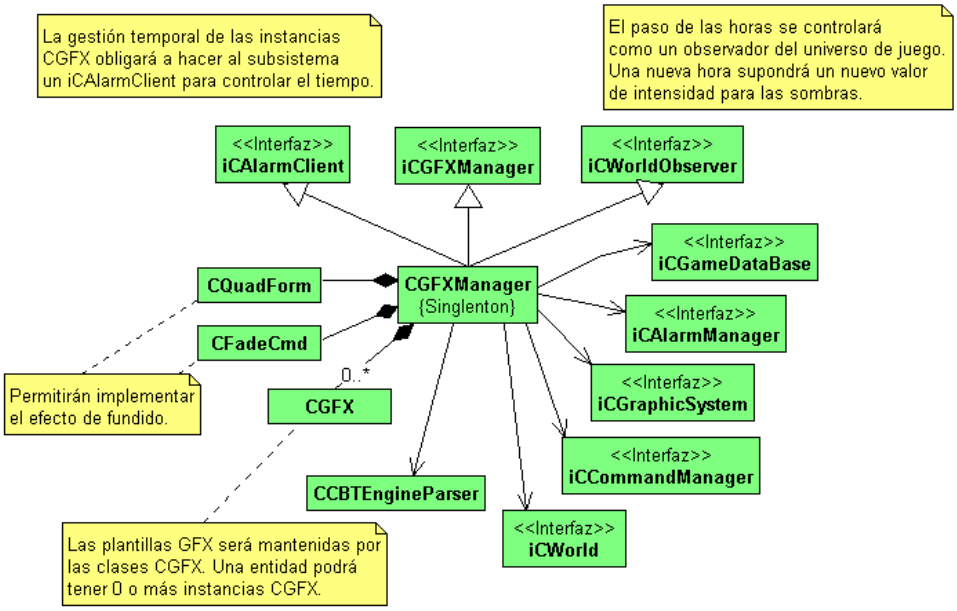


Diagrama del manager de GFX..

4.3.16.1. La clase CGFXManager.

La clase `CGFXManager` representará el núcleo del subsistema. Desde ella se realizará las acciones anteriormente comentadas. `CGFXManager` deberá de convertirse en un cliente del universo de juego para recibir las notificaciones relativas a las nuevas horas ya que en cada nueva hora el valor en la intensidad de las sombras podrá variar. También será necesario que herede de `iCAlarmClient` para instalar alarmas asociadas al tiempo de duración de las plantillas GFX sobre las entidades. Todas las definiciones de GFX en plantilla serán

representadas por instancias a clases `CGFX` las cuales serán mantenidas desde el módulo `CGFXManager`.

El efecto de fundido de pantalla, ampliamente utilizado en el motor, se implementará mediante el uso de un objeto dibujable `CQuadForm` que ocupará toda la pantalla y tendrá color negro. El uso del comando `CFadeCmd` controlará el nivel de transparencia del mismo.

4.3.16.2. La clase CGFX.

Esta clase representará un perfil de efecto especial definido. Tal y como se comentó, los efectos especiales podrán asociarse a las entidades para poder producir efectos tales como explosiones, mágias, etc. No dejarán de ser más que animaciones que podrán ubicarse en determinadas posiciones de una entidad y en un número arbritariamente grande. El diagrama de la clase será el siguiente:

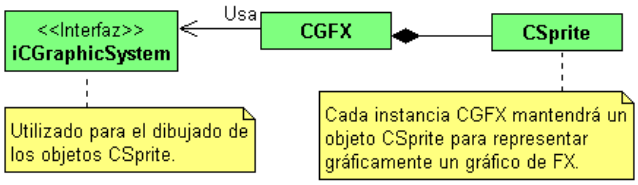


Diagrama de CGFX.

Todas las instancias de `CGFX` serán gestionadas por `CGFXManager`. Su dibujado en pantalla vendrá dado por una petición de las entidades, que solicitarán al manager el dibujado de todas las posibles instancias `CGFX` que tengan asociadas.

4.4. Diseño de CSCompiler.

CSCompiler será el compilador del lenguaje script *CrisolScript*. El compilador, a diferencia del resto de utilidades, y del propio motor, no estará construido en C++ sino en C. Esto se deberá a que la realización del mismo se apoya en la utilización de las utilidades *Flex* y *Bison* para la generación de un analizador léxico y sintático respectivamente [Aaby'96], [Donelly'95], [Johnson], [Niemann]. [Paxon]. Lo que harán realmente estas utilidades será generar código en C que contendrá los analizadores anteriormente mencionados. Para solucionar los problemas de compatibilidad en el trabajo con el código en C generado y C++, se trabajará directamente en C.

El compilador de *CrisolScript* se encargará de tomar un archivo con las definiciones globales (por ejemplo, "`Globals.cst`") de donde leerá las definiciones de constantes y variables globales junto con el nombre de los archivos script a compilar. Una vez procesados, generará un archivo llamado "`CrisolGameScripts.csb`", en formato binario, que contendrá el código intermedio para la inicialización de las posibles constantes y variables globales, junto a la de los distintos archivos script.

Durante el proceso de creación de *CSCompiler*, la documentación más importante utilizada durante la elaboración del compilador fue [Fauerby] (sin contar los manuales de *Flex* y *Bison* ya citados), mientras que en la parte destinada a la máquina virtual se acudió a diversas fuentes como [Cueva'98], [Lind'96], [Mak'96], [Venna], [Vennb], [Vennc], [Vennd] y [Venne].

A continuación se comentarán los aspectos fundamentales del compilador. Por un lado las diferentes partes o módulos del mismo, analizando su propósito y estructuras de datos utilizadas, y por otro la propia gramática, en notación EBNF, de *CrisolScript*.

4.4.1. Los módulos de CSCompiler.

El compilador de *CrisolScript* constará de diferentes módulos, en cada uno de los cuales se aplicará un tratamiento al código fuente de los scripts para que finalmente se pueda emitir el fichero, en formato binario, con el código intermedio equivalente [Cueva'98].

El diagrama general del proceso llevado a cabo por el compilador será el siguiente:

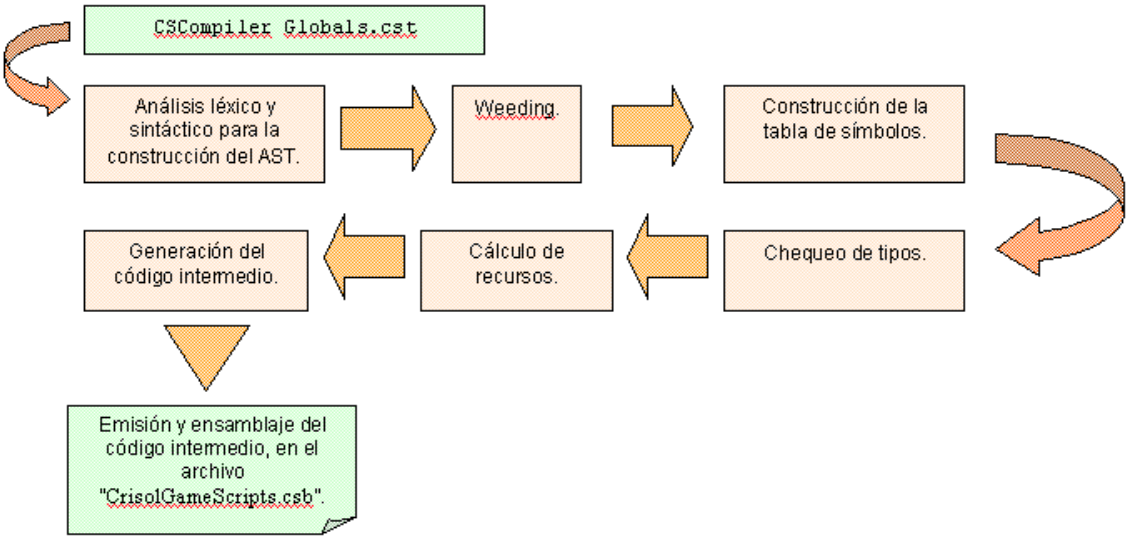


Diagrama del tratamiento llevado a cabo por el compilador.

Como se puede observar en el diagrama, el compilador tomará el código fuente de los distintos archivos script que se indiquen y los pasará por una etapa de análisis léxico y sintáctico, creándose con ello un árbol con todas las sentencias y expresiones. Dicho árbol, al que le denominaremos AST (de *Abstract Syntax Tree*), será esencial para que el compilador pueda trabajar.

El siguiente paso será el de depurar el AST construido con el fin de hallar problemas sintácticos que no hubieran sido localizados usando el analizador léxico y sintático generado por *Flex* y *Bison*. La idea es que el AST quede completado para proceder a realizar los verdaderos chequeos que nos lleven a la generación del código intermedio.

Después del módulo anterior llegará el verdadero trabajo en el tratamiento del AST. En primer lugar se creará la tábla de símbolos, que contendrá los identificadores de constantes, variables y funciones, así como la información asociada a los mismos. A continuación, se procederá a realizar un chequeo de tipos en las distintas expresiones halladas, con el fin de que las asignaciones y operaciones sean válidas. El cálculo de recursos y etiquetas será el paso previo antes de generar el código intermedio. En este paso se determinará el espacio local necesario en scripts y cuerpos de funciones para variables, además de las posiciones de salto en las estructuras de control. El último paso será el de proceder a emitir el código intermedio por cada cuerpo script y de función, ensamblándolo todo en un único archivo.

A continuación se analizarán con mayor detalle cada una de las etapas.

4.4.1.1. Analizador léxico y sintáctico.

Esta será la etapa que se lleve a cabo gracias al uso de las utilidades *Flex* y *Bison*. La primera generará un analizador léxico y la segunda un parser o analizador sintático. En *Flex* indicaremos los patrones que se deberán de reconocer en el código fuente y la forma en la que éstos serán pasados al analizador sintático generado por *Bison*. En dicho analizador sintático irá la gramática de *CrisolScript*, lo que permitirá que se puedan validar, en una forma aún primitiva, los archivos de código fuente.

Ante cualquier patrón de caracteres no reconocido por *Flex* o construcción sintática inválida, el compilador emitirá un mensaje de error informativo y abandonará el proceso de compilación.

Desde el parser creado por *Bison*, se irá construyendo el AST que contendrá las expresiones sintácticas. Dicho árbol será posteriormente recorrido múltiples veces para poder construir la tabla de símbolos, calcular recursos, chequear tipos o crear el código intermedio equivalente.

4.4.1.2. Preparación del AST; etapa de “Weeding”.

Esta etapa tendrá el objetivo de realizar un chequeo sencillo en el AST con el fin de localizar pequeños “bugs” en el código fuente que no pudieron hallarse en la etapa anterior, de ahí que el nombre del modulo sera Weeding, que en inglés viene a significar “*quitar la mala hierba*”.

Las acciones que se tomarán en el módulo implicará:

- Emisión de mensaje de error al no hallar una instrucción de retorno de una función que debe de devolver un valor.

- Emisión de un mensaje de error cuando se detecten divisiones por cero (siempre que se divida por una constante, ya que si es por un identificador no se podrá conocer el valor hasta que se esté en tiempo de ejecución).

- Advertir de la existencia de código después de una instrucción de retorno, tanto en el cuerpo de una función como en el de un script. Debemos de tener en cuenta que dicho código es seguro que no sea alcanzable nunca.

- Inicialización de todas las variables declaradas a las que no se haya asociado un valor. La inicialización será siempre a valores nulos (en el caso de un tipo `number` a 0, en el de un tipo `entity` a `NULL` y en el de un `string` a cadena vacía “”).

4.4.1.3. Módulo para la construcción de la tabla de símbolos.

En la tabla de símbolos se registrarán todos los identificadores de variables, constantes y funciones con el fin de que cuando sean usadas en expresiones, se conozca su tipo. La tabla de símbolos será representada por una tabla hash, lo que incrementará la velocidad en la localización de la información (los identificadores de variables, constantes y funciones serán cadenas de caracteres).

Para la construcción de la tabla de símbolos se generará realizando dos pasadas al AST. En la primera se realizará un trabajo completo sobre los identificadores de constantes y variables, que nada más registrarse serán chequeados siempre que se detecte su utilización en expresiones. Con respecto a las funciones, únicamente se almacenará su identificador en la tabla de símbolos ya que *CrisolScript* permitirá el uso de funciones desde una posición en la que la función utilizada no haya tenido que ser definida con anterioridad (de ahí la necesidad de dos chequeos). En la segunda pasada sí que se realizará una comprobación a fondo del uso de todos los identificadores de función en expresiones que fueron registrados previamente.

Desde este módulo también se realizará la importante tarea de comprobar qué funciones, de las declaradas, no han sido nunca utilizadas en el código fuente, con el fin de que sean completamente descartadas en el resto de procesos a llevar a cabo. Esto permitirá que se ahorre el tiempo que habría que invertir en chequearlas y, lógicamente, se evite la generación de código intermedio asociado a las mismas. Este paso tendrá fundamental importancia a la hora de trabajar con archivos de funciones importadas, donde lo normal podrá ser utilizar un número mínimo de las funciones declaradas en ese archivo al importarse para su uso en un script.

4.4.1.4. Módulo para el chequeo de tipos.

Este será uno de los módulos más complejos e importantes del compilador, pues en él se deberán de comprobar los tipos de cada una de las expresiones, con el fin de que éstas sean compatibles a la hora de aplicarles operadores lógicos, de asignación, de comparación, etc. Al mismo tiempo se deberá de realizar operaciones de cast, esto es, operaciones en las que se valide el trabajo entre dos expresiones de tipos distintos, pero que puedan ser compatibles haciendo que uno de ellos sea pasado al tipo del otro.

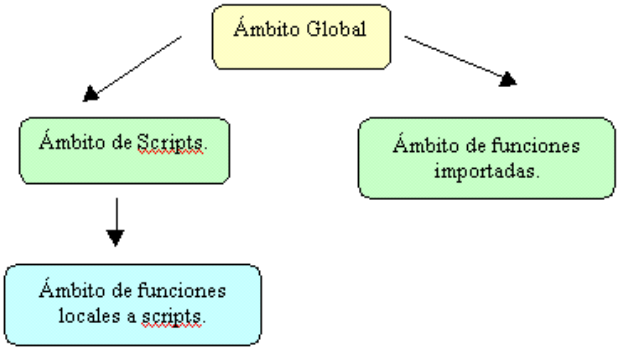
En *CrisolScript* los cast permitidos serán entre valores `number` y valores `string`, de tal forma que se pueda trabajar con ambos tipos en operaciones de comparación y asignación, realizando un cast siempre de derecha a izquierda, es decir, que si en la parte derecha hay un `number` y en la izquierda un `string`, se realizará el cast al `number` y viceversa. Los valores `entity` no tendrán cast, por lo que estará prohibido el trabajo de los mismos con cualquier otro tipo de dato distinto.

4.4.1.5. Módulo para el cálculo de recursos y etiquetas.

Este módulo ya formará parte de las preparaciones que se llevarán a cabo para la generación del código intermedio, pues es seguro que el código que se halle en el AST sea un código válido, listo para comenzar a ser transformado.

En el módulo se recorrerá el AST para calcular el numero de variables y constantes usadas en cada funcion y script, hallando también las etiquetas de salto asociadas a las sentencias de control.

La acción de calcular el número de variables y constantes supondrá calcular un offset por cada una de ellas con el fin de que, más tarde, la pila mantenida en la máquina virtual (ver diseño de *Crisol*) sepa dónde localizarlas. Además, en este proceso se deberá de tener en cuenta el concepto de ámbito. El siguiente diagrama mostrá el ámbito existente en *CrisolScript* sobre variables y constantes:



Ámbito en CrisolScript.

Como se puede observar, una variable o constante declarada en el ámbito global será accesible desde cualquier parte, mientras que en el caso de los scripts, éstas serán accesibles desde el cuerpo del script y desde el de las funciones locales al mismo. En el caso de las funciones locales a un script y funciones importadas, las variables y constantes declaradas en su interior, sólo serán accesibles desde su propio cuerpo de función. Nótese que las funciones importadas a un script no podrán hacer uso de las variables y constantes declaradas en éste último.

Además del cálculo de offsets a variables y constantes, en el módulo también se realizarán dos acciones más muy importantes. Por un lado, todas las funciones pasarán a tener un código numérico, de tal forma que desde los scripts se conozca cuándo se llama a una función utilizando su identificador numérico aquí asignado en lugar de su nombre. Por otro lado, se asignarán etiquetas (también identificadores numéricos) a las porciones de código donde las estructuras de control (`if` y `while`) puedan apuntar después de evaluar una expresión lógica.

4.4.1.6. Módulo de generación de código intermedio.

Aquí se procederá a crear los identificadores de código intermedio equivalmentes a las instrucciones existentes en el AST. La generación del código intermedio se realizará teniendo en mente el uso de la pila que la máquina virtual (implementada en *Crisol*) deberá de utilizar para interpretarlos. Para ello, durante la generación, se seguirán estas dos premisas:

- Toda sentencia dejará invariante la altura de la pila.

- Toda expresión aumentará en uno la altura de la pila.

Gracias a ellas, no habrá posibilidad de que existan incoherencias en el código generado. El listado de las instrucciones de código intermedio serán las siguientes (no se mostrarán los identificadores de código intermedio asociados a las instrucciones de llamada a funciones internas del motor):

`OP_NOP`	No se realizará operación alguna.
`OP_NNEG`	Negará un valor `number` en el tope de la pila.
`OP_NMUL`	Multiplicará dos valores `number` en el tope de la pila y depositará el resultado.
`OP_NADD`	Sumará dos `number` en el tope de la pila y depositará el resultado.
`OP_NMOD`	Hallará el módulo `number` en el tope de la pila y depositará el resultado.
`OP_NDIV`	Dividirá dos `number` en el tope de la pila y depositará el resultado.
`OP_NSUB`	Restará dos `number` en el tope de la pila y depositará el resultado.
`OP_SADD`	Concatenará dos `string` en el tope de la pila y depositará el resultado.
`OP_JMP`	Realizará un salto incondicional.
`OP_JMP_FALSE`	Realizará un salto si en el tope de la pila hay un valor menor de 1.
`OP_JMP_TRUE`	Realizará un salto si en el tope de la pila hay un valor mayor o igual a 1.
`OP_NJMP_EQ`	Realizará un salto si los dos valores `number` que haya en el tope de la pila son iguales.
`OP_NJMP_NE`	Realizará un salto si los dos valores `number` que haya en el tope de la pila son distintos.
`OP_NJMP_GE`	Realizará un salto si el primer `number` que haya en el tope de la pila, es mayor o igual que el siguiente.
`OP_NJMP_GT`	Realizará un salto si el primer `number` que haya en el tope de la pila, es mayor que el siguiente.
`OP_NJMP_LT`	Realizará un salto si el primer `number` que haya en el tope de la pila, es menor que el siguiente.
`OP_NJMP_LE`	Realizará un salto si el primer `number` que haya en el tope de la pila, es menor o igual que el siguiente.
`OP_SJMP_EQ`	Realizará un salto si los dos `strings` que hayan en el tope de la pila son iguales.
`OP_SJMP_NE`	Realizará un salto si los dos valores `string` que haya en el tope de la pila son distintos.
`OP_EJMP_EQ`	Realizará un salto si los dos valores `entity` que haya en el tope de la pila son iguales.
`OP_EJMP_NE`	Realizará un salto si los dos valores `entity` que haya en el tope de la pila son distintos.
`OP_DUP`	Duplicará el valor que haya en el tope de la pila.
`OP_POP`	Descartará el elemento que haya en el tope de la pila.
`OP_NRETURN`	Retornará del cuerpo de una función o script pasando un valor `number`.
`OP_SRETURN`	Retornará del cuerpo de una función o script pasando un valor `string`.
`OP_ERETURN`	Retornará del cuerpo de una función o script

	pasando un valor `entity`.
`OP_RETURN`	Retornará del cuerpo de una función o script sin pasar valor alguno.
`OP_NLOAD`	Leerá el valor asociado a una dirección de variable o constante `number` y lo colocará en el tope de la pila.
`OP_SLOAD`	Leerá el valor asociado a una dirección de variable o constante `string` y lo colocará en el tope de la pila.
`OP_ELOAD`	Leerá el valor asociado a una dirección de variable o constante `entity` y lo colocará en el tope de la pila.
`OP_NSTORE`	Guardará el valor asociado en el tope de la pila, sobre la dirección de variable o constante `number` asociada.
`OP_SSTORE`	Guardará el valor asociado en el tope de la pila, sobre la dirección de variable o constante `string` asociada.
`OP_ESTORE`	Guardará el valor asociado en el tope de la pila, sobre la dirección de variable o constante `entity` asociada.
`OP_NPUSH`	Colocará en el tope de la pila un valor `number`.
`OP_SPUSH`	Colocará en el tope de la pila un valor `string`.
`OP_EPUSH`	Colocará en el tope de la pila un valor `entity`.
`OP_NSCAST`	Realizará un cast de un valor `number` que se halle en el tope de la pila a un valor `string`.
`OP_SNCAST`	Realizará un cast de un valor `string` que se halle en el tope de la pila a un valor `number`.
`OP_CALL_FUNC`	Realizará una llamada a una función.

Tal y como se comenta en el apartado destinado a la máquina virtual en *Crisol*, estas instrucciones, y las no comentadas asociadas a las llamadas a funciones internas del motor, tendrán una representación en forma de clase.

El código intermedio generado será asociado al cuerpo global, a cada script y a cada función local e importada y estará dispuesto en una lista. Una vez generado, se simulará la ejecución de las instrucciones generadas (en todos los caminos posibles, mediante el uso de un algoritmo de backtracking) con el fin de conocer la altura máxima a la que podrá llegar la pila para operaciones intermedidas. Conocer este valor permitirá que en la máquina virtual se pueda prepararar, de forma eficiente, espacio en la pila cuando toque ejecutar el código de un script o función, o bien evitar la llamada cuando en el sistema no haya suficiente espacio en la pila para tratarlo.

4.4.1.7. Módulo de emisión y ensamblaje de código intermedio.

En este módulo se realizará el proceso final asociado al compilador. Por un lado la emisión de los identificadores del código intermedio generado en en el proceso anterior y por otro, el ensamblaje total en un único archivo denominado "`CrisolGameScripts.csb`". El compilador dará la opción de generar un archivo de texto con la información relativa al código intermedio emitido por cada script y cuerpo de función.

4.4.2. Gramática de CrisolScript.

A continuación se mostrará la gramática de *CrisolScript* en formato EBNF. No se incluirán todas las producciones asociadas a las llamadas de funciones de API, métodos `World`, `Game` y variables de tipo `entity`, al igual que tampoco se incluirán todos los tipos de script ya que en

todos los casos el formato será el mismo y de ponerlos complicariamos innecesariamente la tabla.

```
<global> ::= “global” <const> <var> <scripts>
<const> ::= “const” <const_decl_seq>
<const_decl_seq> ::=  <const_decl>
                    | <const_decl_seq> <const_decl>
<const_decl> ::= <type> <identifier_*> “:=” <exp> “;”
<var> ::= “var” <var_decl_seq>
<var_decl_seq> ::=  <var_decl>
                  | <var_decl_seq> <var_decl>
<var_decl> ::= <type> <var_identifier_seq> “;”
<var_identifier_seq> ::=  <identifier> <var_init_decl>
                        | <identifier> <var_init_decl> “,”
                          <var_identifier_seq>
<var_init_decl> ::= “:=” <exp>
<scripts> ::= <script_decl_seq>
<script_decl_seq> ::=  <compile_script> <script_decl>
                     | <script_decl_seq> <compile_script>
                       <script_decl>
<compile_script> ::= “compile” <string_value> “;”
<script_decl> ::= “script” <script_type> <import> <const> <var>
                  <l_function> <scope_stm>
<script_type> ::=  “OnStartGame” “(“ “void” “)”
                   [ Resto de métodos World ]
                 | “OnTradeItem” “(“ “entity” <identifier> “,”
                                     “entity” <identifier> “,”
                                     “entity” <identifier> “)”
<import> ::= <import_seq>
<import_seq> ::=  <import_decl> <g_function>
                | <import_seq> <import_decl> <g_function>
<import_decl> ::= “import” <string_value> “;”
<g_function> ::= <function_decl_seq>
<l_function> ::= “function” <function_decl_seq>
<function_decl_seq> ::=  <function_decl>
                       | <function_decl_seq> <function_decl>
<function_decl> ::= <return_type> “function” <identifier> “(“ <arguments> “)”
                    <const> <var> <scope_stm>
<return_type> ::= “void”
                | <type>
<arguments> ::= “void”
              | <argument_decl_seq>
<argument_decl_seq> ::=  <argument_decl>
                       | <argument_decl> “,” <argument_decl_seq>
<argument_decl> ::=  “ref” <type> <identifier>
                   | <type> <identifier>
<type> ::=  “number”
          | “string”
          | “entity”
<scope_stm> ::=  “begin” “end”
              | “begin” <stm_seq> “end”
<stm_seq> ::=  <stm>
            | <stm_seq> <stm>
<stm> ::=  “;”
         | “return” “;”
         | “return” <exp> “;”
         | “if” “(“ <exp> “)” “then” <stm>
         | “if” “(“ <exp> “)” “then” <stm> “else” <stm>
         | “while” “(“ <exp> “)” “do” <stm>
         | <scope_stm>
         | <exp> “;”
<exp> ::=  <identifier>
         | <number_value>
         | <string_value>
         | <identifier> “:=” <exp>
         | <exp> “==” <exp>
         | <exp> “!=” <exp>
         | <exp> “<” <exp>
```

```
          | <exp> “<=” <exp>
          | <exp> “>” <exp>
          | <exp> “>=” <exp>
          | <exp> “+” <exp>
          | <exp> “-“ <exp>
          | <exp> “*” <exp>
          | <exp> “/” <exp>
          | <exp> “%” <exp>
          | “-“ <exp>
          | “!” <exp>
          | <exp> “&&” <exp>
          | <exp> “||” <exp>
          | “(“ <exp> “)”
          | <identifier> “(“ <params> “)”
          | <api_function_call>
          | “game” “.” <gameobj_method_call>
          | “world” “.” <worldobj_method_call>
          | <identifier> “.” <entityobj_method_call>
<api_function_call> ::=  “APIPassToRGBColor” “(“ <params> “)”
                         [ Resto de llamadas API ]
                       | “APIDisableCrisolScriptWarnings” “(“ <params> “)”
<gameobj_method_call> ::=  “QuitGame” “(“ <params> “)”
                           [ Resto de métodos Game ]
                         | “EndCutScene” “(“ <params> “)”
<worldobj_method_call> ::= “GetAreaName” “(“ <params> “)”
                           [ Resto de métodos World ]
                         | “SetIdleScriptTime” “(“ <params> “)”

<entityobj_method_call> ::=  “GetName” “(“ <params> “)”
                             [ Resto de métodos Entity ]
                           | “HitEntity” “(“ <params> “)”
<params> ::= <params_seq>
<params_seq> ::=  <exp>
                | <exp> “,” <params_seq>
<identifier> ::= {<character> |<digit>}1..*
<number_value> ::=  {<digit>}1..*
                  | ({<digit>}1..* “.” {<digit>}0..*)
<entity_value> ::= {<digit>}1..*
<string_value> ::= {<character> | <digit>}0..*
<character> ::= “A” | “B” | ... | “Z” | “a” | “b” | ... “z”
<digit> ::= “0” | “1” | ... | “9”
```

Gramática de CrisolScript en formato EBNF.

4.5. Diseño de CrisolBuilder.

CrisolBuilder será una utilidad destinada a la verificación de todos los archivos de datos “`.cbt`” creados para *Crisol* con el fin de comprobar que sean válidos y coherentes con las definiciones especificadas en el archivo de reglas. Además, la utilidad se encargará de pasar algunos de dichos archivos a formato binario, con el fin de que *Crisol* pueda acelerar al máximo la carga de los mismos. En concreto, se encargará de pasar a binario los archivos de diseño de áreas, perfiles de entidades y plantillas de animación. El diagrama de la utilidad será muy sencilla:

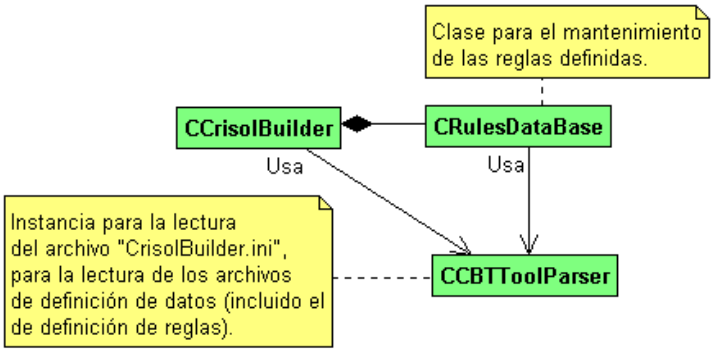


Diagrama de CrisolBuilder.

4.5.1. La clase CCrisolBuilder.

El núcleo de la utilidad estará representado por la clase `CCrisolBuilder` si bien, basará todo su funcionamiento en dos clases ya comentadas: `CRulesDataBase` y `CCBTToolParser`. Con la primera se mantendrá el archivo de reglas a partir del cual se comprobará la validez de todos los demás archivos de datos. Para la lectura de éstos, se hará uso de la instancia `CCBTToolParser` para el trabajo directo con archivos.

La clase `CCrisolBuilder` estará formada por un método público vital denominado `Build`. Internamente, dicho método se encargará de comprobar todos y cada uno de los archivos existentes realizando la conversión a binario cuando sea procedente. Para conocer los nombres de los archivos, la utilidad hará uso de un archivo de configuración denominado "`CrisolBuilder.ini`" en donde, además de los nombres de archivos de datos, se podrá especificar la forma en la que la utilidad mostrará la información, los mensajes de error, etc.

Debido a que todos los archivos dependerán, en lo relativo a su coherencia, al archivo de reglas, cada vez que se utilice *CrisolBuilder* se realizará un chequeo completo de todos los archivos para evitar posibles efectos laterales.

4.6. Diseño de CPAKTool.

`CPAKTool` será una utilidad muy sencilla destinada a la gestión del archivo empaquetado "`GameData.pak`", de uso obligatorio en el engine. Como ya se comentó en el apartado destinado al diseño de *Crisol*, éste mantendrá un subsistema de ficheros capaz de gestionar archivos empaquetados. En teoría, el motor será capaz de trabajar con un número arbritario de este tipo de ficheros sin embargo, en la prática, utilizará uno solo que será, el ya citado, "`GameData.pak`". Con esta utilidad se podrán añadir ficheros, borrar, obtener listado del número de ellos contenidos así como extraerlos. También se podrá crear el fichero "`GameData.pak`" vacío. El diagrama de la utilidad será muy sencillo:

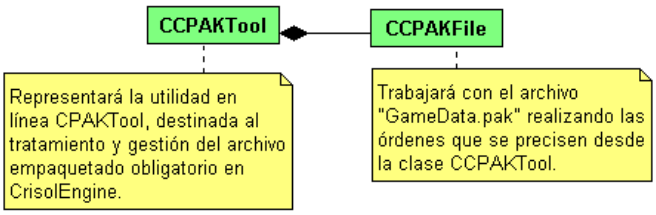


Diagrama de CPAKTool.

4.6.1. La clase CCPAKTool.

La clase `CCPAKTool` será el núcleo de la utilidad y su misión será la de procesar las órdenes que provengan del exterior. Cada orden supondrá crear una instancia `CCPAKFile` asociada al archivo "`GameData.pak`" y ejecutar el método que corresponda. Todas las operaciones necesitarán que exista el archivo "`.pak`" citado, menos la relativa a la creación del mismo.

5. Organización de la implementación.

En este apartado se revisará la organización en los ficheros utilizados para implementar *Crisol* y el resto de herramientas ya comentadas, esto es, *CrisolBuilder*, *CSCompiler* y *CPAKTool*. También se comentarán las notaciones de programación principales utilizadas, de cara a que otra persona ajena al desarrollo del proyecto pueda hacerse con el control del mismo de forma relativamente rápida.

5.1. Notaciones de programación generales utilizadas.

Durante el desarrollo de los diferentes proyectos, se ha tendido a la utilización de una notación que hiciera consistente todo el código realizado. Este ha sido uno de los puntos fundamentales para poder trabajar a largo plazo [Lakos'96], ya que el paso del tiempo ejerce unos efectos muy negativos sobre el conocimiento que el propio desarrollador tiene de lo que ha creado. Contar con archivos de programación documentados para cada método y clase definido fue muy importante.

5.1.1. Uso de una notación unificada para programar.

La práctica totalidad de tipos de datos en todos los proyectos están declarados utilizando notación húngara. Esta notación, inventada por un director de programación en Microsoft, permite obtener información del tipo de una variable a un simple vistazo pues se fundamenta en poner como prefijo en las mismas la inicial del tipo que representa.

En todos los proyectos se utilizarón tipos predefinidos, es decir, se creo un archivo de definiciones con el siguiente formato:

```
typedef unsigned char    byte;
typedef char             sbyte;
typedef unsigned short   word;
typedef short            sword;
typedef unsigned long    dword;
typedef long             sdword;
typedef unsigned __int64 qword;
typedef __int64          sqword;
```

Tipos de datos

En la tabla, se pueden ver los tipos de datos. Para cada uno de ellos tendremos los siguientes identificadores utilizados en notación húngara [Petzold'96]:

Tipo de dato	Notación húngara	Ejemplo
byte	ub	byte ubValue;
sbyte	sb	sbyte sbValue;
word	uw	word uwValue;
sword	sw	sword swValue;
dword	dw	dword sdValue;
sdword	sd	sdword sdValue;
qword	uq	qword uqValue;

En el caso de estructuras, todas comenzarán con una 's' asociada al nombre de la misma, no a la instancia que se cree a partir de ella. Por ejemplo:

```
struct sEjemploEstructura {
 // ....
};
```

En el caso de las clases ocurrirá algo muy parecido, salvo que siempre se comenzará a definir una utilizando el carácter 'C'. Por ejemplo:

```
Class CEjemploClase {
```

```
 // ...
};
```

Añadir que en las clases, todas las variables de miembro comenzarán con "m_" para indicar que pertenecen a la clase.

Las interfaces será declaradas siempre como estructuras, con la particularidad de que comenzarán con el carácter 'i' seguido de 'C'. Las interfaces serán utilizadas para implementar el patrón bridge. Por ejemplo:

```
struct iCEjemploClase {
 // ...
};
```

En este caso, podríamos tener una interfaz al ejemplo de clase declarado más atrás.

En los enumerados, al igual que en las estructuras y clases, se utilizará el carácter 'e'. Por ejemplo:

```
enum eIDTypes {
 // ...
};
```

Además de utilizar este tipo de notación, los archivos de implementación ".cpp" siempre tendrán, por cada método, una cabecera común para describirlo. La cabecerá tendrá el siguiente formato:

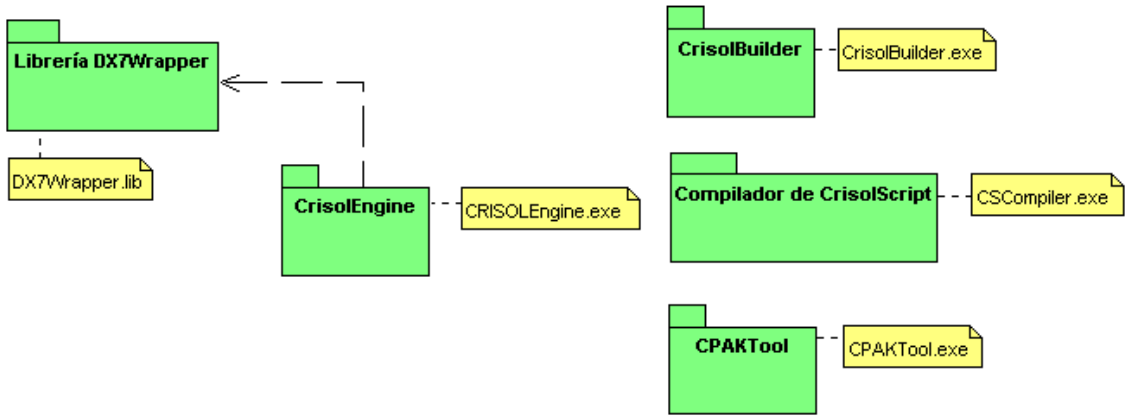
```
//////////////////////////////////////////////////////////////////////
// Descripcion:
// Parametros:
// Devuelve:
// Notas:
//////////////////////////////////////////////////////////////////////
```

Cabecera utilizada para los métodos.

Usando este sistema, se reduce de forma notable el trabajo de identificación de funcionamiento de los métodos asociados a las clases.

5.2. Organización de los proyectos.

Todos los proyectos serán gestionados desde VisualStudio. Habrá uno destinado a *Crisol* como aplicación, otro al compilador de *CrisolScript* y otros dos más destinados a las herramientas. Uno para *CrisolBuilder* y otro para *CPAKTool*. He aquí el diagrama con la organización de los proyectos existentes con respecto a la compilación de los mismos:



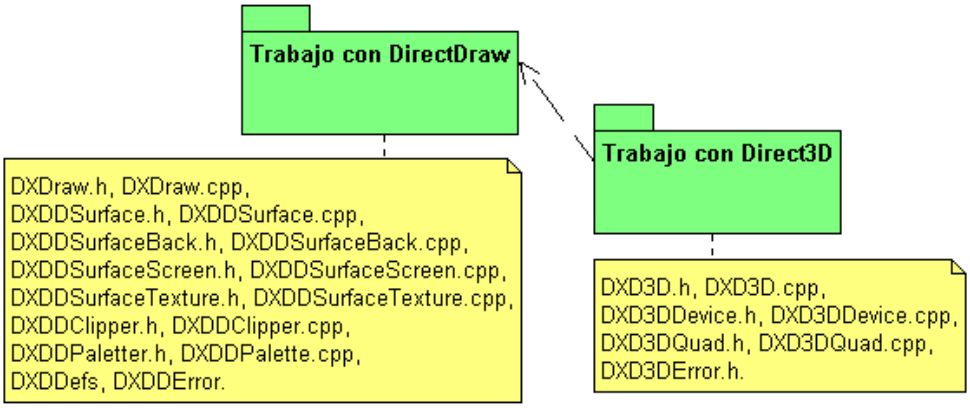
Organización de los proyectos.

Como se puede observar en el diagrama, también se deberá de tener en cuenta el proyecto destinado a crear la librería estática para el trabajo con la librería DirectX.

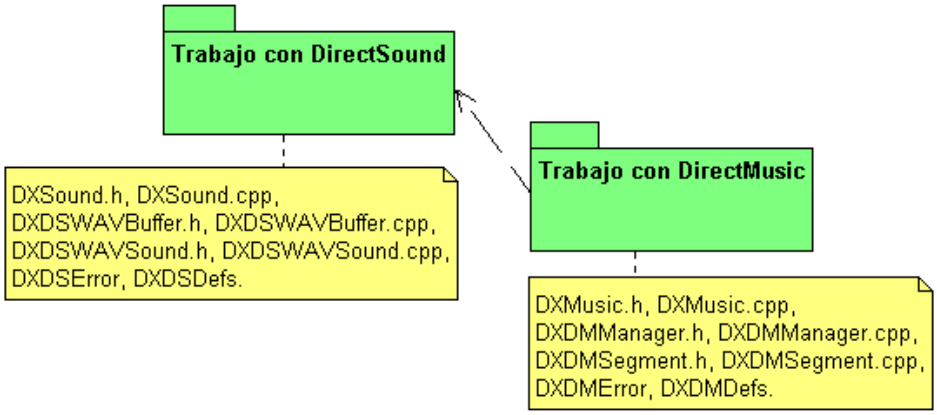
5.2.1. El proyecto DX7Wrapper.

Al compilar el proyecto se obtendrá la librería estática "`DX7Wrapper.lib`", que posteriormente será utilizada en *Crisol* para poder trabajar con DirectX.

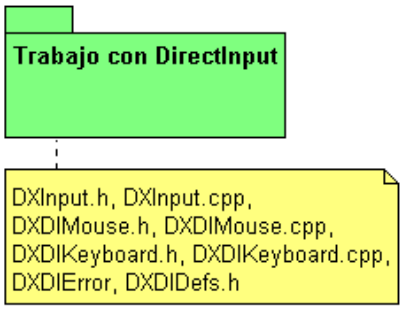
Los diagramas organizativos serán los siguientes:



Organización de las clases relativas a DirectDraw y Direct3D.



Organización de las clases relativas a DirectSound y DirectMusic.



Organización de las clases relativas a DirectInput

5.2.2. El proyecto CRISOLEngine.

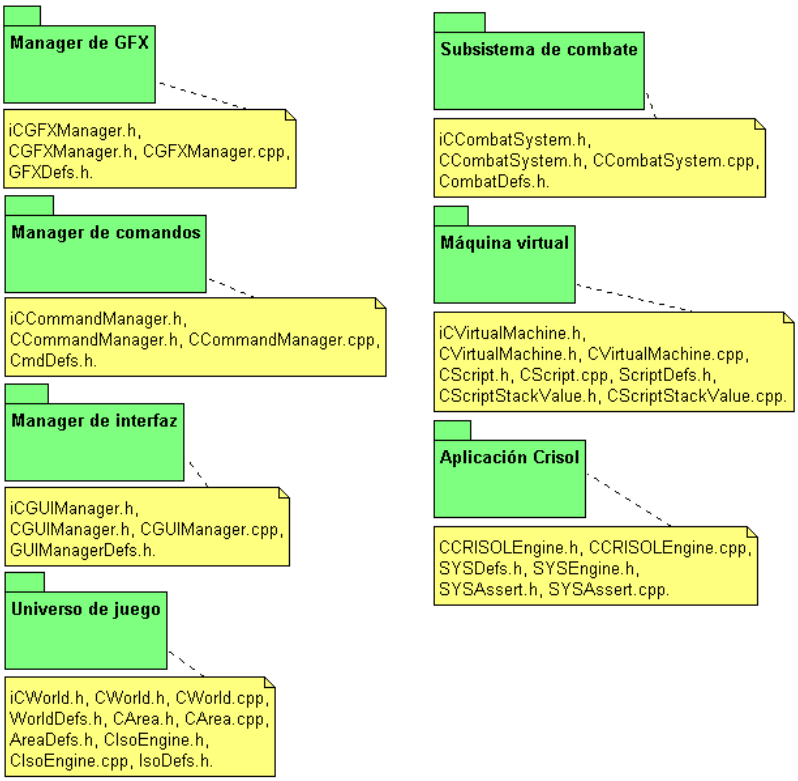
Este será, lógicamente, el proyecto más complejo de todos pues representará a la aplicación *Crisol*. La compilación del mismo estará necesitada del uso de la librería estática "`DX7Wrapper.lib`". También se deberán de tener en cuenta una serie de definiciones a establecer en el preprocesador y que serán:

- `_CRISOLENGINE`. Para indicar que se está compilando la aplicación *Crisol*. Esto será muy importante ya que algunas clases serán reutilizables entre proyectos y en ese caso, se deberán de adaptar según sea para el que se compilen (lease los parsers det exto y la clase encargada de mantener las reglas de juego).

- `_SYSASSERT`. Para indicar que se quieren utilizar los asertos.

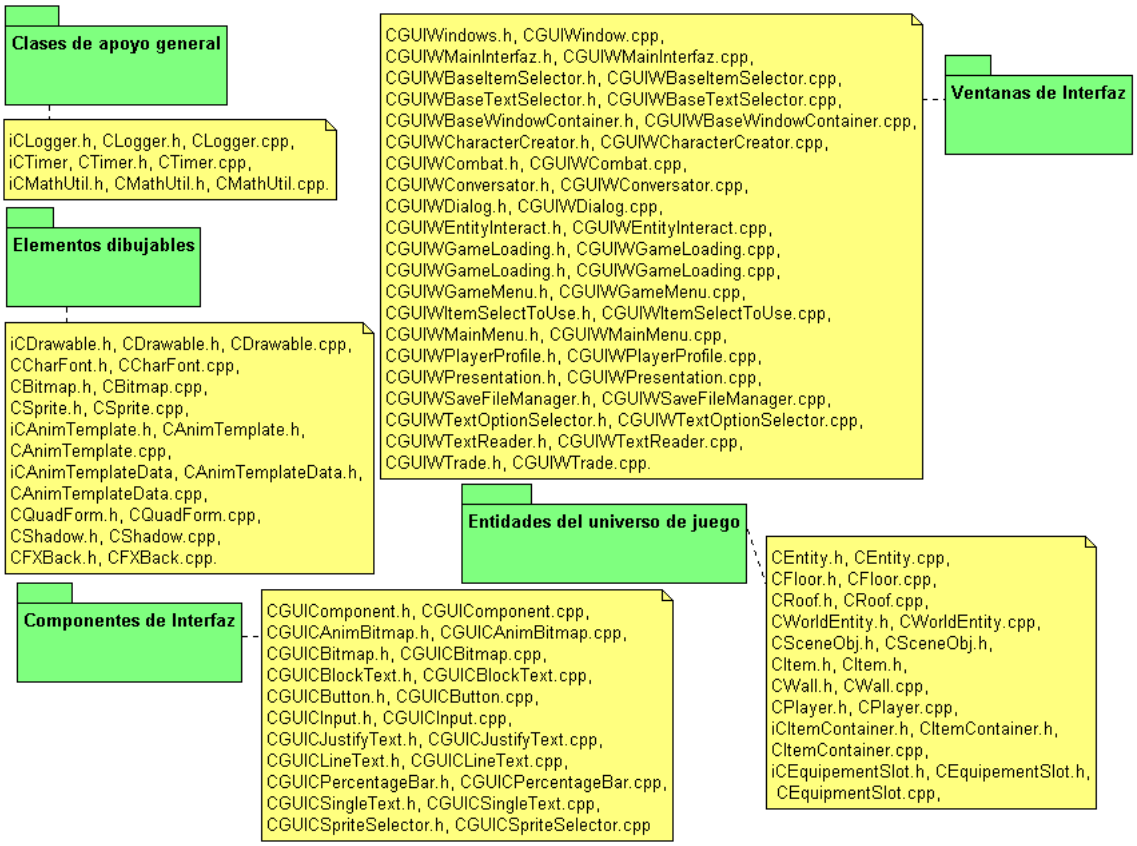
- `ENGINE_TRACE`. Para indicar que se quiere enviar información al logger.

A continuación se mostrará la organización del proyecto:



Organización de los subsistemas principales.

Además de los subsistemas principales, habrá que tener en cuenta el siguiente diagrama que contendrá a aquellas clases que sirvan de apoyo a los mismos. La organización en este caso será la siguiente:



Organización de clases de apoyo.

5.2.3. El proyecto CrisolScriptCompiler.

El compilador de *CrisolScript* estará construido en C y hará uso de las herramientas *Flex* y *Bison*. Para poder trabajar con las mismas, se utilizarán los archivos "`CrisolScript.l`" y "`CrisolScript.y`" respectivamente.

En el caso de utilizar *Flex*, se deberá de ejecutar con:

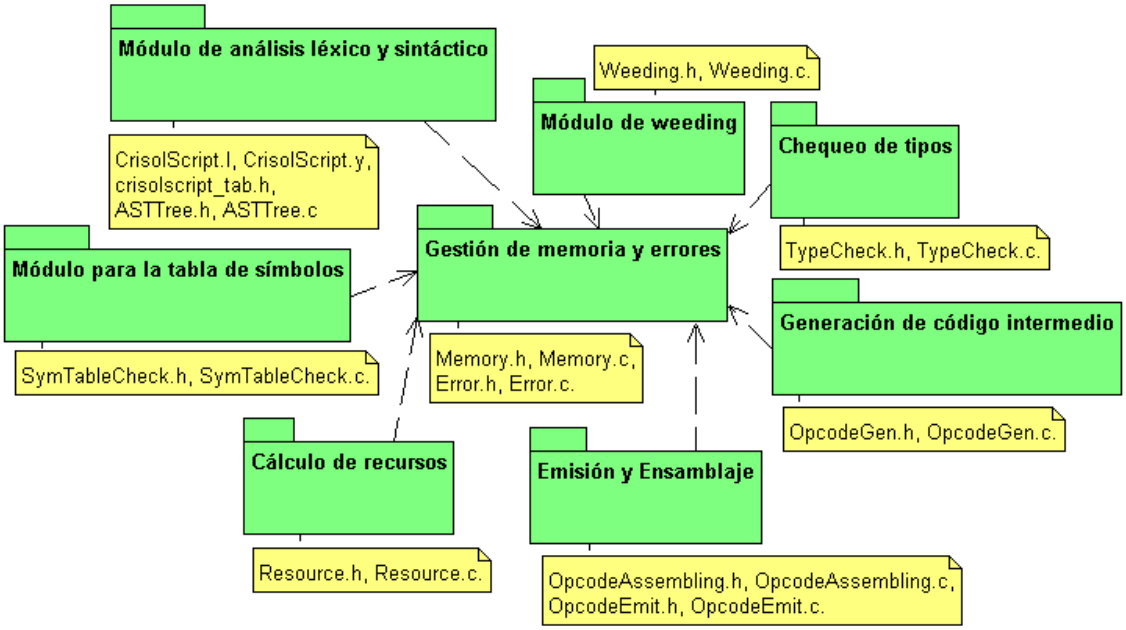
```
Flex -i CrisolScript.l
```

Al quererse utilizar un analizador que no tenga en cuenta la distinción entre mayúsculas y minúsculas se usará el parámetro "`-i`".

En el caso de utilizar *Bison*, se escribirá:

```
Bison -d CrisolScript.y
```

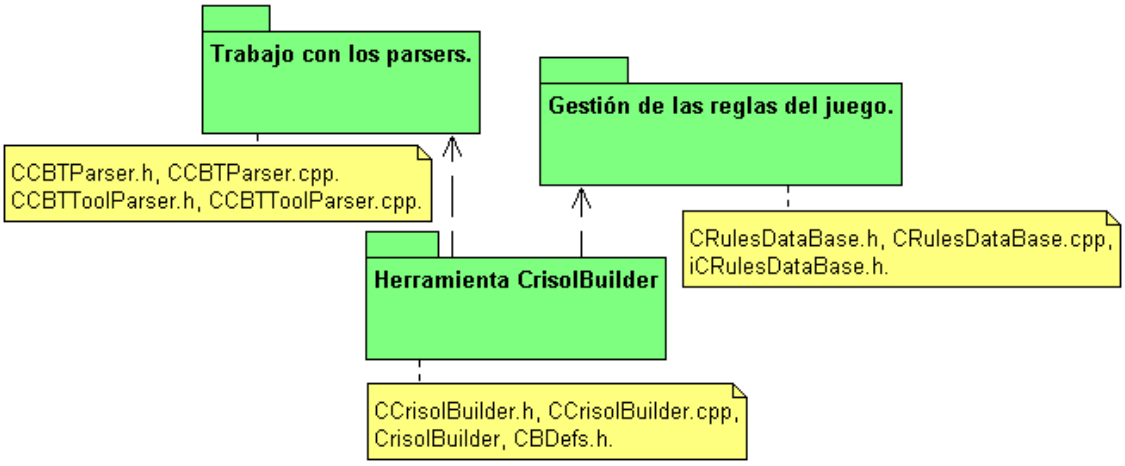
He aquí la organización del proyecto:



Organización de CSCompiler.

5.2.4. El proyecto CrisolBuilder.

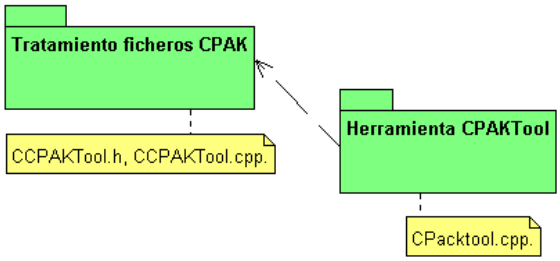
Esta herramienta, al igual que *CPAKTool*, no tendrá una complejidad especial. La organización del proyecto será el siguiente:



Organización de CrisolBuilder.

5.2.5. El proyecto CPAKTool.

Esta herramienta, al ser la más sencilla de todas, poseerá una organización muy elemental. La compilación del proyecto no requerirá ningún tipo de característica especial. La organización del proyecto será la siguiente:



Organización del proyecto CPAKTool.

5.3. Requerimientos hardware y software.

Los requerimientos de ejecución de las herramientas *CrisolBuilder* y *CPAKTool*, así como del compilador *CSCompiler*, no serán relevantes (salvo el hecho de que cuanto más rápido sea el equipo utilizado antes se procesarán los datos).

En el caso de *Crisol* sí que se deberá de tener en cuenta que el equipo deberá de poseer una tarjeta gráfica 3D compatible con DirectX y otra de sonido. En el caso de la tarjeta gráfica, ésta deberá de existir, además, de forma conjunta con la convencional 2D, es decir, no serán aceptados equipos formados por una tarjeta 3D y otra 2D. Todos los requerimientos son considerados como estándar a día de hoy.

La plataforma de ejecución será Windows, habiéndose probado con éxito en Windows 95 y 98, Millenium, 2000 y XP.

6. Bibliografía.

[Aaby'96]
Aaby, Anthony A.. *Compiler Construction using Flex and Bison*.
http://cs.wwc.edu/~aabyan/464/Book/

[Aho'98]
Aho, V. Alfred, Ravi Sethi y Jeffrey D. Ullman. *Compiladores, Principios, técnicas y herramientas*. Addison Wesley, 1998.

[Arévalo'01]
Arévalo, Javier. *Main Loop with Fixed Time Steps.*
http://www.flipcode.com/cgi-bin/msg.cgi?showThread=Tip-MainLoopTimeSteps&forum=totd&id=-1

[Arévalo'00]
Arévalo, Javier. *Object Initialization And Finalization Protocol*
http://www.flipcode.com/cgi-bin/msg.cgi?showThread=Tip-ObjectInitFinal&forum=totd&id=-1

[Baeza'00]
Baeza, Juan Carlos Arévalo. *Locals Functions in C++*.
http://www.flipcode.com/cgi-bin/msg.cgi?showThread=Tip-LocalFunctionsInCPP&forum=totd&id=-1

[Bargen'98]
Bargen, Bradley y Peter Donnelly. *A fondo DirectX*. Mc Graw Hill, 1998.

[Boer'00a]
Boer, James. "Object-Oriented Programming and Desing Techniques". *Game Programming Gems*. Charles River Media, 2000.

[Boer'00b]
Boer, James. "Using the STL in Game Programming". *Game Programming Gems*. Charles River Media, 2000.

[Clark'96]
Clark, Jonathan. "Object Cache Management". *Game Developer*, pág 52. Febrero 1996.

[Creighton'00]
Creighton, Jon. *Fast Allocation Pool*.
http://www.flipcode.com/cgi-bin/msg.cgi?showThread=COTD-FastAllocationPool&forum=cotd&id=-1

[Cueva'95]
Cueva Lovelle, Juan Manuel. *Análisis Sintáctico en Procesadores de Lenguaje*. Cuaderno didáctico nº 61. Departamento de Matemáticas. Universidad de Oviedo, 1995.

[Cueva'98]
Cueva Lovelle, Juan Manuel. *Conceptos Básicos de Procesadores de Lenguaje*. Cuaderno didáctico nº 10. Departamento de Informática. Universidad de Oviedo, 1998.

[Cueva'99]
Cueva Lovelle, Juan Manuel y José Emilio Labra Gayo. *Intérpretes. Procesadores de Lenguaje*. Universidad de Oviedo. Curso 99-00.

[Dawson'99]
Dawson, Bruce. "Structured Exception Handling". *Game Developer*, pág. 52. Enero 1999.

[Donelly'95]
Donelly, Charles y Richard Stallman. *Bison, the YACC-compatible Parser Generator*.
http://dinosaur.compilertools.net/bison/

[Eckel'00]

Eckel, Bruce. *Thinking In C++ Second Edition*. Prentice Hall, 2000.

[Fauerby'01]
Fauerby, Kasper. *PxdScript Language Tutorial Series*.
http://www.peroxide.dk/tuts_scr.shtml

[Fötch'00]
Fötch, Michael. *Fast Bitmap Fonts for DirectDraw*.
http://64.33.37.114/BmpFonts.htm

[Gamma'95]
Gamma, Erich, Helm Richard, Ralph Johnson y John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[Heyes'01]
Heyes-Jones, Justin. *A* Algorithm Tutorial*.
http://www.geocities.com/jheyesjones/astar.html

[Huebner'97]
Huebner, Robert. *Adding Languages to Game Engines*.
http://www.gamasutra.com/features/19971003/huebner_01.htm

[Huebner'99]
Huebner, Robert. *Using Java as an embedded game scripting language*.
http://nihilistic.com

[Iqbal'00]
Iqbal, Gaz. *Reducing Dependencies in C++*.
http://www.flipcode.com/tutorials/tut_cppdepend.shtml

[Johnson]
Johnson, Stephen C.. *Yacc: Yet Another Compiler-Compiler*.
http://www.cs.utexas.edu/users/novak/yaccpaper.htm

[Lakos'96]
Lakos, John. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[Lindholm'96]
Lindholm, Tim y Frank Yellin. *The Java Virtual Machine Specification.*
http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html

[Mark'96]
Mak, Ronald. *Writing Compilers and Interpreters, Second Edition*. Wiley, 1996

[Meyers'96]
Meyers, Scott. *More Effective C++*. Addison-Wesley, 1996.

[Meyers'97]
Meyers, Scrott. *Effective C++, Second Edition*. Addison-Wesley, 1997.

[Meyers'01]
Meyers, Scott. *Effective STL*. Addison-Wesley, 2001.

[Niemann]
Niemann, Thomas. *A Compact Guide to Lex & Yacc*.
http://epaperpress.com/lexandyacc/

[Niestadt'99]
Niestadt, Jan. *Implementing a Scripting Engine*.
http://www.flipcode.com/tutorials/tut_scr01.shtml

[Olse'00]
Olse, John. "Interpolation Methods". *Game Programming Gems*. Charles River Media, 2000.

[Pazera'99]
Pazera, Ernest. *16-bit Color.*
http://www.gamedev.net/reference/articles/article863.asp

[Pazera'01]
Pazera, Ernest. *Isometric Game Programmin with DirectX 7.0*. Prima Tech, 2001

[Paxson]
Paxson, Vern. *Flex*.
http://dinosaur.compilertools.net/flex/manpage.html

[Peers'99]
Peers, Bert. *Writing Modifiable Games*.
http://users.chello.be/sf15772/programming/modgame/

[Petzold'96]
Petzold, Charles. *Programación en Windows 95*. Mc Graw Hill, 1996.

[Ranck'00]
Ranck, Steven. "Frame-Based Memory Allocation". *Game Programming Gems*. Charles River Media, 2000.

[Rabin'00a]
Rabin, Steve. "A* Aesthetic Optimizations". *Game Programming Gems*. Charles River Media, 2000.

[Rabin'00b]
Rabin, Steve. "A* Speed Optimizations". *Game Programming Gems*. Charles River Media, 2000.

[Root'99]
Root, Michael y James Boer. *DirectX Complete*. Mc Graw Hill, 1999.

[Rollings'00]
Rollings, Andrew y Dave Morris. *Game, Architecture and Design*. Coriolis, 2000.

[Rued'01]
Rued, Thomas. *Memory Leaks In VC*.
http://www.flipcode.com/cgi-bin/msg.cgi?showThread=Tip-MemoryLeaksInVC&forum=totd&id=-1

[Salking'98]
Salkin, Steve. "Design Patterns for Game Development". *Game Developer*, pág. 42. Julio 1998.

[Saltzman'99]
Saltzman, Marc. *Game Desing, Secrets of the Sages*. BradyGames, 1999.

[Simmons'00a]
Simmons, Gary. *2D Alpha Blending*.
http://64.33.37.114/ddab.html

[Simmons'00b]
Simmons, Gary. *Direct Draw Enumeration*.
http://64.33.37.114/ddenum.html

[Simmons'00c]
Simmons, Gary. *Transparent and Translucency*.
http://64.33.37.114/transparency.html

[Simmons'00d]
Simmons, Gary. *Texture Stage*.
http://64.33.37.114/texturestages.html

[Stroustrup'97]
Bjarne, Stroustrup. *El Lenguaje de Programación C++, Tercera Edición*. Addison-Wesley, 1997.

[Stout'00]
Stout, Bryan. "The Basics of A* for Path Planning". *Game Programming Gems*. Charles River Media, 2000.

[Towner]
Towner, Jesse. *Resource Files Explained*.
http://www.gamedev.net/reference/articles/article902.asp

[Venners'a]
Venners, Bill. *Bytecode basics*.
http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes_p.html

[Venners'b]
Venners, Bill. *The Java class file lifestyle*.
http://www.javaworld.com/jw-07-1996/jw-07-classfile-p2.html

[Venners'c]
Venners, Bill. *The lean, mean, virtual machine*.
http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm_p.html

[Venners'd]
Venners, Bill. *How the Java virtual machine handles method invocation and return*.
http://www.javaworld.com/javaworld/jw-06-1997/jw-06-hood.html

[Venners'e]
Venners, Bill. *Control Flow*.
http://www.javaworld.com/javaworld/jw-03-1997/jw-03-hood.html

[Wolverson'00]
Wolverson, Herbert. *Using Direct3D for 2D Tile Rendering*.
http://www.gamedev.net/reference/articles/article935.asp