Introducción al desarrollo de juegos bajo QGAMES

2D y 3D

Ignacio Cea Fornies

2022

# Índice

[Índice 1](#_Toc80263804)

[Prologo 3](#_Toc80263805)

[Agradecimientos 5](#_Toc80263806)

[Introducción 6](#_Toc80263807)

[Bloque 0: Instalando QGAMES 10](#_Toc80263808)

[Configuración de las librerías requeridas por QGAMES 11](#_Toc80263809)

[Configuración del entorno 13](#_Toc80263810)

[Configuración de QGAMES 16](#_Toc80263811)

[Herramientas asociadas 18](#_Toc80263812)

[Bloque 1: La arquitectura básica de QGAMES y el módulo Common 19](#_Toc80263813)

[La arquitectura básica de QGAMES 20](#_Toc80263814)

[Wrappers 21](#_Toc80263815)

[Módulos básicos 21](#_Toc80263816)

[Módulos avanzados 22](#_Toc80263817)

[Módulos plantilla de juegos 22](#_Toc80263818)

[Las clases básicas de un juego. Foundation 23](#_Toc80263819)

[El núcleo básico de un juego: Las clases *Game* y *GameImplementation* 24](#_Toc80263820)

[La clase *Game* 24](#_Toc80263821)

[La clase *GameImplementation* 27](#_Toc80263822)

[Como dibujar en QGAMES. Las clases *Screen*, *Camera* y *Transaction* 32](#_Toc80263823)

[La clase *Screen* 32](#_Toc80263824)

[La clase *Camera* 40](#_Toc80263825)

[La clase *Transaction* 49](#_Toc80263826)

[¿Y cómo encajan estas tres piezas? 50](#_Toc80263827)

[La representación de la realidad. La clase *Element* 53](#_Toc80263828)

[El ciclo de vida de Element 56](#_Toc80263829)

[La comunicación interna entre los elementos de un juego. El modelo *Notifier* – *Observer* 58](#_Toc80263830)

[La clase *Event* 58](#_Toc80263831)

[Las clases *Notifier* y *Observer* 59](#_Toc80263832)

[Contadores y Switches 62](#_Toc80263833)

[La clase *Counter* 62](#_Toc80263834)

[La clase *Counters* 64](#_Toc80263835)

[La clase *OnOffSwitch* 66](#_Toc80263836)

[La clase *OnOffSwitches* 66](#_Toc80263837)

[Agregadores de datos. Las clases *OpenValue* y *SetOfOpenValues* 68](#_Toc80263838)

[La clase *OpenValue* 68](#_Toc80263839)

[La clase *SetOfOpenValues* 69](#_Toc80263840)

[Otra vez la clase *Element*. Añadir contadores y switches y salvar y recuperar su estado 72](#_Toc80263841)

[Incluyendo Contadores 72](#_Toc80263842)

[Incluyendo Switches 74](#_Toc80263843)

[Salvando y recuperando el estado de un *Element* 74](#_Toc80263844)

[Dando movimiento a los elementos de un juego. La clase *Movement* 78](#_Toc80263845)

[Movimientos básicos 79](#_Toc80263846)

[Movimientos complejos 80](#_Toc80263847)

[Encadenar movimientos 84](#_Toc80263848)

[Animando los elementos de un juego. *Forms*, *Frames*, *Animations* and *States* 85](#_Toc80263849)

[Formas. La clase *Form* y la clase *Frame* 85](#_Toc80263850)

[Form y Frame para 2D. Las clases *Sprite2D* y *Sprite2DFrame* 91](#_Toc80263851)

[Animación 100](#_Toc80263852)

[Anexos 104](#_Toc80263853)

[Ilustraciones 105](#_Toc80263854)

[Ecuaciones 106](#_Toc80263855)

Prologo

Mi afición por los juegos viene de largo.

Tenía 13 años cuando mi padre me introdujo en el fabuloso mundo de los ordenadores y la programación, pero no fue hasta que tuve más o menos 15 cuando fui más allá de copiar los que leía en revistas especializadas de la época, y desarrollé para un Commodore64® mi primer juego “serio” utilizando código máquina (¡que no ensamblador!).

Texto

Descripción generada automáticamente

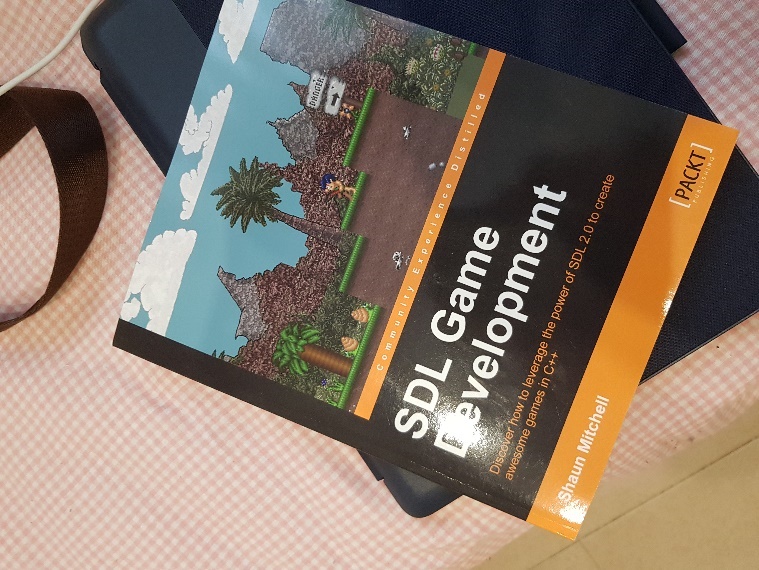
1: Uno de esos juegos que copiaba (de la revista Commodore Magazine número 1)

Desde aquel día me he divertido, relajado, y crecido intelectualmente desarrollando programas de ordenador, aunque no tanto juegos.

Fue en 2014 cuando recuperé la afición de antaño por el desarrollo de juegos. Me picaba la curiosidad de entender (pero sobre todo de hacer) como funcionaba algo tan maravilloso como el **Super Mario Bross®**: ¿Cómo era posible conseguir esa suavidad de movimientos en la pantalla con tantos elementos en movimiento? ¿Cómo se podía hacer para que aparecieran objetos y más objetos por la derecha como si me los encontrara al andar? ¿Cómo se hacían esos increíbles movimientos de salto, nado, o vuelo? ¡Guau! ¡Cuánto retos intelectuales!

Hubiera sido fácil, quizás, no lo sé, buscar bibliografía adecuada o artículos al respecto en internet. ¡Seguro que algo hubiera encontrado! Pero no era eso en realidad lo que yo buscaba. Buscaba demostrarme que era capaz de deducirlo todo por mí mismo, pensando y razonando sobre ello. En realidad, ahora lo sé, ese era (y es) el verdadero reto.

La carne es débil y pequé un poco… me compré un libro:



Simplemente maravilloso. Lo leí en un fin de semana y comencé mi viaje.

Hice un primer y muy básico juego. En realidad, sólo para entrenar lo aprendido, pero empecé a pensar en la creación de una librería que me ayudara y simplificara el desarrollo de los siguientes. Quizás influyó que soy ingeniero y quizás más, que parte de mi vida profesional la he desarrollado como arquitecto de software. No sé, pero sentí que eso es lo que quería hacer.

Varios años después tengo una librería. Siempre incompleta. Siempre en desarrollo. Siempre en mejora. La llamé *QGAMES* (de Quality Games). Tiene más de 1 millón de líneas de código. Quizás me pasé, pero ahora soy capaz de hacer un juego de calidad y complejo en pocos meses y ¡sólo dedicándole fines de semana y noches!

He pensado que os puede ser de utilidad y por eso he decidido escribir este documento.

Me gustaría que fuera parte de mi legado.

Espero que lo disfrutéis, mejoréis y crezcáis.

Papi, tú también. Gracias otra vez.

He creado una página web en sourceforge.com para recoger todos vuestros comentarios: <https://sourceforge.net/projects/qgames-library/>

Agradecimientos

Nuevamente a mi familia. A mi mujer, Mari Paz, y a mis hijas Sandra y María.

A tod@s aquell@s cuyos artículos, blogs, posts, etc. me inspiraron, de una u otra manera, para la creación de esta librería.

A tod@s aquellos que tan generosamente cuelgan en internet recursos (gráficos, sonidos, etc.) que nos facilitan, a los que no tenemos imaginación, el desarrollo de juegos. Se lo devuelvo en forma de librería.

Introducción

Los principios básicos de programación de cualquier juego por ordenador son relativamente sencillos de entender. Quizás lo más complejo de llevar a cabo sea la optimización de los mismos; es decir, conseguir que todas las tareas que se realizan en un juego se realicen lo más rápidamente posible, especialmente en aquellos de alta exigencia gráfica, como los FPS[[1]](#footnote-1).

Pero, estructurar el cuerpo (código fuente) de un juego para que éste pueda ser reutilizado (cuanto más mejor) en otros, requiere de un análisis superior a la simple construcción de aquél[[2]](#footnote-2), pues se ha de imaginar que otros posibles juegos podrían derivarse del que estamos construyendo y programar quizás cosas que no habríamos hecho sólo pensando en solucionar nuestro problema original.

En este punto, podemos distinguir diferentes niveles de abstracción:

El más básico consiste en agrupar bajo funciones determinadas tareas muy habituales en el mundo de los juegos, las cuales, a su vez, se agrupan en **Librerías**. Tareas claras y repetitivas en un juego son las de dibujar en la pantalla, leer el teclado o el joystick, o comunicar cosas a través de la red a otros jugadores. Ejemplos de librerías reconocidas en el mercado para realizar esas funciones serían: SDL para gráficos y manejo de joystick, ratón y teclado, FMOD para la generación de sonido, Racknet para comunicaciones, etc[[3]](#footnote-3).

Un siguiente nivel de abstracción lo encontramos, no sólo en el uso de librerías, sino en la creación de estructuras básica de juego, a modo de plantillas, que hayan de ser completadas para la construcción de un juego final ejecutable, pero que nos ahorren esfuerzo. A esas plantillas, se las conocen como **Frameworks**. Los frameworks más avanzados externalizan incluso muchos de sus parámetros de funcionamiento en ficheros de configuración para así aumentar su flexibilidad.

Si un framework no necesitara refinar su código para crear un juego concreto, sino que bastara sólo con cambiar esos parámetros de funcionamiento definidos en ficheros externos y además esos ficheros pudieran ser manipulados con herramientas habitualmente gráficas asociadas al framework, habríamos alcanzado el concepto de **Engine** de juegos. Ejemplos de ellos son: Unity®, Unreal®, Gamemaker®, etc. Bien es verdad que los motores más potentes permiten incluso añadir código (habitualmente en lenguajes interpretados, como Phyton o Java) para hacerlos incluso más flexibles.

Los lenguajes en los que están desarrollados tanto frameworks como engines son mayoritariamente lenguajes orientados a objeto (C++, C#, Java, Python, …) por la facilidad que estos tienen para encajar en la definición de framework, al ser fácilmente ampliables utilizando los conceptos de clase y herencia.

De entre todos ellos destacan el C++ y el C# (más habitual en el entorno Apple), compilados y no interpretados y, por tanto, mucho más eficientes y rápidos, condición necesaria para la realización óptima de un juego tal y como hemos argumentado anteriormente.

**QGAMES**, es un framework basado en C++ para el desarrollo de juegos 2D, 2.5D y 3D, tanto de mesa, como de acción, como basados en tiles. Se acerca al concepto de engine, pues externaliza muchos de sus parámetros de funcionamiento en ficheros XML que, además, pueden ser modificados con herramientas open source de mercado.

En este documento analizaremos en detalle los principios de funcionamiento de QGAMES. Y aprenderemos también a cómo extenderlo para realizar nuestros propios juegos.

El documento no pretende ser una guía exhaustiva de todas y cada una de las posibilidades que ofrecen todas y cada una de las clases que componen el framework (¡1 millón de líneas de código!). No pretende, por tanto, reemplazar a la ayuda que acompaña al framework, pero sí nos ayudará a navegar luego con facilidad por ella.

Este documento está dividido en 4 grandes bloques. Bloques que, a su vez, se dividen en secciones que van desgranando poco a poco como se construye un juego bajo QGAMES. Esas secciones son:

* **Bloque 0: Instalando y configurando QGAMES**

Donde analizaremos como descargar QGAMES y como configurarlo para empezar a trabajar con él.

* **Bloque 1: La arquitectura básica de QGAMES y el módulo “Common”:**

Quizás el bloque más importante del documento, pues fija los conceptos fundamentales para manejar QGAMES. Entendido este bloque cualquier otro será mucho más sencillo.

Incluye las siguientes secciones:

La arquitectura básica de QGAMES: Identificación de los grandes bloques en los que se agrupa la funcionalidad de QGAMES y de las relaciones que existen entre cada uno de ellos.

Tras esa revisión general, comenzamos a analizar el módulo básico sobre el que se asienta cualquier otro en QGAMES:

Las clases básicas. Foundation: En donde se analizan someramente las clases básicas utilizadas en cualquier juego. Suelen ser clases mayoritariamente ligadas al algebra.

El núcleo básico de un juego. La clase *Game* y *GameImplementation*: En la que se introduce la clase central de cualquier juego, *Game*, su funcionamiento y sus métodos y atributos fundamentales.

Como dibujar en QGAMES. Las clases *Screen*, *Camera* y *Transaction*: En la que se trata la clase *Screen*, entendiendo su forma de trabajar, sus métodos fundamentales, y la manera de integrarla en el esquema general de un juego gobernado por Game.

La representación de la realidad. La clase *Element*: Esta clase es la base para la representación de la realidad simulada que trata de replicar un juego. Cualquier entidad que forme parte de éste (personajes, enemigos, decorado, marcadores, etc.) ha de estar representada por una clase que derive de alguna u otra manera de ésta, por lo que su comprensión es imprescindible. Se trata, por tanto, de una clase fundamental en QGAMES, pero muy compleja que iremos desgranando poco a poco conforme se introducen otras secciones.

La comunicación interna entre los elementos de un juego. El modelo *Notifier* – *Observer*: En un juego es muy habitual que se produzcan situaciones sobre un determinado elemento que afecten o tengan consecuencias en otros. Esos efectos se transmiten de elemento a elemento utilizando este patrón de diseño.

Contadores y Switches: Quizás dos de las estructuras, junto con la que se introduce en la siguiente sección, más utilizadas en QGAMES. Los contadores y los switches se utilizan habitualmente para representar el estado en el que se encuentra cualquier elemento del juego y tienen la característica de ser tratadas de forma estándar por muchas funciones de QGAMES.

Agregadores de datos. Las clases *OpenValue* y *SetOfOpenValues*. La tercera estructura en discordia. Se utiliza habitualmente para almacenar cualquier tipo de información relacionada con el juego.

Otra vez la clase *Element. Añadir contadores y switches y salvar y recuperar su estado*. En dónde vamos a introducir todos los conceptos anteriores.

Dando movimiento a los elementos de un juego. La clase *Movement*: Que nos ayuda a controlar los movimientos de cualquier de las entidades de un juego.

Animando los elementos de un juego. *Forms*, *Frames*, *Animations* and *States*. En esta sección se analizan las herramientas básicas disponibles en *QGAMES* para conseguir animar un elemento cualquiera.

Elementos con animación y movimiento. Las clases *Entity* y *Character*. Volvemos nuevamente a la clase *Element* o, mejor dicho, a clases que derivan de ésta y que incluyen propiedades relacionadas con la animación y el movimiento, siendo, por tanto, las clases adecuadas para la construcción de los personajes que intervienen en cualquier juego.

Las boyas. La clase Buoy. A menudo es necesario marcar situaciones que deben ser procesadas más tarde para no romper el juego.

Un alto en el camino. Para resumir en unos cuantos puntos lo aprendido y quizás generar dudas que hagan que volvamos hacia atrás.

Nuestro primer juego. Pararemos para realizar nuestro primer *Hello World* en QGAMES: Conseguir un ninja corriendo por la pantalla de izquierda a derecha, sobre un fondo estático.

Introducción a los Builders. Construcción de formas, entidades y movimientos desde ficheros de configuración. QGAMES permite que las entidades de un juego se definan en ficheros de configuración, aumentando así su versatilidad y reduciendo la cantidad de código a escribir para un nuevo juego. Su construcción se lleva a cabo empleando Builders.

Los Score Objects, o como construir elementos que reflejen el estado de un juego y que se superpongan a las imágenes en acción. La clase *ScoreObject*.

Gestionado los estados de un juego. La clase *GameState*: Un juego también puede sder entendido como una sucesión de diversos estados. Inicialmente estamos presentado la introducción, luego habilitamos el menú de opciones, luego jugamos, etc.

Nuestro primer juego. Es bueno hacer en este punto un alto en el camino y poner a jugar todos los elementos analizados anteriormente para construir nuestro primer juego, a modo de “Hello World”.

* Bloque 2: Juegos de arcade. El módulo Arcade
* Bloque 3: Juegos de mesa. El módulo Board.
* Bloque 4: Juegos avanzados de arcade. El módulo Advanced Arcade.
* Bloque 5: Juegos de plataformas. El módulo Platform Game.
* Bloque 6: Aproximación a las plantillas de juegos concretos.

Este documento supone unos altos conocimientos en la programación en C++ y los patrones de diseño típicos de arquitectura de software.

También presupone que tiene instalado y configurado Visual Studio 2017 en adelante.

Bloque 0: Instalando QGAMES

# Configuración de las librerías requeridas por QGAMES

Como se ha señalado en la introducción, QGAMES se apoya en diferentes librerías gráficas y de sonido a las que accede apoyándose en el concepto de wrapper. Y aunque QGAMES puede extenderse fácilmente para soportar casi cualquier librería, de fábrica incluye wrappers para soportar las siguientes:

* **SDL** para gráficos, sonidos y manejo de dispositivos como el joystick, el teclado y el ratón.
* **FMOD**, para sonidos.
* **RakNet**. Para comunicaciones en juegos en red.
* **OGRE**, para manejo de escenarios 3D.

Librerías que, como no son parte de QGAMES, hay que descargarse de sus respectivos sitios web e instalarlos.

SDL, a su vez, está formada por una librería core y muchas otras librerías satélite (desarrolladas por terceros) especializadas en otras tareas más concretas como la lectura y manejo de ficheros de imágenes, el manejo de tipos de letra o el manejo del sonido.

Cada una de ellas se encuentran en las siguientes direcciones web:

* **SDL core**: <https://www.libsdl.org/download-2.0.php>
* **SDL Image**, para el manejo de ficheros gráficos (.png, .gif, .jpg, etc.): <http://www.libsdl.org/projects/SDL_image/>
* **SDL GFX**, para el dibujo de figuras geométricas: <http://www.ferzkopp.net/wordpress/2016/01/02/sdl_gfx-sdl2_gfx/>
* **SDL Mixer**, para el manejo del sonido: <http://www.libsdl.org/projects/SDL_mixer/>
* **SDL TTF**, para el manejo de fonts en la pantalla: <https://www.libsdl.org/projects/SDL_ttf/>

Descargue cada una de ellas, descomprímalas y colóquelas en un directorio cada uno, colgando de un directorio común, si es posible, para facilitar su localización conjunta futura. Descargue siempre las versiones ya compiladas en aquellos casos en los que exista también la posibilidad de descargarse el código fuente y el proyecto de compilación.

Se recomienda crear un directorio llamado **Workspace**, y dentro de él, otro llamado **CPPLIBS** en el que, a su vez, ir creando directorio donde ubicar cada una de las librerías anteriores, como, por ejemplo: **SDL**, **SDL\_IMAGE**, **SDL\_GFX**, **SDL\_MIXER** y **SDL\_TTF**.

QGAMES 1.0 está compilado con la versión 2.0.20 de SDL Core, la 2.0.5 de IMAGE, la 1.0.4 de GFX, la 2.0.4 de Mixer y la 2.0.18 de TTF.

Vamos a hacer lo mismo con la librería de sonidos.

FMOD se puede descargar de la página: <https://www.fmod.com/>. Instale FMOD en un subdirectorio llamado FMOD debajo de CPPLIBS. QGAMES está compilado con la versión 2.2.6 de FMOD.

Haga lo propio con RakNet: <http://www.jenkinssoftware.com/download.html>. Instálelo en un subdirectorio llamado RAKNET también debajo de CPPLIBS. QGAMES está compilado con la versión 4.081 de RAKNET.

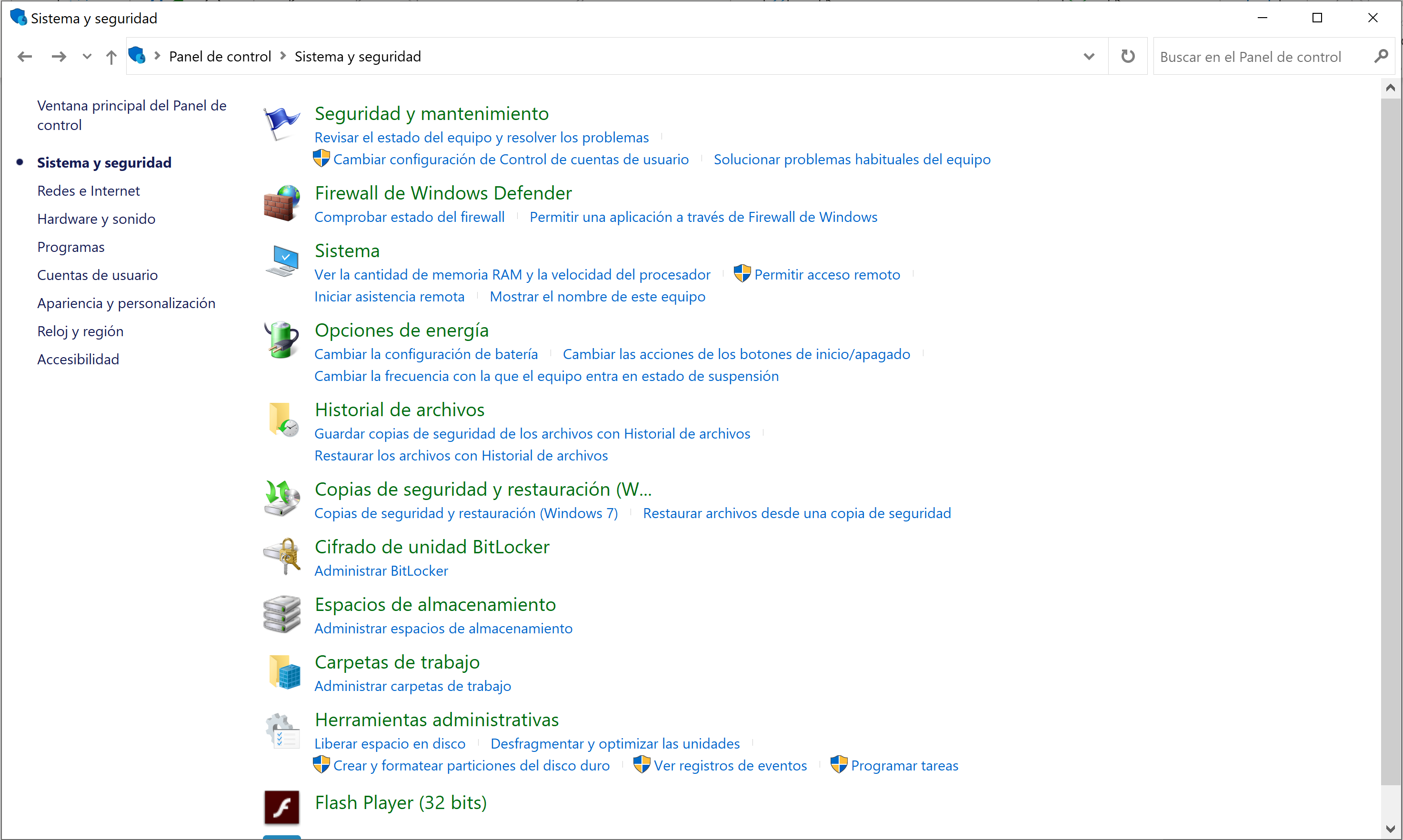
Al descargar RakNet nos descargamos el código fuente el proyecto Visual Studio 2008 para compilarlo. Como llevar a cabo este proceso no se incluye en este documento. Consulte los propios manuales de Raknet.

Por último, hay que descargar OGRE3D, desde: <https://www.ogre3d.org/>. QGAMES está compilado usando la versión 1.11.5 de Ogre. De Ogre puede descargarse tanto el código Fuente y el Proyecto de compilación, como el SDK ya compilado para Windows®. Escoja esta segunda opción.

# Configuración del entorno

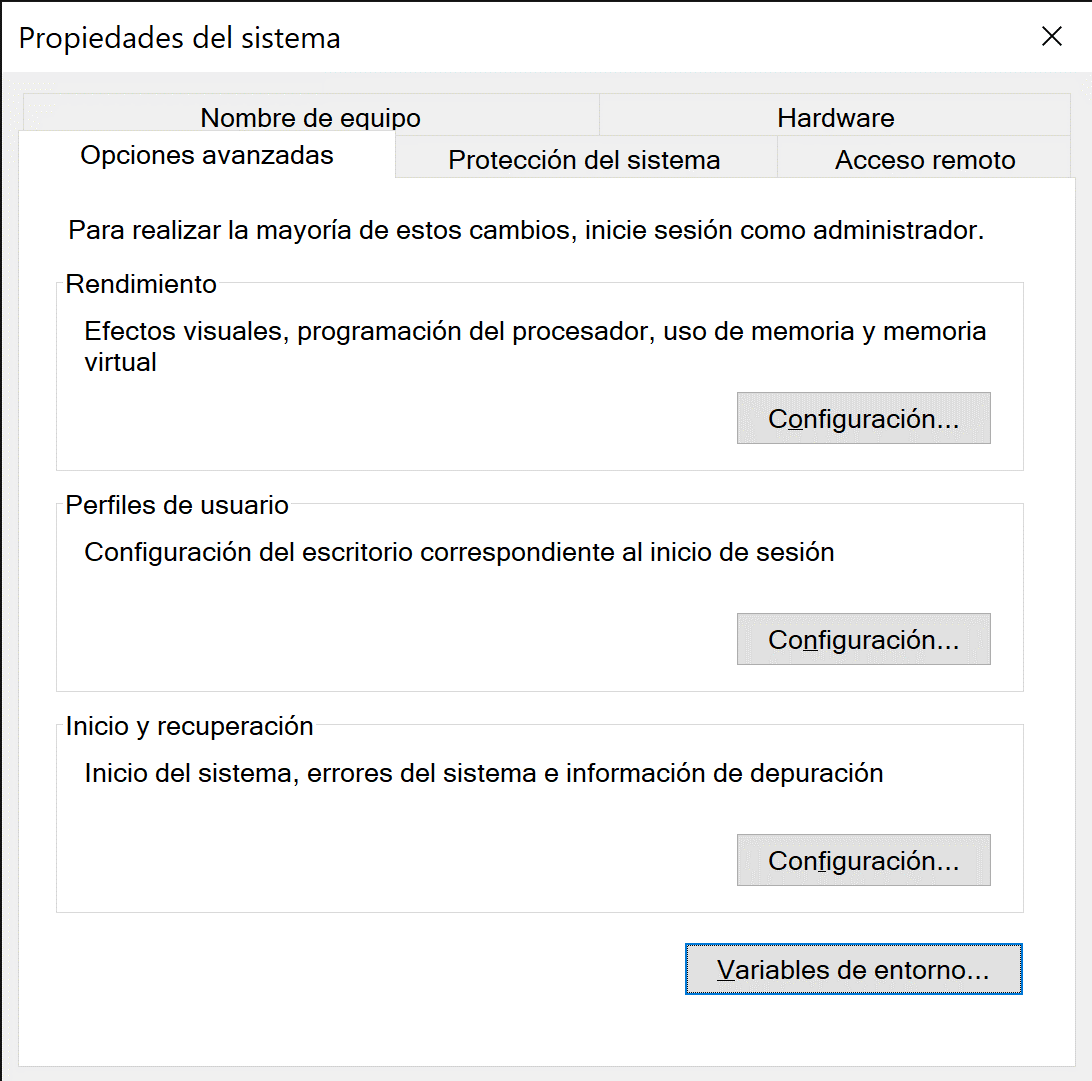
Una vez descargadas todas las librerías en las que se apoya QGAMES, es necesario configurar el entorno para que cuando compilemos un nuevo juego QGAMES con Visual Studio, éste sepa encontrarlo.

Abra para ello el panel de control:



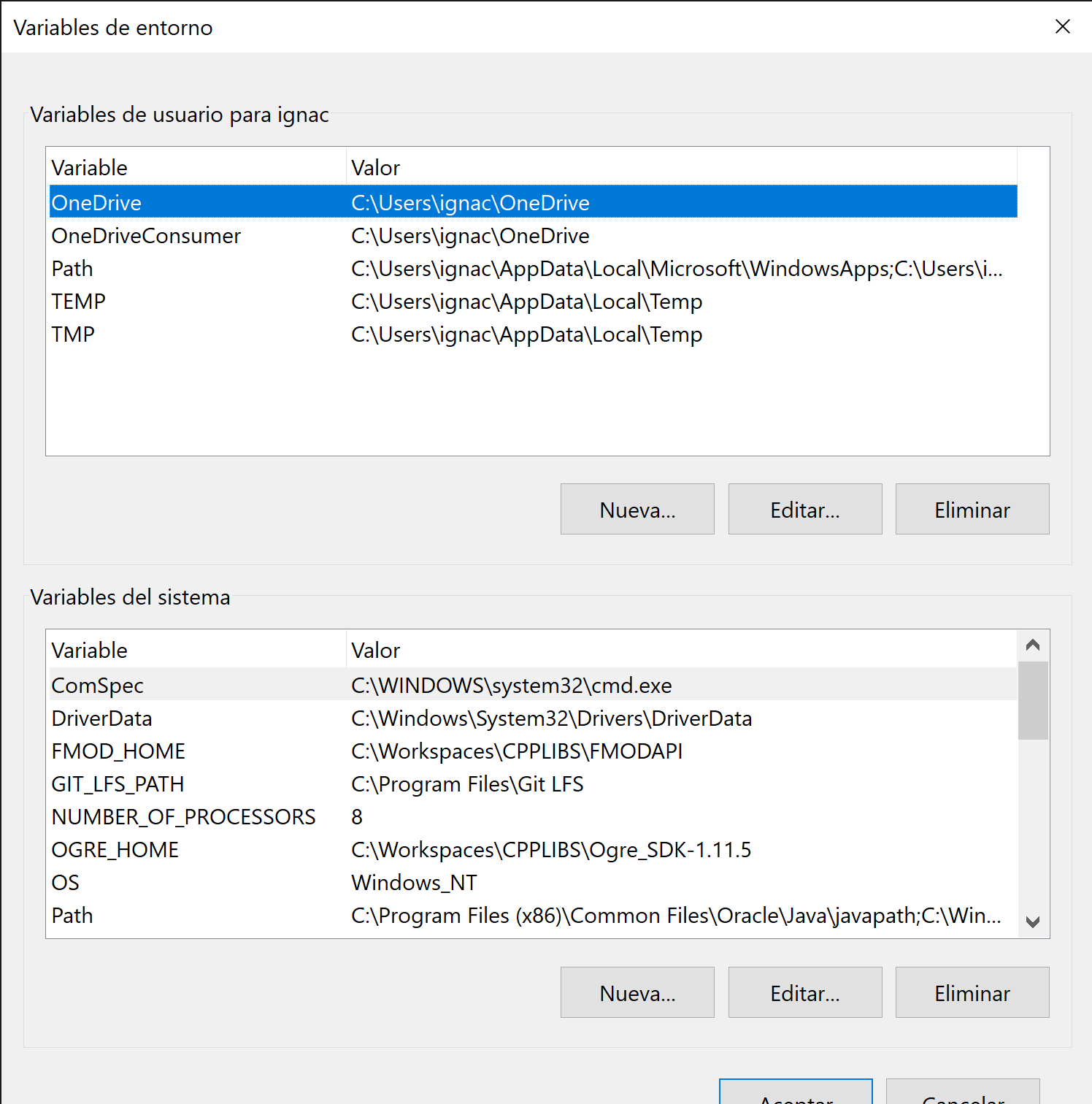
2: Panel de control

Entre en Sistema y seleccione Configuración Avanzada (en la parte izquierda). Se abrirá una ventana con las propiedades avanzadas:



3: Propiedades del Sistema. Opciones avanzadas.

Seleccione entonces Variables de Entorno (en la parte de abajo). Aparecerá una ventana como la siguiente:



4: Variables de entorno

En la parte de debajo añada (Nueva) las siguientes variables de entorno:

*SDL\_HOME*, *SDLIMAGE\_HOME*, *SDLGFX\_HOME*, *SDLMIXER\_HOME* y *SDLTTF\_HOME, FMOD\_HOME, OGRE\_HOME y RAKNET\_HOME,* asignándoles como valor el directorio en el que ha ubicado cada una de las librerías respectivas que antes hemos descargado e instalado, concretamente a partir del cual se pueden acceder a los .h (/include) y a las diferentes librerías dependiendo de si son Debug o Release.

Estas variables de entorno son usadas por los proyectos QGAMES y no están en vigor hasta que el sistema no se re-arranque. Dado que aún tenemos que instalar unas cuantas librerías más, esta acción la dejaremos para dentro de un rato.

# Configuración de QGAMES

Vamos a instalar ahora QGAMES:

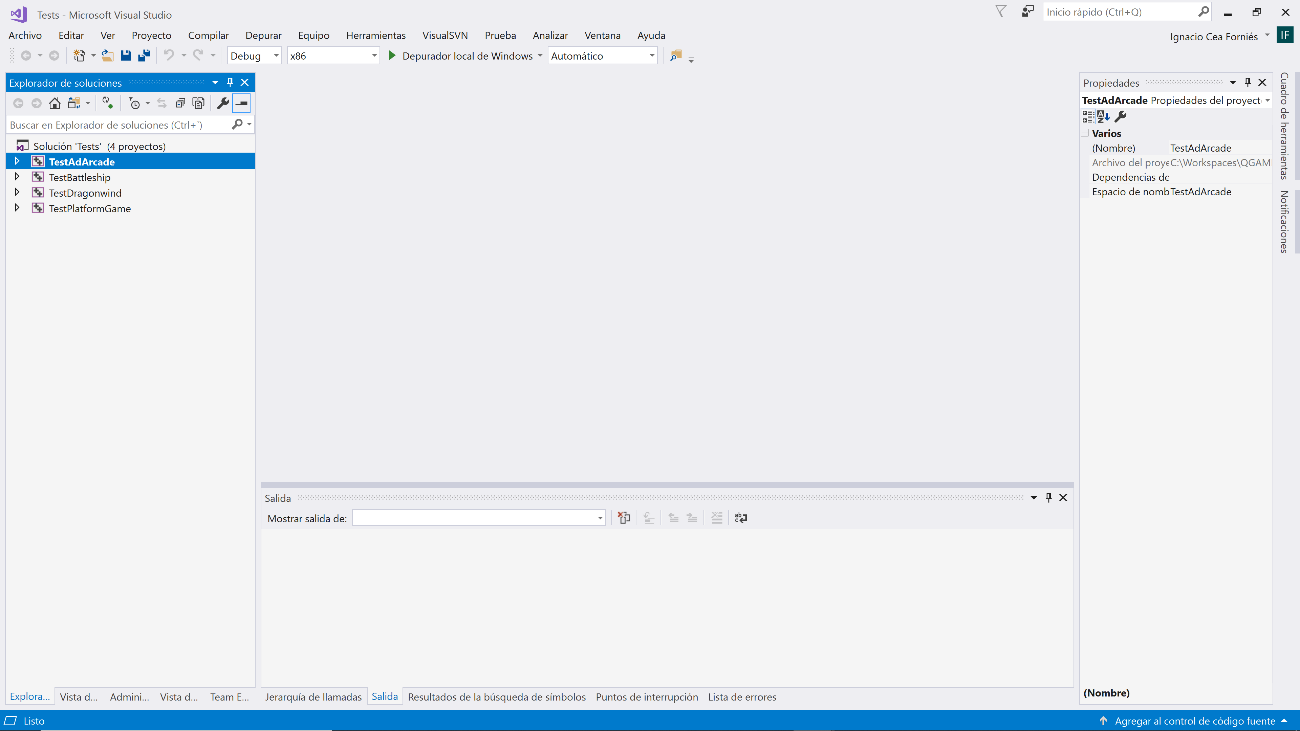
Vaya a: <https://sourceforge.net/projects/qgames-library/>, descargue los siguientes ficheros:

* **html.zip**, que contiene la documentación del QGAMES.
* **QGAMESLib.zip**, que contiene los ficheros básicos de la librería,
* **QGAMESCommonLibs**, que contiene las dll básicas para que funcione cualquier juego.
* **QGAMESProjects.zip**, que contiene 3 proyectos necesarios para la creación de cualquier juego.
* **QGAMESamplesXX.zip**, que contiene ejemplo de juegos ya creados con QGAMES.
* Test.sln y README.rtf para compilar los juegos con Visual Studio y algunas recomendaciones generales sobre el uso de QGAMES.
* **README.rtf**, que le cuenta más o menos lo mismo que está leyendo,
* **Tests.sln**, un proyecto Visual Studio 2018 que le permite acceder a los ejemplos y empezar a jugar con ellos.

Descomprímalos en un directorio, dentro de Workspaces llamado, por ejemplo, QGAMES. Los ejemplos descomprímales en un subdirectorio dentro del anterior llamado simples.

Bien, ya estamos listos para empezar a utilizar QGAMES.

Abra el fichero Test.sln que acaba de instalar en un subdirectorio llamado QGAMES. Se abrirá la siguiente pantalla:

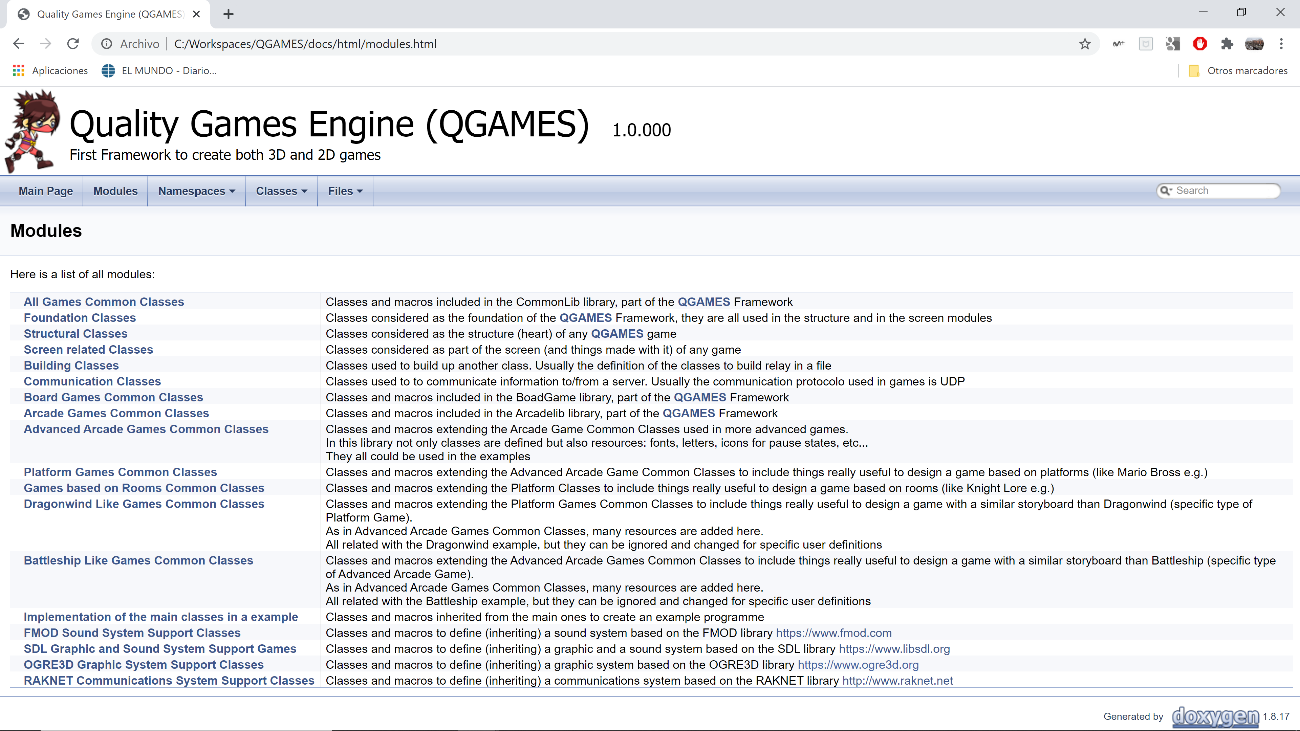


5: Por fin. Ya podemos empezar a trabajar

Para comprobar que todo ha sido instalado correctamente, pruebe a abrir cualquier fichero de código y a compilar la solución.

No pierda de vista la ayuda. Descargue de <https://sourceforge.net/projects/qgames-library/> el fichero **html.zip**. Cree un subdirectorio llamado **doc** dentro de QGAMES (que, a su vez, estaba ubicado dentro de Workspaces) y descomprima html.zip allí.Hay mucha documentación, por lo que tardará un rato. La utilización de la ayuda será fundamental para profundizar en el funcionamiento de QGAMES.

Abra el fichero llamado index.hml. La documentación ha sido creada con doxygen®:



6: La ayuda

# Herramientas asociadas

Como se ha indicado también anteriormente, QGAMES se acerca bastante al concepto de Engine y muchos de sus ficheros de configuración puede manipularse con herramientas externas. Las más importantes, todas OpenSource, son:

* **Tiled Map Editor**®, para la creación de tilemaps (muy típicos en juegos tipo Mario Bross®): <https://www.mapeditor.org/>
* **Sprite Font Builder**®, para la creación de tipografías super útiles en cualquier juego: <https://www.johnwordsworth.com/projects/sprite-font-builder/>
* **Texture Packer**®, para la creación de hojas de gráficos, imprescindibles en juegos de animación especialmente 2D y 2.5D: <https://www.codeandweb.com/texturepacker>.
* **GIMP**®, para dibujar. <http://www.gimp.org.es/>
* **Notepad++**®, para manipular en general los ficheros XML, si bien esto también puede hacerse con el propio Visual Studio: <https://notepad-plus-plus.org/>

Bloque 1: La arquitectura básica de QGAMES y el módulo “Common”

# La arquitectura básica de QGAMES

QGAMES agrupa su funcionalidad en distinto módulos o bloques. Bloques que se apoyan unos en otros de tal manera que el empleo de uno de ellos requiere de todos los que están por debajo de él.

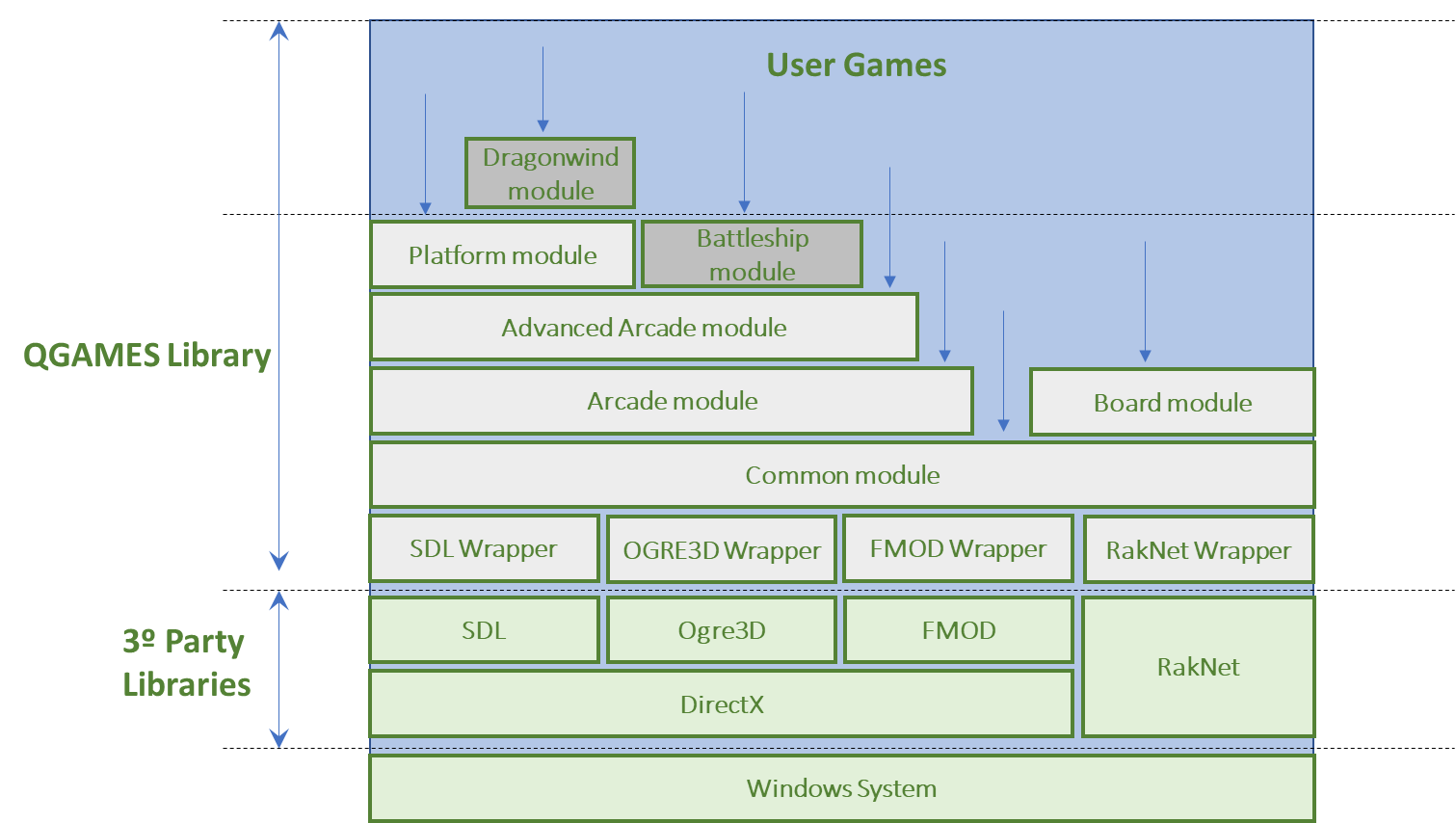
QGAMES no es una librería para dibujar formas 3D o para producir los sonidos que todo juego usa, ni para interactuar con el joystick o el ratón utilizados por los jugadores para controlar a los personajes. QGAMES se apoya en librerías estándar multiplataforma para llevar a cabo todas esas tareas, focalizándose “sólo” en organizar y dar estructura a la lógica global de funcionamiento inherente a todo juego, de tal manera que el programador pueda centrarse sólo en programar la trama del juego.

QGAMES permite la creación de multitud de tipos de juegos de manera sencilla y rápida. Con QGAMES se pueden construir juegos de arcade (vamos, ¡los de marcianos!), de plataformas (como el Mario Bross®), de rol (como Clash of Clans®), o de mesa (como el Trivial®) o mezclas de todos ellos. Esos juegos pueden ser en 3D, en 2D o en 2.5D (2D simulando 3D mediante proyecciones en isométrica o caballera). Y todos ellos admiten además la posibilidad de jugar en red gracias a las capacidades de comunicación que también proporciona QGAMES.

QGAMES está programado en C++ 11 estándar y compilado con Visual Studio 2017.

Como se introducía anteriormente, QGAMES se organiza en diversos módulos, cada uno de los cuales provee siempre de funcionalidad adicional a aquel sobre el que se apoya. El módulo base de todos ellos es el conocido como **Common**. Dentro de Common se encuentran integrados **wrappers[[4]](#footnote-4)** de diferentes librerías estándar que finalmente son los responsables de interactuar con el Sistema Operativo de la máquina y éste con el hardware de aquella.

Todo ello se muestra en la siguiente figura:



7: Arquitectura general de QGAMES

Cada uno de los módulos se ocupa, de forma breve, de las siguientes responsabilidades:

## Wrappers

* **SDL Wrapper**: SDL es el acrónimo de Simple Direct Media Library. SDL es una libraría estándar de mercado y multiplataforma que se encarga de proveer funciones para simplificar el acceso a los sistemas de audio, teclado, joystick y gráficos de un computador (o smartphone).

SDL Wrapper define una capa de abstracción sobre SDL para facilitar su integración en la estructura de funcionamiento definida en el módulo Common. Existen otras muchas librerías similares a SDL®, como SFML® o Allegro®. QGAMES está diseñada para facilitar la creación de wrappers sobre ellas que puedan integrarse fácilmente en “Common”.

* **OGRE3D Wrapper**: OGRE3D es una librería que se ocupa de dibujar en 3D. No se ocupa de simplificar la interacción con el Joystick o de producir sonidos. Este wrapper tiene un objetivo semejante al SDL wrapper visto anteriormente. Y también de manera semejante, QGAMES proporciona mecanismos suficientes para crear wrappers sobre otros motores 3D, como Unity® o Unreal®. Ogre3D es nativo C++ y posiblemente el motor de dibujo 3D más potente bajo este lenguaje.
* **FMOD Wrapper**: La especialidad de FMOD es el sonido.
* **Raknet Wrapper**: Es un wrapper sobre la librería de comunicaciones estándar Raknet® muy utilizada en el mundo de los videojuegos.

## Módulos básicos

Módulos elementales para la realización de cualquier juego:

* **Common module**: Define la estructura elemental de cualquier juego.
* **Board module**: Añade a Common las características inherentes a cualquier juego de mesa (de tablero).
* **Arcade module**: Añade a Common las propiedades de cualquier juego de acción.

## Módulos avanzados

Módulos para realizar juegos complejos:

* **Advanced Arcade Module**: Añade al módulo Arcade características adicionales que permite la realización de juegos más complejos.
* **Platform module**: Añade a Advanced Arcade las funciones necesarias para desarrollar juegos basados en plataformas.
* **RoomGame module**: Añade capacidades para simplificar el diseño de juegos basados en habitaciones.

## Módulos plantilla de juegos

Estos módulos ya son algo más que librerías. Incorporan la lógica completa y funcional (e incluso gráficos y sonidos) de un juego completo. Sirven para primero aprender a desarrollar un juego y, segundo, pueden ser fácilmente modificados y ampliados para desarrollar juegos similares a ellos. Por tanto, ¡con estas plantillas es fácil desarrollar la típica 2ª y 3ª parte de un juego!:

* **Battleship like games module**: Battleship es un ejemplo de juego arcade construido sobre el módulo Advanced Arcade.

El protagonista es una nave espacial que ha de eliminar todos los asteroides, naves espaciales enemigas, etc. que se le acercan, para llegar a alcanzar su objetivo.

* **Dragonwind like game module**: Dragonwind es un ejemplo de juego de plataformas (tipo Mario Bross).

El protagonista en un/a ninja que ha de ir pasando por diversos niveles de plataformas, recogiendo cosas, en busca de las cenizas secuestradas de su padre.

* **PacMan like game module**: ¡Adivina para que sirve!. Con él es posible el desarrollo de cualquier juego tipo PacMan (Ms. PacMan, propios,…).

Cualquier nuevo juego puede construirse usando cualquiera de las librerías, pero sabiendo que la utilización, por ejemplo, del módulo Platform arrastrará a todos los otros sobre los que éste se ha construido; es decir, en este caso: Advanced Arcade, Arcade, Common, y los diferentes wrappers.

Bien, vamos a analizar más en detalle un juego QGAMES.

# Las clases básicas de un juego. Foundation

Quedan recogidas en la siguiente tabla:

# El núcleo básico de un juego: Las clases *Game* y *GameImplementation*

## La clase *Game*

La puedes encontrar en el fichero *game.hpp*.

El núcleo básico de un juego lo constituye la clase *Game* o una derivada de ésta.

El diseño de esta clase sigue dos patrones básicos: Singleton y Facade[[5]](#footnote-5). Es decir, en un juego existe una única instancia (Singleton) de la clase *Game* que, además, concentra muchos métodos que permiten acceder de forma más simple a otros elementos de la estructura de un juego (Facade).

Abarcar de un primer vistazo todos los métodos que conforman la clase *Game* ahora, es imposible y además innecesario, no sólo por su cantidad, sino por el momento de comprensión de la librería en el que nos encontramos. Es mejor irla desgranando poco a poco. Y así lo haremos en el documento.

El método más importante de esta clase es ***exec***. Es el método que permite la ejecución del juego. Cualquier juego, del más sencillo al más complicado puede resumirse en casi dos líneas de código, como demuestra un típico ***main*** de cualquiera de los ejemplos incluidos en la librería:

/\*\*

\* @file

\* File: main.cpp \n

\* Framework: Commty Game Library (CGL) \n

\* Author: Ignacio Cea Forniés (Community Networks) \n

\* Creation Date: 15/01/2018 \n

\* Description: Main program for SDL Test

\* Versions: 1.0 Initial

\*/

#include "stdafx.h"

#include "TestSDL.hpp"

using namespace TestSDL;

#ifndef \_CONSOLE

#include <SDL.h>

#ifdef \_\_cplusplus

#define C\_LINKAGE "C"

#else

#define C\_LINKAGE

#endif /\* \_\_cplusplus \*/

#if \_MSC\_VER >= 1900

extern C\_LINKAGE FILE \_\_iob\_func[3] = { \*stdin,\*stdout,\*stderr };

#endif

extern C\_LINKAGE int main (int argc, char \*argv [])

#else

int \_tmain (int argc, char \*argv [])

#endif /\* \_CONSOLE \*/

{

#ifdef NDEBUG

// In the release version, the resources are in a zip file

// This instruction reads them and store in temporaly files when needed!

// The resource reader is deleted at the end, and all the temporal file with them!

QGAMES::InZipResourceReader rR

(std::string (".") + \_\_PATH\_SEPARATOR\_\_ + std::string ("temp"));

#endif

Game game;

game.setLinkDrawToFrameRate (true);

game.setPreLoad (true);

game.exec ();

return (0);

}

8: Un ejemplo de juego

Es decir, para crear un juego basta instanciar un objeto de la clase *Game* (o derivadas) e invocar al método *exec*. *exec* es el corazón de cualquier juego y desde él se termina invocando cualquier otro de los que hablemos a lo largo de este documento. Es, por tanto, el método que vertebra un juego. Y, por tanto, hay que entenderlo bien.

Analicemos un poquito más este método. Su código fuente es el siguiente:

// ---

void QGAMES::Game::exec ()

{

// The log is activated when the execution starts...

// Just if the mode is not debugging...

#ifdef NDEBUG

setLog (false);

#else

setLog (true);

#endif

\_\_WLOGPLUS ("Starting execution...");

// Initialize for random numbers...

srand ((int) time (NULL));

int loopCounter = 0;

\_\_WLOGPLUS ("Initilizing...");

initialize ();

if (!isInitialized ())

return;

\_\_WLOGPLUS ("Initilized...");

\_running = true;

while (\_running)

{

handleEvents ();

inEveryLoop ();

if (QGAMES::GUISystem::system () != NULL) // If it exists...

QGAMES::GUISystem::system () -> inEveryLoop ();

// Is the main loop linked to the frame rate or not?

// if it is, then maybe this loop is not the right one to do anything...

bool mustDoLoop = true;

if (\_linkDrawToFrameRate)

{

if (++loopCounter >= (timer () -> loopsPerSecond () / \_framesPerSecond))

loopCounter = 0;

else

mustDoLoop = false;

}

// If finally the loop has to be done...do it!

if (mustDoLoop)

{

// Actualize the positions...

if (!\_paused)

{

if (\_currentGameState != NULL)

\_currentGameState -> updatePositions ();

if (mustIUpdate ()) // Only if needed!

updatePositions ();

}

// Actualize the image in the screens...

for (QGAMES::Screens::const\_iterator i = \_screens.begin ();

i != \_screens.end (); i++)

{

QGAMES::Screen\* scr = (\*i).second;

if (mustIUpdate ())

scr -> updatePositions (); // If there is something to move...

scr -> openTransaction ();

if (drawGameBeforeState ())

drawOn (scr); // The screen itself (world, entities of the world,...)

if (\_currentGameState != NULL)

\_currentGameState -> drawOn (scr); // The state itself...

if (QGAMES::GUISystem::system () != NULL) // And if it exists the gui system...

QGAMES::GUISystem::system () -> drawOn (scr);

if (!drawGameBeforeState ())

drawOn (scr); // The screen itself...

scr -> drawOnYourOwn (); // Sometimes there is something to draw on its own!...

scr -> closeTransaction ();

}

// Actualize the counters, and do additional tasks...

if (!\_paused)

{

for (QGAMES::LoopCounters::const\_iterator j = \_loopCounters.begin ();

j != \_loopCounters.end (); j++)

(\*j) -> count ();

additionalTasks ();

}

}

}

\_running = false;

finalize ();

\_\_WLOGPLUS ("...Finishing execution");

}

9: El método exec de la clase Game

No pretendemos ahora que comprenda todas las líneas, pero en ellas distinguimos tres grandes partes: Un bucle principal definido por *while (\_running)* y sendos bloques de código antes y después del mismo.

Dentro de ese bucle, se realizan cuatro acciones principales:

1. **Leer los eventos que llegan de dispositivos** como el teclado, el ratón, el joystick y, en general cualquier mecanismo de interacción con el jugador. Esta acción se lleva a cabo en el método *handleEvents ()*.
2. **Alinear el tiempo que se emplea en la realización de cada bucle** con la ratio de bucles por segundo que se haya definido para el juego. Ésta es quizás una de las acciones más técnicas de cualquier juego, pero imprescindible para mantener su fluidez.

Es evidente que en un juego las acciones a realizar en cada bucle variarán. En un bucle, por ejemplo, tendremos que mover un personaje y en otro quizás dos o tres más. Por tanto, el tiempo, a priori, que el ordenador va a emplear en ejecutar cada bucle también variará. Si no hacemos nada por intentar mantenerlo constante podríamos notar una falta de fluidez (sobre todo grafica) en el juego y no podríamos quizás, por ejemplo, ligar determinadas acciones a tiempos o momentos concretos de la ejecución de una acción (e.j. que algo explote a los tres segundos de juego).

Las líneas que, dentro del bucle principal, se ocupan de esa tarea son:

bool mustDoLoop = true;

if (\_linkDrawToFrameRate)

{

if (++loopCounter >= (timer () -> loopsPerSecond () / \_framesPerSecond))

loopCounter = 0;

else

mustDoLoop = false;

}

// If finally the loop has to be done...do it!

if (mustDoLoop)

{

…

}

10: Volante de inercia de un juego. Detalle del método exec de la clase Game

Actúan a modo de volante de inercia en ingeniería para mantener constante, en la medida de lo posible, el tiempo de ejecución del código que haya dentro del bloque *if(\_mustDoLoop)*. No importa el detalle, pero *timer ()* devuelve un objeto que es capaz de medir cuantos bucles se están ejecutando en cada unidad de tiempo y \_*framesPerSecond* es una variable que se inicializa en el momento de creación del objeto *Game*.

1. La tercera acción es la **actualizar el estado del juego**. Algo que se llega a cabo al ejecutar el método ***updatePositions*** (y similares).
2. Y, por último, **Dibujar el resultado**. Algo que se realiza en los diferentes métodos que incluyen la palabra ***draw***.

Dentro del bucle se realiza una quinta acción, ***inEveryLoop ()***. Esta acción no está vinculada a la velocidad del juego. En ella, por tanto, deberían realizarse tareas muy generales sin implicación en la dinámica del juego. Una de las tareas que precisamente se realiza, es la de calcular cuantos bucles por segundo se están ejecutando en cada instante.

Fuera del bucle principal podemos destacar que justo antes de él, se invoca al método interno ***initialize ()*** que se encarga de realizar aquellas acciones que son necesarias para la inicialización de un juego. De igual manera tras el bucle, que se ejecuta mientras la variable *\_running* sea true, se invoca al método ***finalize ()*** que hace o debe hacer, justo lo contrario que el primero.

Una tarea típica del método *initialize ()* es, por ejemplo, cargar en memoria los recursos más utilizados en el juego para acelerar su ejecución posterior. De manera contraria, *finalize ()*, debería liberarlos. Volveremos a estos métodos más adelante.

## La clase *GameImplementation*

Como se ha señalado en la introducción de este documento, QGAMES no se ocupa de realizar las tareas más pegadas al hardware, como la de dibujar o la manejar el joystick. Para ello se apoya en wrappers sobre librerías de mercado especializadas y reconocidas en esas tareas.

Si el acceso a esos wrappers se hiciera directamente a través de métodos definidos en la propia clase *Game*, la introducción de un nuevo wrapper sobre una nueva librería, diferente por ejemplo de la que por defecto considerara QGAMES, implicaría tener que crear una nueva clase derivada de *Games* que instanciara el nuevo wrapper, complicando enormemente la evolución y uso de la librería en sí misma.

QGAMES hace uso del patrón **Bridge** para evitar este problema. *Game* se apoya en *GameImplementation*, y está definida en el fichero *gameimplementation.hpp*:

/\*\* \ingroup Game \*/

/\*\* \ingroup Structure \*/

/\*@{\*/

/\*\*

\* @file

\* File: gameimplementation.hpp \n

\* Framework: Commty Game Library (CGL) \n

\* Author: Ignacio Cea Forniés (Community Networks) \n

\* Creation Date: 01/12/2014 \n

\* Description: Defines an specific implementation for a game (SDL?). \n

\* Versions: 1.0 Initial

\*/

#ifndef \_\_QGAMES\_GAMEIMPLEMENTATION\_\_

#define \_\_QGAMES\_GAMEIMPLEMENTATION\_\_

#include <Common/event.hpp>

#include <Common/inputhandler.hpp>

namespace QGAMES

{

class Game;

class FormBuilder;

class SoundBuilder;

class Screen;

class \_ExEngine;

/\*\*

\* The class GameImplementation represents the way many different thing

\* are really implemented, and in many ocassions they will depend on the libraries

\* used behind this framework. e.g. It is not the same to create an input handler

\* under SDL library than in OGRE. \n

\* Those different thigns are usually the way forms are loaded and represented,

\* the way sounds are implementd, and the way graphics (through the screen object) are managed too. \

\* So these are the methods to define for a specific implementation.

\*/

class GameImplementation

{

public:

GameImplementation ()

: \_game (NULL)

{ }

virtual ~GameImplementation ()

{ }

/\*\* This method is used when the game class is built. \n

The constructor receives always a reference to the basic implementation,

and then the reference to the game class owning that implementation is set. \*/

void setGame (Game\* g)

{ \_game = g; }

Game\* game ()

{ return (\_game); }

/\*\* To create the external engine if the implementation needs it.

By default it is not needed. \*/

virtual \_ExEngine\* createExternalEngine ()

{ return (NULL); }

// These all methods are invoked from the exec method of the game class...

/\*\* Sometimes the implementation libraries have to be initialized. \n

By default nothing is done, and then true is always returned. \*/

virtual bool initialize ()

{ return (true); }

/\*\* How to create the form builder and the sound builder is something usually really linked

to the libraries finally used behind. So these methods are invoked from the game main loop at its initialization,

and they have to be redefined per library. \*/

virtual FormBuilder\* createFormBuilder () = 0;

virtual SoundBuilder\* createSoundBuilder () = 0;

/\*\* A similar situation happens with the input handler and any screen.

Those methods are also invoked from the game mainloop just once the builders

have been created. The input handler creation process needs a reference to the behaviour. \*/

virtual InputHandler\* createInputHandler (InputHandler::Behaviour\* bhv) = 0;

virtual Screen\* createScreen (const std::string& t, const Position& pos, int w, int h, int pX, int pY) = 0;

virtual void initializeScreens ()

{ } // Some times, screens created have to be initialized...

// This method is called after creating screens in the game main loop!

virtual void finalizeScreens ()

{ } // Some times, screens created (or around) have to be finalized

// This method is invoked when the main loop finishes...

/\*\*

\* Once the main loop starts to run,

\* how to manage getting events from the system, handle and treat them can be different. \n

\* A basic implementation is added, but it can be changed. \n

\* The basic implementation consists on getting the events from the input handler

\* decalared at game class level, invoke the treatment methods defined there also,

\* and delete the event once it has been treated (@see SDL implementation to see how it changes).

\*/

virtual void inEveryLoop ()

{ }

virtual void handleEvents ();

virtual void treatEvent (const Event& evnt);

/\*\* When the main loop is about to finish this method will be invoked.

Some ocassions the libraries used have to be finalized.

By default nothing is done. \*/

virtual void finalize ()

{ }

protected:

Game\* \_game;

};

}

#endif

// End of the file

/\*@}\*/

11: La clase GameImplementation

Tampoco importa mucho en este momento el detalle de la clase, pero en ella se pueden apreciar los siguientes métodos:

* ***createFormBuilder ()***. Este método debe retornar un builder (ya analizaremos más adelante qué es y cómo se usan) que, a su vez, se encarga de la construcción de imágenes representadas por la clase wrapper ***Form*** (sobre objetos gráficos).
* ***createSoundBuilder ()***. Devuelve otro builder que, en este caso, se encarga de la construcción de sonidos representados por la clase wrapper ***Sound*** (sobre sonidos).
* ***createInputHandler ()***. Responsable de devolvernos un objeto de la clase wrapper ***InputHandler*** que agrupa toda la funcionalidad de interacción con los dispositivos de entrada.
* ***createScreen ()***, utilizada para crear una ventana representada por la clase wrapper **Screen** y sobre la que se dibujarán los elementos del juego.

Todos esos métodos están definidos, como se aprecia en la ilustración, como virtuales puros, lo que implica que necesariamente debe crearse una clase hija de ésta que los defina e implemente.

Es esa clase hija la que debe terminar relacionándose con las librerías estándar de mercado.

Por ejemplo, QGAMES define una versión de esta clase para manejar los recursos usando SDL (se mezcla un poco con el uso de FMOD):

/\*\* \ingroup SDL \*/

/\*\* \ingroup Structure \*/

/\*@{\*/

/\*\*

\* @file

\* File: sdlgame.hpp \n

\* Framework: Commty Game Library (CGL) \n

\* Author: Ignacio Cea Forniés (Community Networks) \n

\* Creation Date: 01/12/2014 \n

\* Description: Defines a sdl implememtation of a game implemenmtation. \n

\* Versions: 1.0 Initial

\*/

#ifndef \_\_QGAMES\_SDLGAME\_\_

#define \_\_QGAMES\_SDLGAME\_\_

#include <Common/gameimplementation.hpp>

#include <Common/formbuilder.hpp>

#include <Common/soundbuilder.hpp>

#include <Common/inputhandler.hpp>

#include <SDL.h>

/\*\* SDLGame makes three main things in a different way. \n

Initialize the SDL system, needed to manage SDL library. \n

Read and manage the events in a different way than the standard library does.

and close the SDL system (destroying basic elements created at the beginning) when the game finishes.

SDL uses by default it's own sound system, but it can initiated to run with FMOD. \*/

class SDLGame : public QGAMES::GameImplementation

{

public:

SDLGame (bool fmod = false)

: QGAMES::GameImplementation (),

\_usingFMOD (fmod)

{ }

/\*\* To know whether the implementation is or not using FMOD as sound system. \*/

bool usingFMOD () const

{ return (\_usingFMOD); }

/\*\* @see parent. \*/

virtual bool initialize ();

virtual QGAMES::FormBuilder\* createFormBuilder ();

virtual QGAMES::SoundBuilder\* createSoundBuilder ();

virtual QGAMES::InputHandler\* createInputHandler (QGAMES::InputHandler::Behaviour\* bhv);

virtual QGAMES::Screen\* createScreen (const std::string& t, const QGAMES::Position& pos,

int w, int h, int pX, int pY);

virtual void handleEvents ();

virtual void treatEvent (const QGAMES::Event& evnt);

virtual void finalize ();

virtual void treatEvent (SDL\_Event& evnt);

private: // NOT protected

bool \_usingFMOD;

};

#endif

// End of the file

/\*@}\*/

12: La clase SDLGame que implementa GameImplementation

En donde ya se define una implementación específica de los métodos virtuales anteriores.

La clase *GameImplementation* se conecta con *Game* a través del constructor de esta última que, de manera estándar (en game.hpp), está definido como:

Game (GameImplementation\* imp, int fps = 60);

Que recibe una instancia no nula de *GameImplementation* y el número de frames por segundo máximo del juego[[6]](#footnote-6).

Dentro de la clase *Game* podemos encontrar los siguientes métodos públicos que, de alguna manera u otra que ahora no viene aún al caso, terminan invocando a sus equivalentes de *GameImplementation*:

* ***formBuilder ()***: Devuelve una instancia al un builder responsable de construir los recursos gráficos (*Form*). La primera vez que se invoque a este método dentro de la lógica del juego se invocará a su vez a *createFormBuilder* dentro del objeto *GameImplementation* (visto anteriormente) asociado, ejecutando así el patrón Bridge.
* ***soundBuilder ()***: Ídem con el builder usado para construir recursos de sonidos.
* ***inputHandler ()***: Con este método obtenemos una referencia al wrapper que accede a los dispositivos que interaccionan con el jugador.
* ***mainScreen ()*** y ***screen ()***: Para obtener ventanas de visualización[[7]](#footnote-7).

Vamos a profundizar un poco en como dibujar, dado que, quizás, es la función más vital de un juego.

# Como dibujar en QGAMES. Las clases *Screen*, *Camera* y *Transaction*

Para comprender como dibujar, es necesario entender, aunque sea someramente, el funcionamiento de cualquier tarjeta gráfica moderna.

Las tarjetas gráficas actuales trabajan con dos buffers internos de memoria. Uno de ellos es temporal y el otro representa la realidad de lo que hay en la pantalla.

En los inicios de los juegos por ordenador, allá por los tiempos del Spectrum® y del Commodore64®, para dibujar evitando un cierto efecto de “agua” en la pantalla, había que tener muy en cuenta la posición del haz de luz en el tubo de rayos catódicos (si, ¡las pantallas eran de rayos catódicos!).

Sólo existía el concepto de buffer de pantalla y había que irlo rellenando de información por detrás de la posición equivalente en la pantalla del aquel haz. Idealmente, lo que hacíamos era esperar a que ese haz abandonara la zona visible (antes la pantalla tenía un marco) y volviera a empezar por arriba. Sin duda que la complejidad técnica de aquello era alta y sólo alcanzable mediante código ensamblador, que aseguraba la velocidad y el control necesario para llevar a cabo esa tarea.

Hoy es mucho más sencillo. Las tarjetas gráficas vienen con SDKs que simplifican esta complejidad técnica. Así, se rellena un buffer temporal y se le dice a la tarjeta que vuelque dicho contenido en el buffer real en cuanto pueda. Y es ella la que tiene en cuenta la velocidad de refresco de la pantalla para hacerlo en el momento adecuado. En realidad, hay mucha mayor complejidad en el manejo de una tarjeta gráfica, pero no la necesitamos aquí y ahora.

La tarjeta gráfica tiene que saber, por tanto, dos cosas básicas: Cuando debe limpiar el buffer temporal porque empieza un nuevo dibujo, y cuando volcarlo al buffer principal. Eso se realiza introduciendo el concepto de **transacción**, de manera equivalente a como se realizan las operaciones contra una base de datos (de ahí viene en realidad el concepto).

Los objetos y elementos que intervienen en un juego ocupan posiciones en el espacio. Ese espacio se puede observar de muy diversas maneras (proyecciones) y desde muy diversas posiciones (sistemas de referencia). Todos esas posibilidades se agrupan en QGAMES bajo el concepto de cámara (Camera en QGAMES). La realidad (el juego) se puede observar con muchos tipos de **cámaras** e incluso con varias simultáneamente.

Aquello que es observado por la cámara termina casi siempre proyectado en una **pantalla o Screen** en QGAMES que incluso puede incorporar sus propios efectos, como añadir objetos tipo marcador por encima de todo, mostrando así una forma de estado del juego.

Bien, introducidos los tres conceptos fundamentales que intervienen en el dibujo de cosas, analicemos el detalle.

## La clase *Screen*

Es la responsable última de dibujar.

Su estructura básica se define en el fichero *screen.hpp*:

/\*\*

\* The screen is the place where all elements, objects, entities,...

\* being part of a game are drawn eventually. \n

\* So it defines basic instructions to draw basic elements being part of any other

\* element involved in the game: forms, circles, lines,... \n

\* The implementation of those primitives depends on the graphical library finally used behind. \n

\* \n

\* The best way to understand how the screen class works is imaging you wear a pair of glasses. \n

\* The game elements passing throught the limits of the glasses are projected

\* in the display. \n

\* \n

\* But what it is shown in your glasses is filmed through a camera always. \n

\* All things in the screen have to be drawn using a camera. \n

\* There have to be a default camera minimum created first time a drawing is tried. \n

\* \n

\* To draw something in the screen a graphical transaction is required. \n

\* Opening and closing a graphical transaction depends on the graphical library behind.

\* \n

\* A default camera is added. This camera is used to draw the score objects by default \n

\* but it can be also used for further purposes.

\*/

class Screen

{

public:

// Basic standard types of font supported by QGAMES library (for the method drawText)

static const int \_FONTBOLD;

static const int \_FONTITALIC;

static const int \_FONTUNDERLINE;

static const int \_FONTSTRIKETHROUGH;

static const int \_FONTNORMAL;

/\*\*

\* This class is to fix the screen over something (the position of that "something" is kept) \n

\* Depending on that the screen is moved/oriented in every loop automatically. \n

\* So, this is not a camera, it is to follow something and move the screen automatically. \n

\* For instance it can be used to turn around an object or to maintain the main

\* character at the center of the screen. \n

\* At the end of the day this class uses the current camera associated to the screen.

\*/

class ToFixScreen

{

public:

ToFixScreen (Screen\* s);

virtual ~ToFixScreen ()

{ }

virtual void moveScreen () = 0;

protected:

/\*\* A reference to the Screen this automatic camara controller is working over.

It can't be null. Something that is verified at construction time in debug mode. \*/

Screen\* \_screen;

};

/\*\*

\* A buffer could be in some graphical systems a temporal region of the memory,

\* where drawing actions happen. \n

\* Many graphical systems work using this as part of its standard. \n

\* When the transaction finishes it content can be knocking over the real visible part

\* of the graphical card. \n

\* @see Mask class.

\*/

class Buffer

{

public:

// The different types of blending supported by QGAMES library

static const int \_PUREBLENDMODE;

static const int \_ADDBLENDMODE;

static const int \_MODBLENDMODE;

static const int \_NOBLENDMODE;

/\*\* -1 means to get the size of the screen.

The buffer is always created over an screen. \*/

Buffer (Screen\* s, int w = -1, int h = -1);

virtual ~Buffer () // Just in case...

{ }

int width () const

{ return (\_width); }

int height () const

{ return (\_height); }

/\*\* In some graphical systems, the buffer has to be open and close.

It's something to be explicity defined under the graphical system itself. \*/

virtual void open () = 0;

virtual void close () = 0;

/\*\* To move the content to the screen. \*/

virtual void moveToScreen (int x = 0, int y = 0) = 0;

/\*\* To move the content of the buffer to other buffer.

This function allows to mix buffer before knocking tem over the real screen. \*/

virtual void moveToBuffer (Buffer\* b,int x = 0, int y = 0) = 0;

protected:

Screen\* \_screen;

int \_width, \_height;

};

Screen (const std::string& t, int w, int h,

const MapOfCameras& cms = MapOfCameras ());

virtual ~Screen (); // By default it destroyes the cameras...and the ToFixScreen method.

// Managing the main attributes of the screen...

const std::string& title () const

{ return (\_title); }

virtual void setIcon (Form\* f)

{ /\*\* Nothing by default \*/ }

int visualWidth () const

{ return (\_visualWidth); }

int visualHeight () const

{ return (\_visualHeight); }

Position visualCenter () const

{ return (Position (\_\_BD (\_visualWidth >> 1),

\_\_BD (\_visualHeight >> 1), \_\_BD 0)); }

Rectangle visualZone () const

{ return (Rectangle (Position::\_cero,

Position (\_\_BD \_visualWidth, \_\_BD \_visualHeight, \_\_BD 0))); }

virtual void windowAtPosition (int x, int y)

{ /\*\* Nothing by default. \*/ }

virtual void windowAtCenter ()

{ /\*\* Nothing by default. \*/ }

void setVisualAspect (int w, int h)

{ \_visualWidth = w; \_visualHeight = h; }

ToFixScreen\* fix ()

{ return (\_fix); }

void setFix (ToFixScreen\* f)

{ delete (\_fix); \_fix = f; }

// Mamaning the cameras...

MapOfCameras cameras ()

{ return (\_cameras); }

bool existsCamera (const std::string& id) const;

Camera\* camera (const std::string& id); // NULL when not find...

/\*\* No two cameras with the same id... \*/

void addCamera (Camera\* c);

void removeCamera (const std::string& id);

void removeAllCameras ();

Camera\* currentCamera () // It can be NULL, take care!

{ return (\_currentCamera); }

void setCurrentCamera (const std::string& id) // The result can be NULL...

{ \_currentCamera = camera (id); }

void setCurrentCamera (Camera\* c); // The camera has to be within the list of those...

/\*\* Can be overloaded. By default it takes the first in the list if any! \*/

virtual Camera\* defaultCamera ()

{ return ((\_cameras.size () > (int) 0)

? (\*\_cameras.begin ()).second : NULL); }

// These methods have been thought to manage the camera from the screen

// In many ocasions, there is only one camera operating in the game...

// Most of the methods don't work when the currentCamera has not been set!

const Position& position ();

void setPosition (const Position& p);

void move (const Position& p);

const Vector& orientation ();

void rotateAround (const Vector& n, const Position& p, bdata a);

void yaw (bdata a);

void pitch (bdata a);

void rotate (bdata a);

void yawPithRotate (bdata y, bdata p, bdata r);

bool isVisible (const Position& p);

bool isVisible (const Rectangle& r);

/\*\* The positions of the screens can be updated every game loop. \n

By default the screen doesn't move unless a fix element has been defined,

and update also the cameras associated. \*/

virtual void updatePositions ();

/\*\*

\* Most of the graphical libraries (SDL, OpenGL i.e) draws first everything on

\* a buffer that becomes visible with a single instruction. \n

\* The buffer has to be cleaned up before drawing a new content. \n

\* The instructions to do both (clear the buffer, and become the buffer visible

\* depends on the graphical library being used. \n

\* Two methods have been declared in this library: openTransaction, and closeTransaction. \n

\*/

void openTransaction ();

/\*\* When a buffer is created all drawing instructions must go to the buffer until it is destroyed. \*/

Buffer\* openBuffer (int tp, int w = -1, int h = -1);

void drawForm (const Position& p, Form\* f, int px, int py, int w, int h, int a = 255); // All need a camera selected!

void drawForm (const Position& p, int oX, int oY, Form\* f, int px, int py, int w, int h, int a = 255);

// Takes into account that an offset is possible in the form and it has oto be affected by camara conversions...

void drawPoint (const Position& p, const Color& c);

void drawLine (const Position& p1, const Position& p2, const Color& c, int t = 1); // t = thickness...must be positive

void drawRectangle (const Rectangle& r, const Color& c, bool f = false); // f = true, when a filled rectangle is requested

void drawRoundRectangle (const Rectangle& r, int rdx, const Color& c, bool f = false); // f = true when a filled rounded rectangled is requested

void drawPolygon (const Polygon& r, const Color& c, bool f = false); // f = true, when a filled rectangle is requested

void drawTriangle (const Position& p1, const Position& p2, const Position& p3,

const Color& c, bool f = false); // f = true, when a filled triangle is requested

void drawImage (const Position& p, const std::string& img, int a = 255);

void drawCircle (const Position& p, const Vector& o, bdata a1, bdata a2,

const Color& c, bool f = false); // F = true, when a filled ellipse is requested

void drawText (const Position& p, const std::string& s,

const std::string& f, int w, int h /\*\* Just if the system supports w and h different. \*/,

const Color& c, int lt = \_FONTNORMAL, int ot = 0);

/\*\* Some engines need the screen to be specificly update...But it not the normal situation! \*/

virtual void drawOnYourOwn ()

{ }

void closeTransaction ();

void closeBuffer (Buffer\* b);

protected:

/\*\*

\* Internal class to control the number of graphical transactions opened. \n

\* Only when first is created, or when the last one is closed eventually

\* the internal screen / buffer methods to open and close the transaction are invoked. \n

\* In some graphical libraries any time a transaction is open, even if another

\* one is already in use, a special instruction has to be invoked. \n

\* The transactions can be opened against the screen or a buffer of that.

\*/

class Transaction

{

public:

Transaction (Screen\* s)

: \_screen (s),

\_numberOfTx (0)

{ assert (s); }

virtual ~Transaction ()

{ }

Screen\* screen ()

{ return (\_screen); }

const Screen\* screen () const

{ return (\_screen); }

void open ();

void close ();

int numberOfTx () const

{ return (\_numberOfTx); }

protected:

virtual void openFirst () = 0;

virtual void closeLast () = 0;

virtual void openOne () = 0;

virtual void closeOne () = 0;

protected:

Screen\* \_screen;

int \_numberOfTx;

};

protected:

Transaction\* currentTransaction ()

{ return (\_transaction); }

const Transaction\* currentTransaction () const

{ return (\_transaction); }

// The very internal methods to define depending on the graphical libray behind.

virtual void drawForm (int x, int y, Form\* f, int px, int py, int w, int h, int a) = 0;

virtual void drawPoint (int x, int y, const Color& c) = 0;

virtual void drawLine (int x1, int y1, int x2, int y2, const Color& c, int t) = 0;

virtual void drawRectangle (int x1, int y1, int x2, int y2, const Color& c, bool f) = 0;

virtual void drawRoundRectangle (int x1, int y1, int x2, int y2, int rdx, const Color& c, bool f)

{ drawRectangle (x1, y1, x2, y2, c, f); } // By default it does the same that the previous one (not all graphical engines support it)

virtual void drawPolygon (int\* x, int\* y, int np, const Color& c, bool f) = 0;

virtual void drawTriangle (int x1, int y1, int x2, int y2, int x3, int y3, const Color& c, bool f) = 0;

virtual void drawImage (int x, int y, const std::string& i, int a) = 0;

virtual void drawCircle (int x, int y, int a1, int a2, const Color& c, bool f) = 0;

virtual void drawText (int x, int y, const std::string& s,

const std::string& f, int w, int h, const Color& c, int lt, int ot) = 0;

virtual void activeScale () { } // It is not necessary by default...

virtual void desactiveScale () { } // It is not necessary by default...

virtual Transaction\* createTransaction () = 0;

virtual Buffer\* createBuffer (int tp, int w, int h) = 0;

private:

// A very internal method very used when drawing a polygon from a list of positions

// It is supossed the positions are in order!

// This method can't be overloaded (not virtual) and it invokes the other one drawPolygon (int\* x, ...)

void drawPolygonFromScreenCoordinates (const ListOfScrCoordinates& cL, const Color& c, bool f);

protected:

std::string \_title;

int \_visualWidth, \_visualHeight;

ToFixScreen\* \_fix;

MapOfCameras \_cameras;

Camera\* \_currentCamera;

// Implementation

Transaction\* \_transaction;

};

// To simplify the manipulation of a set of screens...

typedef std::map <int, Screen\*> Screens;

}

13: La clase Screen, definida en screen.hpp

Es una clase larga y compleja. Quizás una de las que más, pero tampoco se trata de comprenderla por completo en estos momentos.

En ella podemos distinguir tres bloques de métodos fundamentales:

* Uno, el primero, que permite manejar las cámaras que proyectan sobre la pantalla. Ese bloque lo componen los métodos: ***cameras ()***, ***existsCamera ()***, ***addCamera ()***, ***removeCamera ()***, ***removeAllCameras ()***, ***currentCamera ()***, ***setCurrentCamera ()***, y ***defaultCamera ()***. No es muy difícil entender, atendiendo a su nombre, que función concreta desarrolla cada uno de ellos.

Como hemos dicho anteriormente, la información de la realidad puede estar siendo observada por muchas cámaras, de las cuales habremos de definir una como la que esté activa en ese preciso momento (***setCurrentCamera ()*** y ***currentCamera ()***).

* Otro bloque de código que nos permite manejar las transacciones de escritura en la pantalla, dado por los métodos: ***openTransaction ()*** y ***closeTransaction ()***.

Y, por último, un tercer bloque de código que es el que, en si nos permite dibujar, dado por todos los métodos públicos que contienen la palabra ***draw***.

Como habrá observado, estos últimos métodos públicos no son declarados como virtuales y, por lo tanto, no pueden sobrecargarse. Cada uno de esos métodos tiene un método asociado protegido (*protected*) que, si es declarado como virtual y, por tanto, sí que puede sobrecargarse.

Por ejemplo, el método:

void drawPoint (const Position& p, const Color& c);

tiene asociado:

virtual void drawPoint (int x, int y, const Color& c) = 0;

virtual puro.

El primer método invoca de alguna manera al segundo: Recibe como parámetro una referencia al punto a dibujar y el color de éste. Ese método, utilizará la cámara (ahora lo veremos) seleccionada para convertir los puntos del rectángulo en coordenadas de la pantalla, antes de invocar al segundo.

He aquí su código fuente como ejemplo:

// ---

void QGAMES::Screen::drawPoint (const QGAMES::Position& p, const QGAMES::Color& c)

{

if (!\_currentCamera)

return;

QGAMES::VisualCloudPoints pts = \_currentCamera -> calculateVisualPositions (p);

if (!pts.\_areVisible)

return; // If the figure delimited by the points is not going to be visible,

// then nothing to do!

\_currentCamera -> beforeDrawing ();

drawPoint (pts.\_points [0].\_x, pts.\_points [0].\_y, c);

\_currentCamera -> afterDrawing ();

}

14: Método drawPoint en la clase Screen

Las coordenadas del punto que recibe el primer método están relacionadas con el espacio total del juego (coordenadas universales, en 3 dimensiones), mientras que los valores que recibe el segundo son ya píxeles de la pantalla (en 2 dimensiones). La conversión de unos en otros se realiza mediante la cámara.

La clase *Screen* es, en realidad y como hemos apuntado anteriormente, por tanto, un wrapper sobre una librería gráfica cuya responsabilidad sea la de dibujar píxeles en la pantalla.

Por ejemplo, la implementación concreta del wrapper sobre SDL, tiene esta forma:

/\*\* \ingroup SDL \*/

/\*\* \ingroup Screen \*/

/\*@{\*/

/\*\*

\* @file

\* File: sdlscreen.hpp \n

\* Framework: Commty Game Library (CGL) \n

\* Author: Ignacio Cea Forniés (Community Networks) \n

\* Creation Date: 01/12/2014 \n

\* Description: Defines a sdl implememtation of an screen. \n

\* Versions: 1.0 Initial

\*/

#ifndef \_\_QGAMES\_SDLSCREEN\_\_

#define \_\_QGAMES\_SDLSCREEN\_\_

#include <Common/screen.hpp>

#include <SDL.h>

#include <SDL\_ttf.h>

#include <map>

/\*\* Class to represent a zommer in SDL.

This is a software zoomer. It declares methods to activate and desactivate the zoom using

the main directives of the SDL framework. \*/

class SDLZoommer : public QGAMES::Zoommer

{

public:

SDLZoommer (QGAMES::bdata v, int mW, int mH, SDL\_Renderer\* r,

QGAMES::Zoommer::Effect e = QGAMES::Zoommer::Effect::\_TOUPLEFT);

/\*\* @see parent. \*/

virtual void beforeDrawing ();

virtual void afterDrawing ();

private:

SDL\_Renderer\* \_render;

};

/\*\* Class to represent an screen in SDL framework. \*/

class SDLScreen : public QGAMES::Screen

{

public:

class SDLBuffer : public QGAMES::Screen::Buffer

{

public:

SDLBuffer (QGAMES::Screen\* s, SDL\_BlendMode bM, int w = -1, int h = -1);

virtual ~SDLBuffer ();

/\*\* @see parent. \*/

virtual void open ();

virtual void close ();

virtual void moveToScreen (int x = 0, int y = 0);

virtual void moveToBuffer (QGAMES::Screen::Buffer\* b, int x = 0, int y = 0);

private:

// Implementation

SDL\_Texture\* \_textureBuffer;

};

/\*\* The scale is only used in this version of SDL to determinate where to center the screen,

just because the SDL members returning the resolution of the screen don't take into account

the windows DPI adjust. The rest of SDL functions work properly. \*/

SDLScreen (const std::string& t, const QGAMES::Position& p, int w, int h,

int wPX, int wPY, float sX = 1.0f, float sY = 1.0f);

virtual ~SDLScreen ();

virtual void setIcon (QGAMES::Form\* f);

virtual void windowAtPosition (int x, int y);

virtual void windowAtCenter ();

SDL\_Window\* window ()

{ return (\_window); }

SDL\_Renderer\* render ()

{ return (\_render); }

protected:

class SDLTransaction : public QGAMES::Screen::Transaction

{

public:

SDLTransaction (QGAMES::Screen\* s)

: QGAMES::Screen::Transaction (s)

{ }

protected:

/\*\* @see parent. \*/

virtual void openFirst ();

virtual void closeLast ();

virtual void openOne () { } // Nothing special opening one transaction

virtual void closeOne () { }

};

friend SDLTransaction;

friend SDLBuffer;

protected:

virtual void drawForm (int x, int y, QGAMES::Form\* f, int px, int py, int w, int h, int a);

virtual void drawPoint (int x, int y, const QGAMES::Color& c);

virtual void drawLine (int x1, int y1, int x2, int y2, const QGAMES::Color& c, int t);

virtual void drawRectangle (int x1, int y1, int x2, int y2, const QGAMES::Color& c, bool f);

/\*\* This function won't work with a projection diferent than orthogonal... \*/

virtual void drawRoundRectangle (int x1, int y1, int x2, int y2, int rdx, const QGAMES::Color& c, bool f);

virtual void drawPolygon (int\* x, int\* y, int np, const QGAMES::Color& c, bool f);

virtual void drawTriangle (int x1, int y1, int x2, int y2, int x3, int y3, const QGAMES::Color& c, bool f);

virtual void drawImage (int x, int y, const std::string& i, int a);

virtual void drawCircle (int x, int y, int a1, int a2, const QGAMES::Color& c, bool f);

virtual void drawText (int x, int y, const std::string& s,

const std::string& f, int w, int h, const QGAMES::Color& c, int lt, int ot);

virtual QGAMES::Screen::Transaction\* createTransaction ()

{ return (new SDLTransaction (this)); }

virtual QGAMES::Screen::Buffer\* createBuffer (int tp, int w, int h);

private:

void convertCoordinatesIntToSInt16 (int x, int y, Sint16& X, Sint16& Y);

private:

SDL\_Window\* \_window;

SDL\_Renderer\* \_render;

int \_windowPositionX, \_windowPositionY;

float \_scaleX, \_scaleY;

// Implementation

std::map <std::string, TTF\_Font\*> \_fontsLoaded;

};

#endif

// End of the file

/\*@}\*/

15: La clase SDLScreen. Wrapper sobre SDL para dibujar

En donde, por ejemplo, el código fuente del método ***drawPoint ()*** es:

// ---

void SDLScreen::drawPoint (int x, int y, const QGAMES::Color& c)

{

Sint16 X, Y;

convertCoordinatesIntToSInt16 (x, y, X, Y);

pixelRGBA (\_render, X, Y, c.red (), c.green (), c.blue (), c.alpha ());

}

16; Dibujar un punto en SDL

Se puede observar que se invoca al método ***pixelRGBA ()*** definido en la librería SDL tras convertir las coordenadas recibidas de números enteros al tipo *Sint16* propios de SDL.

## La clase *Camera*

De forma muy elemental podemos conceptualizar una cámara como algo con las características de su equivalente real: Está localizada en una posición del espacio, y dispone de un cristal a través del cual se ve (proyecta) la realidad. Es esa realidad la que luego se traslada a una posición determinada de la pantalla.

La responsabilidad de una cámara es la de convertir los puntos de coordenadas universales a coordenadas concretas sobre su cristal[[8]](#footnote-8).

La cabecera de la clase *Camera* también se encuentra definida en *screen.hpp*:

/\*\* This class represents a pair of coordinates in a visual screen. \n

The coordinates in a visual screen have always x and y value only. \n

When a point in the space is represented on a screen there are only x and y values and

as they are pixels int type is used. \*/

struct ScrCoordinates

{

/\*\* A default constructor is declared to be sure the coordinates are set to 0. \*/

ScrCoordinates ()

: \_x (0), \_y (0) { }

/\*\* And a constructor for basic data. \*/

ScrCoordinates (int x, int y )

: \_x (x), \_y (y) { }

int \_x, \_y;

};

/\*\* To simplify how to manage a list of screen coordinates. \*/

typedef std::vector <ScrCoordinates> ListOfScrCoordinates;

/\*\* A structure to represent a list of visual screen coordinates. \n

Apart of the list itself, this class defines an internal attribute

to indicate whether the list is or not visible in the limits

of the screen using them. This last information can be used by other methods

to determinate whether the points are drawn eventually. \*/

struct VisualCloudPoints

{

/\*\* A default constructor is defined to set the internal variables

to the right default value. \*/

VisualCloudPoints ()

: \_areVisible (false), \_points () { }

/\*\* And a normal constructor too. \*/

VisualCloudPoints (bool v, const ListOfScrCoordinates& lC)

: \_areVisible (v), \_points (lC) { }

bool \_areVisible;

ListOfScrCoordinates \_points;

};

/\*\* A class to represent zoom effects over a coordinate in the screen. \n

When a value is zoomed three options can be taken into account: \n

1.- To maintain the point in the same "square" of the camera. \n

2.- To maintain it at the left up corner of it or

2.- To do same in the down right corner. \*/

class Zoommer

{

public:

/\*\* The possible effects. \*/

enum Effect

{

\_CENTER = 0,

\_TOUPLEFT = 1,

\_TODOWNRIGHT = 2

};

/\*\* To store the internal status of the zoommer. \*/

struct Status

{

Status (bdata v, int mW, int mH, Effect e, const Vector& sV)

: \_value (v),

\_maximumWidth (mW), \_maximumHeight (mH),

\_effect (e),

\_scaleVector (sV)

{ }

/\*\* The value of the zoom itself. \*/

bdata \_value;

/\*\* Effect of the zoom. Teh refecete point actually. \*/

Effect \_effect;

/\*\* The size the rectangle where to apply the zoom in. \*/

int \_maximumWidth, \_maximumHeight;

/\*\* The scale vector calculated. \*/

Vector \_scaleVector;

};

/\*\* Constructor:

The value of the zoom, that can't be less than 0,005 (to avoid divisions by 0),

The parameters receives are:

The maximum width and height of the screen, the type of effect when zoomming,

Whether the zoom is going to be done by software (the framework behind) or

using an algorithm implemented by QGAMES' makers. \*/

Zoommer (bdata v, int mW, int mH, Effect e = \_TOUPLEFT, bool sw = false)

: \_status (v, mW, mH, e, Vector::\_cero), \_useSoftware (sw)

{ calculateVectorScale (); adjustValue (); }

/\*\* To change the value of the zoom. \*/

bdata value () const

{ return (\_status.\_value); }

void setValue (bdata v)

{ \_status.\_value = v; calculateVectorScale (); adjustValue (); }

/\*\* To change the type of zoom. \*/

Effect effect () const

{ return (\_status.\_effect); }

void setEffect (Effect e)

{ \_status.\_effect = e; }

/\*\* To know whether software is used to make the zoomming. \*/

bool useSoftware () const

{ return (\_useSoftware); }

/\*\* To change the limits where the zoom happens. \*/

void adjustLimits (int mW, int mH)

{ \_status.\_maximumWidth = mW; \_status.\_maximumHeight = mH; calculateVectorScale (); }

/\*\* To move the zoomer to the original position.

This instruction doesn't modify the dimensions the zommer is working on. \*/

void toInitialPosition ();

/\*\* Affecting the different coordinates.

This are the very important methods to be used. \*/

bdata effectOverX (bdata v) const

{ return ((\_status.\_value == (bdata) 1) ? v : effectOver (v, true)); }

bdata effectOverY (bdata v) const

{ return ((\_status.\_value == (bdata) 1) ? v : effectOver (v, false)); }

/\*\* To get and set the internal status. \*/

const Status& currentStatus () const

{ return (\_status); }

void setStatus (const Status& st);

/\*\* These methods are used before and after drawing to

prepare the context. By default they don't do anything.

But usually, when the framework is used to zoom instead the internal algorithm,

this methods are useful to activate and desactivate the framework's

internal zooming methods. \*/

virtual void beforeDrawing () { }

virtual void afterDrawing () { }

protected:

/\*\* When zoom is done by the underlined drawing framework used,

This method could have to to be rewritten maybe. \*/

virtual bdata effectOver (bdata v, bool x = true) const;

private:

// Implementation

void calculateVectorScale ();

void adjustValue () // To avoid division by cero...

{ if (\_status.\_value <= (bdata) 0) \_status.\_value = (bdata) 0.005; }

public:

Status \_status;

bool \_useSoftware;

};

/\*\* A class to represent a camera. \n

The camera is the element used to project things on a screen. \n

So, the camera has the intelligence to scale, zoom and move the point of view in any scene. \n

A camera has always to be defined before drawing anything on the screen. \*/

class Camera : public Observer

{

public:

/\*\* To know the very internal status of the camera.

This structure could be used both for reading and setting back a status. \*/

struct Status

{

Status ()

: \_position (Position::\_cero),

\_maximumWidth (-1), \_maximumHeight (-1),

\_orientation (Vector::\_zNormal), \_horizontalVector (Vector::\_xNormal),

\_yawAngle ((bdata) 0), \_pitchAngle ((bdata) 0), \_rollAngle ((bdata) 0),

\_screenRatio ((bdata) 1), \_eulerMatrix ((bdata) 1), \_calculateVisibiliyBySw (true),

\_conicLimitZ ((bdata) -100) // The very default...

{ }

Status (int mW, int mH,

const Position& p, const Vector& v, const Vector& h,

bdata yA, bdata pA, bdata rA,

bdata sR, const Matrix3& eM, bool vS,

bdata cL)

: \_position (p),

\_maximumWidth (mW), \_maximumHeight (mH),

\_orientation (v), \_horizontalVector (h),

\_yawAngle (yA), \_pitchAngle (pA), \_rollAngle (rA),

\_screenRatio (sR), \_eulerMatrix (eM), \_calculateVisibiliyBySw (vS),

\_conicLimitZ (cL)

{ }

/\*\* The maximum width the camera is watching. \*/

int \_maximumWidth;

/\*\* The maximum height the camera is watching. \*/

int \_maximumHeight;

/\*\* The position of the camera in the space. \*/

Position \_position;

/\*\* The direction the camera is looking to.

Its module should be always 1. \*/

Vector \_orientation;

/\*\* If the camara is like a rectangle. It can be horizontal respect the

ground or rotated. This is vector to indicate the orintation of that limit.

It is usefuill to calculate many other things. \*/

Vector \_horizontalVector;

/\*\* The angles the reference system is turned. \*/

bdata \_yawAngle, \_pitchAngle, \_rollAngle;

/\*\* The ratio of the screen. \*/

bdata \_screenRatio;

/\*\* To implement the movements in the screen. \*/

Matrix3 \_eulerMatrix;

/\*\* To define whether the visibility of a line / poit is calculated by software or not. \*/

bool \_calculateVisibiliyBySw;

// Specific for conic projection

bdata \_conicLimitZ;

};

/\*\* Sometimes the camera has an external representation.

e.g in OGRE to manipulate this camera will imply to do things in a OGRE's camera object. \n

This class is to define specific additional external methods when

the camera's position or orientation are moved. \n

The method has to be redefined with an specific implementation if needed. \*/

class ExternalLink

{

public:

virtual void setPosition (const Position& pos) = 0;

virtual void setOrientation (const Vector& o, const Vector& hv) = 0;

};

/\*\* The camera can project things on the screen in 4 ways ony. \*/

enum ProjectionType

{

// The basic one, when the distance to the focus of the camera doesn't matter.

\_ORTHOGONAL = 0,

\_DIEDRIC = 1,

\_QUICKDIEDRIC = 2,

\_CAVALIER = 3,

\_QUICKCAVALIER = 4,

// The way 3D engines usually work.

\_CONIC = 5

};

/\*\*

\* To create a camera. \n

\* The camera is who calculates the final screen point. \n

\* It has to be done using a type of projection (all types possible are supported),

\* an a zoommer to scale the final point. It can be null meaning no zoom is used. \n

\* The camera controls also the maximum zone visualized,

\* and its position and orientation in the space. \n

\* The position is the position of the left up corner, and the orientation is the normal vector to the rectangle

\* representing the camera. By default the orientation will be looking right to the front, but it can be changed

\* using the right method to do so. \n

\* To take into account in the orientation of the camera at the beginning: \n

\* (Z) x---------> (X) \n

\* | \n

\* | \n

\* | \n

\* | \n

\* v \n

\* y

\*/

Camera (const std::string& n, ProjectionType pt, Zoommer\* z, int mW, int mH,

const Position& pos = Position::\_cero)

: \_name (n),

\_projectionType (pt),

\_zoommer (NULL),

\_status (mW, mH, pos, Vector::\_zNormal, Vector::\_xNormal,

(bdata) 0, (bdata) 0, (bdata) 0,

(bdata) mW / (bdata) mH, Matrix3 ((bdata) 1), true,

-(bdata) mW / (bdata) 2),

\_link (NULL)

{ setZoomer (z); }

virtual ~Camera () // The camera own the zoommer and the external link. Only one zoommer per camera.

{ delete (\_zoommer); delete (\_link); }

// The attributes...

const std::string& name () const // Can't be changed...

{ return (\_name); }

Zoommer\* zoommer () const

{ return (\_zoommer); }

ProjectionType projectionType () const

{ return (\_projectionType); }

void setProjectType (ProjectionType pt)

{ \_projectionType = pt; }

void setZoomer (Zoommer\* z)

{ if ((\_zoommer = z) != NULL) // The zommer has to be adjusted to the screen limits...

\_zoommer -> adjustLimits (\_status.\_maximumWidth, \_status.\_maximumHeight); }

int width () const // The visual width...

{ return (\_status.\_maximumWidth); }

int height () const // The visual height...

{ return (\_status.\_maximumHeight); }

void adjustLimitsTo (int mW, int mH);

void adjustLimitsTo (Screen\* scr);

Position center () const;

bool calculateVisibilityBySoftware () const

{ return (\_status.\_calculateVisibiliyBySw); }

void setCalculateVisibilityBySoftware (bool vS)

{ \_status.\_calculateVisibiliyBySw = vS; }

/\*\* To change the external link if needed. \*/

ExternalLink\* externalLink ()

{ return (\_link); }

void setExternalLink (ExternalLink\* link)

{ \_link = link; }

// Managing the basic attributes related with the position of the camera

const Position& position () const

{ return (\_status.\_position); }

/\*\* Sometimes the game engines manages its own internal cameras and

this method has to be overloaded to change engine's camera also. \*/

virtual void setPosition (const Position& pos)

{ \_status.\_position = pos;

if (\_link) \_link -> setPosition (pos); }

void moveOnDirection (const Vector& d, bdata p)

{ setPosition (position () + (Vector (d).normalize () \* p)); }

// Mamaging the attributes related with the orientation

const Vector& orientation () const

{ return (\_status.\_orientation); }

/\*\* To get the front, back, right, left, up and down direction (normal vector) of the camera,

taking into account its direction. \*/

Vector frontDirection () const

{ return (Vector (\_status.\_orientation).normalize ()); }

Vector backDirection () const

{ return (-frontDirection ()); }

Vector rightDirection () const

{ return (Vector (\_status.\_horizontalVector).normalize ()); }

Vector leftDirection () const

{ return (-rightDirection ()); }

Vector upDirection () const

{ return (\_status.\_horizontalVector.crossProduct (\_status.\_orientation).normalize ()); }

Vector downDirection () const

{ return (-upDirection ()); }

/\*\* Just rotate the orientation around a point and taking into account a orientation.

The eurler matrix is calculated back. \*/

virtual void rotateAround (const Vector& n, const Position& p, bdata a);

/\*\* To yaw.

Like moving the head left and right.

Rotating over the axis OY. \*/

void yaw (bdata a)

{ \_status.\_yawAngle += a;

rotateAround (\_status.\_orientation.crossProduct (\_status.\_horizontalVector), Position::\_cero, a); }

/\*\* To pitch.

Like moving the head up and down.

Rotating over the axis OX. \*/

void pitch (bdata a)

{ \_status.\_pitchAngle += a;

rotateAround (\_status.\_horizontalVector, Position::\_cero, a); }

/\*\* To rotate around the orientation.

Like rotating the head left and right.

Rotating over the axis OZ. \*/

void rotate (bdata a)

{ \_status.\_rollAngle += a;

rotateAround (\_status.\_orientation, Position::\_cero, a); }

/\*\* All together. \*/

void yawPithRotate (bdata y, bdata p, bdata r)

{ yaw (y), pitch (p); rotate (r); }

/\*\* To move the camera to the very initial position. \*/

void toInitialPosition ();

/\*\* To manage the internal status of the camera.

Take care using this very internal method. \*/

const Status currentStatus () const

{ return (\_status); }

void setStatus (const Status& s);

// To convert points to the right projection.

Positions convert (const Positions& ps);

Positions convert (const Position& p1)

{ Positions ps (1); ps [0] = p1;

return (convert (ps)); }

Positions convert (const Position& p1, const Position& p2)

{ Positions ps (2); ps [0] = p1; ps [1] = p2;

return (convert (ps)); }

Positions convert (const Position& p1, const Position& p2, const Position& p3)

{ Positions ps (3); ps [0] = p1; ps [1] = p2; ps [2] = p3;

return (convert (ps)); }

Positions convert (const Position& p1, const Position& p2,

const Position& p3, const Position& p4)

{ Positions ps (4); ps [0] = p1; ps [1] = p2; ps [2] = p3; ps [3] = p4;

return (convert (ps)); }

/\*\*

\* These are the very important method in the class. \n

\* The method receives a set of points in the space to be fit in the zone

\* visualized by the camera, and returns the position of those in the screen,

\* taking into account the position and orientation of the camera and the zoom being

\* used by it, and also the type of projection. \n

\* It is supossed the list of positions passed as parameter represents always a polygon. \n

\* Then the result includes always an attribute to indicate whether is possible the

\* polygon is in the limits or not. \n

\* This paramter is only useful when the method has receive the instruction to check whether

\* the final points are or not with in the visual zone. This is the default behaviour. \n

\*/

VisualCloudPoints calculateVisualPositions (const Positions& ps);

VisualCloudPoints calculateVisualPositions (const Position& p1)

{ Positions ps (1); ps [0] = p1;

return (calculateVisualPositions (ps)); }

VisualCloudPoints calculateVisualPositions (const Position& p1, const Position& p2)

{ Positions ps (2); ps [0] = p1; ps [1] = p2;

return (calculateVisualPositions (ps)); }

VisualCloudPoints calculateVisualPositions (const Position& p1,

const Position& p2, const Position& p3)

{ Positions ps (3); ps [0] = p1; ps [1] = p2; ps [2] = p3;

return (calculateVisualPositions (ps)); }

VisualCloudPoints calculateVisualPositions (const Position& p1,

const Position& p2, const Position& p3, const Position& p4)

{ Positions ps (4); ps [0] = p1; ps [1] = p2; ps [2] = p3; ps [3] = p4;

return (calculateVisualPositions (ps)); }

// Just to project a set of points. Not to zoom them.

Positions justProject (const Positions& ps)

{ return (convert (ps)); }

Positions justProject (const Position& p1)

{ Positions ps (1); ps [0] = p1;

return (justProject (ps)); }

Positions justProject (const Position& p1, const Position& p2)

{ Positions ps (2); ps [0] = p1; ps [1] = p2;

return (justProject (ps)); }

Positions justProject (const Position& p1, const Position& p2, const Position& p3)

{ Positions ps (3); ps [0] = p1; ps [1] = p2; ps [2] = p3;

return (justProject (ps)); }

Positions justProject (const Position& p1, const Position& p2,

const Position& p3, const Position& p4)

{ Positions ps (4); ps [0] = p1; ps [1] = p2; ps [2] = p3; ps [3] = p4;

return (justProject (ps)); }

// Just to Scale a set of points. The position of the point is not change.

VisualCloudPoints justScale (const Positions& ps);

VisualCloudPoints justScale (const Position& p1)

{ Positions ps (1); ps [0] = p1;

return (justScale (ps)); }

VisualCloudPoints justScale (const Position& p1, const Position& p2)

{ Positions ps (2); ps [0] = p1; ps [1] = p2;

return (justScale (ps)); }

VisualCloudPoints justScale (const Position& p1, const Position& p2, const Position& p3)

{ Positions ps (3); ps [0] = p1; ps [1] = p2; ps [2] = p3;

return (justScale (ps)); }

VisualCloudPoints justScale (const Position& p1, const Position& p2,

const Position& p3, const Position& p4)

{ Positions ps (4); ps [0] = p1; ps [1] = p2; ps [2] = p3; ps [3] = p4;

return (justScale (ps)); }

/\*\* To move the camera in every iteration of the screen...if any.

The method can be overloaded in any class inheriting from this one. \n

called from the method updatePositions in the Screen. \*/

virtual void updatePositions ()

{ }

/\*\* These methods are used before and after drawing to

prepare the context. \n

By default they don't do anything except calling

the similar method in the zoommer. \*/

virtual void beforeDrawing ()

{ if (\_zoommer) \_zoommer -> beforeDrawing (); }

virtual void afterDrawing ()

{ if (\_zoommer) \_zoommer -> afterDrawing (); }

/\*\* @see parent.

Not needed by default o observe events. \*/

virtual void processEvent (const Event& e)

{ }

protected:

/\*\* Sometimes the game engine manages its own internal camera and

this method has to be overloaded to change that camera also.

This method is only invoked internally. \*/

virtual void setOrientation (const Vector& o, const Vector& hv)

{ \_status.\_orientation = o; \_status.\_horizontalVector = hv;

if (\_link) \_link -> setOrientation (o, hv); }

protected:

/\*\* The name of the camera. \*/

std::string \_name;

/\*\* The type of projection the camera will generate. \*/

ProjectionType \_projectionType;

/\*\* The zoom effect that the camera will apply.

It could be NULL meaning no zoom will be apply. \*/

Zoommer\* \_zoommer;

/\*\* The internal status. \*/

Status \_status;

/\*\* A potential external reference.

It is null by default. \*/

ExternalLink\* \_link;

};

17: La clase Camera. Definida en screen.hpp

*Camera* es una clase larga también quizás compleja. Quizás una de las más complejas en QGAMES. Desgranémosla un poco:

*Camera* se apoya en otras tres clases definidas antes que ella: ***ScrCoordinates***, ***VisualCloudPoints*** y ***Zommer***, de las que sólo ésta última tiene interés en este documento. Profundizaremos en ella un poco más adelante.

Todos los parámetros que caracterizan a una cámara se definen en una clase anidada dentro de *Camera* llamada ***Status***:

struct Status

{

Status ()

: \_position (Position::\_cero),

\_maximumWidth (-1), \_maximumHeight (-1),

\_orientation (Vector::\_zNormal), \_horizontalVector (Vector::\_xNormal),

\_yawAngle ((bdata) 0), \_pitchAngle ((bdata) 0), \_rollAngle ((bdata) 0),

\_screenRatio ((bdata) 1), \_eulerMatrix ((bdata) 1), \_calculateVisibiliyBySw (true),

\_conicLimitZ ((bdata) -100) // The very default...

{ }

Status (int mW, int mH,

const Position& p, const Vector& v, const Vector& h,

bdata yA, bdata pA, bdata rA,

bdata sR, const Matrix3& eM, bool vS,

bdata cL)

: \_position (p),

\_maximumWidth (mW), \_maximumHeight (mH),

\_orientation (v), \_horizontalVector (h),

\_yawAngle (yA), \_pitchAngle (pA), \_rollAngle (rA),

\_screenRatio (sR), \_eulerMatrix (eM), \_calculateVisibiliyBySw (vS),

\_conicLimitZ (cL)

{ }

/\*\* The maximum width the camera is watching. \*/

int \_maximumWidth;

/\*\* The maximum height the camera is watching. \*/

int \_maximumHeight;

/\*\* The position of the camera in the space. \*/

Position \_position;

/\*\* The direction the camera is looking to.

Its module should be always 1. \*/

Vector \_orientation;

/\*\* If the camara is like a rectangle. It can be horizontal respect the

ground or rotated. This is vector to indicate the orintation of that limit.

It is usefuill to calculate many other things. \*/

Vector \_horizontalVector;

/\*\* The angles the reference system is turned. \*/

bdata \_yawAngle, \_pitchAngle, \_rollAngle;

/\*\* The ratio of the screen. \*/

bdata \_screenRatio;

/\*\* To implement the movements in the screen. \*/

Matrix3 \_eulerMatrix;

/\*\* To define whether the visibility of a line / poit is calculated by software or not. \*/

bool \_calculateVisibiliyBySw;

// Specific for conic projection

bdata \_conicLimitZ;

};

18: La clase Status. Definida en Screen.hpp

Todos ellos son auto explicativos y tiene que ver sobre todo con el tamaño del cristal de la cámara (que puede ser más pequeño incluso que la pantalla en la que luego se va a proyectar), la orientación de ésta en el espacio, etc.

Existen métodos definidos dentro de Camera para obtener este estado de la cámara (***currentStatus ()***) o incluso cambiarlo (***setStatus ()***). Métodos muy empleados si estuviéramos utilizando por ejemplo la misma cámara para obtener diferentes tipos de proyección en el espacio.

En una cámara, el origen de referencia está localizado siempre en la esquina superior izquierda. El eje (1,0,0) es positivo hacia la derecha, el (0,1,0) positivo hacia abajo y el (0,0,1) positivo hacia el fondo. Inicialmente su orientación coincide con la orientación del del espacio universal en el que se van a ubicar los elementos del juego. Algo que no será así conforme movamos la cámara.

*Camera* permite diferentes formas de proyectar los puntos del espacio sobre su cristal: **\_ORTHOGONAL, \_DIEDRIC, \_QUICKDIEDRIC, \_CAVALIER, \_QUICKCAVALIER, y \_CONIC**.

\_ORTOGONAL es habitualmente usado para juegos tipo Mario Bross®, \_DIEDRIC y \_QUICKCALIER para juegos tipo Clash of Clans®, \_CAVALIER y \_QUICKCAVALIER para juegos tipo Entombed® y \_CONIC para juegos 3D.

Los métodos más importantes que se definen en Camera son aquellos que permiten moverla y cambiar su orientación en el espacio:

* ***setPosition ()***: Coloca la esquina superior izquierda de la cámara en una posición determinada del espacio.
* ***moveOnDirection ()***: Para mover la cámara una determinada longitud en la dirección de un determinado vector. Ambos recibidos como parámetro.
* ***yaw (), pitch () y rotate ()***: Para girar la orientación de la cámara en el espacio alrededor respectivamente del eje , y .

El constructor de una cámara es:

Camera (const std::string& n, ProjectionType pt, Zoommer\* z, int mW, int mH,

const Position& pos = Position::\_cero)

: \_name (n),

\_projectionType (pt),

\_zoommer (NULL),

\_status (mW, mH, pos, Vector::\_zNormal, Vector::\_xNormal,

(bdata) 0, (bdata) 0, (bdata) 0,

(bdata) mW / (bdata) mH, Matrix3 ((bdata) 1), true,

-(bdata) mW / (bdata) 2),

\_link (NULL)

{ setZoomer (z); }

19: Constructor de Camera

Necesita los siguientes 6 parámetros:

* Un **nombre** que ha de ser único dentro de la pantalla (*Screen*) para que la que vaya a trabajar.
* Un **tipo de proyección**, que puede ser: \_ORTHOGONAL, \_DIEDRIC, \_QUICKDIEDRIC, \_CAVALIER, \_QUICKCAVALIER, o \_CONIC.
* Un **objeto *Zoomer*** que se encargará de agrandar, mantener o empequeñecer los objetos proyectados.
* **Las dimensiones** de su zona visible, en pixeles y que no tienen por qué coincidir con los de la pantalla en la que va a proyectarse.
* Y la **localización inicial** de la misma en el espacio. Por defecto el origen de coordenadas.

Dentro del fichero *screen.hpp*, se definen también diferentes tipos de cámara estándar (que heredan de la clase *Camera*):

* ***StaticOrthogonalDefaultCamera***, es la más básica de las cámaras y no permite que se cambie ni su posición dentro de la pantalla, ni su orientación. Es eso, una cámara básica estática.
* ***CameraControlledByInputHandler***. Esta cámara puede conectarse a un *InputHandler* (que leería por ejemplo los movimientos del joystick, teclado y ratón) para moverla acorde a ellos a lo largo del espacio. Profundizaremos un poco más adelante.
* ***FreeMovementCamera***: Hereda de la anterior y define un conjunto de teclas (que pueden ser modificadas) para avanzar, retroceder, subir y bajar, más deprisa o más despacio la posición de la cámara.

Cuando se crear un objeto *Screen* se crea siempre una cámara por defecto tipo ***StaticOrthogonalDefaultCamera***. El nombre de ésta es: ***\_\_QGAMES\_DEFAULTCAMERAID\_\_*** (definido en *.\include\Common\definitions.hpp*) que es igual a ***"DEFAULTCAMERA"***.

La clase *Screen* también selecciona como “cámara inicial en funcionamiento” la primera que haya en la lista de cámaras (habitualmente esta que hemos mencionado y que acaba de crearse por defecto) y la ajusta en tamaño de proyección a las dimensiones de la pantalla. Así:

// ---

QGAMES::Screen::Screen (const std::string& t, int w, int h, const QGAMES::MapOfCameras& cms)

: \_title (t),

\_visualWidth (w),

\_visualHeight (h),

\_fix (NULL),

\_cameras (cms), // This list has to be fulfilled first time a drawn is tried...

\_currentCamera (NULL),

\_transaction (NULL)

{

// Adds a default camera for simple purposes.

// This camera is used e.g to draw the score objects.

addCamera (new StaticOrthogonalDefaultCamera (w, h));

// If the list of cameras received is not empty

// Selects the first as current camera and adjust their limits to the screen...

if (!\_cameras.empty ())

{

\_currentCamera = (\*\_cameras.begin ()).second;

\_currentCamera -> adjustLimitsTo (this);

}

}

20: Constructor de la clase Screen

El resto de los métodos que define la clase *Camera* carecen de interés de momento. Son utilizados internamente por *Screen* para terminar finalmente proyectando dibujos, formas, etc. en la pantalla.

## La clase *Transaction*

Dentro de la clase *Screen*, se define (anidada) la clase ***Transaction***.

Esta clase es nuevamente un wrapper sobre el sistema gráfico y que se ocupa de preparar al sistema gráfico para finalmente dibujar sobre él. Es la clase que se encarga de indicarle en primera instancia al sistema gráfico que los dibujos los almacene en el buffer temporal al que aludíamos al principio de este capítulo, y también la que se encarga de volcar ese buffer a la pantalla cuando se ha terminado un bucle del programa y la tarjeta gráfica lo entienda oportuno.

Esos métodos son ***open ()*** y ***close ()***. Esos métodos terminan llamando a ***openFirst ()***, ***openOne ()***, ***closeLast ()*** y ***closeOne ()*** declarados como virtuales puros. Son esos métodos los que hay que sobrecargar en nuestro wrapper específico transaccional sobre el sistema gráfico que corresponda.

class Transaction

{

public:

Transaction (Screen\* s)

: \_screen (s),

\_numberOfTx (0)

{ assert (s); }

virtual ~Transaction ()

{ }

Screen\* screen ()

{ return (\_screen); }

const Screen\* screen () const

{ return (\_screen); }

void open ();

void close ();

int numberOfTx () const

{ return (\_numberOfTx); }

protected:

virtual void openFirst () = 0;

virtual void closeLast () = 0;

virtual void openOne () = 0;

virtual void closeOne () = 0;

protected:

Screen\* \_screen;

int \_numberOfTx;

};

21: La clase Transaction, definida dentro de screen.hpp

En el caso de SDL, esos métodos se implementan, por ejemplo, de la siguiente manera:

class SDLTransaction : public QGAMES::Screen::Transaction

{

public:

SDLTransaction (QGAMES::Screen\* s)

: QGAMES::Screen::Transaction (s)

{ }

protected:

/\*\* @see parent. \*/

virtual void openFirst ();

virtual void closeLast ();

virtual void openOne () { } // Nothing special opening one transaction

virtual void closeOne () { }

};

22: La clase Transaction para SDL

Cuyo código asociado es:

//---

void SDLScreen::SDLTransaction::openFirst ()

{

SDL\_SetRenderDrawColor (((SDLScreen\*) \_screen) -> \_render, 0, 0, 0, 255);

SDL\_RenderClear (((SDLScreen\*) \_screen) -> \_render);

}

//---

void SDLScreen::SDLTransaction::closeLast ()

{

SDL\_RenderPresent (((SDLScreen\*) \_screen) -> \_render);

}

23: Los métodos openFirst () y closeFirst () de Transaction para SDL

Que, como se puede observar, encapsulan funciones propias de SDL.

Por tanto, siempre que definamos un nuevo wrapper para un sistema gráfico extendiendo la clase *Screen*, debemos también extender la clase *Transaction* definida dentro de él.

En la clase *Screen* existe un método que crea un objeto de tipo *Transaction*. El adecuado para el sistema gráfico sobre el que estemos trabajando y que liga finalmente *Screen* con *Transaction*:

virtual Transaction\* createTransaction () = 0;

Que en el caso de *SDLScreen* es:

virtual QGAMES::Screen::Transaction\* createTransaction ()

{ return (new SDLTransaction (this)); }

## ¿Y cómo encajan estas tres piezas?

Volvamos a nuestra clase inicial: *Game* y dentro de ella, al método *exec*.

Allí podemos observar cómo se usa la clase *Screen*, mirando a la parte central de este método:

…

// Actualize the image in the screens...

for (QGAMES::Screens::const\_iterator i = \_screens.begin ();

i != \_screens.end (); i++)

{

QGAMES::Screen\* scr = (\*i).second;

…

scr -> openTransaction ();

…

scr -> closeTransaction ();

}

…

exec, recorre todas las pantallas definidas para el juego, abre transacción sobre cada una de ellas invocando al método ***openTransaction ()*** (que, a su vez, terminará llamando al método *createTransaction ()* de nuestro wrapper *Screen* y al método *openFirst ()* del wrapper *Screen::Transaction*).

Pero, ¿dónde y cuándo se construyen las diferentes pantallas del juego?

Dentro de *Game* se define el método ***createScreens ()*** que devuelve una lista de las pantallas que debe usar el juego (***Screens***). Ese método debe ser sobrecargado (está definido como virtual puro inicialmente en *Game*) para nuestras propias necesidades.

Por ejemplo:

virtual QGAMES::Screens createScreens ()

{ QGAMES::Screens result;

result.insert (QGAMES::Screens::value\_type (\_\_QGAMES\_MAINSCREEN\_\_,

implementation () -> createScreen

(std::string ("Battleship Commander"),

QGAMES::Position::\_cero, \_\_GAMETEST\_SCREENWIDTH\_\_, \_\_GAMETEST\_SCREENHEIGHT\_\_, 100, 100)));

return (result); }

24: Un ejemplo del método createScreen en la clase Game

En el que, como se puede observar, terminamos invocando al método definido en el wrapper *GameImplementation* para crear una ventana específica. Lógico, es ahí donde deben estar como dijimos, todas las funciones específicas de cada sistema gráfico utilizado.

En el caso de SDL, ese método se define como:

// ---

QGAMES::Screen\* SDLGame::createScreen (const std::string& t, const QGAMES::Position& pos,

int w, int h, int pX, int pY)

{

// This is just only because a bug in SDL version

// They say they will solve it in further versions...we will see!!

float sX = (float) std::atof (game () -> parameter (\_\_GAME\_XSCALE\_\_).c\_str ());

float sY = (float) std::atof (game () -> parameter (\_\_GAME\_YSCALE\_\_).c\_str ());

return (new SDLScreen (t, pos, w, h, pX, pY, sX, sY));

}

Que es quien instancia de verdad el wrapper sobre *Screen* para, en este caso, el framework SDL.

Too much?

## Resumen hasta el momento

Repasemos como ir enlazando todos los conceptos definidos:

Para crear un juego:

* Extender la clase *Game*, definiendo el método *createScreens ()*.
* El constructor de nuestro juego deberá usar el wrapper *GameImplementation* del sistema que toque.

Para crear un nuevo wrapper gráfico:

* Definir un wrapper de *Screen* sobre ese sistema gráfico.
* Definir un wrapper de *Transaction* sobre ese sistema gráfico.
* Definir el método del warpper *Screen* *createTransaction ()* para instancia un objeto del wrapper *Transaction* definido en el punto anterior.
* Definir el wrapper *GameImplementation* y extender el método *createScreen ()* para crear el objeto del wrapper *Screen* que hemos creado.

Recordar que ya existen wrappers creados para SDL y para OGRE que, deberían ser suficientes.

¿Mejor?

# La representación de la realidad. La clase *Element*

Quizás la clase core de un juego. Todos los elementos que interactúan en él, terminan de una manera u otra derivando de esta clase.

Se define en *element.hpp*:

/\*\* \ingroup Game \*/

/\*\* \ingroup Structure \*/

/\*@{\*/

/\*\*

\* @file

\* File: element.hpp \n

\* Framework: Commty Game Library (CGL) \n

\* Author: Ignacio Cea Forniés (Community Networks) \n

\* Creation Date: 01/12/2014 \n

\* Description: Defines an element. \n

\* The element is the basic brick of any game. \n

\* Versions: 1.0 Initial

\*/

#ifndef \_\_QGAMES\_ELEMENT\_\_

#define \_\_QGAMES\_ELEMENT\_\_

#include <Common/notifierobserver.hpp>

#include <Common/position.hpp>

#include <Common/buoy.hpp>

#include <Common/counter.hpp>

#include <Common/onoffswitch.hpp>

#include <Common/openvalue.hpp>

#include <Common/\_exengine.hpp>

#include <vector>

namespace QGAMES

{

class Screen;

class Game;

class Element;

typedef std::vector <Element\*> Elements;

/\*\*

\* An Element is something that is part of a game. \n

\* It has always a life-cycle consisting on: \n

\* 1.- To be initialized, \n

\* 2.- to do something in every game loop, \n

\* 3.- and to be finalized. \n

\* Additionaly, and element can be drawn in a screen. \n

\* QGAMES library is disegned to manage the elements but they can also be managed externally (external engine). \n

\* Elements include references to counters, switches and buoys, really

\* usefull to switch off and on things in every loop or to manage conditions. \n

\* Review them at their appropiate include file. \n

\* The runtime status of an element can be represented by a set of values. \n

\* The element can also be initialized (runtime) from a set of these values.

\*/

class Element : public Notifier, public Observer

{

public:

/\*\* What the potential external engine could need. \n

All data is stored in a map of strings. \*/

class ExternalData

{

public:

ExternalData ()

: \_cache ()

{ }

virtual ~ExternalData ()

{ } // Just in case...

std::string getFromCache (const std::string& vN) const

{ std::map <std::string, std::string>::const\_iterator i;

return (((i = \_cache.find (vN)) == \_cache.end ()) ? std::string (\_\_NULL\_STRING\_\_) : (\*i).second); }

bool storeInCache (const std::string& vN, const std::string& vV);

void clearCache ()

{ \_cache.clear (); }

private:

std::map <std::string, std::string> \_cache;

};

/\*\* Any element has to have an id. \n

It is highly recommended that this id is unique in the system. \*/

Element (int id)

: \_id (id),

\_isVisible (true), // Visible by default, but this attribute is no used here...

\_buoys (),

\_counters (NULL),

\_onOffSwitches (NULL),

\_externalEngine (NULL), // An external engine...if any (By default it doesn't exit)

\_externalData (NULL) // Makes no sense if \_externalEngine is NULL

{ }

/\*\* The destructor is virtual, just in case. The element is the owner of its buoys. \*/

virtual ~Element ();

/\*\* Every element is represented by an id.

For further reasons, the id should be unique in the system per element. \*/

int id () const

{ return (\_id); }

// Managing the runtime configuration...

/\*\* In an element the default runtimeValues will be the status

of the switchs and the counters if any (setOfOpenValues),

and whether the element is or not visible (openValue). \*/

virtual SetOfOpenValues runtimeValues () const;

virtual void initializeRuntimeValuesFrom (const SetOfOpenValues& cfg);

/\*\* To know whether the elements will or not be visible on the screen. \n

The attribute has to be used later, when a draw method is finally implemented.

But as this capacity is something common to all drawable elements, the attribute is defined here. \*/

bool isVisible () const

{ return (\_isVisible); }

virtual void setVisible (bool v)

{ \_isVisible = v; }

// To manage the external engine

/\*\* To set the external engine, if any. \*/

\_ExEngine\* externalEngine ()

{ return (\_externalEngine); }

void setExternalEngine (\_ExEngine\* egn)

{ \_externalEngine = egn; }

/\*\* Set additional data, if any. \*/

ExternalData\* externalData ()

{ return (\_externalData); }

void setExternalData (ExternalData\* eD)

{ delete (\_externalData); \_externalData = eD; }

/\*\* To recover the game where the element is in. \n

If this method is invoke and no game is in place in debug mode, the system stops. \*/

Game\* game ();

const Game\* game () const;

/\*\* This method should be used when some initialization for the object is needed. \n

By default the method calls only the initialization for an external engine if any. \*/

virtual void initialize ()

{ if (\_externalEngine) \_externalEngine -> whenInitialize (this); }

/\*\* This method is called in every game loop. \n

By default the buoys are treating, and the external engine invoke, if any.\*/

virtual void inEveryLoop ();

/\*\* An element can also be drawn in an screen. \n

The method admits a position as a reference.

By default that position is noPosition what should mean that the internal element's position

has to be used instead. That internal position should then be calculated. \n

They way it is done depends on the specific type of element.

The external engine is also invoke here. \*/

virtual void drawOn (Screen\* s, const Position& p = Position::\_noPoint)

{ if (\_externalEngine) \_externalEngine -> whenDrawOn (this, s, p); }

/\*\* Sometimes a finalization of the element is needed.

e.g: If the element reserves some memory, this could be the right place to delete

it, before destroying the object. \n

The external engine is invoke also here. \*/

virtual void finalize ()

{ if (\_externalEngine) \_externalEngine -> whenFinalize (this); }

/\*\* The elements can observe other parts of the game. \n

If something happens on those elements being observed,

an event will be received here. The default treatment is to do nothing,

but the user can overload it to do whatever he wishes. \n

The processEvent method is called sync during the notification of an event.

The notify method (@see notifier) iterates over the list of observers executing

this method in each observer (Element is also an observer).

IMPORTANT NOTE: So if this method changed the list of observers, a crash would be generated. \n

Use the buoys to "note" the event and then process it layer during the inEveryLoop method (if any). \*/

virtual void processEvent (const Event& e)

{ }

protected:

// Implementation

// The buoys...

/\*\* To manage the buoys.

@see processEvent method details for further information. \*/

void addBuoy (int id, Buoy\* b); // Adds, the buoy doesn't have to exist before...

void addBuoys (Buoys b); // Adds buoys to the list...

void setBuoys (Buoys b); // Deletes the previous list and a new one is set

void setBuoy (int id, Buoy\* b); // Replace one buoy, the buoy has to exist...

void activeBuoy (int id, bool a);

Buoy\* buoy (int id);

const Buoy\* buoy (int id) const;

const Buoys& buoys () const

{ return (\_buoys); }

Buoys& buoys ()

{ return (\_buoys); }

/\*\* To delete all buoys created. \n

It is something internal only. Never used outside. \*/

void deleteBuoys ();

// Implementation to access just flow control elements...

// Counters...

/\*\* To set the counters.

If anything exists before it is deleted. \*/

void setCounters (Counters\* c);

/\*\* To get a reference to the counters.

If they are NULL, the method createCounters (@see below) is invoked. \*/

Counters\* counters ();

const Counters\* counters () const

{ return (\_counters); } // This one doesn't create the counters object (it has to exist before)

/\*\* To create the counters first time. \n

By default, and to be compatible with all code already created. it returns NULL. \n

@see macro \_\_IMPLEMENTCOUNTERS\_\_ defined in counter.hpp. \*/

virtual Counters\* createCounters ()

{ return (NULL); }

/\*\* To access a counter. \n

Due to counters are to control internal states only, it is a protected method. \*/

Counter\* counter (int nC)

{ return (counters () -> counter (nC)); }

const Counter\* counter (int nC) const

{ return (counters () -> counter (nC)); }

/\*\* To restart all counters. \*/

void reStartAllCounters ()

{ counters () -> reStartAll (); }

// Switches

/\*\* To set the on-off switches. \*/

void setOnOffSwitches (OnOffSwitches\* s);

/\*\* To get a reference to the switches.

If they are NULL, the method createOnOffSwitches (@see below) is invoked. \*/

OnOffSwitches\* onOffSwitches ();

const OnOffSwitches\* onOffSwitches () const

{ return (\_onOffSwitches); } // This ones doesn't create the switches object (it has to exist before)

/\*\* To create the on-off switches first time. \n

By default, and to be compatible with all code already created. it returns NULL.

@see macro \_\_IMPLEMENTSWITCHES\_\_ defined in onoffswitch.hpp. \*/

virtual OnOffSwitches\* createOnOffSwitches ()

{ return (NULL); }

/\*\* To access an on-off switch. \n

Due to switches are to control internal states only, it is a protected method. \*/

OnOffSwitch\* onOffSwitch (int nC)

{ return (onOffSwitches () -> onOffSwitch (nC)); }

const OnOffSwitch\* onOffSwitch (int nC) const

{ return (onOffSwitches () -> onOffSwitch (nC)); }

/\*\* To restart all on-off switches. \*/

void reStartAllOnOffSwitches ()

{ onOffSwitches () -> reStartAll (); }

protected:

int \_id;

bool \_isVisible;

// Very internal...

Buoys \_buoys;

Counters\* \_counters;

OnOffSwitches\* \_onOffSwitches;

\_ExEngine\* \_externalEngine;

ExternalData\* \_externalData;

};

}

#endif

// End of the file

/\*@}\*/

No es una clase muy extensa. Iremos desgranándola poco a poco. En este primer vistazo nos ocuparemos de entender cuatro de sus métodos: ***initialize ()***, ***inEveryLoop ()***, ***drawOn ()*** y ***finalize ()***. Estos cuatro métodos marcan el ciclo de vida de cualquier *Element* en un juego y todos ellos son invocados automáticamente de una manera u otra desde el método *exec* de la clase *Game*. Por tanto, ¡definiendo nuestros propios *Element* y sobrecargando los métodos anteriores para las necesidades de cada juego conseguiremos cualquier efecto!

Esto es la magia de *QGAMES*. Podemos centrarnos solo en la lógica del juego y dejar que el framework haga el resto.

Para entender mejor cómo funcionan cada uno de esos métodos recordemos que un juego (en *QGAMES*) es en realidad un inmenso bucle. Bucle que se ejecuta en el método *exec* de la clase *Game*. Ese bucle está ligado en algunas de sus partes al número de frames por segundo que se quieren representar y en otras no.

## El ciclo de vida de Element

El ciclo de vida de cualquier Element, viene determinado por los siguientes métodos:

* ***initialize ()***, debe contener las instrucciones para inicializar por primera vez un objeto. En este método deben colocarse las instrucciones necesarias para, por ejemplo, establecer el aspecto y forma inicial de un elemento, su posición inicial en el espacio, etc.
* ***inEveryLoop ()***, debe contener instrucciones que deban ejecutarse sobre el elemento en cada bucle del juego, esté este ligado o no a los frames por segundo de ejecución. Por tanto, aquí habrá cosas como actualizar estados, pero no posiciones, pues aquellas deben ligarse a la velocidad del juego. Ya veremos donde ubicar este tipo de cosas.
* ***drawOn ()***, debe contener las instrucciones para dibujar el elemento sobre la pantalla (que se recibe como parámetro).
* ***finalize ()***, que debe, por último, contener las instrucciones para finalizar el objeto, una vez ha sido utilizado. Por ejemplo, salvar su estado, borrar la memoria temporal que pueda haber utilizado y cosas semejantes.

Todo *Element* debe tener un id que lo represente en el juego. Debemos procurar que dicho id sea único en el juego.

En *Element* se definen también un par de métodos ***isVisible ()*** y ***setVisible ()*** para conocer y actualizar respectivamente una variable interna llamada \_isVisible. Esta variable no es por defecto utilizada por *Element*, y debe ser interpretada por cada implementación de *Element*. La razón es que lo que hacer en un elemento relacionado con esta variable no es siempre lo mismo. En algunos casos, por ejemplo, se puede querer actualizar la posición del objeto en el juego pero en otro no, o incluso en algunos juegos cuando un elemento es “invisible” se puede querer dibujarlo de una manera y cuando es visible de otra.

Si ahora volvemos a nuestra principal clase Facade (*Game*) y al método *exec*, veremos que, de alguna manera los nombres de los métodos anteriores aparecen también reflejados, aunque aún no entendamos como encajan. Nuevamente observemos la parte central dl bucle en el que aparecen los nombres de estos métodos. Volveremos más adelante sobre ello.

# La comunicación interna entre los elementos de un juego. El modelo *Notifier* – *Observer*

Habrás observado quizás que *Element* hereda de dos clases llamadas *Notifier* y *Observer*. Entendamos algo más sobre esas clases, muy importantes a la hora de construir un juego bien estructurado y fácil de mantener.

Imagine que dentro de nuestro juego disponemos de dos *Element*, uno que representa nuestro personaje central y otro que representa el mundo en el que se mueve. Cualquier suceso dentro del escenario puede tener consecuencias en nuestro personaje y al revés.

En programación orientada a objetos para poder implementar ese tipo de consecuencias, necesitaríamos que cada uno de los *Element* guardara una referencia al otro, para así transmitirse, ante cambio en uno, órdenes al otro. Imagine un juego en el que haya decenas de objetos o quizás cientos. ¿Tendrían que guardar todos los *Element* referencias a todos los otros y gestionar la interacción entre unos y otros ante cambios? Inmanejable.

Gracias a Gamma, Helm, Johnson y Vlissides[[9]](#footnote-9), disponemos de algo llamado **Patrones de Diseño**. *QGAMES* usa el patrón *Observer – Notifier* para solucionar este problema.

Un ***Notifier*** es un elemento capaz de emitir eventos, de la clase ***Event,*** a diferentes interesados (suscriptores), los ***Observer***, quienes tomarán acciones en función de ellos.

## La clase *Event*

Es una clase muy sencilla, que se define en el fichero *event.hpp*:

/\*\* \ingroup Game \*/

/\*\* \ingroup Foundation \*/

/\*@{\*/

/\*\*

\* @file

\* File: event.hpp \n

\* Framework: Commty Game Library (CGL) \n

\* Author: Ignacio Cea Forniés (Community Networks) \n

\* Creation Date: 01/12/2014 \n

\* Description: Defines a class to represent an event. \n

\* The events are part of the things notified by notifiers. \n

\* Versions: 1.0 Initial

\*/

#ifndef \_\_QGAMES\_EVENT\_\_

#define \_\_QGAMES\_EVENT\_\_

#include <Common/openvalue.hpp>

#include <list>

namespace QGAMES

{

class Event;

typedef std::list <Event> Events;

/\*\*

\* All notifications must be done using this class.

\* The event is not the owner of the data received. \n

\* Don't forget to delete the data (if necessary) when the

\* event has been processed. \n

\* The event can also have a set of parameters (@see SetOfOpenValues). By default they are not needed...

\* The event is the element transmitied among notifiers and observers.

\*/

class Event

{

public:

Event (int c, void\* d, const SetOfOpenValues& v = SetOfOpenValues ())

: \_code (c), \_data (d), \_values (v)

{ }

virtual ~Event ()

{ }

int code () const

{ return (\_code); }

void\* data () const

{ return (\_data); }

const SetOfOpenValues& values () const

{ return (\_values); }

bool operator == (const Event& evnt)

{ return (\_code == evnt.\_code && \_data == evnt.\_data && \_values == evnt.\_values); }

bool operator != (const Event& evnt)

{ return (\_code != evnt.\_code || \_data != evnt.\_data || \_values != evnt.\_values); }

public:

static Event \_noEvent;

private:

int \_code;

void\* \_data;

SetOfOpenValues \_values;

};

/\*\* Define a no event. \*/

class NoEvent : public Event

{

public:

NoEvent ()

: Event (-1, NULL)

{ }

};

}

#endif

// End of the file

/\*@}\*/

25: La clase Event

Un *Event* no es más que una clase que aglutina cierta información:

* Un **id**, identificador del evento.
* Un **puntero void**, a un objeto de datos cuyo contenido debe ser interpretado por quien lo recibe. Puede ser nulo.
* Y un objeto del tipo ***SetOfOpenValues****,* que se explicará más adelante para almacenar colecciones de datos que pudieran ser de interés al receptor del evento.

*Event* sobrecarga además los operadores == y ¡= para poderlos comparar entre sí:

## Las clases *Notifier* y *Observer*

Se definen en *notifierobserver.hpp*:

/\*\* \ingroup Game \*/

/\*\* \ingroup Foundation \*/

/\*@{\*/

/\*\*

\* @file

\* File: notifyobserver.hpp \n

\* Framework: Commty Game Library (CGL) \n

\* Author: Ignacio Cea Forniés (Community Networks) \n

\* Creation Date: 01/12/2014 \n

\* Description: Defines both notify and observer elements. \n

\* A notifier is someone that notify events when something happens on it. \n

\* An observer is someone that read an event over a notifier an interprets it. \n

\* Versions: 1.0 Initial

\*/

#ifndef \_\_QGAMES\_NOTIFIEROBSERVER\_\_

#define \_\_QGAMES\_NOTIFIEROBSERVER\_\_

#include <Common/event.hpp>

#include <list>

namespace QGAMES

{

class Observer;

typedef std::list <Observer\*> Observers;

class Notifier;

typedef std::list <Notifier\*> Notifiers;

class Notifier

{

public:

Notifier ()

: \_observers ()

{ }

/\*\* When a notifier dies it has to warn the observers that

it is not longer valid. \*/

virtual ~Notifier ();

void registerObserver (Observer\* o);

void unRegisterObserver (Observer\* o);

const Observers& observers () const

{ return (\_observers); }

virtual void notify (const Event& e);

virtual void notifyFrom (const Event& e, Notifier\* n);

private:

Observers \_observers;

};

class Observer

{

public:

friend Notifier;

Observer ()

: \_notifiers ()

{ }

/\*\* When an observer dies no notifier can longer notify an event to it. \*/

virtual ~Observer ();

void observe (Notifier\* n);

const Notifiers& notifiers () const

{ return (\_notifiers); }

bool isObserving (Notifier\* n);

void unObserve (Notifier\* n);

void unObserveAll ();

virtual void processEvent (const Event& e) = 0;

/\*\* By default this method doesn't take into account the sender.

This has been done just because back compatibility (written in old versions). \*/

virtual void processEvent (const Event& e, Notifier\* n)

{ processEvent (e); }

private:

/\*\* CRITICAL: This method is used only when the notifier is deleted.

This is why the notifier is declared friend of the observer.

It is like unObserve method but not invoking the unRegisterObserver method in

the notifier back. \*/

void remove (Notifier\* n);

private:

Notifiers \_notifiers;

};

}

#endif

// End of the file

/\*@}\*/

26: Las clases Notifier y Observer

*Notifier* conserva una lista de referencias a objetos tipo *Observer* (*std::list*) y este último a aquellos.

*Observer* define un método llamado ***observe ()*** que recibe como parámetro un puntero no nulo a un *Notifier* y que actualiza las listas de referencias que uno tiene del otro. Es decir, añade en *Notifier* el *Observer* invocado y añade el *Notifier* referenciado sobre el *Observer* invocado.

*Observer* define también el método ***unObserve ()*** que realiza la acción contraria.

Una vez que hemos invocado al método *observe ()* de un *Observer*, este observará cualquier cosa que el *Notifier* le comunique. Es decir, el *Observer* se declara interesado (se suscribe) en cosas comunicadas por un *Notifier*.

Para que el *Notifier* comunique algo a quien pueda interesarle debe invocar al método ***nofity ()***, definido en *Notifier*, pasando como parámetro el evento *Event* a comunicar.

Esto hará que se ejecute un método llamado ***processEvent ()*** definido en *Observer* y que recibe como parámetro ese objeto *Event*.

Por tanto, si un *Element* 1 está interesado en recibir (y procesar) eventos de otro 2:

* Invocar al método *observe ()* en el *Element* 1 pasándole como referencia el *Element* 2.
* El *Element* 2, cuando tenga algo que comunicar, tendrá que invocar al método *notify ()* pasando como referencia el evento a notificar (con los datos que hagan falta).
* El *Element* 1 habrá sobrecargado el método *processEvent ()* para tratar este evento.

*Notifier* define un método adicional: ***notifyFrom ()*** en el que se termina invocando al método ***processEventFrom ()*** en el *Observer*. Este método recibe como parámetros no sólo el evento a procesar sino una referencia de quien lo envía y, por tanto, es algo más completo que el anterior. Por defecto este método invoca a *processEvent ()*, perdiendo la referencia al emisor.

# Contadores y Switches

Un concepto muy abstracto, pero también muy importante en el desarrollo de juegos QGAMES. ¡Me ha costado mucho llegar a él! ¡A ver si soy capaz de exponerlo suficientemente bien!

En un juego son muy habituales dos tareas: Contar y marcar algo como activo o no. Contamos, por ejemplo, un número de bucles determinado antes de cambiar el aspecto de un personaje. Marcamos algo como activo o no activo cuando pensamos por ejemplo en reflejar el estado de una puerta abierta dentro de un escenario concreto.

El código asociado a contar (por bucle) es siempre muy parecido cualquiera que sea el contador: Asociamos el contador a una variable, incrementamos en 1 unidad esa variable por bucle y cuando alcanza un valor determinado, realizamos una acción concreta y, si ha lugar, volvemos a empezar a contar para repetir esa misma acción el mismo número de bucles más tarde, convirtiéndola así en repetitiva.

Aunque parezca mentira esa necesidad se repite muchas veces por elemento del juego. En algunos elementos el número de “contadores” puede ser grande (5, 6…) lo que complejiza el código a construir, su lectura posterior y, por ende, su mantenimiento.

Por otro lado, imaginemos que quisiéramos salvar el estado de uno de esos elementos que incluye contadores y switches. Deberíamos escribir código específico para ello en cada elemento, alejándonos del objetivo de todo framework de centrar al programador en la codificación de la lógica del juego. Volveremos a esto más adelante.

Por todo ello se han creado las clases ***Counter*** y ***OnOffSwitch***.

## La clase *Counter*

*Counter* se define en *counter.hpp*:

/\*\* The counter. \*/

class Counter

{

public:

/\*\*

\* The counter is built from:

\* @param id The name of the counter.

\* @param lV the value limit.

\* @param iV The initial value. By default it is 0.

\* @param u Whether the counter goes up or down

\* if it is going up lV couldn't be less than the iV, and the opposite if going up!

\* By default all counters go up.

\* @param rI Does the counter have to be reinitialized automatically?

\* By default it has to.

\*/

Counter (int id, int lV, int iV = 0, bool u = true, bool rI = true);

virtual ~Counter ()

{ }

/\*\* To get the data as a set of openvalues. \*/

virtual SetOfOpenValues asSetOfOpenValues () const;

virtual void fromSetOfOpenValues (const SetOfOpenValues& oV);

int id () const

{ return (\_id); }

virtual void initialize ()

{ \_count = \_initialValue; }

/\*\* Initialize the counter with other data than the ones passed at construction time. \*/

virtual void initialize (int lV, int iV, bool u = true, bool rI = true);

/\*\* This method "adds one" to the counter and if it reaches the limits then returns true.

Otherwise it'll return false. \n

If the counter has been declared as "to be restarted", it will start again back. \*/

bool isEnd ();

/\*\* To change the value. \n

This is not a normal method to be used.

The new value can not be out of the limit, otherwise crashes. \*/

void setValue (int nV);

// Accesing attributes

/\*\* To know the limit of the counter. \*/

int limitValue () const

{ return (\_limitValue); }

/\*\* To know the initial value. \*/

int initialValue () const

{ return (\_initialValue); }

/\*\* To know whether the counter is going up or not. \*/

bool countingUp () const

{ return (\_countingUp); }

/\*\* To know whether the counter reinit automatically or not. \*/

bool autoReinitialize () const

{ return (\_autoReinitialize); }

/\*\* To know the value of the counter at this specific moment. \*/

int value () const

{ return (\_count); }

protected:

int \_id;

int \_limitValue;

int \_initialValue;

bool \_countingUp;

bool \_autoReinitialize;

// Implementation

int \_count;

};

27: La clase Counter, definida en counter.hpp

El constructor de *Counter*, recibe los siguientes parámetros:

Counter (int id, int lV, int iV = 0, bool u = true, bool rI = true);

* Un i**dentificador**, único en el elemento en el que esté definido.
* **Valor límite** del contador.
* **Valor inicial** del contador, por defecto 0.
* Un indicador de si el contador debe **contar hacia arriba o hacia abajo**. Por defecto, cuenta hacia arriba, pero con el valor contrario y alternando los valores inicial y final, podemos conseguir un contador que vaya hacia atrás.
* Un indicador de si el contador debe o no **comenzar de nuevo** cuando alcance el valor máximo.

Los métodos en los que nos centraremos ahora y también los más importantes son:

* ***initialize ()***: Restaura el contador a su valor inicial. Existe también una versión de initialize () en la que se pueden volver a definir los valores que se establecieron en su construcción inicial.
* ***setValue ()***: Para colocar un contador a un valor determinado. Ese valor debe estar entre los valores inicial y final.
* ***isEnd ()***: El método más importante para manejar un contador.

Al invocarlo suceden dos cosas: De un lado el contador se incrementa en 1 (o se decrementas en 1 si fue definido para contar hacia atrás) y de otro, devuelve true si alcanzó el límite. Si ese límite se alcanzó y el contador se definió para recomenzar automáticamente, se reinicializará además automáticamente. Luego este método hace prácticamente todas las tareas que al inicio de este punto le asignábamos a un contador.

* ***value ()***: Nos devuelve el valor actual de un contador, pero sin alterarlo.

Hay algunos otros métodos más para conocer parámetros de la definición de un contador, y dos métodos especiales **asSetOfOpenValues ()** y **fromSetOfOpenValues ()** que empezaremos a comprender su importancia a partir del capítulo siguiente.

## La clase *Counters*

Como se ha adelantado, dentro de un elemento es habitual usar varios contadores.

La clase ***Counters***, definida también en *counter.hpp* se ha creado para aglutinarlos:

/\*\* A group of counters to be managed alltogether. \n

This is what Element class e.g. has as parameter. \n

The class owns the counters. \*/

class Counters

{

public:

/\*\* Makes no sense to pass a list of counters in the constructor.

The method initialize (@see) call createCounters (can't be overloaded). \n

That method has to be overloaded to add (@see method addCounter) all counters needed. \*/

Counters ()

: \_counters ()

{ }

virtual ~Counters ()

{ delCounters (); }

/\*\* To get the data as a set of openvalues. \*/

virtual SetOfOpenValues asSetOfOpenValues () const;

virtual void fromSetOfOpenValues (const SetOfOpenValues& oV);

/\*\* It invokes to the method createCounters. \n

This method is invoked from the method reStartCounters defined in the class Element (@see Element).

So any time the counters are restarted they are created again. \*/

void initialize ();

void reStartAll (); // Just initialize the counters but it doesn't create them again...

// To manage the indivial counters

int numberOfCounters () const

{ return ((int) \_counters.size ()); }

bool existCounter (int nC) const

{ return (nC >= 0 && nC < (int) \_counters.size ()); }

Counter\* counter (int nC) // If nC bigger than the last, crashes...

{ return (\_counters [nC]); }

const Counter\* counter (int nC) const

{ return (\_counters [nC]); }

void addCounter (Counter\* c); // In the right order

void replaceCounter (int nC, Counter\* c); // counter number nC has to exit

// Operators

Counter\* operator [] (int nC)

{ return (\_counters [nC]); }

const Counter\* operator [] (int nC) const

{ return (\_counters [nC]); }

protected:

void delCounters ();

virtual void createCounters () = 0;

protected:

std::vector <Counter\*> \_counters;

};

28: Clase Counters. Definida en counter.hpp

Esta clase es muy sencilla. Es sólo un wrapper sobre un vector (*std::vector*) de objetos (punteros) del tipo *Counter*. Dispone de los siguientes métodos fundamentales:

* ***addCounter ()***, para añadir un contador nuevo.
* ***counter ()***, para retornar un contador determinado.
* ***replaceCounter ()***, para sustituir un contador por otro.

Además, la clase redefine el ***operador []*** para poder acceder a un contador determinado como en los objetos de un vector.

Cada nuevo contador se añade al final del vector por tanto tienen siempre una posición determinada. El método ***addCounter ()*** comprueba (en modo debug) que el identificador del contador coincide con la posición del vector en la que va a añadirse.

La implementación se ha hecho con un *std::vector* y no con un *std::map* como pudiera parecer más intuitivo (y por tanto la anterior comprobación no sería necesaria) para que, en tiempo de ejecución, la velocidad de manejo de los contadores sea la máxima posible.

La clase *Counter* se puede redefinir. De hecho, *QGAMES* incluye un contador especial llamado ***RandomCounter*** que establece de forma aleatoria el límite a contar entre un rango.

Es por esta razón que los contadores que se añaden sobre *Counters* son referencias a objetos y no los objetos en sí mismos. *Counters* es dueño de los contadores que maneja y, por tanto, los destruye cuando él es destruido.

Observará que, en la parte final de la definición de *Counters* se declara el método ***createCounters ()*** como virtual puro. Es decir, la clase *Counters* es abstracta por la existencia de este método y debe ser heredada y definir al menos este método para que pueda ser usada.

Para simplificar esta tarea (ya veremos que muy habitual) se declara la siguiente macro (también en *counter.hpp*):

#define \_\_DECLARECOUNTERS\_\_(A) \

class A : public QGAMES::Counters { \

public: \

A () : QGAMES::Counters () { } \

virtual void createCounters (); \

}

En la que se ayuda a definir una clase A que deriva de *Counters* y define el método ***createCounters ()***.

De otro lado, la macro:

#define \_\_IMPLEMENTCOUNTERS\_\_(A) \

void A::createCounters ()

Nos sirve para simplificar la declaración del cuerpo del método *createCounters ()* de la nueva clase A. Un ejemplo:

// ---

\_\_IMPLEMENTCOUNTERS\_\_ (BATTLESHIP::WaitTimeActionBlock::Counters)

{

addCounter (new QGAMES::Counter (BATTLESHIP::WaitTimeActionBlock::\_COUNTERTIME,

0 /\*\* It will be set later. \*/, 0, true, true));

}

En donde deben añadirse los contadores controlados por *Counters*.

¿Bien?

## La clase *OnOffSwitch*

Se define en el fichero *onoffswitch.hpp*:

class OnOffSwitch

{

public:

/\*\*

\* The on-off switch is built from:

\* @param id The name of the switch.

\* @param iV The initial value true or false. By default it is false.

\*/

OnOffSwitch (int id, bool iV = false);

virtual ~OnOffSwitch ()

{ }

/\*\* To get the data as a set of openvalues. \*/

virtual SetOfOpenValues asSetOfOpenValues () const;

virtual void fromSetOfOpenValues (const SetOfOpenValues& oV);

virtual void initialize ()

{ \_value = \_initialValue; }

/\*\* This method is to know the status of the switch. \*/

bool isOn () const

{ return (\_value); }

/\*\* This method is to set on off the switch. \*/

void set (bool v)

{ \_value = v; }

/\*\* Change the value. \*/

void change ()

{ \_value = !\_value; }

protected:

int \_id;

bool \_initialValue;

// Implementation

bool \_value;

};

29: La clase OnOffSwitch, definida en onoffswitch.hpp

Muy muy sencilla y autoexplicativa.

## La clase *OnOffSwitches*

Al igual que los contadores se manejan en bloque, los switches se agrupan entorno a la clase *OnOffSwitches*, también definida en el fichero *onoffswitch.hpp*:

/\*\* A group of on-off switches to be managed alltogether. \*/

class OnOffSwitches

{

public:

OnOffSwitches ()

: \_onOffSwitches ()

{ }

virtual ~OnOffSwitches ()

{ delOnOffSwitches (); }

/\*\* To get the data as a set of openvalues. \*/

virtual SetOfOpenValues asSetOfOpenValues () const;

virtual void fromSetOfOpenValues (const SetOfOpenValues& oV);

/\*\* It invokes to the method createOnOffSwitches. \n

This method is invoked from the method reStartOnOffSwitches defined in the class Element (@see Element).

So any time the onOffSwitches are restarted they are created again. \*/

void initialize ();

void reStartAll ();

// To manage the individual switches

int numberOfSwitches () const

{ return ((int) \_onOffSwitches.size ()); }

bool existOnOffSwitch (int nS) const

{ return (nS >= 0 && nS < (int) \_onOffSwitches.size ()); }

OnOffSwitch\* onOffSwitch (int nC)

{ return (\_onOffSwitches [nC]); }

const OnOffSwitch\* onOffSwitch (int nC) const

{ return (\_onOffSwitches [nC]); }

void addOnOffSwitch (OnOffSwitch\* s);

OnOffSwitch\* operator [] (int nC)

{ return (\_onOffSwitches [nC]); }

const OnOffSwitch\* operator [] (int nC) const

{ return (\_onOffSwitches [nC]); }

protected:

void delOnOffSwitches ();

virtual void createOnOffSwitches () = 0;

protected:

std::vector <OnOffSwitch\*> \_onOffSwitches;

};

30: La clase OnOffSwitches, definida en onoffswitch.hpp

También autoexplicativa, una vez se ha comprendido *Counters*.

Y de igual manera se definen las macros:

#define \_\_DECLAREONOFFSWITCHES\_\_(A) \

class A : public QGAMES::OnOffSwitches { \

public: \

A () : QGAMES::OnOffSwitches () { } \

virtual void createOnOffSwitches (); \

};

#define \_\_IMPLEMENTONOFFSWITCHES\_\_(A) \

void A::createOnOffSwitches ()

# Agregadores de datos. Las clases *OpenValue* y *SetOfOpenValues*

Otra estructura muy importante en *QGAMES*. Utilizada a lo largo de todo el framework.

Otra necesidad muy habitual dentro de *QGAMES* es la de manejar diversos tipos de datos (enteros, flotantes, booleanos, strings, etc.) de una forma abstracta.

Por esa razón existe la clase ***OpenValue***.

## La clase *OpenValue*

Se define en *openvalue.hpp*:

class OpenValue

{

public:

typedef enum

{

\_BOOL = 0,

\_INT = 1,

\_STRING = 2,

\_BDATA = 3

} Type;

OpenValue ()

: \_type (\_BOOL), // Bool by default...

\_boolValue (false), \_intValue (\_\_MININT\_\_), \_strValue (\_\_NULL\_STRING\_\_), \_bdataValue (\_\_MINBDATA\_\_)

{ }

OpenValue (Type t, const std::string& v);

OpenValue (bool v)

: \_type (\_BOOL), \_boolValue (v),

\_intValue (\_\_MININT\_\_), \_strValue (\_\_NULL\_STRING\_\_), \_bdataValue (\_\_MINBDATA\_\_)

{ }

OpenValue (int v)

: \_type (\_INT), \_intValue (v),

\_boolValue (false), \_strValue (\_\_NULL\_STRING\_\_), \_bdataValue (\_\_MINBDATA\_\_)

{ }

OpenValue (const std::string& v)

: \_type (\_STRING), \_strValue (v),

\_boolValue (false), \_intValue (\_\_MININT\_\_), \_bdataValue (\_\_MINBDATA\_\_)

{ }

OpenValue (bdata v)

: \_type (\_BDATA), \_bdataValue (v),

\_boolValue (false), \_intValue (\_\_MININT\_\_), \_strValue (\_\_NULL\_STRING\_\_)

{ }

// Comparation operators...

/\*\* Two OpenValues are equal if they have the same type and the same value for that type. \*/

bool operator == (const OpenValue& v) const;

bool operator != (const OpenValue& v) const

{ return (!(\*this == v)); }

Type type () const

{ return (\_type); }

bool boolValue () const

{ return (\_boolValue); }

int intValue () const

{ return (\_intValue); }

const std::string& strValue () const

{ return (\_strValue); }

bdata bdataValue () const

{ return (\_bdataValue); }

friend std::ostream& operator << (std::ostream& s, const OpenValue& val);

friend std::istream& operator >> (std::istream& s, OpenValue& val);

private:

Type \_type;

bool \_boolValue;

int \_intValue;

std::string \_strValue;

bdata \_bdataValue;

};

31: La clase OpenValue, definida en openvalue.hpp

Que es muy sencilla de entender. Sólo sirve para agrupar bajo un mismo objeto (*OpenValue*) valores de muy diverso tipo, incluyendo constructores para crear *OpenValue* desde cada una de sus diferentes posibilidades y luego diversos métodos para conocer el tipo guardado y recuperar su valor.

También se sobrecargan los operadores *==* y *¡=* para poder compararlos y los operadores *<<* y *>>* para poderlos salvar y recuperar de un *std::iostream* o un *std::ostream* respectivamente.

Existen templates standard que realizan una función semejante a *OpenValue* a partir de versión 11 de C++, pero ¡la construcción de la librería se inició mucho antes de que el standard 11 hubiera sido aprobado!

Habitualmente en *QGAMES* se necesitan manejan estructuras de este tipo de objetos. Estructuras que, además pueden ser complejas y anidadas. Por esa razón, se define también la clase ***SetOfOpenValues***.

## La clase *SetOfOpenValues*

Se define también en el fichero *openvalue.hpp*:

class SetOfOpenValues

{

public:

SetOfOpenValues (const std::string& n = std::string (\_\_NULL\_STRING\_\_))

: \_name (n),

\_openValues (),

\_nestedOpenValues ()

{ }

SetOfOpenValues (const OpenValues& val, const std::string n = std::string (\_\_NULL\_STRING\_\_))

: \_name (n),

\_openValues (val),

\_nestedOpenValues ()

{ }

SetOfOpenValues (const std::map <std::string, std::string>& prps,

const std::string& n = std::string (\_\_NULL\_STRING\_\_)); // From a list of properties...

/\*\* The name. \*/

const std::string& name () const

{ return (\_name); }

void setName (const std::string& n)

{ \_name = n; }

/\*\* The total. \*/

int numberValues () const;

/\* The ids of the last elements inserted into the structure.

It could be used to create more scalable layers of data.

If there is no data inside the values returned will be -1, meaning nothing. \*/

int lastOpenValueId () const

{ return (\_openValues.empty () ? -1 : (\*\_openValues.rbegin ()).first); }

int lastNestedOpenValueId () const

{ return (\_nestedOpenValues.empty () ? -1 : (\*\_nestedOpenValues.rbegin ()).first); }

const OpenValues& openValues () const

{ return (\_openValues); }

const SetsOfOpenValues& nestedOpenValues () const

{ return (\_nestedOpenValues); }

bool existOpenValue (int nV) const

{ return (\_openValues.find (nV) != \_openValues.end ()); }

bool existSetOfOpenValues (int nS) const

{ return (\_nestedOpenValues.find (nS) != \_nestedOpenValues.end ()); }

// A list with the path to find the open value through out the nested set of open values...

bool existOpenValue (const std::vector <int>& nV) const

{ return (((int) nV.size () == 1)

? existOpenValue (\*nV.begin ())

: existOpenValue (std::vector <int> (nV.begin ()++, nV.end ()))); }

bool existSetOfOpenValues (const std::vector <int>& nV) const

{ return (((int) nV.size () == 1)

? existSetOfOpenValues (\*nV.begin ())

: existSetOfOpenValues (std::vector <int> (nV.begin ()++, nV.end ()))); }

const OpenValue& openValue (int nV) const

{ return ((\*\_openValues.find (nV)).second); }

const SetOfOpenValues& setOfOpenValues (int nS) const

{ return ((\*\_nestedOpenValues.find (nS)).second); }

const OpenValue& openValue (const std::vector <int>& nV) const

{ return (((int) nV.size () == 1)

? openValue (\*nV.begin ()) // To take into account the nested values...

: openValue (std::vector <int> (nV.begin ()++, nV.end ()))); }

const SetOfOpenValues& setOfOpenValues (const std::vector <int>& nV) const

{ return (((int) nV.size () == 1)

? setOfOpenValues (\*nV.begin ()) // To take into account the nested values...

: setOfOpenValues (std::vector <int> (nV.begin ()++, nV.end ()))); }

void addOpenValue (int nV, const OpenValue& v) // If it already exists, the content is replaced!

{ \_openValues [nV] = v; }

void addSetOfOpenValues (int nS, const SetOfOpenValues& s)

{ \_nestedOpenValues [nS] = s; }

void addOpenValue (const std::vector <int>& nV, const OpenValue& v)

{ if ((int) nV.size () == 1) addOpenValue (\*nV.begin (), v);

else addOpenValue (std::vector <int> (nV.begin ()++, nV.end ()), v); }

void addSetOfOpenValues (const std::vector <int>& nV, const SetOfOpenValues& v)

{ if ((int) nV.size () == 1) addSetOfOpenValues (\*nV.begin (), v);

else addSetOfOpenValues (std::vector <int> (nV.begin ()++, nV.end ()), v); }

void removeOpenValue (int nV)

{ \_openValues.erase (\_openValues.find (nV)); }

void removeSetOfOpenValues (int nS)

{ \_nestedOpenValues.erase (\_nestedOpenValues.find (nS)); }

void removeOpenValue (const std::vector <int>& nV)

{ if ((int) nV.size () == 1) removeOpenValue (\*nV.begin ());

else removeOpenValue (std::vector <int> (nV.begin ()++, nV.end ())); }

void removeSetOfOpenValues (const std::vector <int>& nV)

{ if ((int) nV.size () == 1) removeSetOfOpenValues (\*nV.begin ());

else removeSetOfOpenValues (std::vector <int> (nV.begin ()++, nV.end ())); }

bool operator == (const SetOfOpenValues& s) const;

bool operator != (const SetOfOpenValues& s) const

{ return (!(\*this == s)); }

friend std::ostream& operator << (std::ostream& s, const SetOfOpenValues& vls);

friend std::istream& operator >> (std::istream& s, SetOfOpenValues& vls);

private:

std::string \_name;

OpenValues \_openValues;

SetsOfOpenValues \_nestedOpenValues;

};

32: La clase SetOfOpenValues, definida en openvalue.hpp

Un *SetOfOpenValues* es una colección de objetos *OpenValue* (***typedef std::map <int, OpenValue> OpenValues***) y objetos *SetOfOpenValues* (***typedef std::map <int, SetOfOpenValues> SetsOfOpenValues***). Por tanto, con un objeto *SetOfOpenValues* se puede definir un árbol de información. Implemente el patrón *Composite*.

Observe que para definir un objeto *SetOfOpenValues* sólo es necesario un nombre (constructor) que, por defecto es *\_\_NULL\_STRING\_\_*.

Cualquiera de los *OpenValues* o *SetOfOpenValues* añadido dentro de un *SetOfOpenValues* ocupa una posición (*int*) dentro de éste. Esa posición es necesaria para luego recuperarlo.

Dentro de *SetOfOpenValues* hay dos grandes grupos de métodos: Los que permiten acceder e insertar nuevos *OpenValue* y los que hacen lo propio con objetos *SetOfOpenvalue* (los anidados). No hay límite de anidación. No es posible crear recurrencia, habida cuenta de que lo que se añade dentro de *SetOfOpenValues* son objetos y no punteros a éstos.

Así, por ejemplo:

* ***openValues ()***, devuelve una lista de los objetos *OpenValue* (*OpenValues*) que forman parte a primer nivel del objeto *SetOfOpenValues*. *OpenValues* no deja de ser un *std::vector*, por lo que para acceder a cada elemento basta con utilizar las funciones que éste provee.
* ***setOfOpenValues ()***, lo mismo, pero con los objetos *SetOfOpenValues* (*SetsOfOpenValues*).
* ***existsOpenValue ()***, para comprobar si un determinado *OpenValue* existe o no.
* ***existsSetOfOpenValues ()***, lo mismo, pero con los objetos *SetOfOpenValues*.
* ***addOpenValue ()***, añade un nuevo *OpenValue* en la posición pasada como parámetro.
* ***addSetOfOpenValues ()***, lo mismo, pero para un *SetOfOpenValues*.

Y así, sucesivamente.

*SetOfOpenValues* sobrecarga también los operadores *==* y *¡=* para comparar entre si objetos de ese tipo. También sobrecarga los operadores *<<* y *>>* para poder escribir o leer objetos *SetOfOpenValues* de un *std::iostream* o *std::ostream* respectivamente.

Potente, ¿no?

Veamos cómo se aplican todos estos conceptos tan abstractos sobre *Element*.

# Otra vez la clase *Element*. Añadir contadores y switches y salvar y recuperar su estado

Volvamos sobre la clase *Element*.

Al mencionarla por primera vez, y como recordatorio, comentábamos que todas aquellas cosas con las que se interactúan en un juego derivan de alguna manera u otra de la clase Element. Comentábamos que todo *Element* tiene un ciclo de vida y que los diferentes métodos que representan ese ciclo de vida (*initialize ()*, *finalize ()*, *inEveryLoop ()* y *drawOn ()*) terminan siendo invocados desde el bucle principal del juego definido en el método *exec ()* de la clase *Game*.

## Incluyendo Contadores

Si profundizamos un poco más en la clase *Element* (y en el fichero *element.hpp*), observaremos, en la parte de definición *protected*, que se definen métodos específicos para el manejo de contadores:

// Counters...

/\*\* To set the counters.

If anything exists before it is deleted. \*/

void setCounters (Counters\* c);

/\*\* To get a reference to the counters.

If they are NULL, the method createCounters (@see below) is invoked. \*/

Counters\* counters ();

const Counters\* counters () const

{ return (\_counters); } // This one doesn't create the counters object (it has to exist before)

/\*\* To create the counters first time. \n

By default, and to be compatible with all code already created. it returns NULL. \n

@see macro \_\_IMPLEMENTCOUNTERS\_\_ defined in counter.hpp. \*/

virtual Counters\* createCounters ()

{ return (NULL); }

/\*\* To access a counter. \n

Due to counters are to control internal states only, it is a protected method. \*/

Counter\* counter (int nC)

{ return (counters () -> counter (nC)); }

const Counter\* counter (int nC) const

{ return (counters () -> counter (nC)); }

/\*\* To restart all counters. \*/

void reStartAllCounters ()

{ counters () -> reStartAll (); }

De entre todos ellos destaca el método ***counter ()*** que devuelve una referencia (const o no const) al contador cuyo índica se solicita como parámetro.

Se ve que ese método termina invocando a ***counters ()***, que devuelve una instancia al objeto *Counters* manejado por el elemento, creándolo, y por tanto invocando al método ***createCounters ()***, si no se hubiera definido aún (la primera vez):

// ---

QGAMES::Counters\* QGAMES::Element::counters ()

{

if (\_counters == NULL)

setCounters (createCounters ());

return (\_counters);

}

Se invoca, a su vez, al método ***setCounters ()***:

// ---

void QGAMES::Element::setCounters (QGAMES::Counters\* c)

{

assert (c);

delete (\_counters);

c -> initialize ();

\_counters = c;

}

Método que se ha creado para poder cambiar (en caliente, si fuera necesario) el conjunto de contadores (*Counters*) manejado por un *Element* y que, a su vez, invoca al método ***initialize ()*** del elemento *Counters* que se quiere asignar, el cual:

// ---

void QGAMES::Counters::initialize ()

{

if (!\_counters.empty ())

delCounters ();

createCounters ();

reStartAll ();

}

Que invoca al método ***createCounters ()*** definido como virtual puro en *Counters*. Recuerde que era imprescindible crear una clase hija de Counters para redefinir este mismo método para que añadiera (*addCounter ()*) los contadores específicos del grupo.

Por tanto, para añadir contadores a un Element, bastaría con:

* Crear una clase, mejor si cabe anidada dentro de nuestro nuevo *Element*, **extendiendo *Counters***. Esa nueva clase debe añadir (usando el método *addCounter ()*, los counters que vaya a manejar).
* **Redefinir el método *createCounters ()*** del Element para crear una instancia de la clase *Counters* que se ha extendido.

Para todo ello, nos ayudan las macros definidas en *counter.hpp*

El código sería, el fichero .hpp del *Element* que requiriera de contadores. Suponiendo que llamamos *XXX* a la clase que extiende *Counters* para implementar *createCounters ()*:

\_\_DECLARECOUNTERS\_\_ (XXX);

virtual QGAMES::Counters\* createCounters ()

{ return (new XXX); }

33: Macro definida para añadir Counters, definida en counters.hpp

Y el fichero .cpp de esa nueva clase *Element*:

// ---

\_\_IMPLEMENTCOUNTERS\_\_ (XXX)

{

addCounter (…);

…

}

34: Macro definida para añadir la implementación de Counters, definida en counters.hpp

## Incluyendo Switches

De igual manera se razona para los switches y las líneas a añadir serían, en el fichero .hpp:

\_\_DECLAREONOFFSWITCHES\_\_ (XXX);

virtual QGAMES::OnOffSwitches\* createOnOffSwitches ()

{ return (new XXX); }

35: Macro definida para añadir OnOffSwitches, definida en onoffswitches.hpp

Y en el fichero .cpp:

// ---

\_\_IMPLEMENTONOFFSWITCHES\_\_ (XXX)

{

addOnOffSwitch (…);

…

}

36: Macro definida para implementar OnOfffSwitches, definida en onoffswitches.hpp

En el momento en el que se invoque por primera vez al método *counter ()* desde *Element* para acceder a un contador concreto, se dispara todo el mecanismo.

Es habitual que en el método *initialize ()* del *Element* se fuerce la ejecución de todas estas tareas incluyendo un par de líneas:

// ---

void XXX::initialize ()

{

…

reStartAllCounters ();

reStartAllOnOffSwitches ();

…

}

Estas instrucciones crean los contadores, los switches y los inicializan.

## Salvando y recuperando el estado de un *Element*

Dentro de la clase *Element* se definen otros dos métodos muy importantes:

virtual SetOfOpenValues runtimeValues () const;

virtual void initializeRuntimeValuesFrom (const SetOfOpenValues& cfg);

Estos dos métodos deben servir para recuperar el estado interno de un *Element* el primero, y para establecer ese estado a partir de información recibida.

Ambos métodos pueden sobrecargarse posteriormente, porque los parámetros que definen el estado de cada *Element* son diferentes, pero por defecto salvan y recuperan todos los contadores y todos los switches que forman parte de él. Por lo que, cuantas más cosas relacionadas con el estado de un *Element* soportemos sobre *Counters* y *OnOffSwitches*, más automático será la generación y recuperación del estado de un *Element* cualquiera.

Superpotente.

El código por defecto para ambos métodos es:

// ---

QGAMES::SetOfOpenValues QGAMES::Element::runtimeValues () const

{

// There is no parent class...

// This is the first one!

QGAMES::SetOfOpenValues result (std::string (\_\_QGAMES\_RUNTIMEVALUESELMNTTYPE\_\_));

// Whether the element is or not visible...

result.addOpenValue (0, QGAMES::OpenValue (\_isVisible));

// The counters...if any

// The direct access to the variable \_counters is used insted the internal method counters ()

// This last method will try to create the counters,

// failling whether they haven't been defined by the player for this class.

QGAMES::SetOfOpenValues cVV = \_counters

? \_counters -> asSetOfOpenValues ()

: QGAMES::SetOfOpenValues (std::string (\_\_QGAMES\_RUNTIMEVALUESCOUNTERSTYPE\_\_));

result.addSetOfOpenValues (0, cVV);

// Then the switches...

// The direct access to the variable \_onOffSwitches is used insted the internal method onOffSwitches ()

// This last method will try to create the onOffSwitches,

// failling whether they haven't been defined by the player for this class.

QGAMES::SetOfOpenValues sVV = \_onOffSwitches

? \_onOffSwitches -> asSetOfOpenValues ()

: QGAMES::SetOfOpenValues (std::string (\_\_QGAMES\_RUNTIMEVALUESSWITCHESTYPE\_\_));

result.addSetOfOpenValues (1, sVV);

return (result);

}

Y:

// ---

void QGAMES::Element::initializeRuntimeValuesFrom (const QGAMES::SetOfOpenValues& cfg)

{

assert (cfg.name () == std::string (\_\_QGAMES\_RUNTIMEVALUESELMNTTYPE\_\_));

assert (cfg.existOpenValue (0) &&

cfg.existSetOfOpenValues (0) && cfg.existSetOfOpenValues (1));

setVisible (cfg.openValue (0).boolValue ());

// The counters & switches have to exist but also a value for them...

// The counters and switched are not created at construction time usually

// So, the runtime values could be saved when they are not created yet...

QGAMES::SetOfOpenValues sVV = cfg.setOfOpenValues (1);

assert (sVV.name () == std::string (\_\_QGAMES\_RUNTIMEVALUESSWITCHESTYPE\_\_));

if (\_onOffSwitches && sVV.numberValues () != 0)

\_onOffSwitches -> fromSetOfOpenValues (sVV);

QGAMES::SetOfOpenValues cVV = cfg.setOfOpenValues (0);

assert (cVV.name () == std::string (\_\_QGAMES\_RUNTIMEVALUESCOUNTERSTYPE\_\_));

if (\_counters && cVV.numberValues () != 0)

\_counters -> fromSetOfOpenValues (cVV);

}

Dónde se aprecia que, además, se salva el estado de la variable \_***visible*** quehemos mencionado la primera vez que hablamos de *Element*.

Es importante apreciar que la variable *\_visible* ocupa la posición 0 en el árbol de datos y contadores y switches las posiciones 0 y 1 respectivamente de datos anidados.

La introducción de ambos métodos da pie a explicar cómo deben extenderse estos dos métodos en otras clases *Element* (es claro una recomendación si bien cada programador puede adoptar el método que le parezca más oportuno) Debemos seguir siempre el mismo patrón y así el código será mucho más entendible.

Cojamos como ejemplo la clase ***Character*** del módulo ***Dragonwind***. Para mejor comprensión del código deriva de ***DragonArtist***.

El método *runtimeValues ()* es:

// ---

QGAMES::SetOfOpenValues DRAGONWIND::Character::runtimeValues () const

{

QGAMES::SetOfOpenValues result = DRAGONWIND::DragonArtist::runtimeValues ();

int lE = result.lastOpenValueId ();

int lNE = result.lastNestedOpenValueId ();

result.addOpenValue (lE + 1, QGAMES::OpenValue (\_energy));

result.addOpenValue (lE + 2, QGAMES::OpenValue (\_losingEnergy));

result.addSetOfOpenValues (lNE + 1, \_behaviour

? \_behaviour -> asSetOfOpenValues ()

: QGAMES::SetOfOpenValues (std::string (\_\_DRAGONWIND\_RUNTIMEVALUESBEHAVIOURTYPE\_\_)));

return (result);

}

Como se puede ver la estructura de *runtimeValues ()* sigue el siguiente patrón:

1. Invocar al mismo método de la clase de la que hereda y obtener así un valor base de *SetOfValues* a devolver.
2. Obtener los identificadores de los últimos *OpenValue* y *SetOfOpenValues* añadidos, invocando respectivamente a los métodos *lastOpenValueId ()* y *lastNestedOpenValueId ()* del objeto *SetOfOpenValues* recibido como parámetro
3. Añadir sobre el valor obtenido en la primea línea, nuevos *OpenValue* y nuevos *SetOfOpenValues* en las posiciones siguientes a las que se obtuvieron anteriormente.
4. Repetir esto para todos los valores a añadir.
5. Devolver el resultado.

Para el método *initializeRuntimeValuesFrom ()*:

// --

void DRAGONWIND::Character::initializeRuntimeValuesFrom (const QGAMES::SetOfOpenValues& cfg)

{

int lE = cfg.lastOpenValueId ();

int lNE = cfg.lastNestedOpenValueId ();

assert (cfg.existOpenValue (lE) && cfg.existOpenValue (lE - 1) &&

cfg.existSetOfOpenValues (lNE));

QGAMES::SetOfOpenValues cCfg = cfg;

\_losingEnergy = cCfg.openValue (lE).boolValue ();

cCfg.removeOpenValue (lE);

setEnergy (cCfg.openValue (lE - 1).bdataValue ());

cCfg.removeOpenValue (lE - 1);

QGAMES::SetOfOpenValues bOV = cCfg.setOfOpenValues (lNE);

if (\_behaviour && bOV.numberValues () != 0)

\_behaviour -> fromSetOfOpenValues (bOV);

cCfg.removeSetOfOpenValues (lNE);

DRAGONWIND::DragonArtist::initializeRuntimeValuesFrom (cCfg);

}

1. Asegurarse de que el nombre del *SetOfOpenValues* recibido es lo que se espera.
2. Recuperar de lo recibido los identificadores de los últimos *OpenValue* y *SetOfOpenValues* qe formen parte de él.
3. Comprobar que, a partir de esos identificadores, existen en el objeto recibido todos los otros que necesite recuperar el *Element* para reestablecer su estado.
4. Hacer una copia del elemento de información recibido, dado que vamos a irlo manipulando.
5. Cargar sobre las variables internas a recuperar los valores almacenados en ese elemento copiado, eliminado del mismo cada elemento tratado y tomando como referencia de posición los valores recuperados en la línea 2.
6. Invocar a mismo método de la clase de la que hereda, pasándole como parámetro el objeto de información copiado.

Salvar y recuperar el estado de un elemento es tremendamente importante porque salvar y recuperar el estado de un juego completo no es más que salvar y recuperar el estado de todos los elementos que forman parte de él.

# Dando movimiento a los elementos de un juego. La clase *Movement*

Seguimos avanzando.

Los *Element* que representen personajes y objetos del juego suelen estar dotados de movimiento. Como habrá observado, *Element* no dispone de ningún método específico en su definición (*element.hpp*) orientado a la gestión del movimiento.

Introducimos ahora la clase ***Movement***, que se define en el fichero *movemet.hpp*:

class Movement : public Notifier

{

public:

Movement (int id, std::map <int, double> v)

: \_id (id),

\_variables (v)

{ }

virtual ~Movement ()

{ }

int id () const

{ return (\_id); }

/\*\* To clone a movement. \*/

virtual Movement\* clone () const = 0;

/\*\* To get the runtime status of the movement. Nothing is returned by default, but an empty structure \*/

virtual SetOfOpenValues runtimeValues () const

{ return (SetOfOpenValues (std::string (\_\_QGAMES\_RUNTIMEVALUESENTITYMOVTYPE\_\_))); }

/\*\* To set the runtime info of the movement from a set of values.

Nothing is done by default, but checking the structure received could be the right one. \*/

virtual void initializeRuntimeValuesFrom (const SetOfOpenValues& cfg)

{ assert (cfg.name () == std::string (\_\_QGAMES\_RUNTIMEVALUESENTITYMOVTYPE\_\_)); }

double variable (int id) const;

const std::map <int, double>& variables () const

{ return (\_variables); }

void setVariable (int id, double v);

/\*\* To say which should be the direction of the entity after executing the movement. \n

But it could be only an intention. \n

Remember the real position of the entiy should depend on the movement itself (@see orientation method in Entity) \*/

virtual Vector direction () const = 0;

/\*\* The same with the accleration. \*/

virtual Vector acceleration () const = 0;

/\*\* Before init a movement, this method could be used. \n

By default it does nothing. \*/

virtual void initialize ()

{ }

/\*\* To initialize the movement from the information of another movement.

By default it executes the basic initialization. \*/

virtual void initializeFrom (const Movement\* m)

{ initialize (); }

/\*\* The very important method.

How to move an entity? \*/

virtual void move (const Vector& d, const Vector& a, Entity\* e) = 0;

virtual void finalize ()

{ }

protected:

int \_id; /\*\* Which is the identificator for this movement. \*/

std::map <int, double> \_variables; /\*\* Which are the variables (speed, ...) defining it? \*/

};

37: La clase Movement, definida en movement.hpp

Un *Movement* es un objeto responsable de mover un *Entity* en el espacio. Un *Entity*, por otro lado, es una especialización de un *Element* que, entre otras cosas (como veremos más adelante) se puede mover.

Un *Movement* puede mover más de un *Entity* concreto. Por tanto, *Movement* no es un atributo de *Entity*. *Entity* no es dueña de los movimientos que puede llevar a cabo puesto que éstos pueden, en principio, ser utilizados para mover diferentes *Entity*. Aunque en verdad habitualmente no será así.

El método central de *Movement* es:

virtual void move (const Vector& d, const Vector& a, Entity\* e) = 0;

definido como virtual puro. Por tanto, ya sólo por este método, es necesario crear clases que hereden de *Movement* para construir un nuevo movimiento.

***move ()*** recibe tres parámetros:

* La dirección pretendida para el movimiento,
* La aceleración pretendida para el movimiento,
* Una referencia a la entidad a mover, que no puede ser nula.

Un movimiento viene definido por un identificador (*\_id*, el número de movimiento) y una lista de parámetros (*std::map <int, double>*) que definan características del movimiento. Características pueden ser las del escenario en la que aplica en movimiento (gravedad, velocidad del viento, etc.)

Movement (int id, std::map <int, double> v)

: \_id (id),

\_variables (v)

{ }

En *Movement* también podemos encontrar los métodos *runtimeValues ()* e *initializeRuntimeValuesFrom ()* al igual que se definían en *Element* y con el mismo propósito que allí. Y también *initialize ()* y *finalize ()* con el mismo objetivo que allí.

Al igual que el método *move ()* recibe como parámetro la dirección y aceleración de referencia, existen dos métodos (también virtuales), para saber en qué dirección y con qué aceleración se ha movido la *Entity*. Es decir, cual es el resultado del movimiento.

Los movimientos son críticos en cualquier juego y tal como está definido *Movement* es demasiado general. *QGAMES* define tipos específicos de movimiento.

## Movimientos básicos

Sobre *Movement* se definen algunos movimientos básicos, que deben entenderse ya como ligados a un determinado *Entity*:

* ***NoMovement***, es un objeto que no origina movimiento alguno en la *Entity*.
* ***SimpleLinearMovement*** provoca un movimiento lineal sobre una entidad determinado por dos parámetros: El vector pasado como parámetro al método *move ()* y un parámetro propio del movimiento llamado ***\_speed*** (velocidad). Sin aceleración. Esa velocidad se extrae del parámetro 0 que haya recibido el constructor y puede alterarse en caliente con métodos al efecto.
* ***SimpleXYParabolicMovement*** provoca un movimiento en parábola sobre el plano . La forma de esa parábola viene determinada por 4 parámetros: La longitud a recorrer en el eje , la altura a granar en el eje , el tiempo que se tarda en ganar esa máxima altura y un factor acelerador de todo ello. Esos atributos se deducen de los parámetros 0, 1, 2, y 3 respectivamente recibidos en tiempo de construcción.

## Movimientos complejos

Los movimientos anteriores son muy sencillos. Suficientes para juegos básicos, pero no para juegos evolucionados y complejos.

Los movimientos anteriores (tanto *SimpleLineaMovement* como *SimpeXYParabolicMovement*) pueden producir variaciones significativas en la posición de un objeto. Variaciones de varios píxeles de longitud en cada bucle. Si algo impidiera el movimiento del *Entity* entre la posición inicial y final, las clases definidas no valdrían para detectar, por ejemplo, que debe pararse.

Parece pues lógico que se necesita un tipo de movimiento que soporte un movimiento píxel a píxel entre una posición original y otra final, sea cual fuere la curva que convierte uno en otro y sea éste un movimiento en 2 o 3 dimensiones.

Esa clase es ***IncrementalMovement***, también definido en *movement.hpp*:

/\*\* A movement that is made of little increments that are calculated in each iteration.

Those little increments are also executed one by one (but not pixel by pixel take care) and also

verifying first with the entity that the movement is possible (@see canMove method in the class Entity). \n

The internal information of the movement is kept into the internal structure Data

The users can inherit from this class to add particular fields and values to it. \n

The speed is also taking into account. \*/

class IncrementalMovement : public Movement

{

public:

IncrementalMovement (int id, ::std::map <int, double> v);

virtual ~IncrementalMovement ();

/\* To know wether the entity is or not in the first movement. \*/

bool isFirstMovement () const

{ return (\_firstMovement); }

/\*\* To know and set the speed. \*/

int speed () const

{ return (\_speed); }

void setSpeed (int s)

{ \_speed = s; }

/\*\* @see parent. \n

The information added is the whether it is or not the first movement (openValue),

the speed (openValue too), and the internal Data structure (it can be overloaded, a setOfOpenValues). \*/

virtual SetOfOpenValues runtimeValues () const;

virtual void initializeRuntimeValuesFrom (const SetOfOpenValues& cfg);

/\*\* @see parent. \*/

virtual Vector direction () const

{ return (\_data -> \_nextDirection); }

virtual Vector acceleration () const

{ return (\_data -> \_nextAcceleration); }

/\*\* @see parent. \*/

virtual void initialize ();

/\*\* Call prepareMovement first time, the calculateNextMovement after very little increment, then

somethingElseToDoAfterMoving when the movement has finished and if the movement has not been possible

in any direction the method stopDuringMovement is invoked. \*/

virtual void move (const Vector& d, const Vector& a, Entity\* e);

protected:

class Data;

/\*\* To prepare the movement. This method is called once, during the first movement. \n

It must be overloaded by the classes inheriting this one. \*/

virtual void prepareMovement (Data\* d, const Vector& dr, const Vector& a, Entity\* e) = 0;

/\*\* Any time a movement has been done, the next one is calculated using this method. \*/

virtual void calculateNextMovement (Data\* d, const Vector& dr, const Vector& a, Entity\* e) = 0;

/\*\* After any movement, this a "exit" routine to do something else related with the movement. \n

By default there is nothing else to do. \*/

virtual void somethingElseToDoAfterMoving (const Vector& d, const Vector& a, Entity\* e)

{ /\*\* Nothing to do by default. \*/ }

/\*\* Impossible to increment the position during the calculated previous movement. \*/

virtual void stopDuringMovement (const Vector& dr, const Vector& a, Entity\* e)

{ /\*\* Nthing to do by default. \*/ }

Data\* data ()

{ return (\_data); }

const Data\* data () const

{ return (\_data); }

/\*\* To set the data internal element.

A potential other data element can be passed as a reference,

to copy all internal attributes. \*/

void setData (const Data\* o = NULL);

/\*\* To reset the internal data. \*/

void resetData ()

{ delete (\_data); \_data = NULL; }

/\*\* To create the basic element.

By default the basic one is created. \*/

virtual Data\* createData ()

{ return (new Data ()); }

protected:

/\*\* Very internal structure to keep the information of the movement.

Every movement con need different information, so this class can

be inherited. \*/

class Data

{

public:

Data ()

: \_initialPosition (Position::\_cero),

\_nextDirection (Vector::\_cero), \_nextAcceleration (Vector::\_cero) ,

\_incX (INT\_MIN), \_incY (INT\_MIN), \_incZ (\_\_MININT\_\_) // INT\_MIN means not initialized!

{ }

virtual ~Data () // It is virtual becaus it can be overloaded....

{ }

/\*\* Add all parameters (openValues).

It can be overloaded later as new data classes are created inheriting from this one. \*/

virtual SetOfOpenValues asSetOfOpenValues () const;

virtual void fromSetOfOpenValues (const SetOfOpenValues& cfg);

/\*\* To init the data from another data. \*/

virtual void initFrom (const Data\* o);

public:

Position \_initialPosition;

Vector \_nextDirection, \_nextAcceleration; // Direction after finishing the movement...

Vector \_nextOrientation; // Orientation after finishing the movement...

int \_incX, \_incY, \_incZ; // Pixels to advance next time...

};

bool \_firstMovement;

int \_speed;

Data\* \_data;

};

38: Movimientos incrementales pixel a pixel, definido en movement.hpp

Que, como se puede observar tiene ya un aspecto bastante más complejo que el original *Movement*.

El código del método *move ()* se ha redefinido en dos sentidos. En primer lugar, contiene el código necesario para mover píxel a píxel, y siguiendo el algoritmo 3D de Bresenham[[10]](#footnote-10), desde la posición actual de un *Entity* un determinado número de pixeles en los ejes , y . Valores que se almacenan dentro de una clase anidada llamada ***Data*** en las variables ***\_incX***, ***\_incY*** e ***\_incZ***.

En segundo lugar, ese método invoca, dentro de su lógica, a otros métodos (*protected*) que tienen dos misiones:

De un lado calcular los incrementos de *\_incX*, *\_incY* e *\_incZ* dependiendo de la curva que esté tratando de emular el movimiento concreto y que están definidas como virtuales puras. Esos métodos son: ***prepareMovemet ()***, ***calculateNextMovement ()***.

De otro, realizar acciones después de terminar el movimiento o tomar decisiones sobre cuando el movimiento incremental en alguno de los ejes no hay sido posible. Esos métodos son: ***somethingElseToDoAfterMoving ()*** y ***stopDuringMovement ()*** que también se definen como virtuales, pero no puros, aunque su implementación por defecto es no hacer nada (vacía).

Todos ellos reciben los siguientes parámetros:

* ***const Vector&***: Una referencia a la dirección requerida para el movimiento.
* ***const Vector&***: Una referencia a la aceleración requerida para el movimiento.
* ***Entity\****: Una referencia a la entidad sujeta al movimiento.

Los métodos *prepareMovement ()* y *calculateNextMovement ()*, además, reciben:

* ***Data\****: Una referencia al objeto Data (recuerde que es algo privado a la clase y que, por tanto, no pueden ser invocados desde fuera de la clase).

Es decir, los parámetros que estos métodos necesitan son los que ya recibió el método *move ()* desde el que son invocados, más una referencia al objeto Data\* que contiene los incrementos a realizar píxel a píxel para que puedan ser actualizados.

Los métodos se invocan de la siguiente manera desde *move ()* y tienen las siguientes responsabilidades:

* ***prepareMovement ()***: Se invoca únicamente la primera vez que es invocado *move ()* y su responsabilidad principal es la de calcular los primeros valores de *Data*.
* ***calculateNextMovement ()***: Se invoca cada vez que *move ()* es invocado (excepto la primera vez) y su misión básica es la de calcular, en función de la curva que *IncrementalMovement* está tratando de replicar, los siguientes valores para *\_incX*, *\_incY* e *\_incZ*. Se invoca una vez ha finalizado el movimiento planificado en el bucle anterior. Es decir, si, por ejemplo, *prepareMovement ()*, calculo que *\_incX* inicial eran 5 pixels, en la siguiente invocación a *move ()* se recorrerán esos 5 pixeles y luego se invocará al método *calculateNextMovement ()*.
* ***somethingElseToDoAfertMovement ()***: Este método también se invoca al finalizar el movimiento, y se suele utilizar como puerta de escape para realizar otras tareas adicionales asociadas al movimiento, como calcular si hay algo o no de lo que informar.
* ***stopDuringMovement ()***: Este método se invoca cada vez que el método *move ()* mueve un píxel y si se ha detectado que el movimiento en ese píxel no era posible. Profundicemos un poco más en el comportamiento de este método.

El proceso para mover píxel a píxel es un *Entity* (sobrecarga del método *move ()*), una vez calculados los *\_incX*, *\_incY* e *\_incZ* a realizar es el siguiente:

* En primer lugar, se comprueba si el movimiento es posible, invocando al método ***canMove ()*** definido en *Entity* y pasando como parámetro un Vector con el valor en el que se pretende incrementar la posición del *Entity*. Ese método debe hacer las comprobaciones que toquen teniendo en cuenta, entre otras cosas, la propia situación del *Entity* y el contexto en el que se está moviendo y devolver en consecuencia *true* o *false*.
* Si el valor retornado fuera ***false*** se lanzaría el evento (*notify ()*) ***\_\_QGAMES\_MOVEMENTNOTPOSSIBLE\_\_*** y, en segundo lugar, se invocaría al método *stopDuringMovement ()*, pasando como dirección de referencia la que se pretendía ejecutar y como aceleración la recibida como parámetro en *move ()*. En este caso, el movimiento no se ejecutaría.
* Si el valor retornado fuera, por el contrario, ***true***, se actualizaría la posición del objeto.
* En cualquier caso, se termina invocando al método *somethignElseToDoAfertMovement ()* y al método *calculateNextMovement ()*.

En el objeto *Data* también se guardan otros datos además del incremento.

* ***\_initialPosition***: Posición inicial del Entity cuando empieza el movimiento. Este valor se establece automáticamente en el primer movimiento.
* ***\_nextDirection***, ***\_nextAcceleration***: Vector que señala la dirección y aceleración que seguirá el *Entity* en la siguiente ejecución de *move ()*. Estos valores deben ser actualizados (si procede) por el programador en los métodos *prepareMovement ()* y *calculateNextMovement ()*. Por defecto son inicializados a ***Position::\_null***.
* ***\_nextOrientation***: Vector que define la siguiente orientación de *Entity* (puede moverse haca la izquierda, por ejemplo, pero estar mirando hacia la derecha). Este valor también debe actualizarse (si procede) cuando se actualicen los dos anteriores.

*Data*, puede sobrecargase para almacenar otros datos que necesite otro *Movement* que herede de *IncrementalMovement*.

Dentro de *IncrementalMovement* se define un método para instanciar el objeto *Data* (o heredado) que se necesite. Por defecto, crea un *Data*:

virtual Data\* createData ()

{ return (new Data ()); }

Este es quizás el tipo de movimiento más importante que se define en todo *QGAMES* y, seguramente, el más utilizado.

*IncrementalMovement* redefine los métodos ***runtimeValues ()*** y ***initializeRuntimeValuesFrom ()***, para contemplar y recuperar los valores de *Data*. Por esa razón, *Data* define sendos métodos: ***asSetOfOpenValues ()*** y ***fromSetOfOpenValues ()*** que devuelven, el primero, con un *SetOfOpenValues* los datos almacenados en *Data* y almacena, el segundo, en la estructura *Data* los valores recibidos como *SetOfOpenValues*.

## Encadenar movimientos

En un juego es muy habitual que los movimientos se encadenen. Si un *Entity* estuviera corriendo y el jugador quisiera, por ejemplo, saltar, el salto sería seguro diferente que si este se produjera cuando estuviera parado. En el primer caso, tendría seguramente forma de parábola tanto más larga cuando mayor fuera la velocidad que tuviera de partida, mientras que, en el segundo caso, ese salto sería vertical.

Los movimientos, por tanto, han de tener en cuenta situaciones anteriores al inicializarse.

Por ese motivo, en *Movement* se define como virtual el método ***initializeFrom ()*** que recibe como parámetro otro *Movement*. Por defecto, este método invoca *initialize ()*, pero puede sobrecargarse para hacer cosas diferentes en función del movimiento recibido como parámetro y sus circunstancias.

El método *initializeFrom ()* se invoca desde el método que dentro de *Entity* sirve para cambiar de movimiento activo que, a su vez, se invoca desde muchos otros lugares de éste. Lo veremos más adelante.

# Animando los elementos de un juego. *Forms*, *Frames*, *Animations* and *States*

¿Bien hasta el momento?

Empezamos ahora a analizar cómo dar vida a nuestros *Element*. Algo ya hemos empezado a vislumbrar al estudiar la clase *Movement*, pero era solo el principio.

## Formas. La clase *Form* y la clase *Frame*

En el capítulo anterior hemos comenzado a hablar de la clase *Entity* como una clase que hereda de *Element* y que, entre otras cosas, dota a aquella de movimientos.

Un *Entity* también tiene animación. Lo habitual por otro lado, es que el movimiento venga acompañado de animación.

El efecto de animación se consigue habitualmente “engañando” al ojo y mostrando diferentes aspectos de la *Entity* con pequeñas variaciones de forma sobre el anterior y con la suficiente fluidez como para que esos cambios parezcan continuos.

En el mundo 2D y 2.5D es habitual que todos esos posibles aspectos se integren bajo un fichero habitualmente con el formato *.jpg* o similar. En el mundo 3D se consigue algo parecido con ficheros soporte más complejos.

Para una mejor comprensión de *Form* y *Frame*, centrémonos únicamente en el mundo 2D y 2.5D. Al fichero tipo *.jpg* al que nos referíamos anteriormente se le suele llamar sprite sheet.

Un ejemplo sería:



39: Ejemplo sprite sheet

En una sprite sheet, como se puede observar, pueden aparecer muchos diferentes aspectos de un mismo personaje para muchas diferentes situaciones de movimiento: andando hacia la derecha, hacia la izquierda, corriendo, saltando, etc.

Un juego puede simular la realidad usando una proyección ortogonal, diédrica, caballera, cónica, etc. Los sprite sheets se pueden utilizar en cualquiera de estas situaciones, pero habitualmente más para las proyecciones 2D (ortogonal) y 2.5D (diédrica y caballera). En el ejemplo de la ilustración anterior aparece un personaje creado para una proyección ortogonal (se ve siempre de lado).

Cada uno de los diferentes aspectos que puede tener un *Entity* se denomina *Frame*. Varios *Frame* se agrupan bajo un *Form*. En la ilustración anterior encontramos 70 *Frame* incluidos en un sólo *Form*.



40: Ejemplo de sprite sheet

En este otro ejemplo, encontramos 12 *Frame* en un solo *Form* y que utiliza una representación isométrica para los mismos.

Aunque profundizaremos más adelante en ello al tratar en detalle la clase *Entity*, es útil imaginarse cada *Frame* “envuelto” en una caja (paralelepípedo) límite (que en 2D sería un rectángulo).

Para detectar, por ejemplo, si hay o no colisión entre dos *Entity*, podemos calcular si hay o no colisión entre las cajas límite de sus diferentes aspectos actuales. La matemática ligada a detectar intersecciones entre cajas es más sencilla (y rápida de procesar) que la ligada a cálculos píxel a píxel.

Para calcular las coordenadas de esa caja límite es necesario saber, para empezar, el tipo de proyección gráfica que emulan los *Frame* de un *Form*. No es lo mismo calcular las coordenadas de esa caja límite sabiendo que la proyección es ortogonal, que sabiendo que es isométrica.

La clase *Form* puede utilizarse para muchas otras situaciones que iremos detallando. De momento nos ceñiremos a su utilización dentro de un *Entity*, pero no es difícil imaginar que también pudiera, por ejemplo, representar el fondo fijo de una escena, o el logotipo de la compañía que ha desarrollado el juego.

Con todo esto en la cabeza, podemos empezar a introducir la clase ***Frame***, definida en *form.hpp*:

/\*\* A Frame is a part of the Form. \*/

class Frame

{

public:

Frame (Form\* f, int nf)

: \_originalForm (f),

\_numberFrame (nf)

{ assert (f); }

virtual ~Frame ()

{ }

Form\* originalForm ()

{ return (\_originalForm); }

int numberFrame () const

{ return (\_numberFrame); }

/\*\* The frame can be drawn in a screen at an specific position. \n

How to draw a Frame also will depend on the Form implementation. \*/

virtual void drawOn (Screen\* s, const Position& p = Position::\_noPoint, int a = 255) = 0;

protected:

Form\* \_originalForm;

int \_numberFrame;

};

41: La clase Frame, definida en form.hpp

Que, como puede observarse, tiene poca complejidad.

Se construye proporcionando una referencia no nula a la *Form* de la que forma parte y un identificador del número de *Frame*.

La clase define el método virtual puro ***drawOn ()*** que sirve para dibujar el *Frame* en una pantalla. Evidentemente no se dibuja igual un frame que esté definido “estáticamente” a partir de un fichero .jpg que otro que haya que dibujar, por ejemplo, a “mano” trozo por trozo, línea a línea, o triángulo a triángulo. Esa es la razón por la que es virtual puro.

Previo a la definición de Form, se define:

typedef std::vector <Frame\*> Frames;

Para simplificar la definición de listas de *Frames* que, por ejemplo, componen *Form*.

*Form*, se define en consecuencia, y también en *form.hpp*, como:

/\*\*

\* A Form is the graphical representation of a part of the Game (e.g. entity moving). \n

\* A Form can usually be made up of several Frames (e.g. different positions of that entity moving). \n

\* So a form could be some kind of what in 2D games is call the "sprite sheet", but trying even to cover 3D world. \n

\* There are many ways to define a Form.

\* The FormBuilder (@see FormBuilder) is the entity in charge of creating them. \n

\* That definition has 2 important group of attributes to take into account specially when drawing:

\* 1.- The size of every frame, that can be the same for all or one per frame. \n

\* 2.- The offset of every frame, that can be the same for all or one per frame. \n

\* Every form is defined also by a name and an id.

\*/

class Form

{

public:

/\*\* A form is made up of several frames. \n

It is defined by an id, a name, and the number of frames.

The constructor creates "the place" for the frames, but it doesn't instantiate them. \*/

Form (int id, const std::string& n, int nf);

/\*\* When the form is deleted, all its frames are also deleted (if they were created). \*/

virtual ~Form ();

/\*\* When a form-frame is requested first time (@see frame method), it is created an kept into memory

to improve performace in the following acceses. \n

It is possible to create all form's frames at once with this instruction

to improve overall game performance. \n

This method is used e.g. from the form builder (@see FormBuilder), \*/

void preLoad ();

int id () const

{ return (\_id); }

const std::string& name () const

{ return (\_name); }

int numberFrames () const

{ return (\_numberFrames); }

// The "sizes" of the form. These methods are heavily used in the rest of the game

/\*\* The width of the form (the full form) in pixels. \*/

virtual int width () const = 0;

/\*\* The height of the form (the full form) in pixels. \*/

virtual int height () const = 0;

/\*\* The depth of the form (the full form) in pixels. \*/

virtual int depth () const = 0;

// The "sizes" of every frame. These methods are heavily used inthe rest of the game

/\*\* The generic width of a frame of the form in pixels. \n

A generic size for all frames can be defined and also one specific per frame. \*/

virtual int frameWidth () const = 0;

/\*\* The generic height of a frame of the form in pixels. \n

A generic size for all frames can be defined and also one specific per frame. \*/

virtual int frameHeight () const = 0;

/\*\* The generic depth of a frame of the form in pixels. \n

A generic size for all frames can be defined and also one specific per frame. \*/

virtual int frameDepth () const = 0;

/\*\* To know the width of a specific frame. By default the generic one is returned. \*/

virtual int frameWidthForFrame (int nF) const

{ return (frameWidth ()); }

/\*\* To know the height of a specific frame. By default the generic one is returned. \*/

virtual int frameHeightForFrame (int nF) const

{ return (frameHeight ()); }

/\*\* To know the height of a specific frame. By default the generic one is returned. \*/

virtual int frameDepthForFrame (int nF) const

{ return (frameDepth ()); }

/\*\* A vector to know all "sizes" at once. \*/

Vector frameSizeVector () const

{ return (Vector ((bdata) frameWidth (), (bdata) frameHeight (), (bdata) frameDepth ())); }

/\*\* The same but taken into accoun the specific dimensions of a frame. \*/

Vector frameSizeVectorForFrame (int nF) const

{ return (Vector ((bdata) frameWidthForFrame (nF), (bdata)

frameHeightForFrame (nF), (bdata) frameDepthForFrame (nF))); }

// The offset information for frame

/\*\* To know the generic offset in the x axis of a frame definition (if any). \*/

virtual int frameXOffset () const = 0;

/\*\* To know the generic offset in the y axis of a frame definition (if any). \*/

virtual int frameYOffset () const = 0;

/\*\* To know the generic offset in the z axis of a frame definition (if any). \*/

virtual int frameZOffset () const = 0;

/\*\* To know the specific offset in the x axis of a frame definition (if any).

By default the generic one is returned. \*/

virtual int frameXOffsetForFrame (int nF) const

{ return (frameXOffset ()); }

/\*\* To know the specific offset in the y axis of a frame definition (if any).

By default the generic one is returned. \*/

virtual int frameYOffsetForFrame (int nF) const

{ return (frameYOffset ()); }

/\*\* To know the specific offset in the z axis of a frame definition (if any).

By default the generic one is returned. \*/

virtual int frameZOffsetForFrame (int nF) const

{ return (frameZOffset ()); }

/\*\* A vector to know all offsets at once. \*/

Vector frameOffsetVector () const

{ return (Vector ((bdata) frameXOffset (), (bdata) frameYOffset (), (bdata) frameZOffset ())); }

/\*\* Same, but taking into account the specific frame information. \*/

Vector frameOffsetVectorForFrame (int nF) const

{ return (Vector ((bdata) frameXOffsetForFrame (nF),

(bdata) frameYOffsetForFrame (nF), (bdata) frameZOffsetForFrame (nF))); }

// Important places in the form

/\*\* Methods to locate special zones of the form. \n

The way these positions are calculated will depend on how the form is defined. \n

These method are used from many other elements in the game, specially from entities (@see Entity definition). \n

To understand what each method is consider a "cube" with the form inside. \n

This first method is to know the "core" (usually the center) of that hypothetical cube. \*/

virtual Position centerPosition (int nF) const = 0;

/\*\* Someplace (usually The center) in the base of the cube. \*/

virtual Position basePosition (int nF) const = 0;

/\*\* A rectangle rounding the central zone. \*/

virtual Rectangle roundingCentralZone (int nF) const = 0;

/\*\* A top rectangle of the cube. \*/

virtual Rectangle coverZone (int nF) const = 0;

/\*\* The bottom rectangle of the cube. \*/

virtual Rectangle baseZone (int nF) const = 0;

/\*\* A box rounding the full form. \*/

virtual Box roundingBox (int nF) const = 0;

// The "visual dimensions" of the form

/\*\* The "visualLength" in a 2D frame-form representing some also in 2D is obviouly the same than its width,

but would it be also the same when it is representing something is a 2.5D (cavalier perspective e.g)? \*/

virtual int visualLength (int nF) const

{ return (frameWidthForFrame (nF)); }

/\*\* Similar with height \*/

virtual int visualHeight (int nF) const

{ return (frameHeightForFrame (nF)); }

/\*\* Similar with depth \*/

virtual int visualDepth (int nF) const

{ return (frameDepthForFrame (nF)); }

/\*\* To get an specific form's frame. \n

The frame information (after it is created) is allocated in memory to speed up

following accesses to the same frame. \*/

Frame\* frame (int nf);

/\*\* The Form can be drawn in an screen. \n

The frame to draw is mandatory. \*/

virtual void drawOn (Screen\* s, int nf, const Position& p = Position::\_noPoint,

int a = 255);

protected:

/\*\* How to create a Frame from a Form will depend the way that From is implemented. \n

e.g. Under SDL framework will not the same than under OGRE.

One is a 2D engine, and the other ine is a 3D engine. \*/

virtual Frame\* createFrame (int nf) = 0;

protected:

int \_id;

std::string \_name;

int \_numberFrames;

Frames \_frames;

};

42: La clase Form, definida en form.hpp

Ya más compleja, pero aún fácil de entender con las explicaciones aportadas hasta el momento.

Todo *Form* viene determinado por un identificador (***\_id***), un nombre (**\_name**), un número de frames que lo forman (**\_numberFrames**) y la lista de éstos (**\_frames**). El identificador debe ser único por aplicación, aunque es recomendable que también lo sea el nombre.

El constructor de *Form* recibe los tres primeros e inicializa la última variable con el tamaño de *\_numberFrames* y con todos los *Frame* *NULL*.

Cada *Frame* se inicializará la primera vez que se requiera a través del método ***frame ()*** que, como se puede observar, invoca al método ***createFrame ()*** declarado como virtual puro dado que, como hemos comentado anteriormente, es necesario definir un *Frame* específico para nuestros intereses (y también el método *drawOn ()*), atendiendo al tipo de representación del juego en cuestión y su fichero soporte (si hubiera).

El código del método *frame ()* es:

// ---

QGAMES::Frame\* QGAMES::Form::frame (int nf)

{

return ((\_frames [nf] == NULL) ? (\_frames [nf] = createFrame (nf)) : \_frames [nf]);

}

Por lo demás, en *Form* podemos encontrar los siguientes métodos:

* ***frameWidth ()***, ***frameHeight ()*** y ***frameDepth ()***, deben devolver las dimensiones de la caja ficticia que rodea al *Frame*. Estos métodos no reciben el número de *Frame*. En algunas implementaciones de *Form* pudieran hacerse suponiendo que todos los Frame tienen el mismo tamaño. Todos ellos están declarados como virtuales puros.
* ***frameWidthForFrame ()***, ***frameHeightForFrame ()*** y ***frameDepthForFrame ()***, realizan la misma función que los anteriores, pero para un *Frame* concreto. Están definidos como virtuales, pero por defecto invocan a sus homólogos que no requieren el número de *Frame*.
* ***centerPosition ()***, ***basePosition ()***, ***roundingCentralZone ()***, ***coverZone ()***, ***baseZone ()*** y ***roundingBox ()***, deben devolver posiciones y zonas representativas de la caja ficticia que rodea al *Frame*. Así, por ejemplo:
  + *centerPosition ()* debe devolver un objeto Position señalando a lo que podríamos considerar el centro del *Frame*,
  + *basePosition ()* la parte central de “los pies”,
  + *coverZone ()* la tapa de la caja,
  + *baseZone ()* la base de la caja (habitualmente, *basePosition ()* suele devolver el centro de *baseZone ()*) y
  + *roundingCentralZone ()* un rectángulo en la parte más core (habitualmente *centrerPosition ()* es el centro de *roundingCentralZone ()*).
  + Y *roundingBox ()* debe devolver la caja límite completa. Evidentemente debería apoyarse en *coverZone ()* y *baseZone ()* básicamente para ser coherente con aquellos.
* ***visualLength ()***, ***visualHeight ()*** y ***visualDepth ()*** pueden introducir un pequeño matiz sobre *width ()*, *height ()* y *depth ()* que analizamos antes. Deben devolver la apariencia visual de esas dimensiones. Por ejemplo, si el *Frame* representara una proyección isométrica *frameWidthFormFrame ()* devolvería un valor y *visualLength ()* la mitad. Por defecto devuelven los mismos valores que los métodos referidos.
* Otro tipo de métodos son los que nos proporcionan información sobre la *Form* o realizan cosas sobre ellas: Ya hemos analizado los métodos *frame ()* y *createFrame ()*.

Adicionalmente aparece el método ***drawOn ()*** que, aunque también virtual, se define, por defecto de la siguiente manera:

// ---

void QGAMES::Form::drawOn (QGAMES::Screen\* s, int nf, const QGAMES::Position& p, int a)

{

assert (s);

assert (nf < \_numberFrames);

frame (nf) -> drawOn (s, p, a);

}

43: El método drawOn () de Form

Es decir, invocando al método equivalente del *Frame* que se intenta dibujar (y que recordemos estaba declarado como virtual puro pues lo que hacer dependerá de nuestra interpretación de cada *Frame* en el contexto del juego).

## Form y Frame para 2D. Las clases *Sprite2D* y *Sprite2DFrame*

Una vez comprendida la mecánica de funcionamiento de *Form* y *Frame*, vamos a profundizar en cómo estás clases se implementan para el mundo 2D y 2.5D en forma de Sprites (podría hacerse, por qué no, de otra manera).

Esa representación visual Sprite se soporta habitualmente en un fichero de datos tipo *.jpg*,*.png*,*.gif*,*.tiff*, o similar.

Esos ficheros suelen estar optimizados para no desperdiciar ningún píxel en blanco y suelen incluir, por tanto, información sobre la localización exacta y tamaño de la parte útil de cada diferente aspecto incluido en él.

A veces, esa información se proporciona en un fichero adicional (habitualmente tipo *.xml*) cuyo formato concreto depende del motor de juegos para el que haya sido creado.

QGAMES define un formato *.xml* propio para los ficheros de definición de un Spriteque analizaremos más adelante (al estudiar los diferentes *Builder* del sistema) y que también puede ser generado por herramientas estándar de mercado.

Por tanto, para entender dónde y cómo está definido un *Frame* dentro de un Sprite (descrito por un *.jpg* por ejemplo) se necesita conocer, como mínimo: la localización del píxel inicial de éste (coordenadas), el ancho y el alto del dibujo (en pixeles) y, de alguna manera, cuantos píxeles se han eliminado en el proceso de optimización comentado. Esto se traduce en un total de 8 parámetros como mínimo.

La clase ***Sprite2DFrame*** deriva de *Frame*, y también está definida en *form.hpp* de la siguiente manera:

/\*\* The frame of a Sprite2D.

The information received as parameter is not owned by this object. \*/

class Sprite2DFrame : public Frame

{

public:

Sprite2DFrame (Sprite2D\* s, int nf, Sprite2DFrameData\* fD)

: Frame (s, nf),

\_data (fD) // It is not owned by the frame but by the form (see Sprite2DForm destructor...

{ assert (fD); } // Can't be null...'

/\*\* @see parent. \*/

virtual void drawOn (Screen\* s, const Position& p = Position::\_noPoint, int a = 255) override;

protected:

Sprite2DFrameData\* \_data;

};

44: La clase Sprite2DFrame, definida en form.hpp

En la que aparece una referencia a la clase ***Sprite2DFrameData***, definida de la siguiente manera, también *form.hpp*:

/\*\* Class to hold the info about a frame in a Sprite 2D. \*/

class Sprite2DFrameData

{

public:

/\*\* To hold how the visual measures are represented in the frame.

Their values will depend on the type of the projection of the frame. \*/

struct VisualData

{

VisualData ()

: \_Vw (-1), \_Vd (-1), \_Vh (-1), \_Rx (-1)

{ } // Meaning not valid...

VisualData (int vw, int vd, int vh, int rx)

: \_Vw (vw), \_Vd (vd), \_Vh (vh), \_Rx (rx)

{ assert (\_Vw >= 0 && \_Vd >= 0 && \_Vh >= 0 && \_Rx >= 0); } // All them positive

VisualData (const VisualData& vD) = default;

VisualData& operator = (const VisualData& vD) = default;

int Vw () const

{ return (\_Vw); }

int Vd () const

{ return (\_Vd); }

int Vh () const

{ return (\_Vh); }

int Rx () const

{ return (\_Rx); }

private:

int \_Vw, \_Vd, \_Vh, \_Rx; // Will depend on the type of projection...

// The data is directly accesed...

};

/\*\* A structure to keep the main positions and zones of a frame.

Their values will also depend on the type of the projection of the frame. \*/

struct MainRectanglesAndPositions

{

MainRectanglesAndPositions ()

: \_centerPosition (Position::\_noPoint),

\_basePosition (Position::\_noPoint),

\_roundingCentralZone (Rectangle::\_noRectangle),

\_baseZone (Rectangle::\_noRectangle),

\_coverZone (Rectangle::\_noRectangle),

\_roundingBox (Box::\_noBox)

{ } // Meaning not valid

MainRectanglesAndPositions (const Position& cP, const Position& bP,

const Rectangle& rZ, const Rectangle& bZ,const Rectangle& cZ, const Box& rB)

: \_centerPosition (cP),

\_basePosition (bP),

\_roundingCentralZone (rZ),

\_baseZone (bZ),

\_coverZone (cZ),

\_roundingBox (rB)

{ }

MainRectanglesAndPositions (const MainRectanglesAndPositions& mR) = default;

MainRectanglesAndPositions& operator = (const MainRectanglesAndPositions& mR) = default;

const Position& centerPosition () const

{ return (\_centerPosition); }

const Position& basePosition () const

{ return (\_basePosition); }

const Rectangle& roundingCentralZone () const

{ return (\_roundingCentralZone); }

const Rectangle& baseZone () const

{ return (\_baseZone); }

const Rectangle& coverZone () const

{ return (\_coverZone); }

const Box& roundingBox () const

{ return (\_roundingBox); }

private:

Position \_centerPosition;

Position \_basePosition;

Rectangle \_roundingCentralZone;

Rectangle \_baseZone;

Rectangle \_coverZone;

Box \_roundingBox;

};

Sprite2DFrameData () = delete; // Not needed at all...

Sprite2DFrameData (int w, int h, int xP, int yP, int dW, int dH, int xO, int yO, int aT)

: \_width (w), \_height (h),

\_xPosition (xP), \_yPosition (yP),

\_definitionWidth (dW), \_definitionHeight (dH),

\_xOffset (xO), \_yOffset (yO),

\_adjustType (aT),

\_visualData (), \_mainRectanglesAndPositions (), \_Pr (Position::\_noPoint)

// The default values are not valid so,

// In the classes inheriting this one, visual data and main recatangles and positions (and in that order)

// have to be calculated before being used! Otherwise it won't work properly

// And then the value of \_Pr need also to be adjusted!

{

/\*\* In this way:

\_visualData = calculateVisualData ();

\_mainRectanglesAndPositions = calculateMainRectanglesAndPositions ();

\_Pr = Position ((bdata) (\_xOffset + Rx ()), (bdata) \_yOffset, (bdata) 0);

\*/

}

Sprite2DFrameData (const Sprite2DFrameData& fD) = default;

Sprite2DFrameData& operator = (const Sprite2DFrameData& fD) = default;

/\*\* To know the type of projection of the frame.

It has to be overloaded in other classes inheriting from this one. \*/

virtual int projectionType () const = 0;

// Info about the frame

int width () const

{ return (\_width); }

int height () const

{ return (\_height); }

int xPosition () const

{ return (\_xPosition); }

int yPosition () const

{ return (\_yPosition); }

int definitionWidth () const

{ return (\_definitionWidth); }

int definitionHeight () const

{ return (\_definitionHeight); }

int xOffset () const

{ return (\_xOffset); }

int yOffset () const

{ return (\_yOffset); }

int adjustType () const

{ return (\_adjustType); }

// The visual appearance of the frame...

int Vw () const

{ return (\_visualData.Vw ()); }

int Vd () const

{ return (\_visualData.Vd ()); }

int Vh () const

{ return (\_visualData.Vh ()); }

int Rx () const

{ return (\_visualData.Rx ()); }

// Main position within the frame...

const Position& centerPosition () const

{ return (\_mainRectanglesAndPositions.centerPosition ()); }

const Position& basePosition () const

{ return (\_mainRectanglesAndPositions.basePosition ()); }

const Rectangle& roundingCentralZone () const

{ return (\_mainRectanglesAndPositions.roundingCentralZone ()); }

const Rectangle& coverZone () const

{ return (\_mainRectanglesAndPositions.coverZone ()); }

const Rectangle& baseZone () const

{ return (\_mainRectanglesAndPositions.baseZone ()); }

const Box& roundingBox () const

{ return (\_mainRectanglesAndPositions.roundingBox ()); }

protected:

/\*\* It has to be overloaded attenging the type of frame.

That's it: It not the same when the projection is orthogonal, siometric, or cavalier. \*/

virtual VisualData calculateVisualData () const = 0;

virtual MainRectanglesAndPositions calculateMainRectanglesAndPositions () const = 0;

// Implementation

// This point represents the up left corner of the graphical visualization of the frame

const Position& Pr () const

{ return (\_Pr); }

protected:

int \_width, \_height;

int \_xPosition, \_yPosition;

int \_definitionWidth, \_definitionHeight;

int \_xOffset, \_yOffset;

int \_adjustType; // Type of adjustment, used in some type of projections...

// Implementation

VisualData \_visualData;

MainRectanglesAndPositions \_mainRectanglesAndPositions;

Position \_Pr;

};

45: La clase Sprite2DFrameData, definida en form.hpp

Como puede observarse en esta clase se definen los 8 parámetros que enunciábamos como fundamentales para definir la posición de un *Frame* en un Sprite:

***\_xPosition***, ***\_yPosition***, ***\_width***, ***\_height***, ***\_definitionWidth***, ***\_definitionHeight***, ***\_xOffset***, ***\_yOffset***. Los 4 primeros tienen que ver con la localización de la información sobre el *Frame* dentro fichero que define el Sprite, mientras que los 4 últimos se relacionan con la información sobre los pixeles recortados para optimizar el espacio. No nos ocuparemos, por el momento del parámetro ***\_adjustType***.

Estos 8 parámetros (+1) se suministran al constructor que, a partir de ellos, invoca a dos métodos llamados ***calculateVisualData ()*** y ***calculateMainRectanglesAndPositions ()*** que se deben ocupar de calcular los puntos y zonas importantes de cada *Frame* tal y como comentamos al principio de este capítulo. Esos métodos devuelven sendos objetos de las clases ***VisualData*** y ***MainRectanglesAndPositions***. Esas clases son simples aglutinadores de la información relevante de un *Frame*. La forma de calcular los valores que guardan dependerá, por tanto, del tipo de proyección reflejado en el *Frame* como también veíamos al principio.

El resto de los métodos de *Sprite2DFrame* simplemente devuelven el dato que le requieren que, o bien fue suministrado como parte del constructor, o bien fue calculado en este tras invocar a los dos métodos referidos en el párrafo anterior.

Esos dos métodos, *calculateVisualData ()* y *calculateMainRectanglesaAndPositions ()*, se definen en *Sprite2DFrameData* como virtuales puros.

De la clase *Sprite2DFrame*, heredan otras, tantas como sean necesarias, para implementar, acorde con cada tipo de proyección soportada, ambos métodos.

Por ejemplo: ***Sprite2DOrtogonalFrameData*** se define como (también en *form.hpp*):

/\*\* Data for an orthogonal frame. \*/

class Sprite2DOrthogonalFrameData : public Sprite2DFrameData

{

public:

Sprite2DOrthogonalFrameData (int w, int h, int xP, int yP, int dW, int dH, int xO, int yO, int aT /\*\* Not used \*/);

/\*\* @see parent. \*/

virtual int projectionType () const override final

{ return (0); } // Meaning ORTHOGONAL

protected:

/\*\* @see parent.

the adjustType is not taken into account. \*/

virtual VisualData calculateVisualData () const final override

{ return (VisualData (width (), 0 /\*\* Plain\*/, height (),

0 /\*\* Rx from the corner \*/)); }

virtual MainRectanglesAndPositions calculateMainRectanglesAndPositions () const final override;

};

46: La clase Sprite2DOrtogonalFrameData, definida en form.hpp

Y en dónde el método ***calculateMainRectanglesAndPositions ()*** se implementa a su vez como:

// ---

QGAMES::Sprite2DFrameData::MainRectanglesAndPositions

QGAMES::Sprite2DOrthogonalFrameData::calculateMainRectanglesAndPositions () const

{

QGAMES::Position PrDown = Pr () + QGAMES::Vector (\_\_BD 0, \_\_BD Vh (), \_\_BD 0);

QGAMES::Rectangle baseZone (PrDown, PrDown + QGAMES::Vector (\_\_BD Vw (), \_\_BD 0, \_\_BD 0), QGAMES::Vector::\_yNormal);

return (

QGAMES::Sprite2DFrameData::MainRectanglesAndPositions

(QGAMES::Position (Pr () +

QGAMES::Vector (\_\_BD (Vw () >> 1), \_\_BD (Vh () >> 1), \_\_BD 0)), // centralPosition

QGAMES::Position (PrDown +

QGAMES::Vector (\_\_BD (Vw () >> 1), \_\_BD 0, \_\_BD 0)), // basePosition

QGAMES::Rectangle (Pr (),

Pr () + QGAMES::Vector (\_\_BD Vw (), \_\_BD Vh (), \_\_BD 0), QGAMES::Vector::\_zNormal), // roundingCentralZone

QGAMES::Rectangle (Pr (),

Pr () + QGAMES::Vector (\_\_BD Vw (), \_\_BD 0, \_\_BD 0), QGAMES::Vector::\_yNormal), // coverZone

baseZone, // baseZone

QGAMES::Box (baseZone, \_\_BD Vh () // roundingBox

)));

}

*Sprite2DFrame* recibe una referencia a un objeto *Sprite2DFrameData* (puntero) y no una copia de él. Esto está hecho así para, precisamente, y apoyándose en las propiedades de herencia y polimorfismo del *C++*, soportar diferentes tipos de proyección usando la misma y común interfaz de *Sprite2DFrame*.

Por tanto, para crear un *Sprite2DFrame* haría falta una instrucción como:

Sprite2DFrame\* f = new Sprite2DFrame (sprite pointer, number of frame, new Sprite2DOrtogonalFrameData (frame width, frame height, frame def xPos, frame def yPos, frame def width, frame def height, frame xOffset, yOffset, adjust))

Si se quisiera crear un *Frame* para un Sprite que representara una proyección ortogonal.

En *form.hpp* se define también la clase ***Sprite2DFrameDataFactory*** que sólo define el método ***createSprite2DFormData ()***, que recibe como parámetro un tipo de proyección y los datos que definen un *Sprite2DFrame*, y es responsable de crear en función del tipo de proyección el objeto *Sprite2DFrameData* adecuado. Volveremos a esta clase más adelante:

/\*\* A class to create the right Sprite2DFrameData from the type of projection only. \*/

class Sprite2DFrameDataFactory

{

public:

virtual ~Sprite2DFrameDataFactory ()

{ } // Just if neccessary later...

virtual Sprite2DFrameData\* createFrameDataForProjection (int pT, /\*\* type of projection. \*/

int w, int h, int xP, int yP, int dW, int dH, int xO, int yO, int aT) const;

};

47: La clase Sprite2DFrameDataFactory, definida en form.hpp

Para crear el conjunto de Frames que deben formar parte de la definición del Sprite se define, en primer lugar, la siguiente estructura:

using ListOfSprite2DFrameData = std::vector <Sprite2DFrameData\*>;

Para describir un Sprite se necesitan, además de la información de todos los *Frame* que lo conforman, algunos datos generales sobre el mismo, como el tamaño en píxeles (ancho y algo) del fichero (por ejemplo. *jpg*) que soporta la definición y la intensidad (*fade*) con la que debe mostrarse.

Supongamos una situación particular, pero por otro lado muy habitual, en la que todos los *Frame* del Sprite tuvieran el mismo tamaño y no hubieran sido optimizados. Para definir cada *Frame*, bastaría por tanto con saber cuántos *Frame* hay, el tamaño de cada uno de ellos y si hay separación o no entre ellos (algunas herramientas permiten definir esto también) y, por supuesto, el tipo de proyección de dichos *Frame* y al ajuste necesario en esa proyección si ha lugar.

Un ***Sprite2D***, es una clase que deriva de *Form* y representa la hoja de Sprite y se define como, en *form.hpp*:

/\*\* An Sprite2D is a plain Form (equivalent to a 2D Sprite Sheet). \n

Usually the definition of the Form wil be in file. \*/

class Sprite2D : public Form

{

public:

Sprite2D (int id, const std::string& n, int nf, const std::string& s, int w, int h, Sprite2DFormData\* fD)

: Form (id, n, nf),

\_source (s),

\_width (w),

\_height (h),

\_formData(fD)

{ assert (\_formData != NULL); }

virtual ~Sprite2D () // It is the owner...

{ delete (\_formData); }

/\*\* @see parent. \*/

virtual int width () const override

{ return (formData () -> width ()); }

virtual int height () const override

{ return (formData () -> height ()); }

virtual int depth () const override

{ return (0); } // Makes no sense in a plain sprite sheet definition

virtual int frameWidth () const override

{ return (formData () -> standardFrameWidth ()); }

virtual int frameWidthForFrame (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> width ()); }

virtual int frameHeight () const override

{ return (formData () -> standardFrameHeight ()); }

virtual int frameHeightForFrame (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> height ()); }

virtual int frameDepth () const override

{ return (0); } // Makes no sense in a plain sprite sheet definition

virtual int frameDepthForFrame (int nF) const override

{ return (0); } // Makes no sense in a plain sprite sheet definition

virtual int frameXOffset () const override

{ return (0); } // Makes no sense a generical frame offset in this type of definition (COMMTYDEF)

virtual int frameXOffsetForFrame (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> xOffset ()); }

virtual int frameYOffset () const override

{ return (0); } // Makes no sense a generical frame offset in this of definition (COMMTYDEF)

virtual int frameYOffsetForFrame (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> yOffset ()); }

virtual int frameZOffset () const override

{ return (0); } // Makes no sense in a plain sprite sheet definition

virtual int frameZOffsetForFrame (int nF) const override

{ return (0); } // Makes no sense in a plain sprite sheet definition

/\*\* @see parent.

Note that all these method return a copy of the information finally kept. \*/

virtual Position centerPosition (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> centerPosition ()); }

virtual Position basePosition (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> basePosition ()); }

virtual Rectangle roundingCentralZone (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> roundingCentralZone ()); }

virtual Rectangle coverZone (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> coverZone ()); }

virtual Rectangle baseZone (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> baseZone ()); }

virtual Box roundingBox (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> roundingBox ()); }

virtual int visualLength (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> Vw ()); }

virtual int visualHeight (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> Vh ()); }

virtual int visualDepth (int nF) const override

{ return (formData () -> framesInfo ()[nF] -> Vd ()); }

/\*\* The original information. \*/

const Sprite2DFormData\* formData () const

{ return (\_formData); }

Sprite2DFormData\* formData ()

{ return (\_formData); }

const std::string& source () const

{ return (\_source); }

protected:

/\*\* To create the frames of the form. \*/

virtual Frame\* createFrame (int nF) override;

protected:

std::string \_source;

int \_width, \_height;

Sprite2DFormData\* \_formData;

};

48: La clase Sprite2D, definida en form.hpp

En la que se hace referencia a un objeto tipo ***Sprite2DFormData*** definido como (también en form.hpp)

/\*\* Class to hold the information about a 2D form and the frames in it. \*/

class Sprite2DFormData

{

public:

Sprite2DFormData () = delete;

/\*\* All frames are equaled defined in the supporting file, so only glocal info about the form is needed,

and (onbiously) the builder for the frame data that takes into account the projection. \*/

Sprite2DFormData (int nF, int pT, int s, int m, int w, int h, int fw, int fh, int aC, int aT,

Sprite2DFrameDataFactory\* fF);

/\*\* All frames can be different, so not only the general form info is needed but also the detail per frame. \*/

Sprite2DFormData (int nF, int pT, int s, int m, int w, int h, int fw, int fh, int aC, int aT,

const ListOfSprite2DFrameData& fDt);

~Sprite2DFormData (); /\*\* The form data own frames' info \*/

// To know wheter is basic or not...

// That is: info about the frames has or not been defined!

bool isBasic () const

{ return (\_basicFormData); }

// Generic info

int numberFrames () const

{ return (\_numberFrames); }

int projectionType () const

{ return (\_projectionType); }

int spacing () const

{ return (\_spacing); }

int margin () const

{ return (\_margin); }

int width () const

{ return (\_width); }

int height () const

{ return (\_height); }

int standardFrameWidth () const

{ return (\_standardFrameWidth); }

int standardFrameHeight () const

{ return (\_standardFrameHeight); }

int alphaColor () const

{ return (\_alphaColor); }

int adjustType () const

{ return (\_adjustType); }

const ListOfSprite2DFrameData& framesInfo () const

{ return (\_framesInfo); }

protected:

// Generic values...

int \_numberFrames; // The number of frame the form has...

int \_projectionType; // Type of projection used in the frame...

int \_spacing, \_margin; /\*\* The \_spacing between two frame's information.

The \_margin at the right of the form's definition. \*/

int \_width, \_height; /\*\* width and height of the form's information. \*/

int \_standardFrameWidth, \_standardFrameHeight; /\*\* Valid only unless it's been defined per frame. \*/

int \_alphaColor; /\*\* The alpha color used to transparent the sprite information when drawn. \*/

int \_adjustType; // Within the projection type which adjust is required! It is translated to the frames info.

ListOfSprite2DFrameData \_framesInfo; // The information about the frames

// Implementation

bool \_basicFormData; // When all elements are almost similar...

};

49: La clase Sprite2DFormData, definida en form.hpp

En el que se pueden apreciar todos los datos que definen el *Sprite2D* y a los que hacíamos referencia.

Como puede apreciar, hay dos constructores diferentes: Uno que recibe información general del Sprite y la lista detallada de definición de cada *Frame* y, otro, que recibe esa información general, los datos que permiten determinan la posición de cada *Frame* en el Sprite y la referencia a un objeto *Sprite2DFrameSDataFactory* que se utilizará para crear el *Sprite2DFrameData* adecuado para cada *Sprite2DFrame*.

Por tanto, para crear un Sprite2D:

* En el caso de que no todos los Frames sean del mismo tamaño y / o no estén distribuidos en forma de matriz dentro del fichero soporte del *Form*:
  + Crear tantos *Sprite2DFrameData* como *Frame* tenga el Sprite del tipo ***Sprite2DOrtogonalFrameData***, ***Sprite2DIsometricFrameData*** o ***Srpite2DCavalierFrameData*** según sea la proyección allí representada. Utilizando, si se quiere un objeto *Sprite2DFrameDataFactory*.
  + Crear un objeto ***ListOfSprite2DFrameData*** con todos los *Frame* creados.
  + Crear un Objeto *Sprite2DFormData* utilizando el constructor que recibe sólo información general sobre el tamaño del fichero soporte y el objeto *ListOfSprite2DFrameData*.
  + Crear un Objeto tipo *Sprite2D* utilizando los datos anteriores.
* En el caso de que todos los *Frame* son del mismo tamaño y, además, se encuentran distribuidos en forma de matriz dentro del fichero soporte *Form*:
  + Crear un objeto *Sprite2DFormData* utilizando el constructor que recibe información general del tamaño del fichero soporte y una referencia a un objeto *Sprite2DFrameDataFactory* para crear con él los *Frame* necesarios.

Quizás todo esto haya sido demasiado complejo. Iremos viendo como funciona en una implementación concreta. La complejidad es necesaria para implementar la abstracción necesaria que nos permita la implementación de Forms 2D, 2.5D y 3D sin que la lógica del juego posterior se tenga que encargar de tratarlo.

## Animación

Las animaciones están asociadas a las *Entity*. Son una forma de producir cambios en su aspecto. La clase ***Animation*** se define en *Entity* como una clase anidada de ésta (en *entity.hpp*):

class Animation

{

public:

typedef std::map <std::string, std::string> Parameters;

friend Entity;

Animation (int id, const Parameters& prms)

: \_id (id),

\_entity (NULL),

\_parameters (prms)

{ }

virtual ~Animation ()

{ }

/\*\* To clone the animation. \n

It has to be redefined in every class inhereting this one. \*/

virtual Animation\* clone () const = 0;

/\*\* Add all parameters (openValues).

It can be overloaded later as new data classes are created inheriting from this one.

By default the don't do anything. \*/

virtual SetOfOpenValues asSetOfOpenValues () const

{ return (SetOfOpenValues (std::string (\_\_QGAMES\_RUNTIMEVALUESANIMATIONTYPE\_\_))); }

virtual void fromSetOfOpenValues (const SetOfOpenValues& cfg)

{ assert (cfg.name () == std::string (\_\_QGAMES\_RUNTIMEVALUESANIMATIONTYPE\_\_)); }

int id () const

{ return (\_id); }

const Parameters& parameters () const

{ return (\_parameters); }

void setParameters (const Parameters& prms)

{ \_parameters = prms; }

const std::string parameter (const std::string& prmName) const;

bool existParameter (const std::string& prmName) const;

void setParameter (const std::string& prmName, const std::string& val);

void setParameter (const std::string& prmName, int val)

{ setParameter (prmName, std::to\_string (val)); }

void setParameter (const std::string& prmName, float val)

{ setParameter (prmName, std::to\_string (val)); }

void setParameter (const std::string& prmName, bool val)

{ setParameter (prmName, std::to\_string ((int) val)); }

int parameterAsInt (const std::string& prmName) const

{ return (std::stoi (parameter (prmName))); }

float parameterAsFloat (const std::string& prmName) const

{ return (std::stof (parameter (prmName))); }

bool parameterAsBool (const std::string& prmName) const

{ return (std::stoi (parameter (prmName)) != 0); }

/\*\* To indicate whether the animation links automatically with other or not.

By default it returns \_\_MININT\_\_ wich means none. \*/

virtual int link () const

{ return (\_\_MININT\_\_); }

/\*\* To know whether the animation is at the beginning.

It must be redefined in classes inheriting this one. \*/

virtual bool beginning () const = 0;

/\*\* To indicate whether is the end of the animation.

It has to be redefined in every class inheriting this one. \*/

virtual bool end () const = 0;

protected:

/\*\* To set the entity.

This is done by the Entity when a new animation is added. \*/

void setEntity (Entity\* ety)

{ \_entity = ety; }

// The life-cycle of the animation

/\*\* The initialization. \n

It is invoked when the animation is set at Entity level (@see setAnimation method).

By default it does nothing. \*/

virtual void initialize ()

{ }

/\*\* To inherit the animation state from another one previous, if any.

By default nothing is done. \*/

virtual void inheritStateFrom (Animation\* a)

{ }

/\*\* Update the animation per loop. \n

Invoked from updatePositions at Entity level (if an animation is set).

By default it does nothing. \*/

virtual void updatePositions ()

{ }

/\*\* The finalization. When the animation finishes. \n

Invoked from finalize at Entity level (if an animation is set).

By default it does nothing. \*/

virtual void finalize ()

{ }

protected:

int \_id;

Entity\* \_entity;

Parameters \_parameters;

};

Como puede observarse, la clase básica no es más que un identificador y un conjunto de atributos (*std::map <std::string, std::string>*), ambos recibidos en tiempo de construcción. Existen varios métodos para tratar con esos parámetros (devolver sus valores en forma de difeerentes tipos básicos), incluso para poder cambiar su valor en tiempo de ejecución.

Lo que nos importa, en estos momentos son los métodos: ***initialize ()***, ***updatePositions ()***, y ***finalize ()***, todos declarados como virtuales y, por tanto, con capacidad de ser modificados en clases que hereden de ésta.

Todos esos métodos se invocan, ya lo veremos, desde *Entity*, desde cada uno de los homólogos.

Ya se va viendo que la estructura de todos los componentes de la librería tiene muchas similitudes, especialmente en lo relacionado con estos métodos. ¿no?

La clase más importante que hereda de ésta es ***FormAnimation***, definida también en *entity.hpp* como clase anidada dentro de *Entity*:

/\*\* The standard animation based on changes in the form and aspect. \n

The form, the aspect, the speed of the change, etc... are defined in parameters.

The standard entity builder reads them the entity definition file (@see EntityBuilder). \*/

class FormAnimation : public Animation

{

public:

FormAnimation (int id, const Parameters& prms)

: Animation (id, prms),

\_counterAspect (NULL),

\_counterToChange (NULL),

\_beginning (true),

\_end (false)

{ }

virtual ~FormAnimation ();

/\*\* @see parent. \*/

virtual Animation\* clone () const

{ return (new FormAnimation (\_id, \_parameters)); }

/\*\* @see parents. \

Add the information of the counters (setOfOpenValue). \*/

virtual SetOfOpenValues asSetOfOpenValues () const;

virtual void fromSetOfOpenValues (const SetOfOpenValues& cfg);

// To change the internal attributes

int formId () const

{ return (parameterAsInt (\_\_QGAMES\_ENTITYANATTRFORM\_\_)); }

void setFormId (int f)

{ setParameter (\_\_QGAMES\_ENTITYANATTRFORM\_\_, f); }

int initialAspectId () const

{ return (parameterAsInt (\_\_QGAMES\_ENTITYANATTRINITIALASPECT\_\_)); }

void setInitialAspectId (int iA)

{ setParameter (\_\_QGAMES\_ENTITYANATTRINITIALASPECT\_\_, iA); }

int finalAspectId () const

{ return (parameterAsInt (\_\_QGAMES\_ENTITYANATTRFINALASPECT\_\_)); }

void setFinalAspectId (int fA)

{ setParameter (\_\_QGAMES\_ENTITYANATTRFINALASPECT\_\_, fA); }

bool cyclic () const

{ return (parameterAsBool (\_\_QGAMES\_ENTITYANATTRCYCLIC\_\_)); }

void setCyclic (bool c)

{ setParameter (\_\_QGAMES\_ENTITYANATTRCYCLIC\_\_, c); }

int fractionToChange () const

{ return (parameterAsInt (\_\_QGAMES\_ENTITYANATTRFRACTIONTOCHANGE\_\_)); }

void setFractionToChange (int fc)

{ setParameter (\_\_QGAMES\_ENTITYANATTRFRACTIONTOCHANGE\_\_, fc); }

int link () const

{ return (existParameter (\_\_QGAMES\_ENTITYANATTRLINK\_\_)

? parameterAsInt (\_\_QGAMES\_ENTITYANATTRLINK\_\_) : Animation::link ()); } // Optional

void setLink (int lk)

{ setParameter (\_\_QGAMES\_ENTITYANATTRLINK\_\_, lk); }

/\*\* @see parent. \*/

virtual bool beginning () const

{ return (\_beginning); }

virtual bool end () const

{ return (\_end); }

protected:

/\*\* @see parent. \*/

virtual void initialize ();

virtual void inheritStateFrom (Animation\* a); // Take care when using this method...

virtual void updatePositions ();

/\*\* Exit to do something when an specific aspect is reached.

By default, there is nothing to do. \*/

virtual void onAspect (int)

{ }

protected:

// Implementation

Counter\* \_counterAspect;

Counter\* \_counterToChange;

bool \_beginning;

bool \_end;

};

Que es una animación basada en el uso de *Form*.

Esta clase irá alternando entre diferentes *Frame* de un *Form* en cada bucle del juego cuando se invoca el método *updatePositions ()* desde *updatePositions ()* de *Entity*.

Los parámetros que definen a *FormAnimation* son:

\_\_QGAMES\_ENTITYANATTRFORM\_\_: id (int) del Form utilizado para la animación.

\_\_QGAMES\_ENTITYANATTRINITIALASPECT\_\_: número (int) del Frame inicial dentro del Form que representa el frame inicial en la animación.

\_\_QGAMES\_ENTITYANATTRFINALASPECT\_\_ : número (int) del Frame final dentro del Form que representa el frame final de la animación.

\_\_QGAMES\_ENTITYANATTRFRACTIONTOCHANGE\_\_: Número de veces (int) que tiene que pasar por *updatePositions ()* para que cambie de Frame.

\_\_QGAMES\_ENTITYANATTRCYCLIC\_\_: Indica si la animación es o no cíclica (int). Es decir, si cuando se alcanza el Frame final se vuelve a comenzar por el principio. Lo normal es que sí. 1 = YES, 0 = NO.

\_\_QGAMES\_ENTITYANATTRLINK\_\_: Id de la animación (dentro de la misma *Entity*) al que se ligará cuando llegue al final. Evidentemente este parámetro no tiene sentido si la *Animation* es cíclica (\_\_QGAMES\_ENTITYANATTRCYCLIC\_\_ = 1).

Se define también el método *onAspect ()*, que se invoca cada vez que se cambia de aspecto.

# Elementos con animación y movimiento. Las clases Entity y Character

Aunque Entity ya la hemos mencionado de refilón al hablar de animación y movimiento, profudicemos en ella. Character no es más que un Entity con capacidad de “automovimiento”.

## La clase que mejor representa a los personajes de un juego: La clase Entity

La clase Entity se define en el fichero entity.hpp (es la clase superior a Animation)[[11]](#footnote-11).

/\*\*

\* An entity is like an "sprite" shown in the game. So it has forms and aspects in its base. \n

\* The concept "form" represents a set of "aspects", and the "aspect" represents a specific drawing for the entity.

\* So during Entity's life-cycle form and aspect can change.

\* A new method called updatePositions is added to do so and managed by the game. \n

\* The entities can also move. They include a reference to a position and a orientation and methods to control its movement.

\* The updatePositions method should control also how those position and orientation are changed. \n

\* The Entity defines also a set of potential movements, and a reference to the current one that can be change. \n

\* And finally an Element is also defined by physical variables (weight, size,...) an properties (color,...).

\* All those structural attributes commented above are grouped in an internal structure called "data". \n

\* To simplify the way the forms and aspects of an entity changes along the game, the Animation class is defined.

\* An entity can have many Animations defined but just only one active (or none) at the same time. \n

\* When an animation is grouped with a movement a State cames up.

\* It is again a way to simplify how the entities are managed. The user can manage both separately.

\*/

class Entity : public Element

{

public:

typedef std::map <int, bool> Capacities;

typedef std::map <int, double> PhysicalVariables;

typedef std::map <std::string, std::string> Parameters;

… (Animation definition here, but already seen in the previous paragraph)

/\*\* To represent different states of the entity along the game. \n

\* The state is a combination of an animation and a movement by default,

\* but it can be extended to cover more things. \n

\* Again a state can have parameters. By default two parameters can be provided (@see definitions.hpp): \n

\* \_\_QGAMES\_ENTITYSTATEPRMDIRECTION\_\_ with the direction of the movement. \n

\* \_\_QGAMES\_ENTITYSTATEPRMACCELERATION\_\_ with the acceleration. \n

\* These two parameters are read by the default EntityBuilder (@see EntityBuilder).

\* An State only should set the animation and the movement, but it shouldn't actualize them per loop.

\* This is done direclty by the entity in the method updatePositions.

\* So the method updatePositions is to update only the state itself if it is needed (can be overloaded).

\*/

class State

{

public:

typedef std::map <std::string, std::string> Parameters;

friend Entity;

State (int id, int aId, int mId, const Parameters& prms)

: \_id (id), \_animationId (aId), \_movementId (mId),

\_entity (NULL),

\_parameters (prms)

{ }

virtual ~State ()

{ }

/\*\* To clone the state.

The default state is useful, so a clone method is defined and valid. \*/

virtual State\* clone () const

{ return (new State (\_id, \_animationId, \_movementId, \_parameters)); }

/\*\* @see parents. \

Adds the runtim information of the state. By default, nothing is done. \*/

virtual SetOfOpenValues asSetOfOpenValues () const

{ return (SetOfOpenValues (std::string (\_\_QGAMES\_RUNTIMEVALUESSTATETYPE\_\_))); }

virtual void fromSetOfOpenValues (const SetOfOpenValues& cfg)

{ assert (cfg.name () == std::string (\_\_QGAMES\_RUNTIMEVALUESSTATETYPE\_\_)); }

int id () const

{ return (\_id); }

int animationId () const

{ return (\_animationId); }

int movementId () const

{ return (\_movementId); }

const Parameters& parameters () const

{ return (\_parameters); }

protected:

/\*\* To set the entity.

This is done by the Entity when a new state is added. \*/

void setEntity (Entity\* ety)

{ \_entity = ety; }

// The life-cycle

/\*\* The initialization. \n

It is invoked when the state is set at Entity level (@see updatePositions method there). \n

By default the animation and the movement are again set,

unless the parameter passed is true (cA = Continue Animation?). \n

When the animation and the state are set, they are initialized. \*/

virtual void initialize (bool cA = false);

/\*\* The update per loop. \n

Invoked in updatePositions at Entity level (@see updatePositions methid there). \n

By default nothing is done as everything id done in the entity loop. \*/

virtual void updatePositions ()

{ }

/\*\* Finalize the state.

Finalize both the animation and the movement. \*/

virtual void finalize ();

protected:

int \_id;

int \_animationId;

int \_movementId;

Entity\* \_entity;

Parameters \_parameters;

};

typedef std::map <int, State\*> States;

struct Data

{

Data ()

: \_position (),

\_orientation (),

\_capacities (),

\_physics (),

\_parameters (),

\_movements ()

{ }

Data (const Position& pos, const Vector& o,

const Capacities& c, const PhysicalVariables& pV, const Parameters& prms, const Movements& movs)

: \_position (pos),

\_orientation (o),

\_capacities (c),

\_physics (pV),

\_parameters (prms),

\_movements (movs)

{ }

// Operator (structural data has to be equal)

bool operator == (const Data& dt) const;

bool operator != (const Data& dt) const

{ return (!(\*this == dt)); }

/\*\* Entity's position. \*/

Position \_position;

/\*\* The entity is pointing to a place following that vector. \*/

Vector \_orientation;

/\*\* Capacities can be variables to indicate whether the

entity can or not do something, such as flying i.e. \*/

Capacities \_capacities;

/\*\* Physycal variables defining the entity, such as weight, size,... \*/

PhysicalVariables \_physics;

/\*\* Any other parameter interesting in the defintion of the entity (and strings). \*/

Parameters \_parameters;

/\*\* What movements the entity can do. \*/

Movements \_movements;

};

Entity (int eId, const Forms& f = Forms (), const Data& d = Data ());

~Entity (); // The entity owns its animations...

/\*\* In some situations could be interesting to replicate an entity.

e.g: The entity represents a bullet that can be shooted. \n

It could be interesting to replicate the bullet if many of them

can be shooted at the same time, because their behaviour will be quite similar later. \n

Take care with the id you assign. If you are going to register the entity in the game it has to be unique. \n

Additionally you are accountable to delete the entity cloned as

it hasn't been created using the builder. Bearing this in mind. \*/

virtual Entity\* clone () const = 0;

/\*\* To clone the animations managed by the entity. \n

As they are not part of the entity (like the original ones are),

the one invoking this method is accountable to destroy them. \n

Usually this method is invoked from the method clone of the entity (@see above). \*/

Animations cloneAnimations () const;

/\*\* To clone the states managed by the entity. \n

As they are not part of the entity (like the original ones are),

the one invoking this method is accountable to destroy them.

Usually this method is invoked from the method clone of the entity (@see above). \*/

States cloneStates () const;

/\*\* @see parent. \n

The information added is: \n

The state id (openValue), the position (openValue),

and the information comming from the movement (setOfMovements)

if any or empty if there is no movement. \*/

virtual SetOfOpenValues runtimeValues () const;

virtual void initializeRuntimeValuesFrom (const SetOfOpenValues& cfg);

const Data& data () const

{ return (\_data); }

const Position& position () const

{ return (\_data.\_position); }

void setNotifyWhenChangePosition (bool n)

{ \_notifyWhenChangePosition = n; }

/\*\* When the position changed, a notification (\_\_QGAMES\_POSITIONCHANGED\_\_)

is thrown if the switch to do so is on (@see above). \n

This is the default behaviour but it can be changed (only for this attribute). \*/

virtual void setPosition (const Position& p)

{ \_data.\_position = p;

if (\_notifyWhenChangePosition) notify (Event (\_\_QGAMES\_POSITIONCHANGED\_\_, this)); }

/\*\* To set the position of the entity from the position of the base.

This is tipically done in platform games. \*/

void setPositionOnBase (const Position& p) // The position of the base...

{ setPosition (p - (basePosition () - position ())); }

const Vector& orientation () const

{ return (\_data.\_orientation); }

void setOrientation (const Vector& o)

{ \_data.\_orientation = o; }

// To know different parts of the entity

/\*\* Methods to locate special zones of the entity based on the current form and aspect (@see Form). \n

All of them are used in many other parts of the game. e.g. to detect collisions. \n

They can be overloaded. e.g. in a particular game the basePosition could be a corner of the object,

instead the center of the bottom part. \n

Taking a 3D form as example, it could be contained into a box, so: \*/

/\*\* The center of the box. \*/

virtual Position centerPosition () const;

/\*\* The center of bottom rectangle part of the box rounding the object. \*/

virtual Position basePosition () const;

/\*\* A rectangle rounding the central part of the object. \*/

virtual Rectangle roundingFormZone () const;

/\*\* The top rectangle of the box rounding the object. \*/

virtual Rectangle coverZone () const;

/\*\* The bottom rectangle of the box rounding the object. \*/

virtual Rectangle baseZone () const;

/\*\* The nox rounding the object. \*/

virtual Box roundingBox () const;

/\*\* How length (figure out the x axis) is the box rounding the object? \*/

virtual int visualLength () const;

/\*\* How height (figure out the y axis) is the box rounding the object? \*/

virtual int visualHeight () const;

/\*\* How depth (figure out the z axis) is the box rounding the object? \*/

virtual int visualDepth () const;

/\*\* All the current dimensions in a single instruction. \*/

void currentVisualDimensions (int& vL, int& vH, int& vD)

{ vL = visualLength (); vH = visualHeight (); vD = visualDepth (); }

/\*\* Considering all different forms and aspects that the object can adopt, what are the max dimensions?

This is useful to know in some contexts. \*/

void maxVisualDimensions (int& vL, int& vH, int& vD) const;

// Managing the movements

void setMovements (const Movements& m)

{ \_data.\_movements = m; }

const Movements& movements () const

{ return (\_data.\_movements); }

Movement\* currentMovement ()

{ return (\_currentMovement); }

const Movement\* currentMovement () const

{ return (\_currentMovement); }

/\*\* If the parameter is \_\_MININT\_\_, then

there will not be movement in place. Otherwise the movement has to exist. \*/

void setCurrentMovement (int nM);

/\*\* To know whether can or not move depending on their current situation. \n

The direction and the acceleration received as parameters have to be considered as "potential".

By default any movement is always possible. \*/

virtual bool canMove (const Vector& d, const Vector& a)

{ return (true); }

// To manage the forms

void setForms (const Forms& f)

{ \_forms = f; }

Form\* currentForm ()

{ return (\_currentForm); }

const Form\* currentForm () const

{ return (\_currentForm); }

void setCurrentForm (int nf);

int currentAspect () const

{ return (\_currentAspect); }

void setCurrentAspect (int cA)

{ \_currentAspect = cA; }

// To manage the animations

void addAnimation (Animation\* a);

bool existAnimation (int id);

Animation\* animation (int id);

const Animation\* animation (int id) const;

const Animations& animations () const

{ return (\_animations); }

void setAnimations (const Animations& as)

{ deleteAnimations (); \_animations = as; }

void setCurrentAnimation (int nA);

Animation\* currentAnimation ()

{ return (\_currentAnimation); }

const Animation\* currentAnimation () const

{ return (\_currentAnimation); }

// To manage the states

void addState (State\* st);

bool existState (int id);

State\* state (int id);

const State\* state (int id) const;

const States& states () const

{ return (\_states); }

void setStates (const States& sts)

{ deleteStates (); \_states = sts; }

void setCurrentState (int st, bool cA = false); // Continue or nbot the animation? (false by default)

void setCurrentStateDeferred (int st, bool cA = false); // Using the buoy...

State\* currentState ()

{ return (\_currentState); }

const State\* currentState () const

{ return (\_currentState); }

/\*\* To get the hot collision zone of the entity. \n

By default it returns the full size of the entity, taking into account its current frame. \*/

virtual Rectangle collisionZone () const;

/\*\* To determinate whether the entity has or not collisioned with other. \n

By default it returns true when the two collision zones overlap. \*/

virtual bool hasCollisionWith (Entity\* e) const;

/\*\* To determinate whether the entity receives is or not near of another. \n

Two parameters are needed: The entity and the distance considered as "near".

In the default implementation the centers are taken into account. \*/

virtual bool isNearOf (Entity\* e, bdata d) const;

/\*\* The entities acts across the game. \n

Their positions / aspect should be actualized in its life-cycle.

This is the right method to do so. \n

By default the actualization is delegated into a external engine, if any. \*/

virtual void updatePositions ();

/\*\* @see parent.

The current frame will be drawn by default, taking into account its alpha level. \n

If it visible or not is NOT taken into account. \*/

virtual void drawOn (Screen\* s, const Position& p = Position::\_noPoint);

/\*\* To control the alpha level of visualization. \*/

virtual void setAlphaLevel (int aL);

/\*\* The entity represents something "alive" in the game. \n

Then, the entity can collision with other entities.

Following method is created to manage that situation.

A typical behaviour should be to throw an event when the collision is detected.

Who is observing the entity will react. \n

By default no behaviour is defined. \*/

virtual void whenCollisionWith (Entity\* e)

{ }

private:

void deleteAnimations ();

void deleteStates ();

protected:

/\*\*

A default buoy to change the state of the entity. \n

In many circumstances, when the state is changed into the updatePosition method the system can crash.

This is due to many reasons:

e.g: A movement notifies something, the processEvent changes the state and also the movement associated.

The system crashes because the old movement is unobserved and after the treatment of

the event notified the loop dealing with the notifications becomes inconsistent and then crashes. \n

In that circumstances the state has to be changed in the inEveryLoop method, so a Buoy should be used. \n

This buoy is declared and set at construction time of the Entity.

\*/

class ChangeStateBuoy : public Buoy

{

public:

ChangeStateBuoy ()

: Buoy (\_\_QGAMES\_CHANGESTATEBUOY\_\_, (bdata) 0),

\_targetState (0),

\_continueAnimation (false)

{ }

void setTargetState (int tS)

{ \_targetState = tS; }

void setContinueAnimation (bool cA)

{ \_continueAnimation = cA; }

/\*\* @see parent. \*/

virtual void\* treatFor (Element\* e);

private:

int \_targetState;

bool \_continueAnimation; // To continue the animation or not?

};

Forms \_forms; /\*\* The different forms the entity can adopt. \*/

Data \_data; /\*\* The internal attributes defining the entity. \*/

Animations \_animations; /\*\* The list of animations the entity has. \*/

States \_states; /\*\* The list of potential states the entity has. \*/

// Implementation

Form\* \_currentForm;

Movement\* \_currentMovement;

Animation\* \_currentAnimation; /\*\* Null if none. \*/

State\* \_currentState; /\*\* Null if none. \*/

int \_currentAspect;

int \_alphaLevel;

bool \_notifyWhenChangePosition;

// The maximum dimensions of the entity taking into account all its different possible aspects

// They are implementation variables. They are calculated first time and never more...

// They are defined as mutable, so they can be changed in a const method

mutable int \_maxVisualLength, \_maxVisualHeight, \_maxVisualDepth;

};

// To simplify the definition of set of entities...

typedef std::map <int, Entity\*> Entities;

Esta ya una clase realmente potente y compleja. Es la base de prácticamente cualquier elemento de interacción en el juego.

Lo primero que identificamos dentro de la clase es una estructura anidada llamada Data:

struct Data

{

Data ()

: \_position (),

\_orientation (),

\_capacities (),

\_physics (),

\_parameters (),

\_movements ()

{ }

Data (const Position& pos, const Vector& o,

const Capacities& c, const PhysicalVariables& pV, const Parameters& prms, const Movements& movs)

: \_position (pos),

\_orientation (o),

\_capacities (c),

\_physics (pV),

\_parameters (prms),

\_movements (movs)

{ }

// Operator (structural data has to be equal)

bool operator == (const Data& dt) const;

bool operator != (const Data& dt) const

{ return (!(\*this == dt)); }

/\*\* Entity's position. \*/

Position \_position;

/\*\* The entity is pointing to a place following that vector. \*/

Vector \_orientation;

/\*\* Capacities can be variables to indicate whether the

entity can or not do something, such as flying i.e. \*/

Capacities \_capacities;

/\*\* Physycal variables defining the entity, such as weight, size,... \*/

PhysicalVariables \_physics;

/\*\* Any other parameter interesting in the defintion of the entity (and strings). \*/

Parameters \_parameters;

/\*\* What movements the entity can do. \*/

Movements \_movements;

};

Data, aglutina elementos definitorios de cualquier Entity:

* \_position: Posición de Entity en el espacio en cada momento del juego.
* \_orientation: Vector que indica hacia dónde está mirando la entidad. Puede o no recoger la dirección de su movimiento. Lo habitual es que así lo haga, pero podría darse el caso de, por ejemplo, un personaje que camina hacia atrás.
* \_capacities: un mapa (std::map <int, bool> de elementos lógicos que debe utilizarse para identificar si determinadas habilidades o capacidades identificadas por un número (int) están o no habilitadas. Puede utilizarse para discernir el comportamiento de Entity.
* \_physics: un mapa (std::map <int, double> de elementos double que debe utilizarse para identificar características físicas de la entidad, como su peso, su longitud, su altura, etc. Nuevamente estas variables pueden marcar el comportamiento del Entity.
* \_parameters: un mapa (std::map <string, string>) de otros posibles parámetros que puedan describir a nuestra Entidad. Utilizada con propósitos más generales.
* \_movements: un mapa (std::map <int, Movement\*> de todos los posible movimientos que pueda tener nuestra Entity. Recuerda que cada Movement puede estar compartido con otras entidades. Por tanto, ésta no será dueña de esas referencias.

\_position y \_orientation serán actualizados muy a menudo y para y por eso, se proporcionan 4 métodos en Entity:

position () y orientation (): Para conocer la posición y la orientación del Entity en cada momento.

setPosition () y setOrientation: Que permite actualizar la posición y la orientación de la entidad.

En determinadas circunstancias, la actualización de la posición puede implicar realizar acciones en algún otro punto del juego, por lo que, utilizando el patrón Notifier – Observer debería lanzase una notificación (notify ()). Sin embargo, la actualización de posiciones es tan habitual en un juego que levantar una notificación y tratarla cada vez que eso se produce en todas y cada una de las entidades del juego (que pueden ser miles) enlentecería de manera relevante el juego, por lo que esa notificación puede desconectarse o activarse mediante el método setNotifyWhenChangePosition (). Por defecto, las notificaciones están desactivadas.

Todos estos datos son también tratados en runtimeValues () y en initializeRuntimeValuesFrom ()[[12]](#footnote-12).

El segundo elemento que destacar dentro de la definición de Entity, es State:

class State

{

public:

typedef std::map <std::string, std::string> Parameters;

friend Entity;

State (int id, int aId, int mId, const Parameters& prms)

: \_id (id), \_animationId (aId), \_movementId (mId),

\_entity (NULL),

\_parameters (prms)

{ }

virtual ~State ()

{ }

/\*\* To clone the state.

The default state is useful, so a clone method is defined and valid. \*/

virtual State\* clone () const

{ return (new State (\_id, \_animationId, \_movementId, \_parameters)); }

/\*\* @see parents. \

Adds the runtim information of the state. By default, nothing is done. \*/

virtual SetOfOpenValues asSetOfOpenValues () const

{ return (SetOfOpenValues (std::string (\_\_QGAMES\_RUNTIMEVALUESSTATETYPE\_\_))); }

virtual void fromSetOfOpenValues (const SetOfOpenValues& cfg)

{ assert (cfg.name () == std::string (\_\_QGAMES\_RUNTIMEVALUESSTATETYPE\_\_)); }

int id () const

{ return (\_id); }

int animationId () const

{ return (\_animationId); }

int movementId () const

{ return (\_movementId); }

const Parameters& parameters () const

{ return (\_parameters); }

protected:

/\*\* To set the entity.

This is done by the Entity when a new state is added. \*/

void setEntity (Entity\* ety)

{ \_entity = ety; }

// The life-cycle

/\*\* The initialization. \n

It is invoked when the state is set at Entity level (@see updatePositions method there). \n

By default the animation and the movement are again set,

unless the parameter passed is true (cA = Continue Animation?). \n

When the animation and the state are set, they are initialized. \*/

virtual void initialize (bool cA = false);

/\*\* The update per loop. \n

Invoked in updatePositions at Entity level (@see updatePositions methid there). \n

By default nothing is done as everything id done in the entity loop. \*/

virtual void updatePositions ()

{ }

/\*\* Finalize the state.

Finalize both the animation and the movement. \*/

virtual void finalize ();

protected:

int \_id;

int \_animationId;

int \_movementId;

Entity\* \_entity;

Parameters \_parameters;

};

Un State es la agrupación de una animación (Animation) y de un movimiento (Movement). Es decir, es algo que liga la representación visual de nuestra una entidad con el movimiento que realiza. Es decir, es algo que va a representar, por ejemplo, un Ninja corriendo (animación) hacia la derecha (movimiento). O alguien parado (movimiento, o mejor no movimiento) mientras mueve el pie izquierdo en señal de impaciencia (animación).

Los estados pueden tener “vida”, y por tanto en ellos se definen los métodos típicos y ya mencionados del ciclo de vida: initialize (), finalize () y updatePositions ().

Dentro de Entity, se definen y guardan (y Entity es su propietaria) los diferentes estados por lo que puede pasar en un mapa (std::map <int, State\*>). Esos estados son accesibles mediante los métodos:

* state (): Para recuperar un estado determinado.
* states (): Para recuperar la lista de todos los State definidos en el Entity.
* currentState (): Para recuperar una referencia al State que actualmente tiene la Entity.
* setState (): Para activar un estado determinado (su animación y su movimiento).
* setStates (): Para activar de golpe una lista de State.
* setCurrentState (): Para activar un determinado estado en el Entity (por defecto no hay ninguno). Este método, además de requerir el identificado (id) del State a activar, puede recibir (es opcional) un parámetro que indique si la animación, en caso de ser la misma que la que pudiera haber en el estado previo de existir éste, se reinicializa o continua desde el punto en el que el estado anterior la dejara. Esto es útil para facilitar la fluidez visual del juego cuando entre dos estados lo único que cambia es su forma de moverse pero no su animación (por ejemplo, un personaje empieza a caer pero sigue pataleando como si corriera).

Los estados que puede gestionar un Entity, además, se controlan mediante los métodos:

* addState (): Para añadir un nuevo State.
* existState (): Para comprobar si un State está o no definido para el Entity.

Lo normal es manejar estados de un Entity, mediante los anteriores métodos, pero para gestionar la apariencia y el movimiento de un Entity no obligatorio su uso. Puede hacerse de manera independiente. Quizás requiera más coordinación, pero puede hacerse. Los métodos para ellos son muy equivalente a los descritos anteriormente: addAnimation (), existAnimation (), animation (), animations (), setAnimations (), setCurrentAnimation (), y currentAnimation () para el manejo de las animaciones y setMovements (), movements (), currentMovement (), setCurrentMovement () para el manejo de los movimientos.

Cada vez que se invoca al método setCurrentAnimation (), se invoca al método finalize () de la anterior (si es que había) y al método initialize () de la nueva. Lo mismo con el método setCurrentMovement ().

Los movimientos son también objetos (como hemos visto anteriormente) que implementan el patrón Notifer – Observer y que son observables por la Entity que lo está usando (otra razón más para en el fondo usar Movement diferentes por Entity). Cada vez que se fija uno se comienza a observar y se deja de observara el anterior. Además el nuevo movimiento se inicializa siempre partiendo del anterior (initializeFrom ()).

En cualquier caso, recomendamos manejar State y no Animation y Movement por separado. El código y, por tanto, su mantenimiento posterior, se simplifican enormemente por regla general.

Cada Entity está representada por un conjunto de Form. La entity tendrá una de ellas activa en cada momento. Evidentemente esa Form vendrá determinada por la Animation que este activa en cada momento. Existen varios métodos dentro de un Entity para conocer y manejar sus Form:

// To manage the forms

void setForms (const Forms& f)

{ \_forms = f; }

Form\* currentForm ()

{ return (\_currentForm); }

const Form\* currentForm () const

{ return (\_currentForm); }

void setCurrentForm (int nf);

int currentAspect () const

{ return (\_currentAspect); }

void setCurrentAspect (int cA)

{ \_currentAspect = cA; }

Bastante autodescriptivos. setCurrentForm () y setCurrentAspect () serán invocados habitualmente desde setAnimation () o setState (). Aspect en este caso, hace referencia al Frame concreto de la Form activo.

La Entity es dueña de sus Form.

El tercer elemento que destaca de un Entity, son los métodos que permiten investigar su aspecto:

// To know different parts of the entity

/\*\* Methods to locate special zones of the entity based on the current form and aspect (@see Form). \n

All of them are used in many other parts of the game. e.g. to detect collisions. \n

They can be overloaded. e.g. in a particular game the basePosition could be a corner of the object,

instead the center of the bottom part. \n

Taking a 3D form as example, it could be contained into a box, so: \*/

/\*\* The center of the box. \*/

virtual Position centerPosition () const;

/\*\* The center of bottom rectangle part of the box rounding the object. \*/

virtual Position basePosition () const;

/\*\* A rectangle rounding the central part of the object. \*/

virtual Rectangle roundingFormZone () const;

/\*\* The top rectangle of the box rounding the object. \*/

virtual Rectangle coverZone () const;

/\*\* The bottom rectangle of the box rounding the object. \*/

virtual Rectangle baseZone () const;

/\*\* The nox rounding the object. \*/

virtual Box roundingBox () const;

/\*\* How length (figure out the x axis) is the box rounding the object? \*/

virtual int visualLength () const;

/\*\* How height (figure out the y axis) is the box rounding the object? \*/

virtual int visualHeight () const;

/\*\* How depth (figure out the z axis) is the box rounding the object? \*/

virtual int visualDepth () const;

/\*\* All the current dimensions in a single instruction. \*/

void currentVisualDimensions (int& vL, int& vH, int& vD)

{ vL = visualLength (); vH = visualHeight (); vD = visualDepth (); }

/\*\* Considering all different forms and aspects that the object can adopt, what are the max dimensions?

This is useful to know in some contexts. \*/

void maxVisualDimensions (int& vL, int& vH, int& vD) const;

Todos esos métodos se apoyan en los métodos equivalentes de la Form activa en cada instante y son muy utilizados a lo largo del juego para cosas tan básicas como por ejemplo detectar la colisión de la Entity con otras o con otros elementos del juego o para determinar si puede o seguir avanzando en una determinada dirección. Es decir que todas esas tareas se sustentan en la simplificación de que nuestra Entity sea concebida como envuelta en un Paralelogramo. Evidentemente el resultado de estos métodos dependerá del tipo de Form creado y éste, a su vez, de, por ejemplo, la proyección utilizada.

Vamos a describir cada método:

* centerPosicion (): La posición central del paralelogramo.
* basePosition (): La posición central del rectángulo base del paralelogramo que envuelve a Entity.
* roundingFormZone (): Un Rectangle que envuelve la zona central del paralelogramo.
* coverZone (): El rectángulo de la parte superior del paralelogramo.
* baseZone (): Idem con el de la parte inferior.
* roundingBox (): El paralelogramo en sí completo (Box).
* visualLength (), visualHeight (), visualDepth (): La longitud en pixels que tiene el personaje en su forma actual. Evidentemente esa sensación visual dependerá del tipo de proyección.
* currentVisualDimensions (): Nos da los anteriores valores de golpe, depositándolos en variables cuyas referencias se reciben como parámetro.
* maxVisualDimensions (): Igual que el método anterior, pero teniendo en cuenta todas las posibles formas y aspectos que pudiera adoptar en el tiempo la Entity. Nos da una idea de “cuanto grande” puede llegar a ser. No es un buen método para utilizar en un juego si hay demasiada variación en tamaño de formas y aspectos dentro de una misma entidad. Es mejor utilizar en ese caso diferentes entidades.

Observe que todos los métodos anteriores han sido definidos como virtuales y, por tanto, pueden ser redefinidos en clases que hereden de Entity. Esto es así porque quizás para un juego concreto o para un personaje concreto de un juego o para un aspecto o forma concreta de esa Entity en el juego, la base de la Entity (basePosition ()), por ejemplo, no pueda ser el “centro de la base del paralelogramo” y deba programarse exprofeso. No es habitual, sin embargo, y por eso se han definido los comportamientos descritos anteriormente.

Guau, vamos ligando más conceptos.

Y, por último, el cuatro bloque que destaca en una Entity, son sus métodos relacionados con su comportamiento:

* canMove (): recibe como parámetros dos vectores, uno indicando el incremento en la posición que se plantea y el otro la aceleración con la que se pretende llevar a cabo ese movimiento. Este método, que por defecto devuelve siempre true, debe devolver si el movimiento planteado es o no posible para el Entity. El método debe analizar, por tanto, si se van a superar o no los límites máximos de movimiento permitidos para la entidad, si va a chocar o no con algo, etc. Es, por tanto, un método que debe ser redefinido para cada Entity que creemos.
* collisionZone (): Que nos devuelve la zona (Rectangle) de la Entity que debe ser tomada como referencia para determinar colisiones con otras entidades o elementos del juego. Por defecto devuelve roundingFormZone (), pero de igual manera que aquel método podría ser redefinido, este también para acomodarse a las necesidades concretas de cada juego.
* hasCollisionWih (): Recibe como referencia otra Entity y comprueba si sus collsionZone interactúan o no, devolviendo true o false según el caso. Nuevamente puede ser redefinida, pero suele ser habitual que el comportamiento por defecto sea suficiente.

Una clase, sin duda maravillosa. Pero la siguiente lo es sólo un poquito más.

## Los protagonistas: La clase Character

También definida en el fichero entity.hpp, derivando de Entity:

class Character : public Entity

{

public:

Character (int cId, const Forms& f = Forms (), const Data& d = Data ())

: Entity (cId, f, d),

\_stepsControlMonitor (NULL),

\_direction (Vector::\_cero),

\_acceleration (Vector::\_cero)

{ }

~Character () // The characters owns the monitor if any

{ delete (\_stepsControlMonitor); }

/\*\* @see parent. \n

The information added is: \n

the direction (openValue) and the aceleration (openValue) of the movement (just in case),

the monitor id (openValue) (it has to be possible to get it back fromthe the builder),

and the information comming from that monitor (setOfOpenValues).

IMPORTANT: There must exist a steps monitor builder created at game level.

Otherwise initialization will crash!! \*/

virtual SetOfOpenValues runtimeValues () const;

virtual void initializeRuntimeValuesFrom (const SetOfOpenValues& cfg);

/\*\* To get the direction of the movement. \*/

const Vector& direction () const

{ return (\_direction); }

/\*\* To get the acceleration of the movement. \*/

const Vector& acceleration () const

{ return (\_acceleration); }

/\*\* Use this method to change the parameters of the current movement. \*/

virtual void setMove (const Vector& d,

const Vector& a = Vector ((bdata) 0, (bdata) 0, (bdata) 0))

{ \_direction = d; \_acceleration = a; }

// To control the steps of the character

/\*\* To add a step's monitor. \n

The same method can be used to remove it if the parameter is null. \*/

void addControlStepsMonitor (CharacterControlStepsMonitor\* s);

CharacterControlStepsMonitor\* stepsMonitor () // There can be only one at the same time (if any)

{ return (\_stepsControlMonitor); }

/\*\* Moves automatically the entity,

using the direction, acceleration and Movement currently defined. \n

If there is no Movement set nothing happens.

It also actualize the steps monitor. \*/

virtual void updatePositions ();

protected:

CharacterControlStepsMonitor\* \_stepsControlMonitor;

Vector \_direction;

Vector \_acceleration;

};

// To simplify the definition of a set of characters...

typedef std::map <int, Character\*> Characters;

Un Entity actualiza su aspecto de forma automática en cada bucle del juego (Animation), pero no su Movement. Es decir que, en un Entity, aunque fijemos su State y eso implique que se fije un Movement concreto, no se invocara al método move () de ese Movement de manera automática en el método updatePositions () del Entity.

Un Character no es más que un Entity que hace eso. Es decir que, en su método updatePositions () invoca al método move () fijado con el método currentMovement () del Character. Pero bien, seguro que recuerdas que ese método debe recibir tres parámetros: dirección del movimiento, aceleración y una referencia a la Entity (en este caso Character) que debe mover.

Character distingue entre orientación y dirección de movimiento. Recuerde que la orientación del Character (y de la Entity) se almacena en la estructura interna Data, en el atributo \_orientation.

En Character se define el método setMove () que establecerá los atributos que en cada pasada de updatePositions () se terminen por pasar al método move () de Character (y de Entity).

Un Character puede también moverse y animarse de manera autónoma sin que los cambios que sufra de animación y movimiento sean consecuencia de que el jugador/a haya hecho algo con el joystick o el teclado, por ejemplo. Pero aprenderemos eso más adelante. Funcionalidad que es fundamental para representar a los otros personajes del juego que no son el protagonista.

# Las boyas de señalización: La clase Buoy

Imaginemos un poco cómo podría funcionar una Entity/Character en un juego:

Nuestro personaje está corriendo hacia la derecha.

Se ejecuta el método updatePositions () del Character que invoca a move () del Movement activo. move () invoca de vuelta sobre nuestro Character, y antes de ejecutar la actualización de \_position y \_orientation, al método canMove () para comprobar si el método es posible o no.

El método retorna false. Movement lanza una notificación indicando esta situación.

Character está suscrito a los eventos que vienen de sus movimientos y lo procesa en su método processEvent (). Ante esta situación, se decide que el estado de animación y movimiento de nuestro Character debe cambiar y se invoca (dentro de processEvent) al método setCurrentState ().

Ese método cambia la animación y el movimiento, invocando, entre otras cosas al anterior (a su método finalize ()) y dejándolo de observar, alterando así la lista de observers del movimiento anterior.

processEvent () termina y retorna al procesamiento de una lista (la de observadores de Movement) que ha quedado inconsistente (se ha desactualizado en el anterior paso).

Esto origina que el sistema se rompa. Es necesario, por tanto, que la actualización de estados este desacoplada, por regla general, del procesamiento de eventos que se reciben de los objetos observados.

Pero, ¿Cómo puede hacerse eso?: Simplemente “apuntándonos” en la ejecución del método “processEvent” que debemos hacerlo una vez termine la gestión de eventos y gestionara ese recordatorio, por ejemplo, en la siguiente iteración de upodatePositions () o, mejor, en la siguiente ejecución de “inEveryLoop ()” dado que es algo ya decidido y que no debe estar ligado a los frames.

Para hacer esos “apuntes” se utilizan las boyas. Las boyas pueden utilizarse para muchas otras cosas pero habitualmente para situaciones muy similares a la descrita arriba de inconsistencia en cadenas de notificación.

# Una parada en el camino

Además de comentar la estructura general del framework *QGAMES*, a estas alturas de documento ya deberíamos tener claras las siguientes cosas, como mínimo:

1. Todo en un juego se articula alrededor del método *exec ()* de la clase *Game* que, en el fondo no es más que un inmenso bucle que siempre hace 3 grandes cosas: lee eventos, actualiza los elementos del juego, y los dibuja.
2. Cualquier elemento que interviene en un juego se representa con una clase que derive de *Element*.
3. Un *Entity* no más que un *Element* dotado de movimiento y animación, representadas ambas cosas respectivamente mediante las clases *Movement* y *Animation* o hijas de éstas.
4. *Form* es la manera de representar una estructura compleja de información que acabará dibujada en una pantalla (*Screen*). Un *Form* está compuesto de 1 o más *Frame*, que son aspectos diferentes que puede podría tener ese *Form* en el tiempo.
5. En un juego 2D o 2.5D los *Form* son habitualmente guardados como ficheros gráficos (tipo .png, .jpg, .tiff, etc.). En esos ficheros se encuentran todas sus posibles formas o *Frame*. La clase que dentro de *QGAMES* sabe tratar con ellos son *Sprite2D* y *Sprite2DFrame*.
6. La *Animation* más habitual para un *Entity*, *FormAnimation*, es la que se articula alrededor de cambios entre los *Frames* de un *Form* en cada bucle del juego.
7. *Movement* puede tratar un solo *Entity* o varios. Cuando el movimiento es complejo, lo normal será que *Movement* esté relacionado unívocamente con un *Entity* dado que casi seguro habrá de guardar datos de estados previos asociados a entidad de cuyo movimiento se ocupa.
8. *IncrementalMovement* se encarga de mover un *Entity* píxel a píxel entre dos puntos definidos por el incremento en pixeles desde una posición inicial, comprobando en cada paso si el movimiento es o no posible, preguntándoselo a la entidad que mueve (*canMove ()*).
9. Todo *Element* define *Counters* y *OnOffSwitches* que sirven para gestionar tareas muy habituales relacionadas con su estado. Además, el valor de esos *Counters* y *OnOffSwitches* se puede muy fácilmente salvar y recuperar (y por supuesto cambiar), simplificando tareas muy comunes luego como salvar el estado de un juego en un momento determinado, y facilitando la lectura y mantenimiento del código.
10. Para almacenar valores generales del juego se definen las clases *OpenValue* y *SetOfOpenValues*.
11. Todo el juego termina dibujado en una o varias pantallas (*Screen*). Cada pantalla representa el estado de un juego a través de una o varias cámaras (*Camera*), aunque sólo una puede estar activa en cada momento. Cada *Camera* se ubica y orienta en una posición determinada del espacio absoluto en el que se mueven los *Element* del juego y proyecta lo que ve sobre su pantalla usando diferentes métodos: ortogonal, isométrico, caballera o cónico.

Guau. ¿Merece la pena que volvamos a algunos de los puntos anteriores? Hágalo por favor si así lo requiere. En cualquier caso, vamos a introducir el primer ejemplo de un juego. Y vamos a irlo desgranado.

# Nuestro primer juego

Vamos a construir la estructura básica de un juego.

Este será nuestro primer Hello World en QGAMES: Un ninja corriendo por la pantalla. Desgranaremos todos los conceptos aprendidos.

Anexos

# Ilustraciones

[1: Uno de esos juegos que copiaba (de la revista Commodore Magazine número 1) 3](#_Toc80263706)

[2: Panel de control 13](#_Toc80263707)

[3: Propiedades del Sistema. Opciones avanzadas. 14](#_Toc80263708)

[4: Variables de entorno 15](#_Toc80263709)

[5: Por fin. Ya podemos empezar a trabajar 16](#_Toc80263710)

[6: La ayuda 17](#_Toc80263711)

[7: Arquitectura general de QGAMES 21](#_Toc80263712)

[8: Un ejemplo de juego 25](#_Toc80263713)

[9: El método exec de la clase Game 26](#_Toc80263714)

[10: Volante de inercia de un juego. Detalle del método exec de la clase Game 27](#_Toc80263715)

[11: La clase GameImplementation 29](#_Toc80263716)

[12: La clase SDLGame que implementa GameImplementation 31](#_Toc80263717)

[13: La clase Screen, definida en screen.hpp 37](#_Toc80263718)

[14: Método drawPoint en la clase Screen 38](#_Toc80263719)

[15: La clase SDLScreen. Wrapper sobre SDL para dibujar 40](#_Toc80263720)

[16; Dibujar un punto en SDL 40](#_Toc80263721)

[17: La clase Camera. Definida en screen.hpp 47](#_Toc80263722)

[18: Constructor de Camera 48](#_Toc80263723)

[19: Constructor de la clase Screen 49](#_Toc80263724)

[20: La clase Transaction, definida dentro de screen.hpp 49](#_Toc80263725)

[21: La clase Transaction para SDL 50](#_Toc80263726)

[22: Los métodos openFirst y closeFirst de Transaction para SDL 50](#_Toc80263727)

[23: Un ejemplo del método createScreen en la clase Game 51](#_Toc80263728)

[24: La clase Event 59](#_Toc80263729)

[25: Las clases Notifier y Observer 61](#_Toc80263730)

[26: La clase Counter, definida en counter.hpp 63](#_Toc80263731)

[27: La clase OnOffSwitch, definida en onoffswitch.hpp 66](#_Toc80263732)

[28: La clase OnOffSwitches, definida en onoffswitch.hpp 67](#_Toc80263733)

[29: La clase OpenValue, definida en openvalue.hpp 69](#_Toc80263734)

[30: La clase SetOfOpenValues, definida en openvalue.hpp 70](#_Toc80263735)

[31: Macro definida para añadir Counters, definida en counters.hpp 73](#_Toc80263736)

[32: Macro definida para añadir la implementación de Counters, definida en counters.hpp 73](#_Toc80263737)

[33: Macro definida para añadir OnOffSwitches, definida en onoffswitches.hpp 74](#_Toc80263738)

[34: Macro definida para implementar OnOfffSwitches, definida en onoffswitches.hpp 74](#_Toc80263739)

[35: La clase Movement, definida en movement.hpp 79](#_Toc80263740)

[36: Movimientos incrementales pixel a pixel, definido en movement.hpp 82](#_Toc80263741)

[37: Ejemplo sprite sheet 85](#_Toc80263742)

[38: La clase Frame, definida en form.hpp 87](#_Toc80263743)

[39: La clase Form, definida en form.hpp 90](#_Toc80263744)

[40: El método drawOn () de Form 91](#_Toc80263745)

[41: La clase Sprite2DFrame, definida en form.hpp 92](#_Toc80263746)

[42: La clase Sprite2DFrameData, definida en form.hpp 94](#_Toc80263747)

[43: La clase Sprite2DOrtogonalFrameData, definida en form.hpp 95](#_Toc80263748)

[44: La clase Sprite2DFrameDataFactory, definida en form.hpp 96](#_Toc80263749)

[45: La clase Sprite2D, definida en form.hpp 98](#_Toc80263750)

[46: La clase Sprite2DFormData, definida en form.hpp 99](#_Toc80263751)

# Ecuaciones

No se encuentran elementos de tabla de ilustraciones.

1. First Person Shooter [↑](#footnote-ref-1)
2. Abstracción [↑](#footnote-ref-2)
3. Todas ellas multiplataforma y disponibles bajo en C++ [↑](#footnote-ref-3)
4. Un wrapper es un concepto de diseño de software que se utiliza para describir un conjunto de funciones de nombre “conocido” que enmascaran posibles diferentes formas de acceder a otras. Un wrapper, a fin de cuentas, es una especie de traductor. [↑](#footnote-ref-4)
5. Design Patterns: Elements of Reusable Object-Oriented Software (Addison Wesley professional computing series 1994) [↑](#footnote-ref-5)
6. Por defecto 60 y siempre y cuando la tarjeta gráfica y el display lo soporten [↑](#footnote-ref-6)
7. Un juego QGAMES puede tener varias. El método *mainScreen* se utiliza para acceder aquella que se haya definido como principal. [↑](#footnote-ref-7)
8. Ver documento: Matemáticas básicas para el desarrollo de juegos (<https://www.linkedin.com/posts/ignacio-cea-fornies-b36071b_matem%C3%A1tica-b%C3%A1sica-para-el-desarrollo-de-juegos-activity-6735935254676287488-S82S>) también desarrollado por el autor. [↑](#footnote-ref-8)
9. Design Patterns. Addison Wesley [↑](#footnote-ref-9)
10. Ver matemáticas básicas para el desarrollo de juegos. [↑](#footnote-ref-10)
11. Se ha eliminado del código mostrado la definición de la clase Animation, vista en capítulos anteriores. [↑](#footnote-ref-11)
12. Recuerda que esos métodos nos permiten salvar y recuperar el estado de un elemento, facilitando así acciones como salvar y recuperar el estado de un juego. [↑](#footnote-ref-12)