Librería QGAMES

En este documento se describe como utilizar la librería QGAMES.

Está pensado para ser utilizado por quien quiera usarla para desarrollar sus propios juegos.

No contiene, por tanto, información detallada sobre su diseño que permita su administración y mantenimiento. Si contiene, por el contrario, información acerca de cómo ampliarla, pensando en aquellos juegos en los que las características básicas o avanzadas de QGAMES no resultaran suficientes para cubrir las necesidades del usuario.

El documento está estructurado en tres grandes niveles, partiendo de lo más general a lo más particular, para facilitar la comprensión por parte del lector.

# Contenido

[Contenido 2](#_Toc41843028)

[Nivel Básico 3](#_Toc41843029)

[Arquitectura general 4](#_Toc41843030)

[Wrappers 5](#_Toc41843031)

[Módulos básicos 5](#_Toc41843032)

[Módulos avanzados 5](#_Toc41843033)

[Módulos plantilla de juegos 5](#_Toc41843034)

[Estructura básica de un juego 7](#_Toc41843035)

[El Juego. Clase QGAMES::Game 7](#_Toc41843036)

[Los recursos de un juego 10](#_Toc41843037)

[Las formas. Clase QGAMES::Form 12](#_Toc41843038)

[Los elementos de un juego. Clase QGAMES::Element 13](#_Toc41843039)

[Las Entidades. Clase QGAMES::Entity 14](#_Toc41843040)

[Los Builders 14](#_Toc41843041)

[El acceso al hardware. La implementación. Clase QGAMES::GameImplementation 17](#_Toc41843042)

[El receptor de eventos. Clase QGAMES::InputHandler y clase QGAMES::InputHandler::Behaviour 19](#_Toc41843043)

[La Pantalla. Clase QGAMES::Screen 22](#_Toc41843044)

[Las cámaras. Clase QGAMES::Camera 23](#_Toc41843045)

[La estructura básica de un juego 23](#_Toc41843046)

# Nivel Básico

El objetivo es conseguir una comprensión básica del funcionamiento de la librería.

# Arquitectura general

La librería de juegos QGAMES se compone de diversos módulos.

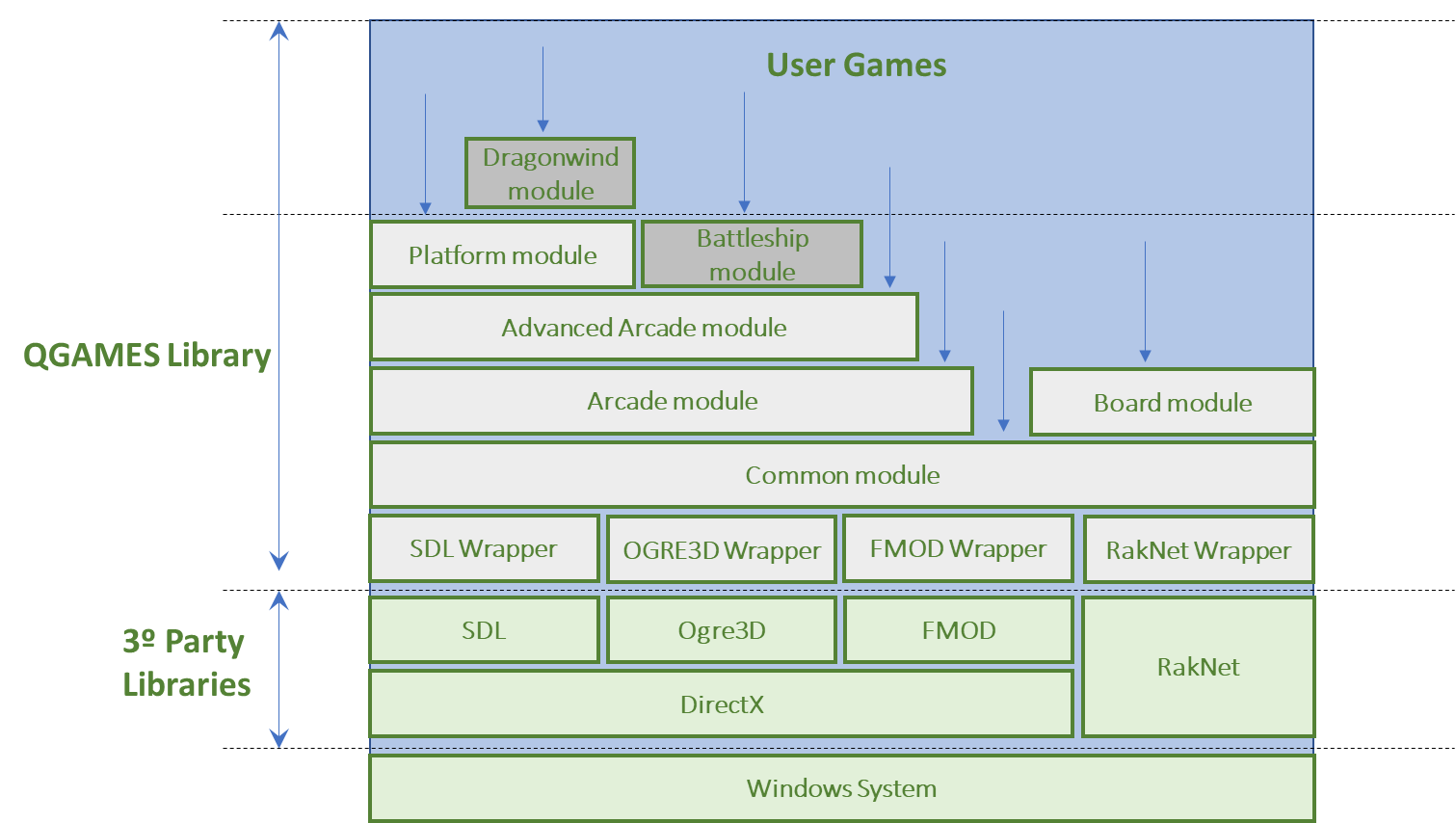
QGAMES no es una librería para dibujar formas 3D o para producir los sonidos que todo juego lleva, o para interactuar con el joystick o ratón utilizados por los jugadores. QGAMES se apoya en librerías estándar multiplataforma para llevar a cabo esas tareas, focalizándose sólo en organizar y dar estructura a la lógica global de funcionamiento inherente a todo juego, de tal manera que el programador deba 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, de plataformas, de rol, o de mesa. 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.

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 los wrappers 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:



1: 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 de funciones para simplificar el acceso a los sistemas de audio, teclado, joystick y gráficos del computador. SDL Wrapper define una capa de abstracción sobre SDL para facilitar su integración en la estructura de funcionamiento definida en “Common”. Existen otras muchas librerías tipo SDL, como SFML o Allegro, y QGAMES está diseñada para facilitar la creación de wrappers que puedan integrarse 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, 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 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 tablero.
* **Arcade module**: Añade a Common las propiedades de cualquier juego de acción.

## Módulos avanzados

Módulos para realizar juegos complejos:

* **Advanced Arcade Module**: Añade a Arcade características adicionales que permiten juegos más complejos.
* **Platform module**: Añade a Advanced Arcade las funciones necesarias para desarrollar juegos basados en plataformas.

## Módulos plantilla de juegos

Estos módulos son algo más que librerías. Incorporan ya la lógica completa y funcional de un juego completo. Sirven para primero aprender y pueden ser fácilmente modificados para desarrollar juegos similares:

* **Battleship like games module**: Battleship es un ejemplo de juego arcade construido sobre el módulo Advanced Arcade que se ha convertido además en plantilla modificable para simplificar la realización de juegos de lógica parecidos. El protagonista es una nave espacial que ha de eliminar todos los asteroides que se le acercan, para llegar a alcanzar su objetivo. La velocidad y potencia de los asteroides se irá incrementando conforme nuestro protagonista vaya superando diferentes niveles.
* **Dragonwind like game module**: Dragonwind es un ejemplo de juego de plataformas (tipo Mario Bross) que, al igual que sucede con Battleship, se ha convertido en plantilla para simplificar el desarrollo de juegos parecidos.

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.

# Estructura básica de un juego

En el siguiente capítulo vamos a recorrer de manera ligera los elementos más importantes para la creación de juego, de tal manera que podamos hacernos una idea de qué es QGAMES y cómo funciona. En capítulos posteriores bajaremos al detalle de cada uno de esos elementos.

## El Juego. Clase QGAMES::Game

Cualquier juego, en última instancia, se basa en un bucle infinito que sólo finaliza cuando termina el juego y dentro del cual se realizan siempre 5 funciones básicas:

1. **Leer** los dispositivos de entrada (joystick, teclado, ratón, …),
2. **Actualizar** la posición / forma / aspecto de todos los elementos que intervienen en el juego en base a esa lectura,
3. **Tomar decisiones** relacionadas con esa situación, como por ejemplo si hay alguien que muere o no, o si se ha contactado con un interruptor que abre una puerta. Es decir, la lógica propia del juego.
4. **Dibujar** en pantalla o pantallas lo que haya de ser dibujado atendiendo a la
5. **Comunicar** a través de la red (opcional) la situación del juego. Esto es algo que tiene sentido sólo en los juegos en red.

A partir de ahora denominaremos a este bucle como “**mainloop**”.

Para que un juego se aprecie con realismo suficiente la pantalla debe refrescarse un mínimo de 60 veces por segundo, lo que implica dos cosas: que mainloop se ejecute al menos 60 veces por segundo y que todas y cada una de las tareas de ese bucle se diseñen para ser óptimas. Es ahí donde la librería QGAMES ayuda a estructurar el bucle y a la optimización de las tareas básicas del mismo.

Por otro lado, y también para garantizar el realismo del juego, el mainloop debe ejecutarse siempre en un tiempo similar. Es decir que no es admisible que durante un periodo de tiempo el bucle se ejecute 40 veces por segundo y al momento siguiente lo haga 60 veces por segundo. Se apreciarían aceleraciones y retardos en el movimiento de los personajes del juego que afectarían a su realismo y jugabilidad.

Por tanto, es necesario incluir una sexta tarea en el mainloop que haga de buffer temporal variable para **asegurar que la uniformidad en sus tiempos de ejecución**.

Por otro lado, reservar y liberar memoria es seguramente no sólo una de las tareas con los que más cuidado hay que tener al programar en C o C++, sino también una de las que más tiempo implica. Es recomendable, por tanto, ubicar la mayor parte del trabajo posible de reserva de memoria antes del inicio de la ejecución del mainloop, y dejar el de liberación de memoria para cuando aquel finalice. Nuevamente la arquitectura con la que está diseñada QGAMES facilita esa tarea.

El mainloop se define en la clase QGAMES::Game, en el método exec. Algunos de los métodos de esta clase son virtuales puros, por lo que necesitan una definición explícita. Por tanto, para crear un nuevo juego, las primeras tareas a realizar son: crear una clase que herede de QGAMES::Game, implementar al menos los métodos declarados como virtuales puros y escribir un main que, como mínimo, instancia la nueva clase e invoque al método exec.

Desde el método exec se invocan fundamentalmente muchos métodos definidos en la propia clase. Todos ellos tienen una implementación por defecto básica (se detallará más adelante), pero obviamente pueden ser sobrecargados para cubrir objetivos diferentes.

El método exec se describe inicialmente con dos métodos:

* initialize (): Conjunto de acciones a realizar para inicializar el juego. Dentro de este método deberían incluirse, por ejemplo, tareas como cargar los gráficos más pesados e importantes, cargar los sonidos de fondo, crear y estructuras mapas de acceso, etc.
* finalize (): Conjunto de acciones a realizar para finalizar el juego. Dentro de este método habrá de, entre otras cosas, por ejemplo: eliminar los objetos previamente inicializados.

Ambos métodos se ejecutan dentro del método exec, pero fuera del mainloop. Justo antes y después del mismo respectivamente.

Dentro del mainloop hay tareas que se realizan una vez por bucle y otras que se llevan a cabo una vez por frame. Recuerde que, como se adelantó al principio del capítulo, QGAMES garantiza que el número de frames por segundo que se dibujan se mantiene constante y, dado que no se puede fijar el tiempo que por bucle tardan las diferentes acciones del juego (pues estas irán variando en el tiempo), QGAMES implementa un buffer de tiempo que fuerza a ejecutar bucles vacíos cuando la velocidad de ejecución del juego es superior a la de los frames por segundo requeridos.

Por otro lado, QGAMES incluye funciones para poder detener el juego. Cuando el juego está detenido, debe seguir dibujándose, pero no deben actualizarse, por ejemplo, las posiciones de los elementos en la pantalla o considerar las colisiones entre ellos.

Teniendo ambas cosas presentes, los métodos que exec ejecutan dentro del mainloop son:

* handleEvents (): Método para recoger y tratar las interacciones del usuario con los diversos dispositivos conectados (joysticks, ratones, teclado, etc.). Se ejecuta por bucle. Cuando el método esté “detenido” este método seguirá ejecutándose.
* inEveryLoop (): En la implementación por defecto, entre otras tareas, se actualiza el reloj del sistema. Se ejecuta por bucle. Cuando el método esté “detenido” este método seguirá ejecutándose.
* updatePositions (): Sólo se ejecuta antes de pintar un frame. Por tanto, puede llegar a ejecutarse menos veces que el anterior. Nunca más. Cuando el juego esté “detenido” este método no se ejecuta.
* drawOn (): Para actualizar el contenido de las diferentes pantallas que maneje el juego. Se ejecuta por frame. Cuando el método esté “detenido” este método seguirá ejecutándose.
* additionalTasks (): Concebido como un lugar de expansión abierto, sólo se ejecuta cuando el juego no esté “detenido” y una vez por bucle.

Todos y cada uno de esos métodos pueden redefinirse según sus propias necesidades. Todos definen acciones por defecto tremendamente importantes para el correcto funcionamiento del juego, por lo que si usted necesita sobrecargarlos no se olvide de invocar al método parent en su implementación.

Analice más detenidamente la implementación del método exec. Quizás haya varias cosas que no comprenda. No sé preocupe, las iremos desgranado poco a poco.

void QGAMES::Game::exec ()

{

#ifdef NDEBUG

setLog (false);

#else

setLog (true);

#endif

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

srand ((int) time (NULL));

int loopCounter = 0;

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

initialize ();

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

if (!isInitialized ())

return;

\_running = true;

while (\_running)

{

handleEvents ();

inEveryLoop ();

if (QGAMES::GUISystem::system () != NULL)

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

bool mustDoLoop = true;

if (\_linkDrawToFrameRate)

{

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

loopCounter = 0;

else

mustDoLoop = false;

}

if (mustDoLoop)

{

if (!\_paused)

{

if (\_currentGameState != NULL)

\_currentGameState -> updatePositions ();

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

updatePositions ();

}

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

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

{

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

if (mustIUpdate ())

scr -> updatePositions ();

scr -> openTransaction ();

if (drawGameBeforeState ())

drawOn (scr);

if (\_currentGameState != NULL)

\_currentGameState -> drawOn (scr);

if (QGAMES::GUISystem::system () != NULL)

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

if (!drawGameBeforeState ())

drawOn (scr);

scr -> drawOnYourOwn ();

scr -> closeTransaction ();

}

if (!\_paused)

{

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

(\*j) -> count ();

additionalTasks ();

}

}

}

\_running = false;

finalize ();

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

}

## Los recursos de un juego

En un juego pueden llegar a intervenir muchos tipos de elementos (recursos). A lo largo del documento irán poco a poco todos ellos, siendo los más importante los siguientes:

* **Sonidos**: Representados por la clase base QGAMES::Sound.
* **Formas**: Representados por la clase base QGAMES::Form. Una forma es un gráfico. Cualquier forma está constituida por diversos frames (QGAMES::Frame). Las formas pueden ser 2D o 3D.
* **Entidades y Caracteres**: Uno de los recursos más importantes de cualquier juego. Están representados respectivamente por las clases QGAMES::Entity y QGAMES::Character.

Las entidades suelen representar elementos del juego con los que el jugador interactúa de manera directa o indirecta. Una entidad es el personaje que el jugador mueve por la pantalla usando su joystick, los enemigos a los que éste se enfrenta o los utensilios que ha de recoger por el camino.

Una entidad puede estar en diferentes formas (QGAMES::Form) y tener diferentes movimientos (QGAMES::Movement). Sobre una entidad se pueden establecer animaciones que son maneras de poder cambiar con cada bucle la forma adoptada por una entidad, creando así el efecto de vida.

Cualquier entidad tiene siempre un ciclo de vida por el que pasa por diferentes estados (QGAMES::Entity::State) que, en última instancia, no son más que una forma de agrupar una animación y un movimiento.

Un Carácter es una entidad dotada de automovimiento. Es decir que una vez fijado su movimiento mediante una dirección y aceleración incial, éste se mantendrá hasta que aquellos parámetros cambien.

* **Textos**: Representados por la clase QGAMES::ScoreText, que deriva de QGAMES::Entity.
* **Estados de Juego**: Representados por la clase QGAMES::GameState.

Los estados son fundamentales para controlar el flujo del juego. La Introducción, El menú de selección inicial, el propio juego, cuando el personaje muere, los títulos de crédito o el ”hall of fame”, son todos ejemplos de diferentes estados por los que pasa un juego. Como ir de uno a otro es lo que se conoce como el flujo de juego. Por tanto, esta es una de las clases más importantes a considerar para construir un juego relevante.

* **Objetos**: Representados por la clase QGAMES::Object. Objetos son líneas, círculos, elipses, etc, sólidos o transparentes, 2D o 3D.
* **Sistemas de Interacción Gráfica**: Representados por la clase QGAMES::GUISystem. Un sistema de interacción gráfica es por ejemplo el menú de selección inicial de cualquier juego, o el menú que se abre cuando durante el juego pulsamos ESC para acceder a opciones adicionales para poder, por ejemplo, salvar el estado del juego. Manejarlos correctamente es algo muy importante.
* **Movimientos**: En la clase QGAMES::Movement. Ya hemos indicado que los movimientos están vinculados a las entidades y caracteres. Un movimiento es, por ejemplo, la parábola que describe en el aire el personaje principal cuando salta, o el bote repetitivo al chocar con el suelo de una bomba lanzada por el enemigo.
* **Monitores de pasos**: Representados en la clase QGAMES::CharacterControlStepsMonitor.

Un monitor de pasos es un objeto capaz de automatizar los estados por los que pasa una entidad. Por tanto, es el elemento necesario para poder, por ejemplo, automatizar el movimiento de los enemigos en cualquier escena del juego.

* **Máscaras de presentación**: Representadas en la clase QGAMES::Mask. Las máscaras con elementos que se añaden sobre la pantalla antes de pintar. Sirven para, por ejemplo, conseguir efectos como el de mirar por un catalejo.
* **Escenas**: Representados por la clase QGAMES::Scene.

Una escena es el lugar en el que transcurre el juego. En cualquier escena siempre hay entidades y caracteres (se o mueven o no por ella). En la escena es en dónde se programa la lógica del juego.

Esa lógica puede agruparse en bloques. Bloques que pueden ser reutilizados entre escenas y que se representan con la clase QGAMES::SceneActionBlock.

Una escena se conecta con otra a través de puertas. Las puertas sirven para determinar la lógica típica de un juego de “paso de niveles”.

* **Mapas**: Representados por la clase QGAMES::Map.

Cualquier escena utiliza de mapas para presentarse en pantalla. Los mapas pueden ser 2D o 3D y están formados por Capas (clase QGAMES::Layer). Hay muchos tipos de mapas: QGAMES::TiledMap, utilizados típicamente en los juegos de plataformas. QGAMES::ObjectMap, para agrupar QGAMES::Object, etc. Los desgranaremos en profundidad más adelante.

El manejo adecuado de las capas permite, como también aprenderemos, realizar efectos complejos sobre la escena, así como marcar caminos para modificar el comportamiento de las entidades y caracteres que hay en ella.

Una escena sólo puede tener un mapa (una forma de pintarse) activa en cada momento.

* **Mundos**: QGAMES::World. Un mundo no es más que la agrupación de diferentes escenas. Un mundo sólo puede tener una escena activa en cada momento.
* **Tableros**: Representados en la clase QGAMES::Board. Un tablero es el ámbito sobre el que tiene lugar un juego de mesa.

## Las formas. Clase QGAMES::Form

El recurso más importante de cualquier juego es quizás QGAMES::Form.

Por hacer una primera definición, una forma es el equivalente a un mapa de bits dibujado en la pantalla. Eso es sólo cierto en un mundo 2D, ¿pero sigue siendo válido en un mundo 3D? ¿Y en un mundo en 2,5D? (Usar proyecciones isométricas para simular realidades 3D). Lo que parece evidente es que una forma no es lo mismo en un entorno 2D que en uno 3D.

QGAMES::Form define muchos métodos virtuales puros para calcular datos relevantes dentro de ese mapa de bits, como su anchura, su altura, su profundidad (que en 2D no tiene sentido por ejemplo), la posición central (que se calcula diferente incluso en 2D cuando la forma representa una proyección ortogonal que cuando representa una isométrica o caballera), etc.

Por esa razón de QGAMES::Form heredan clases casi igual de importantes que ella misma. Una de ellas es, por ejemplo, QGAMES::Sprite2D que implementa todos los métodos virtuales que su clase padre define para el caso de una proyección 2D, bien sea ésta puramente ortogonal, isométrica o caballera.

QGAMES::Form puede tener muchos aspectos diferentes. Eligiendo, por ejemplo, pintar un aspecto diferente por bucle, podríamos conseguir efectos de animación, algo imprescindible en el movimiento de personajes por la pantalla. A cada uno de esos aspectos se le denomina Frame, representado por la clase QGAMES::Frame.

## Los elementos de un juego. Clase QGAMES::Element

Los recursos antes mencionados pueden dividirse en principales y accesorios. Una entidad o un carácter son elementos fundamentales en un juego. Sin embargo, un sonido es accesorio. Puede o no haberlos. En un juego de arcade es fundamental el manejo de niveles (QGAMES::Scene). En un juego mesa lo es el tablero (QGAMES::Board), etc…

QGAMES define la forma en la que los recursos principales deben ser manejados.

Todos ellos son clases que derivan en última instancia de una clase llamada de QGAMES::Element.

Cualquier elemento viene definido fundamentalmente por un id, una propiedad que define si éste es visible o no y un conjunto de métodos básicos que la arquitectura de QGAMES utiliza para manipularlo. Esos métodos son:

* initialize (): Que engloba las acciones básicas a realizar antes de que el elemento empiece a formar parte activa del juego.
* finalize (): Acciones básicas a realizar después de que el elemento deja de ser parte activa del juego.
* inEveryLoop (): Acciones que debe realizar el elemento a realizar por el elemento en cada bucle del juego, esté éste “detenido” o no.
* updatePositions (): También incluye acciones a realizar en cada bucle del juego, pero enfocadas a cuando el juego no está “detenido”. Este método además se invoca tantas veces por segundo como frames deban dibujarse, a diferencia del anterior que no está ligado a este ratio.
* drawOn (Screen\* scr, const Position& p): Para dibujar el objeto en la pantalla.

El método recibe como parámetro una referencia a la pantalla en la que dibujarse (QGAMES puede manejar varias simultáneamente) y una indicación de posición base en la que hacerlo. El primer parámetro es obligatorio y no puede ser NULL. El segundo es optativo y si se suministra la posición del objeto será tomada como relativa respecto de la recibida como parámetro, si no se suministra o es igual a QGAMES::Position::\_noPosition, no se tendrá en consideración.

Como ya habrá supuesto, estos métodos deberían ser invocados desde los correspondientes de QGAMES::Game.

Por tanto, y en líneas muy generales, **crear un juego en su mínima expresión consistiría en**:

* Crear una clase X que herede de QGAMES::Game.
* Crear objetos Y que hereden de QGAMES::Element.
* Redefinir el método initialize () de nuestra clase X para que instancie todos los objetos Y que necesite.
* Redefinir los métodos inEveryLoop () y updatePositions () de la clase X para que, a su vez, invocaran a los equivalentes de los objetos Y creados, cuando las condiciones de cada objeto lo requieran (sean visibles por ejemplo). Cosa que vendrá determinada por la lógica del juego.
* Redefinir los métodos initialize (), finalize (), inEveryLoop (), updatePositions () y drawOn () de cada Y (heredero de QGAMES::Element) definido, para que se comporten según se requiera para cada tipo de elemento.

Evidentemente QGAMES nos ayuda mucho más, pero baste el anterior ejemplo para entender el espíritu de crear un juego.

## Las Entidades. Clase QGAMES::Entity

Como se enunciaba anteriormente, las entidades son elementos fundamentales de cualquier juego. Representan nuestro personaje, los enemigos, los elementos a recoger por el camino, etc.

Una Entidad viene definida por un conjunto de formas (QGAMES::Forms), un conjunto de animaciones (QGAMES::Entity::Animations) y un conjunto de movimientos (QGAMES::Movements). La agrupación de una animación y un movimiento se denomina estado. Por lo que, en definitiva, una Entidad viene definida por un conjunto de formas y un conjunto de estados (QGAMES::Entity::States).

En cada momento del tiempo la entidad tiene una forma y un estado activo (QGAMES::Entity::State); es decir: una animación (QGAMES::Entity::Animation) y un movimiento (QGAMES::Entity::Movement).

Esta clase es realmente crítica, por lo que se profundizará ampliamente en su estructura y funcionamiento más adelante.

## Los Builders

QGAMES se ocupa básicamente de ayudar a ligar los recursos que hemos visto.

Cualquier recurso involucrado en un juego puede (y debe) definirse en un fichero, lo que facilita la parametrización del juego y reduce (como veremos) la necesidad de código al programarlo. Si bien este punto no es obligatorio para la construcción de un juego tal y como se ha puesto de manifiesto en el punto anterior.

Un Builder no es más que una clase especializada en la creación de un tipo concreto de recurso a partir de la información que encuentra en su fichero de definición. Para definir, por tanto, un nuevo juego basado en builders, deben especificarse cuales usar según el tipo de recurso necesario. En este sentido, debe especificarse el builder a utilizar para los gráficos (forms), para los sonidos, para los objetos que se mueven, etc. Y uno y sólo un builder para cada tipo de recurso por juego.

Los builders, por regla general, publican un método que permite recuperar a través de un id, los recursos de su competencia. Cuando un recurso (mediante su id) es solicitado por primera vez al builder, éste buscará sus características en un fichero de definición establecido en tiempo de construcción del builder, lo construirá y lo devolverá, guardando además su referencia en un caché interno. Si ese mismo recurso fuera solicitado una segunda vez (mismo id), el builder devolvería la referencia guardada y no construiría uno nuevo.

Cuando el builder se destruye se destruyen con él los recursos creados. Por tanto, usando un builder, la aplicación (el juego) no tiene por qué preocuparse de en dónde y en qué momento se creó determinado recurso, delegando esa responsabilidad en el builder.

Cada builder, además de exponer un método público para acceder a sus recursos por id, define un método privado que recibe la información de definición de un recurso (la que está en el fichero) y devuelve una instancia a un nuevo elemento. La estructura interna del builder invocará a este método para crearlo por primera vez. Ese método puede ser redefinido por el programador del juego para crear sus propios recursos con las características especiales que aquellos requieran.

Veamos como ejemplo, la estructura básica del builder utilizado para crear recursos del tipo QGAMES::Sound y que representa muy bien la estructura de la mayor parte de los builders:

class SoundBuilder

{

public:

…

SoundBuilder (const std::string& fDef);

virtual ~SoundBuilder ();

…

Sound\* sound (int sId);

…

protected:

…

virtual Sound\* createSound (const SoundDefinition& def) = 0;

…

};

Como se puede observar el método público para acceder a un sonido es el método sound (int sId) y el método privado para crear uno nuevo es Sound\* createSound (const SoundDefinition& def) que, en este caso, no tiene ninguna implementación por defecto y que recibe como parámetro la definición del sonido que el builder haya recogido del fichero. Este es el método que los programadores deben sobrecargar para crear de la manera adecuada (veremos más adelante como) los sonidos.

La clase QGAMES::Game define métodos para acceder a cada uno de los builders básicos (principales y accesorios) que necesita cualquier juego.

QGAMES define, a su vez, clases que heredan de QGAMES::Game y que se utilizan para manejar tipos específicos de juegos (de tablero, de arcade, de plataforma, …) que, entre otras cosas, requieren de recursos más y más complejos cada vez y con ello, definen nuevos métodos nuevos para poder acceder a los builders que los construyen.

Los builders básicos necesarios (principales y accesorios) para cualquier juego, y los métodos para acceder a ellos son los siguientes.

* formBuilder (): devuelve una instancia a QGAMES::FormBuilder para construer recursos del tipo QGAMES::Form.
* textBuilder (): QGAMES::TextBuilder para construir QGAMES::ScoreObjectText.
* maskBuilder (): QGAMES::MaskBuilder para construir QGAMES::Mask.
* objectBuilder (): QGAMES::ObjectBuilder para construir QGAMES::Object.
* entityBuilder (): QGAMES::EntityBuilder para construer QGAMES::Entity.
* guiSystemBuilder (): QGAMES::GUISystemBuilder para construer QGAMES::GUISystem.
* movementBuilder (): QGAMES::MovementBuilder para construer QGAMES::Movement.
* soundBuilder (): QGAMES::SoundBuilder para construer QGAMES::Sound.
* characterMonitorBuilder (): QGAMES::CharacterStepsMonitorBuilder para construir QGAMES::CharacterStepsMonitor.
* communicationsBuilder (): Devuelve un builder capaz de crear sistemas de comunicaciones con el exterior. Muy útil cuando el juego se desarrolla en red.
* gamestateBuilder (): QGAMES::GameStateBuilder para construir QGAMES::GameState.

Por otro lado, las clases que derivan de QGAMES::Game son: QGAMES::ArcadeGame, QGAMES::BoardGameApp, QGAMES::AdvancedArcade. QGAMES::PlatformGame, QGAMES::BattleshipGame y QGAMES:DragonwindGame, y los recursos, builders y métodos para acceder a ellos que cada uno añade son los siguientes:

En la clase QGAMES::ArcadeGame:

* worldBuilder (): QGAMES::WorldBuilder ara construir QGAMES::World y QGAMES::Scene asociados, así como las conexiones entre ellas.
* mapBuilder (): QGAMES::MapBuilder para construir QGAMES::Map, con sus diferentes capas asociadas.

En la clase QGAMES::BoardGameApp:

* boardBuilder (): QGAMES::BoardBuilder para construir QGAMES::Board y otros recursos asociados a él que en última instancia también heredan de QGAMES::Element.
* boardGameBuilder (): Un builder para crear tableros de juego.

El resto de clases que derivan de QGAMES::Game no añaden nuevos recursos y con ello tampoco nuevos builder.

Ha quedado antes claro que siempre es posible redefinir nuevos recursos como nuevos comportamientos. Aquellos nuevos recursos que, a su vez, hereden directamente de QGAMES::Element es, además, obligatorio redefinir el comportamiento de ciertos métodos (initialize, finalize, …).

Si nuestro juego se apoyara en el uso de builders para construir dichos recursos (ya hemos adelantado que no es obligatorio, aunque sí muy recomendable para minimizar el código programado), habremos de redefinir también sus correspondientes builders.

Dentro de QGAMES::Game se definen métodos, algunos abstractos pero todos protected (por tanto no accesibles desde el exterior), para crear la instancia concreta de cada builder que se necesita. Los métodos públicos que antes hemos indicado (por ejemplo, formBuilder ()) invocan a su equivalente protegido la primera vez con el fin de crear el builder concreto. En el resto de las ocasiones harán ya uso de la instancia de builder antes creada.

Así, por ejemplo, formBuilder () llama a createFormBuilder () la primera vez que es invocado. Para que el juego considere nuestro propio builder, es necesario por tanto redefinir éste último en una nueva clase que herede de QGAMES::Game o de cualquiera hija de ésta incluida en la librería (QGAMES::ArcadeGame, QGAMES::BoardGame, QGAMES::AdvancedArcadeGame, QGAMES::PlatformGame, QGAMES::DragonwindGame o QGAMES::BattleshipGame).

De manera semejante a lo siguiente:

class GameTest : public QGAMES::Game

{

…

protected:

…

FormBuilder\* createFormBuilder () { return (…); }

…

};

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

{

TestGame::GameTest game;

game.exec ();

return (0);

}

## El acceso al hardware. La implementación. Clase QGAMES::GameImplementation

QGAMES se centra, sobre todo y como ya hemos adelantado anteriormente, en ayudar a estructurar la lógica de un juego.

Para todas aquellas tareas relacionadas con el manejo de los recursos propios de la máquina (control de dispositivos de entrada, manejo de gráficos y sonidos, y comunicaciones), QGAMES se apoya en librerías estándar de sobrado prestigio en el mercado de los videojuegos. Librerías que integra dentro de su arquitectura para que su manejo sea transparente al programador. Los más avanzados pueden incluso llegar a integrar bajo la arquitectura de QGAMES sus propias librerías (algo que se trata en los últimos capítulos de este libreo).

Por ejemplo, para el manejo de gráficos 2D se integra con SDL ([www.libsdl.org](http://www.libsdl.org)), para los gráficos 3D con ogre3D ([www.ogre3D.org](http://www.ogre3D.org)), para los sonidos con FMOD ([www.fmod.com](http://www.fmod.com)) para el manejo de los dispositivos de entrada también con SDL y para la comunicaciones con Raknet (<http://www.jenkinssoftware.com/>).

La forma de integrar cada librería depende del tipo de recurso a integrar. Por ejemplo, para sonidos basados en la librería SDL se define un tipo de sonido nuevo, QGAMES::SDLSound, que hereda de QGAMES::Sound que redefine los métodos int play (int cht) y similares para utilizar las funciones específicas que SDL provee para manejar sonidos y música.

class SDLSound : public QGAMES::Sound

{

public:

SDLSound (int id, sound\_type t, const std::string& s, int l, Mix\_Music\* m)

: QGAMES::Sound (id, t, s, l),

\_music (m),

\_sound (NULL)

{ }

SDLSound (int id, sound\_type t, const std::string& s, int l, Mix\_Chunk\* m)

: QGAMES::Sound (id, t, s, l),

\_music (NULL),

\_sound (m)

{ }

virtual ~SDLSound ();

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

virtual int play (int ch);

virtual void pause ();

virtual void resume ();

virtual void stop ();

virtual bool isPlaying ();

protected:

Mix\_Music\* \_music;

Mix\_Chunk\* \_sound;

};

Para los sonidos basados en FMOD, se implementa una clase similar:

class FMODSound : public QGAMES::Sound

{

public:

typedef enum

{

\_2D = 0,

\_3D,

} Type;

FMODSound (int id, FMOD::System\* sys, FMOD::Sound\* sd,

QGAMES::Sound::sound\_type t, const std::string& s, int l)

: QGAMES::Sound (id, t, s, l),

\_fmodSystem (sys),

\_fmodSound (sd),

\_fmodChannel (NULL)

{ }

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

virtual int play (int ch);

virtual void pause ();

virtual void resume ();

virtual void stop ();

virtual bool isPlaying ();

protected:

Type \_fmodSoundType;

FMOD::System\* \_fmodSystem;

FMOD::Sound\* \_fmodSound;

// Implementation

FMOD::Channel\* \_fmodChannel;

};

Como ya vimos en el punto anterior, para crear sonidos (y cualquier otro tipo de recurso) se recomienda la utilización de builders. Y para crear el builder de sonido debemos redefinir un método dentro de la clase QGAMES::Game llamado SoundBuilder\* createSoundBuilder (…) lo que nos llevaría a que, para utilizar sonidos basados en SDL, deberíamos finalmente tener una clase llamada algo así como QGAMES::SDLGame. Si los gráficos, por otro lado, fueran, por ejemplo 3D, seguiríamos un razonamiento similar con la clase QGAMES::Form, lo que finalmente nos llevaría a combinaciones infinitas y a una complejidad enorme en el manejo de la clase QGAMES::Game y sus derivadas.

Para evitar esto y dotar de más flexibilidad a QGAMES, la creación de builders relacionados con los recursos más pegados al hardware se delega en la clase QGAMES::GameImplementation (y derivadas).

El constructor de QGAMES::Game requiere dos parámetros. El segundo es optativo y se refiere al número de frames por segundo que se utilizarán como referencia (por defecto es 60). El primero, por el contrario, es obligatorio y es una referencia a un objeto QGAMES::GameImplementation.

Es en la clase QGAMES::Game, el método SoundBuilder\* createSoundBuilder () delega en el objeto QGAMES::Implementation para construir el builder adecuado con los sonidos. Algo análogo sucede con recursos tipo QGAMES::Forms.

Así:

class GameImplementation

{

public:

GameImplementation ()

: \_game (NULL)

{ }

virtual ~GameImplementation ()

{ }

…

virtual FormBuilder\* createFormBuilder () = 0;

virtual SoundBuilder\* createSoundBuilder () = 0;

…

};

## El receptor de eventos. Clase QGAMES::InputHandler y clase QGAMES::InputHandler::Behaviour

Todo juego necesita interactuar con dispositivos de entrada y de salida. Dispositivos de entrada son, generalmente, el teclado, el ratón y el joystick. Dispositivos de salida son, habitualmente, la o las pantallas.

Para interactuar con dispositivos de entrada se utiliza la clase QGAMES::InputHandler o cualquier otra que herede de esta. En un juego debe haber una y sólo una instancia de QGAMES::InputHandler.

Por otro lado, esa instancia debe ser creada antes de comenzar la primera ejecución del mainloop pues, según se ha comentado antes, una de las 5 cosas que se hacen cíclicamente en el bucle principal es invocar al método handleEvents (), en el que deben leerse y procesarse los eventos recibidos de ese input handler.

Por último, y dado que los dispositivos de entrada son algo cercano al hardware, y siguiendo la lógica de diseño aplicada en puntos anteriores, la instancia concreta debe crearse en QGAMES::GameImplementation.

Poniendo juntos todos los condicionantes aportados:

Dentro de QGAMES::GameImplementation se define el método virtual puro QGAMES::InputHandler\* createInputHandler (QGAMES::InputHandler::Behaviour\* bhv), y que deberá ser especificado para el tipo de librería de bajo nivel concreto (SDL por ejemplo) que se esté utilizando.

Dentro de QGAMES::Game se define el método QGAMES::InputHandler\* inputHandler (), que llama al anterior la primera vez que es invocado, y sirve para devolver la instancia de QGAMES::InputHandler concreta utilizada en el juego.

Este método es invocado, a su vez, por la implementación por defecto del método QGAMES::Game::initialize () ejecutado en QGAMES::Game::exec () antes de entrar en la ejecución del mainloop.

Por tanto, con simplemente redefinir el método para crear un InputHandler definido en la clase QGAMES::GameImplementation que se pasa como parámetro de construcción a nuestra instancia de QGAMES::Game, sería suficiente.

Habrá observado que para construir una instancia de QGAMES::InputHandler es necesario proporcionar una referencia a un objeto del tipo QGAMES::InputHandler::Behaviour. QGAMES implementa, en el manejo de los dispositivos de entrada, un nivel adicional de indirección, delegando en un objeto QGAMES::InputHandler:.Behaviour lo que hacer en concreto con cada uno de los eventos leídos en el sistema.

El proceso de lectura y tratamiento de los eventos es el siguiente:

Dentro del mainloop de QGAMES::Game::exec se invoca al método handleEvents (), que invocará al método equivalente de QGAMES::GameImplementation, quien recuperará y procesará (uno a uno) todos los eventos pendientes del sistema.

Para tratar los eventos recuperados se invoca al método void treatEvent (const QGAMES::Event& evnt). Ese método, por defecto, y según el origen del evento (teclado, ratón o joystick), invocará a diferentes métodos definidos en QGAMES::InputHandler.

void QGAMES::GameImplementation::treatEvent (const Event& evnt)

{

switch (evnt.code ())

{

case \_\_QGAMES\_MOVEJOYSTICKEVENT\_\_:

{

JoystickMovementEventData\* dt = (JoystickMovementEventData\*) evnt.data ();

assert (dt); // Event is something can be generated by the user...

\_game -> inputHandler () -> onJoystickAxisMoveEvent (dt);

delete (dt);

}

break;

case \_\_QGAMES\_BUTTONJOYSTICKEVENT\_\_:

{

JoystickButtonEventData\* dt = (JoystickButtonEventData\*) evnt.data ();

assert (dt); // Event is something can be generated by the user...

\_game -> inputHandler () -> onJoystickButtonEvent (dt);

delete (dt);

}

break;

case \_\_QGAMES\_KEYBOARDEVENT\_\_:

{

KeyBoardEventData\* dt = (KeyBoardEventData\*) evnt.data ();

assert (dt); // Event is something can be generated by the user...

\_game -> inputHandler () -> onKeyboardEvent (dt);

delete (dt);

}

break;

case \_\_QGAMES\_MOUSEMOVEMENTEVENT\_\_:

{

MouseMovementEventData\* dt = (MouseMovementEventData\*) evnt.data ();

assert (dt); // Event is something can be generated by the user...

\_game -> inputHandler () -> onMouseMovementEvent (dt);

delete (dt);

}

break;

case \_\_QGAMES\_MOUSEBUTTONEVENT\_\_:

{

MouseButtonEventData\* dt = (MouseButtonEventData\*) evnt.data ();

assert (dt); // Event is something can be generated by the user...

\_game -> inputHandler () -> onMouseButtonEvent (dt);

delete (dt);

}

break;

default:

break;

}

}

Cada uno de esos método realiza, por defecto, dos acciones: Una invocar al método equivalente del objeto QGAMES::InputHandler::Behaviour con el que se construyó la instancia de QGAMES::InputHandler y otra notificar el evento para que cualquier otro objeto que esté observando al handler pueda tomar, si acaso, las acciones pertinentes.

virtual void onJoystickAxisMoveEvent (JoystickMovementEventData\* dt)

{ \_behaviour -> onJoystickAxisMoveEvent (dt); // Uses the behaviour...

notify (Event (\_\_QGAMES\_MOVEJOYSTICKEVENT\_\_, dt)); }

Por tanto, la definición específica de qué hacer con cada evento en un juego concreto debe ser definida en una instancia hija de la clase QGAMES::InputHandler::Behaviour. QGAMES define un comportamiento estándar, QGAMES::InputHandler::StandardBehaviour, que puede ser más que suficiente para la mayor parte de los juegos.

## La Pantalla. Clase QGAMES::Screen

El dispositivo de salida por excelencia de todo juego es la pantalla, representado en QGAMES por la clase QGAMES::Screen.

Esta clase define los métodos encargados de dibujar cualquier forma básica manejada por QGAMES (una línea, un círculo, una forma, relleno o no, …). Dado que esas tareas están íntimamente ligadas con el hardware (con la tarjeta gráfica en este caso), la creación de pantallas se delegada en la clase QGAMES::GameImplementation, siguiendo así el mismo patrón de diseño al que se ha aludido anteriormente.

La clase QGAMES::Game define el método protegido virtual QGAMES::Screens createScreens (), responsable de crear todas las pantallas que intervienen en el juego. Como observará se devuelve un objeto tipo QGAMES::Screens que no es más que un mapa del tipo std::map <int, QGAMES::Screen\*>.

Este método es invocado automáticamente durante la inicialización del juego que, como recordará, se invoca además desde en el método exec () de QGAMES::Game antes de comenzar la ejecución del mainloop.

Un ejemplo de método sería el implementado en el módulo Advanced Arcade:

// ---

QGAMES::Screens QGAMES::AdvancedArcadeGame::createScreens ()

{

QGAMES::Screens r;

QGAMES::Screen\* scr = implementation () -> createScreen (

parameter (\_\_QGAMES\_GAMENAMEPARAMETER\_\_),

QGAMES::Position (0,0),

std::stoi (parameter (\_\_QGAMES\_GAMESCREENWIDTHPARAMETER\_\_)),

std::stoi (parameter (\_\_QGAMES\_GAMESCREENHEIGHTPARAMETER\_\_)),

std::stoi (parameter (\_\_QGAMES\_GAMESCREENXPOSPARAMETER\_\_)),

std::stoi (parameter (\_\_QGAMES\_GAMESCREENYPOSPARAMETER\_\_)));

assert (scr);

scr -> windowAtCenter ();

r.insert (QGAMES::Screens::value\_type (\_\_QGAMES\_MAINSCREEN\_\_, scr));

return (r);

}

El programador es responsable de crear las pantallas que empleará utilizando este método. Al menos una de ellas debe estar insertada en el mapa de respuesta con la clave \_\_QGAMES\_MAINSCREEN\_\_.

Las pantallas creadas por este método se almacenan automáticamente dentro de una variable privada de QGAMES::Game que puede ser accedida a través de los métodos:

* QGAMES::Screen\* QGAMES::Game::mainScreen (): Que recupera la pantalla principal, es decir aquella creada con la clave \_\_QGAMES\_MAINSCREEN\_\_.
* QGAMES::Screen\* QGAMES::Game::screen (int id): Que recupera una pantalla por su id.
* bool QGAMES::Game::existScreen (int id): Que verifica si una determinada pantalla (por su id clave) existe o no.

Dentro del método createScreens (), para crear cada pantalla específica, hay que invocar al método definido dentro de QGAMES::GameImplementation: virtual QGAMES::Screen\* QGAMES::GameImplementation::createScreen (const std::string& t, const QGAMES::Position& pos, int w, int h, int pX, int pY) , que recibe los siguientes parámetros:

* **t**: Título de la pantalla
* **pos**: Posición de la esquina superior izquierda de la pantalla.
* **w**, **h**: Ancho y alto en pixels de la pantalla.
* **pX**, **pY**: Posición en pixles de la esquina superior izquierda de la pantalla.

## Las cámaras. Clase QGAMES::Camera

La pantalla es sólo el lugar donde se dibujan los elementos del juego, pero ¿Bajo que condiciones se dibujarán? Esas condiciones se basan en los datos proporcionados por un objeto cámara del tipo QGAMES::Camera.

Toda pantalla debe tener al menos asociada una cámara. Es habitual que la cámara por defecto se instancie y añada en el propio constructor de la pantalla. Si en el momento de dibujar, no hubiera creada y asociada ninguna cámara a la pantalla, no sucedería nada. La cámara determina desde que posición se están observando los elementos del juego y por tanto como éstos deben ser proyectados en la propia pantalla, que actuaría, por así decirlo, a modo del cristal de la cámara.

Pantalla y cámara son muy importantes en el funcionamiento de un juego. Profundizaremos evidentemente, más adelante en las características y forma de controlar una y otra.

QGAMES define un tipo de cámara básico que es útil para la mayor parte de los juegos.

## La estructura básica de un juego

A estas alturas ya tenemos los conocimientos básicos para entender como construir nuestro primer “Hello World”:

Es necesario definir una clase hija de QGAMES::Screen que defina e implemente los métodos necesarios para dibujar las formas básicas de cualquier juego, utilizando la librería hardware / software que queramos. QGAMES viene equipada con clases específicas para trabajar en el mundo 2D, por ejemplo, con SDL.

Hay que definir una clase hija de QGAMES::InputHandler que se encargue de interpretar los eventos recibidos por el sistema, nuevamente utilizando las librerías hardware / software adecuadas. QGAMES viene, nuevamente, equipada con clases específicas para utilizar SDL en esta labor.

Lo mismo pasa con el mundo de las formas. SDL, por ejemplo, identifica en memoria los mapas de bits con la clase SDL\_texture. Igualmente QGAMES ya incluye clases adecuadas para representar formas bajo SDL.

Idem para los sonidos. En este caso, QGAMES trabaja tanto con SDL como con FMOD.

Se necesita definir clase que herede de QGAMES::GameImplementation que se ocupe de crear los builders asociados a aspectos cercanos al manejo del software: Forms, Sounds, InputHandlder y Screen. QGAMES nuevamente incorpora ya una implementación propia para SDL / FMOD.

Como resumen, para trabajar con SDL, basta con usar una instancia tipo SDLGame que hereda de QGAMES::GameImplementation y pasarla como parámetro al constructor de nuestro juego. Clase que, además, está preparada para usar o no FMOD o SDL como generador de sonidos gracias a un parámetro que recibe en tiempo de construcción. Esa clase está definida de la siguiente manera en QGAMES:

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;

};

Hay que definir una clase hija de QGAMES::InputHandler::Behaviour que implemente lo que hacer en el juego ante cada tipo de evento; es dedir como éstos afectan a nuestro personaje, o a las cosas que le rodean.

Hay que definir diferentes clases que hereden de QGAMES::Entity implementando las características y lógica de cada entidad que interviene en el juego: Personajes principales, monstruos, etc.

Hay que definir un fichero con las características de las entidades del juego:

…

<!-- Male entity used when introducing the game -->

<Entity id="50000">

<Forms>

<!-- Ninja male looking to the right -->

<Form id="50050"/>

</Forms>

<Movements>

<!-- This movement means totally stopped -->

<Movement id="10000"/>

</Movements>

<Animations>

<!-- When paused -->

<Animation id="0" type="0">

<Attributes>

<Attribute id="FORM" value="50050"/>

<Attribute id="INITIALASPECT" value="80"/>

<Attribute id="FINALASPECT" value="90"/>

<Attribute id="FRACTIONTOCHANGE" value="5"/>

<Attribute id="LOOP" value="1"/>

</Attributes>

</Animation>

</Animations>

<States>

<!-- Running with no movement -->

<State id="0" animation="0" movement="10000"/>

</States>

</Entity>

…

Hay que definir un fichero con las formas que dichas entidades manejan. En el caso del ejemplo anterior la 50050.

…

<Form id="50050" name="NinjaMaleRight" file="forms/NinjaMaleRight.png" width="946" height="878"

fSpacing="0" fMargin="0" fWidth="1" fHeight="1" aColor="255" nFrames="110">

<Frame id="000" fPosX="005" fPosY="005" fWidth="060" fHeight="90" xoffset="08" yoffset="03" width="060" height="90"/>

<Frame id="001" fPosX="070" fPosY="005" fWidth="058" fHeight="90" xoffset="08" yoffset="02" width="058" height="90"/>

<Frame id="002" fPosX="133" fPosY="005" fWidth="062" fHeight="91" xoffset="02" yoffset="01" width="062" height="91"/>

<Frame id="003" fPosX="200" fPosY="005" fWidth="099" fHeight="90" xoffset="08" yoffset="02" width="099" height="90"/>

<Frame id="004" fPosX="304" fPosY="005" fWidth="100" fHeight="90" xoffset="07" yoffset="03" width="100" height="90"/>

…

</Form>

…

Lo mismo con los movimientos. En este caso el 10000:

…

<!-- Ther very basic movement is a NO Movement -->

<Movement id="10000"/>

…

Hay que definir al menos un builder para las entidades del juego que, en función del id que haya leído en el fichero instancia un tipo u otro de entidad definida.

Lo mismo para el builder de movimientos.

Y, por último has que definir una clase que herede de QGAMES::Game y que implemente todos los métodos para crear los builders anteriormente definidos y que instancie en su constructor un objeto del tipo QGAMES::GameImplementation definido arriba.

¡Luego basta con invocar al método exec () de esa clase y…voila! El juego empezará:

int main (int argc, char \*argv [])

{

MyGame game;

game.exec ();

return (0);

}

Como puede observar parece que sólo nos tenemos que ocupar mayoritariamente de definir la lógica del juego.