Escuela Superior de Informática
Universidad de Castilla-La Mancha

CURSO DE EXPERTO EN DESARROLLO DE VIDEOJUEGOS

TRABAJO FIN DE CURSO

RAMSOM

JULIO 2013

Escuela Superior de Informática
Universidad de Castilla-La Mancha

CURSO DE EXPERTO EN DESARROLLO DE VIDEOJUEGOS

TRABAJO FIN DE CURSO

RAMSOM

Blanco León, Matías
Moreno Montes, Diego

JULIO 2013

CURSO DE EXPERTO EN DESARROLLO DE VIDEOJUEGOS

Calificación Trabajo Fin de Curso

CONVOCATORIA: Julio 2013

TÍTULO DEL PROYECTO: RAMSON

AUTORES (ORDEN ALFABÉTICO):

BLANCO LEÓN, MATÍAS

MORENO MONTES, DIEGO

TRIBUNAL:

Presidente: _____

Vocal1: _____

Vocal2: _____

Secretario: _____

FECHA DE DEFENSA: _____

CALIFICACIÓN: _____

PRESIDENTE	**VOCAL 1**	**VOCAL 2**	**SECRETARIO**
Fdo.:	Fdo.:	Fdo.:	Fdo.:

Matías Blanco León y Diego Moreno Montes
Ciudad Real – España

Resumen

El objetivo principal de este juego es abarcar todas o casi todas las cosas que se han dado a largo de del Curso de Experto en Desarrollo de Videojuegos 2012/2013 [7].

Para cumplir este objetivo se ha decidido hacer un juego de táctica en tiempo real completo, para el cual nos hemos tenido como inspiración juegos tipo Commandos [1] y Frozen Synapse [2]; aunque este último título es el que más nos ha inspirado para realizar nuestro proyecto.

La temática de nuestro juego se basa en el que Jon Doe (que es el héroe de nuestro juego) tiene que adentrarse dentro de un edificio, controlado por una banda, para liberar a los rehenes que tienen cautivos; sin que nuestro héroe muera en el intento.

La implementación de nuestro juego se basa en tres grandes pilares, sobre los que ha rotado el curso, los cuales son:

- OGRE3D [1]: Es el motor de renderizado de gráficos orientado a objetos que se ha usado en el curso y el cual hemos utilizado.
- Bullet [4]: Es el motor de física, que junto con OgreBullet [5], nos permite dar propiedades y comportamientos físicos realistas a nuestro juego.
- Blender [6]: Es la herramienta con la que se ha modelado, animado y creado todo y cada uno de los gráficos que se han utilizado en el juego.

La principal característica de estas tres herramientas es que son software libre y el uso de herramientas de esta característica ha sido uno de los pilares del curso.

Índice general

Resumen IX
Índice general XI
Índice de figuras XIII
Índice de listados XV
Listado de acrónimos XVII
1. Introducción 1
1.1. Estructura del documento. 2
2. Objetivos 3
2.1. Objetivo general. 3
2.2. Objetivos específicos. 3
3. Arquitectura de la solución. 7
3.1. Componentes implementados. 7
3.1.1. Personajes del juego. 7
3.1.2. Inteligencia Artificial. 10
3.1.3. Estados del Juego. 15
3.1.4. Configuración del Juego. 17
3.1.5. Controlador de las cámaras. 19
3.1.6. Registro de Records. 20
3.1.7. Lógica del Disparo. 22
3.1.8. Visualización de los niveles de vida. 24
3.2. Técnicas gráficas. 25
3.2.1. Animación de los personajes. 25
3.2.2. Precalculo de iluminación. 26
3.3. Herramientas. 27
4. Análisis de costes. 29
4.1. Costes humanos. 29
5. Manual de Usuario. 31
5.1. Interfaz del juego. 31
5.2. Pantalla del juego. 33
5.3. Ejecución del juego. 39
6. Conclusiones y propuestas. 41

6.1. Conclusiones. 41
6.2. Líneas futuras de desarrollo e investigación. 45
Bibliografía 47

Índice de figuras

Figura 3.1 – Diagrama de clases de los personajes 8
Figura 3.2 – Imagen del enemigo 9
Figura 3.3 – Imagen de Jon Doe 9
Figura 3.4 – Imagen de Rehen 10
Figura 3.5 – Angulo de visión del enemigo 11
Figura 3.6 – Textura de la visión del enemigo 12
Figura 3.7 – Maquina de estado del enemigo 13
Figura 3.8 – Diagrama de clases del gestor de los estados del juego 16
Figura 3.9 – Estados del juego 16
Figura 3.10 – Diagrama de clases de GameConfig 18
Figura 3.11 – Diagrama de clases de XMLCharger 18
Figura 3.12 – Diagrama de clases de CameraController 19
Figura 3.13 – Visualización de las cámaras en la escena 20
Figura 3.14 – Diagrama de clases de Records 21
Figura 3.15 – Pantalla de Records 22
Figura 3.16 – Barra de vida del enemigo 24
Figura 3.17 – Barra de vida del Jon Doe 24
Figura 3.18 – Creación de animaciones en Blender 26
Figura 3.19 – Pieza 1 antes del precalculo 27
Figura 3.20 – Pieza 1 despues del precalculo 27
Figura 5.1 - Pantalla de records 31
Figura 5.2 - Pantalla de records 32
Figura 5.3 - Pantalla de créditos 32
Figura 5.4 - Pantalla de carga 33
Figura 5.5 - Pantalla del juego 34
Figura 5.6 - Estado del personaje 35
Figura 5.7 - Cruce de disparos 35
Figura 5.8 - Pantalla de pausa 36
Figura 5.9 - Barra de vida de los enemigos 36
Figura 5.10 - Muerte del enemigo 37
Figura 5.11 - Liberación de rehén 37
Figura 5.12 - Pantalla de Game Over 38
Figura 5.13 - Pantalla de TO BE CONTINUED… 38

Índice de listados

Listado 3-1 – Fragmento de la actualización del disparo. 23
Listado 3-2 – Fragmento de la creación del disparo. 23

Listado de acrónimos

GCC	GNU Compiler Collection
GDB	GNU Project Debugger
GNU	GNU is Not Unix
GUI	Graphical user interface
IA	Inteligencia Artificial
OGRE3D / OGRE	Open Source 3D Graphics Engine
SDL	Simple DirectMedia Layer

Capítulo 1.

1. Introducción

1.1. Estructura del documento.

Durante el transcurso del curso hemos aprendido diferentes técnicas (programación, inteligencia artificial, graficas, etc..) y el manejo de diferentes herramientas y librerías centradas para el desarrollo de videojuegos.

El conjunto de todo este conocimiento y la aplicación del mismo nos han permitido realizar nuestro juego; obteniendo un resultado del mismo bastante aceptable.

El juego se basa en conseguir rescatar a un número de rehenes, los cuales están siendo vigilados por enemigos a los que tendremos que abatir.

La mecánica de nuestro juego será algo similar al mítico juego Commandos de Pyro Studios, llevando a nuestro héroe por unos escenarios donde tendremos que localizar a los rehenes y se nos presentarán obstáculos y enemigos que tendremos que enfrentarnos.

El juego estará compuesto por un único nivel en el cual el escenario se creará aleatoriamente, por lo que cada partida será una nueva aventura. Además las dimensiones del escenario podrán ser configurables.

El desarrollo del proyecto se ha realizado íntegramente en C++ bajo entornos Linux y Windows, utilizando varias librerías, de las que cabe destacar:

- ✓ Boost [8] en su versión 1.53. Que es un conjuntos de librerías que nos ofrece librerías para el manejo de xml, hilos, etc..
- ✓ Bullet [4] en su versión 2.81 rev2613. Es la librería la cual se encarga de manejar el motor de física del juego.
- ✓ Ogre3D [1] en su versión 1.8.1. Es la librería que implementa el motor de renderizado del juego.
- ✓ OgreBullet [5]. Es la librería que nos permite manejar Ogre3D junto con Bullet.
- ✓ SDL [9] en su versión 1.2. Es la librería que nos ha permitido agregar a nuestro juego tanto música como sonidos FX.

Todos y cada uno de los modelos gráficos han sido modelados y animados en Blender [6] y exportados posteriormente para su uso con Ogre3D.

1.1. Estructura del documento.

Este documento está compuesto por cinco capítulos. El segundo capítulo contiene los objetivos funcionales y no funcionales que han tenido en cuenta para realizar este trabajo.

En el capítulo tercero se describe toda la arquitectura desarrollada en este juego. Describiendo la estructura del código desarrollado representado mediante diagramas de clases, patrones de diseño utilizados, metodologías utilizadas, herramientas, etc…

El capítulo cuarto se trata de desglosar las tareas que ha desarrollado cada uno de los componentes del grupo de trabajo.

El manual de usuario estará descrito en el capítulo de quinto.

Y por último, en el capítulo sexto se reflejaran las conclusiones que hemos obtenido durante la realización del proyecto. Y se plantearan unas posibles futuras líneas de trabajo que puedan complementar el trabajo realizado hasta la fecha.

Capítulo 2.

2. Objetivos

2.1. Objetivo general.

2.2. Objetivos específicos.

2.1. Objetivo general.

El objetivo del proyecto es **crear un objetivo completamente funcional, utilizando todo o casi todo lo aprendido durante la realización del curso**.

Para ello se desea crear un juego completamente funcional que contenga las siguientes partes: Una pantalla de inicio o de menú, una pantalla de records, una pantalla de títulos, la pantalla de juego y pantalla de pausa.

Enfocado a la temática del juego que se ha planteado en el capítulo primero.

2.2. Objetivos específicos.

Se plantea una serie de objetivos específicos para llegar al objetivo general durante la realización del proyecto:

- ✓ **Implementación de cada una de las pantallas que conforman el juego y la interacción entre cada una de ellas.** El juego debe de tener una pantalla principal la cual estará formada un menú que nos de paso al resto de pantallas (Juego, Records, Títulos y Salir). En la pantalla de records se mostraran las mejores puntuaciones realizadas en el juego; teniendo en cuenta el tiempo en el

que se ha tardado en rescatar a todos los rehenes y el número de rehenes; junto con la fecha en la que se ha obtenido dicho registro. En la pantalla de Títulos se mostrará los nombres de los componentes que han desarrollado este proyecto. Y por último la pantalla de juego que es la más importante porque es la contiene toda la mecánica del juego en sí.

- ✓ **Diseño y modelado de escenario.** Se han diseñar y modelar gráficamente cada una de las piezas que van a conformar nuestro escenario. Así como la integración de cada una de las mismas en la escena del juego.

- ✓ **Inserción de personajes en la escena.** Creación o búsqueda de todos los recursos gráficos para los personajes de nuestro juego (Hero, Enemigos y Rehenes), así como la personalización de los mismo, animación e inserción en la escena.

- ✓ **Creación de todos los diseños de las pantallas.** Esto incluye la creación de fondos, diseño de menús, etc…

- ✓ **Implementación de los movimientos de los personajes del juego.** El personaje se debe de mover de una forma natural a lo largo de la escena del juego y poder interactuar con los diferentes componentes de la misma.

- ✓ **Inteligencia artificial de los enemigos.** Los enemigos deben de tener un comportamiento autónomo y ser capaces de actuar de una cierta manera dependiendo de la circunstancia en la que se encuentren.

- ✓ **Lógica de rescate de rehenes.** Se debe implementar el método por el cual el Jon Doe pueda liberar cada uno de los rehenes.

- ✓ **Mecanismos para mostrar el estado de cada uno de los personajes.** Se deben crear mecanismos tanto gráficos como sonoros que nos indique el estado actual de cada uno de los personajes; como pueden ser: el nivel de vida, el cambio de estado de los enemigos, etc…

- ✓ **Mecanismos de física para que todos los componentes de la escena interactúen correctamente.** Cada uno de los componentes de la escena tiene que poder interactuar uno con otros de una forma natural (cálculo de colisiones, etc…).

- ✓ **Mecanismo de disparo para el héroe y los enemigos.** Se debe implementar el mecanismo por el cual tanto el héroe como los enemigos pueden dispararse unos a otros con el fin de dañarse mutuamente.

- ✓ **Mecanismos para la eliminación de los personajes de la escena.** Se debe de implementar mecanismos para hacer que los personajes desaparezcan de la escena, ya sea porque hayan muerto (como en el caso del hero y de los enemigos) o porque sean liberados (como el caso de los rehenes).

- ✓ **Sistema de configuración el juego.** El juego debe ser configurable y se deben implementar mecanismos para que el juego pueda configurarse por parte del usuario y que se pueda cargar dicha configuración.

- ✓ **Manejo de records.** Se debe de crear una herramienta que se encargue de almacenar las mejores puntuaciones del juego y luego de mostrarlas en la pantalla de records.

- ✓ **Minimapa del juego.** En la pantalla del juego aparecerá un minimapa que mostrará al jugador una perfectica más alejada y general de la escena.

- ✓ **Manejo de la cámara de una forma natural y que siga al personaje.** La cámara del juego, así como la del minimapa, deben de seguir al personaje a lo largo de la escena de una forma natural.

- ✓ **Inserción de recursos sonoros al juego.** Cada una de las partes del juego debe de tener una música de fondo y la mecánica del juego debe de ir acompañando por diferentes efectos sonoros que den más realismo al mismo.

- ✓ **Que el juego sea multisistema.** El juego se debe de poder ejecutar tanto en sistemas Linux como en sistemas Windows.

Capítulo 3.

3. Arquitectura de la solución.

3.1. Componentes implementados.

3.2.Técnicas gráficas.

3.3. Herramientas.

3.1. Componentes implementados.

En este apartado vamos a describir las partes más importantes que hemos implementado durante la creación de nuestro juego.

Para obtener más información detallada sobre los componentes implementados se puede consultar la documentación (generada con Doxygen [20]) que se encuentra en el directorio **doc/devel**, la página principal es **index.html**.

3.1.1. Personajes del juego.

En nuestro juego tenemos el siguiente diagrama de clases para los personajes empleados.

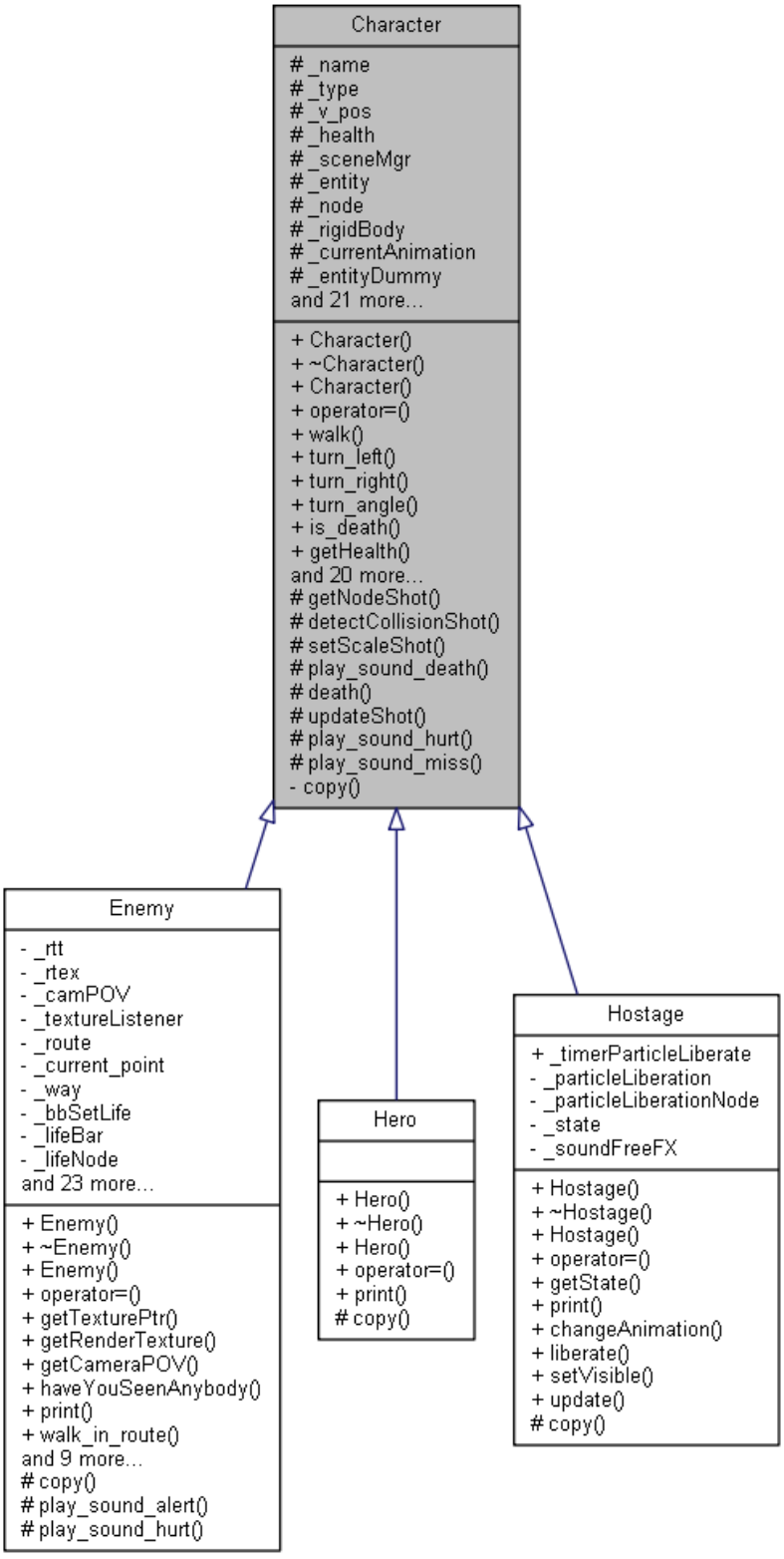

Figura 3.1 – Diagrama de clases de los personajes

3.1.1.1. Clase Character.

La clase Character es la clase principal de manejo de los personajes, ya que implementa toda la funcionalidad común para nuestros personajes, es decir, para

funcionalidades como caminar, girar, atributos del personaje como nivel de vida, etc, etc...

3.1.1.2. Clase Enemy.

La clase enemigo hereda toda la funcionalidad de la clase Character añadiendo aquellas funciones específicas del enemigo, como son:

- Funcionalidad de detección del héroe en el campo de visión.
- Caminar por ruta de forma autónoma.
- Máquina de estados del enemigo.
- Billboards para la visualización del nivel de vida del enemigo en el escenario.
- Etc, etc...

Figura 3.2 – Imagen del enemigo

3.1.1.3. Clase Hero.

Al igual que pasa con la clase Enemy, ésta hereda toda la funcionalidad de la clase Character añadiendo algunas funciones particulares del héroe, como son:

- Objeto Dummy (Cubo) para la detección por parte de los enemigos.

Figura 3.3 – Imagen de Jon Doe

3.1.1.4. Clase Hostage.

Aunque hereda toda la funcionalidad de la clase Character no hace uso de mucha funcionalidad, pero sí de parte de ella, como ejemplo usa todos los objetos relacionados con Ogre y Bullet, mientras que luego no usaría todo lo relacionado al movimiento del personaje, ya que éste permanece quieto. Nuestro rehén, aun estando quieto, tiene una animación cíclica, que consiste en agitar los brazos solicitando ayuda para su rescate. Como funcionalidad añadida tendríamos:

- Tema de sistema de partículas para la liberación de éste.

Figura 3.4 – Imagen de Rehen

3.1.1.4.1. Lógica de Rescate aplicada a los Rehenes.

Para el rescate de los rehenes hemos seguido el siguiente procedimiento:

1. Una vez localizado a nuestro rehén en la pantalla.
2. El héroe colisiona con nuestro rehén.
3. Nuestro rehén desaparece de la pantalla, una vez rescatado.
4. Empleo de un sistema de partículas para dar un efecto vistoso al éxito conseguido.

3.1.2. Inteligencia Artificial.

Dentro de la Inteligencia Artificial que hemos dotado a nuestro juego, vamos a desarrollar nuestros dos principales funcionalidades usadas. Las cuales son:

- Visión artificial del enemigo
- Máquina de estados del enemigo

3.1.2.1. Visión del Enemigo.

Para la implementación de la visión artificial del enemigo hemos usado una funcionalidad proporcionada por nuestro motor usado (OGRE) llamada ***RenderTargetListener***, la cual nos permite recibir notificaciones por eventos

relacionados con el objeto ***RenderTarget***. El objeto RenderTarget puede ser desde una escena, una pantalla o una textura, que para nosotros era lo que nos interesaba, ya que nosotros necesitábamos un ***Render a Textura***. Vamos a explicar ahora que es eso de Render a Textura.

3.1.2.1.1. Render a Textura.

Antes de explicar que es el render a textura, vamos a dar un contexto a su uso.

¿Por qué usamos ésta técnica?

Esta técnica nos proporciona un mecanismo de visión artificial fácil de implementar, ya que consiste en hacer una "foto" de la escena que estaría viendo nuestro enemigo y después interpretar ésta foto para determinar si el enemigo está viendo a nuestro héroe o no.

¿Cómo se ha implementado ésta técnica?

Vamos por partes; usando una cámara que colocamos delante de nuestro enemigo tenemos una visión panorámica de lo que tiene delante, tal y como se puede en la Figura 3.5.

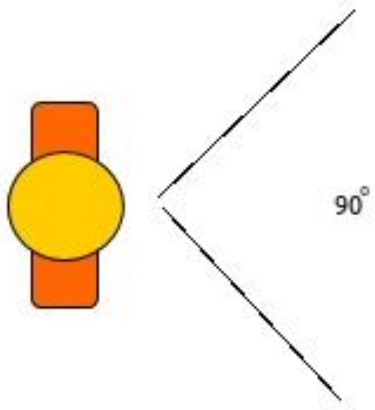

Figura 3.5 – Angulo de visión del enemigo

Nos creamos una textura "manual", de la cual usamos su objeto interno *render a textura* para asignarle nuestra cámara. De manera, que ahora tenemos vinculada la cámara a éste objeto. Posteriormente, a éste objeto render a textura le añadimos como listener, nuestro RenderTargetListener, de manera que cada vez que se renderiza un frame obtenido por la cámara nos pasa a nuestro listener para que nosotros lo tratemos. Y ahora viene lo mejor, ya que con éste listener que se ejecuta a cada frame, nosotros realizamos el siguiente proceso:

I. Antes de que se calcule el frame (con nuestro listener tenemos dos callbacks antes y después de calcular el frame de la escena). Nosotros tenemos dos objetos en el héroe, uno visible (el modelo de nuestro héroe) y otro invisible (un cubo blanco);

el objeto invisible esta adjunto al héroe, de manera que cualquier movimiento asociado al modelo del héroe aplica al cubo. Pues bien, nosotros ponemos a invisible el modelo del héroe y a visible nuestro cubo completamente blanco.

II. Hacemos una foto de éste frame mediante el uso del objeto textura, el cual nos proporciona un método para volcar ésta a una imagen.

III. Con la imagen obtenida muestreamos los píxeles, de manera que aquellos píxeles que son blancos los contamos. Y mediante el uso de un umbral, tenemos nuestro "*detector de héroes*". (Ver la Figura 3.6).

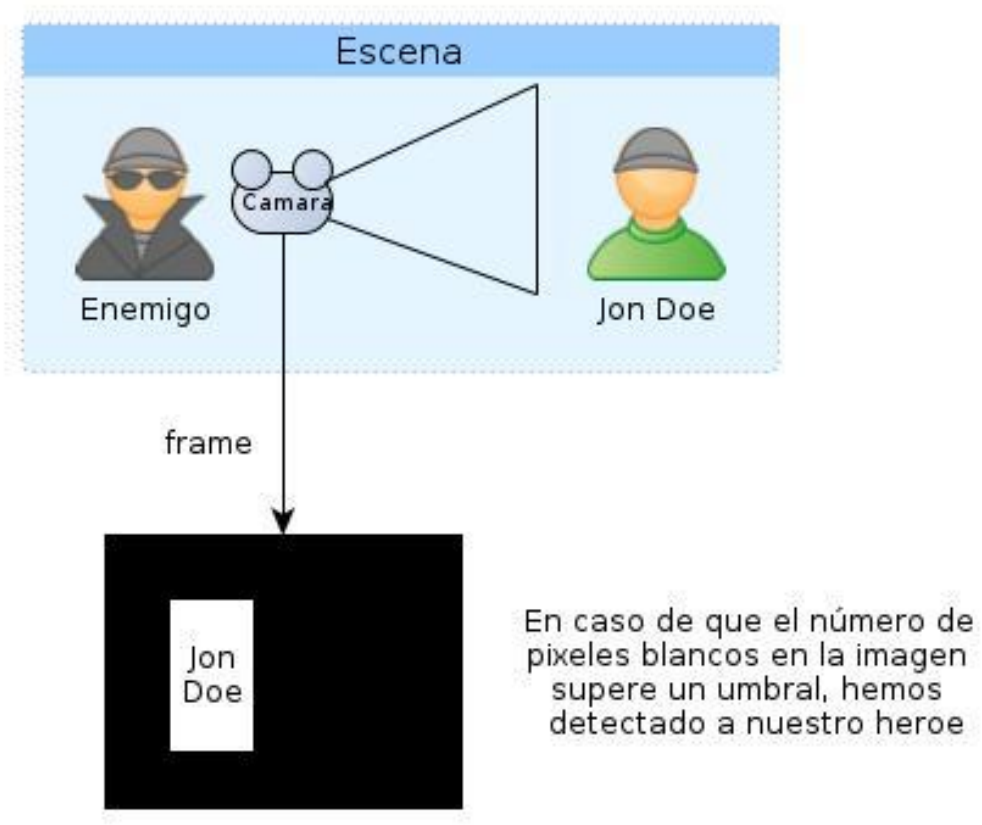

Figura 3.6 – Textura de la visión del enemigo

3.1.2.2. Máquina de Estados.

Para la máquina de estados usada para nuestros enemigos hemos tenido en cuenta los siguientes estados:

Figura 3.7 – Maquina de estado del enemigo

Cada uno de nuestros enemigos implementa una pequeña máquina de estados, de manera que en función del estado en que se encuentre obrará de una manera u otra.

Vamos a desglosar cada uno de los estados y lo que implica estar en ellos.

3.1.2.2.1. Estado WATCHING.

Este estado es el estado "natural" de nuestro enemigo, es decir, nuestro enemigo, mientras no vea a nuestro héroe, irá por nuestro escenario siguiendo una ruta (Ver apartado de Rutas del Enemigo) vigilando por si aparece nuestro héroe.

En el momento de que nuestro enemigo aviste al héroe, pasaríamos al estado ALERT. Además, en éste caso, almacenaremos la posición donde vimos al enemigo para poder usarla en el siguiente estado.

3.1.2.2.2. Estado ALERT.

En éste estado quiere decir que nuestro enemigo a entrado en modo alerta, ya que nos ha visto y por lo tanto, si una vez que estamos dentro de éste estado persistimos dentro de su campo de visión pasaríamos al estado SHOOTING.

En caso de habernos perdido de vista durante éste estado, pueden ocurrir dos cosas:

- Si nos ha perdido de visto antes de unos 2 segundos pasa a modo WATCHING, ya que no ha sido tiempo suficiente como para asegurar que lo que hemos visto sea el heroe.
- Pero si nos ha perdido de vista despues de haber permanecido delante de él más de éstos 2 segundos, entonces pasa al estado CHASING. Almacenando el instante temporal del momento en cuanto arrancamos la persecución.

3.1.2.2.3. Estado SHOOTING.

Al pasar a éste estado, pueden ocurrir dos cosas:

- Que el enemigo nos dispare, ya que nos mantenemos en su campo de visión, de manera que si no estuviésemos orientados hacia nuestro héroe, reorientaríamos a nuestro enemigo hacia él para que el disparo fuese lo más certero posible. También entre cada disparo controlamos que no supere un tiempo inferior a 2 segundos entre cada uno.
- Si estamos en éste estado y desaparecemos de su campo de visión, entonces pasa al estado CHASING. Almacenando el instante temporal del momento de iniciar la persecución.

3.1.2.2.4. Estado CHASING.

En éste estado, partiendo de la premisa de que si hemos entrado en éste estado es porque inicialmente nos vio el enemigo y hemos desaparecido de su campo de visión; tenemos las siguientes casuísticas:

- Que el enemigo nos vea y por lo tanto, si persistimos en su campo de visión pasamos al estado ALERT.
- Que el enemigo no nos vea, pero:
    1) Puede que no haya llegado al punto donde nos vio por última vez, así que le hacemos ir hasta el punto último donde nos vio. Como éste desplazamiento provoca que nuestro enemigo se salga de la ruta, realizamos un almacenamiento de puntos para poder usar este conjunto de

puntos, como puntos de retorno a la ruta original. Ver la 1ª Nota Adicional al pie para más información.

2) Si estamos en el punto último donde nos avistó, nos detendremos y entonces realizaremos un barrido de 360º en busca del héroe, por si lo podemos volver a ver.
3) Si después de haber realizado el barrido de 360º no vemos a nuestro héroe, usando nuestro conjunto de puntos almacenados como puntos de retorno a la ruta original, volveremos a la ruta y cambiaremos al estado WATCHING, para que prosiga por la ruta.

1ª Nota Adicional

Puede ocurrir que los enemigos en el momento de iniciar la persecución vayan por zonas del escenario que luego, por sí solo, no sabría cómo obrar para volver a su ruta original. Para esto, se implementó el almacenamiento de puntos de recuperación (a modo de "migas de pan"), los cuales se van almacenando cada poco tiempo y con cada desplazamiento del enemigo por el escenario mientras la persecución persista.

2ª Nota adicional

Hemos aplicado para el instante en el cual nuestro enemigo recibe un disparo un principio de Acción/Reacción, es decir, si nuestro enemigo no nos está viendo y de manera "sibilina" le disparamos por la espalda, éste reaccionará a nuestro disparo girándose e intentando localizarnos, para pasar al estado ALERT para en el caso de mantenemos en su campo de visión poder pasar al estado SHOOTING y plantar batalla.

3.1.3. Estados del Juego.

Partiendo de nuestra clase raíz RansomApp, la cual contiene una instancia de la clase AppStateManager, la cual es el gestor de estados de nuestro juego.

La siguiente imagen muestra el diagrama de clases de nuestra clase principal del juego:

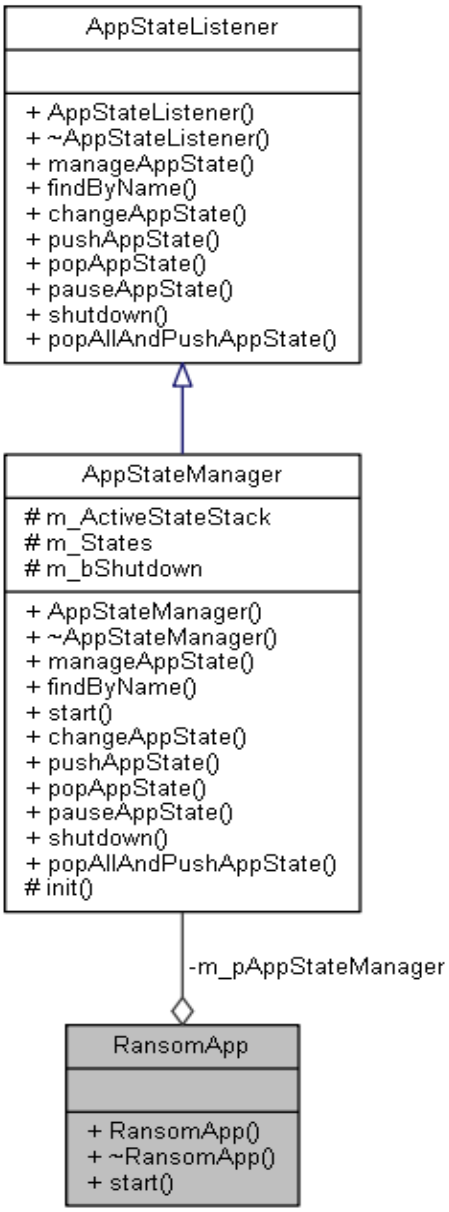

Figura 3.8 – Diagrama de clases del gestor de los estados del juego

Nuestro juego se basa en un diagrama de estados implementado por el sistema de estados de la aplicación dado por ***AdvancedOgreFramework*** y el cual sigue el siguiente diagrama:

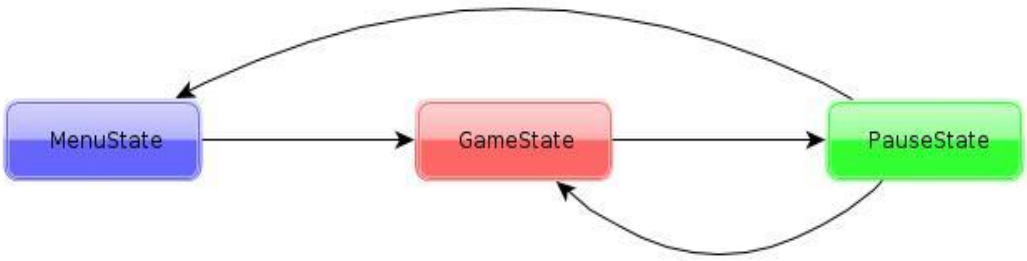

Figura 3.9 – Estados del juego

Mediante el uso del manager antes mencionado (***AppStateManager***) podemos cambiar entre los tres estados de nuestro juego. Los cuales son:

- MenuState
- GameState
- PauseState

3.1.3.1. Estado MenuState.

Este estado conlleva la pantalla de menú principal de nuestro juego, donde encontraremos la pantalla de créditos del juego (botón Credits), pantalla de records (botón Best Times) y el menú principal del juego. Desde éste estado sólo podríamos pasar al estado ***GameState***, no teniendo opción de poder pasar al ***PauseState***.

Nota adicional: Para el desarrollo de ésta clase se ha usado una funcionalidad aportada por Ogre llamada ***SDKTrays***, la cual nos proporciona una GUI simple pero eficaz para nuestro sistema de menus. La cual se basa en el uso de un conjunto de elementos gráficos dados (agrupados en un contenedor ZIP) para los botones, ventanas, dialogos, etc... relacionados con la GUI del juego.

3.1.3.2. Estado GameState.

En éste estado tenemos la pantalla principal del juego, donde se realiza toda la acción al fin y al cabo. Desde éste estado sólo podríamos pasar al estado ***PauseState***.

3.1.3.3. Estado PauseState.

En éste estado nos aparece un menú el cual nos dará la opción de volver al juego (Resume Game) pasando al estado GameState, volver al menú principal (Return to Main Menu) pasando al estado MenuState o salir completamente del juego (Exit Game).

3.1.4. Configuración del Juego.

Para la configuración del juego usamos la carga de un fichero de configuración vía XML, que se haya en la carpeta config del juego. Donde establecemos:

- Posición inicial del héroe en el escenario.
- Número de piezas que componen el escenario. Nuestro escenario lo compondrán piezas creadas en Blender, indicando, a modo de tabla, el número de filas y columnas que la componen.
- Por cada pieza:
  - Nombre del fichero MESH asociado a la pieza.
  - Posición de la pieza dentro de la escena.
  - Ancho y Alto de la pieza.

- Rehenes y posición (relativa a la pieza y no global) de éstos en la pieza.
- Rutas de los enemigos dentro de la pieza. Por ende, a cada ruta le corresponderá 1 enemigo, luego habrá tantas rutas como enemigos.
- Por cada ruta:
  - Tendremos puntos que componen la ruta.

Para el almacenamiento de ésta configuración XML usamos una clase llamada GameConfig, la cual sería:

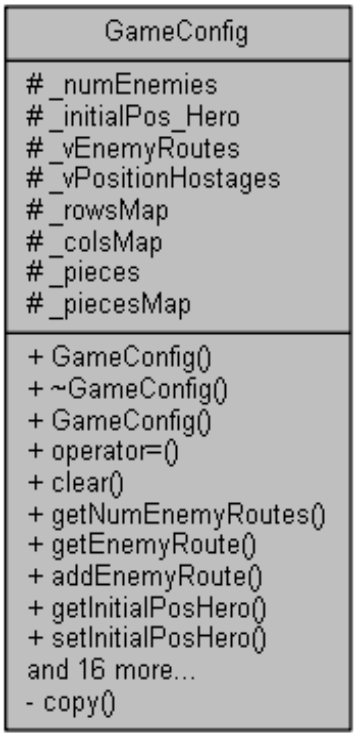

Figura 3.10 – Diagrama de clases de GameConfig

Para el volcado de los datos XML a la clase GameConfig hacemos uso de la clase XMLCharger, la cual a su vez usa las librerías boost (concretamente la clase ptree) para la lectura y parseado del fichero XML. La cual sigue la siguiente estructura:

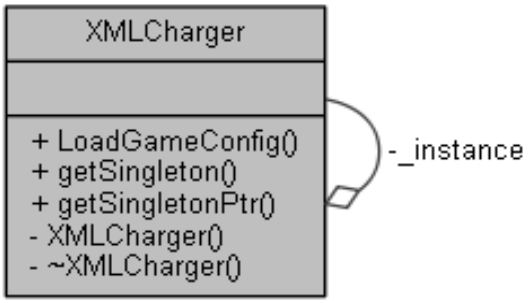

Figura 3.11 – Diagrama de clases de XMLCharger

3.1.5. Controlador de las cámaras.

Para el manejo de las cámaras relacionadas con nuestro juego, nos hemos creado un controlador de cámaras, el cual se encuentra implementado en nuestra clase llamada CameraController, ver el siguiente diagrama:

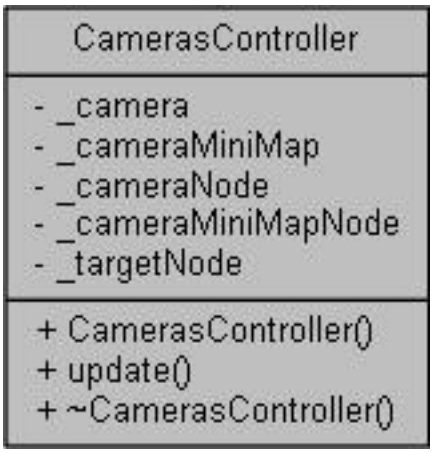

Figura 3.12 – Diagrama de clases de CameraController

3.1.5.1. Cámara principal del Juego.

Dentro de nuestro juego usamos un sistema de cámara, en visión cenital sobre nuestro heroe, basado en un movimiento natural que realiza un seguimiento de nuestro personaje, pero no estando ligado a éste de manera férrea, sino que permite un desplazamiento suave sin llegar a tener la sensación brusca del arranque/parada de nuestro personaje al andar, sino que, si nuestro personaje, por ejemplo, se para entonces la cámara va parando poco a poco de forma gradual.

En la definición de la cámara (camera) podemos apreciar que existe adjunto un nodo de escena (cameraNod) que es el que usaremos para provocar el movimiento de la cámara por el escenario.

Para el objetivo de la cámara se ha usado como referencia el nodo _targetNode, que mediante el uso del método AutoTracking hacemos que la cámara esté enfocando en todo momento a nuestro héroe.

3.1.5.2. Cámara para el minimapa.

Esta cámara realiza el mismo movimiento que la cámara principal, es decir, se aplica todo lo dicho a la cámara principal pero con la salvedad de que se encuentra a una distancia mucho más lejana sobre nuestro héroe, para tener una visión más global del escenario.

En la definición de la cámara (cameraMiniMap) podemos apreciar que existe adjunto un nodo de escena (_cameraMiniMapNod) que es el que usaremos para provocar el movimiento de la cámara por el escenario.

Lo que captura esta cámara es mostrado en un Overlay que se haya en la parte de arriba a la derecha de la pantalla del juego. Esto se hace en el método *CreateMiniMap()* de la clase *GameState*, este método lo que se hace es una textura (TexturePtr de OGRE) a la cual se le asocia un render a textura (al render a textura (RenderTexure de OGRE) se le añade un *Listener* (nuestra clase MiniMapTextureListener, que se basa en el patrón de Observador) que se va a encargar en el prerender de ocular los modelos de los personajes y visualizar los cubos que representa a cada uno de ellos; y en el postrender vuelve a dejar la escena tal y como estaba), después crea un material con la textura y por ultimo relaciona el material con el overlay del mapa.

A continuación podemos ver cómo queda la visualización de la cámara principal y la del minimapa:

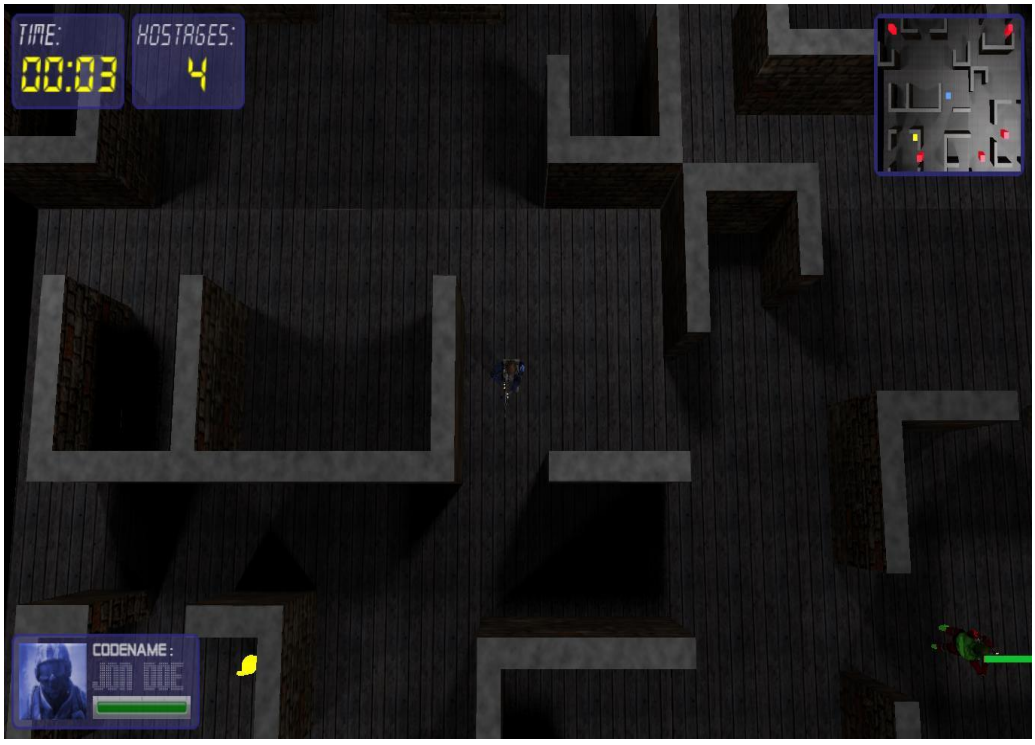

Figura 3.13 – Visualización de las cámaras en la escena

3.1.6. Registro de Records.

Para el registro de los records hemos usado una clase singleton llamada ***Records***, ver el siguiente diagrama:

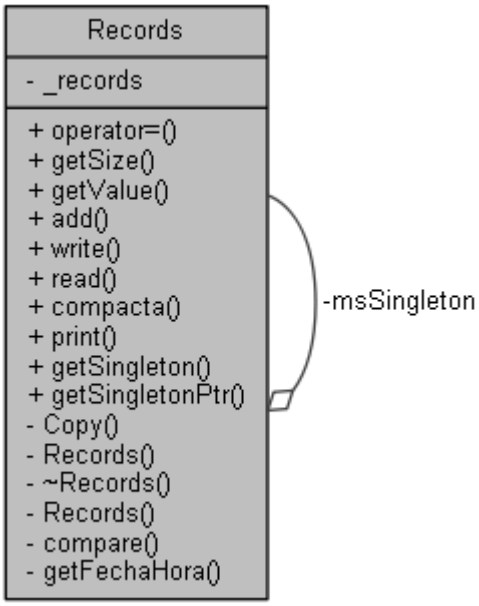

Figura 3.14 – Diagrama de clases de Records

La cual realiza las siguientes operaciones:

- Carga de fichero.
- Grabación a fichero.
- Añadir record.
- Compactación (sólo por temas de almacenar los X mejores records y no tener todos), aunque ésto es una funcionalidad opcional que ha sido aplicada.
- Impresión por pantalla (para depuración).

Un record lo compone:

- Tiempo empleado en rescatar a todos los rehenes (o antes de haber muerto nuestro héroe en caso de fallar).
- Número de rehenes rescatados.
- Fecha de realización del record.
- Hora de realización del record.

Toda ésta información será presentada en la pantalla de records del juego, como se puede ver en la Figura 3.15.

Figura 3.15 – Pantalla de Records

3.1.7. Lógica del Disparo.

Para la implementación del disparo hemos usado el objeto ClosestRayResultCallback como callback en la invocación al método rayTest() dado por Bullet.

Con éste método conseguimos hacer una consulta de aquel objeto con el que hemos colisionado si lanzamos un rayo desde un punto A a un punto B. Recibiendo en el callback ésta información, es decir, si hemos colisionado con algo y si lo hemos hecho nos diría con que objeto ha sido. Aquí detallamos la parte del código encargada de la detección de colisiones.

```
// Start and End are vectors

        btCollisionWorld::ClosestRayResultCallback rayCallback(posIni, posFin);



        // Perform raycast

        world->getBulletDynamicsWorld ()->rayTest(posIni, posFin, rayCallback);



        if(rayCallback.hasHit())

          {

            btCollisionObject* obA = (btCollisionObject*)(rayCallback.m_collisionObject);

            OgreBulletCollisions::Object *obOB_A = world->findObject(obA);
```

```
            int sizeCharacteres = characteres.size();

            Character *character = NULL;



            if (obOB_A != getRigidBody())

             {

                for (int i = 0; i < sizeCharacteres && (*characterCollision) == NULL; i++)

                 {

                     character = characteres[i];

                     if (obOB_A == character->getRigidBody() && character->_stateCaracter == LIVE)

                             {

                  (*characterCollision) = character;

                       }

                 }

              }

           isCollition = true;

        }
```

Listado 3-1 – Fragmento de la actualización del disparo.

Para el aspecto visual del disparo hemos empleado un objeto manual de OGRE (***createManualObject***) dando dos puntos y la linea que los une, nos permite dar la sensación de estela en el disparo. Aquí adjuntamos las lineas de código encargadas de la creación de ésta:

```
void Utilities::put_shot_in_scene ( Ogre::SceneManager* sceneMgr,  string name, Ogre::SceneNode*
nodeShot, Ogre::ManualObject** shot)

  {

    *shot =  sceneMgr->createManualObject(name + "LineSHOT");

   (*shot)->begin("MaterialShot", Ogre::RenderOperation::OT_LINE_LIST);

   (*shot)->position(Ogre::Vector3::ZERO);

   Ogre::Vector3 posFin(nodeShot->getOrientation().zAxis() * 1);

   (*shot)->position(posFin);

   (*shot)->end();

   (*shot)->setCastShadows(true);

   nodeShot->attachObject ( (*shot) );

 }
```

Listado 3-2 – Fragmento de la creación del disparo.

3.1.8. Visualización de los niveles de vida.

3.1.8.1. Nivel de vida del Enemigo.

Para la visualización de la barra con el nivel de vida del enemigo hemos usado un Billboard, ver imagen a continuación:

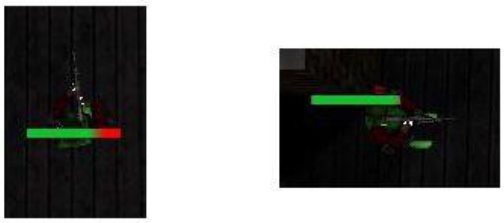

Figura 3.16 – Barra de vida del enemigo

Para la creación de ésta barra hemos realizado los siguientes pasos:

1. Creamos un BillboardSet el cual nos servirá como contenedor de 1 o varios Billboards asociados a nuestro personaje.
2. Dentro de éste BillboarSet que tiene nuestro personaje nos creamos un Billboard, con la propiedad de que su posición está sobre nuestro personaje.
3. Nos creamos un nodo de escena al cual le adjuntamos el BillboardSet creado.
4. Al nodo de nuestro personaje le adjuntamos este nuevo nodo creado con el BillboardSet. De manera que los movimientos que le aplicamos al enemigo se aplicarán igual a éste y por ende, a nuestra barra de nivel de vida.

3.1.8.2. Nivel de vida de Jon Doe.

Para la visualización del nivel de vida de Jon Doe hemos usado un overlay, el cual está compuesto por dos partes que podemos ver en la siguiente imagen:

Figura 3.17 – Barra de vida del Jon Doe

Para ello se siguen los siguientes pasos:

1. La foto de nuestro héroe. La cual se verá afectada por el impacto de un disparo de nuestros enemigos pasando del color azul “normal” a color “rojo” en el momento del impacto.

2. Barra de nivel de vida de nuestro personaje, la cual se subdivide en 4 trozos, lo que quiere decir que podremos recibir como mucho 4 impactos antes de fallecer y perder entonces la partida.

3.2. Técnicas gráficas.

En este apartado vamos a hablar de las técnicas graficas más relevantes que hemos utilizado durante la realización de nuestro juego.

3.2.1. Animación de los personajes.

Cada uno de los personajes que se han utilizado en el juego van acompañado de un conjunto de imágenes creadas para dar más realismo a los mismos.

Para crear las animaciones se ha utilizado Blender [6], para ello se han seguido los siguientes pasos:

1) Se ha creado el esqueleto del personaje.

2) Se atacha el esqueleto al persona y se asocia cada uno de los huesos del esqueleto a un grupo de vectores que se quiera mover con el mimos.

3) Creamos la animación moviendo cada uno de los huesos del personaje y creando frames claves con cada una de las posiciones que conforme la animación. Blender se encargará crear los frames intermedios entre cada frame clave.

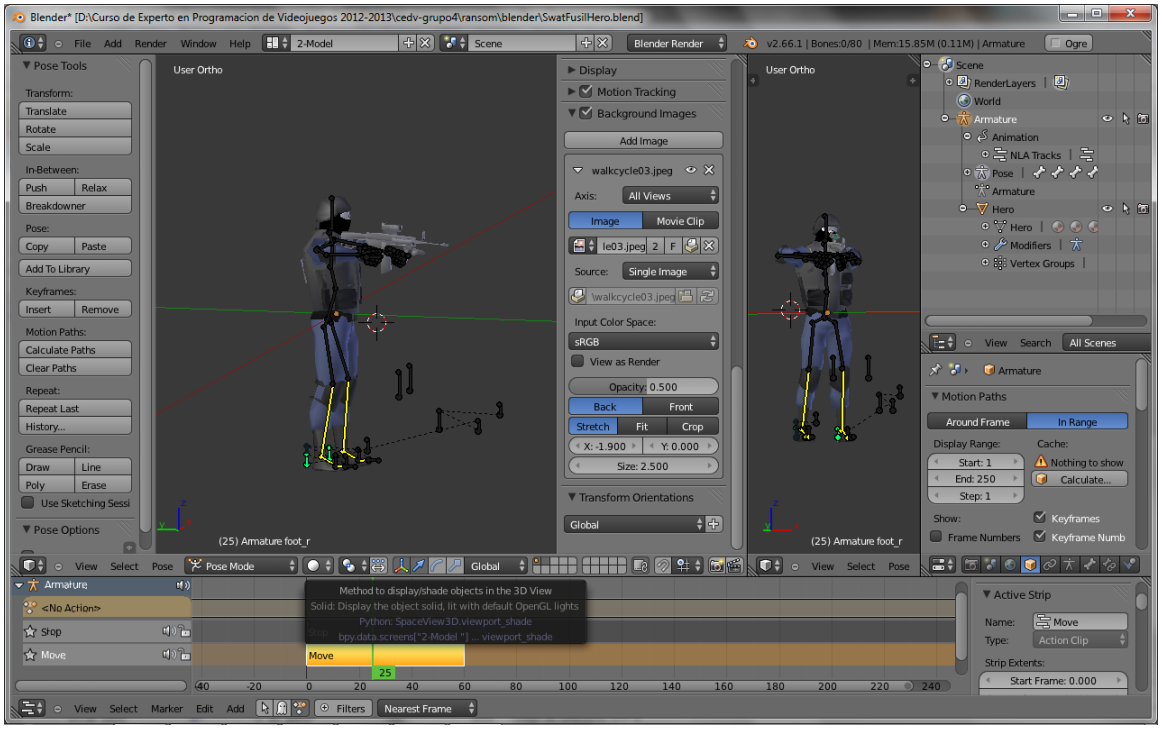

Figura 3.18 – Creación de animaciones en Blender

3.2.2. Precalculo de iluminación.

Durante la ejecución del juego se crean sombras dinámicas, que dan más nivel de visualización al jugador.

A parte de esto se ha decido hacer un prepalculo de iluminación aplicada a cada una de las piezas del escenario, aplicada a las texturas de cada una de ellas, para mejorar sus aspecto visual. Esto también optima el rendimiento, ya que al ser texturas estáticas el consumo es muchísimo inferior que el cálculo de sombras dinámicas que se nombra en el primer párrafo. Esto ha sido realizado con Blender.

En la Figura 3.19Figura 3.19 – Pieza 1 antes del precalculo se muestra como queda la pieza antes de crearle una textura con el precalculo de la iluminación, cuyo resultado se puede ver en la Figura 3.20.

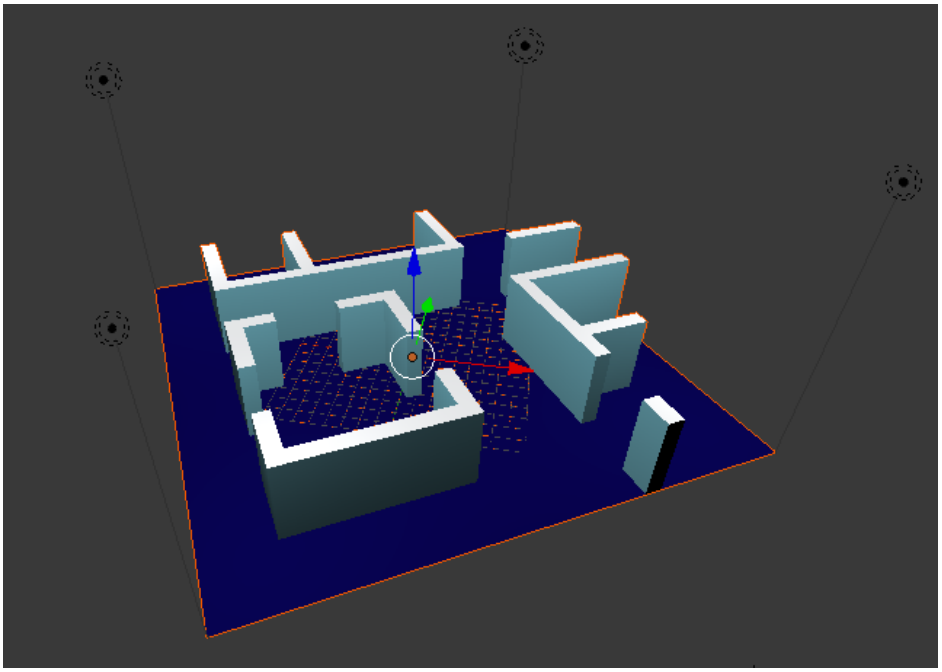

Figura 3.19 – Pieza 1 antes del precalculo

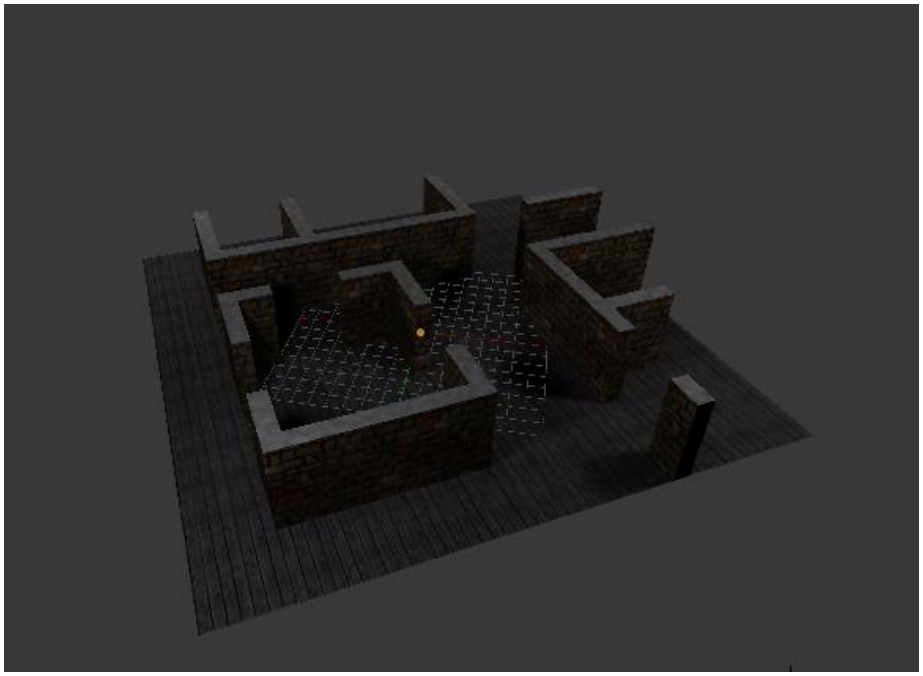

Figura 3.20 – Pieza 1 despues del precalculo

3.3. Herramientas.

Durante la realización de este juego se han utilizado las siguientes herramientas:

- ✓ **Code-Blocks [15] (Desarrollo)**

  Entorno IDE multi-sistema para el desarrollo en C/C++. Nosotros hemos usado su versión para GNU/Linux.

- ✓ **Visual C++ 2010 Express [18] (Desarrollo)**

  Entorno IDE para el desarrollo en C/C++ bajo Windows.

- ✓ **GDB [17] (Desarrollo)**

  Depurador para GNU/Linux usado para comprobar posibles errores de desarrollo.

- ✓ **GCC [16] (Desarrollo)**

  Es un conjunto de compiladores, de entre los cuales, nosotros hemos usado la versión C++ para la generación de nuestro juego.

- ✓ **GIMP [13] (Diseño Gráfico)**

  Programa multi-sistema para retoque fotográfico, usado para desarrollar las pantallas del juego, tales como la pantalla de menú, créditos, records, … los botones de los menús, los overlays empleados, … en definitiva todo lo relacionado con el aspecto visual "estático" del juego.

- ✓ **Audacity [19] (Editor de Sonido)**

  Programa multi-sistema de edición de sonido usado para modificar/convertir sonidos y así poder usarlos en nuestro juego.

- ✓ **Blender [4] (Diseño Gráfico)**

  Programa encargado de diseñar, modelar y animar escenarios y personajes en 3D relacionados con nuestro juego.

- ✓ **Doxygen [20] (Documentación)**

  Programa usado para la generación de la documentación relacionada con el desarrollo del juego. Con ésta aplicación podemos obtener, además, diagramas de clases mediante el empleo de un paquete adicional llamado Graphviz [21].

Capítulo 4.

4. Análisis de costes.

4.1. Costes humanos.

4.1. Costes humanos.

Los costes que ha tenido la realización de este proyecto han sido humanos, concretamente en número de horas trabajadas por cada uno de los componentes que conforman el equipo de trabajo.

El juego se empezó a desarrollar el 24 de Mayo del 2013 hasta el 21 de Julio del 2013, esto hace un total de 59 días (8.4 semanas). Teniendo en cuenta que hemos trabajado de una forma constante desde el primer día hasta el último trabajando en el juego unas 6 horas cada día, 5 días a la semana dos componentes; esto hace un total:

8.4 Semanas * 5 días/semana * 6 horas/día * 2 componentes = 504 horas

De este tiempo se ha implicado en diferentes tareas, cuya proporción es la siguiente:

- 5% Análisis, diseño y documentación. (25,2 horas)
- 20% Recursos del juego. (100,8 horas)
  - 15% Recursos 3D. (75,6 horas)
  - 4% Recursos 2D. (20,16 horas)
  - 1% Recursos sonoros. (5,04 horas)
- 75% Desarrollo. (378 horas)

Al ser un grupo tan reducido los dos componentes han tocado toda y cada una de las partes del juego y así poder obtener un conocimiento total del mismo. Se tomó esta decisión para que cada integrante se aplicase de todo lo impartido durante curso y llevarlo

a la práctica, y al ser solo dos personas esto no ha entorpecido la sincronización entre ellos.

Capítulo 5.

5. Manual de Usuario.

5.1. Interfaz del juego.

5.2. Pantalla de juego.

5.3. Ejecución del juego.

5.1. Interfaz del juego.

El juego costa de una pantalla principal (véase Figura 5.1) el cual contiene un menú con las siguientes opciones:

Figura 5.1 - Pantalla de records

- **Start Game:** Con esta opción se inicial el juego.
- **Best Times:** Muestra los mejores 10 mejores tiempos que se han obtenido en el juego, priorizados por el número de rehenes rescatados; mostrando la hora y la fecha en la que se ha conseguido la marca de la cual se podrá volver al menú principal pulsando la tecla de Escape. (véase Figura 5.2)

Figura 5.2 - Pantalla de records

- **Credits:** Pantalla de cedidos en la que se muestra la lista de los nombres de los integrantes del desarrollo de juego de la cual se podrá volver al menú principal pulsando la tecla de Escape. (véase Figura 5.3)

Figura 5.3 - Pantalla de créditos

- **Exit Game:** Opción para salir del juego.

Para moverse por los menús basta con hacerlo mediante el uso del ratón.

5.2. Pantalla del juego.

Nada más pulsar la *Star Game* del menú principal nos saldrá la ventana de carga, tal y como se puede ver en la Figura 5.4, que se mostrará mientras se carga todo el escenario del juego.

Figura 5.4 - Pantalla de carga

Cada vez que iniciamos una partida se genera un escenario aleatoriamente, de las dimensiones que se hayan indicado en el archivo de configuración, por lo que cada partida será diferente.

El juego consiste en rescatar a todos los rehenes, capturados por los enemigos, en el menor tiempo posible. Teniendo que esquivar a los enemigos, abatiéndolos y evitando que nos maten.

La pantalla principal está formada por varios componentes que nos muestran el estado del juego y del personaje principal, tal como se muestra en la Figura 5.5, los cuales son:

Figura 5.5 - Pantalla del juego

- ✓ Tiempo (Time): Arriba a la izquierda se muestra el tiempo que ha trascurrido desde el inicio de la partida.
- ✓ Rehenes capturados (Hostages): También arriba a la izquierda se muestra el número de rehenes que aún quedan capturados por los enemigos.
- ✓ Minimapa: Arriba a la derecha se muestra el minimapa del juego, que muestra una visión general al jugador del estado del juego. En el cual el cubo de color azul es el Jon Doe, los cubos de color rojo muestran la posición de los enemigos y los cubos de color amarillo muestra la posición de los rehenes.
- ✓ Estado de Jon Doe: Abajo a la izquierda se muestra el estado del personaje. El cual está formado por dos componentes principales, tal y como se puede ver en la Figura 5.6, que son:
  - Nivel de salud de Jon Doe: Barra que indica el nivel de vida del personaje.
  - Imagen del Jon Doe: Que cuando el personaje es dañado la imagen se tiñe de rojo.

Figura 5.6 - Estado del personaje

Las teclas durante la ejecución del juego son las siguientes:

- ✓ Subir: Mueve a Jon Doe hacia arriba.
- ✓ Bajar: Mueve a Jon Doe hacia abajo.
- ✓ Izquierda: Mueve el personaje a la izquierda.
- ✓ Derecha: Mueve el personaje hacia la derecha.
- ✓ Espacio: Ejecuta el disparo de Jon Doe. En la Figura 5.7 se muestra un ejemplo de cruce de disparos.

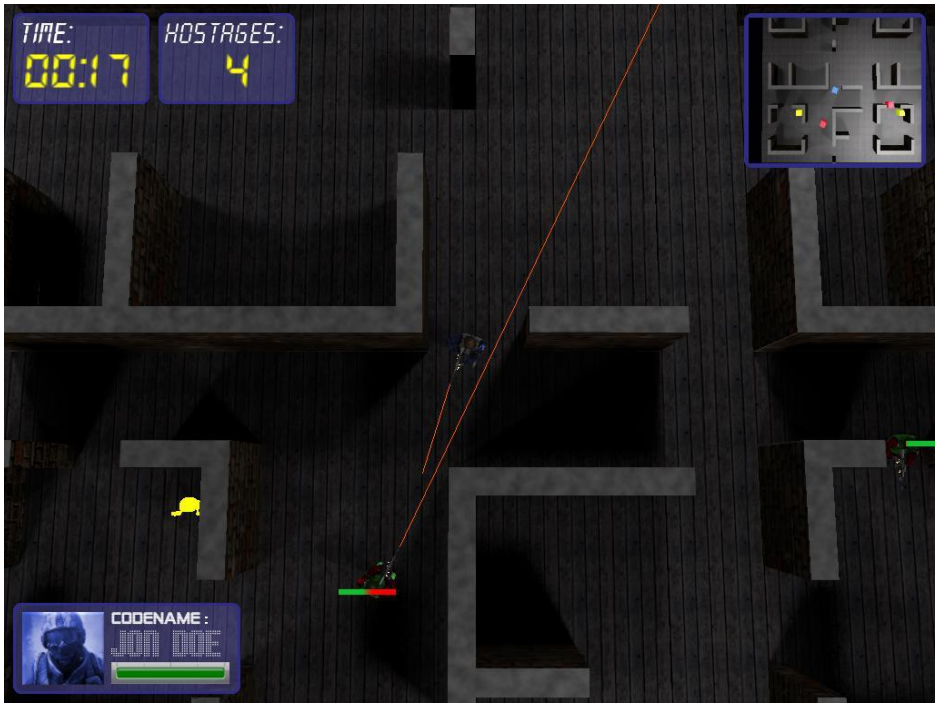

Figura 5.7 - Cruce de disparos

- ✓ Escape: Pausa el juego y muestra la venta de pausa. En la que podremos retornar al juego (Resume Game), volver al menú principal (Return to Main Menu) o salir del juego (Exit Game), tal y como se ve en la Figura 5.8.

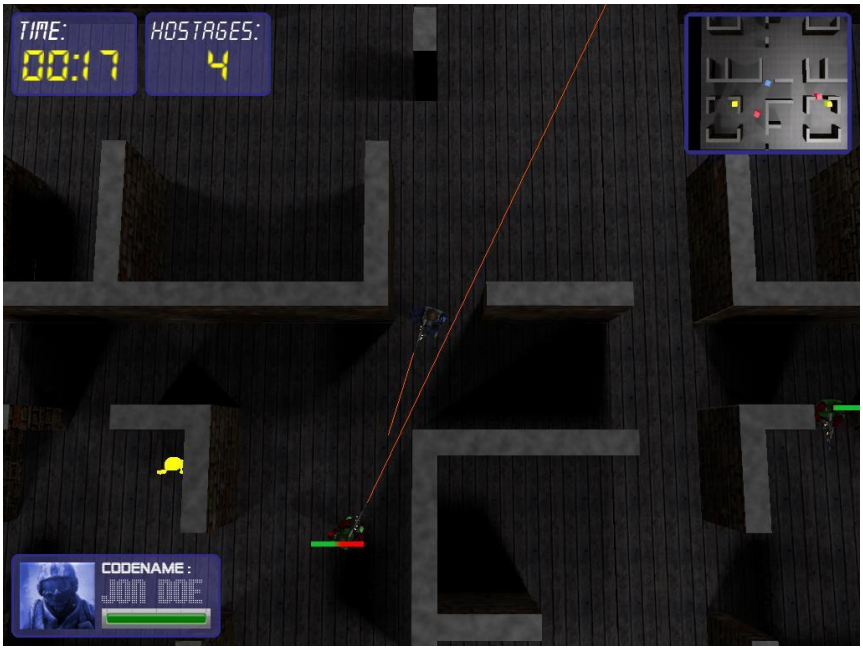

Figura 5.8 - Pantalla de pausa

Cada enemigo va acompañado de una barra que muestra el estado de vida del mismo, se puede ver un ejemplo en la Figura 5.9.

Figura 5.9 - Barra de vida de los enemigos

Cuando el enemigo sea abatido por completo este desaparecerá de la escena, tal y como se muestra en la Figura 5.10.

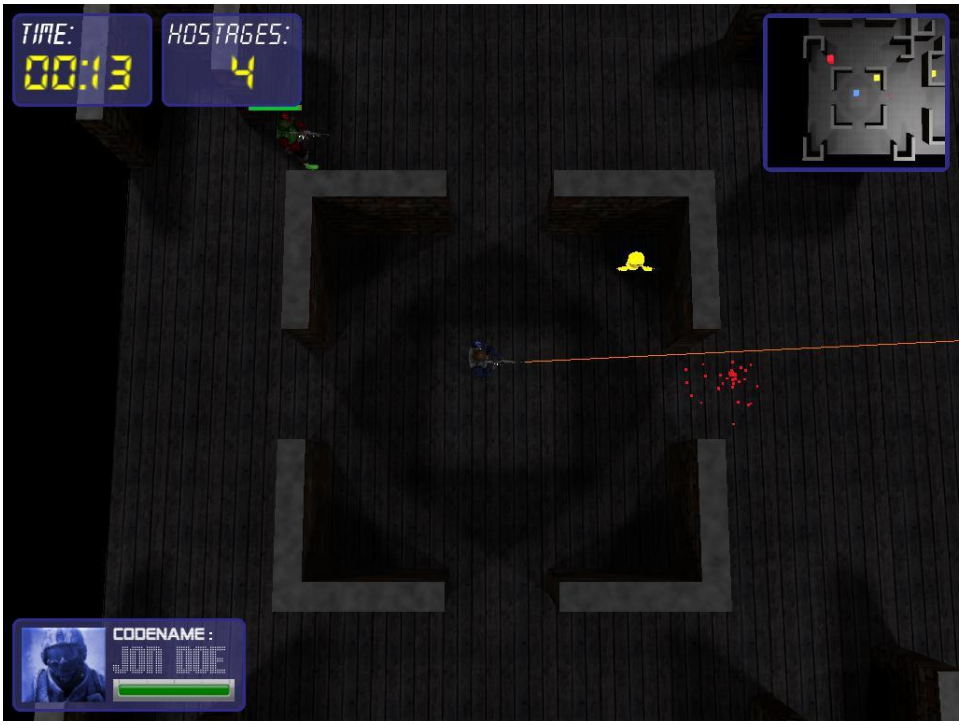

Figura 5.10 - Muerte del enemigo

Para liberar a los rehenes bastará con que acerquemos a Jon al rehén y este quedará liberado, como podemos ver en la Figura 5.11.

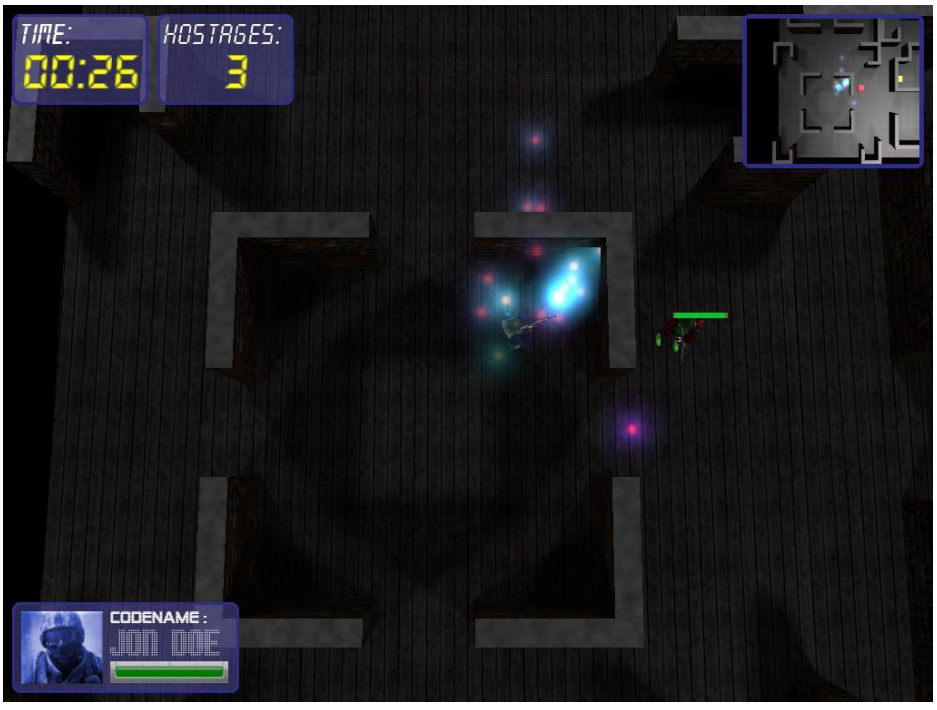

Figura 5.11 - Liberación de rehén

Si los enemigos han acabado con la vida de Jon Doe, el juego finalizará. Véase la Figura 5.12.

Figura 5.12 - Pantalla de Game Over

Si hemos liberado todos los rehenes se guardará el record y se mostrará la pantalla de la Figura 5.13.

Figura 5.13 - Pantalla de TO BE CONTINUED…

5.3. Ejecución del juego.

El juego ha sido desarrollado tanto para Linux como para Windows.

Para ejecutar la versión de Linux basta con entrar en la carpeta "**bin/linux**" y ejecutar el ejecutable **ramson**.

Para ejecutar la versión de Linux basta con entrar en la carpeta "**bin/windows**" y ejecutar el ejecutable **Ramson.exe**.

Capítulo 5.

6. Conclusiones y propuestas.

6.1. Conclusiones.

6.2. Líneas futuras de desarrollo e investigación.

6.1. Conclusiones.

En este capítulo se ve cómo se ha llegado a cumplir cada uno de los objetivos propuestos y cuya consecución ha ayudado a cumplir el objetivo general.

- ✓ **Implementación de cada una de las pantallas que conforman el juego y la interacción entre cada una de ellas.** La implementación de cada uno de los estados del juego (Juego, Records, Títulos, Salir, Pausa, etc..) los hemos realizado la utilización de Advanced Ogre Framework [10]. Los menús han sido realizados con SdkTrays [11] que es el sistema de GUI que viene integrado en Advanced Ogre Framework. Los fondos de cada una de las pantallas se han realizado con el uso de Overlays de Ogre3D.

- ✓ **Diseño y modelado de escenario.** Cada una de las piezas del escenario han sido diseñadas y modeladas en Blender, donde hemos aplicado varias técnicas para darle más realismo a las mismas, como por ejemplo el precalculo de iluminación.

- ✓ **Inserción de personajes en la escena.** Los modelos de los personajes de nuestro juego los hemos descargado de BlenSwap [12], que una página en la que tiene modelos en Blender de licencia libre. Dichos modelos los hemos tenido que

adaptar para nuestro juego (inserción de esqueleto, creación de animación, cambio de materiales y texturas, reducción de polígonos y vectores, etc…) mediante el retoque en Blender.

- ✓ **Creación de todos los diseños de las pantallas.** Todos los diseños de las pantallas de nuestro juego han sido modelados con la herramienta libre GIMP [13].

- ✓ **Implementación de los movimientos de los personajes del juego.** Cada uno de los personajes se le ha dotado de un conjunto de animaciones para darle más realismo, dichas animaciones han sido realizadas con Blender y ejecutadas en el juego gracias al soporte que nos da Ogre3D para el manejo de las mismas.

- ✓ **Inteligencia artificial de los enemigos.** Hemos dotado de IA a los personajes mediante la implementación de una máquina de estados, en donde cada uno de los estado dota de un conjunto de acciones al enemigo ante ciertas situaciones y dependiendo de la acción realizada pasar a un estado u otro.

- ✓ **Lógica de rescate de rehenes.** Los rehenes son liberados cuando el hero se aproxima a ellos. Para hacer esto hemos hecho uso del sistema de colisiones que nos ofrece Bullet [4] (junto con OgreBullet [5]). Y para dar las peso visual a la liberación le hemos añadido un efecto de partículas, implementado con el uso de Partículas que nos ofrece Ogre3D.

- ✓ **Mecanismos para mostrar el estado de cada uno de los personajes.** Los mecanismos para mostrar el estado del personaje principal (nivel de vida, daño por disparo, etc…) han sido implementado visualmente mediante el uso de Overlays. Para mostrar el nivel de vida de los enemigos hemos implementado mediante el uso de sistemas de Billboards que nos ofrece Ogre3D. Cada uno de los cambios de estado de los personajes (liberación del rehén, dañar a un personaje, cambios de estado del enemigo, etc…) van acompañados de un efecto sonoro que han sido añadidos al juego gracias al uso de la librería SDL [9].

- ✓ **Mecanismos de física para que todos los componentes de la escena interactúen correctamente.** Todos los mecanismos de física que se han dado al juego han sido realizados con la librería Bullet que junto con OgreBullet nos han

permitido sincronizar la física con los componentes de Ogre3D de la escena del juego.

- ✓ **Mecanismo de disparo para el héroe y los enemigos.** El mecanismo de disparo ha sido implementado usando los mecanismos de rayos de colisiones (para saber cuándo un disparo impacta contra un elemento de la escena) que nos ofrece Bullet; junto con componentes visuales de Ogre3D (para dar un carácter visual al mismo) y sonoros con SDL.

- ✓ **Mecanismos para la eliminación de los personajes de la escena.** Para eliminación de los personajes de la escena, para darle un carácter más artístico, lo hemos hecho usando los sistemas de Partículas que tiene Ogre3D acompañado de efectos sonoros con SDL.

- ✓ **Sistema de configuración el juego.** Hemos desarrollado la clase GameConfig que junto con la clase XMLCharger, nos ha permitido cargar la configuración almacenada en un archivo XML y llevarla a la escena del juego.

- ✓ **Manejo de records.** Hemos desarrollado la clase Records por medio de la cual almacenamos las mejores puntuaciones y las recupera para que sean mostradas en la pantalla de *Best Times*.

- ✓ **Minimapa del juego.** Hemos creado un render a textura, compuesto a partir de lo que visualiza una cámara definida para esta función. El renderizado del render a textura avisa un *Listener*, que hemos implementado, que nos ha permitido mostrar el mapa tal y como queríamos sin afectar a la visualización general del juego. El material que contiene la textura del minimapa es mostrado en Overlay en la escena.

- ✓ **Manejo de la cámara de una forma natural y que siga al personaje.** Para poder manejar las cámaras del juego, tanto la cámara principal como la cámara del minimapa, hemos implementado la clase CamerasController que se encarga de mover las cámaras de tal manera que sigan al personaje principal de una forma natural.

- ✓ **Inserción de recursos sonoros al juego.** El juego va acompañado de diferentes recursos sonoros, ya sean efectos sonoros como música para ambientar la escena y

los menús. Dicho recursos sonoros han sido añadidos a nuestro juego gracias al uso de la librería SDL.

- ✓ **Que el juego sea multisistema.** El juego ha sido desarrollado tanto en Linux (con la herramienta Code::Blocks [15], GCC [16] y GDB [17]) como en Windows (con Visual C++ 2010 Express [18]). Lo cual nos ha permitido compilar nuestro juego tanto para sistemas Linux como para Windows.

6.2. Líneas futuras de desarrollo e investigación.

A continuación se van a exponer algunas de las líneas futuras de desarrollo en investigación para que ayuden a mejorar nuestro juego:

- Mejora de la IA de los enemigos. Se puede mejorar la inteligencia de los enemigos mediante el uso de diferentes algoritmos de IA y así poder mejorar el comportamiento de los enemigos.
- Realizar un editor de niveles del juego, que nos permita componer mediante diferentes piezas varios niveles de dificultad para el juego.
- Creación de una pantalla de configuración donde el jugador pueda cambiar diferentes parámetros del juego como activación y desactivación de música y sonidos, nivel de dificultad, opciones de rendimiento, etc…
- Que el juego sea portado a plataformas móviles, porque el juego solo es soportado para plataformas PC.
- Añadir diferentes armas a los personajes. Ya que actualmente solo se hace uso de un tipo de armas.
- Añadir más variedad de personajes (héroes, enemigos y rehenes), dando incluso la opción al usuario de poder elegir entre diferentes héroes.

Bibliografía

[1] OGRE – Open Source 3D Graphics Engine. OGRE – Open Source 3D Graphics Engine. http://www.ogre3d.org. Última visita el 18/07/2013.

[2] Commandos. http://es.wikipedia.org/wiki/Commandos. Última visita el 18/07/2013.

[3] Frozen Synapse. http://www.frozensynapse.com/. Última visita el 18/07/2013.

[4] Bullet. http://bulletphysics.org/wordpress/. Última visita el 18/07/2013.

[5] OgreBullet. http://www.ogre3d.org/tikiwiki/OgreBullet. Última visita el 18/07/2013.

[6] Blender. http://www.blender.org/. Última visita el 18/07/2013.

[7] Curso de Experto en Desarrollo de Videojuegos 2012/2013. http://www.cedv.es/. Última visita el 18/07/2013.

[8] Boost. http://www.boost.org/. Última visita el 18/07/2013.

[9] Simple DirectMedia Layer (SDL). http://www.libsdl.org/. Última visita el 18/07/2013.

[10] Advanced Ogre Framework. http://www.ogre3d.org/tikiwiki/Advanced%20Ogre%20Framework. Última visita el 18/07/2013.

[11] SdkTrays. http://www.ogre3d.org/tikiwiki/tiki-index.php?page=SdkTrays. Última visita el 18/07/2013.

[12] BlenSwap. http://www.blendswap.com/. Última visita el 18/07/2013.

[13] GIMP. http://www.gimp.org.es/. Última visita el 18/07/2013.

[14] SDL. http://www.libsdl.org/. Última visita el 18/07/2013.

[15] Code::Blocks. http://www.codeblocks.org/. Última visita el 18/07/2013.

[16] GCC. GCC, the GNU Compiler Collection. http://gcc.gnu.org/. Última visita el 18/07/2013.

[17] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/. Última visita el 18/07/2013.

[18] Visual C++ 2010 Express. http://www.microsoft.com/visualstudio/esn/products/visual-studio-2010-express. Última visita el 18/07/2013.

[19] Audacity. http://audacity.sourceforge.net/?lang=es. Última visita el 18/07/2013.

[20] Doxygen. http://www.stack.nl/~dimitri/doxygen/. Última visita el 18/07/2013.

[21] Graphviz. http://www.graphviz.org/. Última visita el 18/07/2013.