ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

IBEROGRE, WIKI DE OGRE3D EN ESPAÑOL
Y SION TOWER, VIDEOJUEGO DE ESTRATEGIA

David Saltares Márquez

16 de agosto de 2011

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

IBEROGRE, WIKI DE OGRE3D EN ESPAÑOL Y SION TOWER, VIDEOJUEGO DE ESTRATEGIA

- Departamento: Lenguajes y sistemas informáticos
- Directores del proyecto: Manuel Palomo Duarte y Francisco Palomo Lozano
- Autor del proyecto: David Saltares Márquez

Cádiz, 16 de agosto de 2011

Fdo: David Saltares Márquez

Agradecimientos

Me gustaría agradecer y dedicar este texto a:

- Manuel Palomo y Francisco Palomo por sus consejos y labor de tutorización.
- Mis compañeros Javier, Jose, Daniel, José Tomás y Alberto por su ayuda en los momentos de mayor presión y sus empujoncitos para que se encienda la bombilla.
- Daniel Pellicer, Antonio Caro, Francisco Martín, Antonio Rodríguez y Daniel Belohlavek por el gran trabajo artístico que han realizado.
- Ainhoa por apoyarme tanto en las épocas de éxito como de frustración así como por su labor de testeo.
- Toda mi familia por el cariño incondicional en todo momento.
- La fantástica comunidad de usuarios que han ofrecido su colaboración y sugerencias.

Licencia

Notación y formato

Con el objetivo de mantener un estilo uniforme y cómodamente legible, a lo largo de esta memoria de **Proyecto Fin de Carrera** se ha utilizado la siguiente notación:

- Para referirnos a nombres de ficheros, órdenes del sistema o funciones de un lenguaje utilizaremos: `ogre.cfg`.
- Cuando se nombre a una tecnología o biblioteca se hará uso del formato: OGRE3D.
- Los nombres de las clases se escriben en cursiva: *Game*.
- Cuando se mencione el nombre de un programa se hará con el siguiente formato: *Blender*.
- En caso de adjuntar un fragmento de código se utilizarán bloques como el siguiente:

```
1  // Ejemplo de programa en C++
2
3  #include <iostream>
4
5  int main(void) {
6      std::cout << "Hola mundo" << std::endl;
7
8      return 0;
9  }
```

Índice general

Indice de figuras

Indice de tablas

Capítulo 1

Introducción

1.1. Proyecto propuesto

IberOgre es una documentación libre en formato wiki sobre desarrollo de videojuegos en tres dimensiones utilizando el motor de renderizado también libre OGRE3D. No sólo cubre el uso del conocido motor gráfico sino que incluye documentación relacionada con otras bibliotecas y tecnologías complementarias necesarias para el desarrollo de videojuegos. Además, tiene como objetivos recordar y repasar los conceptos matemáticos fundamentales para continuar estudiando esta disciplina como álgebra o geometría del espacio. Cada lección intercalará conceptos teóricos y pequeños ejemplos prácticos para finalizar con un paquete descargable completamente documentado que muestre una aplicación práctica sobre dicha lección.

Sion Tower es un videojuego de estrategia y acción en 3D de ambientación fantástica cuyo objetivo es servir de ejemplo final a la wiki. Por supuesto, hace uso de OGRE3D y otras tecnologías complementarias documentadas en **IberOgre**. En **Sion Tower** controlamos a un joven hechicero iniciado que se ve obligado a defender la Torre Sagrada de una invasión enemiga mientras sus compañeros están celebrando un rito secreto en el exterior. Cada nivel es un piso de la Torre y para poder avanzar tenemos que acabar con todos los enemigos que atacan en oleadas la zona. Para hacerlo tendremos que emplear nuestra energía mágica o maná en invocar proyectiles de diversa índole. El videojuego debe estar completamente documentado para que los lectores de **IberOgre** tengan acceso directo a una aplicación práctica lo más realista posible de la tecnología.

1.2. Contexto

En esta sección destacaremos la importancia de la industria del videojuego empleando como escenario concreto España. Asimismo, haremos hincapié sobre el éxito de los videojuegos 3D sobre los bidimensionales en términos de mercado. Finalmente, realizaremos un pequeño repaso sobre qué es un motor de videojuegos y adjuntaremos un breve comentario sobre algunos de ellos, libres y privativos.

1.2.1. Auge de los videojuegos 3D

Es innegable que los videojuegos son un mercado en auge que está soportando como pocos el efecto de la crisis económica mundial. En la sección 1.2.2 ofreceremos cifras que apoyan estas afirmaciones dentro del contexto del mercado español.

Existe mucha discrepancia acerca de cuál es el primer videojuego de la historia ya que la propia definición de videojuego resulta a la vez confusa. Muchos consideran que Alenxander Douglas fue el padre del primer videojuego moderno cuando en el año 1952 recreó en el computador EDSAC de la Universidad de Cambridge una versión del popular tres en raya llamada OXO con gráficos completamente primitivos (ver figura 1.1) [48].

Figura 1.1: OXO, primer videojuego de la historia (1952)

Pasaron casi 30 años desde el primer videojuego moderno hasta que llegaron los primeros videojuegos con sensación de profundidad en los años 80. Battlezone (ver figura 1.2) era un videojuego de acción con tanques y artillería para la consola Atari lanzado en 1980. Utilizaba vectores verdes y rojos que delineaban el contorno de los objetos sobre el fondo negro del escenario. Los videojuegos con gráficos vectoriales supusieron un gran avance para la época ya que, con pocos recursos se podía conseguir una aceptable sensación de profundidad [48].

Figura 1.2: Battlezone, videojuego con gráficos vectoriales (1980)

Los gráficos poligonales en tres dimensiones se popularizaron rápidamente a finales de los años 80 y comienzos de los 90. No obstante, las técnicas de programación no estaban lo suficientemente evolucionadas y, al aumentar el detalle, aumentaba de manera exponencial la potencia de cálculo necesaria. Alone in the Dark (ver figura 1.3), publicado en 1992, fue el primer videojuego completamente en 3D en cosechar un éxito importante, además fue uno de los primeros en pertenecer al género de terror. Exis-

ten otros muchos ejemplos de videojuegos que simulaban mundos 3D utilizando un mundo 2D que se dibujaba en 3D (llamado a veces *dos dimensiones y media*), y que tuvieron éxito durante los 90 como Wolfenstein 3D (Id Software, 1992) o Doom (Id Software, 1993) [48].

Figura 1.3: Alone in the Dark (PC - 1992)

A mediados de los 90 los gráficos 3D saltaron del PC a las consolas de sobremesa. En 1994 Sega presentó Sega Saturn y Sony hizo lo propio con PlayStation, en 1996 Nintendo lanzó al mercado Nintendo 64. La penetración de estas consolas en el mercado fue tal que los gráficos en 3D llegaron a millones de usuarios. PlayStation fue la consola más exitosa llegando a vender 100 millones de unidades. En esa época llegaron títulos revolucionarios como el juego de infiltración y acción Metal Gear Solid (ver figura 1.4) o el plataformas Super Mario 64 (ver figura 1.5). Los gráficos en 3D llegaron para quedarse [48].

Figura 1.4: Metal Gear Solid (PlayStation - 1998)

La siguiente generación de consolas llegó en el año 2000 con la PlayStation 2 de Sony. Un año más tarde Microsoft entró en el mercado con Xbox y Nintendo lanzó Game Cube. PlayStation 2 se convirtió en la consola más vendida de la historia hasta el momento. Tanto en PC como en consolas de sobremesa, los gráficos en 3D ya se habían consolidado. Llegaron títulos como God of War (ver figura 1.6) y Half-Life 2 (ver figura 1.7) [48].

Figura 1.5: Super Mario 64 (Nintendo 64 - 1996)

Figura 1.6: God of War (PlayStation 2 - 2005)

Otro hito en el mundo de los videojuegos tridimensionales fue su llegada a las consolas portátiles, hasta entonces con insuficiente potencia. En 2004 Nintendo lanzó su portátil Nintendo DS y poco después Sony respondió con PlayStation Portable (PSP). La reducción de los procesadores permitió que dispositivos de pequeño tamaño fueran capaces de mostrar en pantalla un elevado número de polígonos y efectos. Actualmente este fenómeno está tomando importancia en los dispositivos móviles iPhone y Android. Incluso conocidos motores han llegado a iOS en demos técnicas como Epic Citadel (ver figura 1.8) [48].

El último paso han sido las consolas de la última generación Xbox 360 (2005), Wii (2006) y PlayStation 3 (2006) y la llegada de la tecnología DirectX 11 al PC. Los entornos 3D se han hecho cada vez más grandes, los efectos consiguen un mayor realismo y las físicas no difieren apenas de la realidad. Las compañías luchan por estar a la vanguardia y los equipos más avanzados cuentan con decenas de ingenieros especializados en campos muy concretos. Un ejemplo de videojuego vanguardista podría ser Uncharted 2 (ver figura 1.9) [48].

Podemos concluir que los videojuegos tridimensionales han experimentado una crecida exponencial a lo largo de los últimos años. Esto se debe a que consiguen un mayor realismo e inmersión por parte del usuario dentro del universo recreado.

Figura 1.7: Half-Life 2 (PC - 2005)

1.2.2. Los videojuegos como industria

Si acudimos al último informe publicado por aDeSe [1] (Asociación Española de Distribuidores y Editores de Software de Entretenimiento), es decir, al correspondiente al pasado año 2010, podemos percatarnos de la fuerza del sector en nuestro país. A pesar del efecto provocado por la crisis económica que ha hecho disminuir el valor de la industria un 5,2 % durante el último año, dicho valor llega a los 1.245 millones de euros. Además, la industria del videojuego copa el 50 % de todo el mercado relacionado con el ocio visual e interactivo.

Si prestamos atención a la figura 1.10 podremos observar que el sector del software sólo ha descendido un 0,38 % mientras que el de los periféricos incluso ha crecido un 5,6 %. Por su parte, el sector de las consolas ha disminuido un 13 %. Estas variaciones son comprensibles ya que en 2010 las consolas actuales se aproximan poco a poco al final de su ciclo de vida. Por ejemplo, la sucesora de la consola portátil Nintendo DS [52], la recientemente lanzada Nintendo 3DS [51], fue anunciada de forma oficial durante el año 2010. El aumento del segmento de los periféricos ha podido crecer gracias al lanzamiento de Kinect [49] durante la campaña navideña del año 2010. Kinect es un sensor de movimiento desarrollado por Microsoft compatible con la consola Xbox 360.

En definitiva, salvo pequeñas variaciones, la industria del videojuego en España muestra salud y un gran potencial.

Para destacar la importancia de los videojuegos en 3D podemos recurrir a la lista de los más vendidos en España durante el pasado año 2010 reflejada en la tabla 1.1.

Como puede observarse, un 85 % de los juegos más vendidos en España durante el año 2010 en cualquier plataforma son en tres dimensiones. Dicha cifra ya resulta un claro indicador del atractivo que suponen las 3D para la comunidad de usuarios. No obstante, si tenemos únicamente en cuenta los títulos para consolas de sobremesa (17 entre los 20 más vendidos), vemos que el 100 % de ellos utilizan gráficos tridimensionales.

Figura 1.8: Epic Citadel (iOS - 2009)

Figura 1.9: Uncharted 2 (PlayStation 3 - 2009)

1.2.3. Motores de videojuegos

Como indica la publicación Game Engine Architecture de Jason Gregory [19] el término *motor de videojuegos* apareció a mediados de los años 90 con el videojuego de acción en primera persona Doom [45] de Id Software (ver figura 1.11). Fue el primer videojuego cuya arquitectura estaba bien definida y separaba componentes como el motor de renderizado, el sistema de audio, la detección de colisiones y los recursos artísticos. Esto permitió que los desarrolladores pudiesen reutilizar estos elementos para lanzar nuevos productos distintos al original. Podían cambiar armas, niveles y vehículos con ninguno o pocos cambios al motor.

Fue entonces cuando nació lo que se conoce como la comunidad del *modding*. Usuarios y profesionales que se dedican a modificar juegos aprovechando la independencia del motor de los recursos. Estos arreglos pueden ir desde modificar reglas de juego, añadir elementos hasta incluso crear un juego radicalmente diferente al original. A finales de la década de los 90 ya se desarrollaban motores con estas modificaciones en mente. Un ejemplo es Quake III Arena [55] de Id Software (ver figura 1.12) y su lenguaje de scripting Quake C.

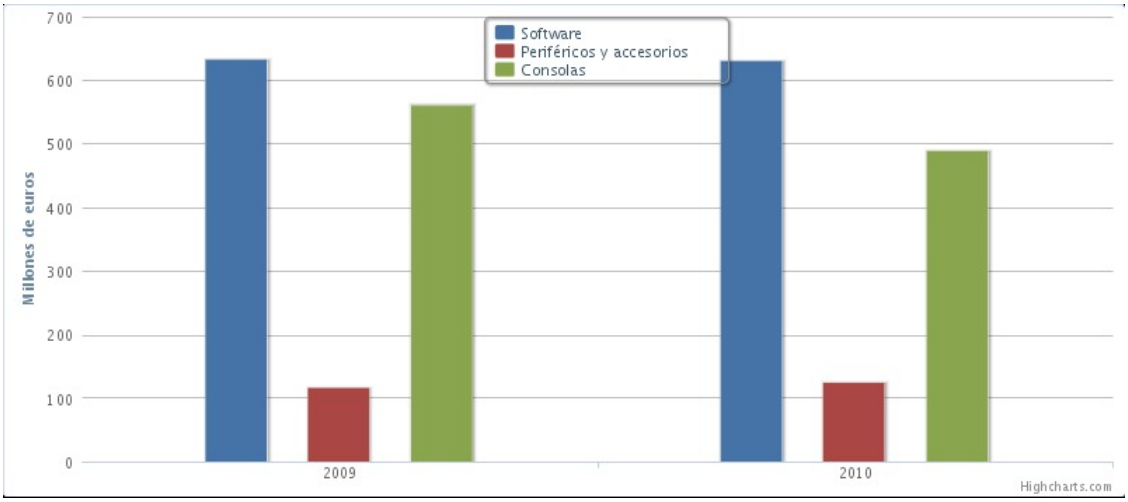


Figura 1.10: División del sector de los videojuegos

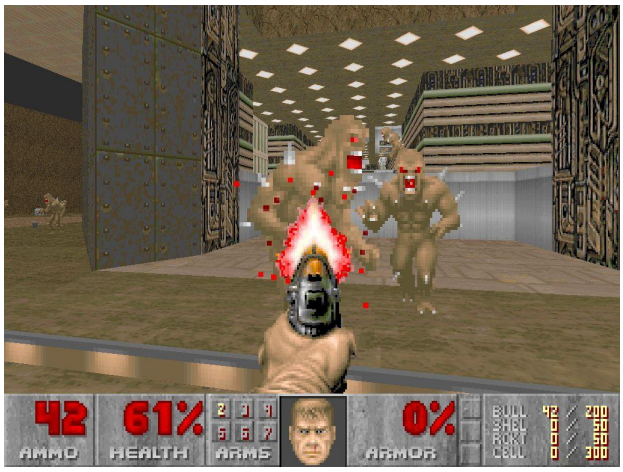


Figura 1.11: Doom (PC - 1993)

Hoy en día es muy común que muchas compañías pequeñas no puedan permitirse económicamente el desarrollo de su propio motor. Por ello, adquieren licencias de motores de compañías externas para comenzar sus proyectos. A pesar de que dichas licencias conllevan un desembolso importante, suele ser más asequible que el desarrollo completo del motor.

La línea que separa lo que es un motor de videojuegos de lo que no lo es es muy difusa. Un motor podría saber cómo dibujar un personaje específico mientras que otro simplemente recibiría una colección de vértices y triángulos de un fichero en memoria secundaria. Por tanto, el segundo motor sería más generalista que el segundo y más reutilizable. Cuando un motor incluye lógica de juego y elementos muy concretos dentro de su código fuente, es prácticamente imposible reutilizarlo para otro propósito. La situación opuesta tampoco se da, no existe un motor capaz de ser óptimo en cualquier situación. Hay algunos que están preparados para mostrar escenarios gigantescos con bajo nivel de detalle y miles de personajes en pantalla mientras que otros se centran en entornos limitados con un nivel de detalle y efectos impresionante.

Puesto	Título	Plataforma	Género	Distribuidor	¿3D?
1	Call of Duty: Black Ops	PlayStation 3	Acción	Activision Blizzard	Sí
2	New Super Mario Bros.	Wii	Plataformas	Nintendo	Sí
3	Wii Fit Plus	Wii	Fitness	Nintendo	Sí
4	Gran Turismo 5	PlayStation 3	Conducción	Sony	Sí
5	Wii Party	Wii	Party Game	Nintendo	Sí
6	FIFA 11	PlayStation 3	Deportivo	Electronic Arts	Sí
7	Pro Evolution Soccer 2011	PlayStation 3	Deportivo	Konami	Sí
8	Super Mario Galaxy 2	Wii	Plataformas	Nintendo	Sí
9	God of War III	PlayStation 3	Acción	Sony	Sí
10	Art Academy	Nintendo DS	Habilidad	Nintendo	No
11	Mario Kart	Wii	Conducción	Nintendo	Sí
12	Formula 1 2010	PlayStation 3	Conducción	Warner Interactive	Sí
13	Call of Duty: Modern Warfare 2	PlayStation 3	Acción	Activision Blizzard	Sí
14	Wii Sports Resort	Wii	Deportes	Nintendo	Sí
15	Red Dead Redemption	PlayStation 3	Acción	Take 2	Sí
16	Pokémon Soulsilver	Nintendo DS	RPG	Nintendo	No
17	Assassin's Creed: La Hermandad	PlayStaton 3	Acción	Ubisoft	Sí
18	Wii Play	Wii	Party Game	Nintendo	Sí
19	Pokémon Heartgold	Nintendo DS	RPG	Nintendo	No
20	Starter Pack	PlayStation 3	Party Game	Sony	Sí

Tabla 1.1: Tabla de los 20 videojuegos más vendidos en España durante el año 2010

Figura 1.13: Videojuego libre Abaddon

Afortunadamente, existen motores libres como OGRE3D (*Object Oriented Graphics Rendering Engine* o motor gráfico de renderizado orientado a objetos). Básicamente es una biblioteca compatible con el lenguaje de programación C++ que abstrae las funcionalidades de bajo nivel de DirectX [44] y OpenGL [53] para comunicarse con el hardware. Su versión 1.0 de nombre en clave Azathoth fue lanzada en

Figura 1.12: Quake III Arena (PC - 1999)

2005 y desde entonces ha cosechado grandes éxitos como el premio al proyecto del mes en SourceForge. Únicamente proporciona el sistema de renderizado (dibujado de una escena 3D en patalla) por lo que es necesario implementar otros subsistemas o recurrir a otras bibliotecas (colisiones, audio, entrada del usuario, red, etc) [24].

Figura 1.14: Videojuego libre Stunt Rally

Su licencia permisiva (MIT) ha permitido la creación tanto de juegos libres como comerciales y cerrados. Abaddon (figura 1.13) y Stunt Rally (figura 1.14) son videojuegos libres de géneros muy distintos (rol y conducción respectivamente) que han sido desarrollados utilizando OGRE3D, lo que demuestra su enorme versatilidad. Asimismo, como títulos cerrados podemos mencionar a Torchlight (figura 1.15) y a Victory The Age of Racing (figura 1.16) [24].

Figura 1.15: Torchlight (Windows, Mac - 2009)

OGRE3D cuenta con una arquitectura (figura 1.17) modular extensible a través de plugins. Si una característica necesaria para nuestro proyecto no viene incluida en el motor es posible implementarla e integrarla sin demasiados problemas. Por supuesto, también es recomendable acudir a su excelente comunidad ya que es probable que lo que busquemos esté hecho y pueda ser reutilizado. Queda patente por las capturas de pantalla que OGRE3D es un motor gráfico moderno con grandes capacidades [24].

Figura 1.16: Victory (Windows - 2011)

Actualmente, OGRE3D es uno de los mejores motores gráficos libres que existen. Si bien es cierto que su potencia y extensibilidad hacen que la curva de aprendizaje sea elevada, el hecho de tomar elementos similares al de otros motores ampliamente utilizados en la industria lo hace ideal para el aprendizaje a aquellos interesados en la materia de forma profesional.

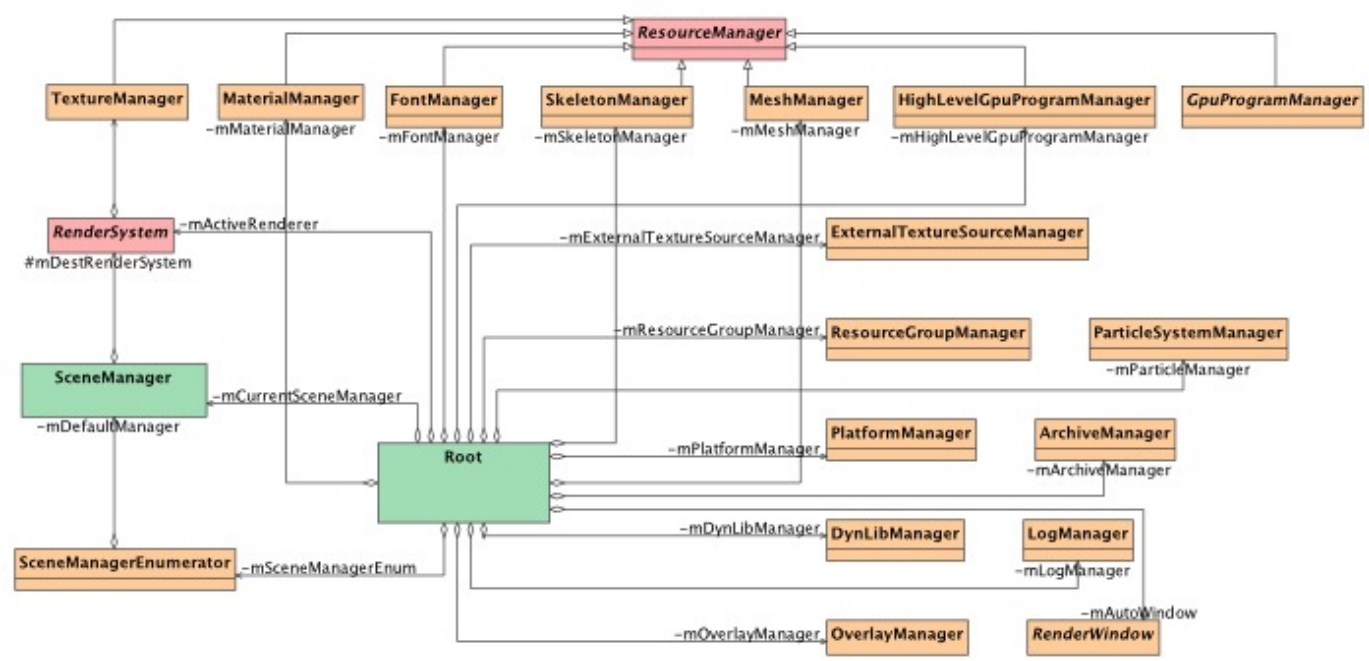


Figura 1.17: Arquitectura de OGRE3D

1.3. Motivaciones

Comenzar a utilizar una tecnología desconocida como OGRE3D o cualquier otra siempre entraña cierta dificultad y, si le añadimos la barrera del idioma, aún más. La principal motivación para embarcarme en la confección de una documentación sobre desarrollo de videojuegos en 3D como es **IberOgre** ha sido la ausencia de información al respecto en castellano. Hasta ahora sólo se habían publicado pequeños tutoriales en blogs personales pero no existía una plataforma de aprendizaje y documentación equivalente a la wiki oficial de OGRE3D [30] en español.

Diseño de Videojuegos es una asignatura optativa de tercer curso de la titulación Ingeniería Técnica en Informática de Sistemas dentro de la Universidad de Cádiz. En dicha asignatura los alumnos se organizan en grupos de tres con el objetivo de desarrollar un juego sencillo en dos dimensiones durante el cuatrimestre. Las únicas restricciones son que el videojuego resultante debe ser completamente libre y que han de utilizar la forja de código de RedIRIS [33] junto a *Subversion* [36] como sistema de control de versiones. Los alumnos de la asignatura suelen utilizar bibliotecas libres para desarrollar aplicaciones multimedia en dos dimensiones como LIBSDL [34] o GOSU [23]. Toda la documentación estaba en inglés, pero para la primera de ambas bibliotecas ya se publicó una excelente documentación en castellano en formato wiki llamada **Wikijuegos** [2].

De esta manera, los alumnos podían seguir la extensa documentación sin muchos problemas a la vez que trabajaban sobre los múltiples ejemplos aprendiendo conceptos que, más tarde, incorporarían en sus proyectos. No obstante, cuando terminasen la asignatura no disponían de una bibliografía en español para continuar ampliando sus conocimientos en el terreno de las tres dimensiones.

Cualquiera que desee iniciarse en el mundo del desarrollo de aplicaciones en tiempo real en tres dimensiones (ya sean videojuegos o no) debe adquirir o recordar una fuerte base de matemáticas. Principal-

mente, son necesarios conocimientos relativos a álgebra lineal, trigonometría y geometría del espacio. En numerosas ocasiones también nos vemos obligados a utilizar conceptos de física como óptica, cinemática, dinámica, etc. Lo usual es que el aprendiz se va obligado a recurrir a varias fuentes de información no conexas entre sí. Una motivación importante era proporcionar en un sólo lugar, el uso de una tecnología y los conocimientos matemáticos necesarios para trabajar con ella.

La wiki oficial de OGRE3D [30] y el resto de documentación tanto accesible en la red como impresa se centran principalmente en el desarrollo para plataformas cerradas como es Windows con herramientas privativas tales como Microsoft Visual Studio. Lo natural en una documentación libre es tratar de ofrecer un soporte multiplataforma utilizando herramientas libres como, por ejemplo, el compilador GNU GCC [47].

Mi interés en aprender a desarrollar videojuegos en tres dimensiones fue el motor principal que me llevó a desarrollar **Sion Tower**. Son tantos los conceptos nuevos, las tecnologías desconocidas y las aproximaciones diametralmente opuestas que las posibilidades de adquirir conocimiento son enormes. En cualquier caso, **IberOgre** necesitaba un gran ejemplo final que pusiera en práctica todo lo visto a lo largo de las lecciones. Una puesta en práctica lo más cercana a la realidad completamente documentada para que cualquier lector pudiera comprender de primera mano cómo se desarrolla un videojuego en 3D (aunque sea de pequeñas dimensiones).

La idea de que **Sion Tower** tuviera componentes fácilmente reutilizables por los lectores interesados también resultaba muy atractiva. La posible distribución por separado de subsistemas libres documentados en detalle como el motor de colisiones quizás fuera útil a otros usuarios que estuviesen comenzando.

Es posible resumir las motivaciones que me han llevado a desarrollar este proyecto en cubrir el vacío de documentación en castellano, el hecho de que pocas fuentes hagan hincapié en herramientas libres y el deseo personal de aprender a crear videojuegos en tres dimensiones.

1.4. Objetivos

El objetivo principal del proyecto viene extraído de forma lógica de las motivaciones (sección 1.3). Con **IberOgre** se pretende crear una documentación libre que facilite el aprendizaje del desarrollo de videojuegos en 3D. Por su parte, con **Sion Tower** se desea proporcionar una aplicación práctica ampliamente documentada y accesible de todo el contenido tratado en la wiki.

IberOgre debe cumplir con los siguientes requisitos básicos:

- El contenido debe abarcar los principales subsistemas de OGRE3D de forma que el lector sea capaz de desarrollar aplicaciones en tres dimensiones y pueda ampliar sus conocimientos de forma autónoma. Para ello deben darse pocos conceptos por supuestos y explicar los temas más básicos en primer lugar para ir ascendiendo en complejidad de forma progresiva. A pesar de ello, para continuar las lecciones será imprescindible tener una mínima experiencia con videojuegos en dos dimensiones.

- Debe proporcionarse una sección con los contenidos matemáticos mínimos sobre geometría del espacio para comprender el resto de material de la wiki. No se pretende publicar material educativo sobre matemáticas en el sentido más estricto sino ofrecer los conceptos con un enfoque eminentemente práctico. El nivel ofrecido será similar al impartido en segundo de bachillerato.

- OGRE3D es simplemente un motor de renderizado y no proporciona subsistemas imprescindibles para la creación de un videojuego como entrada de usuario o sonido. En la documentación debe existir una sección para otras tecnologías que cubran bibliotecas complementarias a OGRE3D.
- Emplear una aproximación lo más práctica posible. Deben intercalarse los contenidos teóricos del desarrollo de cada lección con pequeños ejemplos. Para finalizar cada artículo debe adjuntarse un ejemplo de mayores dimensiones que evidencie las técnicas comentadas en el texto.
- Prestar especial atención y soporte a herramientas de desarrollo libre y multiplataforma.
- Ofrecer técnicas y buenas prácticas para el desarrollo de software multiplataforma. Todas las herramientas y bibliotecas empleadas lo son y carecería de sentido que el código escrito por los lectores fuera dependiente de una arquitectura o plataforma sin necesidad de ello.
- Construir una comunidad que colabore de forma activa con la documentación probado ejemplos, informando de errores encontrados y creando nuevo contenido de interés para el resto de usuarios.

Sion Tower debe cumplir con los siguientes requisitos:

- Servir de ejemplo lo más cercano a la realidad posible de un desarrollo de videojuego 3D. Es muy habitual terminar de leer documentación, hacer pequeñas prácticas pero no sentirse capaz de embarcarse en un proyecto de mayores dimensiones utilizando la misma tecnología. Para ello es imprescindible que se documente cada fase del desarrollo.
- Crear un pequeño motor orientado a la producción de contenido, una de las prácticas mejor valoradas. Cualquier diseñador podría añadir niveles al juego sin necesidad de tocar el código fuente ni recompilarlo.
- Convertirse en un ejemplo de trabajo con un equipo multidisciplinar. En un videojuego profesional deben trabajar juntos desde unas pocas personas hasta varios centenares por lo que la coordinación entre profesionales de distintas áreas es muy importante. Será necesario encontrar artistas relacionados con el modelado 3D, música y efectos de sonido.
- No sólo debe ser útil a los interesados en el desarrollo sino también de cara al usuario final. **Sion Tower** debe ser cómodo de utilizar, intuitivo y divertido.
- Crear subsistemas independientes del videojuego reutilizables por la comunidad de usuarios. Por supuesto, deben venir acompañados de la documentación pertinente: dependencias, instalación, uso y licencia.

1.5. Sobre este documento

La presente memoria de **Proyecto Fin de Carrera** posee la siguiente estructura en capítulos y apéndices:

- En el capítulo 1 que actualmente nos ocupa, se hace una breve introducción sobre el contexto en el que nos situamos. Se ofrece una visión del mundo de los videojuegos como industria y se habla sobre el actual mercado de los motores para videojuegos. Finalmente se evidencia la necesidad de documentación en castellano sobre uno de estos motores libres (OGRE3D) y se adjuntan el resto de motivaciones y objetivos relacionados con el proyecto.
- En el capítulo 2 se expone la organización temporal de todo el desarrollo a través de un diagrama de Gantt. Más adelante se comenta brevemente cada una de las tareas que conforman la planificación.

- En el capítulo 3 se documenta en profundidad el desarrollo de la wiki **IberOgre**. Se incluye la metodología utilizada, el análisis de requisitos, el diseño de la plataforma, los detalles de la redacción de los artículos, la implementación de los ejemplos y las pruebas realizadas.
- En el capítulo 4 se refleja el proceso de desarrollo para el videojuego **Sion Tower**. Comenzando con el documento de diseño, se sigue con la fase de análisis, diseño e implementación. Se hace especial hincapié en los subsistemas que entrañan mayor complejidad como el sistema de detección de colisiones. Se finaliza con las pruebas realizadas.
- En el capítulo 5 se hace una breve reflexión personal seguida de otra técnica tratando de aglutinar los conceptos aprendidos, la riqueza que ha proporcionado la experiencia y lo que podrá aportar a la comunidad. Finalmente se termina con un pequeño listado sobre las posibilidades de ampliación del proyecto.
- En el apéndice Software utilizado se hace un repaso por todas las herramientas empleadas a lo largo del desarrollo del proyecto. Asimismo, se adjuntan comentarios y las razones de su uso.
- En el apéndice Manual de usuario se incluye un completo manual de usuario para **Sion Tower**. El documento contiene la introducción a la historia, personajes y hechizos del juego así como una completa guía de instalación en sistemas GNU/Linux y Windows. Posteriormente se detallan las mecánicas de juego y los controles en detalle. Finalmente se añade una completa guía para añadir niveles adicionales a **Sion Tower** creados con la herramienta *Blender*.
- En el apéndice Comunidad y difusión se realiza un recorrido por las actividades relacionadas con el presente proyecto y la comunidad del Software Libre. Se mencionan actividades como el **V Concurso Universitario de Software Libre** y los medios de difusión utilizados.

Capítulo 2

Organización temporal

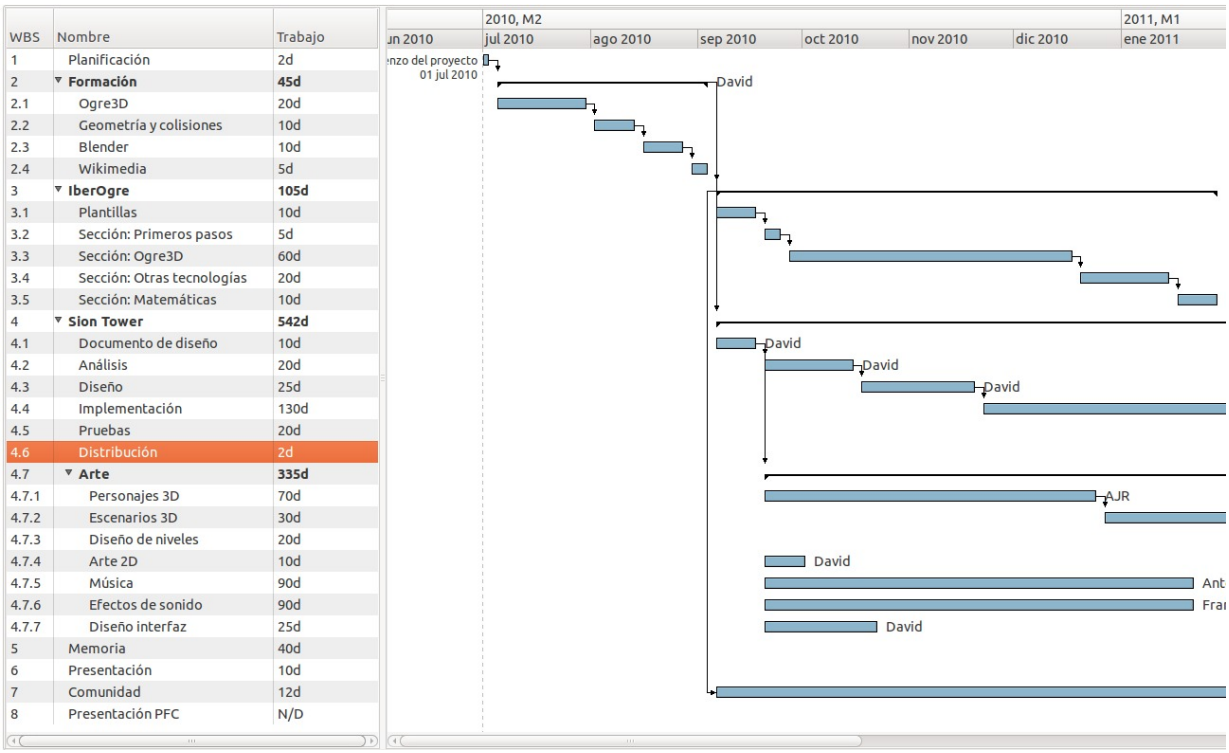
En este capítulo se expone la planificación de tareas que se ha seguido para desarrollar **IberOgre** y **Sion Tower**. En primer lugar se adjunta el diagrama de Gantt completo para, más adelante, complementarlo con un breve comentario de cada tarea.

2.1. Diagrama de Gantt

Como puede observarse en el diagrama de Gantt, en tareas relacionadas con el apartado artístico de **Sion Tower** participan más personas. Esto se debe a que no poseía ni poseo los conocimientos o destrezas requeridos para crear modelos tridimensionales animados, piezas musicales ni grabar y procesar efectos de sonido. Por ello, he contactado con expertos en dichas materias dispuestos a colaborar en el desarrollo de un videojuego completamente libre. Finalmente, los participantes adicionales son:

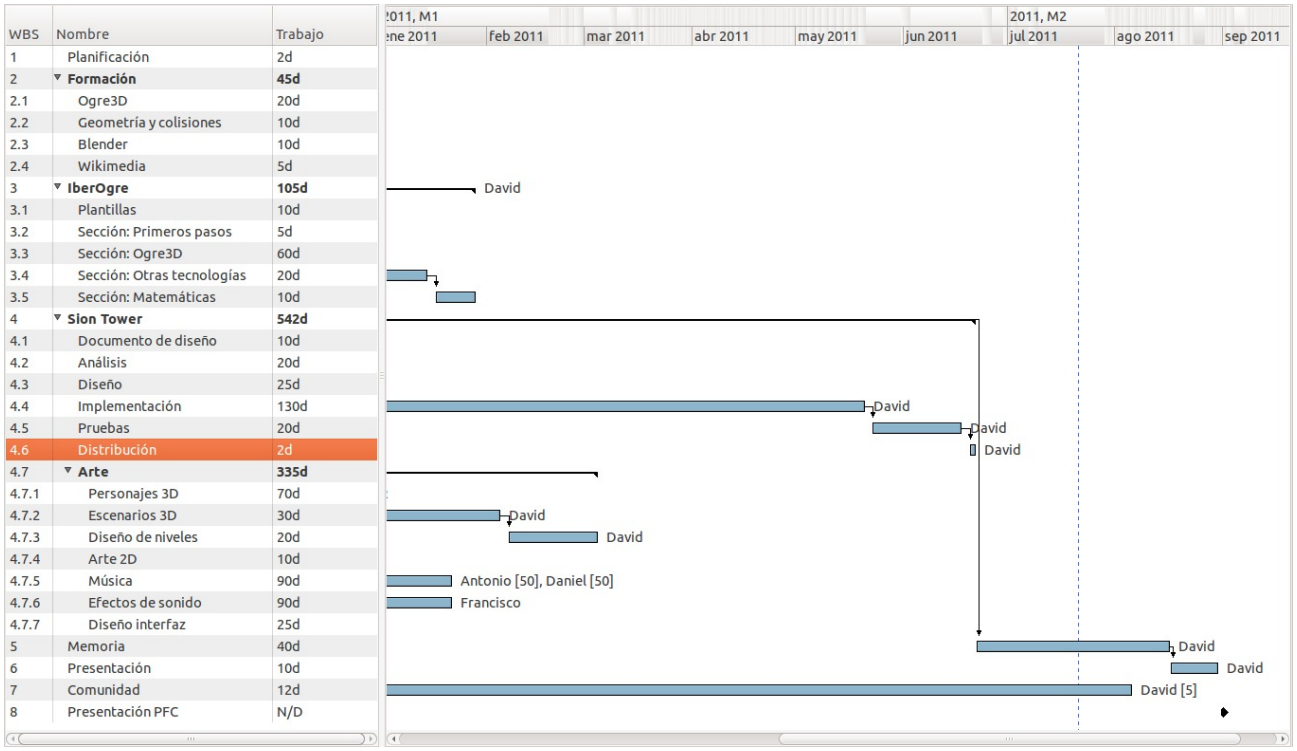
- Antonio Jiménez Rodríguez: artista 3D encargado de diseñar, modelar, texturizar al protagonista y a los enemigos del juego.
- Daniel Belohlavek: artista 2D encargado de los iconos de los hechizos.
- Daniel Pellicer García y Antonio Caro Oca: encargados de componer y producir la banda sonora completa del juego.
- Francisco Martín Márquez: técnico de sonido encargado de grabar y procesar los efectos de sonido para el juego.

Por cuestiones de espacio y formato, el diagrama de Gantt ha sido divido en dos. En la figura 2.1 puede observarse la planificación del proyecto correspondiente al intervalo comprendido entre julio de 2010 y enero de 2011. Por otro lado, la figura 2.2 muestra el resto de la planificación, desde febrero de 2011 hasta septiembre de 2011.



WBS	Nombre	Trabajo
1	Planificación	2d
2	**Formación**	**45d**
2.1	Ogre3D	20d
2.2	Geometría y colisiones	10d
2.3	Blender	10d
2.4	Wikimedia	5d
3	**IberOgre**	**105d**
3.1	Plantillas	10d
3.2	Sección: Primeros pasos	5d
3.3	Sección: Ogre3D	60d
3.4	Sección: Otras tecnologías	20d
3.5	Sección: Matemáticas	10d
4	**Sion Tower**	**542d**
4.1	Documento de diseño	10d
4.2	Análisis	20d
4.3	Diseño	25d
4.4	Implementación	130d
4.5	Pruebas	20d
4.6	Distribución	2d
4.7	**Arte**	**335d**
4.7.1	Personajes 3D	70d
4.7.2	Escenarios 3D	30d
4.7.3	Diseño de niveles	20d
4.7.4	Arte 2D	10d
4.7.5	Música	90d
4.7.6	Efectos de sonido	90d
4.7.7	Diseño interfaz	25d
5	Memoria	40d
6	Presentación	10d
7	Comunidad	12d
8	Presentación PFC	N/D

Figura 2.1: Planificación del proyecto desde julio de 2010 hasta enero de 2011



WBS	Nombre	Trabajo
1	Planificación	2d
2	**Formación**	**45d**
2.1	Ogre3D	20d
2.2	Geometría y colisiones	10d
2.3	Blender	10d
2.4	Wikimedia	5d
3	**IberOgre**	**105d**
3.1	Plantillas	10d
3.2	Sección: Primeros pasos	5d
3.3	Sección: Ogre3D	60d
3.4	Sección: Otras tecnologías	20d
3.5	Sección: Matemáticas	10d
4	**Sion Tower**	**542d**
4.1	Documento de diseño	10d
4.2	Análisis	20d
4.3	Diseño	25d
4.4	Implementación	130d
4.5	Pruebas	20d
4.6	Distribución	2d
4.7	**Arte**	**335d**
4.7.1	Personajes 3D	70d
4.7.2	Escenarios 3D	30d
4.7.3	Diseño de niveles	20d
4.7.4	Arte 2D	10d
4.7.5	Música	90d
4.7.6	Efectos de sonido	90d
4.7.7	Diseño interfaz	25d
5	Memoria	40d
6	Presentación	10d
7	Comunidad	12d
8	Presentación PFC	N/D

Figura 2.2: Planificación del proyecto desde febrero de 2011 hasta septiembre de 2011

2.2. Etapas de desarrollo del proyecto

1. **Planificación**

   Dada la envergadura del proyecto, era necesaria una etapa de planificación en la que se ha estudiado de forma cuidadosa el alcance del mismo y las posibles dificultades a encontrar durante el desarrollo.

2. **Formación**

   Al comienzo del proyecto desconocía por completo el uso de bibliotecas imprescindibles como OGRE3D, OIS o MYGUI y herramientas importantes como *Blender*. Tampoco conocía con la necesaria profundidad los fundamentos del desarrollo de videojuegos en tres dimensiones. Fue necesaria, por tanto, una larga etapa de formación personal utilizando varios recursos bibliográficos y pequeñas pruebas prácticas.

3. **IberOgre**

   Tras el periodo de formación inicial, dio comienzo el montaje de la wiki **IberOgre**. La Oficina de Software Libre de la Universidad de Cádiz (OSLUCA) llevó a cabo la instalación de la plataforma sobre sus servidores y, desde dicho momento, se pudo comenzar a trabajar en la documentación de OGRE3D.

   a) *Plantillas*

   Fue necesario transformar la apariencia que incluye una plataforma *Wikimedia* por defecto para conseguir un aspecto propio en **IberOgre**. Para ello se modificaron las hojas de estilo CSS, se trabajó sobre la portada y se crearon varias plantillas. Estas plantillas proporcionan una forma sencilla de incluir estructuras en varios puntos del sitio como ejemplos, tablas especiales, etc.

   b) *Sección: Primeros pasos*

   Tras preparar la wiki, se procedió a la redacción del artículo correspondiente a la primera sección de **IberOgre**. En dicho artículo se daba una bienvenida a la documentación y se explicaban sus motivaciones, objetivos y estructura en cuanto a lecciones. De esta manera, el lector podría conocer la filosofía de trabajo de la plataforma.

   c) *Sección:* OGRE3D

   Posteriormente se comenzó a elaborar los artículos pertenecientes a la sección del motor OGRE3D junto a sus ejemplos prácticos. Estos textos cubren prácticamente la totalidad del uso del motor en cuanto a aspectos básicos se refiere: instalación, arquitectura, inicialización, configuración, creación de escenas, animación, iluminación, materiales, efectos de partículas, etc.

   d) *Sección: Otras tecnologías*

   OGRE3D no es un motor de videojuegos completo, sólo es un motor de renderizado. Por tanto, el uso de tecnologías complementarias es prácticamente obligado. En **IberOgre** se han elaborado artículos para asistir a los lectores en dichas tecnologías como OIS para gestión de dispositivos de entrada o SDL MIXER para música y efectos de sonido.

e) *Sección: Matemáticas*

En la sección de matemáticas se presentan conceptos fundamentales para el desarrollo e videojuegos. Sobre todo se cubre de manera ligera temas relacionados con la geometría del espacio: puntos, vectores, matrices y cuaternos. Siempre se ofrecen aplicaciones prácticas dentro del ámbito que nos ocupa.

4. **Sion Tower**

Tras el periodo de aprendizaje, casi de manera simultánea al trabajo en **IberOgre**, se comenzó con el videojuego **Sion Tower**. El primer paso fue solicitar la creación de un nuevo proyecto en la *Forja de RedIRIS* y esperar su aprobación. Dicha forja proporciona un repositorio *Subversion* y herramientas web que ayudan en la gestión de un proyecto libre: gestor de tareas, subida de ficheros, publicación de noticias, etc.

a) *Documento de diseño*

En el documento de diseño de un videojuego se detallan elementos como la historia, género, personajes, mecánicas de juego, objetos o enemigos entre otros muchos. En definitiva, es el documento que define de forma más o menos concisa cómo será el videojuego. Se trata de un escrito muy importante ya que ayuda a que todo el equipo albergue la misma idea sobre el videojuego y pueda trabajar de forma más compenetrada.

b) *Análisis*

La fase de análisis dio comienzo tras la redacción y revisión del documento de diseño. Se procedió con la toma de requisitos a partir de dicho documento, se confeccionaron los casos de uso, se elaboró el modelo conceptual de datos y se detalló el modelo de comportamiento.

c) *Diseño*

Durante la fase de diseño, que siguió de forma inmediata al análisis, se elaboraron los diagramasde clases de diseño.

d) *Implementación*

La fase de implementación de **Sion Tower** fue, con diferencia, la más extendida de todo el desarrollo. Quizás viniese motivada por la inexperiencia y por el gran número de subsistemas que se implementaron desde cero.

e) *Pruebas*

Durante la implementación se fueron realizando pruebas de módulos individuales pero fue tras finalizar ficha fase cuando tuvieron lugar las pruebas de integración. No sólo se trabajó para que el código fuese correcto sino que **Sion Tower** fue probado de forma extensiva por colaboradores distintos al desarrollador principal con el claro objetivo de pulir ciertos detalles relacionados con el balanceo de características (personajes muy débiles o poderosos) o la comodidad del control entre otros.

f) *Mantenimiento*

Tras reunir sugerencias del público interesado en el proyecto se realizaron pequeñas mejoras y ajustes que en pocas ocasiones implicaron cambios en el código fuente. Sobre todo, estaban relacionados con el balanceo de personajes.

g) *Distribución*

Se crearon y publicaron paquetes descargables tanto para sistemas GNU/Linux como para Windows. La dificultad que conllevaba empaquetar una aplicación OGRE3D en Debian provocó que se tuviera que distribuir un paquete con el código fuente en GNU/Linux. Para Windows existe un paquete con el ejecutable `.exe`, los ficheros multimedia y las dependencias en forma de bibliotecas dinámicas.

h) *Arte*

El proceso de creación de todo el arte necesario para **Sion Tower** comenzó nada más acabar la redacción del documento de diseño. Desde ese momento se conocía el estilo visual del juego, las pantallas y los personajes que aparecerían. Por su complejidad, el trabajo se extendió prácticamente durante todo el desarrollo. Al ser la única tarea en la que participaron colaboradores, fueron necesarias labores de supervisión y coordinación: estilo, formato de entrega, corrección de fallos, etc.

5. **Memoria**

   Tras el desarrollo de **IberOgre** y **Sion Tower** se procedió a la redacción del presente documento que incluye apéndices adicionales como el manual de usuario del videojuego.

6. **Presentación**

   Como última tarea en la organización temporal figura la elaboración de la exposición de cara a la presentación del **Proyecto fin de Carrera**. Se llevó a cabo tratando de plasmar el trabajo realizado y los objetivos conseguidos con el desarrollo de este proyecto.

7. **Comunidad**

   Una de las partes fundamentales de este proyecto es su objetivo de servir a la comunidad. Hasta el momento no existía una documentación extensa sobre OGRE3D y el desarrollo de videojuegos tridimensionales en castellano. A lo largo de todo el desarrollo ha estado presente la atención e interacción con la comunidad a través de diversos medios como redes sociales, la propia wiki y el blog.

8. **Presentación (hito)**

   Finalmente, el día 1 de septiembre de 2011 se entrega toda la documentación del proyecto y se preparó la presentación final que tendría lugar, aproximadamente, una semana después.

Capítulo 3

Desarrollo de IberOgre

3.1. Metodología

IberOgre es un proyecto de documentación y, por tanto, no hemos podido aplicar una metodología de desarrollo típica de sistemas software como el RUP (*Rational Unified Process*) empleado en **Sion Tower** más adelante. Podemos distinguir dos partes bien diferenciadas en este proyecto aunque inseparables: el texto de los artículos y los ejemplos que los acompañan.

Para la composición de todos los artículos se ha seguido un proceso uniforme para asegurar la calidad de los mismos:

1. **Estudio de necesidades**: búsqueda de los puntos más relevantes a la hora de aprender a desarrollar videojuegos en tres dimensiones así como los apartados más fundamentales de la biblioteca OGRE3D.
2. **Planificación**: estudio para encontrar la mejorar forma de abordar la materia escogida. División del contenido en una estructura uniforme con el resto de la plataforma de aprendizaje.
3. **Redacción y formateado**: recopilación de información y redacción del artículo completo. Adaptación del texto al formato de la plataforma, composición del artículo con imágenes que amenicen la lectura y publicación.
4. **Desarrollo del ejemplo**: creación de un ejemplo descargable que ilustre todo el contenido expuesto en el texto del artículo de forma práctica. Creación de documentación adicional en el propio artículo para explicar el funcionamiento del ejemplo.
5. **Evaluación**: tras la publicación del artículo se buscan las impresiones de los lectores y se anotan todas las sugerencias para mejorar el texto.

3.2. Análisis

En esta sección expondremos el análisis de la documentación sobre desarrollo de videojuegos en tres dimensiones **IberOgre**. En primer lugar realizaremos una breve especificación de requisitos del sistema, posteriormente hablaremos de los tipos de usuarios que accederán a la plataforma y finalmente haremos un listado de los artículos necesarios.

3.2.1. Especificación de requisitos del sistema

Requisitos de interfaces externas

Los lectores accederán al contenido de **IberOgre** mediante su monitor, empleando el navegador de su elección. Por tanto, es importante que la plataforma emplee estándares web soportados por todos los navegadores mayoritarios: Internet Explorer, Mozilla Firefox, Google Chrome, Opera y Safari. Para conseguirlo, será necesario escoger un sistema de publicación compatible con HTML 4.0 y CSS 2. En la medida de lo posible, deben seguirse las directrices del *World Wide Web Consortium* (W3C) [39].

El contenido deberá poder ser visualizado de forma clara en monitores de varias resoluciones y relaciones de aspecto. El texto debe aparecer de forma limpia, clara y poco recargada. No obstante, debe estar acompañado de imágenes que amenicen la lectura de los temas más áridos. En caso de que se adjunte código fuente, éste deberá poder visualizarse en la misma plataforma empleando un resaltado de sintaxis apropiado.

La navegación debe ser intuitiva y no precisar de aclaraciones de ningún tipo. El contenido debe estar perfectamente organizado en secciones temáticas y ordenadas en dificultad creciente, de forma prácticamente secuencial.

3.2.2. Requisitos sobre usuarios

No todos los usuarios que accedan a **IberOgre** tendrán los mismos privilegios. Existirán tres categorías de usuarios cuyos privilegios se detallan a continuación.

- **Lector**: usuario ocasional no registrado en la plataforma.
  - Acceso de lectura a los artículos publicados y a las secciones de la wiki.
  - Posibilidad de enviar sugerencias de mejora.
- **Usuario registrado**:
  - Acceso de lectura a los artículos publicados y a las secciones de la wiki.
  - Posibilidad de enviar sugerencias de mejora.
  - Acceso de escritura a artículos existentes con el objetivo de ampliarlos, corregirlos y/o mejorarlos.
  - Creación de nuevos artículos siguiendo la estructura general de la wiki.
- **Administrador**:
  - Acceso de lectura a los artículos publicados y a las secciones de la wiki.
  - Posibilidad de enviar sugerencias de mejora.
  - Acceso de escritura a artículos existentes con el objetivo de ampliarlos, corregirlos y/o mejorarlos.
  - Creación de nuevos artículos siguiendo la estructura general de la wiki.
  - Control sobre la estructura de la plataforma.
  - Instalación de nuevas extensiones para mejorar la plataforma.
  - Gestión del resto de usuarios.

Cualquier persona que acceda al sitio web de la plataforma será considerado como lector no registrado. En el momento en el que rellene el formulario y envíe sus datos se convertirá en usuario registrado. Lo habitual es que sólo exista un administrador del sistema o exista un grupo reducido de administradores.

3.2.3. Artículos

Los artículos de **IberOgre** están organizados en varios bloques temáticos para abarcar todos los objetivos que nos hemos marcado en la introducción de esta memoria. Esta organización también busca facilitar la navegación al usuario de forma que acceda al contenido que desee en el menor tiempo posible. Los bloques principales de **IberOgre** son:

- **Primeros pasos**: bloque introductorio en el que el lector debe poder conocer la estructura, filosofía y metodología de la plataforma de aprendizaje. Tras leerlo, el usuario debe poder saber qué esperar de **IberOgre** y cómo afrontar los contenidos posteriores.
- **Programación de videojuegos 3D**: sección que se dedicará a ofrecer los conocimientos matemáticos mínimos para desarrollar videojuegos en 3D. Se centrará en conceptos generales de geometría del espacio y álgebra lineal desde una perspectiva eminentemente práctica. Las matemáticas son un medio, una herramienta más para conseguir los objetivos propuestos. Al no ser el fin en sí mismo, priman las aplicaciones prácticas por encima de la rigurosidad (sin llegar a ser incorrectos).
- **Ogre3D**: bloque central que desglosa los apartados más relevantes del motor de renderizado OGRE3D. Tras terminar con este bloque, el lector debe ser capaz de utilizar la biblioteca para crear aplicaciones tridimensionales con elementos como modelos animados, sistemas de partículas y otros efectos de iluminación. Lo más importante es que se comprenda el funcionamiento del motor y se sea capaz de ampliar conocimientos de forma autónoma.
- **Otras tecnologías**: como ya se ha mencionado anteriormente, OGRE3D es sólo un motor de renderizado, no proporciona todos los elementos necesarios para construir un videojuego como gestión de entrada del usuario, sonido, detección de colisiones o juego en red. Por ello, esta sección está enfocada a introducir en el uso de tecnologías complementarias a OGRE3D como OIS (entrada), LIBSDL MIXER (audio) u OGREBULLET (físicas).
- **Videojuegos**: en esta sección se adjuntará documentación sobre el desarrollo de videojuegos libres que hagan uso de OGRE3D. **Sion Tower** contará con el primer artículo de esta sección. El videojuego pretende ser una aplicación práctica lo más realista posible del uso del motor y de la creación de videojuegos 3D en general. Adjuntaremos toda la documentación que se ha generado al respecto para que el lector pueda acceder a la misma de forma sencilla. Sin duda, es el bloque que se presta en mayor medida a ampliaciones futuras por parte de la comunidad.

La sección *Primeros pasos* únicamente cuenta con un artículo introductorio.

- **Comenzando en IberOgre**: en este artículo a modo de preámbulo desgranamos todos los objetivos de la plataforma de aprendizaje. Posteriormente se explica la filosofía de trabajo que deseamos tomar con software compatible con varias plataformas y de carácter abierto.

En el bloque *Programación de videojuegos 3D* se encuentran tres artículos.

- **Introducción, puntos y vectores**: en el primer artículo sobre programación de videojuegos en 3D se hace una introducción al planteamiento que vamos a seguir durante todo el bloque. Posteriormente tratamos los puntos y los sistemas de coordenadas. Más adelante abarcamos los vectores,

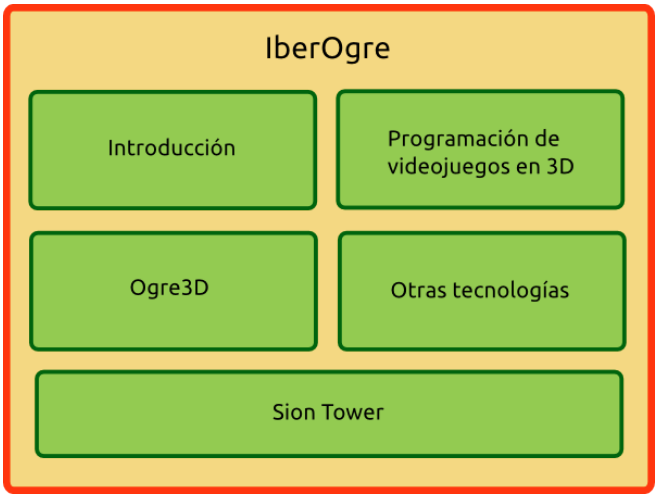


Figura 3.1: Bloques temáticos de IberOgre

las operaciones que podemos realizar con ellos y sus aplicaciones en la materia. Finalmente se dedica una pequeña sección a hablar sobre interpolaciones lineales para suavizar movimientos.

- **Matrices**: en este artículo definimos el concepto de matriz desde el punto de vista matemático para después hacer un repaso por sus operaciones y finalmente tratar su utilidad a la hora de representar transformaciones en tres dimensiones (traslaciones, rotaciones y escalas).
- **Cuaternos**: los cuaternos son una extensión de los números reales que se suelen utilizar a la hora de representar rotaciones en un espacio tridimensional. En el artículo explicamos su definición, la forma de encadenar rotaciones y deshacerlas así como la operación necesaria para realizar interpolaciones y rotar un elemento de forma progresiva.

En el bloque principal, *Ogre3D* tenemos los siguientes artículos.

- **Conociendo Ogre3D**: en el primer artículo relacionado con el motor de renderizado debe explicarse qué es capaz de hacer por nosotros OGRE3D y cuáles son sus limitaciones. Se expone una lista de características que incorpora así como los requisitos de hardware que exige para poder utilizarse. Comentamos su licencia permisiva y los lenguajes con los que es compatible (aunque nosotros sólo utilizaremos C++). Por último, hablamos brevemente de varias alternativas libres a OGRE3D como PANDA 3D o IRRLICHT.
- **Conceptos generales**: en este texto hacemos un recorrido por la arquitectura del motor, sus subsistemas más relevantes y la filosofía que se ha tomado a la hora de diseñarlos. Explicamos de forma rápida su objeto raíz, el gestor de recursos y el grafo de escena.
- **Instalación de Ogre3D 1.7 en GNU/Linux**: el lector debe poder acceder a una sencilla, rápida y directa guía de cómo instalar el entorno de desarrollo en su equipo para poder continuar con las lecciones de la plataforma de aprendizaje. En este artículo se expone el proceso de instalación para sistemas GNU/Linux. Se incluye un apartado especial para la popular distribución Ubuntu, ya que muchas de las dependencias están en los repositorios oficiales o en repositorios *PPA* [54].
- **Instalación de Ogre3D 1.7 en Windows**: los usuarios de Windows también deben contar con una guía de instalación del entorno de desarrollo completo para poder seguir los contenidos de **IberOgre**. En este caso es necesario instalar el compilador *MinGW* y las bibliotecas OGRE3D, BOOST y DIRECTX. Se precisa adjuntar un proyecto de prueba para comprobar la corrección del proceso.

- **Creación de un entorno de trabajo multiplataforma**: es altamente deseable que los usuarios de GNU/Linux y Windows trabajen de la misma forma siguiendo el contenido de la wiki. Para ello, los proyectos debían tener la misma estructura, el mismo código compatible y un proceso de compilación similar. En este artículo se explica la jerarquía de directorios que tomaremos y los makefiles para compilar ejemplos y proyectos personales.
- **Inicialización y cierre de Ogre3D**: en este artículo creamos por primera vez una aplicación básica con OGRE3D. Se repasa la secuencia de inicialización y cierre del motor. Así mismo, conocemos las posibilidades de extensión del motor gracias al uso de plugins y complementos externos. Posteriormente aprendemos a configurar aspectos como la resolución, la frecuencia, la sincronización vertical utilizando ficheros de configuración. Finalmente se explica el sistema de logs de OGRE3D para depurar nuestras aplicaciones.
- **Gestión de recursos**: el motor incluye un sistema de gestión de recursos para optimizar el consumo de memoria evitando duplicidades y proporcionar un método uniforme de acceder a elementos como mallas tridimensionales, materiales o esqueletos. Los usuarios del motor deben conocer como gestionar el ciclo de vida de los recursos para poder cargar modelos entre otras muchas cosas.
- **Creación básica de escenas**: este artículo retoma la inicialización de OGRE3D y continua con el objetivo de ilustrar la creación de escenas. Debe mostrar como crear una ventana, un gestor de escenas, una cámara y un punto de vista. Una vez esté preparado el grafo de la escena para que se le añadan elementos, se hace un recorrido por la creación y gestión de nodos de escena con modelos tridimensionales estáticos. Se hace una breve introducción a la iluminación y se detallan las distintas aproximaciones para abordar el bucle de renderizado (bucle de juego).
- **Materiales**: nuestra biblioteca cuenta con un sistema de materiales que, básicamente, definen cómo un objeto interacciona con la luz que recibe. En OGRE3D existe una pequeña sintaxis para definir materiales. En este artículo hablamos de los detalles de los scripts para definir materiales y sus posibles parámetros así como de las maneras para cargar dichos materiales y aplicarlos a entidades.
- **Manipulación de nodos**: las escenas de OGRE3D están formadas por nodos que contienen mallas tridimensionales, puntos de luz, cámaras u otros elementos. Para dotar de dinamismo a las escenas estos nodos deben trasladarse, ser rotados o escalados. En definitiva, deben ser dotados de movimiento. En este artículo hacemos un repaso por cómo se representan los vectores y transformaciones en el motor y comenzaremos a gestionar los nodos de la escena. Se exponen las diferencias entre los distintos espacios de transformación (local, parental y global) y aprendemos a aplicar transformaciones a los nodos (traslación, rotación y escalado). Finalmente se trata el tema de la interpolación entre puntos para producir movimiento suavizado.
- **Luces, sombras y entorno**: todos los videojuegos tridimensionales utilizan efectos de iluminación para crear la ambientación deseada. Era imprescindible incluir un artículo al respecto en **IberOgre**. En la primera sección se habla sobre las distintas técnicas de sombreado disponibles en el motor. Posteriormente abarcamos el tema de la iluminación desde conceptos básicos como la reflexión difusa y especular hasta la creación de puntos que emitan luz en OGRE3D. El siguiente apartado corresponde al efecto de niebla, muy utilizado para ahorrar recursos o para crear una atmósfera terrorífica. Por último, se habla de los fondos en dos dimensiones para representar elementos como el cielo.
- **Animación**: la animación es un tema muy complejo pero imprescindible en todo videojuego. El artículo comienza con los conceptos básicos sobre la animación como el uso de fotogramas clave e interpolaciones entre los mismos. Posteriormente explicamos los tres tipos de animación que existen en OGRE3D y cual es en la que nos centraremos durante el resto del texto. A continuación explicamos la técnica para cargar animaciones en el motor, reproducirlas y manipularlas. Por

último, se hace una introducción a la mezcla de animaciones (*animation blending*).

- **Sistemas de partículas**: en este artículo se abordan los sistemas de partículas de OGRE3D empleados para reproducir efectos como fuego, explosiones, humos, estelas mágicas, etc. Estos efectos se definen en scripts similares a los de los materiales y explicamos su estructura y sintaxis. Tras hacerlo comentamos el método para crear y destruir sistemas de partículas. Finalmente mencionamos un potente editor que nos ahorra tener que modificar de forma manual estos efectos.
- **Sistema de Overlays**: OGRE3D carece de sistema para crear elementos avanzados de interfaz como botones, etiquetas o formularios completos. En cambio, posee un sistema de overlays para mostrar elementos bidimensionales en una escena tridimensional. Estos paneles se definen también empleando scripts cuya estructura, sintaxis y propiedades detallamos a lo largo del artículo. Una vez terminadas las propiedades mostramos cómo cargar y gestionar overlays dentro del juego. Este sencillo sistema puede ser útil para usuarios que no requieran interfaces demasiado complejas.

Dentro de la sección *Otras tecnologías* tenemos los siguientes artículos.

- **Manejo básico de OIS**: es una biblioteca libre de gestión de dispositivos de entrada como joystick, ratón o teclado. Es la recomendada por OGRE3D, de hecho se distribuyen juntas. En este artículo aprendemos a consultar el estado de los dispositivos de entrada así como a responder a los eventos que estos producen en el momento adecuado empleando el patrón Observer [15].
- **Exportar modelos desde Blender**: cuando creamos un modelo tridimensional posiblemente animado en una herramienta como *Blender* debemos exportarlo al formato que emplea OGRE3D. En este artículo se detalla el proceso con todas las opciones que nos proporcionan los scripts de exportación.
- **Colisiones y físicas con OgreBullet**: como ya se ha mencionado, OGRE3D no proporciona un sistema de detección de colisiones ni simulaciones físicas. Por tanto, este artículo está destinado a documentar la instalación y el uso de la biblioteca OGREBULLET, un envoltorio de la popular BULLET.
- **Extender la gestión de recursos, audio**: el sistema de gestión de recursos de OGRE3D permite ser extendido con sencillez para añadir nuevos tipos de recursos. Aprovechando esta capacidad explicamos el proceso incluyendo un sistema de audio utilizando la biblioteca LIBSDL en conjunción con LIBSDL MIXER.

3.3. Diseño

Una vez completado el análisis con los requisitos de **IberOgre** nos centraremos en el diseño de sus componentes. En primer lugar decidiremos la estructura interna de cada uno de los artículos que compondrán la plataforma educativa de desarrollo de videojuegos 3D. Posteriormente pasaremos al diseño de la navegabilidad del sitio y de su apariencia visual. Finalmente especificaremos los ejemplos con los que contarán los artículos.

3.3.1. Estructura de los artículos

La estructura lógica de los artículos queda reflejada en la figura 3.2 y está pensada para que el lector adquiera de la manera más sencilla posible los conocimientos necesarios para continuar. Dicha estructura ha ido siendo refinada a lo largo del desarrollo gracias al continuo flujo de opciones por parte de los lectores. La estructura consta de las siguientes partes.

- **Introducción**: breve resumen del campo que abarcará el artículo para que el lector sepa qué esperar cuando lo siga. Siempre se pretende ofrecer una perspectiva práctica dando ejemplos concretos de aplicaciones reales del contenido.
- **Requisitos previos**: breve sección con los artículos que el lector debería conocer antes de continuar. La estructura de **IberOgre** es aproximadamente secuencial pero en muchas ocasiones existen dependencias adicionales. Es importante que el lector sepa qué conocimientos debe poseer antes de enfrentarse a un texto sin entender nada.
- **Desarrollo**: bloque principal del artículo, que a su vez puede estar compuesto de secciones, en el que se desarrolla el tema a tratar. Se intercalan las explicaciones teóricas con pequeños fragmentos de código mostrando de forma práctica las técnicas comentadas.
- **Ejemplo final**: en todos los artículos se adjunta un ejemplo final en forma de aplicación descargable que ilustra el contenido expuesto en el desarrollo. Al fichero descargable lo acompaña una explicación sobre cómo se ha implementado el ejemplo y se hacen sugerencias para que el lector modifique elementos. Este aspecto es muy relevante ya que la práctica es mucho más efectiva en el proceso de aprendizaje.
- **Conclusiones**: finalmente se incluye una sección que hace las veces de resumen de los contenidos tratados y lista las tareas que el lector debe ser capaz de realizar tras comprender el contenido del artículo.

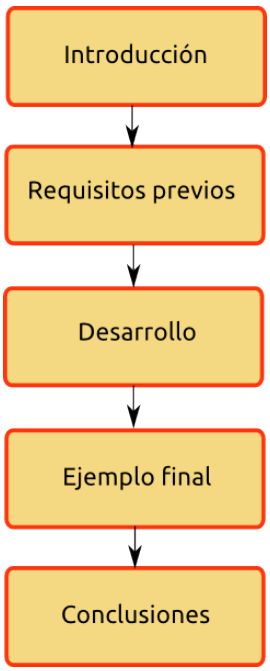


Figura 3.2: Estructura de los artículos en IberOgre

3.3.2. Navegabilidad

Para **IberOgre** se ha elegido el motor *MediaWiki* [41] por ser uno de los más empleados en este tipo de plataformas, contar con funcionalidades más atractivas de personalización o edición y disponer de una comunidad más numerosa dispuesta a ayudar en caso de problemas. Su diseño visual por defecto ha sido modificado de forma que se adapte en la mayor medida posible a la estructura de **IberOgre** y a su

enfoque.

En esta sección haremos un recorrido por el diseño de **IberOgre** comenzando por los bloques de su página principal. La pantalla de bienvenida prácticamente al completo puede observarse en la figura 3.3, en ella se aprecia claramente la división por bloques.

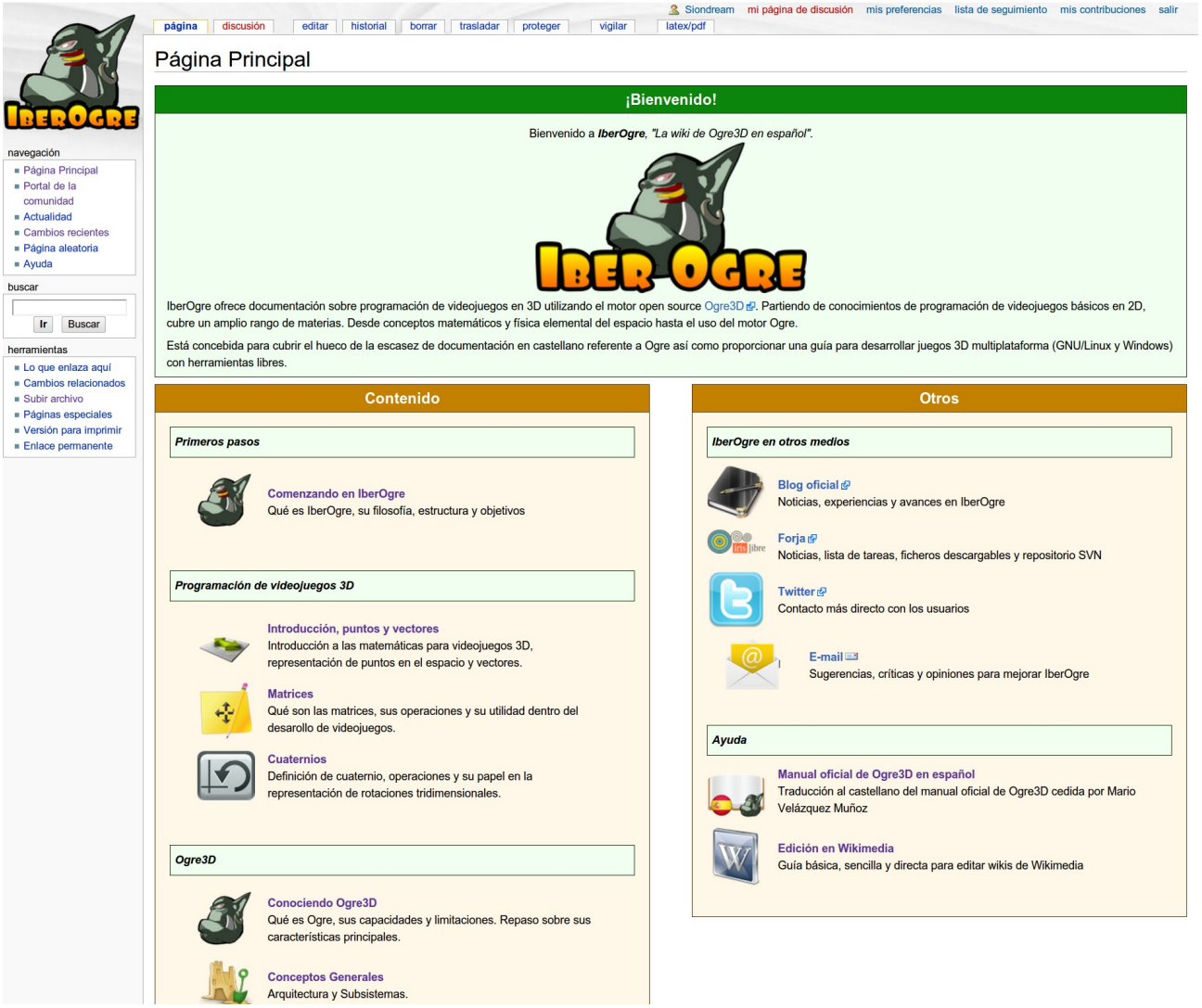


Figura 3.3: Diseño de IberOgre

Bloque de bienvenida

En el bloque de bienvenida se muestra el logo de **IberOgre** y se hace una pequeña introducción en pocas líneas sobre el tema que trata la plataforma. Es sencilla y directa para captar el interés del lector potencial. Puede verse con mayor detalle en la figura 3.4.

Figura 3.4: Bloque de bienvenida a IberOgre

Navegación

El bloque de navegación se sitúa a la izquierda de la página y es útil para acceder de forma directa a distintas secciones de interés para los usuarios de la plataforma. Puede verse en la figura 3.5 y se compone de los siguientes enlaces:

- **Página principal**: accede a la bienvenida de la wiki.
- **Portal de la comunidad**: página con información de interés para los lectores sobre el trabajo actual en **IberOgre**.
- **Actualidad**: sección de noticias.
- **Cambios recientes**: permite ver cuáles son las páginas nuevas o las últimas ediciones, en qué han consistido y quién las ha realizado.
- **Página aleatoria**: nos lleva a cualquiera de los artículos de la wiki de forma aleatoria.
- **Ayuda**: breves consejos para aquellos interesados en colaborar con la plataforma.

Figura 3.5: Panel de navegación en IberOgre

Búsqueda

El cuadro de búsqueda nos permite encontrar artículos que hablen sobre las palabras claves introducidas. En primer lugar trata de buscar un artículo cuyo nombre coincida exactamente con la clave introducida, posteriormente busca coincidencias parciales en el título y, por último, dentro del texto. Muestra los resultados por orden de relevancia de forma que el lector tenga más probabilidades de encontrar lo que busca en la zona alta de la página de resultados. El cuadro es sencillo y puede verse en la figura 3.6.

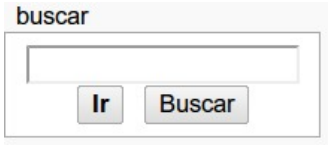


Figura 3.6: Cuadro de búsqueda en IberOgre

Bloque de contenido

El bloque más importante de **IberOgre** es el de contenido, en él se listan todos los artículos organizados por categorías y en orden creciente de dificultad. A pesar de que las dependencias no son exactamente lineales en todos los casos, es buena idea que el lector pueda empezar por el primero y continuar hacia abajo por la lista. En la figura 3.7 puede observarse con más detalle este bloque aunque no se muestra en su totalidad por razones de espacio.

Figura 3.7: Contenido de IberOgre

Bloque otros

El bloque *Otros* se divide en dos secciones diferenciadas. La primera está relacionada con la difusión del proyecto y la comunicación con los usuarios (*IberOgre en otros medios*) mientras que la segunda contiene enlaces a manuales de referencia con información para aquellos que deseen colaborar o profundizar.

- **IberOgre en otros medios**:
  - *Blog oficial*: blog dedicado al desarrollo de **IberOgre** y **Sion Tower**. Contiene tanto noticias sobre los avances como artículos documentando el desarrollo de subsistemas concretos.
  - *Forja*: enlace al repositorio de la forja de RedIRIS que aloja al proyecto. Contiene noticias, lista de tareas, lista de correo, ficheros descargables y el repositorio de código Subversion [36].
  - *Twitter*: enlace a la cuenta oficial del proyecto en la conocida red social de microblogging. Como puede leerse en el capítulo Comunidad y difusión, se ha hecho un uso extensivo de esta herramienta durante todo el desarrollo. De esta manera, se establece un contacto más directo con los lectores.
  - *E-mail*: medio para contactar con la wiki y enviar sugerencias, críticas y otras opiniones para cambiar elementos e introducir mejoras.
- **Ayuda**:
  - *Manual oficial de Ogre3D en español*: traducción completa del manual oficial de referencia de OGRE3D en español. El fichero *PDF* ha sido ofrecido por el colaborador Mario Velázquez Muñoz. Es un manual de consulta que complementa perfectamente al carácter didáctico de los artículos.
  - *Edición en MediaWiki*: guía sencilla sobre edición en sistemas *MediaWiki* creada por Noelia Sales Montes y Emilio José Rodríguez Posada. Aquellos lectores interesados en colaborar pueden hacerlo aunque no tengan experiencia con plataformas de este tipo utilizando este sencillo manual.

Artículos

Cada artículo comienza con una introducción breve como ya hemos mencionado anteriormente. A esta introducción le sigue un índice de contenidos con enlaces que llevan directamente a las subsecciones del texto. Esta funcionalidad facilita el acceso a aquellos lectores que deseen buscar un apartado concreto.

3.3.3. Ejemplos

Los ejemplos en **IberOgre** son una parte imprescindible del aprendizaje, proporcionan una aplicación práctica del contenido teórico del texto que los preceden. Estos ejemplos tratan de aglutinar lo que debería aprenderse tras leer el artículo en una sola y sencilla aplicación descargable. Recordemos que en este caso es más relevante la sencillez que la vistosidad y la calidad del código. Tras cada ejemplo, se anima al lector a estudiar el código y realizar modificaciones para demostrar su comprensión del mismo y dominio de la materia. A continuación hacemos una breve descripción sobre cada uno de los ejemplos de **IberOgre**.

Figura 3.8: Otros temas relacionados con IberOgre

Inicialización y cierre de Ogre

Tras el artículo *Inicialización y cierre de Ogre* el lector no sabe mostrar nada en pantalla, ni siquiera crear una ventana en la que renderizar elementos. Simplemente conoce la secuencia de inicialización del motor y la forma para salir de una aplicación de forma ordenada y correcta. El ejemplo se limita a crear el objeto principal de OGRE3D habiendo configurado previamente el sistema de logs y habiendo mostrado un diálogo de configuración al usuario. Tal y como se muestra en la figura 3.9, simplemente inicializamos el motor para cerrarlo seguidamente.

Creación básica de escenas

En este artículo aprendíamos a crear una ventana de renderizado, configurar el gestor de escenas, utilizar la gestión de recursos, cargar elementos dentro del grafo de la escena y controlar el bucle de renderizado empleando varias aproximaciones. El ejemplo aglutina todos estos conceptos creando una sencilla aplicación que inicializa OGRE3D, carga un personaje en pantalla y permanece a la espera de algún evento de salida. El resultado puede verse claramente en la figura 3.10. Se requieren conocimientos de gestión de eventos de entrada con OIS, por lo que es recomendable leer dicho artículo y seguir su ejemplo en primer lugar.

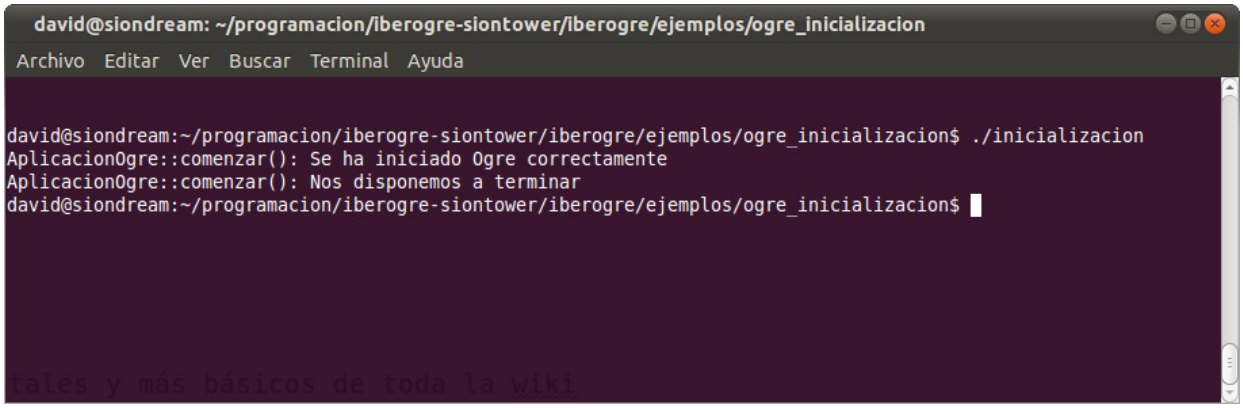


Figura 3.9: Ejemplo de inicialización y cierre de Ogre3D

Figura 3.10: Ejemplo de creación de escenas

Materiales

En el artículo sobre materiales en OGRE3D aprendemos a escribir scripts definiendo materiales, a cargarlos en nuestra aplicación y a aplicarlos a entidades como mallas tridimensionales. En el ejemplo para este artículo tenemos una sencilla escena con una paredes y una esfera en el centro iluminada por varios puntos de luz. Empleando el teclado numérico podemos cambiar el material de la esfera para ver los efectos producidos. De esta manera se resume la creación de materiales mediante scripts, su carga en memoria y aplicación sobre entidades, puede verse una captura del ejemplo en ejecución en la figura 3.11.

Manipulación de nodos

En el artículo sobre manipulación de nodos aprendemos a gestionar la información básica de los nodos como visibilidad, nombre, sus descendientes, etc. Partimos de la escena con el personaje vista en el ejemplo de la creación básica de escenas. Utilizando la gestión de eventos proporcionada por OIS capturamos las pulsaciones de las teclas W, A, S y D para desplazar y rotar al personaje por el mundo. Empleando la rueda del ratón podemos escalarlo. Puede verse una captura en la figura 3.12.

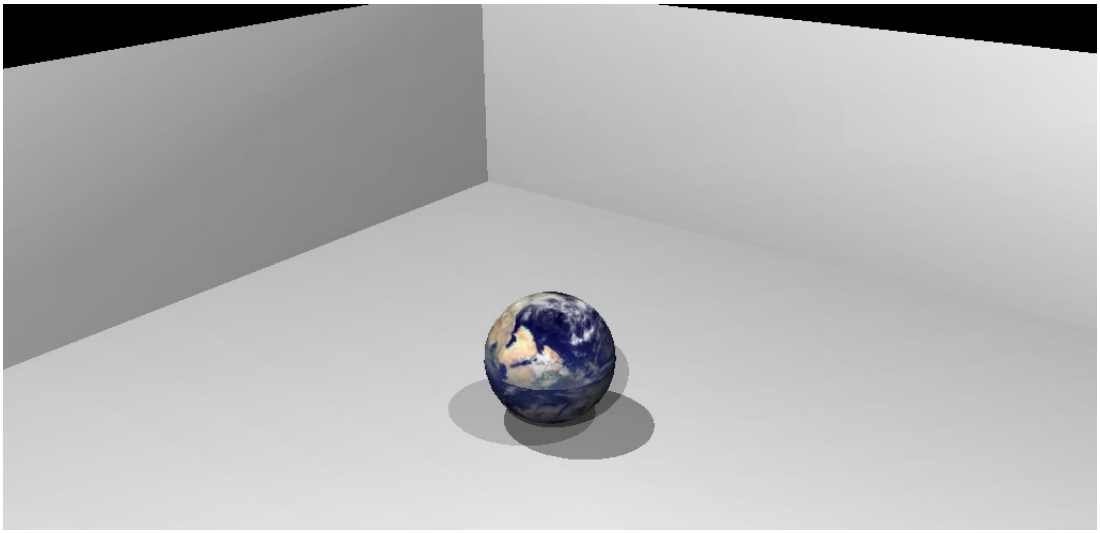


Figura 3.11: Ejemplo de materiales

Figura 3.12: Ejemplo de manipulación de nodos

Luces, sombras y entorno

En el artículo sobre iluminación aprendemos a gestionar varios efectos lumínicos y de ambientación como la niebla, las fuentes de luz, las técnicas de sombreado y los fondos. El ejemplo trata de aglutinar todos estos conceptos en una aplicación interactiva que nos permite activar y desactivar varios de los efectos anteriormente mencionados. Se crea una escena con un plano texturizado a modo de suelo y un personaje sobre él. Los controles son los siguientes y el resultado puede observarse en la figura 3.13.

- **s**: alterna entre distintas técnicas de sombreado. Es la mejor forma de ver la diferencia entre ambas.
- **d**: activa o desactiva el fondo.
- **n**: activa o desactiva la niebla.
- **1**: apaga o enciende la luz número 1 (roja).
- **2**: apaga o enciende la luz número 2 (azul).
- **3**: apaga o enciende la luz número 3 (verde).
- **4**: apaga o enciende la luz número 4 (amarilla).

Figura 3.13: Ejemplo de iluminación

Animación

En el artículo de animación se exponen las distintas técnicas para animar entidades en OGRE3D así como la forma de activar y mezclar varias animaciones al mismo tiempo. En el ejemplo cargamos a la mascota del motor llamada *Simbad* sobre un sencillo escenario y ofrecemos controles al usuario para animarlo. Utilizando las teclas de dirección *Simbad* se desplaza con una animación y pulsando *D* comienza o para de bailar. Puede verse el resultado en la figura 3.14.

Figura 3.14: Ejemplo de animaciones

Sistemas de partículas

En este artículo se explican las técnicas de definición de sistemas de partículas a través de scripts en texto plano así como su carga dentro de una escena. En el ejemplo simplemente cargamos varios sistemas de partículas entre los que el usuario puede alternar empleando las teclas numéricas del 1 al 9. El resultado puede verse en la figura 3.15.

Figura 3.15: Ejemplo de sistemas de partículas

Sistema de Overlays

La lección sobre el sistema de Overlays de OGRE3D incluye la sintaxis y propiedades de los scripts que los definen así como su carga dentro del juego. En el ejemplo se incluyen varios scripts para definir estas plantillas bidimensionales ofreciendo cierta interactividad. Partimos del resultado del ejemplo de materiales y añadimos un panel informativo con el nombre del material seleccionado y el número de cuadros por segundo (*FPS*) a la que se ejecuta la aplicación. Si cambiamos de material con el teclado numérico, veremos el cambio reflejado en el panel como se aprecia en la figura 3.16.

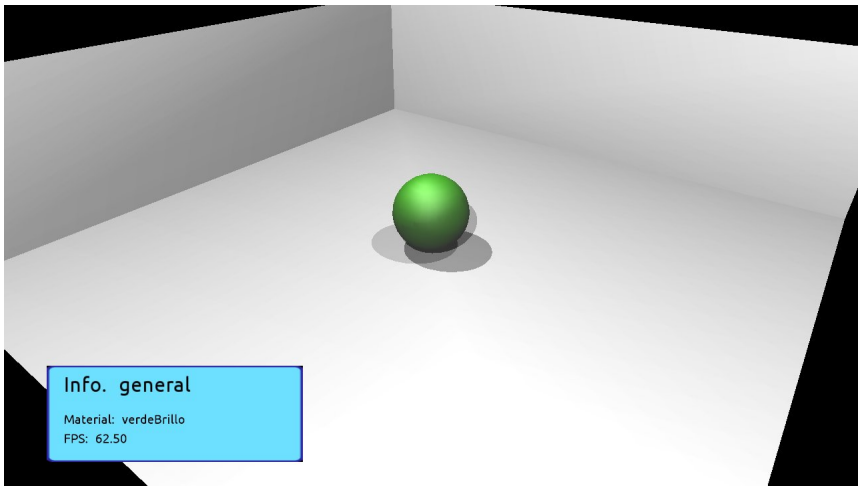


Figura 3.16: Ejemplo de Overlays

Manejo básico de OIS

OIS es la biblioteca que utilizamos en **IberOgre** para la gestión de dispositivos de entrada de usuario. En el artículo se detalla el proceso de inicialización, configuración y cierre de la biblioteca así como el manejo de dispositivos (joysticks, ratones y teclados) para consultar su estado en un momento dado. También se explica la aproximación empleando *Listeners* [15] para dar respuesta a eventos. En el ejemplo se crea una aplicación de OGRE3D sin ventana y permanecemos a la espera de eventos que mostramos por la terminal.

Extender la gestión de recursos, audio

En este artículo se explica el proceso de extensión del sistema de gestión de recursos de OGRE3D a través del desarrollo de un subsistema de audio empleando la biblioteca LIBSDL MIXER. En el ejemplo se carga una escena con un escenario, una silla y un micrófono a modo de teatro para monólogos. El usuario se pone en el papel de regidor indicando qué sonido o melodía debe reproducirse en un momento dado empleando el teclado numérico, se muestra un sencillo panel con la leyenda.

- **1**: sintonía de comienzo del programa.
- **2**: sintonía del contenido del programa.
- **3**: sintonía del final del programa.
- **4**: aplausos de un público ficticio.
- **5**: risas enlatadas.
- **6**: abucheos del público ficticio.

El resultado puede verse en la figura 3.17.

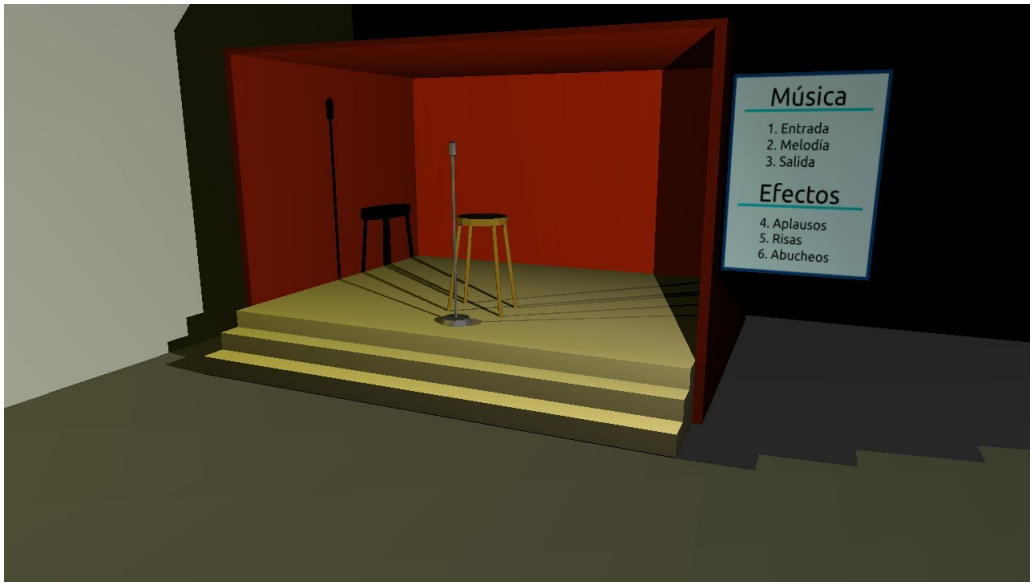


Figura 3.17: Ejemplo de extensión del sistema de recursos, audio

3.4. Implementación

Tras enumerar los usuarios, artículos, detallar la estructura de los mismos y la navegabilidad de la plataforma pasamos a la fase de implementación. Nos centraremos en los puntos más reseñables de esta fase. En primer lugar hablaremos de la elección del motor de wikis *MediaWiki* y de sus características principales. Posteriormente trataremos las plantillas que hemos empleado para formatear la documentación. Finalmente abordaremos la estructura general de los ejemplos. La escritura de los artículos en sí mismos es trivial y por ello no es comentada en esta sección.

3.4.1. El motor MediaWiki

MediaWiki es un popular motor para aplicaciones wikis desarrollado por la propia Fundación Wikimedia y utilizado en todos sus sitios como Wikipedia, Wiktionary y Wikinews entre otros muchos. Está escrito en el lenguaje de programación PHP y precisa de una base de datos de tipo SQL para almacenar la información. Su primera versión fue liberada en enero de 2002 y actualmente, la 1.17.0 es la última versión estable y fue lanzada en junio de 2011. [50].

Figura 3.18: Logo del motor MediaWiki

MediaWiki es, con diferencia, el software para creación y gestión de wikis más utilizado en todo el mundo. Es fácilmente extensible mediante complementos y existen bots que automatizan tareas como la prevención y corrección de vandalismos. Uno de estos bots es AVBot desarrollado por Emilio José Rodríguez Posada [12].

Escribir en la sintaxis utilizada por el motor es bastante sencillo, ya que utiliza un lenguaje mucho más simple que HTML. Esto es muy importante ya que buscamos colaboradores para la plataforma educativa. Cuanto más accesible sea, más probabilidades de éxito. Por ello se ha adjuntado la guía redactada por Noelia Sales Montes y Emilio Rodríguez Posada *Edición de wikis WikiMedia* [29].

El sistema proporciona un método de edición sencillo y una navegación cómoda con contenido multimedia. Podemos crear espacios de nombres y secciones bien diferenciadas. Es altamente configurable y podemos ajustarlo para que encaje perfectamente con los objetivos de **IberOgre**. Del proceso de instalación y configuración se ha encargado la Oficina de Software Libre de la Universidad de Cádiz ya que dicha instalación se ha realizado en sus servidores.

Su extensibilidad, posibilidades de configuración, la cantidad de documentación disponible y la enorme y activa comunidad han hecho que nos decantemos por el uso de *MediaWiki* como software para dar soporte a **IberOgre**.

3.4.2. Plantillas

Las plantillas son fragmentos de código que aceptan parámetros a los que podemos llamar desde otras páginas utilizando una sintaxis especial. Esto nos ayuda a generalizar ciertas estructuras y ahorrar en recursos por parte del servidor. En esta sección haremos un recorrido por todas las plantillas utilizadas,

sus parámetros y el resultado que producen al utilizarlas.

- **Artículo**

  Esta plantilla se utiliza por cada artículo en el bloque de contenido. Incluye el nombre con el enlace al texto, una descripción y un icono representativo. Su código es el siguiente, pueden verse los parámetros identificados por su orden de aparición en la llamada a la plantilla y encerrados en llaves triples.

```
{| style= "border: 0;
           margin: 0;
           background-color: inherit;"
           cellpadding="3"

 | valign="top" | [[Imagen:{{{1}}}|64px|link={{{2}}}]]
 | valign="top" |
 | '''[[{{{2}}}]]'''<br>{{{3}}}
|}
```

  Una llamada como esta:

```
{{ Articulo
|fuego.png
|Sistemas de particulas
|Aprende a crear efectos especiales con Ogre como llamas, explosiones
 o nubes de humo para mejorar tus videojuegos.
}}
```

  Produce el resultado de la figura 3.19:

Sistemas de partículas
Aprende a crear efectos especiales con Ogre como llamas, explosiones o nubes de humo para mejorar tus videojuegos.

Figura 3.19: Plantilla Artículo

- **RecursoExterno**

  Para referenciar enlaces externos de una forma similar a los artículos se emplea esta plantilla. Cuenta con un título que lleva al destino y un icono relacionado que hace lo propio. Además, se añade una descripción. El código que consigue el efecto es el siguiente.

```
{| style= "border: 0;
           margin: 0;
           background-color: inherit;"
           cellpadding="3"

| valign="top" | [[Imagen:{{{1}}}|64px|link={{{2}}}]]
| valign="top" |
| '''[{{{2}}} {{{3}}}]'''<br>{{{4}}}
```

```
9  |}
```

Una llamada como la siguiente:

```
1  {{RecursoExterno
2  |cuaderno.png
3  |http://siondream.com/blog/category/proyectos/pfc/
4  |Blog oficial
5  |Noticias, experiencias y avances en IberOgre
6  }}
```

Produce el resultado que puede apreciarse en la figura 3.20.

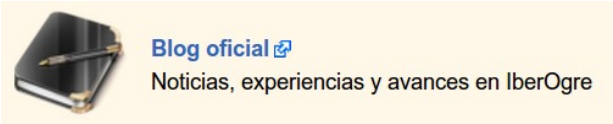


Figura 3.20: Plantilla RecursoExterno

- **BloqueSeccion**

  Esta plantilla se emplea para conseguir el bloque de que preceden a las secciones de la wiki dentro del bloque contenido. Su código es el siguiente.

```
1  <div style="padding-top:2px;
2              padding-bottom:2px;
3              padding-left:5px;
4              padding-right:2px;
5              border:1px solid #092C00;
6              margin: 4px;
7              background-color: #F0FFF0;">
8  '''{{{1}}}'''
9  </div>
```

- **Calendario**

  En diversas páginas interesaba colocar un calendario con la fecha de la edición. Por ello, se ha creado esta plantilla que coloca la fecha actual en un formato de calendario clásico. Su código wiki y HTML es el siguiente.

```
1  <div style= "border:solid #ccc;
2               background: #fff;
3               border-width: 1px 3px 3px 1px;
4               text-align: center;
5               padding-top:3px;
6               float:left;
7               font-size: smaller;
8               line-height: 1.3;
9               margin-right: 4px;
```

```
                width: 7em">

    [[{{CURRENTDAYNAME}}]]
    [[{{CURRENTDAY}} de {{CURRENTMONTHNAME}}|
    <span style= "font-size: x-large;
                  width: 100%;
                  display: block;
                  padding:6px 0px">
        {{CURRENTDAY}}
    </span>]]

    <span style="display: block;"> [[{{CURRENTMONTHNAME}}]]</span>

    <span style= "background: #aaa;
                  color: #000;
                  display: block;">
    '''[[{{CURRENTYEAR}}]]'''
    </span>
</div>
```

El resultado producido es el que podemos ver en la figura 3.21.

Figura 3.21: Plantilla Calendario

- **FicheroDescargable**

  Los ejemplos finales de cada artículo de **IberOgre** pueden descargarse en forma de paquete para ser compilados, ejecutados o modificados por los lectores. Para ello, empleamos una plantilla que añade un enlace al fichero descargable, el título del artículo una descripción del ejemplo. Su código es el siguiente:

```
{| style= "border: 0;
           margin: 0;
           background-color: inherit;"
           cellpadding="3"
| valign="top" | [[Imagen:{{{1}}}|64px|]]
| valign="top" |
| '''[[media:{{{2}}}|{{{3}}}]]'''<br>{{{4}}}
|}
```

Empleado la siguiente llamada:

```
{{FicheroDescargable
|ejemplo.png
|ogre_escenas.zip
|Ejemplo de creacion basica de escenas
```

```
5   |Pequena aplicacion que inicia Ogre y carga un personaje en la escena
6   }}
```

Obtenemos el siguiente bloque (ver figura 3.22).

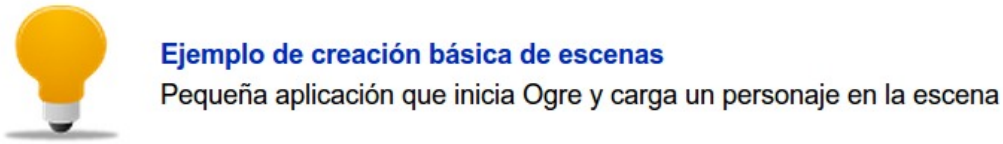


Figura 3.22: Plantilla FicheroDescargable

3.4.3. Estructura general de los ejemplos

Todos los ejemplos de **IberOgre** se han implementado de manera similar aunque cada uno tiene sus particularidades en los aspectos concretos. Esta decisión viene motivada por la intención de conseguir uniformidad tanto en el estilo como en las soluciones. De esta forma, los lectores comprenderán los ejemplos cada vez con mayor facilidad y se centrarán únicamente en las partes cambiantes, lo realmente interesante y lo que se imparte en cada lección.

El esquema general puede verse en el diagrama de clases de diseño en la figura 3.23. La clase *AplicacionOgre* es la encargada de iniciar el motor OGRE3D y la biblioteca OIS a través de los métodos privados *iniciarOgre()* e *iniciarOIS()*. Hereda de las clases observadoras [15] *WindowEventListener*, *KeyListener* y *MouseListener*. La primera se encarga de capturar eventos de pantalla (movimiento, cierre y redimensión), la segunda toma eventos de teclado (pulsar y soltar tecla) mientras que la tercera hace lo propio con el ratón (movimiento, pulsar botón y liberar botón). Como puede verse, existe un gestor de evento para cada uno de ellos.

Con una llamada a *buclePrincipal()* comienza el bucle de juego hasta que se presione la tecla escape o se cierre la ventana. Esta clase está preparada para proporcionar lo básico de una aplicación interactiva que use OGRE3D. Está pensada para que heredemos de ella en cada ejemplo que creemos. Por este motivo, el bucle de juego y los gestores de eventos son virtuales, podemos sobrecargar su comportamiento.

Efectivamente, en cada ejemplo creamos una clase hija que suele llevar por nombre *EscenaSimple*. Esta clase se encarga de crear, gestionar la escena que se desarrolle en el ejemplo y controlar su lógica (interactividad). Siempre cuenta con varios métodos de configuración: *prepararRecursos()*, *configurarSceneManager()*, *crearCamara()* y *crearEscena()*. En cada ejemplo concreto se pueden añadir más métodos y atributos según sea necesario.

Esta aproximación centrada en la reutilización de código y la sencillez ha facilitado la implementación de ejemplos de forma rápida y directa. Además, como ya hemos mencionado, facilita el aprendizaje a los lectores. Para saber más sobre los ejemplos concretos, puede accederse a cualquier artículo de **IberOgre** y descargar el paquete correspondiente.

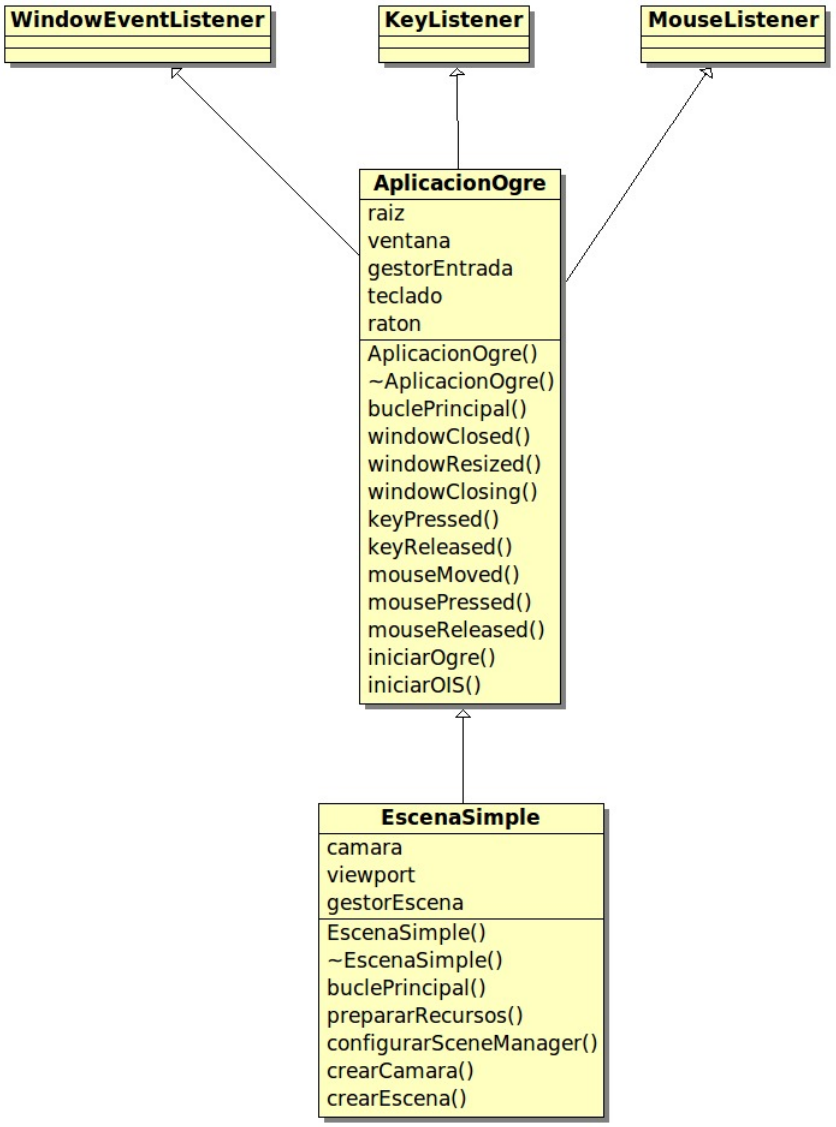


Figura 3.23: Diagrama de clases para los ejemplos

3.5. Pruebas

Un plan de pruebas elaborado nos garantiza un mínimo de calidad para el proyecto. En este caso no podemos aplicar el concepto de prueba de caja blanca ni prueba de caja negra ya que hablamos de un proyecto de documentación (exceptuando los ejemplos finales de cada artículo). En esta sección enumeraremos las pruebas que se han realizado sobre la plataforma de aprendizaje en su totalidad.

Es necesario mencionar que no se han realizado pruebas en un servidor de preproducción ya que la instalación y configuración corría a cargo de la Oficina de Software Libre de la Universidad de Cádiz. No obstante, el motor *MediaWiki* ha sido probado de forma extensiva y es muy estable a día de hoy.

La plataforma era pública (aunque no abierta a ediciones) para todo el mundo y con los primeros artículos llegaron los primeros lectores. **IberOgre** ha sido sometido a pruebas por parte de muchos usuarios.

Éstos han podido ofrecer su opinión a través del blog de desarrollo, de Twitter o del correo electrónico.

A continuación adjuntamos la lista de pruebas realizadas junto a los resultados obtenidos.

1. **Estructura de los artículos**

   ¿Se cumple con la estructura acordada? ¿Cumple la estructura con los objetivos?

   IberOgre se ha dividido en las cuatro secciones establecidas durante la fase de análisis (capítulo 3.2 y página 21). Los lectores han manifestado que la división en introducción, matemáticas OGRE3D y otras tecnologías les facilitó la organización de su aprendizaje. Por tanto, se han cumplido con los objetivos y el resultado de la prueba es positivo.

2. **Errores en el texto**

   ¿Existen errores ortográficos, de sintaxis o expresión en los textos?

   Cada artículo ha sido revisado tanto por su redactor inicial como por varios usuarios que se han prestado a colaborar con las correcciones. El uso del corrector ortográfico se ha combinado con lecturas humanas para detectar y eliminar los fallos de este tipo.

3. **Plantillas**

   ¿Las plantillas se visualizan correctamente? ¿Existen plantillas que devuelvan algún error?

   Se ha comprobado que todas las llamadas a plantillas funcionan como se esperaba de ellas. Todas las llamadas se visualizan de forma correcta.

4. **Etiquetado**

   ¿Están todos los artículos debidamente etiquetados?

   En *MediaWiki* los artículos pueden etiquetarse de forma que encontrarlos sea más sencillo. En definitiva facilita la navegación ya que se crea una página de etiquetas listando a los artículos que contienen. En **IberOgre** todos los artículos han sido etiquetados por lo que el resultado del test es positivo.

5. **Usuario no registrado**

   ¿Son los permisos de estos usuarios los correctos? ¿Pueden editar artículos o crear otros nuevos?

   Como se dice en el análisis de la wiki, los usuarios no registrados pueden acceder a todo el contenido de **IberOgre** aunque no pueden hacer ningún tipo de modificación. Se ha comprobado cerrando la sesión de nuestro usuario que, sin estar registrados en el sistema, no podemos crear ni modificar página alguna. Por tanto, el resultado es positivo.

6. **Usuario registrado**

¿Pueden los usuarios registrados crear o modificar contenido? ¿Tienen acceso de administración?

Tal y como se menciona en la prueba anterior, ya habíamos definido los permisos que debía tener cada usuario en el análisis. El usuario registrado debía poder acceder tanto al contenido en modo lectura como en modo escritura. Asimismo debía contar con la capacidad para crear nuevos artículos. No obstante, no se le permite el acceso a la configuración del sistema.

Se han probado que puede trabajar con el contenido pero no así con la configuración de **IberOgre**. Sólo el usuario administrador cuenta con acceso al servidor de producción y a la base de datos.

7. **Usuario administrador**

   ¿Cuenta el usuario administrador con todos los permisos?

   El usuario administrador cuenta con los privilegios del usuario registrado y, además, debe poder acceder a parámetros de configuración así como poder instalar nuevas extensiones o gestionar las actuales. Durante el desarrollo, fueron necesarias varias tareas administrativas y este usuario pudo resolverlas sin problemas confirmando que el esquema de permisos funciona como se había planeado.

8. **Ayuda**

   ¿Es la ayuda lo suficientemente clara para que los lectores puedan empezar a colaborar? ¿Es ampliable la plataforma? ¿Los usuarios pueden enviar opiniones?

   La Ayuda en **IberOgre** es de extrema importancia ya que debe informar a los usuarios de las posibilidades que tienen para colaborar y animarles a hacerlo. La guía de edición en *WikiMedia* de Noelia Sales Montes y Emilio José Rodríguez Posada cumplió su cometido ya que, lectores que nunca habían editado una wiki con este motor pudieron hacerlo.

   La plataforma se ha mostrado ampliable en gran medida. Tras la conclusión de este proyecto, la wiki cuenta con temas en los que se puede profundizar aún más. Algunos usuarios ya se han mostrado interesados en colaborar.

   En la página inicial de **IberOgre** se adjuntan enlaces tanto al blog oficial de desarrollo, como a la cuenta oficial de Twitter como al correo electrónico. Los usuarios pueden ponerse en contacto (y de hecho lo han hecho en numerosas ocasiones) a través de cualquiera de estos medios.

9. **Ejemplos**

   ¿Funcionan correctamente los ejemplos? ¿Abarcan el contenido de los artículos? ¿Están debidamente documentados?

   Esta prueba se ha aplicado a todos los ejemplos de **IberOgre** y en la totalidad de los casos el resultado ha sido satisfactorio. Lo especificado en la fase de diseño de los ejemplos (sección 3.3, página 26) se cumple con exactitud.

Su diseño tiene como objetivo ilustrar todo el contenido del texto que precede a cada ejemplo y siempre se logra cumplir todo el temario. Cada ejemplo viene acompañado de una explicación que informa de su cometido, su diseño y funcionamiento. Esto, junto a las críticas positivas por parte de los usuarios, nos hace considerar que están debidamente documentados.

10. **Multiplataforma**

¿Es todo el código expuesto en la wiki multiplataforma? ¿Funcionan todos los ejemplos tanto en Windows como en GNU/Linux?

Uno de los objetivos principales de **IberOgre** era ofrecer directrices sobre desarrollo de software multiplataforma, al menos para GNU/Linux y Windows. Todo el código de la plataforma de aprendizaje ha sido probado en ambos sistemas operativos con éxito.

Cada ejemplo ha sido compilado y ejecutado en Ubuntu (GNU/Linux) y en Windows 7 utilizando el método expuesto en el artículo de creación de un entorno de desarrollo multiplataforma [7] obteniendo resultados positivos.

Capítulo 4

Desarrollo de Sion Tower

4.1. Metodología

Sion Tower será un sistema software complejo y para su desarrollo se ha decidido emplear la metodología *RUP* (Rational Unified Process) [26]. Así mismo, se ha escogido *UML* como la notación para modelar nuestro sistema.

Mediante la metodología *RUP* podemos realizar un desarrollo orientado a objetos de forma iterativa. Las ventajas que nos proporciona el paradigma de la orientación a objetos consisten en una mayor reusabilidad (uno de los objetivos principales del juego), escalabilidad, legibilidad y sencillez de mantenimiento.

4.2. Análisis

4.2.1. Especificación de requisitos del sistema

En esta sección haremos un recorrido por los requisitos que debe cumplir **Sion Tower** tanto en el plano de interfaces, como de rendimiento o de funcionalidades entre otros.

Requisitos de interfaces externas

En este apartado describiremos los requisitos relacionados con la conexión entre el hardware y el software además de la interfaz con el usuario. En lo referente a la conexión entre hardware y software utilizaremos varias tecnologías en forma de bibliotecas libres. Como motor de renderizado nos decantamos por OGRE3D, para capturar la entrada del usuario, emplearemos OIS (*Object Oriented Input System*) [58], para reproducir sonidos emplearemos LIBSDL y su extensión LIBSDL MIXER [34]. Finalmente, para gestionar los elementos de la interfaz a un nivel de abstracción superior se hará uso de MYGUI [28].

Sion Tower deberá soportar varias resoluciones de pantalla siempre que se mantenga su resolución de aspecto 16:9. Además, debe ser posible establecer un modo a pantalla completa si el usuario lo desea. Varias resoluciones aceptadas por el videojuego deben ser:

- 640 x 360
- 960 x 544
- 1280 x 720
- 1920 x 1080

La navegación por los menús debe ser intuitiva y se realizará utilizando el ratón. En la figura 4.1 pueden observarse todas las pantallas de **Sion Tower** dispuestas en un diagrama de flujo para mostrar su navegación.

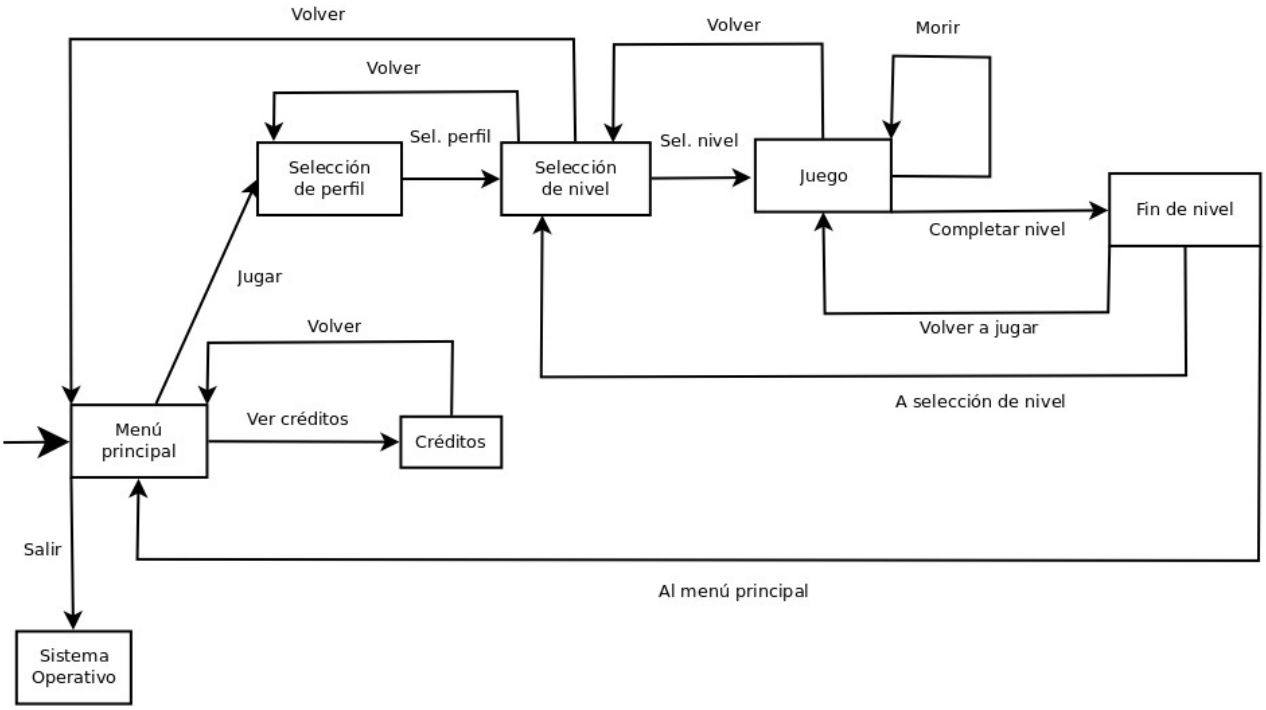


Figura 4.1: Diagrama de flujo de las pantallas de Sion Tower

El **Menú principal** (figura 4.2) debe presentar al jugador una escena de la Torre Sagrada con los enemigos aproximándose. Mediante la opción *Jugar* podrá dirigirse a la pantalla de selección de perfil, utilizando *Créditos* le será posible llegar a la pantalla de mismo nombre y pulsando sobre *Salir* se le llevará de vuelta al sistema operativo.

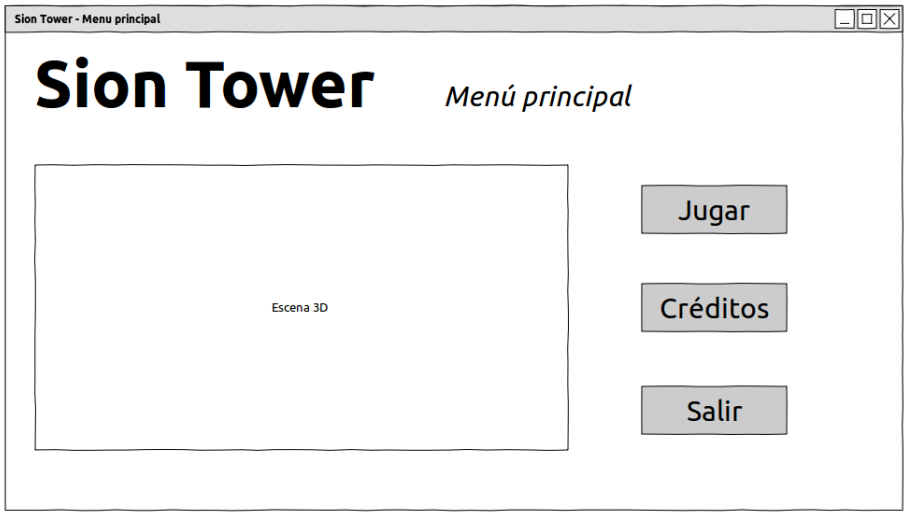


Figura 4.2: Boceto del menú principal

En la pantalla de **Selección de perfil** (figura 4.3) se mostrará una escena tridimensional del interior de la Torre. En un panel se listarán todos los usuarios registrados en el sistema. El jugador podrá seleccionar uno de ellos y pulsar *Aceptar* para dirigirse a la pantalla de selección de nivel. Si pulsa sobre *Eliminar* el perfil será destruido de los registros. Para crear un nuevo perfil debe introducir el nombre deseado en el panel inferior y presionar *Crear*. Desde este menú debe ser posible regresar al menú principal.

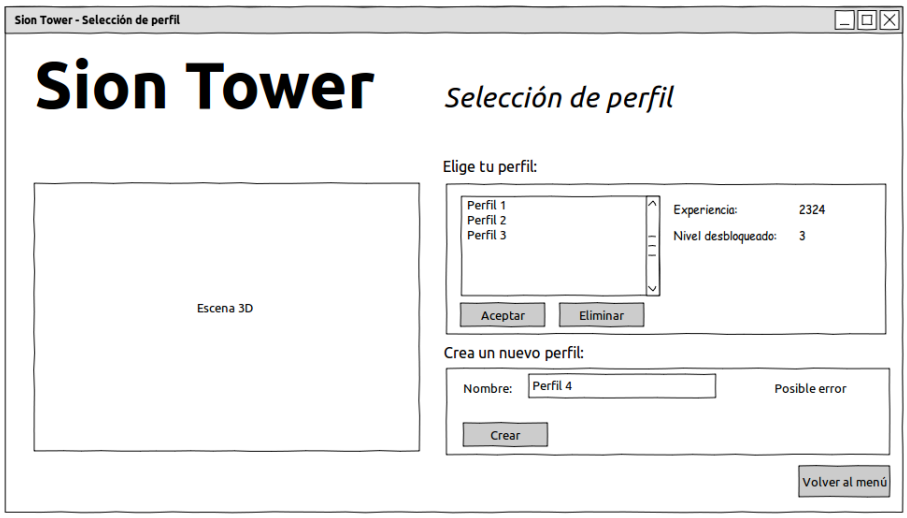


Figura 4.3: Boceto de la pantalla de selección de perfil

Dentro de la pantalla de **Selección de nivel** (figura 4.4) el usuario debe ver una escena del exterior de la Torre. Se mostrará un panel con la lista de niveles de todo el juego. Cada nivel vendrá acompañado de una miniatura, su nombre y descripción. El usuario irá desbloqueando niveles a medida que avanza por el juego. Los niveles bloqueados aparecerán en tonos grisáceos y se indicará que son inaccesibles. Pulsando sobre la miniatura de un nivel disponible, el usuario accederá a la pantalla de juego. Puede volver atrás pulsando el botón *Atrás*.

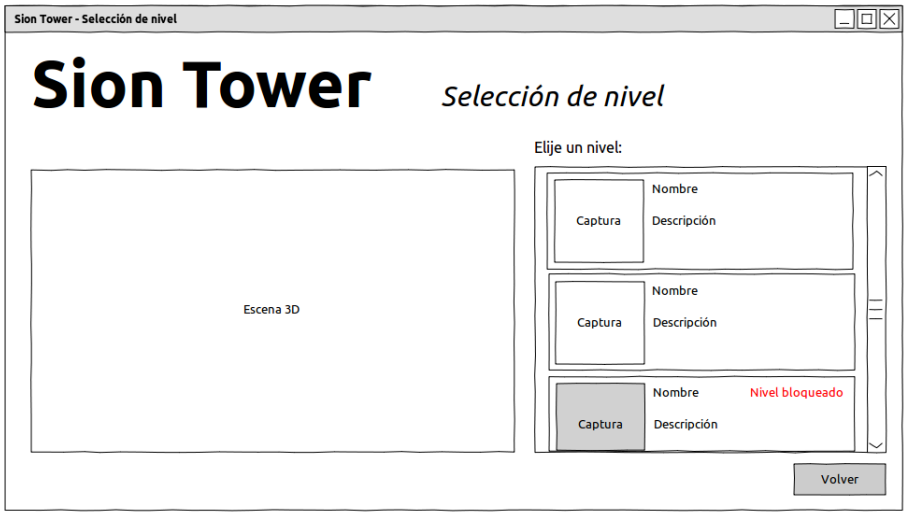


Figura 4.4: Boceto de la pantalla de selección de nivel

En la pantalla de **Juego** (figura 4.5) destacará la escena 3D con el escenario, el personaje principal y los enemigos. La barra inferior mostrará información relevante como la energía vital y maná restantes, la experiencia obtenida y la lista de hechizos disponibles. El hechizo seleccionado aparecerá resaltado y el usuario podrá seleccionar otro distinto pulsando sobre su icono. Al pasar el ratón por encima de un hechizo se mostrarán sus atributos: nombre, descripción y maná necesario. El botón *Pausa* abrirá el menú de pausa que nos permitirá volver al juego, regresar a la pantalla de selección de nivel o desplazarnos directamente al menú principal.

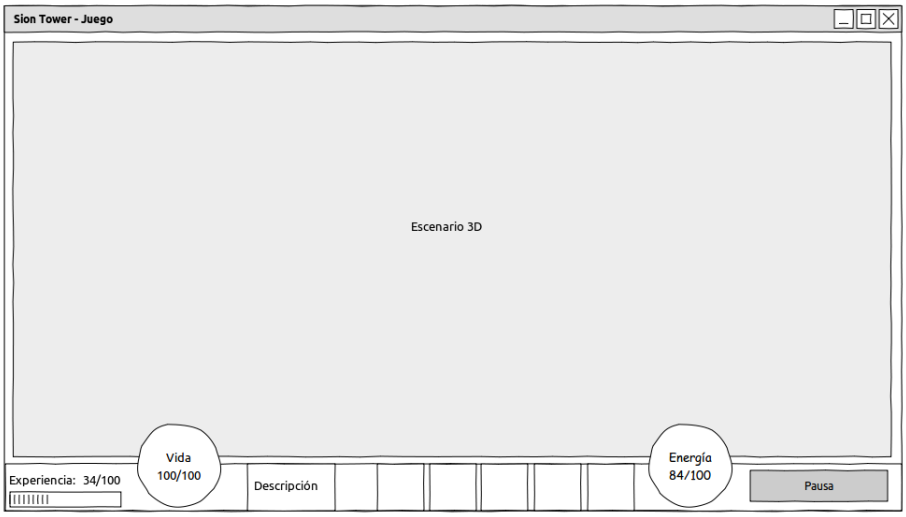


Figura 4.5: Boceto de la pantalla de juego

Cuando terminemos el nivel de forma exitosa el usuario verá la pantalla de **Fin de nivel** (figura 4.6). Se mostrará al personaje celebrando su victoria y un panel resumiendo la actuación del jugador durante la partida. La información será: enemigos eliminados, vida restante, maná utilizado, tiempo empleado y

los puntos obtenidos en cada categoría. Para volver a jugar el usuario podrá pulsar sobre *Volver a jugar*, para seleccionar un nuevo nivel lo hará sobre *Seleccionar nivel* y si desea volver al menú principal, presionará *Volver al menú*.

Figura 4.6: Boceto de la pantalla de fin de nivel (victoria)

La **Pantalla de créditos** (figura 4.7) consiste en una pequeña escena 3D dentro de la Torre y un panel mostrando a los creadores de **Sion Tower**. Utilizando el botón *Volver al menú* el usuario volverá al menú principal.

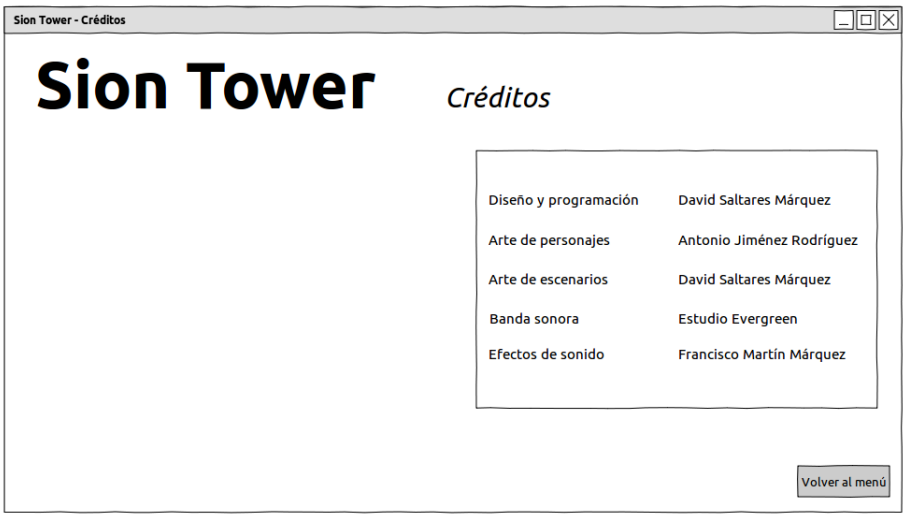


Figura 4.7: Boceto de la pantalla de créditos

Requisitos funcionales

Sion Tower cuenta con la siguiente lista de requisitos funcionales:

- Salir de la aplicación cerrando la ventana en cualquier momento.
- Gestionar varios perfiles de jugadores, cada uno con su último nivel desbloqueado y experiencia acumulada. Se permite la selección, creación y eliminación de perfiles.
- Seleccionar un nivel disponible para combatir contra la inteligencia artificial.
- Mover al personaje por el escenario.
- Lanzamiento de hechizos siempre y cuando se disponga del maná necesario.
- Selección del hechizo a lanzar.
- Capacidad de mover la cámara alrededor del personaje.
- Posibilidad de pausar y reanudar el juego.

Requisitos de rendimiento

Para disfrutar de **Sion Tower** de forma satisfactoria será necesaria, al menos, una resolución de 640 x 360 y una tasa de fotogramas por segundo igual o superior a 25. El juego deberá estar lo suficientemente optimizado en términos de uso de la CPU y consumo de memoria para alcanzar dicho rendimiento en un equipo de características similares al siguiente:

- **Sistema operativo**: GNU/Linux, Windows XP Service Pack 2, Windows Vista o Windows 7.
- **Procesador**: Pentium 2GHz o AMD.
- **Memoria**: 100 MB de memoria RAM disponibles.
- **Tarjeta de vídeo**: 128 MB de memoria y aceleración 3D.
- **Espacio en disco**: 100 MB.
- **Control**: ratón y teclado.

Además, la tasa de cuadros por segundo debe mantenerse estable. Es preferible contar con una tasa media estable que con una alta que oscile en gran medida.

Restricciones de diseño

En un videojuego es mucho más relevante la fluidez que su consumo de memoria. Actualmente, los equipos suelen contar con grandes cantidades de memoria principal con velocidades aceptables pero no todos incorporan tarjetas gráficas demasiado potentes. A la hora de diseñar **Sion Tower** nos centraremos en optimizar el uso de la CPU y del procesador de la tarjeta gráfica, recursos que tienen mucho más impacto en la percepción del usuario acerca del juego.

Requisitos del sistema software

El sistema software deberá adherirse a los siguientes requisitos:

- Todo el código debe ser multiplataforma ofreciendo exactamente el mismo comportamiento en todos los sistemas operativos para los que se compile.
    - **Windows**: el juego se probará en Windows XP, Windows Vista y Windows 7.

- **GNU/Linux**: el juego será probado en Ubuntu 10.10.

- Se hará uso del ratón y el teclado. No obstante, cuando sea posible, se proporcionarán teclas de acceso rápido para un manejo más eficiente.
- El control deberá ser intuitivo, el jugador debería poder utilizar **Sion Tower** sin necesidad de haber acudido al manual de usuario. La interfaz también debe presentar un manejo lógico y sencillo.
- El juego debe ser fácilmente ampliable con nuevos niveles y elementos.

4.2.2. Modelo de casos de uso

Para el modelo de casos de uso se ha seguido, como se ha mencionado anteriormente la notación *UML*. Los pasos para obtener el modelo de casos de uso han sido los siguientes:

1. Identificar a todos los posibles usuarios del sistema y sus roles.
2. Para cada rol, determinar todas las maneras posibles que éste cuenta para interactuar con el sistema.
3. Creación de un caso de uso por cada objetivo que deba cumplir el sistema.
4. Estructurar todos los casos de uso, por ejemplo, empleando relaciones de inclusión o extensión.

Diagrama de casos de uso

En la figura 4.8 se muestra el diagrama de casos de uso para **Sion Tower**.

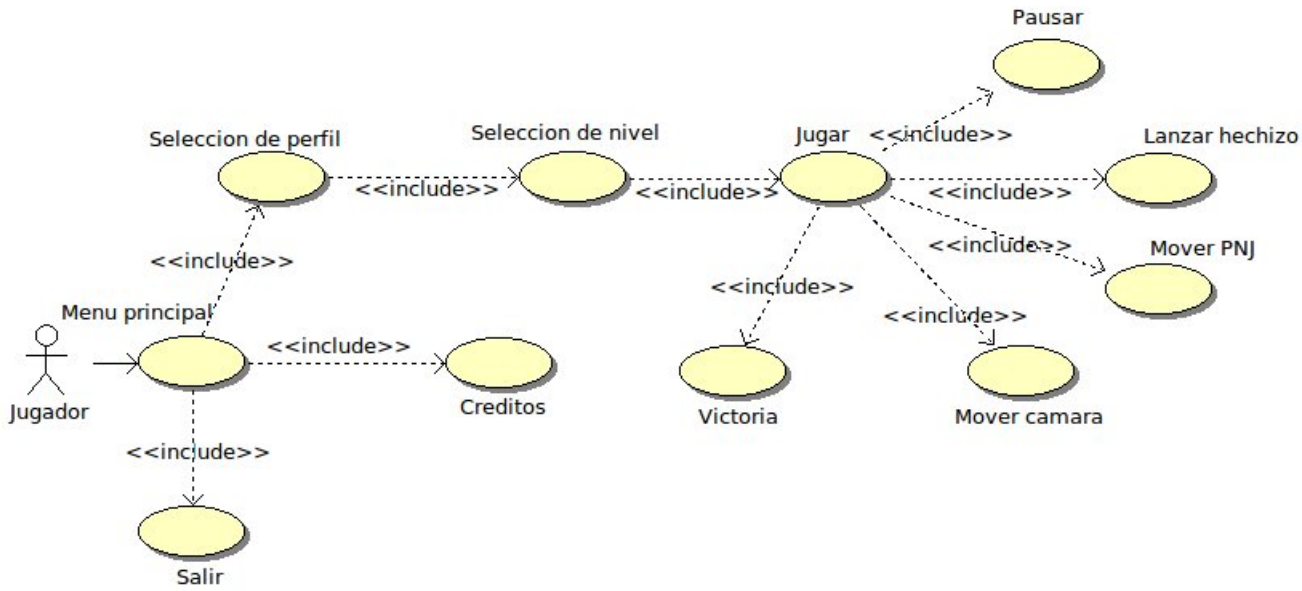


Figura 4.8: Diagrama de casos de uso

Descripción de los casos de uso

En esta sección adjuntaremos las descripciones de todos los casos de uso anteriormente expuestos. Para ello emplearemos una notación en texto utilizando un formato completo con plantilla. Se pretende que

su lectura sea sencilla y resulte accesible a la vez que directo.

Caso de uso: Menú principal

Caso de uso Menú principal

Descripción Se le muestra al *Jugador* el menú principal desde el cual es posible acceder a la selección de perfil o a la pantalla de créditos.

Actores *Jugador*.

Precondiciones Ninguna.

Postcondiciones Ninguna.

Escenario principal

1. El *Jugador* inicia la aplicación.
2. El *Sistema* inicia el motor del juego y muestra el menú principal en la pantalla.
3. El *Jugador* selecciona la opción Jugar.
4. El *Sistema* accede a la pantalla de Selección de perfil.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

3a El *Jugador* selecciona la opción Créditos.

1. El *Sistema* accede a la pantalla de Créditos.

3b El *Jugador* selecciona la opción Salir.

1. El *Sistema* libera los recursos y sale de la aplicación.

Caso de uso: Selección de perfil

Caso de uso Selección de perfil

Descripción Se le muestra al *Jugador* la pantalla de selección de perfil. Es posible seleccionar, crear o eliminar perfiles. Una vez seleccionado un perfil, se le conduce hasta la pantalla de selección de nivel. También es posible volver al menú principal.

Actores *Jugador*.

Precondiciones Ninguna.

Postcondiciones Se selecciona un perfil.

Escenario principal

1. El *Jugador* desea acceder a la pantalla de selección de perfil.
2. El *Sistema* muestra la pantalla de selección de perfil y carga los perfiles existentes en la lista.
3. El *Jugador* selecciona un perfil de la lista y pulsa sobre la opción Aceptar.
4. El *Sistema* accede a la pantalla de selección de nivel.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

3a El *Jugador* selecciona un perfil de la lista y pulsa sobre la opción Eliminar.

1. El *Sistema* elimina el perfil seleccionado.

3b El *Jugador* decide crear un nuevo perfil. Introduce el nombre del nuevo usuario y pulsa sobre Crear.

1. El *Sistema* comprueba que el nombre introducido no existe y crea el nuevo perfil.

3c El *Jugador* decide crear un nuevo perfil. Introduce el nombre del nuevo usuario y pulsa sobre Crear.

1. El *Sistema* comprueba que el nombre introducido existe y muestra el correspondiente error.

3d El *Jugador* selecciona la opción Volver.

1. El *Sistema* vuelve al menú principal.

Caso de uso: Selección de nivel

Caso de uso Selección de nivel

Descripción El *Jugador* ve la pantalla de selección de nivel. Puede escoger uno de entre los disponibles y acceder a la pantalla de juego.

Actores *Jugador*.

Precondiciones Se ha seleccionado un perfil.

Postcondiciones Se selecciona un nivel.

Escenario principal

1. El *Sistema* carga los niveles (icono, nombre y descripción) en la lista de niveles distinguiendo los desbloqueados de los accesibles y muestra la pantalla.
2. El *Jugador* selecciona un nivel desbloqueado.
3. El *Sistema* guarda el nivel como seleccionado y accede a la pantalla de juego.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

2a El *Jugador* selecciona la opción Volver.

1. El *Sistema* regresa a la pantalla de selección de nivel.

Caso de uso: Jugar

Caso de uso Jugar

Descripción El *Jugador* comienza un nivel, interactúa con el y puede que gane o pierda.

Actores *Jugador*.

Precondiciones Se ha seleccionado un nivel.

Postcondiciones Se completa un nivel (con éxito o no).

Escenario principal

1. El *Sistema* carga el nivel, los enemigos, la música y el personaje principal.
2. El *Sistema* inicializa las estadísticas: enemigos eliminados y puntos por haber eliminado dichos enemigos, maná utilizado, vida restante y tiempo transcurrido.
3. El *Jugador* y el *Sistema* interactúan durante la partida.
4. El *Sistema* detecta que el *Jugador* ha eliminado todos los enemigos.
5. El *Sistema* informa al *Jugador* de su victoria y se dirige hacia la pantalla de victoria.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

3a El *Sistema* comprueba que un enemigo ha atacado al personaje y le ha impactado.

1. El *Sistema* le resta una cantidad de vida al personaje correspondiente con el poder del enemigo.

5a El *Sistema* comprueba que la vida del protagonista llega a 0 y pierde la partida.

1. El *Sistema* le muestra un mensaje de derrota: *¡Has fallado! ¡Pulsa espacio para intentarlo de nuevo!*.
2. El *Jugador* pulsa espacio.
3. El *Sistema* carga la pantalla de selección de nivel.

Caso de uso: Pausar

Caso de uso Pausar

Descripción El *Jugador* selecciona pausar el nivel y puede reanudarlo o volver a otro menú.

Actores *Jugador*.

Precondiciones Se está jugando un nivel.

Postcondiciones Ninguna.

Escenario principal

1. El *Jugador* pulsa el botón Pausa o presiona la tecla escape.
2. El *Sistema* detiene todos los elementos del juego y muestra un menú de pausa.
3. El *Jugador* selecciona el botón Reanudar.
4. El *Sistema* vuelve a reanudar la partida.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

3a El *Jugador* selecciona el botón Selección de nivel.

1. El *Sistema* termina la partida y vuelve a la pantalla de selección de nivel.

3b El *Jugador* selecciona el botón Menú principal.

1. El *Sistema* termina la partida y regresa a la pantalla del menú principal.

Caso de uso: Lanzar hechizo

Caso de uso Lanzar hechizo

Descripción El *Jugador* selecciona un hechizo de entre los disponibles y lo lanza.

Actores *Jugador*.

Precondiciones Se está jugando un nivel.

Postcondiciones Ninguna.

Escenario principal

1. El *Jugador* selecciona un hechizo de la barra inferior o a través de las teclas de acceso rápido 1, 2 o 3.
2. El *Jugador* apunta hacia donde desea lanzar el hechizo con el ratón y presiona el botón izquierdo del mismo para lanzarlo.
3. El *Sistema* comprueba que el personaje posee maná (energía mágica) suficiente para lanzar el hechizo seleccionado y lanza el hechizo al mundo.
4. El *Sistema* detecta que el hechizo ha colisionado con un enemigo y hace explotar el proyectil.
5. El *Sistema* le resta una cantidad de vida el enemigo correspondiente al poder del hechizo lanzado pero el enemigo no muere.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

3a El *Sistema* detecta que el personaje no posee el maná suficiente para lanzar el hechizo.

1. El *Sistema* informa del error al *Jugador* y finaliza el caso de uso.

4a El *Sistema* detecta que el hechizo ha colisionado con un objeto del escenario.

1. El *Sistema* hace explotar al proyectil y es eliminado del mundo.

4b El *Sistema* no detecta que el hechizo haya colisionado.

5a El *Sistema* le resta la vida al enemigo correspondiente al poder del hechizo y esta llega a 0 o menos.

1. El *Sistema* elimina al enemigo del juego.

Caso de uso: Mover personaje

Caso de uso Mover personaje

Descripción El *Jugador* desplaza al personaje por el escenario.

Actores *Jugador*.

Precondiciones Se está jugando un nivel.

Postcondiciones Ninguna.

Escenario principal

1. El *Jugador* presiona una de las teclas de movimiento. W (hacia donde mira la cámara), A (hacia la izquierda de la cámara), S (hacia la cámara) y D (hacia la derecha de la cámara).
2. El *Sistema* mueve al personaje en dicha dirección teniendo en cuenta la orientación de la cámara y la velocidad del personaje. El *Sistema* orienta al personaje en dicha dirección.
3. El *Sistema* comprueba que el personaje no ha colisionado con ningún elemento del escenario o enemigo.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

3a El *Sistema* detecta que el personaje ha colisionado con un enemigo o alguna zona del escenario.

1. El *Sistema* restaura la posición anterior del personaje.

Caso de uso: Mover cámara

Caso de uso Mover cámara

Descripción El *Jugador* rota la cámara alrededor del personaje principal.

Actores *Jugador*.

Precondiciones Se está jugando un nivel.

Postcondiciones Ninguna.

Escenario principal

1. El *Jugador* pulsa el botón derecho del ratón y lo mueve en una dirección (arriba, abajo, izquierda o derecha).
2. El *Sistema* desplaza y rota la cámara de forma que gire alrededor del personaje.
3. El *Sistema* comprueba que la cámara no se haya salido de sus límites de rotación.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

3a El *Sistema* percibe que la cámara se ha salido de sus límites de rotación.

1. El *Sistema* corrige la rotación de la cámara.

Caso de uso: Victoria

Caso de uso Victoria

Descripción El *Jugador* ve un resumen con su actuación en la partida y acumula puntos de experiencia. Según el nivel que se haya jugador, es posible que desbloquee un nuevo nivel. Puede volver a jugar, volver a la pantalla de selección de nivel o regresar al menú principal.

Actores *Jugador*.

Precondiciones El *Jugador* acaba de finalizar un nivel.

Postcondiciones Se actualiza la experiencia del jugador.

Escenario principal

1. El *Sistema* calcula la experiencia obtenida en la partida y la muestra desglosada en los apartados: enemigos eliminados, tiempo empleado, vida restante y maná utilizado. Actualiza la experiencia total del perfil seleccionado.
2. El *Sistema* comprueba que el nivel jugado era el último nivel desbloqueado para el perfil seleccionado. Desbloquea el siguiente nivel y muestra el mensaje: *Has desbloqueado un nuevo nivel*. Se actualiza el último nivel desbloqueado para el perfil actual.
3. El *Jugador* selecciona la opción Selección de nivel.
4. El *Sistema* acude a la pantalla de selección de nivel.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

2a El *Sistema* comprueba que el nivel jugado no era el último desbloqueado para el perfil seleccionado y muestra el mensaje: *Has completado este nivel de nuevo*.

2b El *Sistema* comprueba que el nivel jugador era el último nivel desbloqueado para ese perfil pero ya no quedan niveles adicionales, por tanto muestra el mensaje: *Has conseguido salvar la Torre Sagrada*.

3a El *Jugador* selecciona la opción Volver a jugar.

1. El *Sistema* inicia la pantalla de juego con el mismo nivel seleccionado.

3b El *Jugador* selecciona la opción Volver al menú.

1. El *Sistema* regresa al menú principal.

Caso de uso: Créditos

Caso de uso Créditos

Descripción Se muestra la pantalla de créditos con los creadores del juego, el *Jugador* puede volver al menú principal.

Actores *Jugador*.

Precondiciones Ninguna.

Postcondiciones Ninguna.

Escenario principal

1. El *Sistema* muestra la pantalla de créditos.
2. El *Jugador* selecciona la opción Volver.
3. El *Sistema* regresa al menú principal.

Extensiones — flujo alternativo

***a** El *Jugador* cierra la ventana.

1. El *Sistema* libera los recursos y sale de la aplicación.

4.2.3. Modelo conceptual de datos

En esta sección del análisis expondremos el modelo conceptual de datos del sistema. Se listarán las clases conceptuales y las relaciones que existen entre ellas junto a una breve descripción. Para ello emplearemos un diagrama de clases siguiendo la notación *UML*. Nótese que los nombres de las clases están en inglés, se decidió tomar dicha aproximación con el objetivo de poder conseguir colaboradores en un futuro. Cualquier desarrollador de habla no hispana tendría dificultades para colaborar aunque el proyecto le interesase.

State clase que modela un estado genérico del juego, cada estado correspondería a una pantalla diferente. Los estados cuentan con una canción (*Song*) como música ambiental y un número no determinado de efectos de sonido (*SoundFX*).

StateMenu estado que corresponde al menú principal del juego (ver figura 4.2). Cuenta con una escena 3D y los elementos de interfaz necesarios expuestos en el boceto anteriormente referenciado.

StateProfileSelection estado que modela la pantalla de selección de perfil cuyo boceto puede verse en la figura 4.3. Además de la escena tridimensional y los elementos de la interfaz, permite poder gestionar perfiles (*Profile*).

StateLevelSelection estado que representa la pantalla de selección de nivel, su boceto aparece en la figura 4.4. Cuenta con una escena 3D, elementos de interfaz y una lista de niveles para seleccionar (*Level*). Los niveles estarán bloqueados o desbloqueados en función del perfil seleccionado (*Profile*).

StateGame estado de juego, su boceto puede verse en la figura 4.5. El estado de juego gestiona un nivel (*Level*), las apariciones de enemigos (*EnemySpawn*), el jugador (*Player*), los enemigos que actúan durante un momento determinado (*Enemy*) y los hechizos lanzados (*Spell*). Además, lleva las estadísticas de juego (*GameStats*). Se encarga de gestionar el HUD (*Heads Up Display* o controles de juego) y de renderizar la escena. Por último, hace un seguimiento de la cámara y los dispositivos de entrada (teclado y ratón).

StateVictory es el estado que modela la pantalla de victoria tras haber completado un nivel de juego y puede observarse su esquema en la figura 4.6. Además de la escena 3D y los elementos necesarios de la interfaz, se encarga de actualizar el perfil seleccionado (*Profile*) añadiéndole experiencia y desbloqueando un nuevo nivel en caso de ser necesario.

StateCredits corresponde al estado que gestiona la pantalla de visualización de créditos. Simplemente cuenta con una escena 3D y elementos sencillos de interfaz. Su boceto puede verse en la figura 4.7.

Profile representa un perfil de usuario dentro de **Sion Tower**. Cada perfil almacena información como el nombre de usuario, la cantidad de experiencia acumulada y el número del último nivel desbloqueado. Al crear un nuevo perfil la experiencia es igual a cero y el nivel desbloqueado es el primero.

Level modela y gestiona toda la información de un nivel. De cada nivel necesitamos saber su identificador, nombre y descripción. Así mismo, también almacena información sobre el nombre de la canción que debe reproducirse mientras se juega a dicho nivel y la posición inicial del protagonista. Contiene todas las apariciones de los enemigos (*EnemySpawn*) y la malla de navegación (*NavigationMesh*). Por supuesto, también contiene los detalles sobre los objetos del escenario y la iluminación.

Enemy Spawn modela la aparición de un enemigo a lo largo del desarrollo de una partida. De cada aparición interesa saber el tipo de enemigo al que se refiere, su posición y orientación iniciales así como el momento (en segundos desde el inicio del nivel) en el que hará aparición.

NavigationMesh la malla de navegación representa la zona transitable por los enemigos (*Enemy*) y está formada por una colección de vértices y triángulos conexos. En definitiva es un grafo conexo sobre el que se realizan operaciones de inteligencia artificial y búsqueda de caminos.

PointPath representa una ruta formada por puntos en el espacio que lleva de un punto del escenario a otro.

GameObject modela de forma genérica objetos del juego con un modelo colisionable (*Body*). Cuentan con una posición, escala y orientación determinadas.

Actor esta clase engloba de forma genérica a los elementos dinámicos y activos del juego como el jugador (*Player*) y los enemigos (*Enemy*). Hereda de *GameObject* pero además aporta energía vital máxima, energía vital actual, poder, velocidad actual, aceleración, velocidad lineal máxima y velocidad angular. Incluye efectos de sonido (*SoundFX*) como el de aparición, el de ataque y el de recibir daño.

Player hereda de *Actor* y representa al personaje protagonista de la aventura. Aporta su energía mágica actual, el hechizo seleccionado, su posición anterior (en caso de que colisione con un obstáculo, ésta sería restaurada) y el tiempo de recuperación de la energía mágica o maná.

Enemy los enemigos cuentan con su tipo concreto (goblin, diablillo o gólem de hielo) y un tiempo de recuperación entre ataques. Cuando un enemigo desea ir de un punto a otro del escenario obtiene una ruta de puntos *PointPath* que ha de seguir poco a poco.

Spell representa los hechizos o proyectiles mágicos del juego que lanza el protagonista (*Player*). De cada hechizo interesa saber su tipo (Bola de fuego, Furia de Gea o Ventisca), la dirección hacia la que se dirige, su nombre, descripción, poder, coste en maná, velocidad, tiempo que dura su explosión y los efectos de sonido (*SoundFX*) que se producen al ser lanzado y al impactar.

GameStats modela las estadísticas de juego que se generan con la actuación del protagonista (*Player*) durante la partida. Se cada partida es necesario conocer el código del nivel que se juega, el número de enemigos eliminados (*Enemy*), los puntos que se ganan al eliminar a cada tipo de enemigo, los puntos por haber destruido enemigos, el maná utilizado y los puntos por ello. También queremos

conocer la vida restante y los puntos que se ganan con ella. Por último es necesario recuperar la duración de la partida y los puntos que recibe el jugador por haber vencido en dicho tiempo.

SoundFX representa un efecto de sonido de corta duración hace las veces de abstracción de la biblioteca de audio LIBSDL MIXER. Estos efectos deben estar en formato `.wav`.

Song modela una pista de audio de mayor duración y se utiliza para reproducir la banda sonora. También abstrae ciertos aspectos de la biblioteca LIBSDL MIXER y sólo acepta ficheros en formato `.ogg`.

Body representa un cuerpo colisionable de algún elemento del juego, es decir, algo con lo que se puede colisionar y el sistema está preparado para detectarlo. De ellos interesa saber las formas que componen el cuerpo colisionable (*Shape*) y su tipo. El tipo es importante ya que podremos detectar colisiones entre dos tipos concretos de cuerpos y detectar las demás, por ejemplo entre enemigos y hechizos.

Shape modela una forma geométrica tridimensional sencilla de forma genérica. Son los componentes que forman los cuerpos colisionables (*Body*). De ellas sólo interesa saber su nombre.

Sphere clase hija de *Shape* que modela una esfera sencilla. De cada esfera necesitamos saber su centro en coordenadas locales y su radio.

Plane una nueva especialización de *Shape* para modelar un plano infinito en tres dimensiones. Para representar el plano utilizamos un punto del plano (distancia con respecto al origen) y un vector normal (perpendicular) al plano.

AxisAlignedBox se trata de un nuevo tipo de forma (*Shape*) que representa un hexaedro rectangular alineado con los ejes de coordenadas. De él necesitamos saber para poder representarlo correctamente sus vértices mínimo y máximo.

OrientedBox es el cuarto y último tipo de forma (*Shape*) del modelo de datos inicial del videojuego. Es un hexaedro rectangular no alineado con los ejes, sino que cuenta con rotación. Es necesario conocer su centro, las tres distancias a las caras de cada eje local y los ejes locales y rotados sobre los que se basa.

CollisionManager es el gestor de colisiones del juego, gestiona todos los cuerpos colisionables (*Body*) y es capaz de conocer cuándo entran en contacto y avisar de tal evento.

Por razones de espacio y de dificultad a la hora de distribuir todas las clases conceptuales en un mismo diagrama, se ha decidido separarlas en dos esquemas diferentes. El primero corresponde al de la figura 4.9 e incluye los subistemas relacionados con las pantallas de juego. Por otro lado, el segundo diagrama puede verse en la figura 4.10 y abarca el subsistema de objetos de juego, modelos colisionables e inteligencia artificial.

4.2.4. Modelo de comportamiento del sistema

En esta sección desarrollaremos el modelo de comportamiento del sistema. Consideraremos al sistema como ente que engloba a todos los objetos. El modelo se dividirá en dos partes bien diferenciadas:

- Diagramas de secuencia de sistema: nos muestran la secuencia de eventos entre actores y sistema. También nos ayudan a identificar las operaciones del sistema.
- Contratos para las operaciones del sistema: describen en detalle qué hace cada operación del sistema.

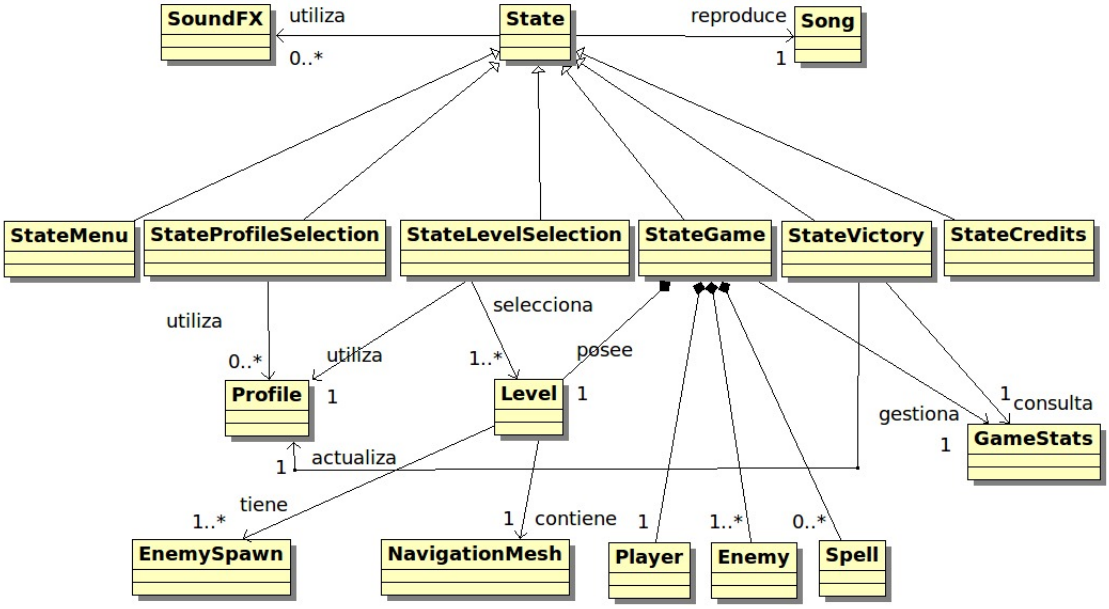


Figura 4.9: Diagrama de clases conceptuales (parte 1)

No todos los posibles diagramas de secuencia ni contratos de operaciones aparecerán especificados. En este documento nos centraremos en los más relevantes, es decir, los que impliquen algún tipo de cambio en el sistema.

Caso de uso: Menú (escenario principal)

Operación InicioAplicacion()

Actores *Jugador*, *Sistema*.

Responsabilidades iniciar la aplicación y preparar el sistema para la ejecución del juego. Mostrar el menú principal.

Precondiciones ninguna.

Postcondiciones

- Creación de un objeto *stateMenu* de la clase *StateMenu*.
- Creación de un objeto *song* de la clase *Song*.

Operación SeleccionarJugar()

Actores *Jugador*, *Sistema*.

Responsabilidades salir del menú principal y entrar en la pantalla de selección de perfil.

Precondiciones

- Existe un objeto *stateMenu* de la clase *StateMenu*.

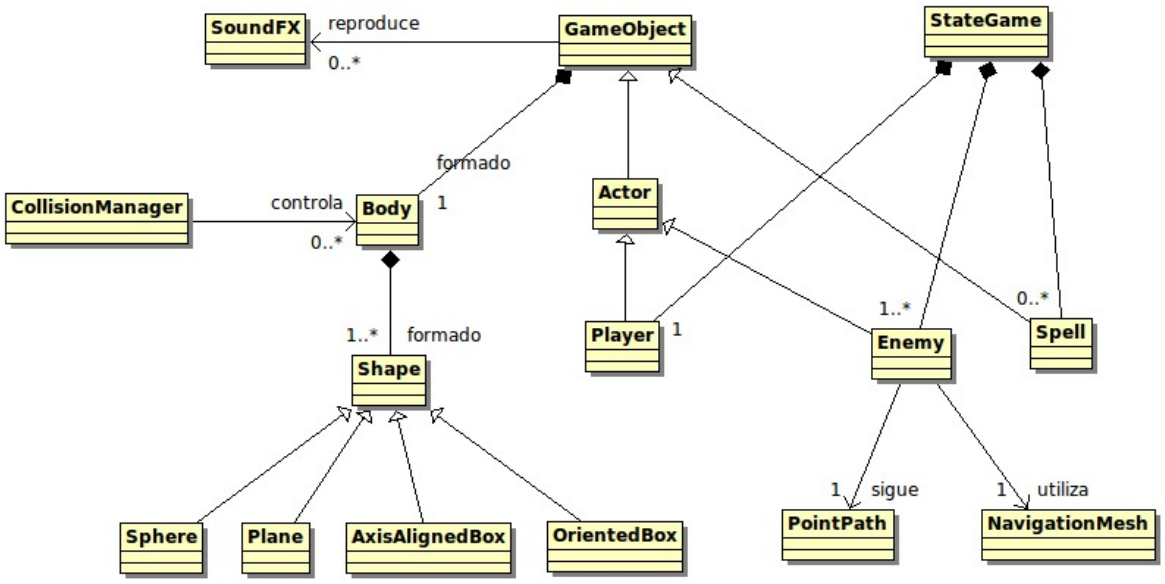


Figura 4.10: Diagrama de clases conceptuales (parte 2)

Postcondiciones

- Destrucción del objeto *stateMenu*.
- Se destruye el objeto *song*.

Caso de uso: Menú (escenario 3a)

Operación SeleccionarCreditos()

Actores *Jugador*, *Sistema*.

Responsabilidades salir del menú principal y entrar en la pantalla de créditos.

Precondiciones

- Existe un objeto *stateMenu* de la clase *StateMenu*.

Postcondiciones

- Destrucción del objeto *stateMenu*.
- Se destruye el objeto *song*.

Caso de uso: Menú (escenario 3b)

Operación SeleccionarSalir()

Actores *Jugador*, *Sistema*.

Responsabilidades salir del menú principal y cerrar la aplicación.

Precondiciones

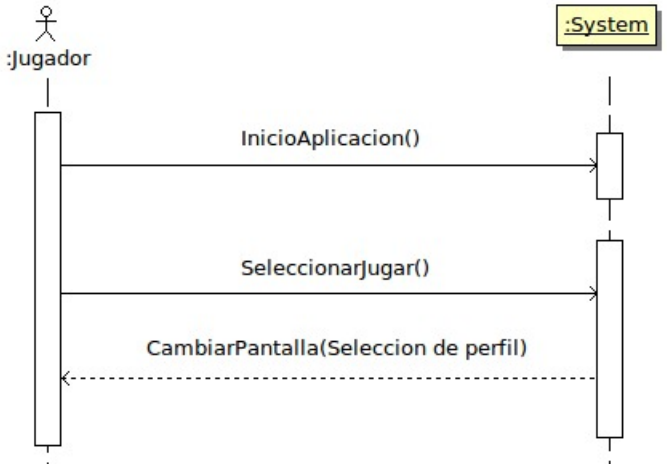


Figura 4.11: Diagrama de secuencia: Menú (escenario principal)

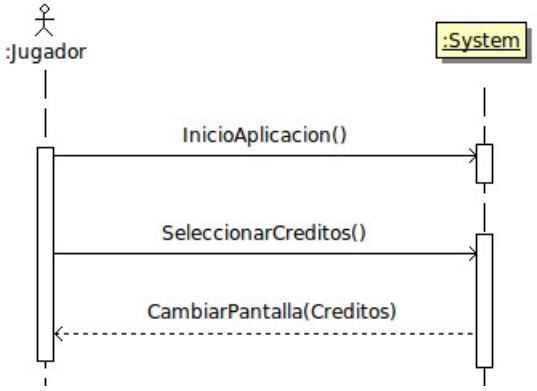


Figura 4.12: Diagrama de secuencia: Menú (escenario 3a)

- Existe un objeto *stateMenu* de la clase *StateMenu*.

Postcondiciones

- Destrucción del objeto *stateMenu*.

Caso de uso: Selección de perfil (escenario principal)

Operación PantallaSeleccionPerfil()

Actores *Jugador*, *Sistema*.

Responsabilidades cargar y mostrar la pantalla de selección de perfil.

Precondiciones ninguna.

Postcondiciones

- Creación de un objeto *stateProfile* de la clase *StateProfile*.
- Creación de objetos de la clase *Profile* por cada perfil que exista en memoria secundaria.

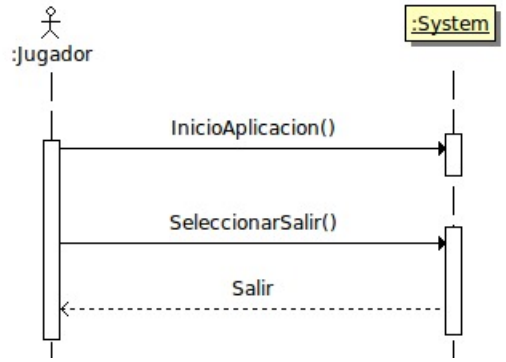


Figura 4.13: Diagrama de secuencia: Menú (escenario 3b)

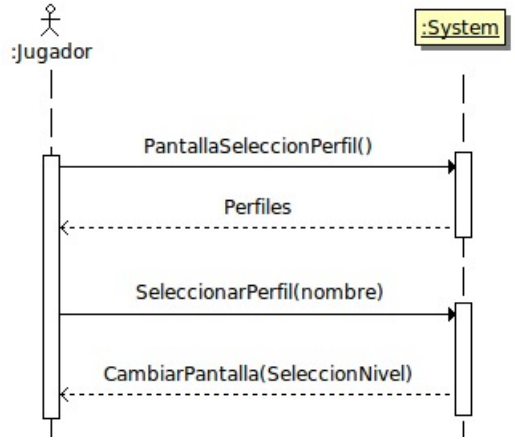


Figura 4.14: Diagrama de secuencia: Selección de perfil (escenario principal)

- Creación de un objeto *song* de la clase *Song*.

Operación SeleccionarPerfil(nombre)

Actores *Jugador*, *Sistema*.

Responsabilidades marcar un perfil determinado como seleccionado.

Precondiciones

- Existe un perfil *profile* con *profile.name = nombre*.

Postcondiciones

- Marcado del perfil *profile* como seleccionado.
- Se destruye el objeto *song*.
- Destrucción del objeto *stateProfile*.

Caso de uso: Selección de perfil (escenario 3a)

Operación EliminarPerfil(nombre)

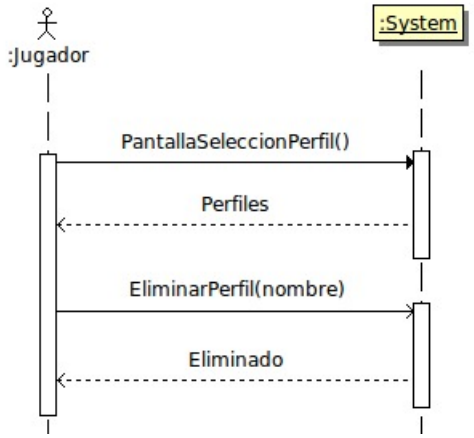


Figura 4.15: Diagrama de secuencia: Selección de perfil (escenario 3a)

Actores *Jugador*, *Sistema*.

Responsabilidades elimina un perfil del sistema.

Precondiciones

- Existe un perfil *profile* con *profile.name = nombre*.

Postcondiciones

- Destrucción del objeto *profile*.

Caso de uso: Selección de perfil (escenario 3b)

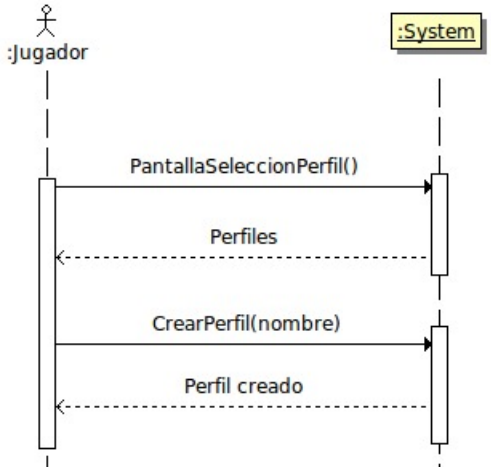


Figura 4.16: Diagrama de secuencia: Selección de perfil (escenario 3b)

Operación CrearPerfil(nombre)

Actores *Jugador*, *Sistema*.

Responsabilidades crea un nuevo perfil en el sistema.

Precondiciones

- No existe un perfil *profile* con *profile.name = nombre*.

Postcondiciones

- Creación de un objeto *profile* de la clase *Profile*.
- Modificación de atributos: *profile.name = nombre*, *profile.unlockedLevel = 1* y *profile.experience = 0*.

Caso de uso: Selección de nivel (escenario principal)

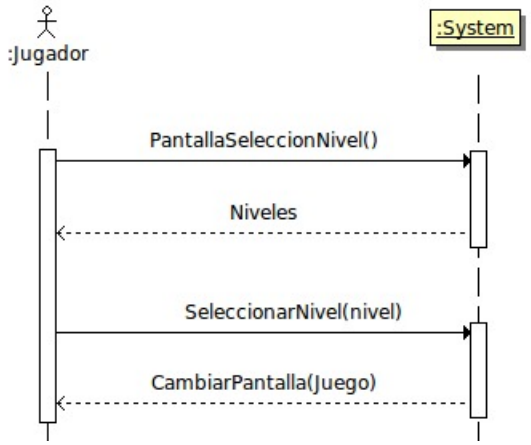


Figura 4.17: Diagrama de secuencia: Selección de nivel (escenario principal)

Operación PantallaSeleccionNivel()

Actores *Jugador*, *Sistema*.

Responsabilidades crea y muestra la pantalla de selección de nivel.

Precondiciones ninguna.

Postcondiciones

- Creación de un objeto *stateLevel* de la clase *StateLevel*.
- Creación de objetos de la clase *Level* por cada nivel encontrado en memoria secundaria.
- Creación de un objeto *song* de la clase *Song*.

Operación SeleccionarNivel(nivel)

Actores *Jugador*, *Sistema*.

Responsabilidades crea y muestra la pantalla de selección de nivel.

Precondiciones

- Existe un objeto *level* de la clase *Level* de forma que *level.id = nivel*.

Postcondiciones

- Marcado del objeto *level* como nivel seleccionado.
- Se destruye el objeto *song*.
- Destrucción del objeto *stateLevel*.

Caso de uso: Jugar (escenario principal)

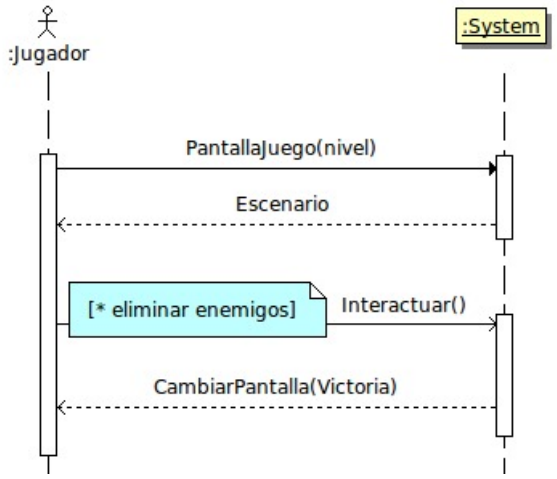


Figura 4.18: Diagrama de secuencia: Jugar (escenario principal)

Operación PantallaJuego(nivel)

Actores *Jugador*, *Sistema*.

Responsabilidades carga y muestra la pantalla de juego con el nivel seleccionado, inicia el juego.

Precondiciones ninguna.

Postcondiciones

- Creación de un objeto *stateGame* de la clase *StateGame*.
- Creación de un objeto *song* de la clase *Song*.
- Creación de un objeto *plater* de la clase *Player* en la posición que indique el nivel.
- Creación de una relación entre *player.body* y *CollisionManager*.
- Creación de un objeto de la clase *Enemy* por cada enemigo que exista en el juego en la posición que indique el nivel.
- Creación de una relación entre cada *enemy.body* y *CollisionManager*.
- Creación de un objeto *gameStats* de la clase *GameStats* con las estadísticas iniciales.
- Creación de un objeto *navigationMesh* de la clase *NavigationMesh* con la malla inicializada.
- Creación de enlaces entre los objetos de la clase *Enemy* y *navigationMesh*.

Operación Interactuar()

Actores *Jugador*, *Sistema*.

Responsabilidades permite al *Jugador* interactuar con el mundo 3D y los elementos que lo rodean.

Precondiciones

- Existe un objeto *stateGame*.
- Existe un objeto *player*.
- Existe algún objeto de la clase *Enemy*.

Postcondiciones ninguna.

Caso de uso: Jugar (escenario 3a)

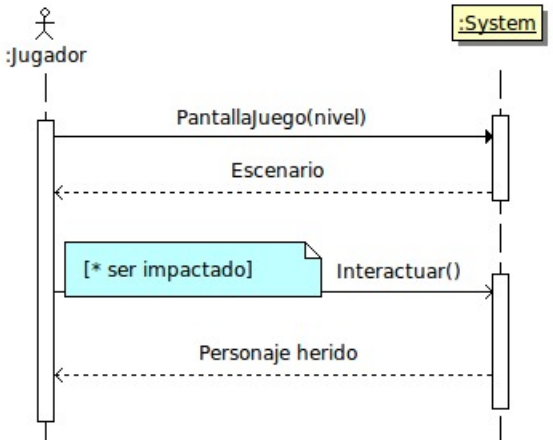


Figura 4.19: Diagrama de secuencia: Jugar (escenario 3a)

Caso de uso: Jugar (escenario 5a)

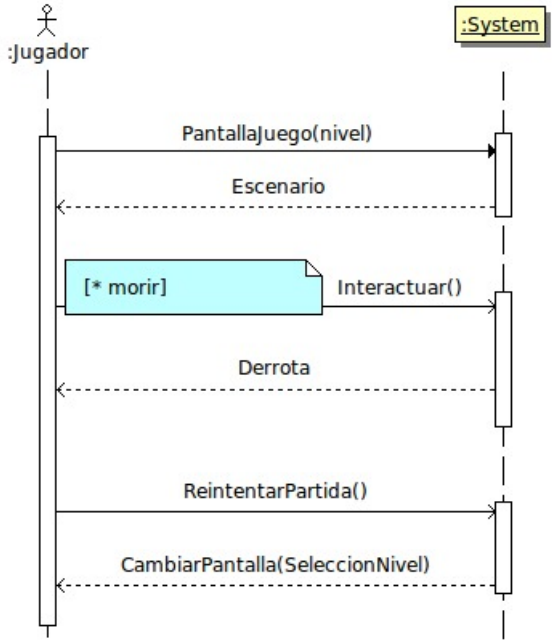


Figura 4.20: Diagrama de secuencia: Jugar (escenario 5a)

Operación ReintentarPartida()

Actores *Jugador*, *Sistema*.

Responsabilidades se destruye el estado de juego junto a sus elementos y se regresa a la pantalla de selección de nivel.

Precondiciones

- Existe un objeto *stateGame*.
- Existe un objeto *player* de forma que *player.life <= 0*.

Postcondiciones

- Se destruye el objeto *navigationMesh*.
- Se destruye el objeto *song*.
- Se eliminan el enlace entre *player.body* y *CollisionManager*.
- Se destruye el objeto *player*.
- Se eliminan los enlaces entre los *enemy.body* y *CollisionManager*.
- Se destruyen todos los objetos de la clase *Enemy*.
- Se eliminan los enlaces entre los *spell.body* y *CollisionManager*.
- Se destruyen todos los objetos de la clase *Spell*.
- Se destruye el objeto *gameStats*.
- Se destruye el objeto *stateGame*.

Caso de uso: Pausar (escenario principal)

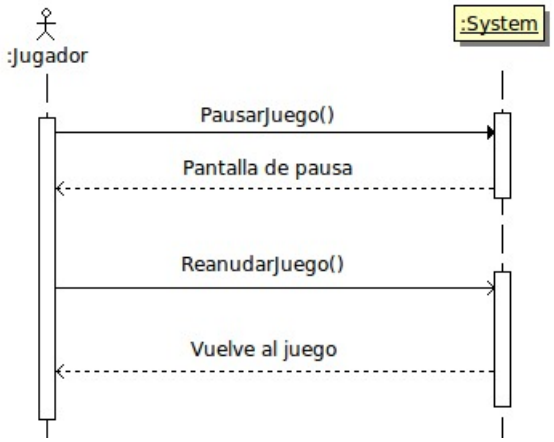


Figura 4.21: Diagrama de secuencia: Pausa (escenario principal)

Operación PausarJuego()

Actores *Jugador*, *Sistema*.

Responsabilidades interrumpe la partida y muestra el menú de pausa.

Precondiciones

- Existe un objeto *stateGame* y se está jugando una partida.

Postcondiciones ninguna.

Operación ReanudarJuego()

Actores *Jugador*, *Sistema*.

Responsabilidades oculta el menú de pausa y continúa la partida.

Precondiciones

- Existe un objeto *stateGame* y el juego estaba pausado.

Postcondiciones ninguna.

Caso de uso: Pausar (escenario 3a)

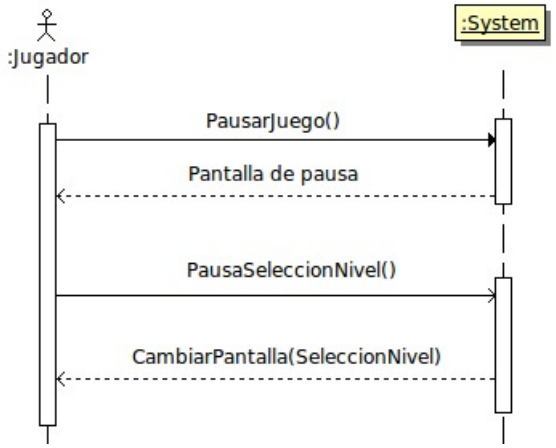


Figura 4.22: Diagrama de secuencia: Pausa (escenario 3a)

Operación PausaSeleccionNivel()

Actores *Jugador*, *Sistema*.

Responsabilidades oculta el menú de pausa, destruye el nivel y regresa a la pantalla de selección de nivel.

Precondiciones

- Existe un objeto *stateGame* y el juego estaba pausado.

Postcondiciones

- Se destruye el objeto *navigationMesh*.
- Se destruye el objeto *song*.
- Se eliminan el enlace entre *player.body* y *CollisionManager*.
- Se destruye el objeto *player*.
- Se eliminan los enlaces entre los *enemy.body* y *CollisionManager*.
- Se destruyen todos los objetos de la clase *Enemy*.
- Se eliminan los enlaces entre los *spell.body* y *CollisionManager*.

- Se destruyen todos los objetos de la clase *Spell*.
- Se destruye el objeto *gameStats*.
- Se destruye el objeto *stateGame*.

Caso de uso: Pausar (escenario 3b)

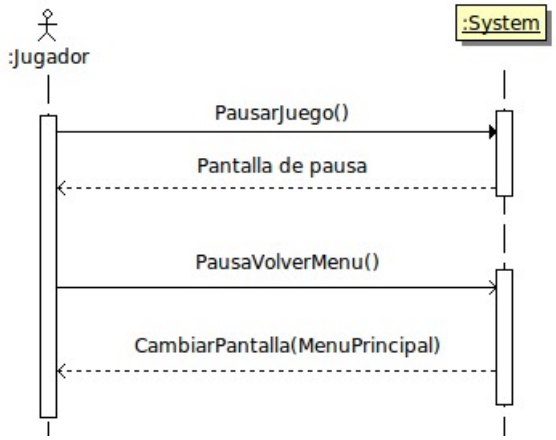


Figura 4.23: Diagrama de secuencia: Pausa (escenario 3b)

Operación PausaVolverMenu()

Actores *Jugador*, *Sistema*.

Responsabilidades oculta el menú de pausa, destruye el nivel y regresa al menú principal del juego.

Precondiciones

- Existe un objeto *stateGame* y el juego estaba pausado.

Postcondiciones

- Se destruye el objeto *navigationMesh*.
- Se destruye el objeto *song*.
- Se eliminan el enlace entre *player.body* y *CollisionManager*.
- Se destruye el objeto *player*.
- Se eliminan los enlaces entre los *enemy.body* y *CollisionManager*.
- Se destruyen todos los objetos de la clase *Enemy*.
- Se eliminan los enlaces entre los *spell.body* y *CollisionManager*.
- Se destruyen todos los objetos de la clase *Spell*.
- Se destruye el objeto *gameStats*.
- Se destruye el objeto *stateGame*.

Caso de uso: Lanzar hechizo (escenario principal)

Operación SeleccionarHechizo(hechizo)

Actores *Jugador*, *Sistema*.

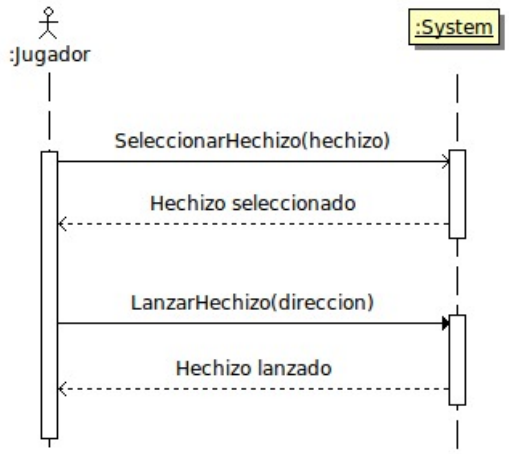


Figura 4.24: Diagrama de secuencia: Lanzar hechizo (escenario principal)

Responsabilidades marca un hechizo como seleccionado.

Precondiciones

- Existe un objeto *stateGame* de la clase *StateGame* y se está jugando una partida.
- Existe un objeto *player* de la clase *Player*.

Postcondiciones

- Modificación de atributo, *player.selectedSpell = hechizo*.

Operación LanzarHechizo(direccion)

Actores *Jugador*, *Sistema*.

Responsabilidades hace que el personaje lance el hechizo seleccionado hacia una dirección determinada.

Precondiciones

- Existe un objeto *stateGame* de la clase *StateGame* y se está jugando una partida.
- Existe un objeto *player* de la clase *Player*.

Postcondiciones

- Se crea un objeto *spell* de la clase *Spell*.
- Atributos: *spell.type = player.selectedSpell*, el resto de atributos del hechizo se determinan según el tipo seleccionado.
- Modificación de atributo: *spell.direction = direccion*.

Caso de uso: Mover personaje (escenario principal)

Operación MoverPersonaje(teclas, camara)

Actores *Jugador*, *Sistema*.

Responsabilidades mueve al personaje por el escenario siguiendo las teclas y en función de la cámara.

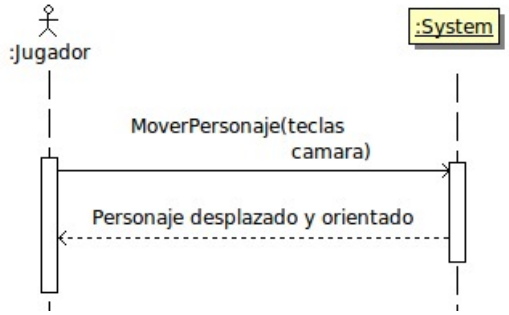


Figura 4.25: Diagrama de secuencia: Mover personaje (escenario principal)

Precondiciones

- Existe un objeto *stateGame* de la clase *StateGame* y se está jugando una partida.
- Existe un objeto *player* de la clase *Player*.

Postcondiciones

- Modificación de atributos: *player.position* y *player.direction* en función de la tecla pulsada y la posición relativa de la cámara.

Caso de uso: Mover cámara (escenario principal)

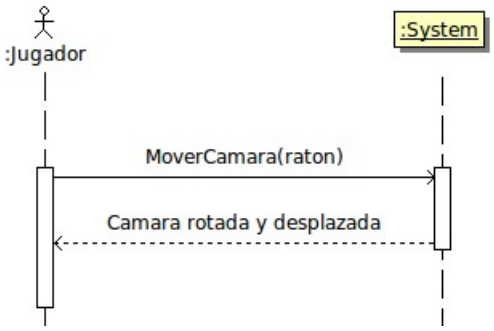


Figura 4.26: Diagrama de secuencia: Mover camara (escenario principal)

Operación MoverCamara(raton)

Actores *Jugador*, *Sistema*.

Responsabilidades mueve la cámara alrededor del personaje.

Precondiciones

- Existe un objeto *stateGame* de la clase *StateGame* y se está jugando una partida.
- Existe un objeto *player* de la clase *Player*.

Postcondiciones

- Modificación de atributos: *camera.position* y *camera.orientation* en función del gesto del ratón y de *player.position*.

Caso de uso: Victoria (escenario principal)

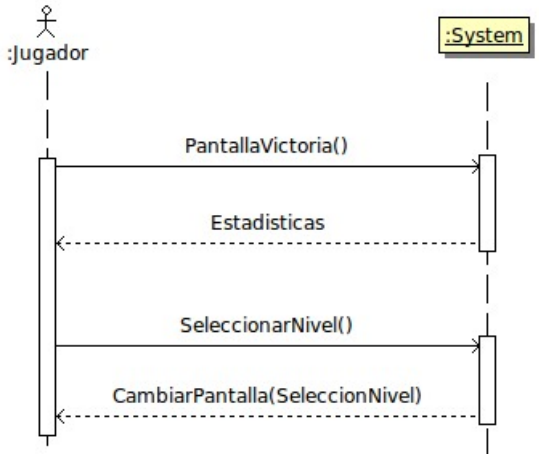


Figura 4.27: Diagrama de secuencia: Victoria (escenario principal)

Operación PantallaVictoria()

Actores *Jugador*, *Sistema*.

Responsabilidades carga y muestra la pantalla de victoria.

Precondiciones

- Existe un objeto *gameStats* de la clase *GameStats* con las estadísticas de juego de la partida anterior. y se está jugando una partida.
- Existe un objeto *profile* de la clase *Profile* correspondiente al perfil seleccionado.

Postcondiciones

- Creación del objeto *stateVictory* de la clase *StateVictory*.
- Creación del objeto *song* de la clase *Song*.
- Modificación de atributos: *player.experiencie* y *player.unlockedLevel* en función del objeto *gameStats* y del nivel que se acabe de jugar.

Operación SeleccionarNivel()

Actores *Jugador*, *Sistema*.

Responsabilidades destruye la pantalla de victoria y vuelve a la de selección de nivel.

Precondiciones

- Existe un objeto *stateVictory* de la clase *StateVictory*.

Postcondiciones

- Destrucción del objeto *song*.

- Destrucción del objeto *stateVictory*.

Caso de uso: Victoria (escenario 3a)

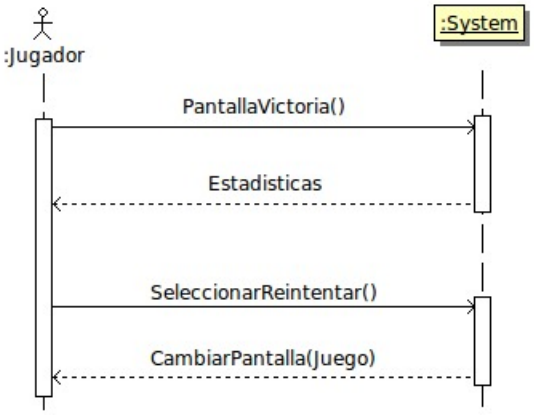


Figura 4.28: Diagrama de secuencia: Victoria (escenario 3a)

Operación SeleccionarReintentar()

Actores *Jugador*, *Sistema*.

Responsabilidades destruye la pantalla de victoria y vuelve a jugar al mismo nivel en estado de juego.

Precondiciones

- Existe un objeto *stateVictory* de la clase *StateVictory*.

Postcondiciones

- Destrucción del objeto *song*.
- Destrucción del objeto *stateVictory*.

Caso de uso: Victoria (escenario 3b)

Operación SeleccionarMenu()

Actores *Jugador*, *Sistema*.

Responsabilidades destruye la pantalla de victoria y vuelve al menú principal.

Precondiciones

- Existe un objeto *stateVictory* de la clase *StateVictory*.

Postcondiciones

- Destrucción del objeto *song*.
- Destrucción del objeto *stateVictory*.

Caso de uso: Créditos (escenario principal)

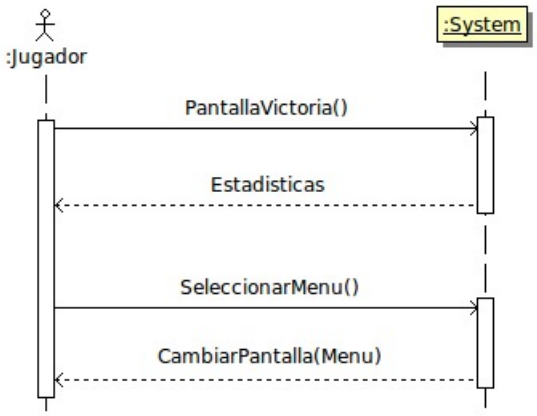


Figura 4.29: Diagrama de secuencia: Victoria (escenario 3b)

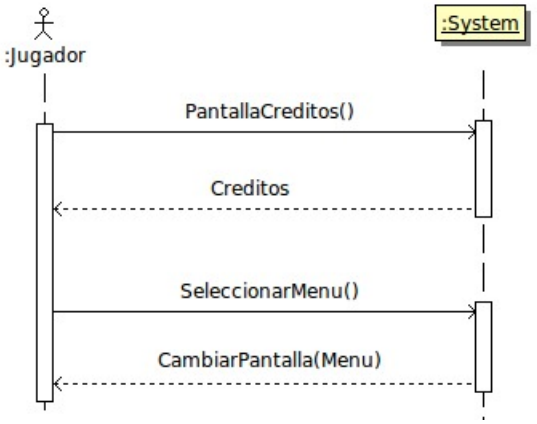


Figura 4.30: Diagrama de secuencia: Créditos (escenario principal)

Operación PantallaCreditos()

Actores *Jugador*, *Sistema*.

Responsabilidades crea y muestra la pantalla de créditos.

Precondiciones ninguna.

Postcondiciones

- Creación del objeto *stateCredits* de la clase *StateCredits*.
- Creación del objeto *song* de la clase *Song*.

Operación SeleccionarMenu()

Actores *Jugador*, *Sistema*.

Responsabilidades destruye la pantalla de créditos y vuelve al menú principal.

Precondiciones

- Existe un objeto *stateCredits* de la clase *StateCredits*.

Postcondiciones

- Destrucción del objeto *song*.
- Destrucción del objeto *stateCredits*.

4.3. Diseño

Al igual que se hizo en la fase de análisis de **Sion Tower** (sección 4.2), durante la fase de diseño también emplearemos la notación *UML*. Nos centraremos en cómo hace el sistema lo que debe hacer para cumplir sus requisitos. Por supuesto, dejaremos espacio para variaciones en la fase de implementación.

Mostraremos el diagrama de clases de diseño dividido por subsistemas para una mayor claridad. Acompañaremos cada figura de una pequeña explicación sobre el subsistema que ilustra.

4.3.1. Diagrama de clases de diseño

Como hemos mencionado anteriormente, separaremos los diagramas de clases de diseño para obtener una mayor claridad. Dicha separación se basará en el cometido de cada subsistema. El primer diagrama (ver figura 4.31) corresponde a las clases encargadas de iniciar el juego, controlar su funcionamiento general y proporcionar soporte a otras clases.

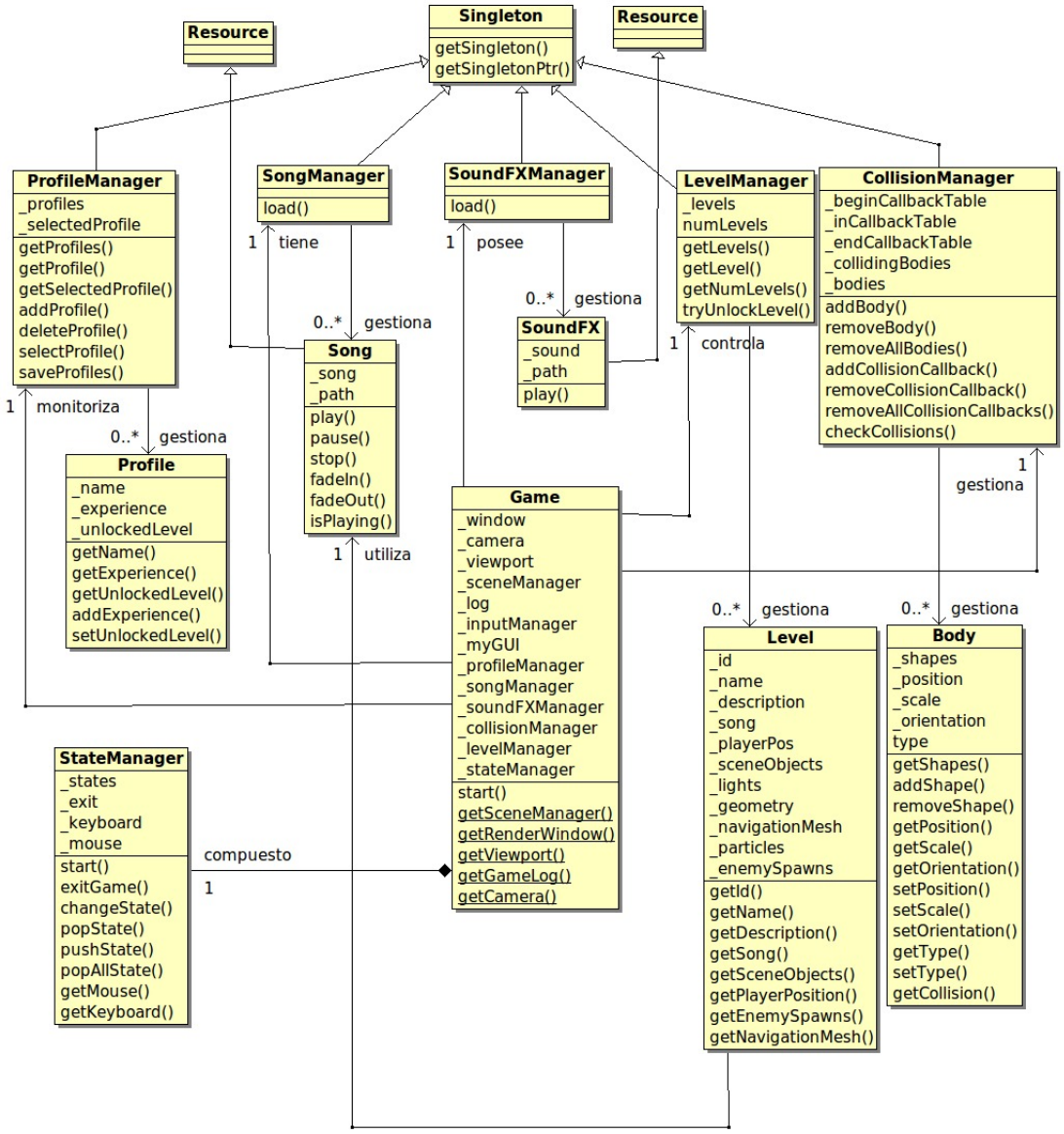


Figura 4.31: Diagrama de clases de diseño: sistema general

El segundo diagrama (ver figura 4.32) representa la gestión de estados de **Sion Tower**. La clase *StateManager* controla los estados y las transiciones entre ellos mientras que cada clase descendiente de *State* representa cada una de las pantallas de juego. Así tenemos *StateMenu*, *StateProfile*, *StateLevel*, *StateGame*, *StateVictory*, y *StateCredits*. Los estados utilizan una melodía (*Song*) y reproducen efectos de sonido *SoundFX*. El estado de selección de perfiles elige un perfil utilizando el gestor (*ProfileManager*) mientras que el selector de niveles consulta cuál es el perfil seleccionado. Así mismo, escoge un nivel entre los disponibles a través del gestor de niveles *LevelManager*. Durante el juego se van actualizando las estadísticas de juego *GameStats* y en la pantalla de victoria se consultan para actualizar el perfil seleccionado.

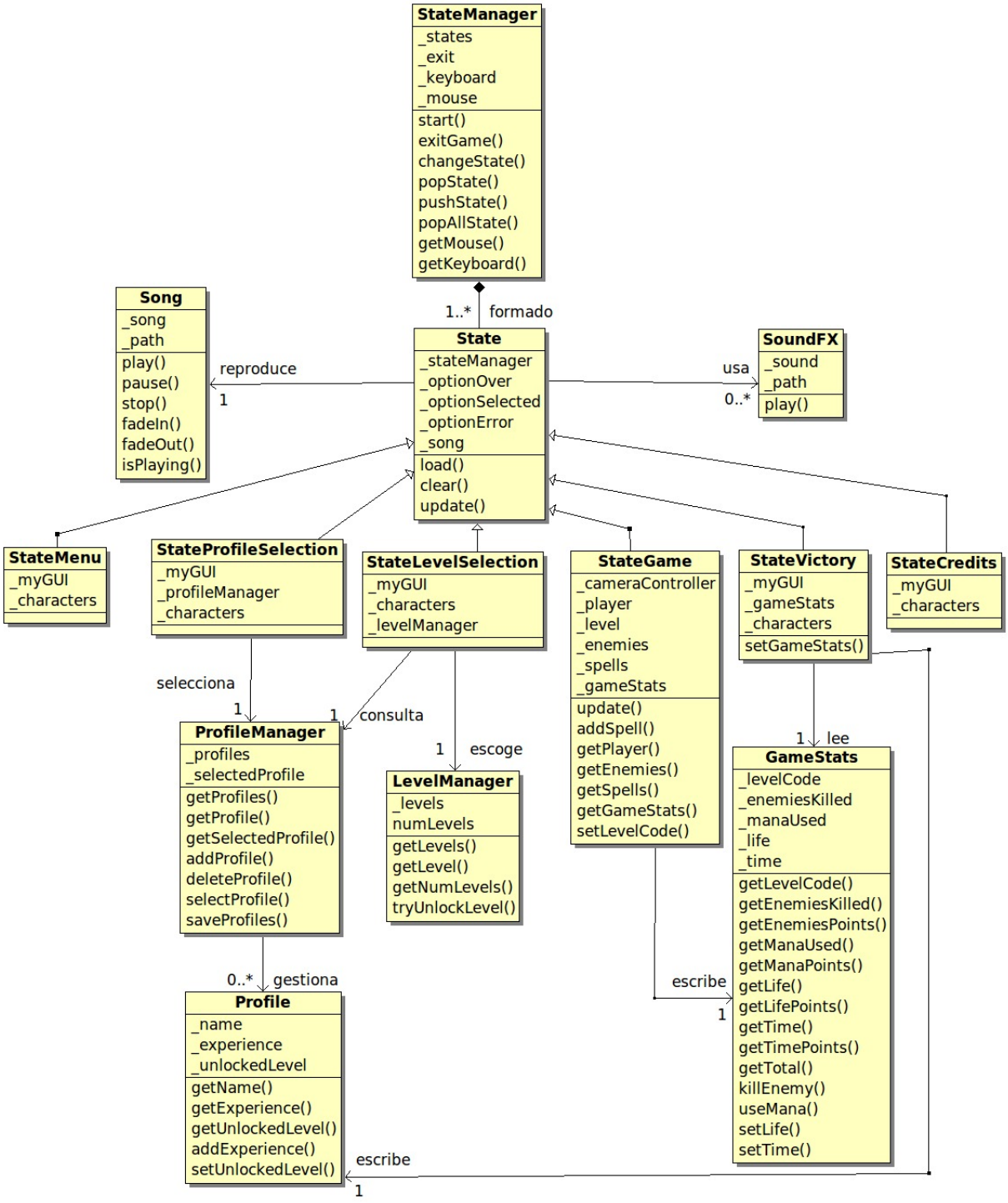


Figura 4.32: Diagrama de clases de diseño: sistema de estados

En el tercer diagrama (ver figura 4.33) se muestran las clases relacionadas con el sistema de juego. Podemos observar la jerarquía de herencia de los objetos de juego (*GameObject*) o el uso de la malla de navegación entre otros (*NavigationMesh* y *Cell*).

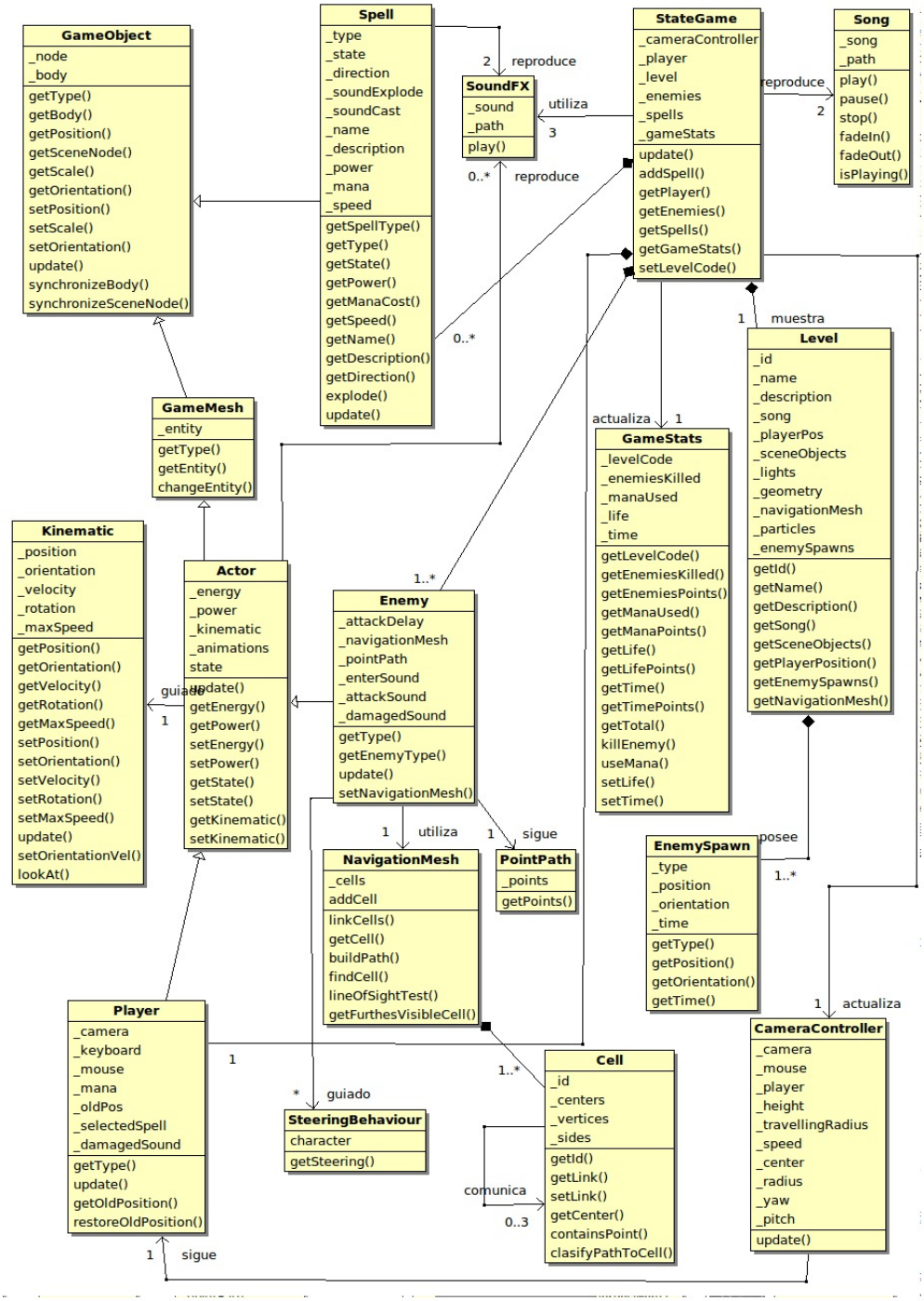


Figura 4.33: Diagrama de clases de diseño: sistema de juego

En el cuarto y último diagrama (ver figura 4.34 se muestra el sistema de inteligencia artificial y algoritmos de movimiento. La clase *SteeringBehaviour* modela los algoritmos de movimiento de forma genérica y de ella heredan implementaciones concretas. Estos algoritmos modifican el aspecto dinámico de los personajes aplicando fuerzas y aceleraciones, veremos más sobre este tema en el punto 4.4.8 en la página 133.

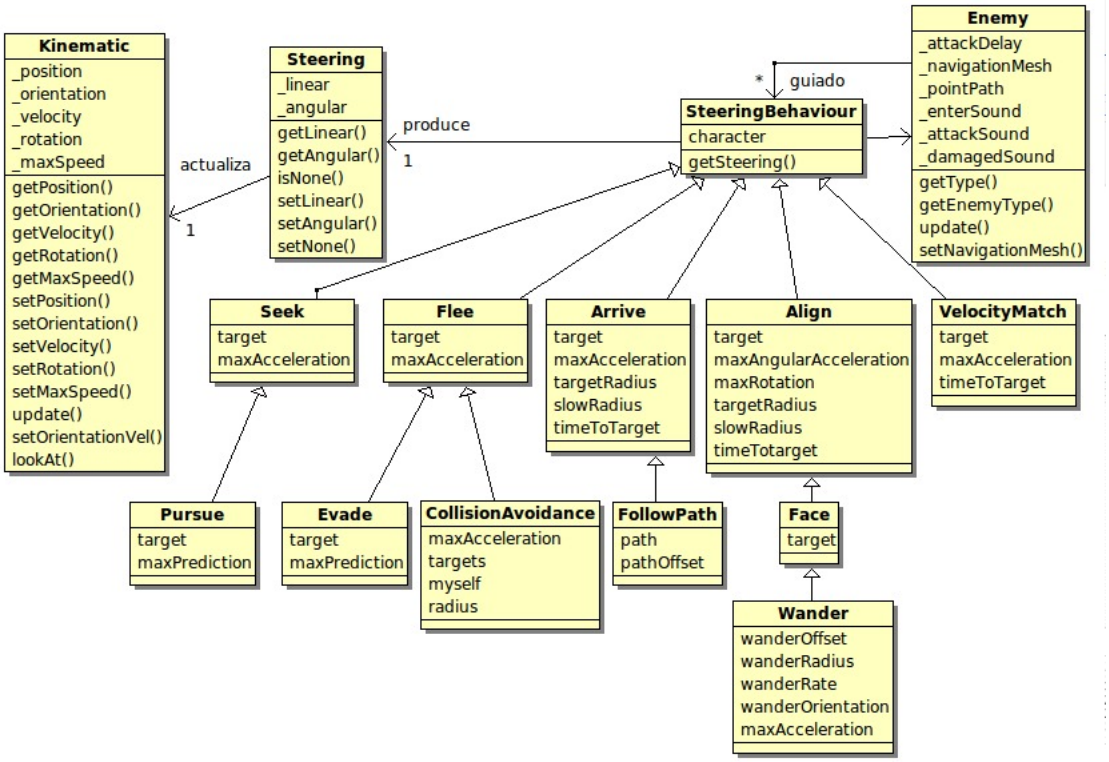


Figura 4.34: Diagrama de clases de diseño: Steering Behaviors

4.4. Implementación

A lo largo de toda la fase de implementación se han ido encontrando diversos obstáculos en distintos subsistemas. Estos han surgido bien por desconocimiento de la materia o por la dificultad que entraña la misma. En cualquier caso, en este capítulo haremos un repaso por los detalles más interesantes de la implementación de **Sion Tower**. En cada uno de ellos se expondrá el problema a resolver, las dificultades encontradas y la solución propuesta adjuntando si es necesario pequeños fragmentos de código.

Para consultar el código fuente completo del juego, lo mejor es acudir al repositorio *Subversion* de la forja de RedIRIS en la siguiente dirección.

`https://forja.rediris.es/scm/?group_id=820`

La documentación del código generada con *Doxygen* [11] facilitará en gran medida la lectura del código. Dicha documentación complementaria puede ser accedida desde la siguiente dirección web.

`http://siondream.com/siontower-doxygen`

4.4.1. Gestión de estados de juego

Estados de juego, clase State

Como hemos visto anteriormente, en **Sion Tower** contamos con varias pantallas entre menús y el estado de juego. Cada una de estas pantallas está modelada por una clase hija de *State*. Se trata de una clase virtual pura que incluye un método de actualización *update()* y varios manejadores de eventos siguiendo el esquema de la biblioteca OIS [58] (pulsar tecla, liberar tecla, pulsar botón del ratón, liberar botón del ratón y mover ratón). El comportamiento por defecto de los manejadores de eventos consiste en no hacer nada. Si los sobrecargamos en una clase hija, podremos indicar la respuesta deseada.

Los estados pueden contar con todos sus elementos cargados en memoria (estado cargado) o pueden estar creados pero no listos para su uso. Eso permite evitar crear y destruir nuevos estados constantemente, simplemente llamaríamos a sus métodos *load()* y *clear()* según sea necesario.

La interfaz del juego utiliza la biblioteca MyGUI [28]. Esta biblioteca se basa en unos ficheros *.layout*, en el fondo son unos XML sencillos para definir los elementos de la interfaz (botones, paneles, etiquetas, etc). Cada estado debe traducir toda su interfaz empleando GETTEXT [18] a través del método *translate()*. Hablaremos de la internacionalización del proyecto en la sección 4.4.2 (página 87)

El método *adjustFontHeight()* se utiliza para establecer el tamaño de la letra de los elementos de la interfaz en función de la resolución de la ventana. Recordemos que **Sion Tower** es independiente de la resolución (siempre y cuando esta tenga una relación de aspecto 16:10).

A continuación se muestra la definición de la clase *State*:

```
1  class State {
2      public:
3          // Constructor y destructor
4          State(StateManager* stateManager);
5          virtual ~State() {};
6
7          void setStateManager(StateManager* stateManager);
8
9
10         // Cargar y liberar escena
11         virtual void load() {};
12         virtual void clear() {};
13
14         // Gestores de eventos
15         virtual void update(Ogre::Real deltaT, bool active) = 0;
16         virtual bool keyPressed(const OIS::KeyEvent &arg);
17         virtual bool keyReleased(const OIS::KeyEvent &arg);
18         virtual bool mouseMoved(const OIS::MouseEvent &arg);
19         virtual bool mousePressed(const OIS::MouseEvent &arg,
20                                   OIS::MouseButtonID id);
21         virtual bool mouseReleased(const OIS::MouseEvent &arg,
22                                    OIS::MouseButtonID id);
23
24     protected:
25         // Log
26         Ogre::Log* _log;
```

```
27
28          // Gestor escena
29          Ogre::SceneManager* _sceneManager;
30
31          StateManager* _stateManager;
32          bool _loaded;
33
34          // Obliga a traducir todos los estados
35          virtual void translate() = 0;
36
37          // Obliga a ajustar el tamano del texto
38          virtual void adjustFontHeight() = 0;
39  };
```

Pila de estados, clase StateManager

La clase *StateManager* se encarga de gestionar los estados y las transiciones entre los mismos. Para ello cuenta con una pila de estados interna de forma que podemos añadir un estado encima de la pila mediante una operación *push()* o sacarlo mediante *pop()*. Esto resulta muy útil cuando avanzamos o retrocedemos de forma lineal por varios menús y nos evita tener que estar creando y destruyendo estados constantemente. Además, podemos eliminar todos los estados de la pila utilizando *popAllStates()* o cambiar el tope de la pila por otro estado con *changeState()*. La figura 4.35 ilustra el proceso.

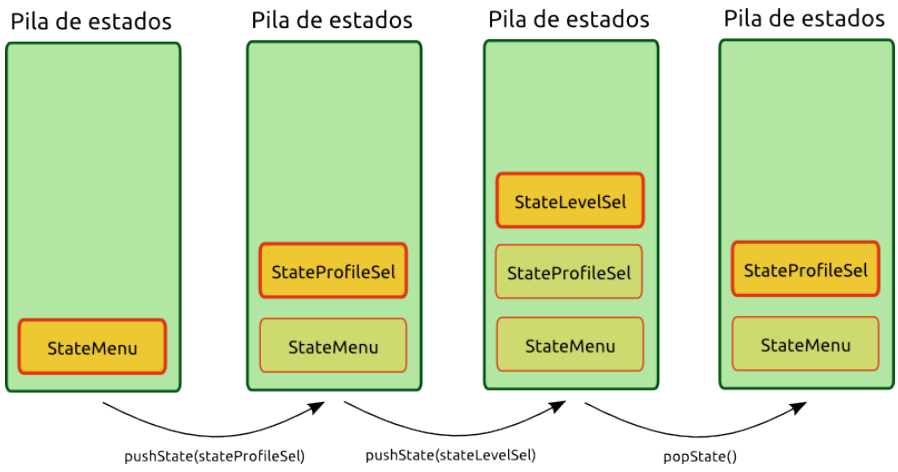


Figura 4.35: Pila de estados

StateManager hereda de las clases *FrameListener*, *WindowEventListener*, *KeyListener* y *MouseListener*. De esta forma, el gestor de estados sigue el patrón de diseño *Observer* [15] para diversos eventos: etapas del renderizado, cambios en la ventana, interacción con el teclado y con el ratón respectivamente. El gestor de estado delega en el estado activo los eventos de teclado y ratón para que éste último les de respuesta.

Cuando un estado detecta que se ha de producir un cambio de estado, informa al *StateManager* de ello. No obstante, no se puede destruir el estado actual para cambiar a otro porque lo normal es que nos encontremos en mitad de una iteración del bucle de juego (game loop) [19]. Podría ocurrir que destruyésemos el estado actual a la vez que se esta ejecutando su método *update()* lo que provocaria accesos

de memoria a basura. El gestor de estados almacena una pila de operaciones pendientes de forma que cuando solicitamos una operación *pop()* o *push()* no se realiza en el mismo instante. Una vez finalizada la iteración del bucle de juego actual, se procede a realizar las operaciones pendientes, esta vez con total seguridad (método privado *perfomOperations()*).

Cuando se llama al método *start()* se inicia el bucle de renderizado gestionado por el motor OGRE3D. Antes de renderizar la escena, se dispara el evento *frameStarted* y el gestor de estados actualiza en orden descendente la pila de estados activos.

A continuación se muestra la definición de la clase *StateManager*:

```
class StateManager: public Ogre::FrameListener,
                    public Ogre::WindowEventListener,
                    public OIS::KeyListener,
                    public OIS::MouseListener {
    public:
        StateManager(Game* game, OIS::InputManager* inputManager);
        ~StateManager();

        void start();
        void exitGame();

        // Control de la pila de estados
        void changeState(State* state);
        State* changeState(const Ogre::String& stateName);
        void popState();
        void pushState(State* state);
        void popAllStates();

        OIS::Mouse* getMouse();
        OIS::Keyboard* getKeyboard();

        // Eventos de entrada
        bool keyPressed(const OIS::KeyEvent &arg);
        bool keyReleased(const OIS::KeyEvent &arg);
        bool mouseMoved(const OIS::MouseEvent &arg);
        mousePressed(const OIS::MouseEvent &arg, OIS::MouseButtonID id);
        bool mouseReleased(const OIS::MouseEvent &arg, OIS::MouseButtonID
            id);

        // Control del proceso de renderizado
        bool frameStarted(const Ogre::FrameEvent& event);
        bool frameEnded(const Ogre::FrameEvent& event);
        bool frameRenderingQueued(const Ogre::FrameEvent& event);

        // Eventos de ventana
        void windowClosed(Ogre::RenderWindow* window) {};
        void windowResized(Ogre::RenderWindow* window) {};
        bool windowClosing(Ogre::RenderWindow* window);

    private:
        // Juego
        Game* _game;
```

```
42
43          // Objetos Ogre
44          Ogre::Log* _log;
45          Ogre::RenderWindow* _window;
46
47          // Managers de entrada OIS
48          OIS::InputManager* _inputManager;
49          OIS::Keyboard* _keyboard;
50          OIS::Mouse* _mouse;
51
52          // Fichero de recursos
53          Ogre::String _resourcesCfg;
54
55          // Metodos auxiliares
56          void configureOIS();
57
58          // Vector de estados
59          std::vector<State*> _states;
60
61          // Iterador inverso
62          std::vector<State*>::const_reverse_iterator rev_it;
63
64          // Numero de estados (temporal)
65          int _numStates;
66
67          // Tipo de operacion (push o pop)
68          enum tOperation {opPush, opPop};
69
70          // Operacion de pop o push sobre un estado que se ejecutara
71          // al comienzo de update
72          struct StateOperation {
73              tOperation _type;
74              State* _state;
75
76              StateOperation(tOperation type, State* state = 0): _type(type)
77                                                                 _state(state
                                                                       ) {};
78          };
79
80          // Cola de operaciones por realizar
81          std::queue<StateOperation> _pendingOperations;
82
83          // Controlamos la salida
84          bool _exit;
85
86          // Realiza las operaciones pendientes
87          void performOperations();
88  };
```

4.4.2. Internacionalización mediante gettext

Sion Tower está completamente internacionalizado y se distribuye tanto en inglés como en castellano aunque no es complicado añadir idiomas adicionales. Se ha seguido el sistema propuesto por la biblioteca de localización libre GETTEXT [18], que es prácticamente un estándar en la materia. Para comprender

el uso de la biblioteca así como la generación y mantenimiento de traducciones se recurrió a la publicación de José Tomás Tocino García *Traducción de proyectos con GNU gettext en 15 minutos* [22].

Ficheros necesarios

GETTEXT funciona de una manera muy sencilla. Trabaja a modo de diccionario clave valor en el que las claves son las cadenas a traducir en un idioma base (normalmente el inglés) y los valores son el texto traducido al idioma destino. Estas duplas claves valor se especifican en ficheros de texto plano de extensión `.po`. No obstante, GETTEXT utiliza internamente una versión binaria de dichos ficheros de extensión `.mo`. El proceso de traducción consta de los siguientes pasos:

1. Creación de la jerarquía de directorios para las traducciones.
2. Obtención de todas las cadenas a partir del código fuente y agrupación de las mismas en un fichero maestro `.pot`.
3. Creación de un fichero `.po` a partir del maestro `.pot` por cada idioma. Para ello empleamos el comando `msginit`.
4. Traducción de las cadenas del fichero `.po`.
5. Binarización del fichero `.po` en uno `.mo`. Utilizaremos el comando `msgfmt`.

La jerarquía de directorios resultante en **Sion Tower** es la siguiente:

```
|-- [siontower]
|     |-- lang
|     |    |-- en
|     |    |    `-- LC_MESSAGES
|     |    |         `-- siontower.mo
|     |    `-- es
|     |         `-- LC_MESSAGES
|     |              `-- siontower.mo
|     |-- po
|     |    |-- en.po
|     |    |-- es.po
|     |    `-- siontower.pot
|
```

En el siguiente fragmento de código se puede apreciar un ejemplo de fichero *.po*:

```
#: src/stateProfile.cpp:202
msgid "#ff0000Error: the profile already exists"
msgstr "#ff0000Error: el perfil ya existe"

#: src/stateProfile.cpp:195
msgid "#ff0000Error: you must enter a name"
msgstr "#ff0000Error: debes introducir un nombre"

#: src/stateLevel.cpp:269
```

```
msgid "#ff0000Locked level"
msgstr "#ff0000Nivel bloqueado"

...
```

Traducción de mensajes en el código

Para traducir mensajes dentro del código hemos de incluir los ficheros de cabecera <*locale.h*> y <*libintl.h*>. Al comienzo del programa hemos de indicar al sistema qué codificación de caracteres emplearemos, qué paquetes de traducciones utilizaremos (en nuestro caso sería *siontower*) y dónde se encontrarán las mismas (para nosotros sería el directorio `lang`). Bastan las siguientes líneas:

```
1 bind_textdomain_codeset("siontower", "UTF-8");
2 setlocale(LC_MESSAGES, "");
3 bindtextdomain("siontower", "lang" );
4 textdomain("siontower");
```

En cada módulo que deseemos traducir tendremos que incluir los mismos ficheros de cabecera. En cada cadena a localizar habrá que colocar una llamada a la función `gettext()` con la clave en el idioma base para que nos devuelva la cadena en el idioma del sistema que esté ejecutando el software. Lo normal es utilizar la directiva del preprocesador `#define _ gettext` para ahorrar espacio y limpiar el código. De esta manera lo que antes era:

```
1 _lblState->setCaption("You have failed! Press space to try again!");
```

Ahora se convierte en:

```
1 _lblState->setCaption(_("You have failed! Press space to try again!"));
```

Traducción de plantillas de MyGUI

Si ejecutamos **Sion Tower** en un sistema cuyo idioma sea el inglés veremos *You have failed! Press space to try again!*. En cambio, si lo hacemos en uno cuyo idioma sea el castellano podremos leer *¡Has fallado! Pulsa espacio para intentarlo de nuevo*. Esto funciona dentro de código *C++* en el juego pero no con las plantillas de la interfaz. MyGUI utiliza ficheros XML de extensión `.layout` para definir los widgets que tiene cada pantalla de menú. En el mismo fichero XML aparecen los textos de botones, etiquetas y otros elementos que no se mostrarán traducidos en la interfaz. Para solucionarlo hay que llevar a cabo dos tareas:

1. Extraer todas las cadenas traducibles a un fichero `.pot`.
2. En tiempo de ejecución indicarle a GETTEXT que traduzca todas las cadenas de los elementos de la interfaz.

GETTEXT es capaz de extraer las cadenas traducibles de forma automática a partir de código *C++* o *Python*, no obstante, desconoce el formato xml con la sintaxis de MyGUI por lo que no puede llevar a cabo dicha extracción. Para subsanar este problema, se ha implementado un pequeño script en *Python* que escanea un directorio en busca de ficheros `.layout`, encuentra las cadenas traducibles y las una

en una plantilla `.pot`. Su sintaxis es la siguiente:

```
python translateLayout.py layoutsDir file.pot
```

El código completo del script es el siguiente:

```
#!/usr/bin/env python
# -*- coding: utf-8  -*-

################################################################################
# This file is part of Sion Tower.                                             #
#                                                                              #
# This program is free software: you can redistribute it and/or modify         #
# it under the terms of the GNU General Public License as published by         #
# the Free Software Foundation, either version 3 of the License, or            #
# any later version.                                                           #
#                                                                              #
# This program is distributed in the hope that it will be useful,              #
# but WITHOUT ANY WARRANTY; without even the implied warranty of               #
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the                #
# GNU General Public License for more details.                                 #
#                                                                              #
# You should have received a copy of the GNU General Public License            #
# along with this program.  If not, see <http://www.gnu.org/licenses/>.#
#                                                                              #
# Copyright (C) 2011, David Saltares Marquez, david.saltares@gmail.com #
################################################################################

import os
import sys

from xml.dom import minidom
from xml.etree.ElementTree import ElementTree

def syntax():
    print 'Syntax error, the correct syntax is:'
    print '    python translateLayout.py layoutsDir file.pot'
    print ''

def translateLayout(layoutFile, potFile, codes):
    # Load XML Document
    print '    * Loading XML document...'
    doc = ElementTree()

    #try:
    doc.parse(layoutFile)
    #except IOError:
    #    print '    * ERROR parsing MyGUI layout'
    #    exit(1)

    root = doc.getroot()

    # Retrieve every "Property" elements
    print '    * Procesing properties...'
```

```
49      properties = root.findall('.//Widget/Property')
50
51      # Open pot file
52      try:
53          f = open(potFile, 'w')
54      except IOError:
55          print '    * ERROR opening pot file'
56          exit(1)
57
58      # Iterate through every Property
59      for p in properties:
60          # Check if it has a Caption property
61          key = p.get('key')
62          if key == 'Caption':
63              # Get Caption
64              value = p.get('value')
65              if not value.isdigit() and not value == '' and not value in
                    codes:
66                  print '        - Caption found: ' + value
67
68                  # Write caption into pot file
69                  f.write('msgid "' + value.encode('utf-8') + '"\n')
70                  f.write('msgstr ""\n\n')
71                  codes.append(value)
72
73
74      # Save file
75      print '    * Saving .pot file...'
76      f.close()
77
78
79  def main():
80      # Header
81      print ''
82      print 'MyGUI Layout - POT converter'
83      print '============================'
84      print ''
85
86      # Check arguments
87      if len(sys.argv) < 3:
88          syntax()
89          sys.exit(1)
90
91      # Iterate through layout dirs converting every .layout file
92      dirName = sys.argv[1]
93      potFiles = []
94      codes = []
95
96      for f in os.listdir(dirName):
97          if os.path.isfile(os.path.join(dirName, f)) and f.endswith('.layout
                ''):
98              baseName = f[:-7]
99              potFile = baseName + '.pot'
100             print '* Converting ' + f + '...'
101             translateLayout(os.path.join(dirName, f),
102                             os.path.join(dirName, potFile), codes)
```

```
103                 potFiles.append(os.path.join(dirName, potFile))
104
105         # Merge all pot files together
106         print '* Merging pot files together...'
107         command = 'cat ' + " ".join(potFiles) + ' > ' + sys.argv[2]
108         os.system(command)
109
110         # Delete all temporary pot files
111         print '* Deleting temporary POT files...'
112         #os.system('rm ' + " ".join(potFiles))
113
114         print ''
115
116
117 if __name__ == "__main__":
118             main()
```

Para solucionar el punto 2 tenemos que recordar que cada estado de juego (*State*) contaba con un método *translate()* cuyo objetivo era traducir todos los elementos de la interfaz. Las claves de las cadenas que buscamos son las propias cadenas en el idioma base por lo que la traducción resulta trivial (pero necesaria). Un ejemplo de método de traducción podría ser el siguiente:

```
1 void StateMenu::translate() {
2     _btnPlay->setCaption(_(_btnPlay->getCaption().asUTF8_c_str()));
3     _btnCredits->setCaption(_(_btnCredits->getCaption().asUTF8_c_str()));
4     _btnExit->setCaption(_(_btnExit->getCaption().asUTF8_c_str()));
5     _lblSubtitle->setCaption(_(_lblSubtitle->getCaption().asUTF8_c_str()));
6 }
```

4.4.3. Sistema de audio

OGRE3D es simplemente un motor de renderizado y carece de subsistema de audio, de detección de colisiones, gestión de entrada o juego en red. Que en **Sion Tower** se reprodujesen efectos de sonido y música de fondo era imprescindible para la inmersión y para ofrecer información complementaria a la visual de cara al jugador sobre lo que ocurre en el mundo que simulamos. Para resolver este problema se ha decidido emplear la biblioteca libre *Simple DirectMedia Layer* (LIBSDL) y su extensión relacionada con el audio LIBSDL MIXER [34].

LIBSDL es una biblioteca compatible con el lenguaje *C* y, por tanto, carece de orientación a objetos. Parte del valor del trabajo realizado para **Sion Tower** ha sido abstraer el bajo nivel del subsistema de audio de la biblioteca en un sistema orientado a objetos.

Gestión de recursos en Ogre3D

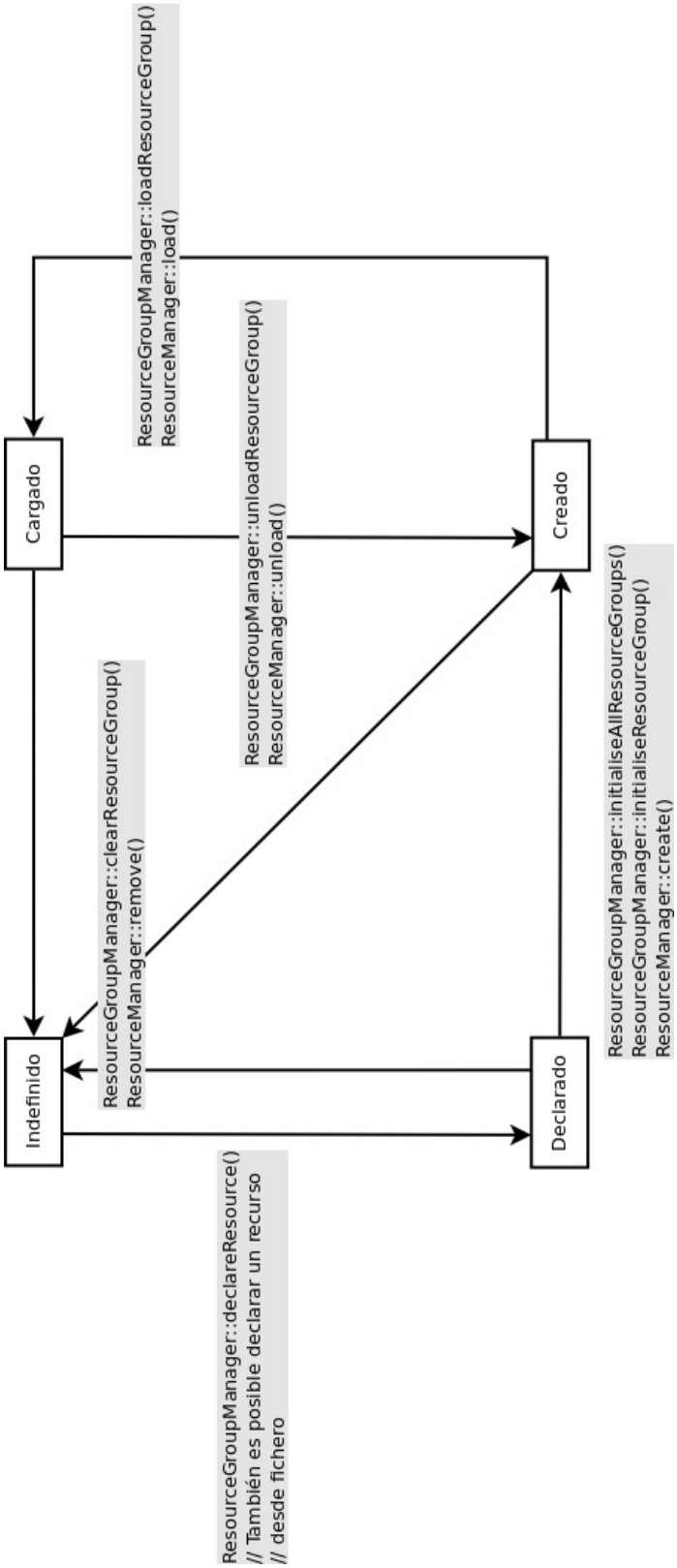


Figura 4.36: Ciclo de vida de los recursos en Ogre3D

OGRE3D cuenta con un sistema de gestión de recursos muy completo para gestionar el ciclo de vida de cada recurso (ver figura 4.36) y así optimizar el consumo de memoria del juego. El concepto recurso es genérico, podemos referirnos a una malla tridimensional, a un script de postprocesado, un conjunto de animaciones, una textura o un recurso definido por nosotros. El carácter configurable y extensible de OGRE3D permite al usuario de la biblioteca crear y gestionar nuevos tipos de recursos de forma idéntica a como se hace con los propios del sistema. En nuestro caso definiremos los recursos *Song* (pista de música) y *SoundFX* (efecto de sonido). Esto nos reportará las siguientes ventajas:

- Cada recurso sólo se instanciará una vez en memoria aunque se utilice por varias entidades con el consiguiente ahorro.
- Sencillo acceso al recurso sin importar su ruta dentro del sistema de ficheros.
- Gestión de su ciclo de vida de cara a optimizar el consumo de memoria y los tiempos de carga.

Extensión de la gestión de recursos de Ogre3D

Por cada recurso nuevo que deseemos integrar en OGRE3D tendremos que crear una clase hija de *Ogre::Resource*, una derivada de la clase paramétrica *Ogre::Shared_ptr* (para asegurarnos instancia única del recurso) y otra descendiente de *Ogre::ResourceManager* que gestionará la carga y destrucción de recursos del mismo tipo. Cada una de estas clases deberá implementar varios métodos de forma obligatoria. El esquema general se muestra en la figura 4.37. Para conocer detalles adicionales a los expuestos en esta sección, puede consultarse el artículo *Extender la gestión de recursos, audio* en **IberOgre** [8].

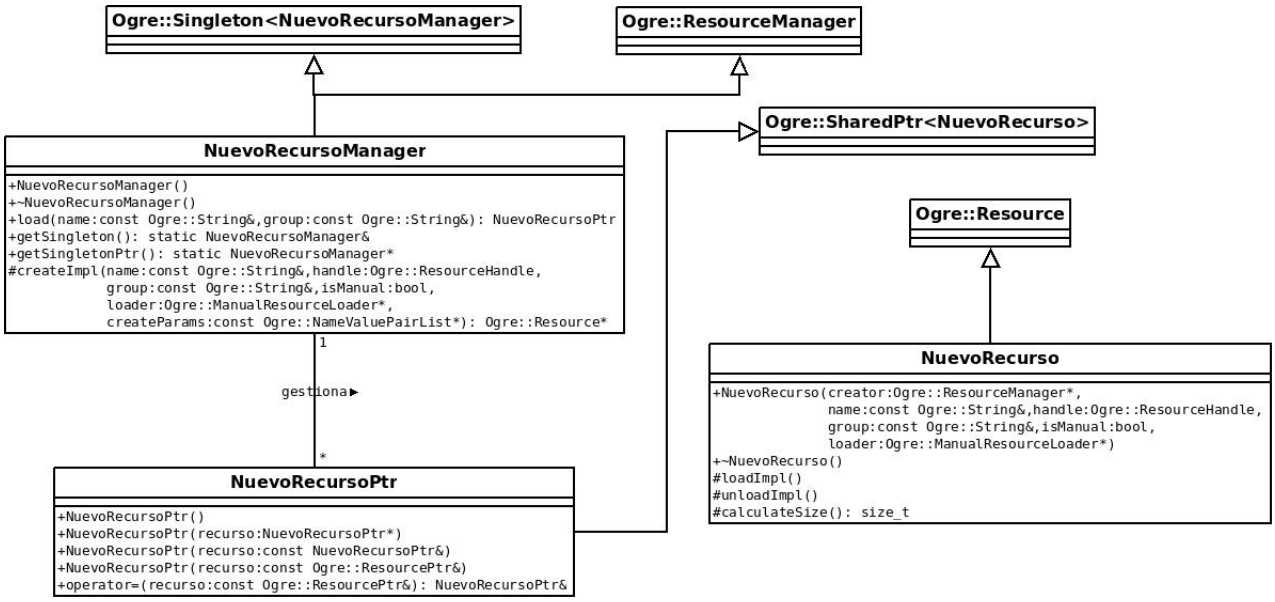


Figura 4.37: Esquema para extender la gestión de recursos en Ogre3D

Song, SongPtr y SongManager

La clase *Song* hereda de *Ogre::Resource* y se encarga de reproducir pistas de música en formato *OGG*. En su constructor es necesario indicarle el gestor de recursos que la controla su nombre y grupo al que pertenece. Donde realmente se carga el recurso y se libera son en los métodos privados *loadImpl()* y *unloadImpl()* llamados por el gestor de recursos. Así mismo, cuenta con métodos para la reproducción,

pausa, fundido de entrada o de salida. Debe proporcionar un método *calculateSize()* porque OGRE3D debe conocer en todo momento cuánto ocupan sus recursos en memoria. Incluso podríamos asignarle un presupuesto en memoria RAM a cualquiera de los gestores de recursos. A continuación se muestra la definición de la clase.

```
class Song: public Ogre::Resource {
    public:

        Song(Ogre::ResourceManager* creator,
             const Ogre::String& name,
             Ogre::ResourceHandle handle,
             const Ogre::String& group,
             bool isManual = false,
             Ogre::ManualResourceLoader* loader = 0);

        ~Song();


        void play(int loop = -1);
        void pause();
        void stop();
        void fadeIn(int ms, int loop = -1);
        void fadeOut(int ms);

        static bool isPlaying();
    protected:
        void loadImpl();
        void unloadImpl();

        size_t calculateSize() const;

    private:
        Mix_Music* _song;
        Ogre::String _path;
        size_t _size;
};
```

SongPtr es una clase sencilla que se limita a heredar de *Ogre::Shared_ptr* y permite asegurarnos de que sólo habrá una instancia de cada canción en todo el sistema. Funciona de manera similar a los *shared_ptr* de la biblioteca BOOST [5].

```
class SongPtr: public Ogre::SharedPtr<Song> {
    public:
        SongPtr(): Ogre::SharedPtr<Song>() {}
        explicit SongPtr(Song* m): Ogre::SharedPtr<Song>(m) {}
        SongPtr(const SongPtr &m): Ogre::SharedPtr<Song>(m) {}
        SongPtr(const Ogre::ResourcePtr &r);
        SongPtr& operator= (const Ogre::ResourcePtr& r);
};
```

La clase *SongManager* es el gestor de recursos que se encarga de manejar pistas de audio (*Song*). Sigue el patrón de diseño *Singleton* (una sóla instancia accesible desde todo el sistema) [15] y cuenta con los

clásicos métodos *getSingleton()* y *getSingletonPtr()*. Para cargar un recurso utilizaremos *load* el cual busca el recurso y, en el caso de no existir, lo crea con su método privado *createImpl()*.

```
class SongManager: public Ogre::ResourceManager,
                   public Ogre::Singleton<SongManager> {
    public:
        SongManager();
        virtual ~SongManager();

        virtual SongPtr load(const Ogre::String& name,
                             const Ogre::String& group = Ogre::
                                ResourceGroupManager::
                                DEFAULT_RESOURCE_GROUP_NAME);

        static SongManager& getSingleton();
        static SongManager* getSingletonPtr();

    protected:
        Ogre::Resource* createImpl(const Ogre::String& name,
                                   Ogre::ResourceHandle handle,
                                   const Ogre::String& group,
                                   bool isManual,
                                   Ogre::ManualResourceLoader* loader,
                                   const Ogre::NameValuePairList*
                                      createParams);
};
```

SoundFX, SoundFXPtr y SoundFXManager

Las clases *SoundFX*, *SoundFXPtr* y *SoundFXManager* funcionan de forma prácticamente idéntica a como lo hacen sus semejantes de música.

Definición de la clase *SoundFX*:

```
class SoundFX: public Ogre::Resource {
    public:
        SoundFX(Ogre::ResourceManager* creator,
               const Ogre::String& name,
               Ogre::ResourceHandle handle,
               const Ogre::String& group,
               bool isManual = false,
               Ogre::ManualResourceLoader* loader = 0);


        ~SoundFX();


        int play(int loop = 0);

    protected:
        void loadImpl();
        void unloadImpl();
        size_t calculateSize() const;
```

```
20
21      private:
22          Mix_Chunk* _sound;
23          Ogre::String _path;
24          size_t _size;
25  };
```

Definición de la clase *SoundFXPtr*:

```
1   class SoundFXPtr: public Ogre::SharedPtr<SoundFX> {
2       public:
3           SoundFXPtr(): Ogre::SharedPtr<SoundFX>() {}
4           explicit SoundFXPtr(SoundFX* s): Ogre::SharedPtr<SoundFX>(s) {}
5           SoundFXPtr(const SoundFXPtr& s): Ogre::SharedPtr<SoundFX>(s) {}
6           SoundFXPtr(const Ogre::ResourcePtr& r);
7           SoundFXPtr& operator= (const Ogre::ResourcePtr& r);
8   };
```

Definición de la clase *SoundFXManager*:

```
1   class SoundFXManager: public Ogre::ResourceManager,
2                         public Ogre::Singleton<SoundFXManager> {
3       public:
4           SoundFXManager();
5           virtual ~SoundFXManager();
6
7           virtual SoundFXPtr load(const Ogre::String& name,
8                                   const Ogre::String& group = Ogre::
                                       ResourceGroupManager::
                                       DEFAULT_RESOURCE_GROUP_NAME);
9
10          static SoundFXManager& getSingleton();
11          static SoundFXManager* getSingletonPtr();
12
13          static int getAvailableChannels();
14      protected:
15          Ogre::Resource* createImpl(const Ogre::String& name,
16                                     Ogre::ResourceHandle handle,
17                                     const Ogre::String& group,
18                                     bool isManual,
19                                     Ogre::ManualResourceLoader* loader,
20                                     const Ogre::NameValuePairList*
                                           createParams);
21
22      private:
23          static int numChannels;
24  };
```

Para detalles adicionales sobre la implementación del subsistema, lo ideal es acudir al propio código fuente en la forja de RedIRIS.

Ejemplo de uso

A continuación se adjunta un pequeño ejemplo de uso del sistema de audio desarrollado para **Sion Tower**.

```
// Durante el inicio de la aplicacion

// Creamos el ResourceManager
SongManager* songManager = new SongManager();
SoundFXManager* soundFXManager = new SoundFXManager();

...

// Cargamos los recursos
SongPtr levelMusic = songManager->load("musicaNivel1.ogg", "Nivel1");
levelMusic->play();

SoundFXPtr explosion = soundFXManager->load("explosion.wav", "Nivel1");
explosion->play();

...

// Durante el cierre de la aplicacion

// Destruimos el ResourceManager
delete songManager;
delete soundFXManager;
```

Como ya se mencionó en los objetivos del proyecto (sección 1.4 en la página 12), se pretende que los subsistemas desarrollados para **Sion Tower** sean reutilizables en alta medida. Por ello, se ha publicado el sistema de audio por separado como paquete descargable en la forja de RedIRIS. Viene acompañado de la documentación generada con *Doxygen*, de las instrucciones de integración y de su licencia *GPL v3*. Puede obtenerse en la siguiente dirección.

http://forja.rediris.es/frs/download.php/2075/siontower-3dsound-v0.1.tar.gz

4.4.4. Detección de colisiones

En **Sion Tower** es necesario detectar colisiones entre muchos de los elementos del juego y OGRE3D no proporciona un subsistema que nos ayude a llevar a cabo esta tarea. Era posible acudir a alternativas como los motores de físicas BULLET y ODE pero eran demasiado complejos para el reducido número de requisitos que teníamos. En **Sion Tower** había que detectar colisiones entre los siguientes elementos.

- personaje-enemigo
- personaje-escenario
- enemigo-hechizo
- hechizo-escenario

El área de colisión de los personajes y escenario vendría limitado por varias formas geométricas agrupadas. Entre estas formas podrían encontrarse: cajas, esferas y planos. Las funcionalidades que requiere

el sistema de colisiones son:

- Soporte para varias formas (clase *Shape*) como esferas, planos, AABB y OBB.
- Tests de colisión para varias combinaciones de estas formas.
- Cuerpos (clase *Body*) compuestos por varias formas colisionables de manera que se ajusten a contornos complejos.
- Gestor de colisiones que mantenga el control de los cuerpos existentes y pueda detectar colisiones entre ellos.
- Detección de colisiones eficiente.
- Filtrado de colisiones por tipo de cuerpo. Sólo nos interesan las especificadas en la lista anterior.
- Registro de funciones a disparar ante colisiones entre dos cuerpos de un tipo de terminado, también conocidas como *callbacks*.

Diseño general

Como puede verse en los diagramas de la fase de diseño en la sección 4.3 en la página 79, el sistema de detección de colisiones está formado por:

Shape clase virtual que modela una forma geométrica en el espacio de forma genérica. De ella descienden *Plane*, *Sphere*, *AxisAlignedBox* y *OrientedBox*.

Body modela un cuerpo colisionable formado por varias formas geométricas.

GameObject agrupa un cuerpo colisionable (*Body*) y un nodo de la escena 3D (*Ogre::SceneNode*). De esta forma encapsulamos la faceta colisionable y visual de los objetos de juego. Posteriormente, los elementos que cuenten con una malla tridimensional pueden heredar de esta clase y añadir un *Ogre::Entity* como ocurre en la clase *GameMesh*. En cambio, si lo que desean es mostrar un sistema de partículas con modelo de colisión pueden heredar y añadir un *Ogre::ParticleSystem* como ocurre en *Spell*.

CollisionManager controla todos los cuerpos colisionables (*Body*) y detecta colisiones entre ellos en cada iteración del bucle de juego filtrando los tipos de colisión que carecen de callback.

La clase Shape y RTTI

La clase *Shape* cuenta con un método estático *getCollision()* que recibe punteros a formas genéricas y devuelve verdadero o falso en función de si se produce intersección entre las mismas o no. De forma interna cuenta con métodos privados para detectar colisiones entre formas concretas, por ejemplo el test para planos y cajas alineadas sería *getCollisionPlaneAABB()*.

La tarea que consiste en dados dos punteros a formas genéricas, elegir el test de colisión adecuado se llama *Collision dispatching*. Podríamos emplear muchos bloques `if` y llamadas a `dynamic_cast` [27] pero el resultado sería de lo más ineficiente y el tiempo es un recurso muy preciado en la detección de colisiones. Necesitamos una forma de hacer detección de tipos en tiempo de ejecución (*RTTI* o Real Time Type Identification) de forma eficiente y segura.

La solución por la que se ha optado consiste en incluir un método virtual puro en *Shape* llamado *getType())* que obligue a las clases descendientes a implementarlo y devuelva su tipo concreto del enumerado

Shape::Type. La clase *Shape* mantiene un conjunto desordenado (*boost::unordered_map*) [5] que dados dos tipos de formas nos devuelve el test de colisión adecuado en un objeto función *boost::function*. Buscar en estos conjuntos es mucho más rápido que utilizar el clásico *std::map* de la biblioteca estándar de plantillas (STL).

Los tests de colisión entre formas concretas pueden hacer un *static_cast* para obtener un puntero a la clase hija en lugar de a la genérica. La conversión de tipos estáticas es mucho rápida aunque peligrosa. No obstante, ya hemos determinado el tipo con seguridad plena gracias a *getType()*. Al inicio de la aplicación es necesario llamar a *configureCollisionDispathing()* para inicializar la tabla estática de tests de colisión. Si creamos formas nuevas y deseamos añadir nuevos tests, es posible hacerlo mediante el método *addCollisionTest()*. A continuación, se adjunta la definición completa de la clase *Shape*.

```
1  class Shape {
2      public:
3          enum Type {SPHERE = 1, AABB = 2, PLANE = 3, OBB = 4};
4          typedef boost::function<bool(Shape*, Shape*)>
               CollisionCheckFunction;
5
6          Shape(const Ogre::String& name = "");
7          virtual ~Shape();
8
9          const Ogre::String& getName() const;
10
11         virtual int getType() const = 0;
12
13         virtual void applyTransform(Shape* localShape,
14                                     const Ogre::Vector3& traslation = Ogre
                                        ::Vector3::ZERO,
15                                     const Ogre::Vector3& scale = Ogre::
                                        Vector3::UNIT_SCALE,
16                                     const Ogre::Quaternion& orientation =
                                        Ogre::Quaternion::IDENTITY) = 0;
17
18         virtual Shape* getTransformedCopy(const Ogre::Vector3& traslation =
               Ogre::Vector3::ZERO,
19                                           const Ogre::Vector3& scale = Ogre
                                              ::Vector3::UNIT_SCALE,
20                                           const Ogre::Quaternion&
                                              orientation = Ogre::Quaternion
                                              ::IDENTITY) = 0;
21
22         static void configureCollisionDispatching();
23
24         static void addCollisionTest(CollisionCheckFunction test, int typeA
               , int typeB);
25
26         static bool getCollision(Shape* shapeA, Shape* shapeB);
27
28     protected:
29         Ogre::String _name;
30
31     private:
32         typedef boost::unordered_map<int, boost::unordered_map<int,
```

```
            CollisionCheckFunction> > CollisionDispatchTable;

        static CollisionDispatchTable  _collisionDispatcher;

        static bool getCollisionSphereSphere(Shape* shapeA, Shape* shapeB);
        static bool getCollisionAABBAABB(Shape* shapeA, Shape* shapeB);
        static bool getCollisionPlanePlane(Shape* shapeA, Shape* shapeB);
        static bool getCollisionOBBOBB(Shape* shapeA, Shape* shapeB);
        static bool getCollisionSphereAABB(Shape* shapeA, Shape* shapeB);
        static bool getCollisionSphereOBB(Shape* shapeA, Shape* shapeB);
        static bool getCollisionPlaneSphere(Shape* shapeA, Shape* shapeB);
        static bool getCollisionPlaneAABB(Shape* shapeA, Shape* shapeB);
        static bool getCollisionOBBPlane(Shape* shapeA, Shape* shapeB);
        static bool getCollisionOBBAABB(Shape* shapeA, Shape* shapeB);
};
```

En el siguiente fragmento, inicializamos la tabla de tests de colisión:

```
void Shape::configureCollisionDispatching() {
    // Completamos la tabla de chequeos de colision
    _collisionDispatcher[SPHERE][SPHERE] = boost::bind(&Shape::
        getCollisionSphereSphere, _1, _2);
    _collisionDispatcher[AABB][AABB] = boost::bind(&Shape::
        getCollisionAABBAABB, _1, _2);
    _collisionDispatcher[PLANE][PLANE] = boost::bind(&Shape::
        getCollisionPlanePlane, _1, _2);
    _collisionDispatcher[OBB][OBB] = boost::bind(&Shape::getCollisionOBBOBB
        , _1, _2);

    _collisionDispatcher[AABB][SPHERE] = boost::bind(&Shape::
        getCollisionSphereAABB, _1, _2);
    _collisionDispatcher[AABB][PLANE] = boost::bind(&Shape::
        getCollisionPlaneAABB, _1, _2);
    _collisionDispatcher[AABB][OBB] = boost::bind(&Shape::
        getCollisionOBBAABB, _1, _2);

    _collisionDispatcher[SPHERE][AABB] = boost::bind(&Shape::
        getCollisionSphereAABB, _1, _2);
    _collisionDispatcher[SPHERE][PLANE] = boost::bind(&Shape::
        getCollisionPlaneSphere, _1, _2);
    _collisionDispatcher[SPHERE][OBB] = boost::bind(&Shape::
        getCollisionSphereOBB, _1, _2);

    _collisionDispatcher[PLANE][AABB] = boost::bind(&Shape::
        getCollisionPlaneAABB, _1, _2);
    _collisionDispatcher[PLANE][SPHERE] = boost::bind(&Shape::
        getCollisionPlaneSphere, _1, _2);
    _collisionDispatcher[PLANE][OBB] = boost::bind(&Shape::
        getCollisionOBBPlane, _1, _2);

    _collisionDispatcher[OBB][SPHERE] = boost::bind(&Shape::
        getCollisionSphereOBB, _1, _2);
    _collisionDispatcher[OBB][PLANE] = boost::bind(&Shape::
        getCollisionOBBPlane, _1, _2);
```

```
22      _collisionDispatcher[OBB][AABB] = boost::bind(&Shape::
            getCollisionOBBAABB, _1, _2);
23  }
```

Cuando alguien desea saber si dos formas colisionan llama a *getCollision()* con dos punteros a formas genéricas. En dicho método es cuando se busca en la tabla de tests de colisión.

```
1   bool Shape::getCollision(Shape* shapeA, Shape* shapeB) {
2       // Comprobamos si la forma A esta registrada
3       CollisionDispatchTable::iterator itA;
4       itA = _collisionDispatcher.find(shapeA->getType());
5
6       if (itA == _collisionDispatcher.end()) {
7           cout << "Shape::getCollision(): no existe el tipo " << shapeA->
                getType() << endl;
8           return 0;
9       }
10
11      // Comprobamos que la forma B esta registrada
12      CollisionDispatchTable::iterator itB;
13      itB = _collisionDispatcher.find(shapeB->getType());
14
15      if (itB == _collisionDispatcher.end()) {
16          cout << "Shape::getCollision(): no existe el tipo " << shapeB->
                getType() << endl;
17          return 0;
18      }
19
20      // Comprobamos que hay un metodo de comprobacion del tipo A - B
21      boost::unordered_map<int, CollisionCheckFunction>::iterator itC;
22      itC = _collisionDispatcher[shapeA->getType()].find(shapeB->getType());
23
24      if (itC == itA->second.end()) {
25          cout << "Shape::getCollision(): no existe un metodo de comprobacion
                entre" << shapeA->getType() << " y " << shapeB->getType() <<
                endl;
26          return 0;
27      }
28
29      // Llamamos al metodo de comprobacion
30      return itC->second(shapeA, shapeB);
31  }
```

Tests de colisión

En esta sección comentaremos con cierto nivel de detalle los tests de colisión para cada pareja de tipos de formas. Para implementar estos tests se ha recurrido a la publicación *Real Time Collision Detection* de Christer Ericson [13]. La explicación de cada test vendrá acompañada del fragmento de código correspondiente y de un diagrama explicativo. Por motivos de claridad, los diagramas se adjuntan en dos dimensiones pero los principios son fácilmente extensibles a las tres dimensiones.

La mayoría de los tests de colisiones se sustentan en el **Teorema del eje de separación** [56]. Asegura

que dados dos objetos convexos en un plano 2D existe una línea sobre la cual, las proyecciones de los dos objetos no se solapan si y sólo si los objetos son disjuntos (no tienen puntos en común). La línea se conoce como eje de separación. Si nos trasladamos a las tres dimensiones, la línea de separación se convierte en plano de separación. Podemos ver un ejemplo del teorema en la figura 4.38.

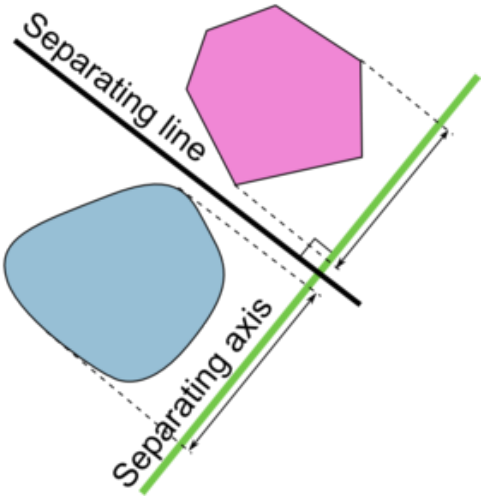


Figura 4.38: Teorema del eje de separación

El test entre una **Sphere** y otra **Sphere** es el más sencillo de todos. Basta con comprobar si la distancia entre los centros de ambas esferas es menor que la suma de sus radios, en tal caso existiría colisión, tal y como puede verse en la figura 4.39.

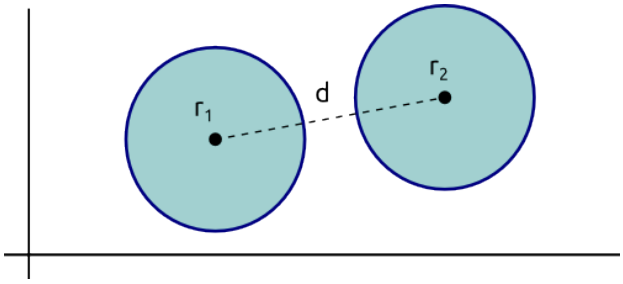


Figura 4.39: Test de colisión Sphere Sphere

Para calcular la distancia entre dos puntos es necesario una raíz cuadrada pero éstas son extremadamente caras en tiempo de procesamiento. Podemos comparar la distancia al cuadrado con el cuadrado de la suma de los radios, una expresión equivalente y de mayor eficiencia. A continuación adjuntamos el código del test.

```
1 bool Shape::getCollisionSphereSphere(Shape* shapeA, Shape* shapeB) {
2     // Hacemos la conversion (estamos seguros de que son esferas)
3     Sphere* sphereA = static_cast<Sphere*>(shapeA);
4     Sphere* sphereB = static_cast<Sphere*>(shapeB);
5 
6     // Hacemos el test
```

```
    Ogre::Vector3 s = sphereA->getCenter() - sphereB->getCenter();
    Ogre::Real totalRadius = sphereA->getRadius() + sphereB->getRadius();

    return (s.squaredLength() <= totalRadius * totalRadius);
}
```

En la intersección entre cajas alineadas (**AABB** y **AABB**) con los ejes emplearemos el teorema del plano de separación. Proyectamos las cajas sobre cada uno de los tres ejes y si algunas de las proyecciones no se solapan podremos asegurar que no existe colisión entre las AABB. La figura 4.40 ilustra el test.

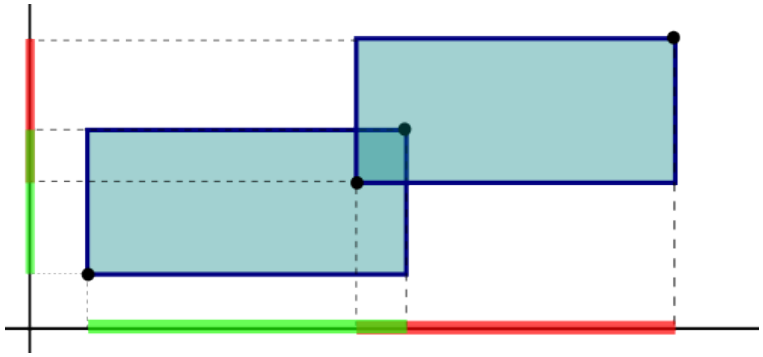


Figura 4.40: Test de colisión AABB AABB

```
bool Shape::getCollisionAABBAABB(Shape* shapeA, Shape* shapeB) {
    // Hacemos la conversion (estamos seguros de que son AABBs)
    AxisAlignedBox* aabb1= static_cast<AxisAlignedBox*>(shapeA);
    AxisAlignedBox* aabb2 = static_cast<AxisAlignedBox*>(shapeB);

    // Hacemos el test
    return (aabb1->getMaxPos().x > aabb2->getMinPos().x &&
            aabb1->getMinPos().x < aabb2->getMaxPos().x &&
            aabb1->getMaxPos().y > aabb2->getMinPos().y &&
            aabb1->getMinPos().y < aabb2->getMaxPos().y &&
            aabb1->getMaxPos().z > aabb2->getMinPos().z &&
            aabb1->getMinPos().z < aabb2->getMaxPos().z);
}
```

El test de colisión entre dos figuras de tipo **Plane** también es sencillo. Los planos son infinitos por lo que la única situación en la que dos planos no colisionan es cuando estos son paralelos y no están a la misma distancia del origen. La orientación de los planos está definida por su vector normal. Si las dos normales son paralelas y la distancia con respecto al origen no coincide podremos asegurar que los planos no colisionan. Dos vectores son paralelos si su producto escalar es igual a 1. Lo vemos ilustrado en la figura 4.41.

```
bool Shape::getCollisionPlanePlane(Shape* shapeA, Shape* shapeB) {
    // Hacemos la conversion (estamos seguros de que son Planes)
    Plane* planeA = static_cast<Plane*>(shapeA);
    Plane* planeB = static_cast<Plane*>(shapeB);

```

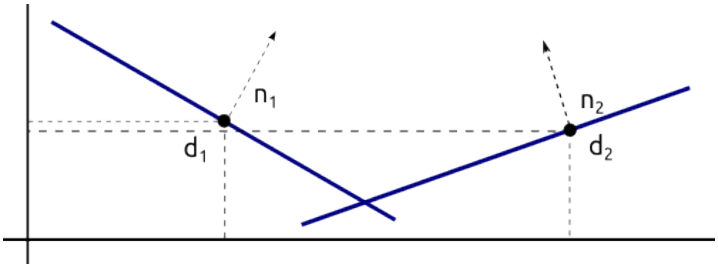


Figura 4.41: Test de colisión Plane Plane

```
6       // Hacemos el test
7       Ogre::Vector3 normalA = planeA->getNormal().normalisedCopy();
8       Ogre::Vector3 normalB = planeB->getNormal().normalisedCopy();
9       return (normalA.dotProduct(normalB) != 1 ||
10              planeA->getPosition() == planeB->getPosition());
11  }
```

En el test entre **Sphere** y **AxisAlignedBox** se pueden producir dos casos en los que existe intersección entre los objetos. El primero se da cuando el centro de la esfera está contenida en el AABB mientras que el segundo tiene lugar cuando el centro está fuera de la caja pero existe intersección (el diagrama 4.42 ilustra el segundo caso). En primer lugar comprobamos si el centro de la esfera está dentro de la caja. Posteriormente recorremos los vértices del AABB y elegimos el más cercano al centro de la esfera. Si la distancia entre ambos es menor que el radio de la esfera las dos formas colisionan.

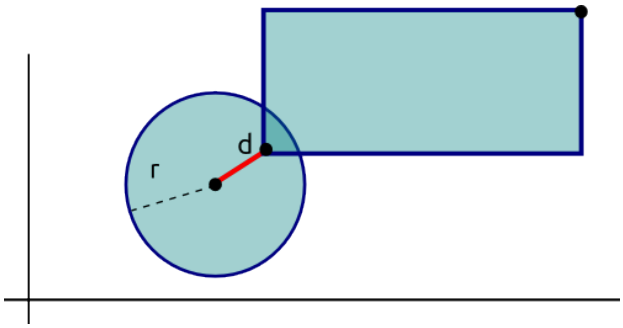


Figura 4.42: Test de colisión Sphere AABB

```
1   bool Shape::getCollisionSphereAABB(Shape* shapeA, Shape* shapeB) {
2       // Hacemos la conversion (estamos seguros de que A es Sphere y B es
            AABB)
3       Sphere* sphere;
4       AxisAlignedBox* aabb;
5       if (shapeA->getType() == SPHERE) {
6           sphere = static_cast<Sphere*>(shapeA);
7           aabb = static_cast<AxisAlignedBox*>(shapeB);
8       } else {
9           sphere = static_cast<Sphere*>(shapeB);
10          aabb = static_cast<AxisAlignedBox*>(shapeA);
11      }
12
```

```
    // Hacemos el test
    Ogre::Real s = 0;
    Ogre::Real d = 0;
    Ogre::Vector3 center = sphere->getCenter();
    Ogre::Vector3 minPos = aabb->getMinPos();
    Ogre::Vector3 maxPos = aabb->getMaxPos();

    // Comprobamos si el centro de la esfera esta dentro del AABB
    bool centerInsideAABB = (center.x <= maxPos.x &&
                             center.x >= minPos.x &&
                             center.y <= maxPos.y &&
                             center.y >= minPos.y &&
                             center.z <= maxPos.z &&
                             center.z >= minPos.z);

    if (centerInsideAABB)
        return true;

    // Comprobamos si la esfera y el AABB se intersectan
    for (int i = 0; i < 3; ++i) {
        if (sphere->getCenter()[i] < aabb->getMinPos()[i]) {
            s = sphere->getCenter()[i] - aabb->getMinPos()[i];
            d += s * s;
        } else if (sphere->getCenter()[i] > aabb->getMaxPos()[i]) {
            s = sphere->getCenter()[i] - aabb->getMaxPos()[i];
            d += s * s;
        }
    }

    return (d <= sphere->getRadius() * sphere->getRadius());
}
```

Comprobar si una **Sphere** colisiona con un **Plane** es tan sencillo como obtener la distancia entre ambos y compararla con el radio de la esfera como hemos hecho en otras ocasiones. La distancia entre el centro y el punto que conocemos del plano no es la distancia real entre ambas formas. Para calcular la distancia real tendremos que proyectar el vector $p - c$ (punto del plano - centro de la esfera) sobre la normal del plano. Sólo nos es necesario el cuadrado de la distancia y lo comprobaremos con el cuadrado del radio (para evitarnos utilizar una raíz cuadrada). Puede verse el test en la figura 4.43.

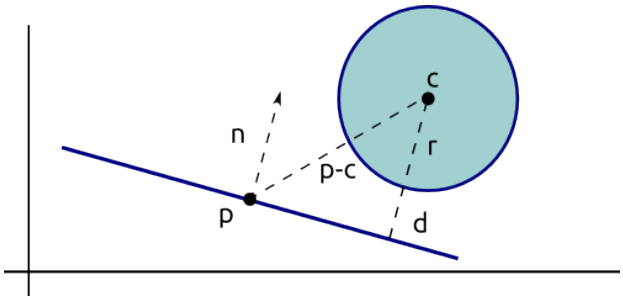


Figura 4.43: Test de colisión Sphere Plane

```
bool Shape::getCollisionPlaneSphere(Shape* shapeA, Shape* shapeB) {
```

```
    // Hacemos la conversion (estamos seguros de que A es Plane y B es
       Sphere)
    Plane* plane;
    Sphere* sphere;
    if (shapeA->getType() == PLANE) {
        plane = static_cast<Plane*>(shapeA);
        sphere = static_cast<Sphere*>(shapeB);
    } else {
        plane = static_cast<Plane*>(shapeB);
        sphere = static_cast<Sphere*>(shapeA);
    }

    // Hacemos el test

    // Distancia del centro de la esfera al plano
    Ogre::Vector3 v = sphere->getCenter() - plane->getPosition();
    Ogre::Vector3 n = plane->getNormal().normalisedCopy();
    Ogre::Real d = abs(n.dotProduct(v));

    // Si d <= radio, hay colision
    return d <= sphere->getRadius();
}
```

Para el test entre un **AxisAlignedBox** y un **Plane** calculamos el vértice más lejano y el más cercano al plano (*pmin* y *pmax* respectivamente). Si cada punto está a un lado distinto del plano podemos asegurar que ambas formas colisionan. El proceso se ilustra en la figura 4.44.

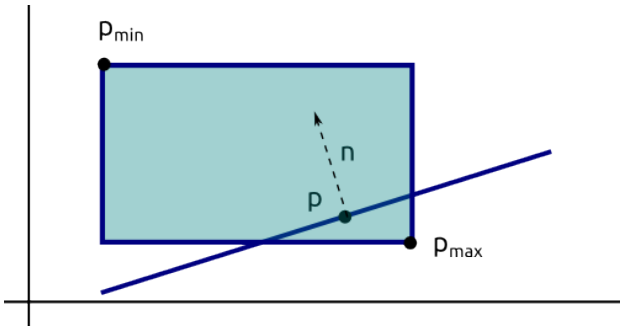


Figura 4.44: Test de colisión AABB Plane

```
bool Shape::getCollisionPlaneAABB(Shape* shapeA, Shape* shapeB) {
    // Hacemos la conversion (estamos seguros de que A es Plane y B es AABB
       )
    Plane* plane;
    AxisAlignedBox* aabb;
    if (shapeA->getType() == PLANE) {
        plane = static_cast<Plane*>(shapeA);
        aabb = static_cast<AxisAlignedBox*>(shapeB);
    } else {
        plane = static_cast<Plane*>(shapeB);
        aabb = static_cast<AxisAlignedBox*>(shapeA);
    }
```

```
12
13
14      // Hacemos el test
15      Ogre::Vector3 p;
16      Ogre::Vector3 n;
17
18      for (int i = 0; i < 3; ++i) {
19          if (plane->getNormal()[i] >= 0) {
20              p[i] = aabb->getMaxPos()[i];
21              n[i] = aabb->getMinPos()[i];
22          } else {
23              p[i] = aabb->getMaxPos()[i];
24              n[i] = aabb->getMinPos()[i];
25          }
26      }
27
28      // Si p esta en un lado diferente del plano que n, hay interseccion
29      Ogre::Real d1 = plane->getNormal().dotProduct(p - plane->getPosition())
            ;
30      Ogre::Real d2 = plane->getNormal().dotProduct(n - plane->getPosition())
            ;
31
32      return ((d1 <= 0 && d2 >= 0) || (d1 >= 0 && d2 <= 0));
33  }
```

Si deseamos conocer si un **OrientedBox** y una **Sphere** buscamos el punto más cercano de la caja a la esfera. Para hacerlo recorremos los vértices y vamos almacenando la distancia mínima. Una vez lo hayamos encontrado es sencillo ya que comparamos la distancia con el radio de la esfera. Como siempre, utilizamos distancias al cuadrado para evitar el uso de raíces en la medida de lo posible. El proceso aparece en la figura 4.45.

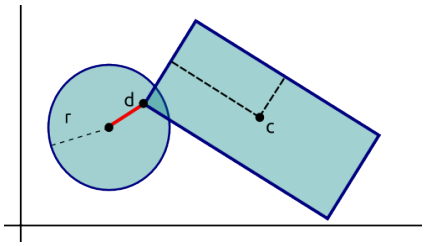


Figura 4.45: Test de colisión OBB Sphere

```
1   bool Shape::getCollisionSphereOBB(Shape* shapeA, Shape* shapeB) {
2       // Hacemos la conversion (estamos seguros de que uno es Sphere y otro
            OBB
3       Sphere* sphere;
4       OrientedBox* obb;
5       if (shapeA->getType() == OBB) {
6           obb = static_cast<OrientedBox*>(shapeA);
7           sphere = static_cast<Sphere*>(shapeB);
8       } else {
9           obb = static_cast<OrientedBox*>(shapeB);
10          sphere = static_cast<Sphere*>(shapeA);
```

```
    }

    Ogre::Vector3 closest = closestPointToOBB(sphere->getCenter(), obb);

    Ogre::Vector3 v = closest - sphere->getCenter();

    return v.dotProduct(v) <= sphere->getRadius() * sphere->getRadius();
}

static Ogre::Vector3 closestPointToOBB(const Ogre::Vector3& p, const
   OrientedBox* obb) {
    Ogre::Vector3 d = p - obb->getCenter();
    Ogre::Vector3 closest = obb->getCenter();
    Ogre::Matrix3 axes = obb->getAxes();

    for (int i = 0; i < 3; ++i) {
        Ogre::Real dist = d.dotProduct(Ogre::Vector3(axes[i][0], axes[i
           ][1], axes[i][2]));
        if (dist > obb->getExtent()[i])
            dist = obb->getExtent()[i];
        if (dist < -obb->getExtent()[i])
            dist = -obb->getExtent()[i];

        closest += dist * Ogre::Vector3(axes[i][0], axes[i][1], axes[i][2])
           ;
    }

    return closest;
}
```

Para el test de colisión entre un **OrientedBox** y un **Plane** proyectamos el primero sobre el segundo.

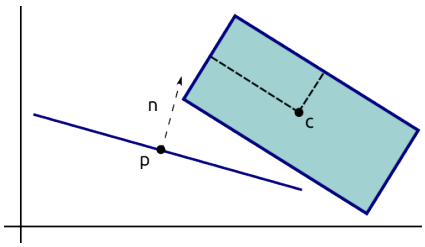


Figura 4.46: Test de colisión OBB Plane

```
bool Shape::getCollisionOBBPlane(Shape* shapeA, Shape* shapeB) {
    // Hacemos la conversion (estamos seguros de que uno es Plane y otro
       OBB
    Plane* plane;
    OrientedBox* obb;
    if (shapeA->getType() == OBB) {
        obb = static_cast<OrientedBox*>(shapeA);
        plane = static_cast<Plane*>(shapeB);
    } else {
        obb = static_cast<OrientedBox*>(shapeB);
```

```
10          plane = static_cast<Plane*>(shapeA);
11      }
12
13      Ogre::Matrix3 axes = obb->getAxes();
14      Ogre::Vector3 extent = obb->getExtent();
15      Ogre::Vector3 normal = plane->getNormal();
16
17      // Radio de la proyeccion de obb en el plano L(t) = obb.center + t *
            plane.normal
18      Ogre::Real r = extent[0] * std::abs(normal.dotProduct(Ogre::Vector3(
            axes[0][0], axes[0][1], axes[0][2]))) +
19                     extent[1] * std::abs(normal.dotProduct(Ogre::Vector3(
                           axes[1][0], axes[1][1], axes[1][2]))) +
20                     extent[2] * std::abs(normal.dotProduct(Ogre::Vector3(
                           axes[2][0], axes[2][1], axes[2][2])));
21
22      // Distancia del centro de la caja al plano
23      Ogre::Real s = normal.dotProduct(obb->getCenter() - plane->getPosition
            ());
24
25      return abs(s) <= r;
26  }
```

Detectar una colisión entre dos **OrientedBox** es el más complejo de todos. Los parámetros de las cajas de colisión están dispuestos en función de los ejes globales pero en este caso calculamos los parámetros de una de las cajas para que estén en función de los ejes definidos por la otra. Una vez hecho eso vamos tratando de trazar planos de separación entre ambas y buscando el descarte (método más rápido). Si no conseguimos encontrar el plano de separación, se habrá producido una colisión. Puede verse en el diagrama 4.47.

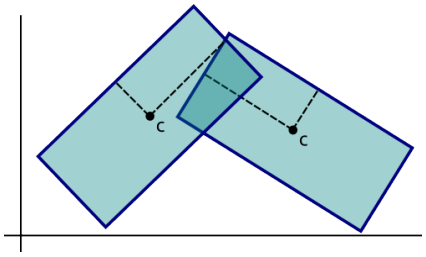


Figura 4.47: Test de colisión OBB OBB

```
1   bool Shape::getCollisionOBBOBB(Shape* shapeA, Shape* shapeB) {
2       // Hacemos la conversion, estamos seguros de que son OBB
3       OrientedBox* obbA = static_cast<OrientedBox*>(shapeA);
4       OrientedBox* obbB = static_cast<OrientedBox*>(shapeB);
5
6
7       // FUENTE: Real Time Collision Detection pag 101
8
9
10      // Obtenemos B en funcion de los ejes locales de A
11      Ogre::Real ra, rb;
```

```
12      Ogre::Matrix3 R, absR;
13      Ogre::Matrix3 axesA = obbA->getAxes();
14      Ogre::Matrix3 axesB = obbB->getAxes();
15
16      for (int i = 0; i < 3; ++i) {
17          for (int j = 0; j < 3; ++j) {
18              Ogre::Vector3 vA(axesA[i][0], axesA[i][1], axesA[i][2]);
19              Ogre::Vector3 vB(axesB[j][0], axesB[j][1], axesB[j][2]);
20              R[i][j] = vA.dotProduct(vB);
21          }
22      }
23
24      // Vector de translacion t en los ejes de A
25      Ogre::Vector3 t = obbB->getCenter() - obbA->getCenter();
26      t = Ogre::Vector3(t.dotProduct(Ogre::Vector3(axesA[0][0], axesA[0][1],
            axesA[0][2])),
27                        t.dotProduct(Ogre::Vector3(axesA[1][0], axesA[1][1],
                              axesA[1][2])),
28                        t.dotProduct(Ogre::Vector3(axesA[2][0], axesA[2][1],
                              axesA[2][2])));
29
30      for (int i = 0; i < 3; ++i)
31          for (int j = 0; j < 3; ++j)
32              absR[i][j] = std::abs(R[i][j]);
33
34      // Test ejes L = A0 L = A1 L = A2
35      for (int i = 0; i < 3; ++i) {
36          ra = obbA->getExtent()[i];
37          rb = obbB->getExtent()[0] * absR[i][0] +
38               obbB->getExtent()[1] * absR[i][1] +
39               obbB->getExtent()[2] * absR[i][2];
40
41          if (std::abs(t[i]) > ra + rb) return false;
42      }
43
44      // Test ejes L = B0 L = B1 L = B2
45      for (int i = 0; i < 3; ++i) {
46          ra = obbA->getExtent()[0] * absR[0][i] +
47               obbA->getExtent()[1] * absR[1][i] +
48               obbA->getExtent()[2] * absR[2][i];
49          rb = obbB->getExtent()[i];
50
51          if (std::abs(t[0] * R[0][i] + t[1] * R[1][i] + t[2] * R[2][i]) > ra
                + rb) return false;
52      }
53
54      // Test eje L = A0 x B0
55      ra = obbA->getExtent()[1] * absR[2][0] + obbA->getExtent()[2] * absR
            [1][0];
56      rb = obbB->getExtent()[1] * absR[0][2] + obbB->getExtent()[2] * absR
            [0][1];
57      if (std::abs(t[2] * R[1][0] - t[1] * R[2][0]) > ra + rb) return false;
58
59      // Test eje L = A0 x B1
60      ra = obbA->getExtent()[1] * absR[2][1] + obbA->getExtent()[2] * absR
            [1][1];
```

```
61      rb = obbB->getExtent()[0] * absR[0][2] + obbB->getExtent()[2] * absR
            [0][0];
62      if (std::abs(t[2] * R[1][1] - t[1] * R[2][1]) > ra + rb) return false;
63
64      // Test eje L = A0 x B2
65      ra = obbA->getExtent()[1] * absR[2][2] + obbA->getExtent()[2] * absR
            [1][2];
66      rb = obbB->getExtent()[0] * absR[0][1] + obbB->getExtent()[1] * absR
            [0][0];
67      if (std::abs(t[2] * R[1][2] - t[1] * R[2][2]) > ra + rb) return false;
68
69      // Test eje L = A1 x B0
70      ra = obbA->getExtent()[0] * absR[2][0] + obbA->getExtent()[2] * absR
            [0][0];
71      rb = obbB->getExtent()[1] * absR[1][2] + obbB->getExtent()[2] * absR
            [1][1];
72      if (std::abs(t[0] * R[2][0] - t[2] * R[0][0]) > ra + rb) return false;
73
74      // Test eje L = A1 x B1
75      ra = obbA->getExtent()[0] * absR[2][1] + obbA->getExtent()[2] * absR
            [0][1];
76      rb = obbB->getExtent()[0] * absR[1][2] + obbB->getExtent()[2] * absR
            [1][0];
77      if (std::abs(t[0] * R[2][1] - t[2] * R[0][1]) > ra + rb) return false;
78
79      // Test eje L = A1 x B2
80      ra = obbA->getExtent()[0] * absR[2][2] + obbA->getExtent()[2] * absR
            [0][2];
81      rb = obbB->getExtent()[0] * absR[1][1] + obbB->getExtent()[1] * absR
            [1][0];
82      if (std::abs(t[0] * R[2][2] - t[2] * R[0][2]) > ra + rb) return false;
83
84      // Test eje L = A2 x B0
85      ra = obbA->getExtent()[0] * absR[1][0] + obbA->getExtent()[1] * absR
            [0][0];
86      rb = obbB->getExtent()[1] * absR[2][2] + obbB->getExtent()[2] * absR
            [2][1];
87      if (std::abs(t[1] * R[0][0] - t[0] * R[1][0]) > ra + rb) return false;
88
89      // Test eje L = A2 x B1
90      ra = obbA->getExtent()[0] * absR[1][1] + obbA->getExtent()[1] * absR
            [0][1];
91      rb = obbB->getExtent()[0] * absR[2][2] + obbB->getExtent()[2] * absR
            [2][0];
92      if (std::abs(t[1] * R[0][1] - t[0] * R[1][1]) > ra + rb) return false;
93
94      // Test eje L = A2 x B2
95      ra = obbA->getExtent()[0] * absR[1][2] + obbA->getExtent()[1] * absR
            [0][2];
96      rb = obbB->getExtent()[0] * absR[2][1] + obbB->getExtent()[1] * absR
            [2][0];
97      if (std::abs(t[1] * R[0][2] - t[0] * R[1][2]) > ra + rb) return false;
98
99
100     return true;
101 }
```

La colisión entre **OrientedBox** y **AxisAlignedBox** es muy sencilla una vez hemos logrado implementar la anterior. Se basa en convertir la caja alineada con los ejes en una caja orientada calculando sus parámetros. Posteriormente, realizamos el test OBB con OBB como puede verse en la figura 4.48.

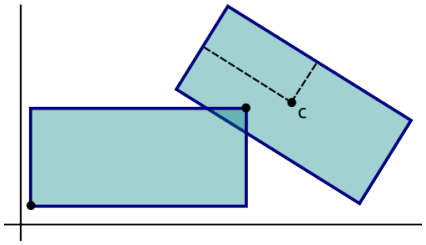


Figura 4.48: Test de colisión OBB ABB

```
bool Shape::getCollisionOBBAABB(Shape* shapeA, Shape* shapeB) {
    // Hacemos la conversion (estamos seguros de que uno es AABB y otro OBB
    AxisAlignedBox* aabb;
    OrientedBox* obb;
    if (shapeA->getType() == OBB) {
        obb = static_cast<OrientedBox*>(shapeA);
        aabb = static_cast<AxisAlignedBox*>(shapeB);
    } else {
        obb = static_cast<OrientedBox*>(shapeB);
        aabb = static_cast<AxisAlignedBox*>(shapeA);
    }

    // Convertimos aabb en obb
    Ogre::Vector3 minPos = aabb->getMinPos();
    Ogre::Vector3 maxPos = aabb->getMaxPos();
    Ogre::Vector3 extent = (maxPos - minPos) * 0.5f;
    Ogre::Vector3 center = (maxPos + minPos) * 0.5f;

    OrientedBox convertedOBB("convertedOBB", center, extent, Ogre::Matrix3
        ::IDENTITY);

    return getCollisionOBBOBB(obb, &convertedOBB);
}
```

Cuerpos colisionables, clase Body

Como hemos mencionado anteriormente, un *Body* representa la parte colisionable de un objeto y está formado por un vector de formas (*Shape*) y una transformación (traslación con respecto al origen, escala y rotación). Los cuerpos tienen un tipo (entero) que permite agruparlos y filtrarlos en la detección de colisiones. Para manejar las colisiones entre dos cuerpos, debemos cruzar las formas de ambos en coordenadas del mundo, en ningún caso locales al objeto. Aplicar la transformación de cada cuerpo a cada forma en todas las iteraciones del bucle de juego es demasiado costoso. Por ello, he decidido almacenar un segundo vector de formas en coordenadas del mundo, en el eterno dilema de la eficiencia tempo/memoria ha ganado el tiempo en este caso. A continuación, mostramos la definición de la clase.

```
class Body {
    public:
        Body(GameObject* gameObject,
             const std::vector<Shape*>& shapes,
             const Ogre::Vector3& position = Ogre::Vector3::ZERO,
             const Ogre::Vector3& scale = Ogre::Vector3::UNIT_SCALE,
             const Ogre::Quaternion& orientation = Ogre::Quaternion::
                 IDENTITY,
             int type = 0);

        Body(GameObject* gameObject = 0,
             const Ogre::Vector3& position = Ogre::Vector3::ZERO,
             const Ogre::Vector3& scale = Ogre::Vector3::UNIT_SCALE,
             const Ogre::Quaternion& orientation = Ogre::Quaternion::
                 IDENTITY,
             int type = 0);

        ~Body();

        GameObject* getGameObject();
        void setGameObject(GameObject* gameObject);

        const std::vector<Shape*> getShapes() const;
        const std::vector<Shape*> getWorldSpaceShapes() const;

        void addShape(Shape* shape);
        bool removeShape(Shape* shape);
        bool removeShape(const Ogre::String& name);

        Ogre::Matrix4 getTransform() const;
        void setTransform(const Ogre::Matrix4& transform);
        const Ogre::Vector3& getPosition() const;
        void setPosition(const Ogre::Vector3& position);
        const Ogre::Vector3& getScale() const;
        void setScale(const Ogre::Vector3& scale);
        const Ogre::Quaternion& getOrientation() const;
        void setOrientation(const Ogre::Quaternion& orientation);

        int getType() const;
        void setType(int type);

        static bool getCollision(Body* bodyA, Body* bodyB);
    private:
        GameObject* _gameObject;
        std::vector<Shape*> _shapes;
        std::vector<Shape*> _worldShapes;
        Ogre::Vector3 _position;
        Ogre::Vector3 _scale;
        Ogre::Quaternion _orientation;
        int _type;

        void updateWorldShapes();
        void createWorldShapes();
```

```
};
```

Gestor de colisiones, clase CollisionManager

El gestor de colisiones sigue el patrón de diseño *Singleton* [15] y lleva el registro de todos los cuerpos colisionables de la escena (*Body*). Es posible añadir o eliminar cuerpos según nos convenga con los métodos *addBody()* y *removeBody()*. Una vez en cada iteración del bucle de juego es recomendable llamar al método *checkCollisions* para detectar e informar de las colisiones que se produzcan.

Sólo se comprobarán colisiones entre cuerpos para cuyo tipo exista un callback. Los callbacks son objetos función de BOOST que reciben dos punteros a *Body* y no devuelven nada. Utilizando *boost:bind* podemos crear un objeto *boost:function* y añadir el callback para dos cuerpos de un tipo determinado. Por ejemplo, podemos hacer que el método *callbackSpellEnemy* sea llamado cuando colisionen cuerpos de los supuestos tipos *Spell* (cuyo número podría ser el 4) y *Enemy* (cuyo número podría ser el 8). Podemos incluso definir callbacks para el momento en el que empieza una colisión, para el tiempo que dure la colisión o para el instante en el que los cuerpos se separen.

En el método checkCollisions no sólo se filtran los cuerpos para los que existe un callback definido sino que no se comprueban aquellos que están a una distancia prudencial. Es cierto que, para cantidades ingentes de elementos este particionado no es suficiente aunque para nuestras necesidades funciona correctamente. A continuación se adjunta la definición de la clase:

```
class CollisionManager: public Ogre::Singleton<CollisionManager> {
    public:b
        typedef boost::function<void(Body*, Body*)> CollisionCallback;
        enum CallbackType {BEGINCOLLISION, COLLIDING, ENDCOLLISION, ALL};

        CollisionManager();
        ~CollisionManager();

        static CollisionManager& getSingleton();
        static CollisionManager* getSingletonPtr();

        void addBody(Body* body);
        bool removeBody(Body* body);
        void removeAllBodies();

        void addCollisionCallback(int typeA,
                                  int typeB,
                                  CollisionCallback callback,
                                  CallbackType callbackType =
                                     BEGINCOLLISION);
        bool removeCollisionCallback(int typeA, int typeB, CallbackType
            callbackType = ALL);
        void removeAllCollisionCallbacks(CallbackType type = ALL);

        void checkCollisions();
    private:
        typedef boost::unordered_map<int, boost::unordered_map<int,
            CollisionCallback> > CollisionCallbackTable;
```

```
26          typedef boost::unordered_map<Body*, std::set<Body*> >
               CollidingBodies;
27
28          CollisionCallbackTable _beginCallbackTable;
29          CollisionCallbackTable _inCallbackTable;
30          CollisionCallbackTable _endCallbackTable;
31
32          CollidingBodies _collidingBodies;
33          std::list<Body*> _bodies;
34
35          Sphere* _sphereA;
36          Sphere* _sphereB;
37
38          bool existsCallback(int typeA,
39                              int typeB,
40                              CallbackType calbackType,
41                              CollisionCallback* collisionCallback);
42
43          bool wereColliding(Body* bodyA, Body* bodyB);
44  };
```

El sistema de colisiones de **Sion Tower** puede obtenerse de forma completamente independiente y debidamente documentado a través de la siguiente dirección:

http://forja.rediris.es/frs/download.php/2091/siontower-collisions-v0.2.tar.gz

4.4.5. Exportación de modelos 3D

El artista de personajes 3D de **Sion Tower**, Antonio Jiménez Rodríguez trabaja con la herramienta privativa *Cinema 4D* [42] (ver figura 4.49). El dilema viene cuando OGRE3D sólo acepta un formato binario propio basado en un sencillo *XML* de extensión *.mesh*. El motor gráfico utiliza este formato propio (pero abierto) para optimizar la organización de los datos en los modelos 3D, siempre mirando hacia el rendimiento. La mayoría de herramientas de modelado y animación tridimensionales como *Blender* cuentan con exportadores a OGRE3D que funcionan correctamente y son de código abierto pero solo existía un exportador para *Cinema 4D*, el cual es privativo.

El plugin de exportación se llama *I/Ogre* y, siguiendo su manual, la exportación se realiza de forma correcta. El artista utilizaba una escala distinta a la mía (una unidad de la herramienta 3D equivale a un metro dentro del juego). Inicialmente no ocurría nada ya que el exportador era capaz de aplicarle escalas a los modelos pero me di cuenta de que no funcionaba correctamente. Decidí escribir un pequeño script en *Python* para corregir este problema en los modelos exportados.

Plugin de exportación privativo

Si tenemos un modelo `personaje`, tras la exportación contaremos con los ficheros:

- `personaje.mesh.xml`: xml en texto plano con la información de los vértices y caras del personaje. Esta información incluye la posición y rotación exacta de cada vértice dentro del espacio local del objeto. Se hacen referencias al fichero con el esqueleto y a sus materiales.

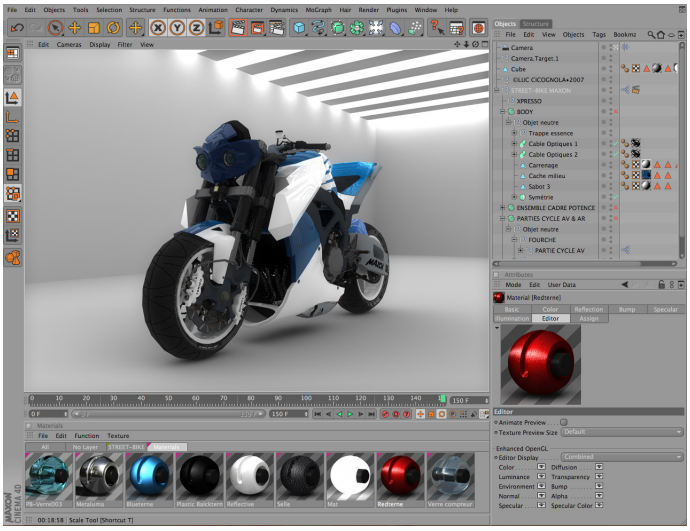


Figura 4.49: Moto modelada con Cinema 4D

- `personaje.skeleton.xml`: xml en texto plano con la información de los huesos que forman el esqueleto del personaje de cara a la animación y su peso sobre los vértices asociados. Así mismo, incluye los fotogramas claves indicando la posición de cada hueso en cada momento.

- `personaje.material`: fichero en texto plano que sigue una sintaxis especial indicando los materiales que componen al personaje (texturas, colores, brillo...). Se puede leer más sobre esto en el artículo *Materiales* de **IberOgre** [9].

Ejemplo abreviado de fichero `.mesh.xml`:

```
<mesh>
  <submeshes>
    <submesh material="pattern" usesharedvertices="false"
             use32bitindexes="false" operationtype="triangle_list">
          <faces count="172">
                <face v1="2" v2="1" v3="0" />
                <face v1="4" v2="3" v3="1" />
                <face v1="5" v2="4" v3="2" />

          ...

      </faces>
      <geometry vertexcount="121">
                <vertexbuffer positions="true" normals="true"
                      texture_coords="1" texture_coords_dimensions_0="2">
          <vertex>
            <position x="-103.78136444" y="77.32855225" z="-123.70773315" /
               >
            <normal x="-0.83172542" y="-0.54570407" z="0.10217582" />
            <texcoord u="0.00000000" v="1.00000000" />
          </vertex>
          <vertex>
```

```
            <position x="-128.57696533" y="259.42965698" z="-35.53633881" />
                >
                        <normal x="-0.70674056" y="-0.68547767" z="
                            -0.17503749" />
                        <texcoord u="0.00000000" v="0.50000000" />
          </vertex>
          <vertex>
            <position x="-124.34030151" y="258.42745972" z="-32.27506256" />
                >
            <normal x="-0.37069467" y="-0.82011324" z="0.43588960" />
            <texcoord u="0.50000000" v="0.50000000" />
          </vertex>

          ...

        </vertexbuffer>
      </geometry>
      <boneassignments>
        <vertexboneassignment vertexindex="0" boneindex="19" weight="
            1.00000000" />
        <vertexboneassignment vertexindex="1" boneindex="19" weight="
            1.00000000" />
        <vertexboneassignment vertexindex="2" boneindex="19" weight="
            1.00000000" />
        <vertexboneassignment vertexindex="3" boneindex="19" weight="
            1.00000000" />

        ...

      </boneassignments>
    </submesh>

        ...

    </submeshes>
</mesh>
```

Ejemplo abreviado de fichero `.skeleton.xml`:

```
<skeleton>
  <bones>
    <bone id="0" name="Skeleton_root">
      <position x="-0.00000000" y="201.51861572" z="0.00000000" />
      <rotation angle="1.57079625">
        <axis x="0.00000000" y="1.00000000" z="0.00000000" />
      </rotation>
      <scale x="1.00000000" y="1.00000000" z="1.00000000" />
    </bone>
    <bone id="1" name="Cintura">
      <position x="49.73685074" y="2.21317220" z="11.21903801" />
      <rotation angle="0.55032432">
        <axis x="-0.00366197" y="-0.99054980" z="-0.13710468" />
      </rotation>
      <scale x="1.00000000" y="1.00000000" z="1.00000000" />
```

```
      </bone>

      ...

    </bones>
    <animations>
      <animation name="idle" length="1.46666670">

        ...

      </animation>

      ...

    </animations>
  </skeleton>
```

Posteriormente, pasaríamos a formato binario los ficheros con la malla y el esqueleto del personaje utilizando el script que se distribuye con OGRE3D llamado `OgreXMLConverter`. Antes de aplicar este script, podemos manipular los ficheros *XML* y corregir la escala de vértices y huesos en fotogramas clave.

Script de corrección de escala

El script que se ha desarrollado simplemente toma uno de estos ficheros, recorre todos los nodos del *XML* y aplica la escala deseada a todas las medidas (posiciones, tamaños, etc). Posteriormente guarda el resultado en un fichero auxiliar y, si lo deseamos, aplica por nosotros el script `OgreXMLConverter`. Su sintaxis es la siguiente:

```
python fixExport.py [mesh|skeleton] originXML destinyXML
scaleFactor [destinyBin]
```

- `[mesh|skeleton]`: indicamos si queremos corregir una malla o un fichero de esqueleto.
- `originXML`: fichero xml de origen.
- `destinyXML`: fichero xml destino, en el que queremos guardar el resultado de la transformación.
- `scaleFactor`: factor por el que escalaremos las medidas. Un factor de $0{,}5$ reduce el tamaño a la mitad y uno de 2 lo duplica.
- `[destinyBin]`: si lo deseamos, indicamos el fichero binario resultante y se aplicará el script `OgreXMLConverter`.

A continuación, adjuntamos el script completo:

```
#!/usr/bin/env python
# -*- coding: utf-8  -*-

##############################################################################
# This file is part of Sion Tower.                                           #
#                                                                            #
# This program is free software: you can redistribute it and/or modify       #
```

```
8  # it under the terms of the GNU General Public License as published by #
9  # the Free Software Foundation, either version 3 of the License, or    #
10 # any later version.                                                    #
11 #                                                                       #
12 # This program is distributed in the hope that it will be useful,      #
13 # but WITHOUT ANY WARRANTY; without even the implied warranty of       #
14 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        #
15 # GNU General Public License for more details.                         #
16 #                                                                       #
17 # You should have received a copy of the GNU General Public License    #
18 # along with this program.  If not, see <http://www.gnu.org/licenses/>.#
19 #                                                                       #
20 # Copyright (C) 2011, David Saltares Marquez, david.saltares@gmail.com #
21 #########################################################################
22
23 import os
24 import sys
25
26 from xml.dom import minidom
27 from xml.etree.ElementTree import ElementTree
28
29 # Ruta al OgreXMLConverter
30 converter = '/usr/local/bin/OgreXMLConverter'
31
32 def sintaxis():
33     print 'Syntax error, the correct syntax is:'
34     print '    python fixExport.py [mesh|skeleton] originXML destinyXML
35           scaleFactor [destinyBin]'
36     print ''
37
38 def processMesh(root, scale):
39     print '* Processing XML mesh...'
40
41     vertices = root.findall('submeshes/submesh/geometry/vertexbuffer/vertex
          ')
42
43     for vertex in vertices:
44         position = vertex.find('position')
45
46         # Convertimos x
47         x = float(position.attrib['x'])
48         x = x * scale
49         position.attrib['x'] = str(x)
50
51         # Convertimos y
52         y = float(position.attrib['y'])
53         y = y * scale
54         position.attrib['y'] = str(y)
55
56         # Convertimos z
57         z = float(position.attrib['z'])
58         z = z * scale
59         position.attrib['z'] = str(z)
60
61
62 def processSkeleton(root, scale):
```

```
63      print '* Processing XML skeleton...'
64
65      positions = root.findall('bones/bone/position')
66
67      for position in positions:
68          # Convertimos x
69          x = float(position.attrib['x'])
70          x = x * scale
71          position.attrib['x'] = str(x)
72
73          # Convertimos y
74          y = float(position.attrib['y'])
75          y = y * scale
76          position.attrib['y'] = str(y)
77
78          # Convertimos z
79          z = float(position.attrib['z'])
80          z = z * scale
81          position.attrib['z'] = str(z)
82
83      translations = root.findall('animations/animation/tracks/track/
84                                   keyframes/keyframe/translate')
85
86      for translation in translations:
87          # Convertimos x
88          x = float(translation.attrib['x'])
89          x = x * scale
90          translation.attrib['x'] = str(x)
91
92          # Convertimos y
93          y = float(translation.attrib['y'])
94          y = y * scale
95          translation.attrib['y'] = str(y)
96
97          # Convertimos z
98          z = float(translation.attrib['z'])
99          z = z * scale
100         translation.attrib['z'] = str(z)
101
102 def main():
103     # Cabecera
104     print ''
105     print 'CINEMA4D - OGRE EXPORT SCALE FIX'
106     print '================================'
107     print ''
108
109     # Comprobamos argumentos
110     if len(sys.argv) < 5:
111         sintaxis()
112         sys.exit(1)
113
114     # Cargamos documento XML
115     print '* Loading XML document...'
116     #doc = minidom.parse(sys.argv[2])
117     doc = ElementTree()
118     doc.parse(sys.argv[2])
```

```
119     root = doc.getroot()
120 
121     # Procesamos XML
122     print '* Checking document type...'
123     if sys.argv[1] == 'mesh':
124         processMesh(doc, float(sys.argv[4]))
125     elif sys.argv[1] == 'skeleton':
126         processSkeleton(doc, float(sys.argv[4]))
127     else:
128         print 'Unknown document type'
129         print ''
130         sys.exit(1)
131 
132     # Guardamos XML
133     print '* Saving XML document...'
134     doc.write(sys.argv[3])
135 
136     # Convertimos XML si definimos OgreXMLConverter
137     if converter != '' and len(sys.argv) == 6:
138         print '* Converting XML document to binary...'
139         command = converter + ' ' + sys.argv[3] + ' ' + sys.argv[5]
140         os.system(command)
141 
142 
143 if __name__ == "__main__":
144         main()
```

4.4.6. Carga de escenarios desde Blender

Creación de niveles y convenciones de nombrado

En **Sion Tower** los niveles se diseñan con la herramienta de modelado y animación 3D *Blender* [20]. Para más detalles sobre el proceso de creación de niveles es posible acudir al apéndice Manual de usuario en la página 157. Tras el diseño del nivel utilizando *Blender*, se procede su exportación al formato *Dotscene*, un sencillo XML que define la posición, escala y orientación de cada objeto en la escena.

El sistema de carga de niveles de **Sion Tower** procesa dicho fichero XML y va creando los elementos de juego. No obstante, no sólo se adjunta información sobre el escenario sino que necesitamos especificar dentro del nivel: oleadas de enemigos, malla de navegación para la búsqueda de caminos, sistemas de partículas... En definitiva, elementos que no pueden ser representados mediante una malla tridimensional sin más. Es necesaria una convención de nombrado para que el sistema que procesa los niveles sepa a qué se refiere cada elemento. La nomenclatura es la siguiente:

- **Escenario**: todos los elementos del escenario (paredes, suelo, mesas, sillas y otros muebles) siguen la regla `scene-nombre.numero`. El nombre del objeto es lo que lo identifica dentro del catálogo para poder recuperar su modelo de colisión y el número ayuda a hacerlo único dentro de la escena (dos objetos no pueden tener el mismo nombre). Ejemplo: `scene-door.005`.
- **Efectos de partículas**: puedes incluir efectos de partículas en cualquier punto de la escena. La regla es `particle-nombre.numero`. Ejemplo: `particle-flame.001`.
- **Luces**: el motor del juego reconoce las luces de manera automática, por lo que no debes preocuparte de sus nombres.

- **Malla de navegación**: la malla de navegación indica a los enemigos cuales son las zonas transitables del escenario y siempre debe llamarse `navMesh`. Veremos más sobre las mallas de navegación en secciones posteriores.
- **Enemigos**: los enemigos siguen la regla `enemy-tipo-t.numero`. La letra t representa el segundo en el que aparecerá el enemigo desde que se inicia la partida. Ejemplo: `enemy-goblin-25.001`.
- **Protagonista**: el elemento cuyo nombre sea `player` definirá la posición inicial del jugador dentro del nivel.
- **Geometría arbitraria**: toda la geometría que no siga la convención de nombrado será tratada como complementos del escenario. No se calcularán colisiones contra ellos (se podrán atravesar).

Para procesar ficheros XML se ha empleado la sencilla, ligera y rápida biblioteca libre PUGIXML [3].

Gestor de niveles, clase LevelManager

La clase *LevelManager* es la encargada de gestionar a todos los niveles de juego. También sigue el patrón de diseño *Singleton* ya que sólo necesitamos una instancia accesible desde varios módulos del sistema [15]. Cuando creamos el gestor de niveles al inicio de la aplicación procesa el fichero `[siontower]/media/levels/levels.xml` que contiene la lista ordenada de los niveles del juego.

```
<levels>
    <level id="level01"></level>
    <level id="level02"></level>
    <level id="level03"></level>
    <level id="level04"></level>
</levels>
```

En ese momento, el gestor de niveles crea todos los niveles cargando en memoria únicamente su información más básica: identificador, nombre, descripción y nombre de la canción que ha de sonar. Esto es necesario ya que hay que mostrar dicha información en la pantalla de selección de nivel. De esta manera, los niveles cuentan con dos estados: creado y cargado (listo para ser jugado). Al gestor de niveles podemos pedirle que nos devuelva la lista completa de niveles en orden o simplemente un nivel individual. Su definición es la siguiente:

```
class LevelManager: public Ogre::Singleton<LevelManager> {
    public:

        enum UnlockCode {
            ALREADYUNLOCKED,
            LASTLEVEL,
            UNLOCKED
        };

        LevelManager();
        ~LevelManager();

        static LevelManager& getSingleton();
        static LevelManager* getSingletonPtr();

```

```
        std::vector<Level*>& getLevels();
        Level* getLevel(const Ogre::String& id);
        int getNumLevels() const;

        Body* createBodyFromCatalog(const Ogre::String& name);

        UnlockCode tryUnlockLevel(const Ogre::String& code, int
            lastUnlockedLevel);
    private:
        std::vector<Level*> _levels;
        int _numLevels;

        typedef boost::unordered_map<Ogre::String, Body* > BodyCatalog;
        BodyCatalog _bodyCatalog;
        std::vector<Shape*> _shapes;

        void lookForLevels();
        void initialiseBodyCatalog();
};
```

La clase Level y el formato DotScene

Los objetos de la clase *Level* se identifican por un código único y obtienen su información de dos ficheros XML distintos. El primero es el que contiene la información básica y el segundo es el resultante de la exportación de Blender y que contiene todos los datos sobre la escena. Su nomenclatura siempre es `id_info.xml` y `id_scene.xml` respectivamente y se almacenan en el directorio `[siontower]/media/levels`.

Level cuenta con sencillos métodos para recuperar su información básica así como conocer si está cargado por completo y poder cargarlo o liberar sus objetos según sea necesario. En caso de que esté cargado podemos recuperar sus elementos (elementos del escenario, la posición inicial del jugador, las oleadas de enemigos *EnemySpawn* o la malla de navegación). El nivel hace, en la práctica, de clase contenedora a la que el estado de juego pregunta por su contenido.

Cuando se le pide a un nivel que cargue todos sus elementos, éste procesa el XML en formato *DotScene* de nombre `id_scene.xml` y va distribuyendo los nodos según sean enemigos, objetos del escenario, etc. A continuación se adjunta un fragmento de fichero *Dotscene* a modo de ejemplo.

```
<scene formatVersion="1.0.0">
  <nodes>
    <node name="scene-staircase.001">
      <position x="10.500000" y="1.400000" z="-9.600000"/>
      <quaternion x="0.000000" y="0.000000" z="-0.000000" w="1.000000"/>
      <scale x="1.000000" y="1.000000" z="1.000000"/>
      <entity name="scene-staircase.001" meshFile="staircase.mesh"/>
    </node>

    <node name="enemy.goblin.30.001">
      <position x="-1.982259" y="0.000000" z="21.304888"/>
      <quaternion x="0.000000" y="-0.999307" z="-0.000000" w="0.037231"/>
      <scale x="1.000000" y="1.000000" z="1.000000"/>
```

```
        <entity name="enemy.goblin.30.001" meshFile="enemy.goblin.mesh"/>
      </node>

      <node name="scene-wall.012">
        <position x="-12.000000" y="1.400000" z="-10.000000"/>
        <quaternion x="0.000000" y="-0.707107" z="-0.000000" w="0.707107"/>
        <scale x="1.000000" y="1.000000" z="1.000000"/>
        <entity name="scene-wall.012" meshFile="wall.mesh"/>
      </node>

      <node name="navMesh">
        <position x="-0.106243" y="0.000000" z="3.796432"/>
        <quaternion x="0.000000" y="0.000000" z="-0.000000" w="1.000000"/>
        <scale x="1.000000" y="1.000000" z="1.000000"/>
        <entity name="navMesh" meshFile="navMesh.mesh"/>
      </node>

      <node name="torch">
        <position x="-3.508866" y="0.965059" z="-11.709224"/>
        <quaternion x="0.000000" y="0.000466" z="-0.000000" w="1.000000"/>
        <scale x="1.000000" y="1.000000" z="1.000000"/>
        <entity name="torch" meshFile="torch.mesh"/>
      </node>

      <node name="Camera">
        <position x="-0.374790" y="7.257391" z="27.672583"/>
        <quaternion x="-0.131759" y="-0.016477" z="0.000481" w="0.991145"/>
        <scale x="1.000000" y="1.000000" z="1.000000"/>
        <camera name="Camera" fov="28.841546" projectionType="perspective">
          <clipping nearPlaneDist="0.100000" farPlaneDist="100.000000"/>
        </camera>
      </node>

      <node name="particle.flame.004">
        <position x="3.825604" y="1.854048" z="19.515015"/>
        <quaternion x="0.000000" y="-1.000000" z="-0.000000" w="0.000429"/>
        <scale x="1.000000" y="1.000000" z="1.000000"/>
        <entity name="particle.flame.004" meshFile="particle-fire.mesh"/>
      </node>

      <node name="Lamp">
        <position x="6.176817" y="5.903862" z="-6.176092"/>
        <quaternion x="-0.284166" y="0.726942" z="0.342034" w="0.523275"/>
        <scale x="1.000000" y="1.000000" z="1.000000"/>
        <light name="Spot" type="point">
          <colourDiffuse r="0.359337" g="0.500903" b="0.380572"/>
          <colourSpecular r="0.359337" g="0.500903" b="0.380572"/>
          <lightAttenuation range="5000.0" constant="1.000000" linear="
              0.033333" quadratic="0.000000"/>
        </light>
      </node>

    <environment>
      <colourAmbient r="0.000000" g="0.000000" b="0.000000"/>
      <colourBackground r="0.386832" g="0.386832" b="0.386832"/>
    </environment>
```

```
</scene>
```

La definición de la clase *Level* es la siguiente:

```
class Level {
    public:
        Level(const Ogre::String& id);
        ~Level();

        void load();
        void unload();
        bool isLoaded() const;

        const Ogre::String& getId() const;
        const Ogre::String& getName() const;
        const Ogre::String& getDescription() const;
        const Ogre::String& getSongName() const;

        SongPtr getSong();
        std::vector<GameMesh*>& getSceneObjects();
        const Ogre::Vector3& getPlayerPosition() const;
        std::vector<EnemySpawn>& getEnemySpawns();
        NavigationMesh* getNavigationMesh();

    private:
        // Informacion basica
        Ogre::String _id;
        Ogre::String _name;
        Ogre::String _description;
        Ogre::String _musicName;
        Ogre::String _musicGroup;
        bool _loaded;

        // Cancion
        SongPtr _song;

        // Elementos del juego y escenario
        Player* _player;
        std::vector<GameMesh*> _sceneObjects;
        std::vector<std::pair <Ogre::SceneNode*, Ogre::Light*> > _lights;
        std::vector<std::pair <Ogre::SceneNode*, Ogre::Entity*> > _geometry
            ;
        std::vector<std::pair <Ogre::SceneNode*, Ogre::ParticleSystem*> >
            _particles;
        NavigationMesh* _navigationMesh;
        Ogre::Vector3 _playerPos;
        std::vector<EnemySpawn> _enemySpawns;

        void loadBasicInfo();

        void loadAmbientInfo(const pugi::xml_document& doc);
        void loadNodesInfo(const pugi::xml_document& doc);
        void loadEntity(const pugi::xml_node& node,
                        const Ogre::String& name,
                        const Ogre::Vector3& position,
```

```
                            const Ogre::Vector3& scale,
                            const Ogre::Quaternion& orientation);
        void loadLight(const pugi::xml_node& node,
                       const Ogre::String& name,
                       const Ogre::Vector3& position,
                       const Ogre::Vector3& scale,
                       const Ogre::Quaternion& orientation);
        void loadCamera(const pugi::xml_node& node,
                        const Ogre::Vector3& position,
                        const Ogre::Quaternion& orientation);
};
```

Catálogo de objetos

Los objetos que forman parte del escenario se cargan dentro del juego y automáticamente cuentan con su modelo colisionable de forma que ni el personaje ni los hechizos pueden atravesarlos. Esto se debe a que existe un catálogo de objetos gestionado por *LevelManager*. Cuando un nivel se encuentra con un objeto de tipo `wall`, le pregunta al gestor de niveles si existe algún modelo colisionable para dicho tipo y en caso afirmativo le asigna una copia actuando según el patrón *Factory* [15].

4.4.7. Búsqueda de caminos

Los enemigos de **Sion Tower** no pueden ver por sí mismos y no son capaces de buscar al personaje para atacarle. Es necesario un sistema de búsqueda de caminos que guíe a estos enemigos por el escenario sin que atraviesen obstáculos como paredes, sillas, mesas o columnas. El sistema debía ser lo suficientemente general como para que fuera sencillo añadir niveles al juego y que los enemigos pudieran seguir comportándose de la forma esperada. En esta sección expicaremos cómo se ha resulto el problema de la búsqueda de caminos en el juego.

Malla de navegación

La solución pasa por asociar a cada nivel información sobre las zonas transitables del escenario. Una malla de navegación es ideal para almacenar este tipo de información y puede crearse junto al nivel utilizando *Blender*. Así conseguimos que no sea necesario que un programador cree los niveles, alguien especializado en dicho aspecto y sin conocimientos de cómo está hecho el sistema puede hacerlo. La malla estará formada por un conjunto de triángulos agrupados de forma que construyan un grafo conexo, dirigido y ponderado como puede verse en la figura 4.50.

Cada una de las celdas de la malla de navegación está modelada por la clase *Cell*. Básicamente contiene información sobre los puntos que forman su triángulo, el plano delimitado por el mismo y las celdas vecinas. Incluye un método para conocer si un punto está contenido en la celda llamado *containsPoint()* y otro para, dado un punto en un plano bidimensional, conocer la altura en la que encaja dentro de la celda llamado *getHeight()*. Este último resulta especialmente útil para incluir desniveles dentro de un escenario (comoe escaleras o rampas). Con el método *classifyPathToCell()* podemos saber si una línea atraviesa la celda, termina en la celda, o no guarda ninguna relación con la misma. Se adjunta la definición de la clase a continuación.

Figura 4.50: Malla de navegación dibujada en modo depuración

```
class Cell {
    public:
        enum CellSide {
            SIDE_AB = 0,
            SIDE_BC,
            SIDE_CA
        };

        enum CellVert {
            VERT_A = 0,
            VERT_B,
            VERT_C
        };

        enum PathResult {
            NO_RELATIONSHIP,
            ENDING_CELL,
            EXITING_CELL,
        };

        Cell(int id,
             const Ogre::Vector3& pointA,
             const Ogre::Vector3& pointB,
             const Ogre::Vector3& pointC);
        ~Cell();

        int getId() const;
        const Ogre::Vector3& getVertex(int index);
        Cell* getLink(CellSide side) const;
        void setLink(CellSide side, Cell* cell);
        bool requestLink(const Ogre::Vector3& pointA,
                         const Ogre::Vector3& pointB,
                         Cell* cell);
        const Ogre::Vector3& getCenter() const;
        bool containsPoint(const Ogre::Vector3& point) const;
        PathResult classifyPathToCell(const Line2D& path,
                                      Cell** nextCell,
                                      CellSide &side,
                                      Ogre::Vector2 &intersection);
```

```
40
41          void getHeight(Ogre::Vector3 &point);
42      private:
43          // ID
44          int _id;
45
46          // Geometria
47          Ogre::Vector3 _vertex[3];
48          Ogre::Vector3 _center;
49          Cell* _links[3];
50          Line2D _sides[3];
51          Plane _plane;
52  };
```

La malla de navegación o *NavigationMesh* toma el fichero XML resultante de la exportación de *Blender* a OGRE3D y crea las celdas (*Cell*) que forman el grafo. Incluye el método *findCell()* para que una entidad sepa en qué celda se encuentra en cada momento. Con *buildPath()* podemos obtener un camino desde cualquier punto de la malla a otro. Utilizando *lineOfSightTest()* podemos conocer si desde un punto determinado existe una linea de visión hacia otro. Más adelante hablaremos sobre el algoritmo utilizado para obtener dichos caminos.

```
1   class NavigationMesh {
2       public:
3           typedef std::vector<Cell*> Cells;
4
5           typedef std::list<Cell*> CellPath;
6
7           typedef std::list<Ogre::Vector3> PointPath;
8
9           NavigationMesh(const Ogre::String& fileName = "");
10          ~NavigationMesh();
11
12          void addCell(int id,
13                       const Ogre::Vector3& pointA,
14                       const Ogre::Vector3& pointB,
15                       const Ogre::Vector3& pointC);
16
17          void linkCells();
18
19          void clear();
20
21          Cell* getCell(int index);
22          int getCellNumber();
23
24
25          bool buildPath(PointPath& path,
26                         const Ogre::Vector3& startPos,
27                         const Ogre::Vector3& endPos,
28                         Cell* startCell = 0,
29                         Cell* endCell = 0);
30
31          Cell* findCell(const Ogre::Vector3& pos);
32
33          bool lineOfSightTest(const Ogre::Vector3& start,
```

```
                                const Ogre::Vector3& end,
                                Cell* startCell,
                                Cell* endCell);

        CellPath::iterator getFurthestVisibleCell(CellPath& path,
                                                  CellPath::iterator
                                                      startIt);

    private:
        void loadCellsFromXML(const Ogre::String& fileName);
        bool makeSpline(PointPath& path);
        int simplifyPath(CellPath& cellPath);

        Cells _cells;
        int _cellNumber;

        // Grafo y Floyd
        Ogre::Real* _graph;
        int* _paths;

        void initGraph();
        void floyd();
        void precomputePaths();
        void recoverPath(int i, int j, CellPath& cellPath);
};
```

Precomputación de caminos

Inicialmente se pensó en emplear el algoritmo A* [14] pero dicho sistema trata de buscar el camino demandado cada vez que éste se consulta. Los enemigos procuran pedir nuevos caminos en el menor número de momentos posibles, pero en muchas ocasiones es estrictamente necesario. Si tenemos unos cinco enemigos en pantalla y en el mismo cuadro todos piden la búsqueda de un camino diferente podemos tener problemas si el algoritmo no es muy rápido. Por esta razón se ha tomado la decisión de emplear el algoritmo de Floyd y precomputar los caminos mínimos. Además, el algoritmo A* no es capaz de asegurar su optimalidad, al contrario que Floyd.

Cuando creamos la malla de navegación debemos extraer el grafo a partir de sus celdas. Almacenamos el gráfo en una matriz `nxn` donde `n` es el número de celdas y `grafo[a][b]` es igual al coste de ir desde el nodo a al nodo b. Esta tarea se lleva a cabo en el método privado *initGraph()* y todos los costes se inicializan a infinito menos entre las celdas vecinas ya que en tal caso sería 1. Inicialmente nos podría preocupar el coste en memoria de esta aproximación. En realidad el coste en memoria no es demasiado elevado. Por ejemplo, en un nivel grande formado por 200 celdas, tendríamos una matriz de 40000 reales, esto implica un espacio en memoria de 156KB.

El algoritmo de Floyd toma una matriz de costes y devuelve la matriz de costes mínimos así como una matriz de caminos para poder reconstruir las rutas [46]. Este algoritmo calcula la ruta mínima entre todos los nodos de un grafo y resulta ideal para precalcular todos los caminos en el juego. Básicamente cruza todos los vértices i y j buscando un tercero k a modo de atajo de forma que $coste(i,k) + coste(k,j) < coste(i,j)$. Cuando encuentra un atajo modifica la matriz de costes y actualiza la matriz de caminos para que el nuevo nodo intermedio se vea reflejado. Su orden es $O(n^3)$ pero sólo debe ejecutarse una vez durante la carga del nivel. El algoritmo integrado en nuestro sistema es el

siguiente:

```
void NavigationMesh::floyd() {
    for (int k = 0; k < _cellNumber; ++k) {
        for (int i = 0; i < _cellNumber; ++i) {
            for (int j = 0; j < _cellNumber; ++j) {
                Ogre::Real ikj = _graph[i * _cellNumber + k] +
                                 _graph[k * _cellNumber + j];
                if (ikj < _graph[i * _cellNumber + j]) {
                    _graph[i * _cellNumber + j] = ikj;
                    _paths[i * _cellNumber + j] = k;
                }
            }
        }
    }
}
```

Creando el grafo tras generar las celdas de la malla y precalculando los caminos mínimos con Floyd, nuestro sistema ya estaría listo para usarse. No obstante, los caminos generados producen un efecto de zig-zag no deseado. Los enemigos podrían saltar celdas intermedias si entre el origen y el destino existe una ruta sin obstáculos. Durante la inicialización recorremos todos las combinaciones de caminos posibles simplificando el camino utilizando los tests de línea de visión (*lineOfSightTest()*) y actualizando la matriz de caminos cuando sea necesario.

```
void NavigationMesh::precomputePaths() {
    // Recorremos todas las combinaciones inicio - destino
    for (int i = 0; i < _cellNumber; ++i) {
        for (int j = 0; j < _cellNumber; ++j) {
            // Recuperamos el camino de i a j
            CellPath cellPath;
            recoverPath(i, j, cellPath);

            // Simplificamos el camino de i a j
            simplifyPath(cellPath);

            // Actualizamos la matriz de caminos con el camino simplificado
                de i a j
            for (CellPath::iterator it = cellPath.begin(); it != cellPath.
               end(); ++it) {
                CellPath::iterator nextIt = it;
                ++nextIt;

                if (nextIt != cellPath.end()) {
                    int idA = (*it)->getId();
                    int idB = (*nextIt)->getId();

                    // El camino entre nodos consecutivos es directo
                    _paths[idA * _cellNumber + idB] = -1;
                }
            }
        }
    }
```

```
}
```

Recuperación del camino

Cuando cualquier enemigo hace una llamada a *NavigationMesh::buildPath()* para encontrar la ruta más corta hasta el protagonista debemos acudir a la matriz de caminos y recuperar la ruta. Se trata de un algoritmo recursivo que se basa en la estructura de la matriz de caminos. Si tenemos los nodos a y b y observamos que $camino(a, b) = c$ significa que el camino más corto entre a y b pasa por c. Crear un vector de puntos o *CellPath* basándonos en esta idea, es sencillo entoces.

```
void NavigationMesh::recoverPath(int i, int j, CellPath& cellPath) {
    // Tomamos el nodo intermedio por el que pasa el camino de i a j
    int k = _paths[i * _cellNumber + j];

    // Si hay atajo
    if (k != -1) {
        // Recuperamos el camino de i a k
        recoverPath(i, k, cellPath);

        // Insertamos la celda de id k en el camino
        cellPath.push_back(_cells[k]);

        // Recuperamos el camino de k a j
        recoverPath(k, j, cellPath);
    }
}
```

Suavizado del camino

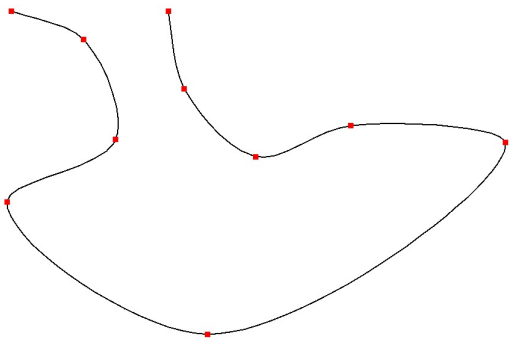


Figura 4.51: Spline cúbico de Catmull-Roll

A pesar de haber reducido celdas intermedias siempre que haya sido posible, en el camino se producen cambios de orientación demasiado bruscos cuando se llega de un punto intermedio a otro. Es posible suavizar el camino empleando una interpolación. Tras recuperar el camino en el método *NavigationMesh::buildPath()*, introducimos puntos intermedios gracias al spline cúbico de Catmull-Roll [43] (ver figura 4.51). Es un algoritmo muy barato en tiempo de computación y mejora en gran medida la calidad del comportamiento de los enemigos al seguir un camino de puntos.

El sistema se ha probado con una malla de navegación formada por 100 celdas y el tiempo de carga y precomputación ha sido de 2 segundos. Cada consulta para recuperar un camino únicamente ha necesitado 0.2 milisegundos.

4.4.8. Inteligencia Artificial: Steering Behaviors

Hemos expuesto la técnica que utilizamos para que los enemigos *vean* los obstáculos y puedan desplazarse por el escenario. No obstante, aún teníamos que conseguir que realizaran el movimiento de seguir la línea de puntos de una forma realista. Además, si dos enemigos se dirigen a capturar al personaje, podrían colisionar entre sí y tenemos que evitarlo a toda costa. En definitiva, necesitamos modelar el comportamiento de los enemigos y su forma de desplazarse por el entorno. En esta sección pondremos solución a dicho problema.

Conceptos básicos de movimiento

Los algoritmos de movimiento toman datos de entrada del propio personaje y de su entorno como otros personajes, escenario, un camino a seguir, obstáculos o un objeto. Finalmente, producen una salida representando el movimiento que debería realizar la entidad. Algunos algoritmos de movimiento requieren muy pocos datos del personaje como su posición. En cambio, otros necesitan su posición, velocidad lineal y circular actuales, etc. Como salida pueden producir simplemente la velocidad deseada (en forma de vector) o una fuerza a aplicar sobre la entidad de forma que resulte en una aceleración para alcanzar el objetivo [14].

Los *Steering Behaviors* son algoritmos de movimiento dinámico, es decir, tienen en cuenta las propiedades cinemáticas completas del personaje y producen una aceleración para cambiar su velocidad actual. Para ir de un punto a otro, el algoritmo dinámico hace que el personaje acelere hacia dicho punto, cuando se va acercando trata de frenar progresivamente para detenerse completamente en el destino [14].

Los algoritmos de movimiento dinámicos producen comportamientos mucho más realistas. Cuando una persona va de un punto a otro su velocidad no es constante, sobre todo al iniciar la marcha. Sería muy extraño que los enemigos de **Sion Tower** fueran de una velocidad nula a la máxima en un sólo frame.

Las clases Kinematic y Steering

Como hemos dicho anteriormente, los cambios repentinos de velocidad pueden parecer extraños y poco naturales. Es necesario almacenar más información para cada personaje de forma que los algoritmos de movimiento puedan trabajar sobre ella. Tendremos una clase *Kinematic* que almacenará estas propiedades: posición, orientación, velocidad lineal, velocidad angular y máxima velocidad. La información sobre la dinámica del personaje cuenta con un método *update()* para ser actualizado por la salida producida por los algoritmos de movimiento (clase *Steering*). Además cuenta con métodos para modificar automáticamente la orientación del personaje en función de su velocidad con *setOrientationFromVelocity()* y para mirar en una dirección determinada con *lookAt()*. A continuación se adjunta la definición de la sencilla clase.

```
1 class Kinematic {
2     public:
```

```
        Kinematic(const Ogre::Vector3& position = Ogre::Vector3::ZERO,
                  Ogre::Real orientation = 0.0f,
                  const Ogre::Vector3& velocity = Ogre::Vector3::UNIT_SCALE
                      ,
                  Ogre::Real rotation = 0.0f,
                  Ogre::Real maxSpeed = 0.0f);

        const Ogre::Vector3& getPosition() const;
        Ogre::Real getOrientation() const;
        const Ogre::Vector3& getVelocity() const;
        Ogre::Real getRotation() const;
        Ogre::Real getMaxSpeed() const;

        void setPosition(const Ogre::Vector3& position);
        void setOrientation(Ogre::Real orientation);
        void setVelocity(const Ogre::Vector3& velocity);
        void setRotation(Ogre::Real rotation);
        void setMaxSpeed(Ogre::Real maxSpeed);

        void update(const Steering& steering, Ogre::Real deltaT);

        void setOrientationFromVelocity();
        void lookAt(const Ogre::Vector3& point);
    private:
        Ogre::Vector3 _position;
        Ogre::Real _orientation;
        Ogre::Vector3 _velocity;
        Ogre::Real _rotation;
        Ogre::Real _maxSpeed;
};
```

La salida de los algoritmos de movimiento la modelaremos utilizando la clase *Steering*. Estos algoritmos funcionan aplicando fuerzas así que simplemente necesitamos almacenar una aceleración lineal para afectar a la velocidad otra angular para la rotación. Es necesario sobrecargar los operadores aritméticos de *Steering* porque será común tener que combinar salidas de varios algoritmos para producir una salida final. La definición de la clase es la siguiente.

```
struct Steering {
    public:
        Steering(const Ogre::Vector3& linear = Ogre::Vector3::ZERO,
                 Ogre::Real angular = 0.0f);

        const Ogre::Vector3& getLinear() const;
        Ogre::Real getAngular() const;
        bool isNone() const;

        void setLinear(const Ogre::Vector3 linear);
        void setAngular(Ogre::Real angular);
        void setNone();


        // Operadores
        bool operator == (const Steering& other);
        Steering& operator + (const Steering& other);
```

```
        Steering& operator += (const Steering& other);
        Steering& operator * (Ogre::Real f);
        Steering& operator *= (Ogre::Real f);
        Steering& operator / (Ogre::Real f);
        Steering& operator /= (Ogre::Real f);

    private:
        Ogre::Vector3 _linear;
        Ogre::Real _angular;
};
```

Para actualizar la información dinámica de un personaje necesitamos la salida del algoritmo de movimiento y el tiempo que ha pasado desde la última iteración del bucle de juego. Es posible utilizar las ecuaciones que se imparten en las clases de cinemática del instituto pero lo habitual es emplear integración de Newton Euler [14].

```
void Kinematic::update(const Steering& steering, Ogre::Real deltaT) {
    // Actualizamos posicion y orientacion
    _position += _velocity * deltaT;
    _orientation += _rotation * deltaT;

    // Actualizamos velocidad y rotacion
    if (steering.getLinear() == Ogre::Vector3::ZERO)
        _velocity = Ogre::Vector3::ZERO;
    else
        _velocity += steering.getLinear() * deltaT;

    _rotation += steering.getAngular() * deltaT;

    // Comprobamos si sobrepasamos la velocidad maxima
    if (_velocity.squaredLength() > _maxSpeed * _maxSpeed) {
        _velocity.normalise();
        _velocity *= _maxSpeed;
    }
}
```

Steering Behaviors

Los *Steering Behaviors* toman información del personaje y otros elementos del juego y producen una salida gracias al método *getSteering()*. La clase *SteeringBehavior* modela esto de forma genérica, si deseamos añadir un comportamiento nuevo, simplemente creamos una clase descendiente de esta e implementamos su método virtual puro. Podemos crear una jerarquía de algoritmos de movimiento de manera que los más complejos utilicen a los más simples para sus cálculos intermedios, lo veremos en la siguiente descripción.

Se han implementado más comportamientos de los necesarios para el juego de forma que otros usuarios puedan reutilizarlos en su sistema. Cada clase modela un comportamiento con un objetivo, los implementados son los siguientes:

Seek toma la posición del personaje y la de un objetivo que trata de buscar alcanzando la velocidad máxima poco a poco. Sigue acelerando hasta que sobrepasa el objetivo, entonces trata de acelerar en dirección opuesta. Es muy simple pero no es útil para llegar a un objetivo y detenerse.

Flee es el opuesto a *Seek*, trata de huir de un punto con la mayor aceleración posible.

Arrive es muy similar a *Seek* ya que busca alcanzar un punto a la velocidad máxima. No obstante, cuando está llegando a un radio predeterminado, comienza a frenar para detenerse en el destino.

Align recibe los datos dinámicos del personaje y una orientación objetivo que trata de igualar. Es útil para que un personaje mire hacia el mismo punto que otro de forma progresiva.

VelocityMatch en este caso lo que pretendemos igualar son las velocidades de dos entidades.

Pursue este comportamiento se diferencia de *Seek* y *Arrive* en que está preparado para perseguir un objetivo en movimiento. Se basa en predecir la posición del objetivo dentro de un tiempo determinado y dirigirse hacia ese punto a velocidad máxima.

Evade es el opuesto a *Pursue*, trata de huir de un perseguidor prediciendo su comportamiento.

Face está diseñado para que un personaje mire hacia otro, deja que *Align* haga el trabajo pero calcula primera la orientación deseada en función de los parámetros.

Wander es un algoritmo de movimiento que hace que el personaje vague por el escenario. Básicamente coloca un objetivo en un radio alrededor del personaje y va seleccionando puntos de forma aleatoria a los que llegar delegando en *Face* y *Seek*.

CollisionAvoidance se trata de un algoritmo de movimiento muy importante ya que evita que un personaje colisione con sus vecinos. Los enemigos lo utilizan para no atravesarse los unos a los otros mientras persiguen al personaje. Consiste en que si se detecta un compañero cerca, se huye en dirección opuesta empleado *Flee*.

FollowPath el algoritmo de movimiento básico para seguir un camino que nos haya devuelto el sistema de búsqueda de caminos. Simplemente se busca el próximo punto más cercano del camino y nos dirigimos a el empleando *Arrive*.

En **Sion Tower**, los algoritmos de movimiento más importantes son *CollisionAvoidance* y *FollowPath*. Por ello, pasamos a explicarlos en mayor profundidad a continuación. La definición de *CollisionAvoidance* es la siguiente:

```
1  class CollisionAvoidance: public Flee {
2      public:
3          Ogre::Real maxAcceleration;
4          std::vector<Enemy*> targets;
5          Enemy* myself;
6          Ogre::Real radius;
7
8          CollisionAvoidance(Kinematic* character,
9                             const std::vector<Enemy*>& targets,
10                            Enemy* myself);
11         void getSteering(Steering& steering);
12 };
```

El algoritmo para evitar las colisiones parece más complejo de lo que en realidad es. Simplemente recorremos la lista de enemigos comprobando que estén a una distancia prudencial. En caso negativo huimos en dirección opuesta. Si no colisionamos con ningún enemigo nos quedamos con el que, si no varía su rumbo actual, creamos que colisionará con mayor antelación y huimos de él. En caso de que no haya enemigos cercanos, no se hace nada. Su implementación se adjunta a continuación.

```
void CollisionAvoidance::getSteering(Steering& steering) {

    // Parametros
    Ogre::Real neededSeparation = 0.0f;
    Ogre::Real posDistance = 0.0f;
    Ogre::Real velDistance = 0.0f;
    Ogre::Real separation = 0.0f;

    Ogre::Real time = 0.0f;
    Ogre::Real minTime = Ogre::Math::POS_INFINITY;

    Ogre::Vector3 posDifference;
    Ogre::Vector3 velDifference;

    Kinematic* selTarget = 0;
    Ogre::Real selSeparation = 0.0f;
    Ogre::Real selNeededSeparation = 0.0f;
    Ogre::Vector3 selPosDifference;
    Ogre::Vector3 selVelDifference;

    // Para cada enemigo
    std::vector<Enemy*>::iterator i;
    for (i = targets.begin(); i != targets.end(); ++i) {
        if (*i != myself) {
            target = &((*i)->getKinematic());

            // 1. Comprobamos que no colisionamos

            // Minima distancia
            neededSeparation = 2 * radius;

            // Distancia al objetivo
            posDifference = target->getPosition() - character->getPosition
                ();
            posDistance = posDifference.length();

            // Si estamos colisionando, huimos
            if (posDistance <= neededSeparation) {
                Flee::getSteering(steering);
                return;
            }

            // 2. Comprobamos si colisionamos en un futuro
            velDifference = target->getVelocity() - character->getVelocity
                ();
            velDistance = velDifference.length();
            time = (posDifference.dotProduct(velDifference)) / (velDistance
                * velDistance);

            if (time > 0.5f)
                continue;

            // Solo si tenemos tiempo de colision y es minimo
            if (time > 0.0f && time < minTime)
                minTime = time;
```

```
                else
                    continue;

                // Calculamos la separacion en ese momento
                separation = posDistance - velDistance * minTime;

                // Si es el mas corto y va a colisionar lo guardamos
                if (separation <= neededSeparation) {
                    selTarget = target;
                    selSeparation = separation;
                    selNeededSeparation = neededSeparation;
                }
            }
        }

        if (selTarget == 0)
            return;

        // Cambiamos la direccion de la velocidad

        // 1. Tomamos un punto lo mas lejano posible del punto de colision
        // 2. Cambiamos la velocidad actual para dirigirnos a ese punto

        // La nueva posicion del personaje al cabo del tiempo
        Ogre::Vector3 charPos = character->getPosition() + character->
            getVelocity() * minTime;
        Ogre::Vector3 targetPos = selTarget->getPosition() + selTarget->
            getVelocity() * minTime;

        // Direccion desde el objetivo al personaje
        posDifference = charPos - targetPos;
        posDifference.normalise();

        // Nueva posicion deseada: en la direccion posDifference
        posDifference = charPos + posDifference * (selNeededSeparation -
            selSeparation);

        // Direccion deseada para llegar al punto seguro
        Ogre::Vector3 desiredDir = posDifference - character->getPosition();
        desiredDir.normalise();
        desiredDir = desiredDir * character->getVelocity().length();

        // Aceleracion deseada
        steering.setLinear(desiredDir - character->getVelocity());
    }
```

La clase *FollowPath* simplemente almacena la información dinámica del personaje y el camino de puntos como puede verse en el siguiente fragmento de código.

```
class FollowPath: public Arrive {
    public:
        NavigationMesh::PointPath* path;
        Ogre::Real pathOffset;

        FollowPath(Kinematic* character, NavigationMesh::PointPath* path);
```

```
        void getSteering(Steering& steering);

    protected:
        Ogre::Vector3 findTargetInPath();
};
```

El mecanismo consiste en buscar el punto del camino más cercano al personaje y tratar de llegar hasta él utilizando *Arrive*. Encontrar el punto más cercano consiste en recorrer todos los puntos e ir calculando y comparando distancias. A continuación se adjunta el fragmento completo.

```
void FollowPath::getSteering(Steering& steering) {
    // 1. Calculamos el target para delegar en Arrive
    target = new Kinematic(findTargetInPath());

    // 2. Delegamos en Arrive
    Arrive::getSteering(steering);

    delete target;
}

Ogre::Vector3 FollowPath::findTargetInPath() {
    Ogre::Real minDistance = Ogre::Math::POS_INFINITY;
    NavigationMesh::PointPath::iterator closestPointIt = path->begin();

    // Recorremos la lista de puntos buscando el mas cercano
    for (NavigationMesh::PointPath::iterator i = path->begin(); i != path->
        end(); ++i) {
        // Calculamos la nueva distancia
        Ogre::Vector3 direction = character->getPosition() - *i;

        if (direction.squaredLength() < minDistance) {
            minDistance = direction.squaredLength();
            closestPointIt = i;
        }
    }

    NavigationMesh::PointPath::iterator targetIt = closestPointIt;
    ++targetIt;

    if (targetIt == path->end())
        return *closestPointIt;

    return *targetIt;
}
```

Para conocer detalles adicionales sobre la implementación de los *Steering Behaviors* es recomendable acudir al propio código fuente en la forja de RedIRIS.

Máquina de estados para los enemigos

Todos los enemigos de **Sion Tower** se comportan de manera similar, se diferencian en atributos como velocidad, energía vital o fuerza. Internamente se implementa una sencilla máquina de estados de forma

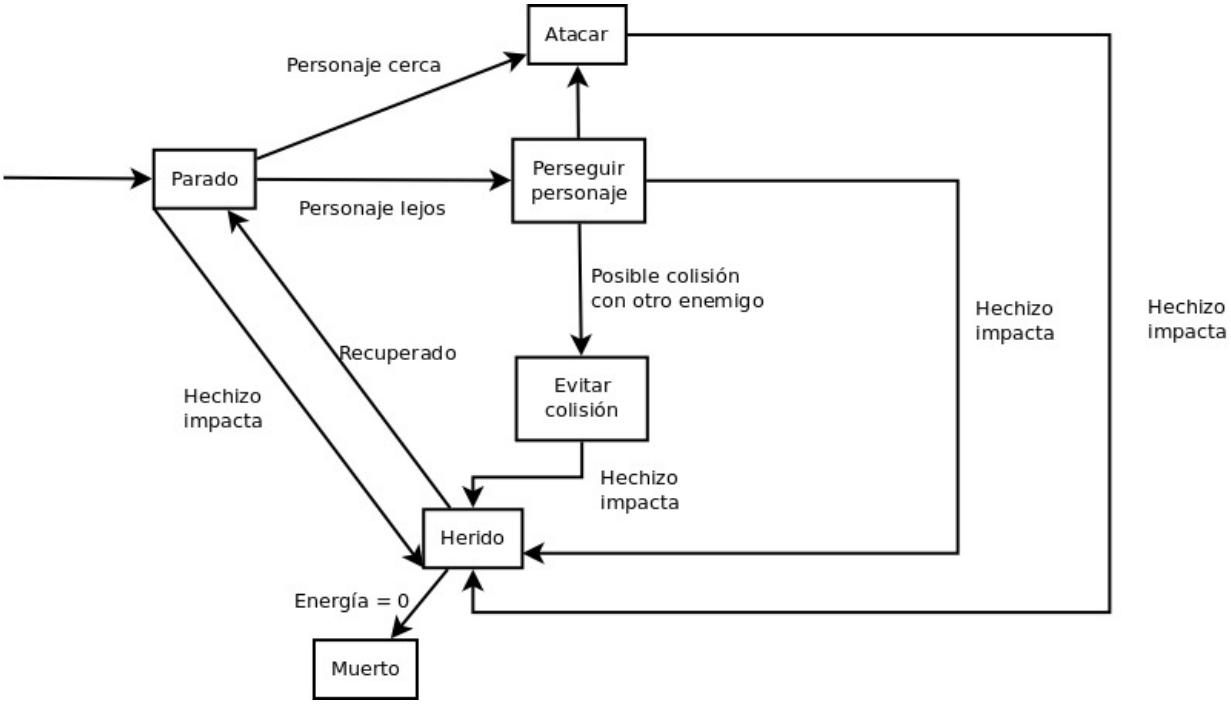


Figura 4.52: Máquina de estados para los enemigos

que utilizamos algoritmos de movimiento para resolver cada comportamiento combinados con consultas a la búsqueda de caminos. El diagrama de estados puede observarse en la figura 4.52.

4.5. Pruebas

Diseñar casos de prueba para un videojuego como **Sion Tower** es una tarea complicada ya que estamos simulando continuamente un mundo lleno de elementos que interactúan entre sí y hay métodos que se ejecutan más de 60 veces en un sólo segundo. Por supuesto, las pruebas son absolutamente necesarias, ya que nos ayudan a desarrollar software de mayor calidad.

La mayoría de módulos han podido ser probados de forma independiente en pruebas unitarias de caja blanca. Entre ellos se encuentran las clases del sistema de detección de colisiones, el gestor de perfiles, el gestor de niveles o la búsqueda de caminos. No obstante, existen módulos que debían trabajar en conjunción con otros por lo que sólo han sido sometidos a pruebas de integración. Este caso se da en los algoritmos de movimiento, el subsistema de audio o la gestión de estados de juego.

Más adelante se llevaron a cabo pruebas sobre la jugabilidad de **Sion Tower**. Colaboradores externos al probaron el juego y ofrecieron sus opiniones. Los aspectos a analizar eran la dificultad, el control del personaje, velocidades, fuerzas y otros parámetros relacionados con el balanceo. Finalmente se llevaron a cabo pruebas sobre la interfaz, su intuitividad y resistencia a valores inadecuados.

Durante las pruebas unitarias y de integración se utilizó software de depuración como *DNU Debugger* [37] y *Valgrind* [38]. El primero nos ayuda a detectar dónde se producen fallos en tiempo de ejecución y nos permite monitorizar el valor de variables y otros parámetros. El segundo vigila la memoria y nos

permite localizar puntos en los que no liberemos de forma correcta los recursos así como otros momentos en los que accedamos a memoria basura.

El proceso, de forma resumida, fue el siguiente:

1. Pruebas unitarias durante la fase de implementación, tras finalizar cada módulo.
2. Pruebas de integración a medida que se completaban pequeños subsistemas que debían colaborar.
3. Pruebas de jugabilidad con las primeas versiones usables del juego y empleando la ayuda de colaboradores externos.
4. Pruebas de interfaz una vez se finalizó el desarrollo del juego.

4.5.1. Pruebas unitarias

Las pruebas unitarias se realizaron junto a la fase de implementación, a medida que se finalizaban los módulos. Se optó por un enfoque estructural con pruebas de caja blanca. Se buscaba tomar todas las bifurcaciones posibles, o al menos las más propensas a fallos, en cada módulo testado. De esta forma todas las sentencias se ejecutarían al menos una vez y los posibles fallos saldrían a relucir con mayor facilidad. Se prestó especial atención a problemas relacionados con el redondeo de números reales ya que se utilizan de forma intensiva para representar posiciones, rotaciones, aceleraciones y otros parámetros imprescindibles. Así mismo, también se procuró que el sistema no accediese a memoria no inicializada a través de punteros inválidos.

El mayor número de errores encontrados estaban relacionados con los accesos inválidos a memoria. En este caso, el depurador del proyecto GNU y *Valgrind* fueron de gran ayuda. Así mismo, el sistema de detección de colisiones presentó varios defectos en algunos de los tests de colisión por la complejidad que entrañaban.

4.5.2. Pruebas de integración

A medida que el desarrollo de varios módulos de un mismo subsistema finalizaba, se procedían a realizar pruebas de integración entre dichos módulos empleando una aproximación de caja negra. Interesaba que el sistema realizara la tarea para la que había sido diseñado de forma correcta. Cuando todos los módulos del sistema estuvieron listos se realizaron pruebas adicionales de integración a mayor escala. No sólo entre los módulos de un mismo subsistema, sino el funcionamiento conjunto de varios subsistemas.

De nuevo, el mayor número de problemas fue localizado en la gestión de memoria. Cuando se destruía un estado de juego para crear otro había elementos que no se liberaban de forma adecuada o que no se inicializaban correctamente y provocaban problemas a posteriori. *Valgrind* y *GDB* fueron extremadamente útiles en dicho momento.

4.5.3. Pruebas de jugabilidad

Una vez se compilaron las primeras versiones usables de **Sion Tower** se pidió a colaboradores externos que probaran el videojuego. Tras estas sesiones se les preguntó por aspectos como la capacidad de respuesta del control, la comodidad de juego, dificultad y otros parámetros. Posteriormente, se analizaron las respuestas de todos los colaboradores y se llevó a cabo un proceso de balanceo para ajustar la dificultad.

La mayoría de colaboradores encontró el juego bastante complicado. Cuando el personaje es rodeado tiene poca escapatoria posible y comienza a recibir golpes sin parar. Esto se solucionó haciendo a los enemigos más lentos, así el jugador podía esquivarlos con mayor facilidad y evitar ser rodeado en la mayoría de ocasiones. Se reforzó al protagonista con una mayor cantidad de energía vital y potenciando el poder destructivo de sus hechizos. Se suavizó el control de la cámara para que fuera más ágil y no fueran necesario gestos con el ratón tan bruscos. Por último, se añadieron atajos de teclado para que el jugador pudiese seleccionar de forma más cómoda los hechizos. Tras aplicar los cambios, los colaboradores se mostraron mucho más satisfechos con el juego.

4.5.4. Pruebas de interfaz

Tras completar las pruebas de jugabilidad, se procedió a realizar pruebas sobre la interfaz del juego. Tanto en los menús como en la propia pantalla de juego. Estas pruebas consistieron en medir los tiempos de carga, comprobar que la interfaz fuera intuitiva y que no aceptaba valores erróneos. Se probó a introducir nombres inválidos como perfil, tratar de seleccionar un nivel bloqueado y utilizar hechizos para los que no disponíamos de maná suficiente.

Se demostró que la interfaz del juego era bastante sólida e intuitiva. Prácticamente no era necesario un manual de usuario para que cualquiera pudiera comprenderla y navegar por ella. El sistema de perfiles no admite nombres erróneos y la pantalla de selección de nivel no nos permite jugar a un nivel bloqueado para un perfil dado.

Capítulo 5

Conclusiones

Tras haber explicado todo el proceso de desarrollo de **IberOgre** y **Sion Tower** a lo largo del presente documento, en esta sección haremos una valoración acerca del proyecto. En primer lugar hablaré de como me ha afectado en el plano personal y académico para después tratar los aspectos técnicos. Finalmente finalizaremos con las posibles ampliaciones que podría sufrir en el futuro.

5.1. Conclusiones personales

Con **IberOgre** y **Sion Tower** he adquirido una gran cantidad de conceptos y me he enriquecido muchísimo como persona y desarrollador. Si bien ya había trabajado de forma independiente en algún videojuego sencillo o proyecto de distinta índole, este sin duda era el reto de mayor envergadura al que me había enfrentado jamás. De ahí que la fase de aprendizaje fuese tan larga al comienzo del proyecto aunque después se extendiera durante todo este tiempo.

En los siguientes puntos repasaré de forma superficial lo que he aprendido:

1. **Juegos 3D**: hasta el momento había desarrollado juegos sencillos en dos dimensiones pero no conocía el mundo de las tres dimensiones. Consideré que el Proyecto Fin de Carrera me brindaba una oportunidad excelente para aprender y me decidí a intentarlo. Es una aproximación muy diferente, atractiva y llena de retos. Es necesario conocer todo un conjunto de técnicas y conceptos totalmente nuevos y me ha resultado de lo más interesante.

2. **Nuevas bibliotecas**: he hecho uso de muchas bibliotecas que no conocía y he necesitado aprender a utilizarlas correctamente. Entre estas bibliotecas se encuentran la propia OGRE3D, MYGUI, BOOST o PUGIXML. Anteriormente había utilizado LIBSDL MIXER pero nunca me había visto obligado a integrarla dentro de un sistema más grande como es la gestión de recursos de OGRE3D. BOOST es una biblioteca de lo más versátil y potente que complementa en muchos aspectos a la estándar de C++. De hecho, muchas de sus características aparecerán en el nuevo C++ 0x.

3. **Matemáticas para videojuegos**: no sólo he tenido que repasar conceptos matemáticos de geometría del espacio y álgebra sino que he tenido que incorporar otros nuevos. Tanto de cara a **IberOgre** como para **Sion Tower** me he visto obligado a aplicar estos conceptos con el objetivo de buscar soluciones a problemas de programación.

4. **Lenguaje C++**: desde el segundo curso de Ingeniería Técnica en Informática de Gestión he estado utilizando C++ no sólo dentro del ámbito estrictamente académico. No obstante, desarrollar **Sion Tower** me ha servido para profundizar en los detalles del lenguaje y aprender a utilizarlos a mi favor.

5. **Python**: este lenguaje de scripting orientado a objetos se ha mostrado extremadamente útil a la hora de desarrollar pequeños programas auxiliares. Por ejemplo, lo he utilizado en el script que soluciona el problema de la escala en la exportación de modelos tridimensionales y en el que extrae las cadenas traducibles de plantillas de interfaz. Es un lenguaje extremadamente sencillo y fácil de aprender pero muy potente y que agiliza el desarrollo de herramientas simples.

6. **Optimización**: los videojuegos son sistemas complejos que, de no prestar especial atención, podrían hacer un consumo irracional de recursos. Es necesario aplicar técnicas de optimización tanto en términos de uso de procesador como de consumo de memoria. Este proyecto me ha servido para conocer varias de estas técnicas y algoritmos concretos cuyo rendimiento es superior al de otros en momentos concretos. Por ejemplo, en primer lugar empleé el algoritmo A* para la búsqueda de caminos aunque la precomputación con Floyd probó ser más eficiente.

7. **Técnicas de IA**: hasta el momento conocía algunas técnicas de inteligencia artificial sobre teoría de juegos, búsqueda de caminos o autómatas para modelar comportamientos. No obstante, no había aplicado dichas técnicas al mundo de las tres dimensiones. En **Sion Tower** se realiza una búsqueda de caminos a partir de una malla definida en tiempo de diseño. Además, los enemigos hacen uso de algoritmos de movimiento (*Steering Behaviors*) que desconocía hasta este momento.

8. **Diseño de un videojuego**: los videojuegos que había desarrollado con anterioridad con contaban con un documento de diseño en el que se detallara la forma de jugar, personajes, historia, etc. En este caso ha sido muy necesario ya que ha ayudado al proceso de análisis y diseño. Además, los colaboradores han podido conocer las necesidades del proyecto en cuanto a recursos artísticos se refiere. Los integrantes del equipo conocíamos en todo momento el estilo de juego y la apariencia que debía tener **Sion Tower**.

9. **Trabajo en equipo**: en **IberOgre** se ha trabajado junto a la comunidad en todo momento. Los lectores enviaban sus opiniones y éstas debían ser tenidas en cuenta. En **Sion Tower** el trabajo en equipo se hizo mucho más evidente ya que colaboramos en todo momento seis personas especializadas en disciplinas muy distintas. Hubo que realizar labores de coordinación, comunicación y resolución de conflictos (sobre todo relacionados con los formatos empleados). Una experiencia muy enriquecedora y que, sin duda, me ayudará en mi futuro profesional.

10. **Redacción en MediaWiki**: hasta el momento mi relación con el motor *MediaWiki* se había limitado a ediciones muy esporádicas en Wikipedia que no requerían conocimientos de la sintaxis que se utilizaba. Me he visto obligado a conocer la sintaxis de redacción en *Wikimedia* para redactar correctamente los artículos.

11. **Trabajo en wikis**: no sólo basta con conocer la sintaxis de *Wikimedia* para poder participar en la confección de una wiki. Existen toda una serie de convenciones de escritura, comportamiento, nomenclatura y estructuras que deben ser conocidas.

12. **Aplicación de conocimientos**: el desarrollo de este proyecto me ha sido especialmente útil para poner en práctica aquellos conocimientos adquiridos durante la Ingeniería Técnica. Sobre todo me refiero a aquellos relacionados con la Ingeniería del Software.

5.2. Conclusiones técnicas

En **IberOgre** y **Sion Tower** se han cumplido con los objetivos propuestos en el capítulo introductorio de esta memoria de Proyecto Fin de Carrera. En la plataforma de aprendizaje se ha conseguido:

- Se ha creado contenido organizado en bloques temáticos tal y como se propuso: introducción, matemáticas para videojuegos, OGRE3D, otras tecnologías y videojuegos desarrollados con OGRE3D.
- La navegación es intuitiva y los artículos están ordenados de forma aproximada en dificultad ascendente.
- Es posible adquirir los conocimientos de geometría del espacio necesarios para desarrollar juegos en 3D.
- Se han cubierto los aspectos básicos del uso del motor de renderizado OGRE3D.
- Otras tecnologías enfocadas al desarrollo de videojuegos han sido explicadas a través de varios artículos.
- Los artículos están dotados de un enfoque práctico gracias a los ejemplos finales y a los pequeños fragmentos de código intermedios.
- Varios usuarios han mostrado interés, han colaborado con el proyecto ya sea mediante correcciones, artículos, sugerencias o ayudando a difundir la plataforma.

En el videojuego **Sion Tower** hemos conseguido:

- Construir un videojuego completo empleando los conocimientos de **IberOgre** y otros conceptos profusamente documentados.
- Creación de un motor orientado a la creación de contenido. Es posible crear nuevos niveles complejos sin necesidad de tocar una sola línea de código.
- Aplicación multilenguaje gracias a GETTEXT.
- Creación de un videojuego que entretiene gracias a las impresiones que han prestado los colaboradores a lo largo de todas las fases del desarrollo.
- Implementación de un motor modular fácilmente ampliable. Varios subsistemas han sido liberados de forma independiente y están siendo utilizados por otros usuarios.

Se han generado estadísticas sobre el uso del repositorio *Subversion* empleando la herramienta libre *StatsSVN* [35]. En total el proyecto está compuesto por 619.000 líneas de código aunque entre ellas se encuentran las distintas ramas (con código duplicado) y líneas de documentación (esta memoria, por ejemplo). Cabe destacar que se han realizado más de 500 commits, lo que permite volver hacia atrás de forma sencilla. Es posible acceder al informe de estadísticas desde la siguiente dirección.

`http://siondream.com/iberogre-siontower-statsvn`

5.3. Trabajos futuros

Es cierto que los objetivos que nos marcamos al comienzo del desarrollo han sido cumplidos satisfactoriamente, no obstante se han detectado puntos en los que el proyecto podría mejorar. A continuación hacemos una lista de las posibles mejoras de **IberOgre**.

- Consolidación de una comunidad de redactores y lectores para la plataforma de pruebas.
- Nuevos artículos sobre videojuegos desarrollados con OGRE3D en los que el propio desarrollador comente la experiencia del desarrollo y proporcione o enlace documentación de interés.
- Artículos en la sección OGRE3D que documenten los sistemas de *shading* con los que cuenta el motor.
- Documentar alguna biblioteca o sistema para incluir juego en red en la sección de otras tecnologías.

En **Sion Tower** podrían incluirse las siguientes mejoras:

- Nuevos hechizos: paralización, teletransporte para huir, colocación de trampas, etc.
- Mayor relevancia de los puntos de experiencia, podrían permitir subir de nivel, mejorar la energía y el maná así como desbloquear hechizos.
- Enemigos adicionales como arañas gigantes o criaturas voladoras con comportamientos diferentes.
- Nuevos niveles de mayor tamaño que permitan la aplicación de técnicas de búsqueda de caminos jerárquicas.
- Escenas narrativas mediante animaciones que fueran contando la historia entre niveles.

Software utilizado

En esta sección hablaremos de las herramientas utilizadas durante el desarrollo de **IberOgre** y **Sion Tower**. Para cada herramienta ofreceremos una pequeña descripción de sus funcionalidades y adjuntaremos las razones por las cuales ha sido elegida para el desarrollo frente a sus competidoras. No haremos mención a las bibliotecas empleadas ya que han sido profusamente comentadas a lo largo de la implementación de ambas partes del proyecto.

Subversion

Subversion [36] es un sistema de control de versiones libre cuya primera versión fue lanzada en el año 2000 por CollabNet. Es compatible en mayor parte con su predecesor *CVS*. *Subversion* nos permite contar con una copia de seguridad tanto de los ejemplos de **IberOgre** como del código de **Sion Tower** en todo momento. Gracias a esta herramienta podemos guardar un historial de todas las versiones de los ficheros fuente del proyecto así como deshacer cambios en caso de que fuera necesario. Con *SVN* conseguimos acceso al código del proyecto desde cualquier equipo. Además, pone a disposición de cualquier interesado el código fuente de forma sencilla.

Existen otros sistemas de control de versiones como *Git* o *Mercurial*. Se ha elegido *Subversion* por su sencillez y por mostrarse suficientemente potente para gestionar el código de un proyecto de un sólo programador. En cualquier caso, su uso era obligatorio según la normativa del V Concurso Universitario de Software Libre (más sobre el concurso en el apéndice Comunidad y difusión).

GNU GCC Compiler

GCC [47] es el compilador del proyecto GNU y se encarga de traducir nuestro código C++ a lenguaje máquina para poder ejecutarlo posteriormente. El proyecto se inició en 1987 y actualmente está disponible para multitud de arquitecturas como móviles Symbian o consolas PlayStation 2.

GCC fue desarrollado inicialmente para soporta el lenguaje C aunque actualmente forma el GNU toolchain y es compatible con C++, Fortran, Pascal, Objective-C, Java y Ada entre otros. Ha sido elegido por ser el compilador libre compatible con el lenguaje más ampliamente usado, estable y eficiente.

Make

Make [16] es una utilidad para automatizar el proceso de programas y bibliotecas a partir de su código fuente a partir de ficheros con una sintaxis especial llamados *makefile*. Estos ficheros le indican a la

herramienta cómo ha de ser compilado el software.

Make acelera el proceso de compilación ya que no vuelve a generar los ficheros objetos ya creados cuyos fuentes no han sido modificados. Además, permite limpiar directorios y gestionar distintos modos de compilación (modo depuración o lanzamiento, por ejemplo). En nuestro proyecto, *Make* también es utilizado para generar la documentación escrita en LaTeX.

GNU Debugger

GDB [37] es como se conoce popularmente al depurador de código del proyecto GNU. Es un depurador destinado a sistemas Unix compatible con varios lenguaje entre los que se encuentran, por supuesto, C y C++. Esta herramienta nos permite marcar puntos de interrupción en el programa y examinar la pila del programa cuando éste ha terminado su ejecución de manera abrupta. Es posible acceder al valor de variables en todo momento para tratar de averiguar las fuentes de los fallos del sistema.

El depurador básicamente nos ayuda a saber qué ocurre exactamente durante la ejecución de nuestra aplicación. En **Sion Tower** ha resultado ser una herramienta imprescindible a la hora de identificar y solucionar problemas en tiempo de ejecución. Sin duda, es una herramienta que colabora en el aumento de la calidad del software si sabe utilizarse correctamente. Cuenta con un sinfín de opciones y funcionalidades que no han sido explotadas en el desarrollo de este proyecto.

Valgrind

Valgrind [38] es una herramienta para la depuración del uso de la memoria, detección de pérdidas de memoria y profiling. Esto nos permite identificar puntos en los que se accede a memoria aún no inicializada y que probablemente contenga basura. También es útil para saber si nuestra aplicación no la libera memoria utilizada por los objetos cuando estos ya han sido destruidos. El profiling consiste en monitorizar cuánto tiempo permanece el código en ejecución dentro de cada función, esto nos ayuda a identificar posibles cuellos de botella. Su nombre viene de la mitología nórdica y se refiere a la entrada del Valhala.

Es compatible con sistemas GNU/Linux y Mac OS X. Ha resultado extremadamente útil en aquellos puntos a los que *GDB* no era capaz de llegar. No obstante, al ser una herramienta de análisis dinámico, la ejecución de **Sion Tower** se ralentizaba en exceso aún trabajando con equipos muy potentes.

Vim

Vim [57] es un editor de textos libres creado en 1991 para el computador Amiga. *Vim* significa *Vi IMproved* ya que es una versión mejorada del anterior editor *Vi*. Es editor compatible con sistemas GNU/Linux extremadamente ligero que no precisa de entorno gráfico ya que puede ejecutarse desde la terminal. Está orientado a la productividad y cuenta con decenas de combinaciones de teclas para acelerar la escritura de código.

Es muy completo, cuenta con corrector ortográfico, resaltado de sintaxis para decenas de lenguajes e ingentes opciones para personalizar el entorno. Incluso es posible ampliar sus capacidades mediante

extensiones. Ha sido utilizado para el proyecto por su ayuda a la productividad y ligereza. Puede observarse *Vim* siendo ejecutado en una terminal en la figura 5.1.

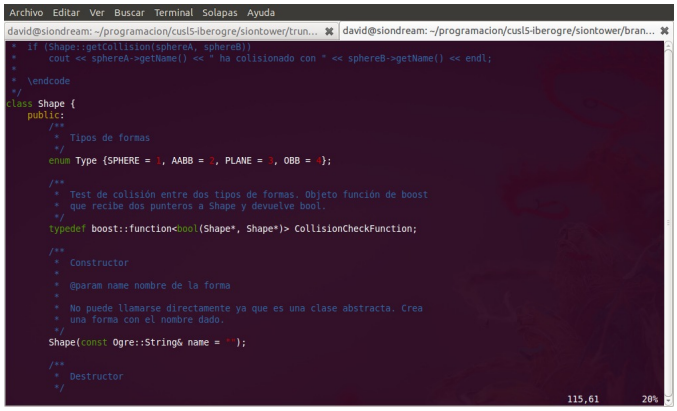


Figura 5.1: Editor de textox Vim

LaTeX

LaTeX[40] es lenguaje de marcado y un sistema de creación de documentos especialmente orientado al mundo científico y técnico. El lenguaje *Tex* fue concebido por Donald Knuth durante los 80 y en 1984 Leslie Lamport creó LaTeXcomo un framework para trabajar con cartas, libros y otro tipo de textos.

Los resultados que producen LaTeXson coherentes, ordenados y muy limpios. Si bien aprender su uso puede ser complejo, los resultados son de una enorme calidad si los comparamos con los documentos que producen los editores convencionales como *Microsoft Word* o *LibreOffice Writer*. Por esta calidad y posibilidad de automatizar formatos ha sido elegido LaTeXpara la redacción de la documentación del proyecto.

Doxygen

Doxygen [11] es una herramienta de documentación automática de código. Mediante la inclusión de comentarios especiales dentro de los ficheros de cabecera de nuestro proyecto, es capaz de generar documentación con una apariencia atractiva tanto en formato HTML como en LaTeXo muchos otros. Es compatible con los lenguajes C, C++, C#, Fortran, Java, Objective-C, PHP, Python, IDL y algunos más.

Con *Doxygen* se produce una documentación legible por usuarios (con los conocimientos necesarios) u otros miembros del equipo. Es posible incluir diagramas de colaboración y herencia gracias a su uso de graphviz. Ha sido elegida como herramienta de documentación de código por su sencillez de uso, limpieza y por su amplia aceptación.

Blender

Blender [20] es una herramienta libre de modelado y animación en tres dimensiones multiplataforma. Es compatible con Windows, Mac OS X, GNU/Linux y otros sistemas operativos. Cuenta con funcionalidades avanzadas de modelado 3D, mapeado UV para las texturas, renderizado, texturizado, animación basada en esqueletos, efectos de partículas y simulaciones físicas.

Integra un motor de físicas y colisiones y proporciona una API compatible con Python para programar complementos e incluso videojuegos completos gracias al *Blender Game Engine*. Ha sido utilizado en el desarrollo de cortometrajes libres como Big Buck Bunny aunque también se emplea de forma profesional en la industria de la publicidad.

Ha sido elegido para ser empleado en el proyecto por ser el editor 3D libre por excelencia y por haber sido probado en entornos de producción muy importantes. Además de ser utilizado para diseñar los elementos del escenario, se emplea como editor de niveles completo.

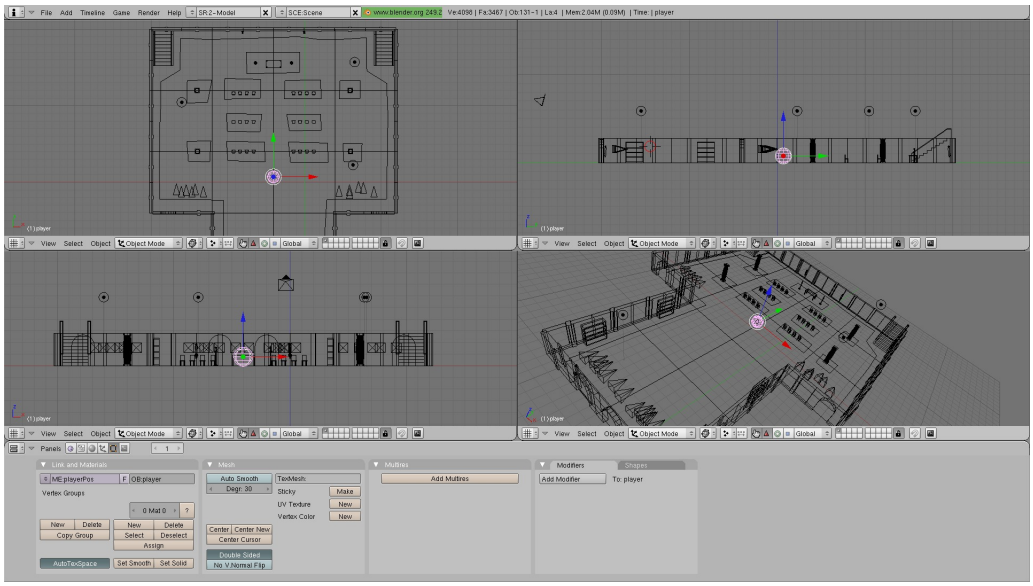


Figura 5.2: Herramienta de modelado y animación 3D Blender

GIMP

GIMP [17] es el editor de imágenes libre del proyecto GNU, de hecho su nombre es un acrónimo de *GNU Image Manipulation Program*. Es multiplataforma y está disponible para Windows, GNU/Linux y Mac OS X. No es comparable a soluciones privativas como *Adobe Photoshop* pero es capaz de realizar operaciones bastante avanzadas de forma sencilla. Su interfaz puede verse en la figura 5.3.

Ha sido utilizado en **Sion Tower** para trabajar con las texturas del escenario. Hemos elegido esta herramienta por contar con una licencia libre, ser lo suficientemente potente para nuestras necesidades y estar disponible en varias plataformas (incluida GNU/Linux).

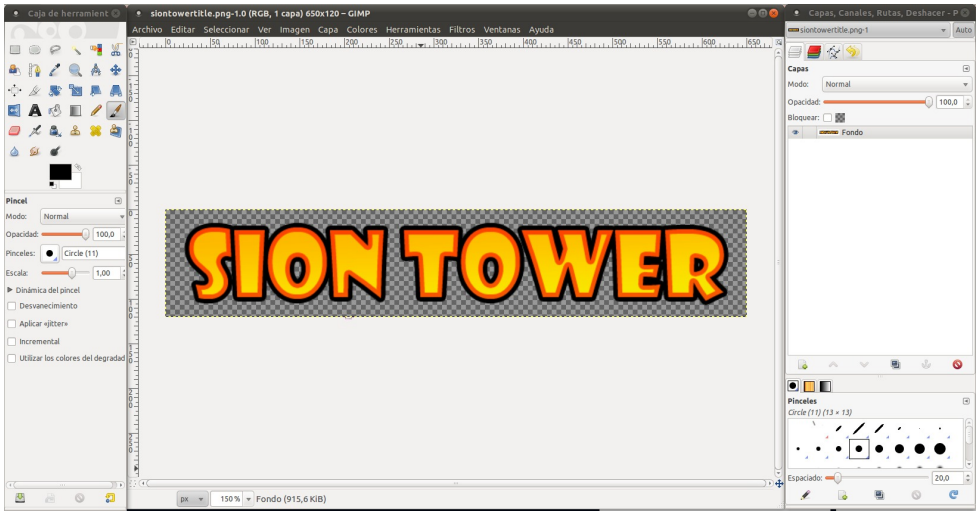


Figura 5.3: Editor gráfico Gimp

Inkscape

Inkscape [21] es una herramienta libre multiplataforma para trabajar con gráficos vectoriales. Estos diseños no están basados en píxeles sino en elementos geométricos como vectores o figuras sencillas. Esto permite un escalado ilimitado sin pérdida de calidad. Inkscape trabaja con ficheros en formato *SVG*.

Ha sido utilizado tanto en **IberOgre** como en **Sion Tower** para trabajar con gráficos bidimensionales tales como elementos de interfaz o diagramas más vistosos. Es prácticamente la única alternativa libre a la altura del software comercial existente.

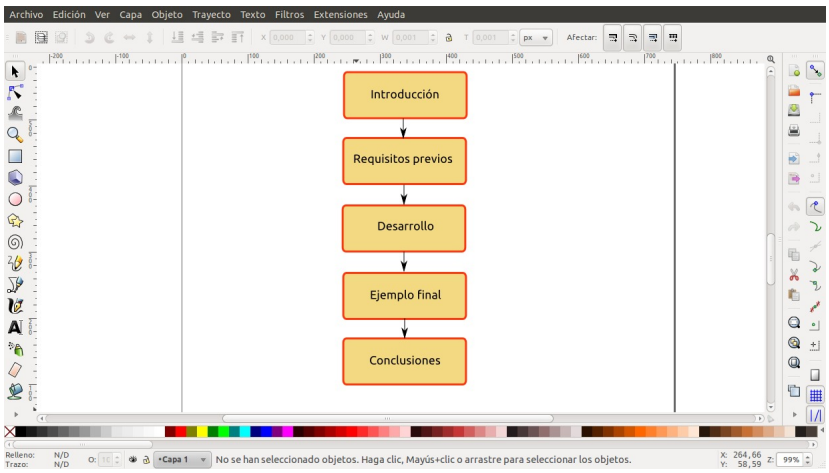


Figura 5.4: Herramienta de gráficos vectoriales Inkscape

MyGUI Layout Editor

La biblioteca de interfaces MyGUI utiliza unas plantillas en formato *XML* para definir los elementos de las pantallas como paneles, botones o imágenes. Es posible escribir estas plantillas manualmente pero es una tarea ardua. Afortunadamente existe *MyGUI Layout Editor*, una herramienta para diseñar interfaces y guardarlas en dicho formato de manera que el motor pueda cargarlas de forma sencilla posteriormente.

Es multiplataforma, libre y trabaja de una manera muy limpia ya que no genera código de ningún tipo, únicamente produce un *XML*. Es muy intuitiva y gracias a ella se han diseñado todas las pantallas de **Sion Tower**. La última versión cuenta con la siguiente interfaz (figura 5.5).

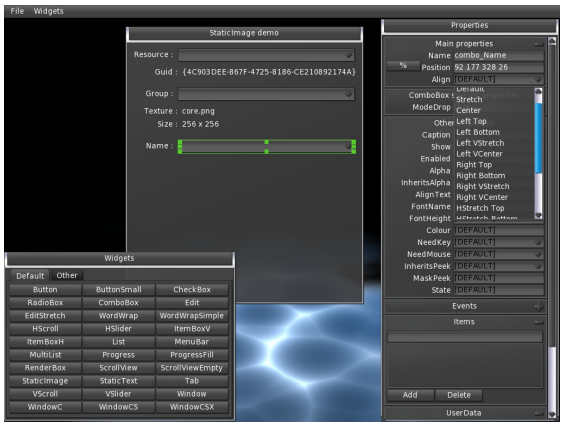


Figura 5.5: Editor de plantillas MyGUI Layout Editor

Particle Editor

Los sistemas de partículas de Ogre3D se definen en scripts con una sintaxis especial. Al igual que ocurre con la interfaz, escribirlos a mano es muy pesado y no permite ver los resultados hasta que se inicie el juego. *Ogre Particle Editor* es un editor de sistemas de partículas para Ogre3D. Nos permite crear partículas y modificar sus parámetros viendo los resultados en tiempo real.

Una vez hayamos terminado, guarda el resultado en un fichero con la sintaxis anteriormente mencionada. Esto acelera enormemente el proceso de creación de efectos de partículas y por ello esta herramienta ha sido utilizada en **Sion Tower**.

Audacity

Audacity [4] es un editor de audio libre y multiplataforma. Nos permite grabar y modificar audio con un gran número de opciones y variantes. Fue lanzado por primera vez en mayo del 2010 y actualmente cuenta con más de 72 millones de descargas. Si bien es cierto que carece de herramientas avanzadas de edición de sonido, para nuestras necesidades era la herramienta ideal.

Audacity ha sido utilizado en **Sion Tower** para convertir y retocar los efectos de sonido y las pistas de la banda sonora que envían los correspondientes artistas. Su interfaz puede observarse en la figura 5.7.

Figura 5.6: Editor de sistemas de partículas Particle Editor

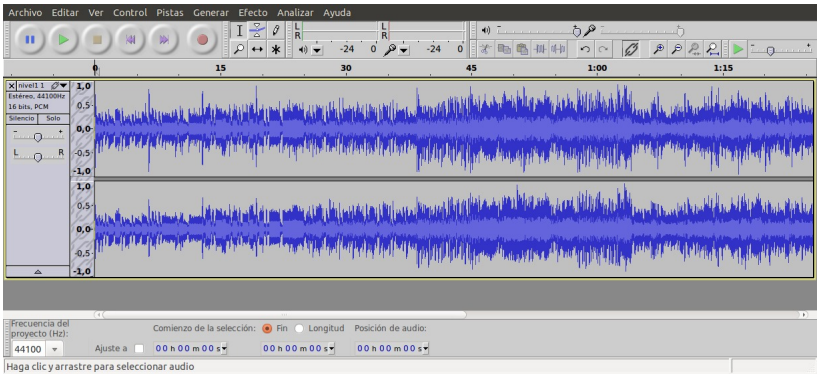


Figura 5.7: Editor de audio Audacity

XVidCap

XVidCap [25] es una utilidad libre para capturar la pantalla en vídeo y audio. Suele utilizarse para crear tutoriales de procesos complejos aunque en **Sion Tower** se emplea para tomar los vídeos con el objetivo de difundir el proyecto. Puede capturar el escritorio completo, ventanas concretas o un área determinada. Nos permite calibrar el formato de vídeo y audio, la calidad así como el número de cuadros por segundo.

Se trata de la herramienta para capturar vídeo del escritorio más eficiente y que menos peso ejercía sobre el rendimiento general del sistema. Su interfaz es muy sencilla y únicamente consta de la barra que puede verse en la figura 5.8.

Figura 5.8: Capturador de pantalla XvidCap

OpenShot Video Editor

OpenShot Video Editor [31] es un sencillo pero potente editor de vídeo para sistemas GNU/Linux. Soporta multitud de formatos de audio y vídeo gracias a su uso de la biblioteca FFMPEG. Creamos el vídeo añadiendo y organizando recursos sobre las pistas que deseemos. Es posible incluir transiciones, subtítulos, carteles y otros efectos. Podemos configurar la exportación de vídeo a través de numerosos parámetros e incluso se proporcionan perfiles para subir vídeos a conocidos servicios como *Youtube*.

Los vídeos de **Sion Tower** creados con *XVidCap* son procesados con *OpenShot Video Editor* y procesados para ser subidos a algún servicio de vídeos por streaming con el objetivo de difundir el proyecto. Es la herramienta libre más potente y sencilla disponible para GNU/Linux.

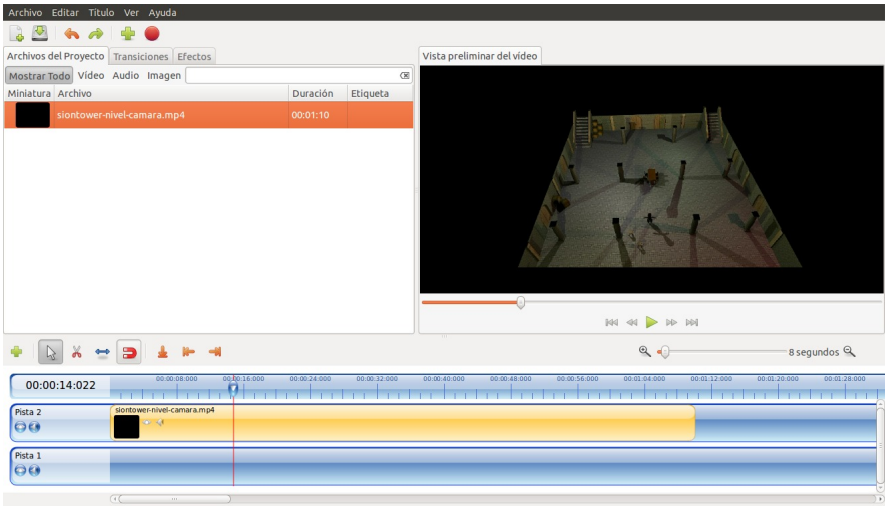


Figura 5.9: Editor de vídeo OpenShot

Planner

Planner [32] es una herramienta libre para entornos GTK que nos permite planificar y realizar un seguimiento de cualquier proyecto. Se definen tareas sobre un calendario especificando sus dependencias, duraciones y fechas límite. Posteriormente, se crean y asignan recursos (materiales o humanos) a dichas tareas. Finalmente se obtiene un diagrama de Gantt con la organización temporal del proyecto. Las planificaciones pueden guardarse en una base de datos *postgresql* o en un sencillo formato *XML*. Incluso puede exportarse a *HTML* para que pueda ser visualizada desde cualquier navegador.

Toda la planificación del proyecto que puede verse en el capítulo 2 ha sido creada con *Planner*. Es una herramienta muy intuitiva y mucho más ligera que alternativas como *GanttProject*.

BOUML

BOUML [6] es una herramienta libre para diseñar diagramas siguiendo la notación *UML*. Cuenta con utilidades de generación automática de código a partir de los diagramas en lenguajes C++, Java, PHP,

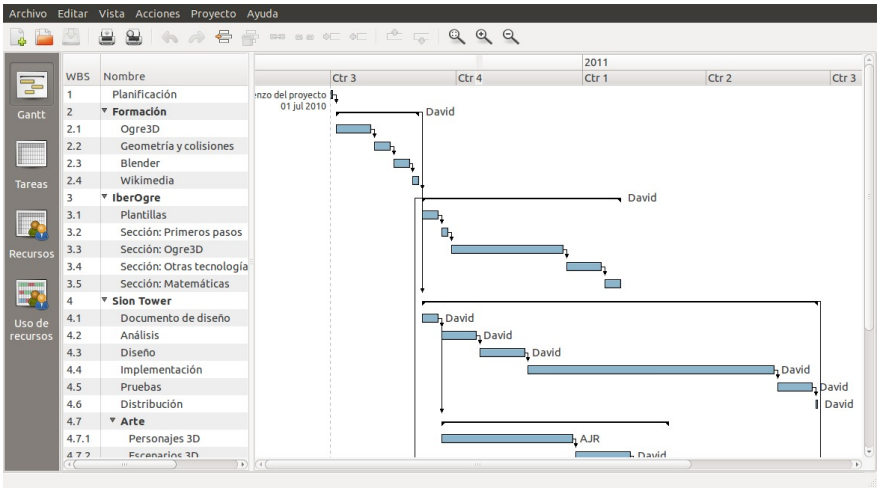


Figura 5.10: Planificador de proyectos Planner

Python e IDL. Es multiplataforma y mucho más ligero que herramientas similares como *Umbrello*. Permite la reutilización de clases entre diagramas y cuenta con una amplia variedad de posibilidades: diagramas de clases, de interacción, de secuencia, de casos de uso, etc.

Tanto en el análisis (capítulo 4.2) como el diseño (capítulo 4.3) ha participado la herramienta *BOUML*. Su interfaz no es elegante pero resulta funcional y sencilla de utilizar, puede verse en la figura 5.11.

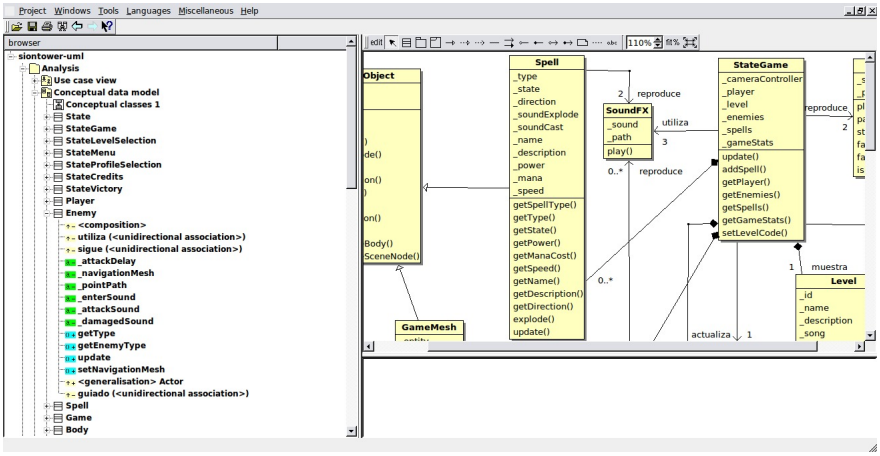


Figura 5.11: Editor de diagramas con notación UML BOUML

Dia

Dia [10] es una herramienta libre para crear diagramas de propósito general. Permite varios tipos como notación *UML*, diagramas de flujo, entidad relación, circuitería y un largo etcétera. Una vez hayamos terminado de trabajar en nuestro diagrama, podemos exportarlo a varios formatos como JPG, PNG o SVG.

Si bien es mucho menos potente que *BOUML* permite realizar otros tipos de diagramas. Concretamente, en **Sion Tower** ha sido empleado para esquemas de flujo como la máquina de estados que puede verse en la figura 5.12.

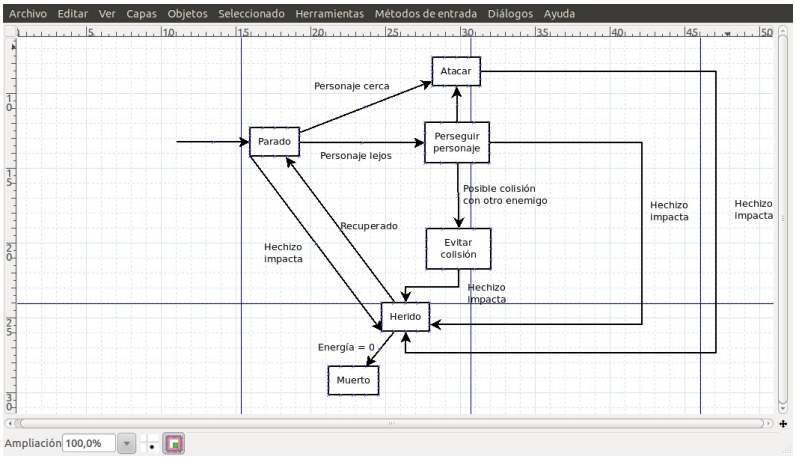


Figura 5.12: Herramienta de creación de diagramas

Manual de usuario

Bienvenido a Sion Tower

Bienvenido a **Sion Tower**, un videojuego de estrategia y acción en 3D ambientado en un mundo fantástico compatible con sistemas GNU/Linux y Windows. A lo largo de estas páginas podrás conocer los detalles sobre la historia y la ambientación del juego. Más adelante encontrarás los requisitos mínimos, de forma que puedas saber si tu equipo podrá ejecutarlo sin problemas. A continuación se ofrece una completa guía de instalación para los dos sistemas operativos soportados. Por supuesto, este manual contiene todos los detalles sobre el control y las posibilidades de **Sion Tower**. Finalmente encontrarás información detallada sobre cómo crear tus propios escenarios y añadirlos al juego.

Figura 5.13: Logo de Sion Tower

Historia

El gremio de magos está localizado en la Torre Sagrada, **Sion Tower**. Allí viven, estudian los libros de hechizos, celebran sus ritos secretos y protegen innumerables riquezas. Un día, el gremio al completo abandona la Torre para celebrar un ritual en el bosque cercano. **Merlín**, protagonista de **Sion Tower** es el único que permanece en el edificio sagrado para protegerlo. A pesar de ser un joven iniciado e inexperto, no iba a suceder nada porque el resto se ausentase durante unas horas.

La imprudencia del gremio resulta ser completamente desastrosa y una horda de monstruos no tarda en rodear **Sion Tower**. La misión del joven aprendiz consiste en detener la invasión piso por piso de la Torre. ¡Debe evitar que roben la **Reliquia Sagrada**!

Protagonista

Merlín pertenece a una de las categorías inferiores del gremio, los iniciados. Se le reconoce fácilmente por su pequeña estatura y sus ropas sencillas. Es un simple estudiante de magia en la Torre Sagrada y lleva tiempo tratando de hacerse un hueco entre sus superiores a base de esfuerzo y entrega. La invasión de la Torre es, en cierta medida, una gran oportunidad para **Merlín** de mostrar su valía. No obstante, está muerto de miedo, nunca se había visto obligado a emplear sus poderes en una situación tan extrema.

Figura 5.14: Protagonista de Sion Tower

Características físicas:

- Vida: 100
- Maná: 100 (regeneración automática)
- Velocidad: $5m/s$
- Hechizos: Bola de fuego, Furia de Gea y Ventisca.

Enemigos

Los enemigos de **Sion Tower** son monstruos abominables de procedencia desconocida. No obstante, su objetivo es claro: eliminar al gremio de magos y robar la Reliquia Sagrada, una leyenda de grandioso poder. Cada enemigo tiene unas características visuales y físicas determinadas (mostradas en la tabla inferior) aunque su comportamiento es similar. Todos tratan de encontrar a los supervivientes para acabar con ellos.

El **Goblin** es una criatura de mente muy básica, verde, enana y muy desagradable. Sólo va equipado con una tosca espada corta y un burdo taparrabos. No son extremadamente duchos en combate pero sí cuentan con una gran velocidad. Su gran ventaja es el número, son capaces de utilizar su superioridad numérica para atosigar al enemigo y acabar con él.

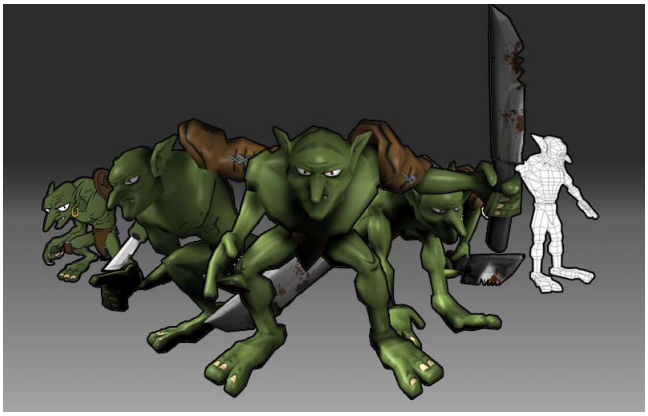


Figura 5.15: Goblin

El **Diablillo** es una criatura venida del propio averno y bastante superior al Goblin en lo que a propiedades físicas se refiere. Sus garras, alas y cola de demonio les basta para atacar desgarrando la carne que se encuentra a su paso, no necesitan ningún tipo de arma adicional.

Figura 5.16: Diablillo

El **Golem de hielo** es una criatura de gran tamaño venida de las montañas heladas del norte. Su estructura corporal está formada por piedra maciza y bloques de hielo. Es una de las criaturas más lentas que existen pero sus golpes son demoledores.

Figura 5.17: Golem de hielo

Nombre del enemigo	Vida	Daño	Velocidad
Goblin	5	10	5
Diablillo	7	15	4.5
Golem de hielo	10	20	2

Tabla 5.1: Comparativa de características de enemigos

Hechizos

Los hechizos son el único arma de **Merlín** para detener el avance de los enemigos. Sus conocimientos mágicos no le permiten hacer un enorme despliegue de proyectiles devastadores. Debe administrar cuidadosamente su limitada energía mágica (maná) para convocar débiles cúmulos de energía.

Con **Bola de fuego** puedes invocar un proyectil mágico que abrase a tus enemigos a su paso. **Furia de Gea** convoca la propia fuerza de la naturaleza para volverla en contra de cualquier criatura. Finalmente, **Ventisca** lanza varios proyectiles mágicos helados y afilados como cuchillas.

Nombre del hechizo	Maná	Daño	Velocidad
Bola de fuego	2	3	8
Furia de Gea	3	6	8
Ventisca	4	8	8

Tabla 5.2: Comparativa de atributos de hechizos

Requisitos mínimos

Para poder disfrutar de **Sion Tower** sin problemas necesitas disponer de un equipo con unas características iguales o superiores a las que se listan a continuación:

- **Sistema operativo**: GNU/Linux, Windows XP Service Pack 2, Windows Vista o Windows 7.
- **Procesador**: Pentium 2GHz o AMD.
- **Memoria**: 100 MB de memoria RAM disponibles.
- **Tarjeta de vídeo**: 128 MB de memoria y aceleración 3D.
- **Espacio en disco**: 100 MB.
- **Control**: ratón y teclado.

En caso de utilizar un sistema operativo GNU/Linux y experimentar problemas, revisa que tengas instalada la última versión de los drivers para tu tarjeta gráfica y que éstos soportan aceleración por hardware.

Descarga e instalación

En esta sección se proporcionan las indicaciones pertinentes para que puedas descargar e instalar **Sion Tower** en tu sistema. En primer lugar, debes acudir a la sección de ficheros del repositorio del juego en la forja de RedIRIS:

`http://forja.rediris.es/frs/?group_id=820`

Simplemente descarga la versión de **Sion Tower** que coincida con tu sistema. No se han publicado binarios para GNU/Linux, por lo que utilizas dicho sistema, tendrás que descargar el paquete con el código fuente y compilarlo tú mismo. En las página 161 encontrarás más información al respecto.

Instalación en GNU/Linux

Para poder jugar a **Sion Tower** en un sistema operativo GNU/Linux es necesario descargar e instalar el entorno de desarrollo completo ya que, por el momento, no se distribuyen binarios del software. El videojuego cuenta con las siguientes dependencias:

- libogre-dev
- libsdl1.2-dev
- libsdl-mixer1.2-dev
- mygui

A lo largo del proceso de instalación, supondremos que se posee una distribución basada en Debian que dispone de la utilidad `apt-get`. De no ser así, el proceso será similar utilizando el programa equivalente para instalar paquetes en la distribución correspondiente. En primer lugar, Debemos instalar todas las dependencias de OGRE3D, entre las que se encuentran el compilador *GCC*, la herramienta *make* o el generador de makefiles *CMake*. Para ello utilizamos la siguiente orden:

```
sudo apt-get install g++ make libfreetype6-dev libboost-date-time-dev \
  libboost-thread-dev nvidia-cg-toolkit libfreeimage-dev zlib1g-dev \
  libzzip-dev libois-dev libcppunit-dev doxygen libxt-dev libxaw7-dev \
  libxxf86vm-dev libxrandr-dev libglu-dev cmake
```

Una vez instaladas las dependencias podemos dirigirnos a la web oficial de OGRE3D y descargar el código fuente para sistemas GNU/Linux.

http://www.ogre3d.org/download/source

El siguiente paso consiste en descomprimir el paquete y acceder al directorio resultante utilizando la terminal. La herramienta *CMake* nos ayuda a generar un fichero `makefile` dependiente de nuestra plataforma de manera muy sencilla. Para generarlo el makefile, compilar e instalar OGRE3D introducimos la siguiente secuencia de órdenes:

```
cmake.
make
sudo make install
```

Instalar LIBSDL y LIBSDL MIXER es muy sencillo puesto que usualmente se encuentran en los repositorios de la distribución. Simplemente introducimos:

```
sudo apt-get install libsdl1.2-dev libsdl-mixer1.2-dev
```

Finalmente procedemos a la instalación de la biblioteca MYGUI. Hemos de acudir a su sección de ficheros en la forja SourceForge y descargar la versión *3.2.0 Release Candidate 1*.

http://sourceforge.net/projects/my-gui/files/MyGUI/

Cuando finalice la descarga, descomprimimos el paquete en formato zip y accedemos al directorio resultante utilizando la terminal. En este caso también generaremos un makefile de forma automática con *CMake* para después compilar e instalar a través de *make*.

```
cmake .
make
sudo make install
```

Llegados a este punto habremos finalizado instalando las dependencias de **Sion Tower**. Ahora simplemente es necesario acceder a su directorio mediante la terminal y comenzar con la compilación. Una vez finalizada con éxito, podremos ejecutarlo.

```
make
./siontower
```

Instalación en Windows

Tras descargar el paquete, utiliza cualquier programa de compresión/descompresión que soporte el formato zip. Si no tienes ninguno instalado, puedes emplear *7Zip* sin problemas ya que es Software Libre. Para descargarlo, dirígete a la siguiente dirección:

`http://www.7-zip.org`

Sion Tower es completamente portable y autocontenido, así que si lo prefieres puedes guardarlo en un lápiz de memoria o en cualquier disco duro externo. Para iniciar el juego, haz doble click sobre el fichero ejecutable `siontower.exe`.

Guía de juego

En este bloque el jugador novel podrá aprender cómo se juega a **Sion Tower**. Recorreremos las pantallas de la aplicación explicando la funcionalidad de cada una de forma que aprenderemos a crear y gestionar perfiles de jugadores, seleccionar nivel, controlar al personaje guiándolo hacia su objetivo y progresar sumando puntos o desbloqueando nuevos escenarios.

Menú principal

Justo al iniciar **Sion Tower** se nos presenta el menú principal del juego con un paisaje montañoso en el que se ve a la Torre Sagrada siendo atacada por enemigos en medio de una intensa lluvia.

Figura 5.18: Menú principal de Sion Tower

Desde esta pantalla podemos llevar a cabo las siguientes acciones:

- **Jugar**: nos lleva a la pantalla de selección de perfil y al camino para comenzar una partida.
- **Créditos**: nos muestra una escena con las personas que han participado en el desarrollo.
- **Salir**: cierra el juego por completo, ¿seguro que quieres dejar de jugar?

Selección de perfil

Sion Tower cuenta con un sistema de perfiles para que varias personas puedan jugarlo y controlar su progreso. Al comienzo de la partida sólo está disponible el primer nivel. Cuando finalices con éxito un escenario, el siguiente nivel se convertirá en accesible. Además, cada perfil cuenta con una puntuación que mide nuestras habilidades en el juego. Esta puntuación puede subir o bajar en función de nuestra actuación en las partidas. Esto permite que varios usuarios compitan en una máquina de forma asíncrona para alcanzar la máxima puntuación.

Figura 5.19: Pantalla de selección de perfil en Sion Tower

La pantalla de selección de perfil consta de dos paneles bien diferenciados. Selección de un perfil ya existente y creación de un nuevo perfil. En el primer bloque puedes visualizar la experiencia y los niveles desbloqueados de cada jugador pulsando con el ratón sobre sus nombres. Si tu perfil se encuentra en la lista y quieres utilizarlo tienes que seleccionar el nombre y pulsar el botón **Aceptar**. Es posible eliminar un perfil de la lista con el botón **Eliminar**. No obstante, hay que ser cautelosos, puesto que un perfil eliminado no es recuperable.

Cuando quieras crear un nuevo perfil, escribe su nombre en el panel inferior y pulsa sobre **Crear**. Al menos debes introducir algún carácter y el nombre elegido no debe figurar entre los perfiles existentes, de lo contrario se mostrará un error.

Para volver al menú principal de **Sion Tower** puedes pulsar sobre el botón **Atrás**.

Selección de nivel

Tras seleccionar nuestro perfil se nos presentará la pantalla de selección de nivel. En la lista de niveles aparecen todos los existentes, ya estén bloqueados o disponibles. Los bloqueados para nuestro perfil lo indican en rojo y su icono es translúcido.

Figura 5.20: Pantalla de selección de nivel en Sion Tower

Para seleccionar un nivel, pulsa con el ratón sobre su icono. Si deseas volver a la pantalla de selección de perfil, pulsa sobre **Atrás**.

Partida

Tras seleccionar el nivel comienza la partida y los enemigos aparecen en pequeñas oleadas. El objetivo consiste en eliminarlos a todos utilizando nuestra energía mágica (maná) para invocar hechizos y mantener nuestra energía vital por encima de cero.

Figura 5.21: Partida de Sion Tower

Los controles son los siguientes:

- **Movimiento**: para mover al personaje, utiliza las teclas W, A, S y D.
- **Cámara**: la cámara siempre hace un seguimiento al personaje para que nunca salga del encuadre. Puedes acercar o alejar la cámara utilizando la rueda del ratón. También puedes moverla alrededor del personaje moviendo el ratón mientras pulsas el botón derecho.
- **Hechizos**: para lanzar un hechizo apunta con el ratón en la dirección deseada y pulsa el botón izquierdo. Si tienes maná suficiente lanzarás el hechizo seleccionado. Para cambiar de hechizo, pulsa sobre el que quieras seleccionar en el menú de juego. También puedes hacer uso de las teclas de acceso rápido: 1, 2 y 3.
- **Pausa**: para activar la pausa puedes pulsar la tecla escape o el botón del menú de juego. En el menú de pausa puedes **Reanudar** la partida, volver a la pantalla de **Selección de nivel** o dirigirte al **Menú principal**.

Si pasas el ratón por encima de un hechizo puedes acceder a más información sobre el mismo: nombre, descripción, maná necesario y daño que causa. Cuando no tengas maná suficiente para lanzar un hechizo determinado, éste se mostrará en gris. La energía mágica se recupera progresivamente de forma automática cuando no se está haciendo uso de ella. No es posible recuperar vida a lo largo de la partida.

Figura 5.22: Menú de pausa en Sion Tower

Si nuestra energía se termina por completo habremos perdido la partida y se nos avisará mediante un cartel. Mientras decides si volver a intentarlo puedes seguir moviendo la cámara y observando cómo los enemigos han alcanzado la victoria. Si pulsas la **barra espaciadora** volverás a la pantalla de selección de nivel.

Figura 5.23: Mensaje de derrota en Sion Tower

Pantalla de victoria

Al eliminar a todos los enemigos de un nivel habrás completado con éxito el escenario y llegarás a la pantalla de victoria en la que el personaje celebra su logro. Si no estabas jugando al último nivel disponible, habrás conseguido desbloquear el siguiente. Tanto la puntuación obtenida como los niveles desbloqueados se guardan automáticamente en este punto.

Figura 5.24: Pantalla de victoria en Sion Tower

La puntuación de **Sion Tower** se obtiene mediante la suma de los siguientes factores:

- **Enemigos**: por cada enemigo eliminado se obtienen puntos. Cuanto más poderoso sea el enemigo, mayor será la recompensa en forma de puntos.

- **Vida**: recibimos un complemento por la energía que nos quede al finalizar la partida con éxito.
- **Maná**: en el juego se valora mucho la puntería y la administración del maná o energía mágica. Cuanto menos maná utilicemos, mayor será la puntuación. Es posible que si se malgasta demasiado, la puntuación sea negativa.
- **Tiempo**: recibimos recompensa en forma de punto por eliminar a los enemigos lo antes posible, este apartado también puede recibir una valoración negativa.

Tras terminar con éxito un nivel puedes elegir la opción **Volver a jugar**, volver a la pantalla de **Selección de nivel** o regresar al **Menú principal**.

Créditos

En la pantalla de créditos aparece una escena de la Torre Sagrada con un panel listando los participantes en el desarrollo de **Sion Tower**. Cuando desees volver puedes pulsar el botón **Atrás**.

Figura 5.25: Autores de Sion Tower

Creación de niveles

¿Tienes una gran idea para un nuevo nivel de **Sion Tower**? ¿Sabes como puedes mejorar sustancialmente la historia? ¿Te gustaría que todos viesen tus ideas? El motor del videojuego permite a los usuarios crear y añadir nuevos niveles de manera muy sencilla utilizando la herramienta de modelado y animación 3D libre por excelencia: *Blender*. El proceso es largo pero sencillo, sobre todo si sigues detenidamente todos los pasos de este manual.

Instalación de Blender y complementos

Blender es la herramienta de modelado y animación 3D de carácter completamente libre más potente que existe. Utilizando *Blender* se han producido en su totalidad cortos de animación como Big Buck Bunny[1],

[1]Big Buck Bunny: http://bigbuckbunny.org

Sintel[2] o videojuegos como Yo Frankie![3]. *Blender* es multiplataforma, no importa si posees Windows, Mac o GNU/Linux ya que no tendrás problema alguno para utilizarlo. Recientemente se ha publicado la serie 2.5 de la herramienta, que incluye mejoras como un cambio radical de la interfaz. No obstante, los scripts de exportación necesarios funcionan en la 2.49, por tanto, esa sera la versión que utilizaremos.

Figura 5.26: Fotograma del cortometraje Sintel

Si utilizas un sistema operativo GNU/Linux basado en Debian, es probable que *Blender* se encuentre en los repositorios de tu distribución. En tal caso, bastaría con introducir en una terminal:

```
sudo apt-get install blender
```

En caso de contar con un sistema operativo de tipo Windows, debes acudir a la sección de versiones anteriores de *Blender* y descargar el instalador:

http://download.blender.org/release/Blender2.49a/blender-2.49a-windows.exe

Simplemente abre el instalador haciendo doble click sobre el ejecutable, sigue los pasos indicados y habrás instalado Blender con éxito.

En ambos sistemas operativos es necesario instalar un plugin de exportación que convierte una escena creada con Blender a un fichero de texto en formato xml procesable por el motor de **Sion Tower**. El formato se conoce como *Dotscene* y es ampliamente utilizado. Asimismo, es necesario otro complemento que convierte un modelo creado con *Blender* a un xml. Es posible descargar ambos plugins desde:

http://ogreaddons.svn.sourceforge.net/viewvc/ogreaddons/trunk/blender\sceneexporter/ogredotscene.py

http://ogre.svn.sourceforge.net/viewvc/ogre/branches/v1-6/Tools/BlenderExport

Para instalar los exportadores en sistemas GNU/Linux, hemos de copiar los ficheros descargados en:

[2]Sintel: http://sintel.org
[3]Yo Frankie!: http://yofrankie.org

```
/.blender/scripts
```

En cambio, si utilizamos Windows el directorio será:

```
[Instalación de Blender]/.blender/scripts
```

Composición del nivel

Para seguir esta guía con soltura es recomendable que poseas unos conocimientos mínimos sobre *Blender*. Deberías saber cómo manejar los elementos básicos de la interfaz y la forma de manipular los objetos dentro de la escena: movimiento, rotación, escala, duplicado, etc. Si nunca has utilizado *Blender* con anterioridad es recomendable que acudas a algún manual para principiantes como el libro gratuito bajo Creative Commons *Aprende Blender en 24 horas*[4] del profesor certificado por la Blender Foundation Carlos González Morcillo.

Crear un nuevo documento

El primer paso es crear un nuevo documento de *Blender*, para ello simplemente abrimos la herramienta por primera vez o hacemos click sobre File → New y aceptamos eliminar la escena actual. Para comenzar no podemos tener objetos presentes por lo que seleccionamos toda la escena con A y eliminamos con X.

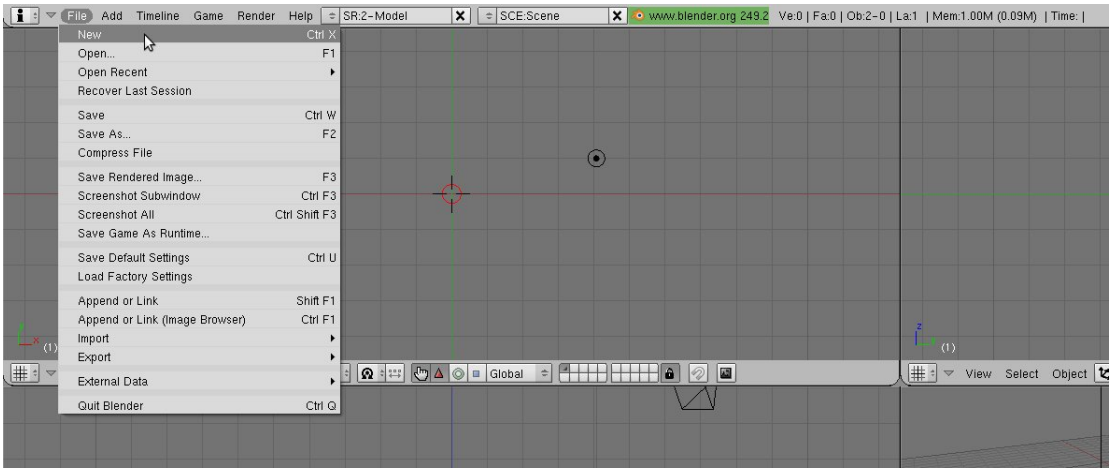


Figura 5.27: Creación de un nuevo documento en Blender

Es recomendable contar con un espacio de trabajo cómodo, la disposición de las vistas tridimensionales de *Blender* es fundamental. Para crear un nuevo nivel lo ideal es contar con las vistas frontal, lateral, cenital y libre. Puedes ver un ejemplo de esta disposición en la figura 5.27.

Convenciones de nombrado

En un escenario de **Sion Tower** se dan cita varios tipos de elementos: objetos del escenario, geometría arbitraria, efectos de partículas, oleadas de enemigos, etc. *Blender* no distingue entre ellos, para la herramienta de modelado todos son objetos tridimensionales comunes. Hemos de seguir unas reglas de

[4] Aprende Blender en 24 horas: http://www.inf-cr.uclm.es/www/cglez/downloads/blender/24hBlenderYafray.pdf

nombrado especiales para que el motor sepa distinguir entre una mesa que forma parte del mobiliario y un icono simbólico que representa la llegada de un enemigo.

Durante todo el proceso de creación del escenario es importante prestar atención para que los objetos tengan un nombre acorde con las reglas establecidas. En la pestaña de edición (tecla F9) aparece el panel *Link and Materials* mostrando el nombre de la malla del objeto y el del propio objeto. En nuestro caso, objetos semejantes comparten la misma malla mientras que el nombre de cada objeto debe ser distinto y ajustarse a las reglas. Puedes verlo en la figura 5.28.

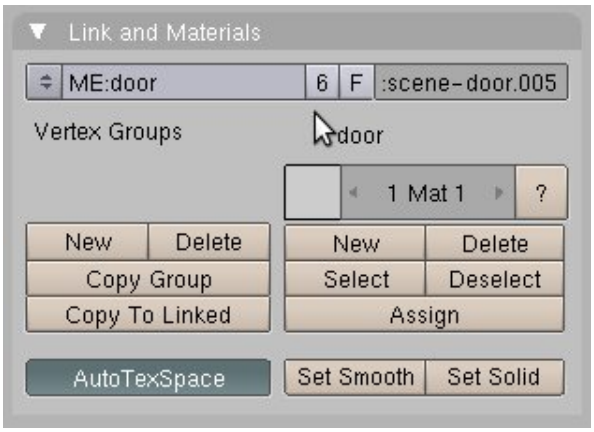


Figura 5.28: Nombre de una puerta en un escenario

Las reglas de nombrado son las siguientes:

- **Escenario**: todos los elementos del escenario (paredes, suelo, mesas, sillas y otros muebles) siguen la regla `scene-nombre.numero`. El nombre del objeto es lo que lo identifica dentro del catálogo para poder recuperar su modelo de colisión y el número ayuda a hacerlo único dentro de la escena (dos objetos no pueden tener el mismo nombre). Ejemplo: `scene-door.005`.
- **Efectos de partículas**: puedes incluir efectos de partículas en cualquier punto de la escena. La regla es `particle-nombre.numero`. Ejemplo: `particle-flame.001`.
- **Luces**: el motor del juego reconoce las luces de manera automática, por lo que no debes preocuparte de sus nombres.
- **Malla de navegación**: la malla de navegación indica a los enemigos cuales son las zonas transitables del escenario y siempre debe llamarse `navMesh`. Veremos más sobre las mallas de navegación en secciones posteriores.
- **Enemigos**: los enemigos siguen la regla `enemy-tipo-t.numero`. La letra t representa el segundo en el que aparecerá el enemigo desde que se inicia la partida. Ejemplo: `enemy-goblin-25.001`.
- **Protagonista**: el elemento cuyo nombre sea `player` definirá la posición inicial del jugador dentro del nivel.
- **Geometría arbitraria**: toda la geometría que no siga la convención de nombrado será tratada como complementos del escenario. No se calcularán colisiones contra ellos (se podrán atravesar).

Enlazar objetos

Los escenarios del juego están repletos de objetos de distintas clases y puedes reutilizarlos sin ningún tipo de problemas para tus propias creaciones. Incluir los objetos ya creados para **Sion Tower** en tu

nuevo nivel es muy sencillo. En el menú de *Blender* seleccionamos File → Append or Link (Image Browser). Se abrirá un pequeño navegador de ficheros para que seleccionemos el `objeto.blend` a importar. Al abrirlo, tendremos que seleccionar Object y el nombre del objeto creado en *Blender*.

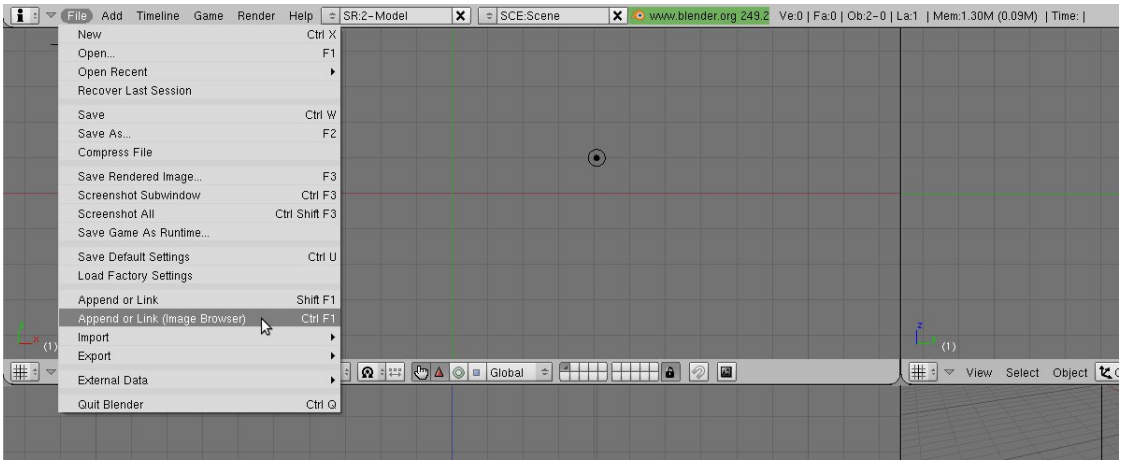


Figura 5.29: Enlazar con un objeto de Blender existente

Al enlazarlo se añadirá al origen de coordenadas el objeto deseado. No obstante, los datos de su malla pertenecen a un fichero externo (contorno azul) y lo deseable es convertirlo en local. De esta manera nuestro proyecto de nivel no contará con indeseables dependencias externas. Para hacerlo seleccionamos el objeto con el botón derecho del ratón, pulsamos L y seleccionamos la opción All. Una vez localizado, podremos mover el objeto, escalarlo y rotarlo por la escena con total libertad.

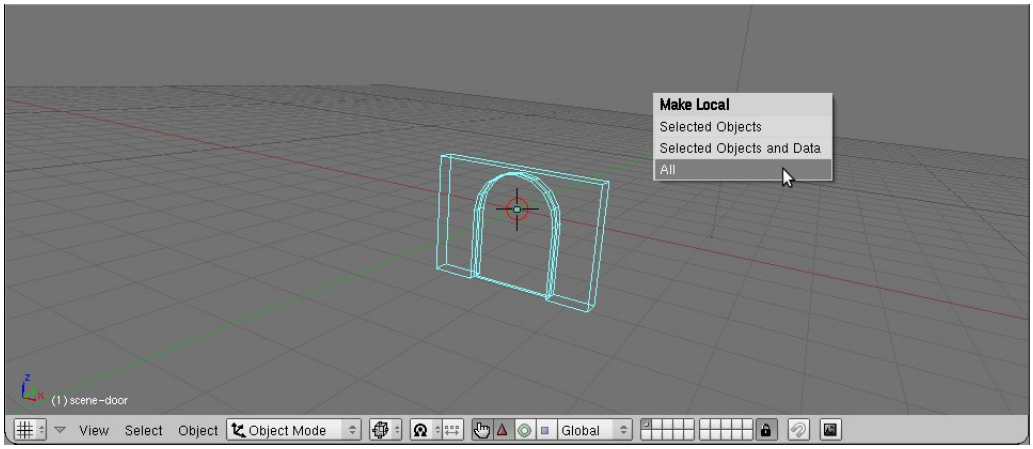


Figura 5.30: Convertir en local un objeto enlazado

Elementos del escenario

Los elementos del escenario son aquellos objetos que cuentan con un modelo de colisión. Esto quiere decir que ni el protagonista ni los hechizos podrán atravesarlos. Los objetos que venían incluidos en **Sion Tower** ya tienen un modelo de colisión asignado y para utilizarlos sólo hemos de enlazarlos siguiendo los pasos de la sección anterior. Estos objetos se encuentran en `[siontower]/media/blender` y la lista completa junto a sus códigos es la siguiente:

- Cama (bed)
- Candelabro (candle)
- Columna (column)
- Muro con arco (door)
- Muro con puerta (door2)
- Muro (wall)
- Suelo $8x8m$ (floor8x8)
- Suelo $4x4m$ (floor4x4)
- Reliquia (reliq)
- Silla (chair)
- Taburete (roundchair)
- Estanterías (shelves)
- Escaleras (staircase)
- Mesa (table)
- Caja de madera (woodenbox)

Cuando enlacemos un objeto y lo convirtamos en local podemos moverlo utilizando la tecla G, rotarlo con R o escalarlo mediante S. Las tres operaciones anteriores pueden restringirse a uno de los tres ejes. Por ejemplo, para desplazar una mesa a lo largo del eje X será necesario seleccionar la mesa con el botón derecho del ratón, pulsar G, pulsar X y desplazarla. Es necesario que no haya desniveles y que el nivel del suelo esté situado en el plano $Z = 0$ de *Blender*

Basta con enlazar al principio cada objeto y duplicarlo cada vez que deseemos uno semejante. En *Blender* es posible duplicar objetos seleccionando el elemento a duplicar y pulsando SHIFT + D. No obstante, dicha operación conlleva tener redundancias en los datos de la malla y lo que deseamos es contar con dos instancias de la misma malla. Para duplicar elementos utilizaremos la combinación ALT + D. Con estas sencillas instrucciones ya posees los conocimientos suficientes para componer escenas como la de la figura 5.31.

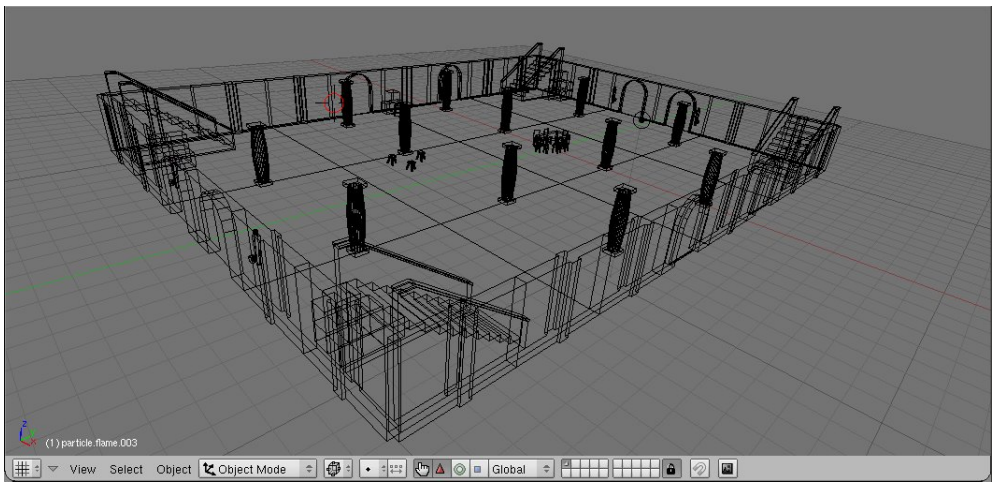


Figura 5.31: Elementos del escenario ya colocados

Nuevos objetos

Puedes añadir nuevos objetos que no estén en el catálogo modelados por tí mismo o que hayas encontrado en algún banco de recursos libres. Una vez modelado y texturizado, será necesario que lo exportes en un formato compatible con el motor gráfico del juego OGRE3D, el formato binario `.mesh`. Puedes obtener más información sobre la exportación en **IberOgre**:

http://wikis.uca.es/iberogre/index.php/Exportar_modelos_desde_Blender

Los modelos exportados se sitúan en el directorio `[siontower]/media/meshes`, los materiales en `[siontower]/media/materials` y las texturas en `[siontower]/media/textures`. El último paso consiste en definir el modelo colisionable del nuevo elemento del escenario. Debes acudir al fichero xml `[siontower]/media/levels/bodycatalog.xml` y editarlo de forma manual.

Cada objeto cuenta con un nombre único y tiene asociado un cuerpo (body) compuesto por una o más formas (shapes). Los objetos tienen un tipo, pueden ser 1 (suelo) o 2 (mobiliario). Cada forma es una figura geométrica sencilla:

- **Esfera**: *sphere*, definida por un centro y un radio.
- **Plano**: *plane*, definido por un punto y un vector perpendicular.
- **Caja alineada**: *aabb*, un hexaedro alineado con los ejes, definido por un centro y unas distancias hasta sus lados.
- **Caja orientada**: *obb*, hexaedro al que se le ha aplicado una rotación, definido por un centro, distancias hasta los lados y los ejes sobre los que se apoya.

Ha de añadirse una nueva entrada por cada objeto nuevo que se añada si se desea que cuente con un modelo colisionable. La sintaxis del fichero xml es bastante sencilla.

```
<?xml version="1.0" encoding="UTF-8" ?>
<bodies>

    <body name="floor8x8" type="1">
        <shape type="plane">
            <position x="0" y="0" z="0" />
            <normal x="0" y="1" z="0" />
        </shape>
    </body>

    <body name="chair" type="2">
        <shape type="obb">
            <center x="0" y="-0.036" z="0.147"/>
            <extent x="0.2275" y="0.4675" z="0.21"/>
            <axes a00="1" a01="0" a02="0" a10="0" a11="1"
                  a12="0" a20="0" a21="0" a22="1" />
        </shape>
    </body>

</bodies>
```

Efectos de partículas

Como ya se ha mencionado anteriormente, los efectos de partículas siguen la nomenclatura `particle -nombre.numero`. Lo ideal es representarlos mediante pequeñas esferas de un color relacionado con el efecto que generan. En **Sion Tower** se incluyen tres tipos de sistemas de partículas por defecto:

- **flame**: llama del tamaño adecuado para ser utilizada en las antorchas.
- **flame2**: llama del tamaño adecuado para colocarse sobre los candelabros.
- **rain**: lluvia intensa utilizada en el menú principal del juego.

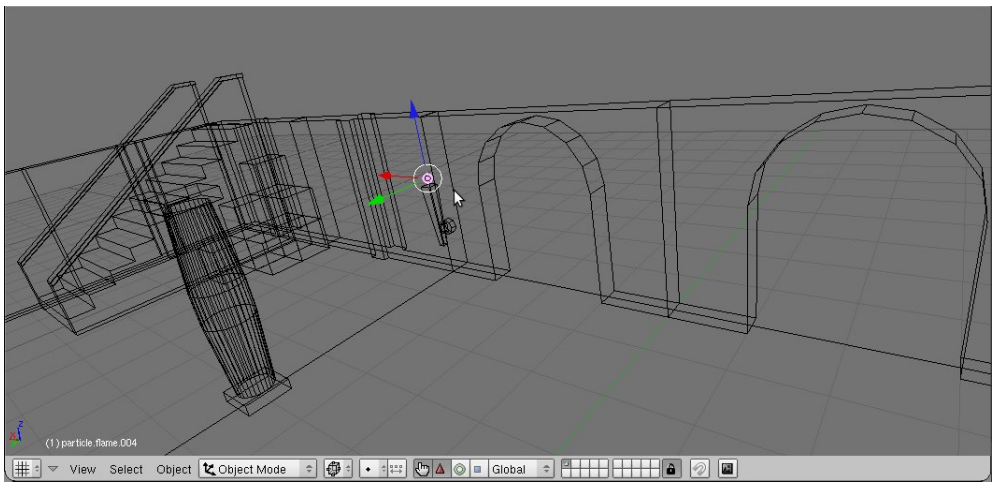


Figura 5.32: Esfera simbolizando un sistema de partículas

Los sistemas de partículas se definen en scripts de extensión `.particle` almacenados en el directorio `[siontower]/media/particles`. Si conoces la sintaxis, puedes añadir los que desees para utilizarlos en tus propios niveles. Hay herramientas como *Ogre Particle Editor*[5] que permite generar dichos scripts utilizando una interfaz gráfica sencilla.

Iluminación

Añadir luces a una escena es imprescindible, de lo contrario esta aparecería completamente oscura dentro del juego. Para hacerlo pulsa la barra espaciadora dentro de la escena 3D y selecciona Add → Lamp y uno de los tipos de luces aceptados por **Sion Tower**: Lamp, Sun y Spot. No tienes porqué preocuparte por ninguna convención de nombrado en este caso. Una vez hayas añadido la luz, puedes moverla por la escena como desees.

Es posible configurar varios parámetros de cada punto de luz como su intensidad y color accediendo al panel de sombreado (F5) y al subpanel de controles de luz. Lo recomendable es que ninguna luz ascienda de 1,0 en intensidad y que los colores sean suaves.

[5] Ogre Particle Editor: http://www.ogre3d.org/tikiwiki/OGRE+Particle+Editor&structure=Tools

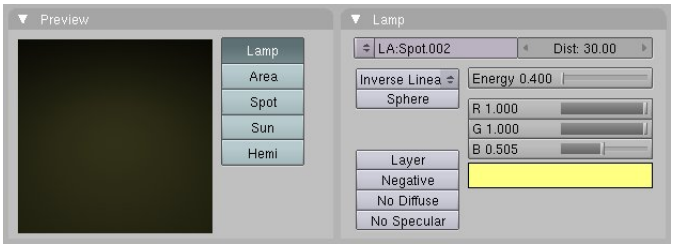


Figura 5.33: Panel de configuración de luces

Malla de navegación

La malla de navegación es un conjunto de triángulos conectados que definen un área transitable para los enemigos. En **Sion Tower** los monstruos no podrían encontrar el camino hacia el personaje si no fuera por la malla de navegación. Es similar a contar con un tablero repleto de casillas conectadas. Una vez tengamos claro dónde se colocarán los obstáculos podemos ir creando la malla. Recuerda que el objeto debe llamarse `navMesh` (se distingue entre mayúsculas y minúsculas). El nombre de la propia malla debe ser único entre las mallas del juego, lo normal es llamarla `navMesh-número` donde número es la posición del nivel en la historia.

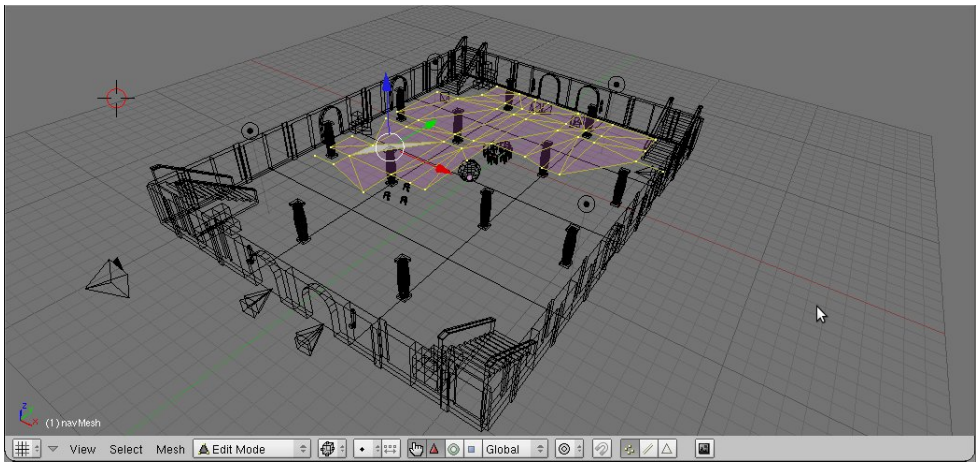


Figura 5.34: Creación de la malla de navegación

Para comenzar a crear la malla puedes añadir un plano pequeño con la barra espaciadora, Add → Mesh → Plane. Entra en modo edición utilizando el tabulador y selecciona uno de los vértices con el botón derecho para eliminarlo con X. Asegúrate de que la malla siempre permanece en el suelo, es importante para que los enemigos no se eleven misteriosamente. Debe estar formada exclusivamente por triángulos, no pueden existir cuadriláteros.

Extender la malla es sencillo, selecciona un vértice y pulsa E para extruirlo creando uno nuevo y una arista de unión. Ve repitiendo la operación para cubrir las zonas transitables del escenario y recuerda dejar un margen prudencial con los obstáculos. Debes crear caras entre los grupos de tres vértices que formen los triángulos, para hacerlo selecciona los tres vértices y pulsa F. Al finalizar es imprescindible que todas las caras de la malla miren hacia el mismo lado (arriba o abajo), para conseguirlo, selecciona todos los vértices con A y pulsa Ctrl + N.

Finalmente, debes exportar la malla para que el motor de **Sion Tower** pueda procesarla. Selecciona la malla fuera del modo edición (pulsa el tabulador) y accede a File → Export → OGRE Meshes.

Figura 5.35: Exportación de la malla de navegación

El nombre del material es irrelevante porque no nos hará falta, asegúrate de marcar la opción *Fix Up Axis to Y* y elegir el directorio `[siontower]/media` como destino de la exportación.

Enemigos y jugador

Los enemigos suelen representarse mediante una pirámide de base cuadrada en posición horizontal. Como ya hemos mencionado, la nomenclatura para los enemigos es `enemy-tipo-t.numero` siendo los tipos posibles: *goblin*, *demon* y *golem*. Para que el diseño de niveles sea más cómodo, los goblins llevan asignado el color verde, los demonios el rojo y los gólems el azul aunque no es obligatorio de cara a la exportación.

Tras añadir un enemigo, puedes duplicarlo utilizando la combinación SHIFT + D, moverlo con G y rotarlo con R (aunque lo normal es hacerlo sobre el eje vertical únicamente). La pirámide de los enemigos debe colocarse atravesando el suelo, de forma que el vértice superior esté en el punto 0 del eje Z.

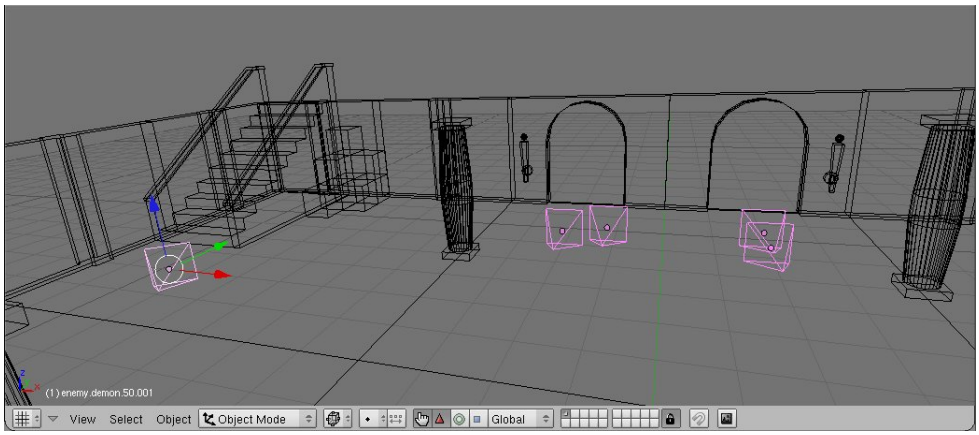


Figura 5.36: Disposición de enemigos en el nivel

Debes añadir otro elemento simbólico como una esfera o cono para indicar la posición inicial del jugador. Simplemente has de nombrarlo `player`.

Integración del nivel

Es conveniente que poco a poco vayas guardando tus progresos diseñando el nivel ya que no es extraño equivocarse o dejar el trabajo a media. Para ello selecciona File → Save, elige una ruta y presiona el

botón de guardado. A continuación se adjuntan las indicaciones para integrar el nivel que acabas de crear en **Sion Tower**.

Script de exportación

Tras haber guardado el nivel como fichero `.blend` es el momento de proceder a la exportación a formato *DotScene*. Para ello selecciona File → Export → Ogre Scene. Se desplegará un panel con todos los elementos de la escena, si quieres que todos aparezcan en el nivel, pulsa sobre All. Recuerda activar la opción *Fix Up Axis To Y* y selecciona el directorio de niveles como destino de la exportación: `[siontower]/media/levels`.

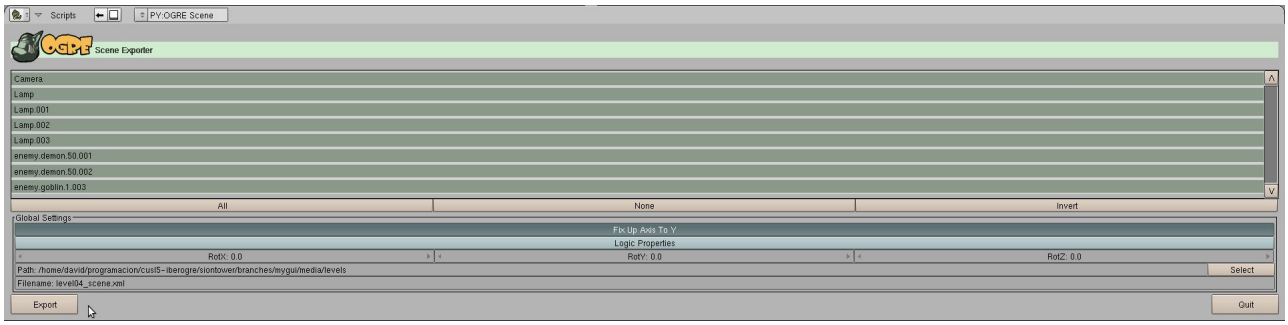


Figura 5.37: Exportación de la escena

Por cada nivel existen dos ficheros xml, el primero es el de la escena y tiene como nombre `codigo_scene.xml`. Por su parte, el segundo contiene información complementaria como el nombre del nivel, descripción y música que sonará, su nomenclatura es `codigo_info.xml`. Por ejemplo, si el código del nivel que deseamos exportar es `level05`, el nombre del fichero escena debería ser `level05_scene.xml`.

Añadir el nivel al juego

Para integrar el nuevo nivel dentro de **Sion Tower** debes crear un fichero `codigo_info.xml` en el directorio `[siontower]/media/levels`. La sintaxis es muy sencilla, a continuación se muestra el ejemplo para el primer nivel del juego. El fichero de música debe estar en formato OGG y estar situado en `[siontower]/media/music`.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <basicInfo>
3     <name>The Hall</name>
4     <description>Some enemies have assaulted the main hall of\nthe Sacred
          Tower. Stop the invasion!</description>
5     <song name="nivel1.ogg" group="" />
6 </basicInfo>
```

Como ya has visto en la página 164, en la pantalla de selección de nivel, cada escenario viene acompañado de un icono. Si no haces nada, el hueco quedará vacío para tu nuevo nivel. No obstante, si lo deseas, puedes añadir un fichero con nombre `codigo.jpg` al directorio `[siontower]/media/textures`. Finalmente, debes añadir tu nuevo nivel al fichero que indexa los niveles, `[siontower]/media/levels/levels`. Por ejemplo, si añadiésemos el quinto nivel de nuestro ejemplo anterior, el fichero quedaría de la siguiente manera.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<levels>
    <level id="level01"></level>
    <level id="level02"></level>
    <level id="level03"></level>
    <level id="level04"></level>
    <level id="level05"></level>
</levels>
```

Con esto habrás conseguido añadir un nivel completamente nuevo y personalizado a **Sion Tower**. Para comprobar que todo está correcto, inicia el juego y trata de seleccionar el escenario. Antes de compartirlo con tus amigos, es recomendable que compruebes que ni es demasiado complicado ni demasiado sencillo, ¡de lo contrario no sería divertido!

Comunidad y difusión

Concurso Universitario de Software Libre

El proyecto **IberOgre** y **Sion Tower** ha participado en el V Concurso Universitario de Software Libre en las categorías de *Comunidad* y *Educación y ocio*. Se trata de un concurso de software, hardware y documentación libre a nivel nacional en el que pueden participar grupos de hasta tres estudiantes universitarios, de bachiller o de ciclos superiores. Se valora el desarrollo del proyecto desde el comienzo del curso hasta la fase final que tuvo lugar el día 12 de mayo de 2011.

En esta edición se presentaron un total de 115 proyectos de diversa índole y los resultados que obtuvo **IberOgre** y **Sion Tower** no pudieron ser más satisfactorios. En la fase local del concurso fue galardonado con el premio al mejor proyecto de *Ocio* mientras que en la fase nacional, que se celebró en Granada, se recibió el premio al mejor proyecto de *Comunidad*.

El concurso se ha mostrado como un aliciente de lo más positivo para desarrollar el proyecto con más entusiasmo y apertura hacia la comunidad. Ha sido muy beneficioso en términos de audiencia y difusión gracias a los medios que se han hecho eco de la convocatoria. Sin duda, ha sido uno de los factores que más me ha motivado a seguir hacia delante tomándomelo como un reto personal y buscando una experiencia enriquecedora junto al resto de participantes.

Figura 5.38: Finalistas del V Concurso Universitario de Software Libre

Creación de comunidad

IberOgre y **Sion Tower** cuenta con un elevado aspecto de comunidad ya que juntos forman una plataforma de aprendizaje de desarrollo de videojuegos en tres dimensiones con OGRE3D. Era imprescindible llevar a cabo acciones para difundir el proyecto y buscar la creación de dicha comunidad. La comunicación con los lectores debía ser fluida, directa y cercana tratando de apelar a su curiosidad e interés por la materia. A continuación, se listan los medios empleados para contribuir a la difusión del proyecto.

- **Blog de desarrollo**

  En mi blog personal se ha creado una sección especial para informar de los avances del proyecto y documentar los subsistemas y algoritmos más relevantes. En total se han redactado más de 70 artículos y se han recibido más de 85.000 visitas. Puede accederse desde la siguiente dirección.

  `http://siondream.com/blog/category/proyectos/pfc`

- **Forja**

  El proyecto se ha alojado en la forja de RedIRIS la cual no ha sido utilizada únicamente por su repositorio *Subversion*. Se ha hecho uso de la sección de noticias para informar de los avances importantes, de la lista de tareas para gestionar el trabajo pendiente y de la lista de ficheros para publicar los resultados (juego y documentación adicional). El sitio del proyecto en la forja de RedIRIS puede ser accedido desde la siguiente dirección web.

  `https://forja.rediris.es/projects/cusl5-iberogre`

- **Twitter**

  Twitter es una red social de microblogging en la que los usuarios publican mensajes cortos. Es ampliamente utilizada para seguir noticias y estar informado de la actualidad en diversos sectores muy específicos. Existía una comunidad de desarrolladores hispanohablantes muy activa dentro de Twitter por lo que se decidió que el proyecto tuviera presencia en dicha red social. La comunicación en Twitter ha sido muy útil para acercarnos a los lectores y recibir sus sugerencias. Actualmente la cuenta @IberOgre posee 98 seguidores y puede accederse desde:

  `http://twitter.com/#!/IberOgre`

- **Web en la forja**

  La forja de RedIRIS proporciona a los proyectos un pequeño espacio web. Si bien no otorga libertad absoluta para publicar contenido (solo se permiten webs estáticas en HTML) cuenta con un posicionamiento extremadamente favorable en buscadores. Se ha aprovechado dicho espacio con una web a modo de índice indicando brevemente en qué consiste el proyecto y enlazando a los medios oficiales. Puede verse en la figura 5.39 y accederse desde la siguiente dirección.

  `http://cusl5-iberogre.forja.rediris.es`

- **Canal de Youtube**

  Durante todo el desarrollo se han ido subiendo los progresos de **Sion Tower** al servicio de vídeo vía streaming por excelencia. Esto ha permitido que el interés por el proyecto crezca progresivamente. En total se han publicado diez vídeos los cuales se han reproducido en más de 2.800 ocasiones. Puede accederse al canal correspondiente en la siguiente dirección.

  http://www.youtube.com/user/davidsaltares

Figura 5.39: Web del proyecto en la forja RedIRIS

Difusión

Desde el momento en el que **IberOgre** comenzó a contar con varios artículos publicados, el público comenzó a tomar interés en el proyecto. Por supuesto, el mayor empuje se produjo tras las fases local y final del V Concurso Universitario de Software Libre. A continuación, haremos un repaso por los medios que se han hecho eco del proyecto.

- **Comunidades de desarrollo**

  Diversas comunidades de desarrollo de videojuegos en español mostraron rápidamente su interés principalmente en **IberOgre** por proporcionar documentación en su idioma nativo. A continua-

ción, enlazamos a algunos de los medios que han publicado artículos sobre el proyecto.

http://www.creagames.es/iberogre-un-proyecto-espanol-de-ogre-engine
http://razonartificial.com/2011/03/iberogre-documentacion-de-ogre-en-espanol
http://programandoideas.com/2011/01/iberogre-tutoriales-de-ogre3d-en-espanol

- **Web oficial de Ogre3D**

  IberOgre aparece mencionada en la cuarta edición de las noticias relacionadas con la comunidad de OGRE3D. La plataforma educativa fue visible en la portada de la web oficial del motor.

  http://www.ogre3d.org/2011/03/01/ogre-news-4

- **Twitter**

  El propio creador de OGRE3D, Steve Streeting, encontró la cuenta oficial de Twitter de **IberOgre** y la recomendó públicamente. Desde entonces el número de seguidores y lectores aumentó considerablemente.

- **Prensa**

  Tras el V Concurso Universitario de Software Libre varios medios de prensa escrita publicaron una noticia al respecto. El proyecto apareció en Viva Cádiz, Diario de Cádiz, La Voz, la web oficial de la Universidad de Cádiz y varios medios más.

Bibliografía

[1] aDeSe (Asociación Española de Desarrolladores y Editores de Software de Entretenimiento). Balance económico de la industria del videojuego 2010. http://adese.es/pdf/balanceeconomico2010.pdf.

[2] Antonio García Alba. Wikijuegos. http://wikis.uca.es/wikijuegos.

[3] Arseny Kapoulkine. pugixml. http://code.google.com/p/pugixml.

[4] Audacity. Web oficial de Audacity. http://audacity.sourceforge.net/.

[5] Beman Dawes y David Abrahams. Boost C++ libraries. http://www.boost.org/.

[6] Bruno Pagès. BoUML. http://bouml.free.fr.

[7] David Saltares Márquez. Creación de un entorno de desarrollo multiplataforma. http://wikis.uca.es/iberogre/index.php/Creaci%C3%B3n_de_un_entorno_de_trabajo_multiplataforma.

[8] David Saltares Márquez. IberOgre - Extender la gestión de recursos, audio. http://osl2.uca.es/iberogre/Extender_la_gesti%C3%B3n_de_recursos,_audio.

[9] David Saltares Márquez. IberOgre - Materiales. http://osl2.uca.es/iberogre/index.php/Materiales.

[10] Dia. Web oficial de Dia. http://projects.gnome.org/dia.

[11] Dimitri Van Heesch. Pagina oficial de Doxygen. http://www.doxygen.org.

[12] Emilio José Rodríguez Posada. AVBOT. http://code.google.com/p/avbot.

[13] Christer Ericson. *Real Time Collision Detection*. Morgan Kaufmann, 2005.

[14] Ian Millington John Fudge. *Artificial Intelligence for Games*. Morgan Kaufmann, 2009.

[15] Erich Gammma. *Design Patterns*. Addison Wesley, 1977.

[16] Gerardo Aburruzaga García. Make. Un programa para controlar la recompilación. http://uca.es/softwarelibre/publicaciones/make.pdf.

[17] GIMP. GNU Image Manipulation Program. http://gimp.org.

[18] GNU Project. gettext. http://www.gnu.org/s/gettext/.

[19] Jason Gregory. *Game Engine Architecture*. A. K. Peters Ltd., 2009.

[20] Roland Hess. *The Essential Blender*. Blender Foundation, 2009.

[21] Inkscape. Web oficial de Inkscape. http://inkscape.org/?lang=es.

[22] José Tomás Tocino García. Traducción de proyectos con GNU gettext en 15 minutos. http://osl.uca.es/files/hackathon/gettext.pdf.

[23] Julian Raschke. Gosu, 2D game development library. http://libgosu.org.

[24] Gregory Junker. *Pro OGRE 3D Programming*. Apress, 2006.

[25] Karl H. Beckers. XvidCap. http://xvidcap.sourceforge.net.

[26] Craig Larman. *Applying UML and Patterns*. Prentice Hall PTR, 2002.

[27] Gerardo Aburruzaga García Inmaculada Medina Bulo Francisco Palomo Lozano. *Fundamentos de C++*. Servicio de publicaciones de la UCA, 2009.

[28] MyGUI team. MyGUI. http://mygui.info.

[29] Noelia Sales Montes Emilio José Rodríguez Posada. Edición de wikis WikiMedia. http://osl2.uca.es/iberogre/images/5/51/Manual_wikimedia.pdf.

[30] Ogre3D. Ogre Wiki. http://ogre3d.org/tikiwiki/Home.

[31] OpenShot. Web oficial de OpenShot Video Editor. http://openshotvideo.com.

[32] Planner. Web oficial de Planner. http://live.gnome.org/Planner.

[33] RedIRIS. Forja de RedIRIS. http://forja.rediris.es.

[34] Sam Latinga. Simple DirectMedia Layer website. http://libsdl.org.

[35] StatsSVN. Web oficial de StatsSVN. http://statsvn.org.

[36] Subversion. Subversion. http://subversion.tigris.org.

[37] The GNU Project. GNU Debugger. http://www.gnu.org/s/gdb/.

[38] Valgrind. Valgrind. http://valgrind.org/.

[39] W3C. World Wide Web Consortium. http://w3c.es.

[40] Wikibooks. The Book of LaTeX. http://en.wikibooks.org/wiki/LaTeX.

[41] Wikimedia Foundation. Wikimedia. http://wikimedia.org.

[42] Wikipedia. Cinema 4D. http://en.wikipedia.org/wiki/Cinema_4D.

[43] Wikipedia. Cubic Hermite spline. http://en.wikipedia.org/wiki/Cubic_Hermite_spline.

[44] Wikipedia. DirectX. http://en.wikipedia.org/wiki/Directx.

[45] Wikipedia. Doom. http://en.wikipedia.org/wiki/Doom_(video_game).

[46] Wikipedia. Floyd-Warshall algorithm. http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm.

[47] Wikipedia. GNU Compiler Collection. http://en.wikipedia.org/wiki/GNU_Compiler_Collectiong.

[48] Wikipedia. Historia de los videojuegos. http://es.wikipedia.org/wiki/Historia_de_los_videojuegos.

[49] Wikipedia. Kinect. http://en.wikipedia.org/wiki/Kinect.

[50] Wikipedia. MediaWiki. http://en.wikipedia.org/wiki/MediaWiki.

[51] Wikipedia. Nintendo 3DS. http://en.wikipedia.org/wiki/Nintendo_3DS.

[52] Wikipedia. Nintendo DS. http://en.wikipedia.org/wiki/Nintendo_DS.

[53] Wikipedia. OpenGL. http://en.wikipedia.org/wiki/Opengl.

[54] Wikipedia. Personal Package Archive. http://en.wikipedia.org/wiki/Personal_Package_Archive.

[55] Wikipedia. Quake III Arena. http://en.wikipedia.org/wiki/Quake_III_Arena.

[56] Wikipedia. Separating Axis Theorem. http://en.wikipedia.org/wiki/Separating_axis_theorem.

[57] Wikipedia. Vim. http://wikis.uca.es/iberogre.

[58] Wrecked Games. OIS (Object Oriented Input System). http://sourceforge.net/projects/wgois.

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.
The "**publisher**" means any person or entity that distributes copies of the Document to the public.
A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.
You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.
The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.
In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.
However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.
Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.
Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.
"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.
"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.
An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.
The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.