Escuela Superior de Informática
Universidad de Castilla-La Mancha

CURSO DE EXPERTO EN DESARROLLO DE VIDEOJUEGOS

TRABAJO FIN DE CURSO

Razor Squadron

JULIO 2014

Escuela Superior de Informática
Universidad de Castilla-La Mancha

CURSO DE EXPERTO EN DESARROLLO DE VIDEOJUEGOS

TRABAJO FIN DE CURSO

Razor Squadron

León Romero, Daniel
Moreno Cañas, Dionisio
Sosa Díaz, Daniel
Valero Arroyo, Juan Jesús

JULIO 2014

Escuela Superior de Informática

CURSO DE EXPERTO EN DESARROLLO DE VIDEOJUEGOS

Calificación Trabajo Fin de Curso

CONVOCATORIA: Julio 2014

TÍTULO DEL PROYECTO: Razor Squadron

AUTORES:

León Romero, Daniel
Moreno Cañas, Dionisio
Sosa Díaz, Daniel
Valero Arroyo, Juan Jesús

TRIBUNAL:

Presidente: ______________________
Vocal: ______________________
Secretario: ______________________

FECHA DE DEFENSA: ______________________

CALIFICACIÓN: ______________________

PRESIDENTE VOCAL SECRETARIO

Fdo.: Fdo.: Fdo.:

Resumen

Este documento aborda el desarrollo de un videojuego de temática espacial utilizando el motor de renderizado *Ogre 3D* [1] e implementado en el *lenguaje C++*.

Para la realización de este videojuego, nos hemos basado principalmente en juegos del estilo de *Forsaken*, *Descent*, *Wing Commander*, *Strike Suit Zero*, *FreeSpace*, *X-Wing Rogue Scuadron*, *X-Com Interceptor*, juegos de simulación espacial. Este juego se basa en un mundo semiabierto en el cual el usuario se pondrá en la piel de un piloto de una nave espacial que tendrá que realizar multitud de misiones, desde destruir una flota de enemigos a defender unos cargueros espacial de mercancías, disponiendo para ello de varios tipos de armas.

Este proyecto se realizará bajo el motor de render llamado *Ogre 3D*, gracias a su versatilidad para la integración de prácticamente casi todos los componentes que componen un videojuego (sonido, periféricos, interfaz . . .).

Para dicho proyecto se creará un mundo semiabierto, donde tendrá lugar toda la acción del videojuego, se implementarán modelos de naves tanto para los enemigos como para los aliados, se dotará de Inteligencia Artificial a los enemigos y aliados mediante el uso de la biblioteca *OpenSteer* [15]. Se diseñará una interfaz futurista para la visualización de los objetivos del juego, para las armas, y para los posibles eventos que puedan aparecer en tiempo real.

Índice general

1. Introducción **1**
1.1. Metodología de desarrollo 1
1.2. Herramientas 2
1.3. Lenguajes de programación 3
1.4. Lenguajes de especificación de documentos 3
1.5. Bibliotecas 3
1.6. Control de versiones 4

2. Objetivos **5**
2.1. Objetivos funcionales 5
2.2. Objetivos no funcionales 6

3. Arquitectura de la solución **7**
3.1. Patrones de diseño utilizados 8
3.1.1. Patrón State 8
3.1.2. Patrón Factory Method 9
3.1.3. Patrón Observer 9
3.1.4. Patrón Mediator 10
3.1.5. Patrón Singleton 11
3.1.6. Patrón Adapter 12
3.2. Implementación de los menús 12
3.2.1. Pantalla de inicio 14
3.2.2. Menú principal 14
3.2.3. Menú de selección del nivel de dificultad 15
3.2.4. Menú de selección de nave 15
3.2.5. Menú de juego o interfaz del juego 15
3.2.6. Menú de pausa 16
3.2.7. Menú de fin de partida 17
3.2.8. Menú de ayuda 17
3.2.9. Menú de puntuaciones 18
3.2.10. Menú de créditos 18
3.3. Desarrollo de la Inteligencia Artificial 19

3.3.1. Inteligencia artificial con OpenSteer . . . 20
3.3.2. Integración en Ogre 3D . . . 24
3.4. Modelado de escenarios . . . 26
3.4.1. Estructura de la descripción del archivo de escena XML . . . 26
3.4.2. Exportador de Blender a XML . . . 35
3.4.3. Importador de XML a Ogre 3D . . . 36
3.5. Sistema de navegación del jugador . . . 37
3.6. Sistema de disparo . . . 38
3.7. Sistema de representación en pantalla (HUD) . . . 38
3.7.1. Sistema HUD . . . 39
3.7.2. Predicción del futuro posicionamiento enemigo . . . 39
3.7.3. Orientación espacial hacia el enemigo más cercano . . . 40
3.8. Sistema de lógica del juego . . . 41
3.8.1. Lógica de los objetivos . . . 41
3.9. Sistema de sonido . . . 42

4. Manual de usuario **43**

5. Conclusiones y trabajos futuros **49**
5.1. Conclusiones . . . 49
5.2. Trabajos futuros . . . 49

A. Imágenes del juego **51**

Bibliografía **55**

1. Introducción

1.1. Metodología de desarrollo 1
1.2. Herramientas 2
1.3. Lenguajes de programación 3
1.4. Lenguajes de especificación de documentos 3
1.5. Bibliotecas 3
1.6. Control de versiones 4

1.1. Metodología de desarrollo

La metodología que se decidió utilizar para el desarrollo es un método de integración continua y desarrollo incremental, ya que nos gustaba más la idea ir actualizando el desarrollo del juego y solucionando los posibles bug antes de implementar futuras actualizaciones.

Con esta integración continua y desarrollo incremental, nos ha permitido adaptarnos bien a las exigencias de las diferentes entregas de versiones y a los cambios necesarios.

Las fases que se llevaron a acabo en la integración continua y desarrollo incremental de todos los componentes y sistemas en el desarrollo del videojuego fueron los siguientes.

1. Implementación de los menús utilizando para ello el patrón State.

2. Creación del sistema de IA en *OpenSteer*.

3. Creación del sistema de navegación del jugador.

4. Integración de *OpenSteer* en el sistema de navegación.

5. Implementación del exportador e importador de escenarios.

6. Creación e integración de HUD.

7. Creación de la lógica de juego e integración final de todos lo sistemas y componentes.

1.2. Herramientas

- **Blender:** [6] es una herramienta de modelado 3D, se ha utilizado para modelar, reducir los polígonos de los modelos y para exportar estos modelos a *Ogre 3D*.
- **Dia:** [12] es una aplicación informática de propósito general para la creación de diagramas. Esta herramienta permite realizar diagramas de entidad-relación, de UML, de redes, de circuitos eléctricos, ... Se ha utilizado para realizar los diagramas de clases UML de la documentación.
- **Eclipse:** [7] es un entorno de desarrollo integrado multiplaforma de código abierto y permite desarrollar proyectos con varios lenguaje de programación. Utilizado para programar el videojuego desde el sistema operativo *Microsoft Windows*.
- **Evolus Pencil:** [5] es una herramienta de prototipo de interfaces libre multiplaforma, para crear bocetos de interfaces de múltiples plataformas. Se ha utilizado para hacer prototipos rápidos de la interfaz de los menús.
- **Gimp** [20] es una herramienta libre de edición y retoque de imágenes. Se ha utilizado para editar los overlays y los billboard del videojuego.
- **GNU Emacs:** [17] es el entorno de desarrollo de *GNU*. Se ha utilizado para programar el videojuego desde el sistema operativo *GNU/Linux*.
- **GNU GCC:** [22] es una colección de compiladores para lenguajes como *C/C++* y *Java*. Permite encontrar los errores léxicos, sintácticos y semánticos de los programas.
- **GNU GDB:** [18] es una herramienta de depuración que facilita la tarea de detección de errores lógicos dentro de los programas.
- **GNU Make:** [16] es una herramienta para la construcción de archivos, especificando las dependencias con sus archivos fuente. Es una herramienta genérica que permite generar cualquier tipo de archivo, a partir de los originales. Como generar imágenes png a partir de las originales en svg, pero el uso mas extendido es la compilación automática de ejecutables para *GNU/Linux* o para *Microsoft Windows* desde sus fuentes. Posee una característica muy importante que permite ahorrar mucho tiempo de compilación, ya que sólo compila los archivos que han sido modificados, no es necesario recompilar todo el proyecto cada vez que se realiza un cambio.
- **Mercurial** [11] es un sistema de control de versiones multiplataforma. las principales características son: rendimiento, escalabilidad, desarrollo distribuido (sin necesidad de un servidor), gestión robusta de archivos de texto y binarios y capacidades avanzadas de ramificación e integración. Tiene licencia GNU GPL2. Se ha utilizado para mantener un control de versiones del proyecto.

- **MinGW, Minimalist GNU for Windows:** [13] es una implementación de los compiladores *GCC* para la plataforma *Win32*.

- **Ogre Meshy:** [23] es una herramienta para la visualización de los archivos *mesh* de *Ogre 3D*.

- **Ogre Particle Lab** [19] es un editor de partículas para *Ogre 3D*.

- **Spacescape - Space Skybox Tool** [14] es una herramienta para la creación de *skybox* del espacio con estrellas y nebulosas.

1.3. Lenguajes de programación

- **C++:** es un lenguaje de programación orientado a objetos. Se ha utilizado para desarrollar el videojuego por que es el lenguaje utilizado por el motor gráfico *Ogre 3D*. Es un lenguaje muy potente y versátil que permite programar a bajo nivel y a alto nivel. Además, existen multitud de bibliotecas implementadas en *C++*.

- **Python:** se ha utilizado este lenguaje para exportar los escenarios de modelados *Blender* un archivo *XML*, debido a que *Blender* tiene integrado el interprete de *Python*. Además este lenguaje es muy sencillo y tiene gran potencia.

1.4. Lenguajes de especificación de documentos

- **BibTEX** [2] es una herramienta asociada a LATEX, para dar formato a las referencias de LATEX.

- **DOT (Graphviz)** [21] es un lenguaje descriptivo en texto plano, que permite realizar autómatas y diagramas de estado de forma muy sencilla.

- **LATEX** [3] es un editor de texto en texto plano que permite maquetar documentos técnicos y científicos rápidamente de forma profesional.

- **XML, eXtensible Markup Language (lenguaje de marcas extensible):** es un lenguaje de marcas utilizado para almacenar datos en forma legible. Se ha usado para almacenar los escenarios del juego, ya que se modelan en *Blender* y se utilizan en *Ogre 3D* a través de *C++*.

1.5. Bibliotecas

- **Ogre3D, Object-oriented Graphics Rendering Engine:** [1] es un motor para gráficos 3D libre multiplataforma, escrito en *C++*, debido a esto existen multitud de bibliotecas compatibles con *Ogre 3D*. Tiene licencia *LGPL*.

- **OIS, Object-oriented Input System Library:** [9] es una biblioteca para gestionar la entrada salida del videojuego, permite conectar multitud de dispositivos, como ratón, teclado o joystick. Esta biblioteca esta integrada de forma nativa en *Ogre 3D*.
- **OpenSteer:** [15] es una biblioteca de *C++* que permite implementar comportamientos de movimiento para personajes autónomos en visualizar, anotar y depurar estos comportamientos escribiendo un *plug-in* para la aplicación *OpenSteerDemo* basada en *OpenGL*. Se ha utilizado para implementar la inteligencia artificial del juego, tanto aliada como enemiga.
- **Parser Xerces-C++:** [8] proporcionar una biblioteca con funcionalidad para parsear, generar, manipular y validar documentos *XML* utilizando las *APIs DOM*, *SAX (Simple API for XML)* y *SAX2*. Con esta biblioteca se puede exportar las escenas *Blender* en un *XML* y luego importarse desde el videojuego.
- **SDL, Simple DirectMedia Layer:** [10] es una biblioteca multimedia y multiplataforma que permite integrar la música y los efectos de sonido en el videojuego. que nos permite la inclusión de sonidos, entre otras cosas.
- **STL, Standard Template Library:** es la biblioteca estándar de *C++*, esta biblioteca provee de cuatro componentes; algoritmos, contenedores, iteradores y funciones. Se ha utilizado para gestionar vectores de enemigos, aliados, disparos, ...

1.6. Control de versiones

El código fuente del proyecto se encuentra en *bitbucket*, que es un servidor para alojamiento de repositorios con interfaz web. La dirección web del repositorio es:

$https://bitbucket.org/cedv20132014/juegofinal$

Este servidor permite utilizar *Mercurial* [11] como sistema de control de versiones. Esta herramienta nos permite ver los cambios realizados en cada actualización, lo que permite hacer un seguimiento del desarrollo del proyecto.

2. Objetivos

2.1. Objetivos funcionales 5
2.2. Objetivos no funcionales 6

En este trabajo fin de curso se pretende realizar un videojuego de simulación espacial en primera persona, para ello se deben de cumplir todos estos objetivos funcionales y no funcionales para proporcionar al usuario una buena experiencia:

2.1. Objetivos funcionales

A continuación se relatan los requisitos minimos que debe satisfacer las funcionalidades descritas en el anteproyecto.

- Generar un escenario semiabierto tridimensional en el cual jugador pueda moverse en todas las direcciones y pueda interactuar con los enemigos u objetos presentes en él.
  - Implementar un exportador para convertir todos los elementos del escenario modelado en *Blender* en un archivo *XML* (eXtensible Markup Language).
  - Implementar un importador que permita convertir el *XML* anterior en un escenario para *Ogre 3D*.
- Sistema para el control de niveles y control de objetivos de cada nivel.
  - Evitar que maten a un aliado.
  - Matar a todos lo enemigos que estén en una determinada zona del escenario.
  - Destruir un objetivo específico.
- Sistema de Inteligencia Artificial (IA) que controle comportamientos aliados y enemigos para el desarrollo del juego (basado en *OpenSteer*).
  - Implementación de varios niveles de dificultad, poniendo mayor cantidad de enemigos o modificando los atributos de estos.
- Sistema de estados para controlar el funcionamiento de los menús del juego.

2.2. Objetivos no funcionales

En esta sección se muestran los objetivos no funcionales que debe de implementar el juego, para permitir una experiencia óptima de juego, pero que no son imprescindibles para disfrutar de él.

- Proporcionar una ambientación futurista y realista.
- Jugabilidad: se buscará que el jugador disfrute jugando.
- Dificultad: se implementarán varios niveles de dificultad para diferentes usuarios que jugarán.
- Inclusión de sonidos que permitan una buena inmersión en el juego: se incluirán músicas y efectos de sonido que harán que el usuario se sumerja en la atmósfera espacial.
- Proporcionar sensación de velocidad.
- Utilización de iluminación para el escenario.
- Personalización de la aeronave del jugador y naves aliadas, posibilidad de cambiar el color de la nave (modificando sus texturas) y/o de nave.

3. Arquitectura de la solución

3.1. Patrones de diseño utilizados **8**
3.1.1. Patrón State 8
3.1.2. Patrón Factory Method 9
3.1.3. Patrón Observer 9
3.1.4. Patrón Mediator 10
3.1.5. Patrón Singleton 11
3.1.6. Patrón Adapter 12
3.2. Implementación de los menús **12**
3.2.1. Pantalla de inicio 14
3.2.2. Menú principal 14
3.2.3. Menú de selección del nivel de dificultad 15
3.2.4. Menú de selección de nave 15
3.2.5. Menú de juego o interfaz del juego 15
3.2.6. Menú de pausa 16
3.2.7. Menú de fin de partida 17
3.2.8. Menú de ayuda 17
3.2.9. Menú de puntuaciones 18
3.2.10. Menú de créditos 18
3.3. Desarrollo de la Inteligencia Artificial **19**
3.3.1. Inteligencia artificial con OpenSteer 20
3.3.2. Integración en Ogre 3D 24
3.4. Modelado de escenarios **26**
3.4.1. Estructura de la descripción del archivo de escena XML 26
3.4.2. Exportador de Blender a XML 35
3.4.3. Importador de XML a Ogre 3D 36
3.5. Sistema de navegación del jugador **37**
3.6. Sistema de disparo **38**
3.7. Sistema de representación en pantalla (HUD) **38**
3.7.1. Sistema HUD 39

3.7.2. Predicción del futuro posicionamiento enemigo 39
3.7.3. Orientación espacial hacia el enemigo más cercano 40
3.8. Sistema de lógica del juego 41
3.8.1. Lógica de los objetivos 41
3.9. Sistema de sonido 42

3.1. Patrones de diseño utilizados

Los patrones de diseño son formas conocidas y probadas de resolver problemas de diseño que son recurrentes en el tiempo. De esta forma, reutilizando soluciones bien probadas y conocidas se ayuda a reducir el tiempo necesario para el diseño [4].

En esta sección se explicarán brevemente en que consisten los patrones utilizados durante el desarrollo de este videojuego y para que se han utilizado.

3.1.1. Patrón State

El patrón *State* es útil para realizar transiciones de estado e implementar autómatas respetando el principio de encapsulación [4].

Gracias al patrón *State* se pueden encapsular las transiciones que tiene un objeto a partir de los estímulos externos. La implementación consiste en una clase abstracta que representa el estado del objeto y contiene todas las posibles transiciones que se pueden realizar. Luego se tienen que hacer una clase por cada estado en el que se pueda encontrar el objeto. Heredando de la clase abstracta anterior e implementando únicamente los métodos de las transiciones necesarias desde ese estado.

En la figura 3.6 se puede ver el diagrama un diseño del patrón *State*, este ejemplo detalla los estados de las posibles acciones que puede realizar un personaje en un juego, así como las transiciones que se pueden realizar desde cada uno de los estados.

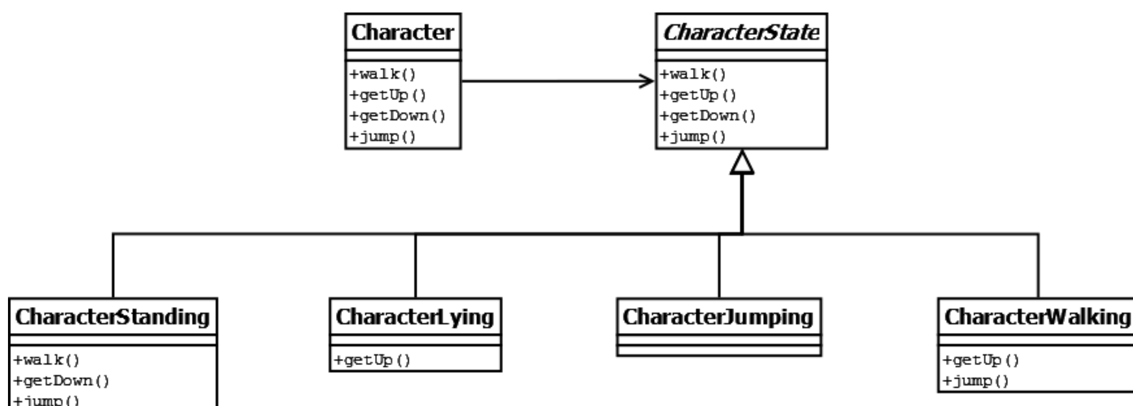


Figura 3.1: Diagrama *UML* de un ejemplo de patrón *State* [4].

El patrón de diseño *State* se ha utilizado en este proyecto para definir las posible transiciones que se pueden realizar entre los estados del menú del videojuego.

3.1.2. Patrón Factory Method

El patrón *Factory Method* consiste en la definición de una interfaz para crear instancias de objetos y permite a las subclases decidir cómo se crean dichas instancias implementando un método determinado. El problema que se resuelve es la creación de múltiples instancias de diferentes tipos de objetos abstrayéndose de la forma que se crean [4].

La figura 3.2 muestra un diagrama de clases para un ejemplo en el que se emplea el patrón *Factory Method* para crear ciudades y la población de estas [4].

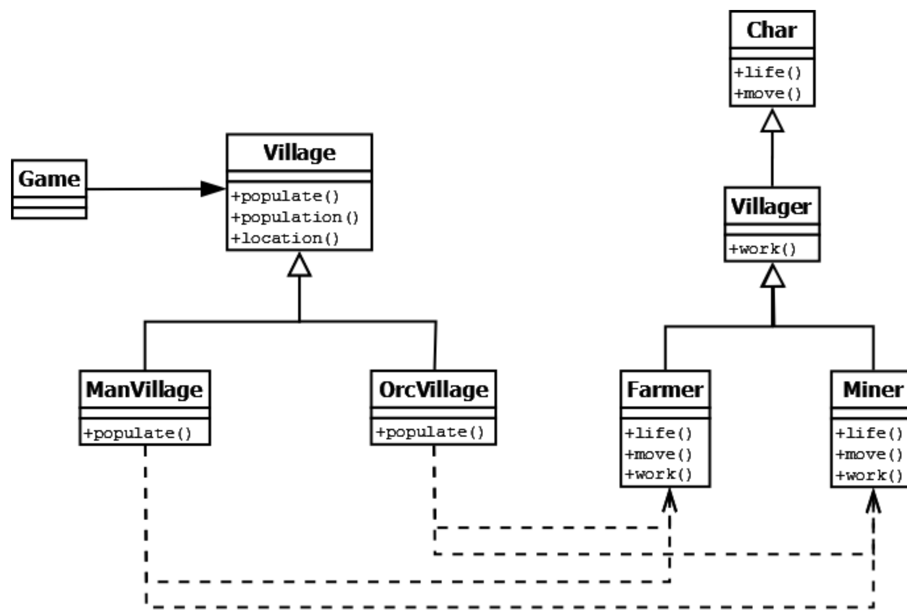


Figura 3.2: Diagrama *UML* de un ejemplo de patrón *Factory Method* [4].

Este patrón de diseño se ha utilizado en el desarrollo del juego para crear todo el escenario de juego a partir de un archivo *XML*. Se crean las nave, los obstáculos y las armas dependiendo de la configuración definida en el archivo *XML*.

3.1.3. Patrón Observer

El patrón *Observer* se utiliza para definir relaciones 1 a N de forma que un objeto pueda notificar y/o actualizar el estado de otros automáticamente. Este tipo de problemas ocurren cuando el estado un elemento tiene influencia directa sobre otros [4].

Uno de los principales objetivos que persigue el patrón *Observer* es el desacoplamiento de los componentes, objetos o clases. Este patrón sigue la filosofía de publicación/subscripción, cuando se produzca un evento el subscriptor se encargará de transmitir la información a todos los objetos subscritos a él. En el diagrama 3.3 se muestra un esquema general del patrón *Observer*.

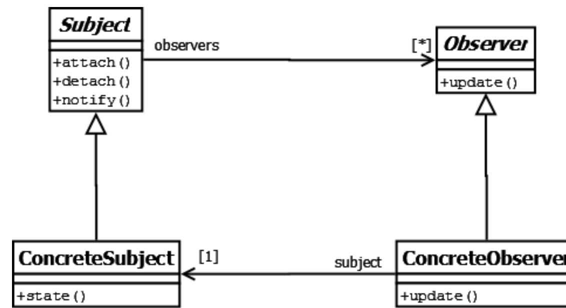


Figura 3.3: Diagrama general del patrón *Observer* [4].

3.1.4. Patrón Mediator

Hay ocasiones en las que muchos objetos interactúan entre ellos, esto produce una estructura muy compleja. Para evitar este efecto se utiliza el patrón *Mediator* que consiste en encapsular todo el comportamiento de comunicación de todo el conjunto de objetos dentro de un único objeto. El patrón mediador es el intermediario que define la interdependencias entre este conjunto de objetos.

Este tipo de patrón favorece un bajo acoplamiento y evita que los objetos se referencien unos a otras de forma explícita, además permite la reutilización ya que permiten variar la interacción sin modificar a otros elementos pertenecientes al conjunto de objetos que encapsula el patrón mediador.

En la figura 3.4 se muestra un diagrama *UML* con las clases necesarias para implementar este patrón. En este ejemplo, la clase *Mediator* define la interfaz para comunicarse con todos los objetos *Colleague*, la clase *ConcreteMediator* conoce a todo los objetos e implementa el comportamiento de comunicación entre ellos y por último las clases *ConcreteColleagues* conocen al objeto *Mediator* y se comunican con el resto de objetos a través de él.

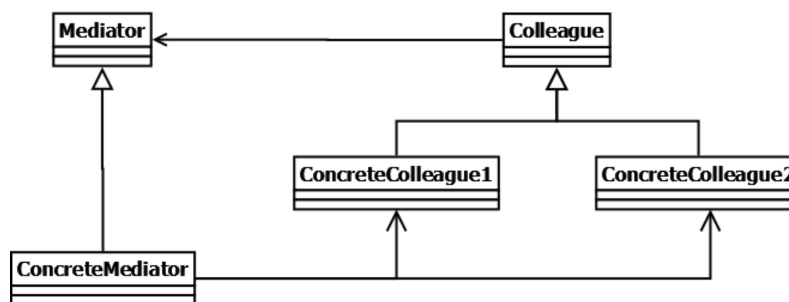


Figura 3.4: Diagrama general del patrón *Mediator*.

Se ha utilizado una solución mixta entre los patrones *Observer* y *Mediator* que se encarga de gestionar la comunicación entre las naves aliadas, naves enemigas y los disparos realizadas por estas.

Las naves añaden los misiles y los disparos del láser al *UpdateManager* mediante los

métodos de subscripción, siendo el *UpdateManager* el encargado de gestionar la comunicación de estas y de actualizar el estado de las naves dentro del contenedor, borrando la instancias si fuera necesario. Este esquema se puede ver en la figura 3.5.

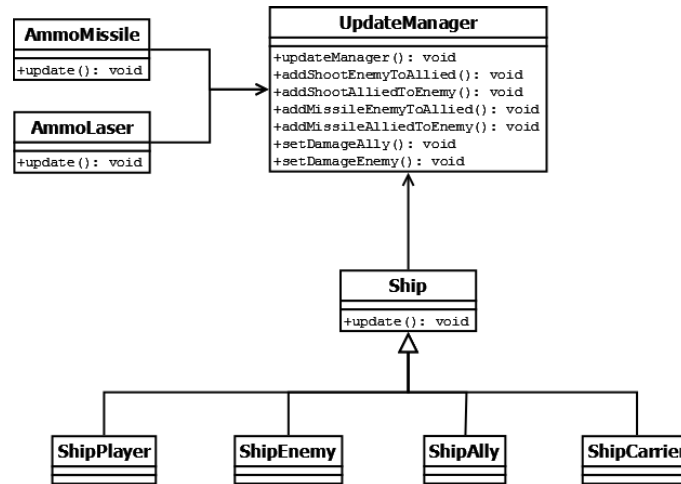


Figura 3.5: Diagrama *UML* de la comunicación utilizando los patrones *Observer* y *Mediator*.

3.1.5. Patrón Singleton

El patrón *Singleton* se utiliza cuando se necesita que sólo haya una instancia de un tipo de objeto. Puede ser útil una única instancia del objeto por diversos motivos como son prevención de errores, seguridad, ...

Para evitar que existan múltiples instancias de este tipo de objeto es necesario que los clientes no puedan acceder al constructor de la clase *Singleton*. Por este motivo, es necesario que el constructor sea al menos *protected* y sólo se debe de proporcionar un único punto de acceso para instanciar este objeto. En la figura 3.6 se muestra el diagrama de clases del patrón *Singleton*.

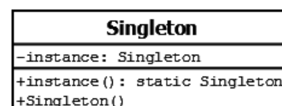


Figura 3.6: Diagrama *UML* de un ejemplo de patrón *Singleton* [4].

En el desarrollo de este proyecto se ha utilizado para implementa una clase auxiliar que almacenaba y transmitía la información entre los diferentes estados del menú de juego. También se ha utilizado en la implementación de las clases destinadas al tratamiento de los archivos *XML*.

Se han utilizado las clases *TrackMamager* y *SoundFXManager* proporcionadas durante este curso, a las cuales se les ha añadido una clase llamada *AudioManager* que envuelve las interfaces de estas clases unificándolas en una sola para facilitar su uso durante la implementación del videojuego.

Además de esto, la implementación interna de *Ogre 3D* hace uso este patrón para poder utilizar los manejadores de sus clases, como: *Root*, *OverlayManager*, *MaterialManager*, *MeshManager*, ...

3.1.6. Patrón Adapter

El patrón *Adapter* se utiliza para proporcionar una interfaz que cumpla con las demandas de los clientes y haga compatible otra interfaz que, a priori, no lo es [4].

Es posible que según se van avanzando los proyectos muy grandes las interfaces de sus componentes no sean las adecuadas para las necesidades actuales. También puede ocurrir que se necesite utilizar una biblioteca externa y la modificación de esta suponga un coste adicional.

Es posible solucionar estos problemas utilizando el patrón *Adapter*, creando una nueva interfaz para acceder a un determinado objeto, por lo que se proporciona un nuevo mecanismo de comunicación adaptada a las demandas del objeto cliente y del objeto que proporciona las funcionalidades. En la figura 3.7 se muestra un diagrama *UML* que muestra el esquema de clases de este patrón.

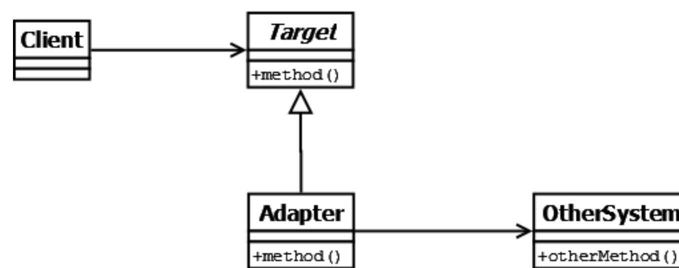


Figura 3.7: Diagrama *UML* de un ejemplo de patrón *Adapter* [4].

Como se ha indicado en la sección anterior, se ha utilizado el patrón *Adapter* para crear una nueva interfaz que contuviera las interfaces de todos los elemento de audio necesarios para el desarrollo del juego.

3.2. Implementación de los menús

Para realizar la implementación de los menús del videojuego se ha utilizado el patrón de diseño *State*, el cual se utiliza para que el menú del juego tenga diferentes comportamientos dependiendo del estado en que se encuentre. En la figura 3.8 se muestra el diagrama de clases que representa la implementación del patrón *State* y la herencia entre los estados del menú. Para realizar esta implementación nos hemos basado en el ejemplo proporcionado durante este curso.

Antes de realizar cada uno de estos menús se ha realizado un boceto de estos menús con la herramienta *Evolus Pencil*, para saber como se tenían que distribuir los botones e imágenes dentro de la pantalla.

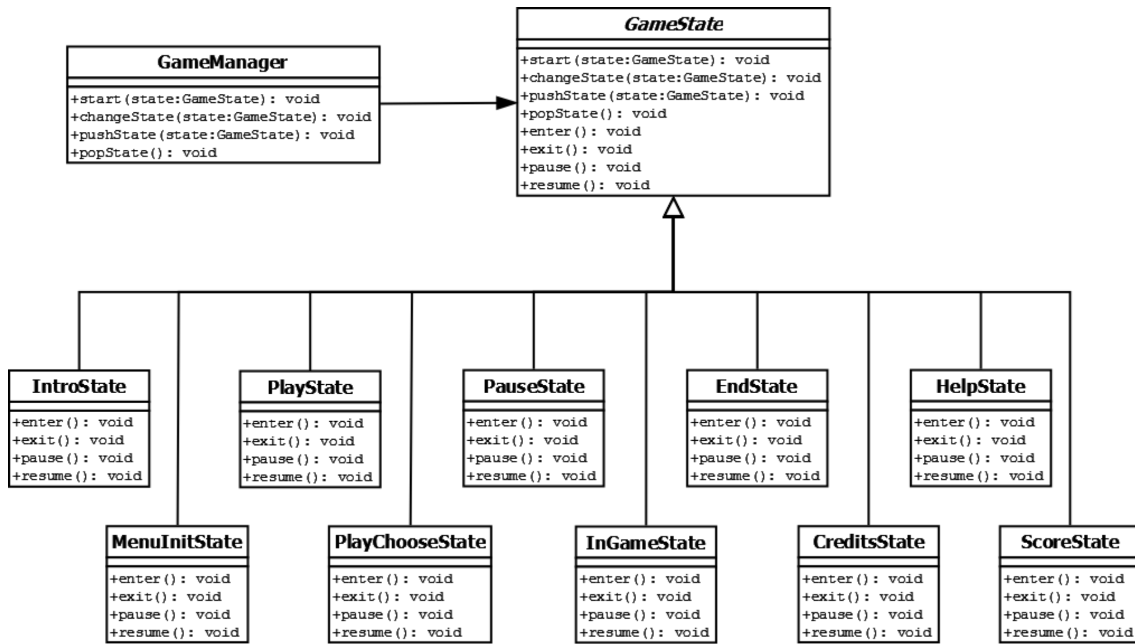


Figura 3.8: Diagrama *UML* del patrón *State* para la implementación de los menús.

La realización de todos los menús de este videojuego ha sido realizada mediante *overlays*, siguiendo una estética futurista de ciencia ficción. Para dar mayor retroalimentación al usuario y saber que elementos del menú son botones y cuales son simples imágenes, si ha implementado el cambio del color del botón al pasar el ratón por encima. Además, se han introducido efectos de sonido al pasar el ratón por encima de los botones o al pulsarlos.

En el siguiente diagrama 3.9 se pueden observar todos los estados implementados junto con sus respectivas transiciones.

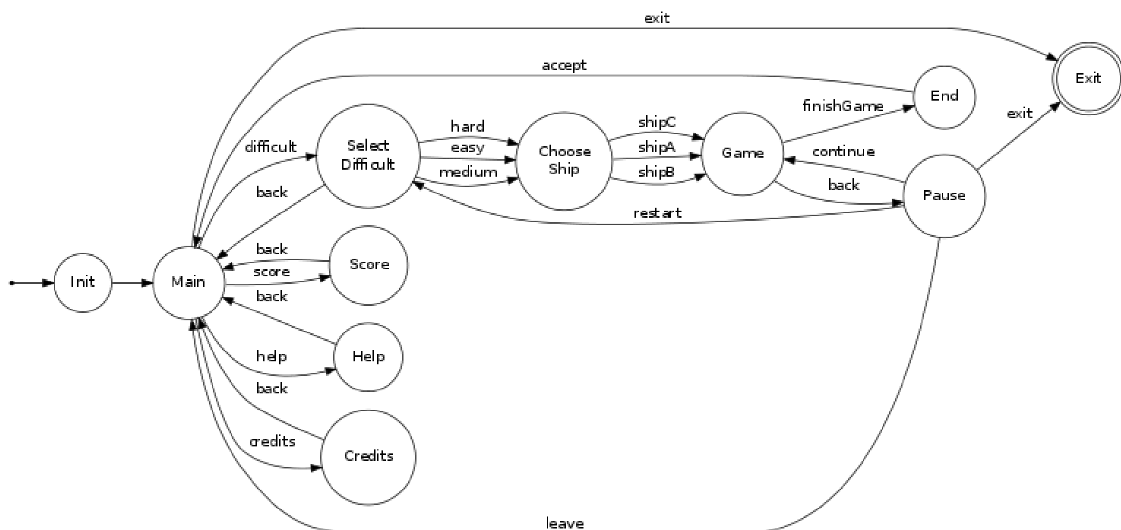


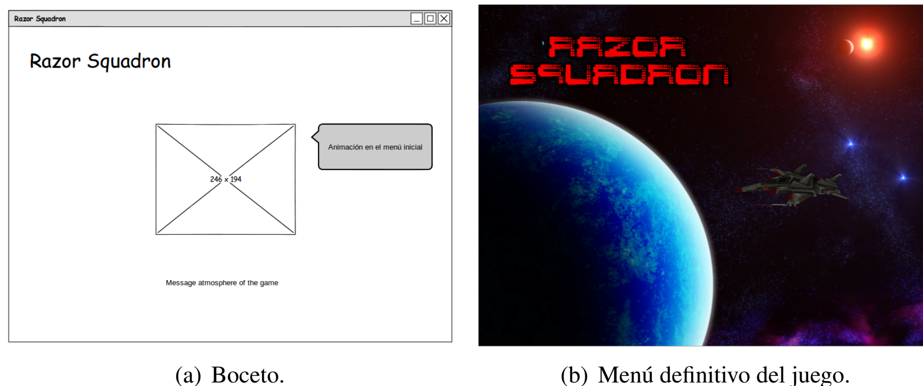
Figura 3.9: Diagrama de transiciones de los estados.

A continuación se explicarán todos los estados que se han utilizado para desarrollar este

videojuego Razor Squadron:

3.2.1. Pantalla de inicio

Este menú es la primera pantalla que aparece cuando se inicia el juego. Es el estado inicial del cual se podrá ir al menú principal del juego. En él se puede ver una pequeña animación de una nave rotando sobre si misma sobre un fondo del espacio. En la figura 3.10(a) se puede ver el boceto diseñado inicialmente, mientras que en la imagen 3.10(b) se muestra el menú definitivo dentro del juego.

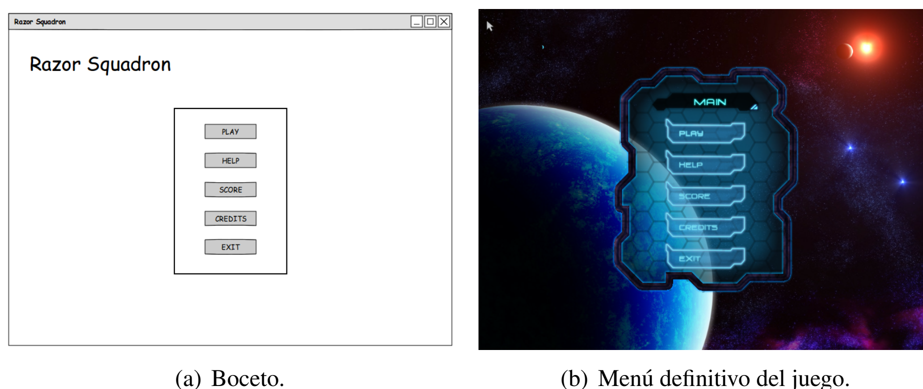


(a) Boceto. (b) Menú definitivo del juego.

Figura 3.10: Imagen de la pantalla de inicio.

3.2.2. Menú principal

Desde el menú de inicio sólo se puede transitar al menú principal. Este menú contiene todas las opciones del videojuego, desde aquí y según las elecciones del jugador se podrá llegar a los diferentes estados de juego: jugar, ayuda, puntuación, créditos y salir del juego (aunque ésta última opción no es propiamente un estado del juego, sino un botón para cerrar la aplicación).



(a) Boceto. (b) Menú definitivo del juego.

Figura 3.11: Imagen del menú principal.

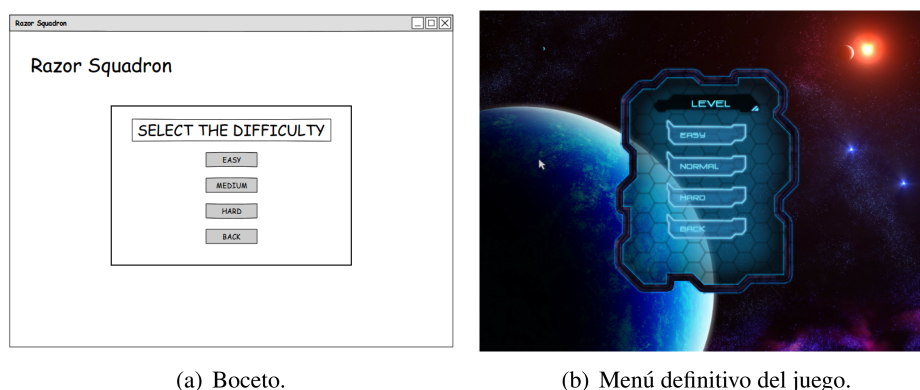
En las figuras 3.11 se pueden observar las posibles opciones mostradas en este menú principal del juego, en la imagen de la izquierda 3.11(a) se muestra el boceto diseñado inicialmente y

en la imagen de la derecha 3.11(b) el menú definitivo dentro del videojuego.

3.2.3. Menú de selección del nivel de dificultad

Desde el menú principal, a través de la opción jugar, se accede al menú de selección del nivel de dificultad. En este menú el usuario puede seleccionar el nivel de dificultad con el que desea jugar. La dificultad de estos niveles radica en la cantidad de enemigos y de aliados que hay en cada nivel, así como los valores de vida y de escudo de los enemigos. De esta forma, se han puesto mayor cantidad enemigos y más poderosos en el nivel difícil, mientras que en el nivel fácil se han puesto menos enemigos y más débiles.

Al igual que en los menús anteriores, también se ha realizado un boceto (figura 3.12(a)) antes de realizar el menú definitivo (figura 3.12(b)).



(a) Boceto. (b) Menú definitivo del juego.

Figura 3.12: Imagen del menú de selección de dificultad.

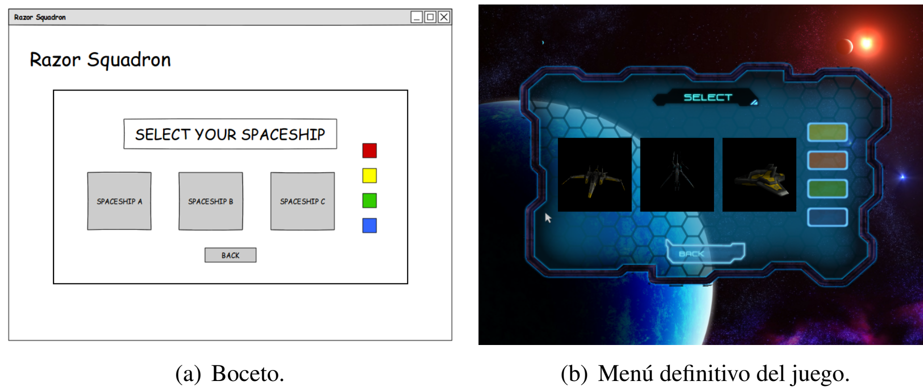
3.2.4. Menú de selección de nave

Después de seleccionar la dificultad se accede al menú de selección de nave, en el cual el jugador puede seleccionar su nave. En este menú se ha realizado un renderizado a textura de las tres posibles naves, las cuales giran cuando se pone el ratón sobre ellas. Además, se puede elegir el color que se desea que tenga la nave espacial, seleccionando en los botones de la derecha el color que se le aplica a la nave elegida.

En este menú también se realizó una prueba de concepto 3.13(a) para vez como distribuir los botones en la interfaz. En la imagen 3.13(b) se puede ver el diseño final dentro del juego.

3.2.5. Menú de juego o interfaz del juego

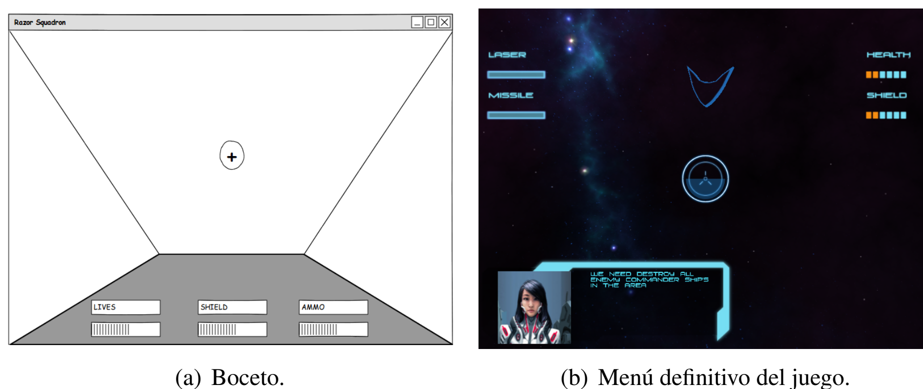
Este estado no es un menú propiamente dicho, sino que es la interfaz que se muestra al usuario con la información del juego. En esta interfaz se puede ver la mirilla, la salud del jugador, la munición disponible para sus armas, un cuadro de texto para indicar que se debe hacer en cada objetivo y un mensaje de alerta que se muestra cuando el jugador esta siendo perseguido por un misil.



(a) Boceto. (b) Menú definitivo del juego.

Figura 3.13: Imagen del menú de selección de nave espacial.

En las figuras 3.14 se observan el boceto de la interfaz diseñada inicialmente (la imagen de la izquierda 3.14(a)) y la interfaz que se ha diseñado e implementado dentro del juego (la imagen de la derecha 3.14(b)).



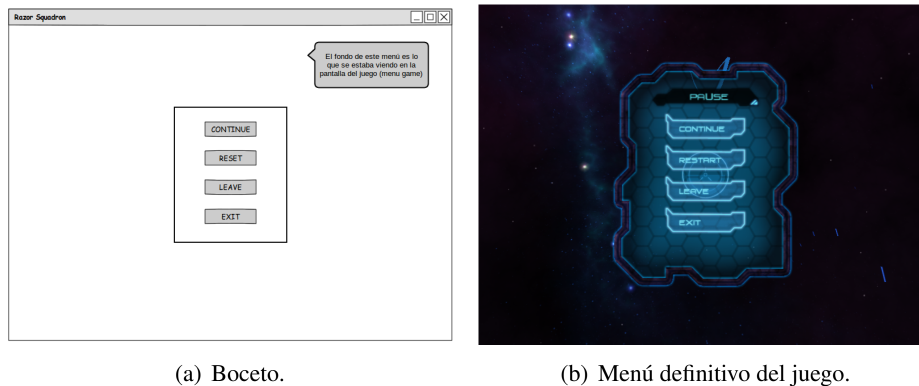
(a) Boceto. (b) Menú definitivo del juego.

Figura 3.14: Imagen de la interfaz de juego.

3.2.6. Menú de pausa

Este menú está oculto a simple vista, se puede acceder a él desde el menú de juego pulsando la tecla **P** y permite hacer una pausa en el juego. Cuando se activa este menú se habilitan las siguientes opciones: continuar la partida (se realiza la transición al estado anterior y se continúa la partida por el punto en el que se pausó), reiniciar la partida (se hace la transición al estado de selección de la dificultad), abandonar la partida (transición al menú principal) y por último la opción de salir del juego.

En las figuras 3.15 se observan el boceto del menú diseñado inicialmente (en la izquierda, la imagen 3.15(a)) y el menú que se ha implementado dentro del juego (a la derecha, la imagen 3.15(b)).



(a) Boceto. (b) Menú definitivo del juego.

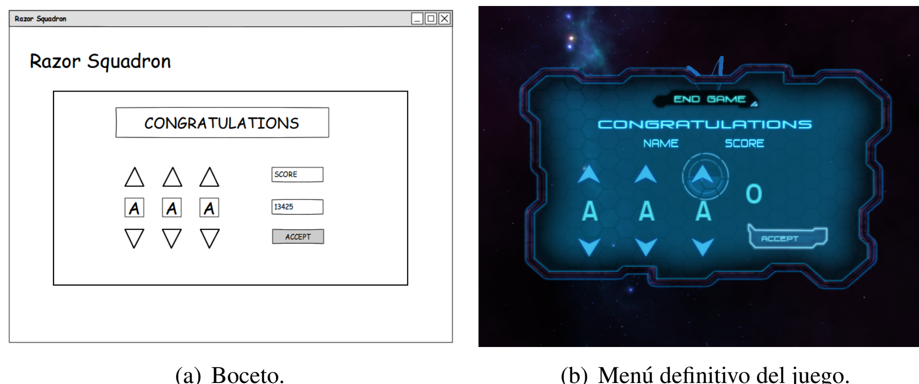
Figura 3.15: Imagen del menú de pausa.

3.2.7. Menú de fin de partida

Una vez que se ha terminado el juego, se realiza la transición al estado de fin de juego. Esta transición se puede realizar de dos maneras; (a) en la cual el jugador ha completado todos los objetivos de todos lo niveles o (b) cuando la vida del jugador principal llega a cero.

En este menú se escribe el nombre del usuario, mediante la selección de las iniciales del jugador con las flechas del menú, simulando el estilo retro de los videojuegos *arcade*. También se muestra la puntuación obtenida en el juego. Una vez escrito el nombre del usuario, se pulsa el botón de aceptar y se produce una transición al menú principal para jugar la próxima partida.

En la siguiente imagen 3.16 se muestran, el boceto inicial de este menú en la imagen 3.16(a) y el resultado final en el videojuego desarrollado 3.16(b).



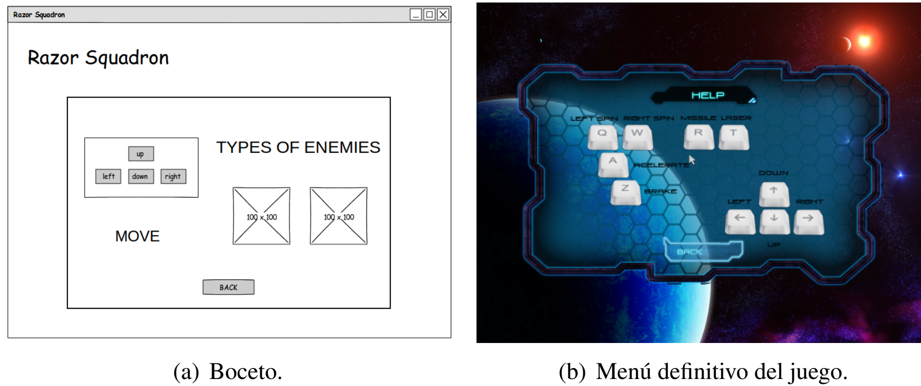
(a) Boceto. (b) Menú definitivo del juego.

Figura 3.16: Imagen del menú de fin de partida.

3.2.8. Menú de ayuda

El menú de ayuda es otra de las opciones disponibles desde el menú principal, en este caso se muestra una pantalla que explica lo controles de la nave del jugador. En la figura 3.17 se muestran tanto el boceto inicial como el pantalla final en el videojuego.

Como se puede observar en la figura 3.17(b), la aceleración de la nave se realiza pul-



(a) Boceto. (b) Menú definitivo del juego.

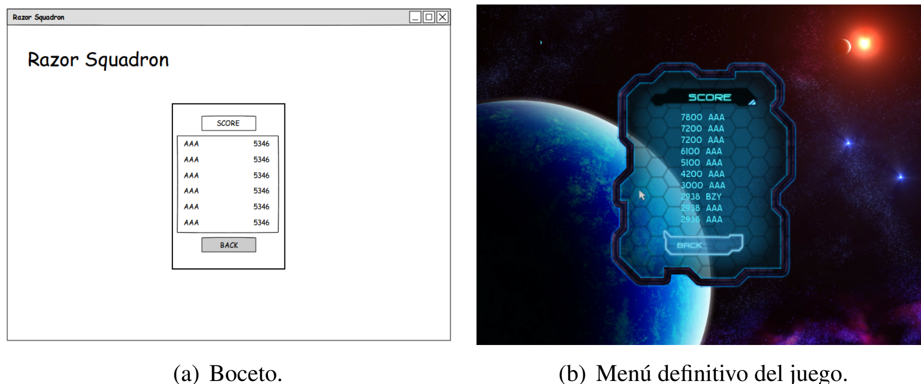
Figura 3.17: Imagen del menú de ayuda.

sando la tecla **A** y para frenar la nave debemos pulsar la tecla **Z**. Para rotar en el eje longitudinal (maniobra acrobática de tonel rápido) se utilizan las teclas **Q** y **W** hacia la izquierda o derecha respectivamente. También se pueden realizar los movimientos de giro hacia la izquierda o derecha y de ascender y descender mediante los cursores. Finalmente, para realizar los disparos del láser tendremos que pulsar la tecla **T** y para realizar los disparos de los misiles la tecla **R**.

3.2.9. Menú de puntuaciones

Desde el menú principal también se puede acceder a la opción de puntuaciones máximas, este menú es un listado con las diez mejores puntuaciones representadas con el nombre introducido en el menú de fin de partida y la puntuación obtenida en la partida.

Se puede ver el menú desarrollado para el videojuego en la imagen 3.18(b), así como el boceto inicial en la imagen 3.18(a).



(a) Boceto. (b) Menú definitivo del juego.

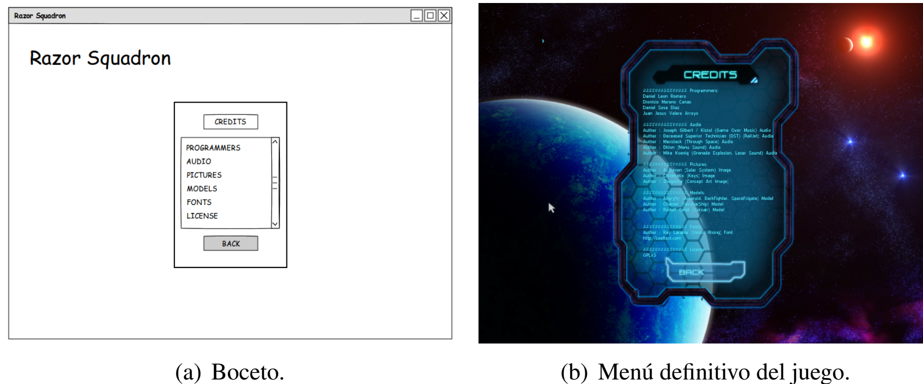
Figura 3.18: Imagen del menú de puntuaciones.

3.2.10. Menú de créditos

Desde última opción existente en el menú principal se accede al menú de créditos, en el cual están reflejados los nombres de los autores del videojuego, así como todas las referencias de

los sitios web o a los autores de donde se han cogido recursos que necesitan ser reconocidos para su realización.

Como en todos los menús explicados anteriormente, sea realizado primero un boceto mostrado en la figura 3.19(a), a partir del cual luego se ha diseñado el menú definitivo dentro del videojuego mostrado en la imagen 3.19(b).



(a) Boceto.

(b) Menú definitivo del juego.

Figura 3.19: Imagen del menú de créditos.

3.3. Desarrollo de la Inteligencia Artificial

La inteligencia artificial es un factor crítico en los videojuegos actuales, ya que el jugador tiene que percibir que los movimiento y las acciones de los enemigos o de los aliados no humanos tienen cierta lógica. Debido a este motivo se decidió dedicar bastante tiempo en el desarrollo de la IA para los personajes no jugadores para que el jugador principal pudiera disfrutar de una experiencia de juego similar a estar jugando con enemigos o aliados humanos.

Lo primero que se hizo fue decidir cuantos tipos de naves no controladas por el jugador se iban a diseñar para este videojuego, se optó por dos tipos de enemigos y otros dos tipos de aliados:

- **Enemigos:** son las naves que se tienen destruir para completar los objetivos de misión. Existen dos tipos de enemigos; soldados y jefes, aunque el comportamiento de la IA de ambos modelos a efectos prácticos es el mismo, no es el mismo para la lógica del juego, ya en algunos objetivos será suficiente con destruir a un tipo concreto de enemigo. Además, las naves de tipo jefe, también son más difíciles de destruir porque tienen mejor equipamiento.

- **Aliados:** son las naves que cooperan con el usuario. Existen dos tipos de estos aliados; aliados y cargueros, el primero de estos ayuda al jugador a completar los objetivos destruyendo a las naves enemigas, mientras que el segundo tipo son las naves a las que hay que proteger durante un trayecto por el espacio para terminar la misión.

Para implementar todos estos tipos de naves no jugadores se utilizó una herencia, desde la clase padre *Ship* a las clases hijas, cada una de las especializaciones descritas anteriormente.

En la figura 3.20, se puede observar un diagrama *UML* que representa esta herencia. Como el comportamiento de todos los enemigos es el mismo se decidió utilizar la misma clase para todos ellos, pero para diferenciar el tipo del enemigo dentro de la lógica del juego se han utilizado dos vectores diferentes.

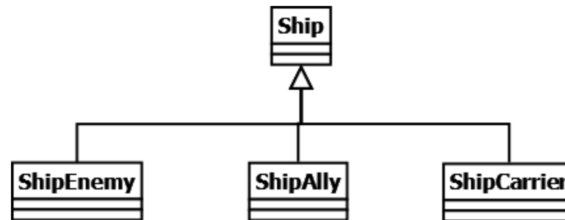


Figura 3.20: Diagrama UML de la herencia de las naves IA.

3.3.1. Inteligencia artificial con OpenSteer

En las sesiones de teoría de IA del Curso de Experto en Desarrollo de Videojuegos se mostró el uso de la biblioteca *OpenSteer* para la implementación del comportamiento del movimiento en los personajes no jugadores de los videojuegos. Nos pareció una potente herramienta para proporcionar este comportamiento a las naves del videojuego que estábamos desarrollando.

Esta biblioteca no tiene mucha documentación en su página web, pero tiene implementada la aplicación *OpenSteerDemo* basada en *OpenGL*. Esta aplicación permite al programador desarrollar comportamientos de movimiento para personajes autónomos de los videojuegos en forma de *plug-ins*, y de una manera rápida y sencilla se puede visualizar, anotar y depurar el comportamiento de estos.

La inteligencia artificial desarrollada para este proyecto está basada en un comportamiento de tipo *boids* (o de IA colectiva), este tipo de comportamiento se basa en el seguimiento de una serie de rutinas:

- No acercarse ni alejarse mucho de los demás *boids* existentes en el espacio virtual.
- Intentar mantener igual la velocidad y la dirección de los demás *boids* del grupo.
- Intentar moverse siempre hacia el centro de los *boids* que se encuentren en la vecindad inmediata.

Partiendo de los comportamientos implementados en esta biblioteca como son los comportamientos de deambular, perseguir y evitar las colisiones con el resto de enemigos o de obstáculos de la escena, se han desarrollado los comportamientos para todos los tipos de naves explicadas anteriormente.

Comportamiento de los cargueros

El comportamiento implementado en los cargueros consiste en seguir una ruta predeterminada. El objetivo de estos consiste en llegar hasta el destino, tratando de esquivar los obstáculos

que vaya encontrando es su camino, ya sean naves o asteroides. Además, se ha programado un comportamiento de huir que se activa si está siendo atacado. Una vez que se ha alcanzado el destino, el carguero realiza el movimiento de deambular alrededor del destino.

El diagrama de la figura 3.21 representa el comportamiento de los aliados de tipo carguero. El único estado que no permite transiciones al resto de los estado es el estado de vagar, porque una vez terminado su recorrido ya no tiene que volver nunca más al estado de avanzar.

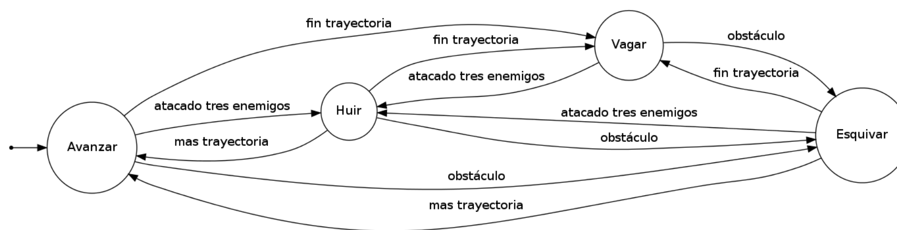


Figura 3.21: Diagrama de estados para el comportamiento de los carguero.

El estado de huir consiste en activar los propulsores de la nave durante un periodo de tiempo, para poder activarse este estado es necesario que la nave esté siendo atacada por tres naves enemigas simultáneamente.

Comportamiento de los enemigos

La idea del comportamiento para el movimiento de los enemigos era que cada enemigo tuviera una zona de vigilancia en la que se estuviera moviendo continuamente y que sólo pudiera atacar y perseguir al jugador principal o a sus aliados cuando estuvieran dentro o muy cerca de esta zona. También se tuvo en cuenta la posibilidad se que el jugador se alejase de la zona de vigilancia del enemigo, en ese caso el enemigo dejaría de atacar al jugador y regresaría a su área. En la imagen 3.22 se observa la máquina de los posibles estados del comportamiento de los enemigos con sus correspondientes transiciones.

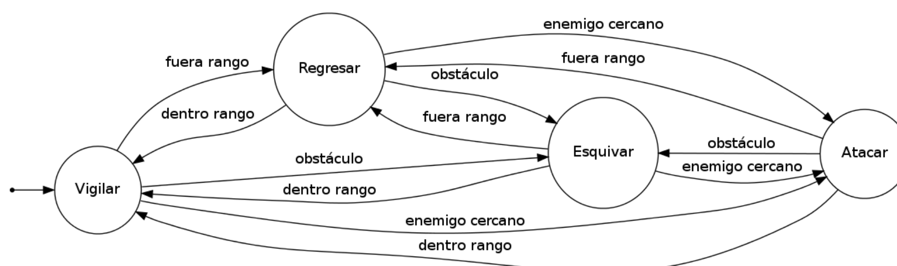


Figura 3.22: Diagrama de estados para el comportamiento de los enemigos.

Hay que remarcar un detalle al hacer el cambio de estado a atacar, sólo pueden atacar un máximo de tres enemigos simultáneamente al jugador o a una nave aliada. De esta forma se evita que en escenarios muy poblados de enemigos ataquen todos al mismo objetivo.

Comportamiento de los aliados

El objetivo principal de los aliados es de proteger a la nave que tienen asignada, esta nave puede ser el usuario o un carguero. En la figura 3.23 se muestra una máquina de estados que representa el comportamiento del aliado. Esta nave se mueve siguiendo a la nave que tiene que proteger, si durante este movimiento un enemigo atacara a la nave asignada, el aliado cambia al estado proteger. Cuando el enemigo sea destruido o regrese a su posición el aliado pasará de nuevo al estado seguir. Otro estado es que un enemigo ataque al propio aliado, en este caso entrará al estado de defenderse.

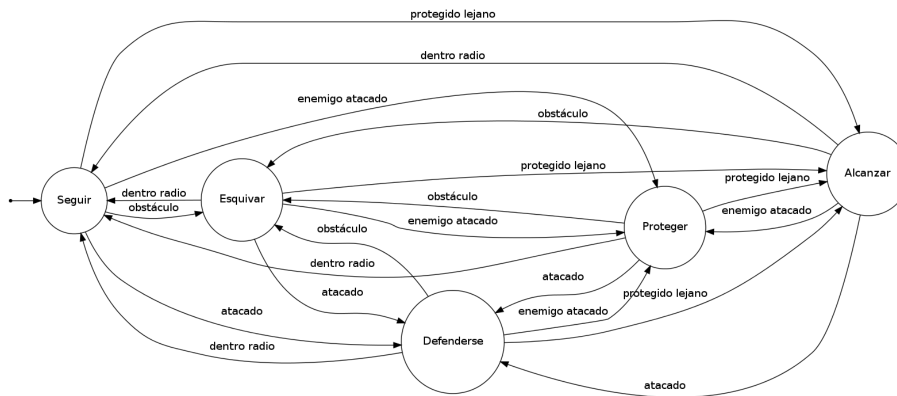


Figura 3.23: Diagrama de estados para el comportamiento de los aliados.

Relacionado con los dos últimos estados explicados: atacar y defender, tiene preferencia el estado de atacar, es decir, si hay enemigos atacando a la nave protegida y al propio aliado. primero atacará al enemigo que este atacando a la nave protegida y luego se de defenderá del enemigo que le está atacando a él.

Resultado en OpenSteer

Además, de los comportamientos detallados anteriormente para cada una de las naves, todas ellas implementan el comportamiento de vagar para evitar que las trayectorias de sus movimientos sean muy rectas y el de evitar las colisiones con el resto de elementos de la escena, ya sean obstáculos u otras naves.

La primera implementación que se realizó de estos comportamientos fue en 2D, ya que de esta forma se podía comprobar mejor que las naves realizaran correctamente sus movimientos. La imagen 3.24 es una pantalla del *plug-ins* implementado para la aplicación *OpenSteerDemo*, en la que se pueden ver todos los tipos de nave diseñados anteriormente. La nomenclatura de los colores del dibujo es la siguiente:

- **Enemigos:** son de color verde y tiene su área de influencia representada por un círculo de color rojo.

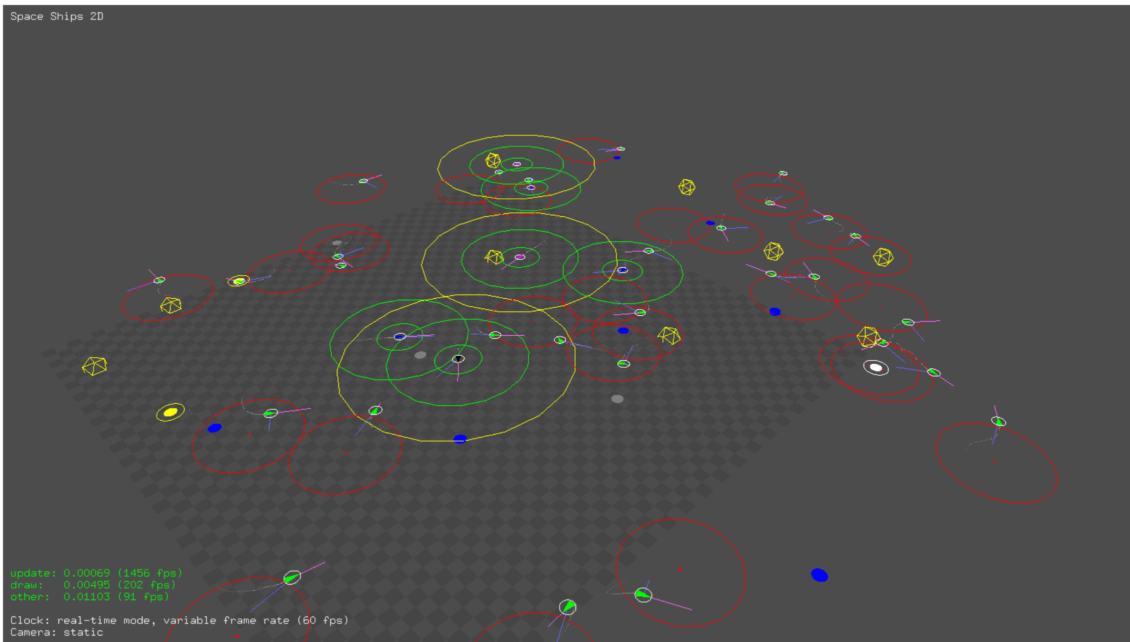


Figura 3.24: Captura del *plug-ing* del comportamiento en 2D.

- **Cargueros:** son de color magenta y tiene unos círculos verdes al rededor de ellos. Si un enemigo entra entre ellos atacará al carguero.
- **Aliados:** son de color azul y también están rodeados de dos círculos verdes.
- **Jugador:** se ha utilizado un objeto *Mock* para simular el comportamiento del jugador, el comportamiento que se le ha proporcionado es el mismo del carguero. Se representa igual que los cargueros pero es de color negro.
- **Obstáculos:** son poliedros de color amarillo.

Después de implementar esta versión para los comportamientos en 2D y verificar su correcto funcionamiento, se tuvo que realizar una segunda versión con el diseño en 3D ya que los objetos debían de moverse en un escenario tridimensional. El resultado de la implementación en 3D se puede observar en la imagen 3.25. En esta imagen se sigue al misma nomenclatura de representación que en la versión de 2D.

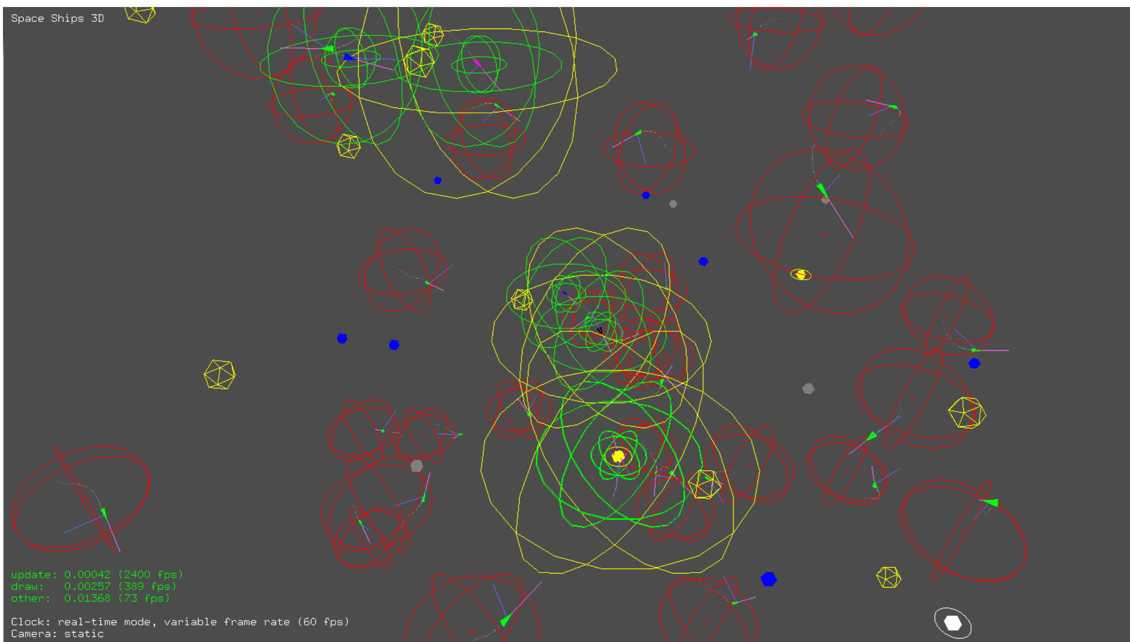


Figura 3.25: Captura del *plug-ing* del comportamiento en 3D.

Figura 3.26: Captura del comportamiento de las naves en 2D utilizando *OpenSteer* y *Ogre 3D*.

3.3.2. Integración en Ogre 3D

Una vez realizada la implementación de los comportamientos de todos los tipos de naves utilizando la biblioteca *OpenSteer*, se tuvo que integrar este diseño entro del motor de renderizado *Ogre 3D*, sobre el que se ha desarrollado este proyecto.

Al igual que para desarrollar el *plug-ins* del comportamiento en *OpenSteer*, primero

se desarrolló la versión en 2D para comprobar el correcto funcionamiento y facilitar su posterior inclusión de 3D. En la imagen 3.26 se muestra el resultado obtenido después de integrar el comportamiento en 2D de los personajes autónomos dentro de *Ogre 3D*.

Después integrar en *Ogre 3D* el comportamiento implementado y de comprobar que funcionaba correctamente dentro en 2D, se migró la versión de tres dimensiones que es la que se utilizó para la versión definitiva de este videojuego. En la figura 3.27 se observa el comportamiento para todas la naves dentro dentro de *Ogre 3D*.

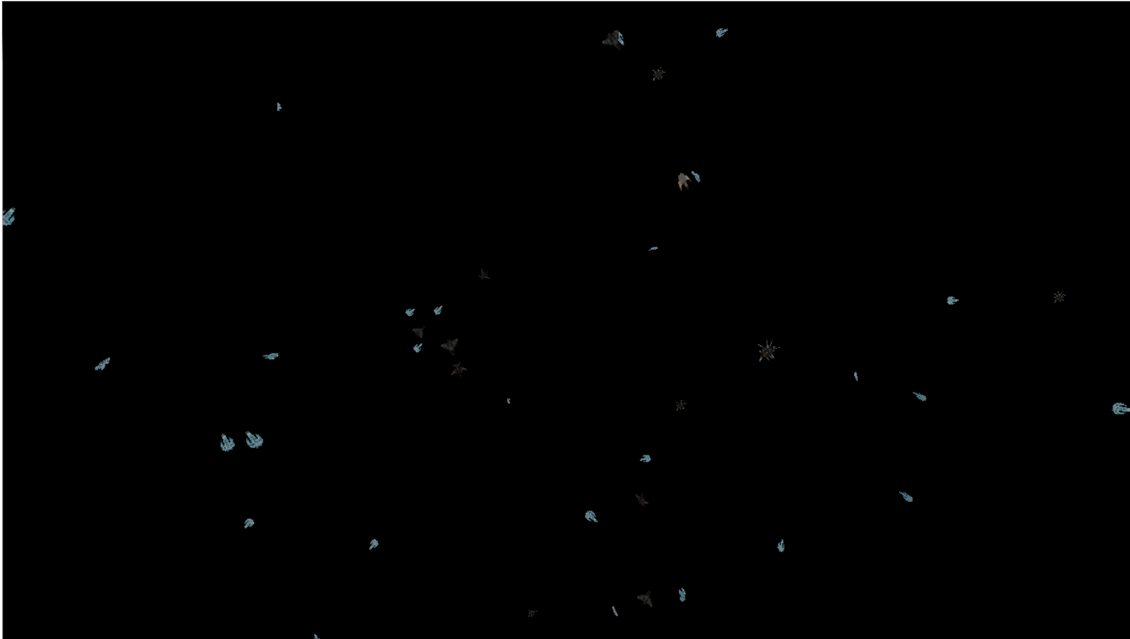


Figura 3.27: Captura del comportamiento de las naves en 3D utilizando *OpenSteer* y *Ogre 3D*.

Los cambios más importantes que se han tenido que realizar para integrar el comportamiento de *OpenSteer* en *Ogre 3D* son:

- **CurrentTime**: tiempo actual de simulación, este valor es necesario para actualizar las posiciones de las naves dentro de la lógica de *OpenSteer* y no existe en *Ogre 3D*. El valor de esta variable se ha calculado incrementándole en todas las iteraciones el valor *deltaTime*.

- **Posición de los objetos:** hay que realizar una conversión entre los vectores que posicionan los objetos en la escena, *OpenSteer* utiliza el tipo *Vec3* mientras que *Ogre 3D* utiliza *Vector3*.

- **Orientación de los objetos:** la orientación de los objetos en *OpenSteer* está definida en con un *Vec3*, se obtiene llamando a la función *forward()*. Mientras que en *Ogre 3D* se define con un cuaternio. Por lo que se tiene que realizar otra conversión para calcular la dirección.

3.4. Modelado de escenarios

El modelado de los escenarios se ha realizado mediante *Blender* utilizando objetos básicos como cajas para ubicar los distintos tipos de elementos de la escena desde el *script* implementado en *Python* poder exportar la escena en formato en *XML* para importarla en *Ogre 3D*.

3.4.1. Estructura de la descripción del archivo de escena XML

El diseño de la estructura del archivos de escena *XML* fue un punto que consumió bastante tiempo, debido a que se pretendía almacenar mucha información en él. En cada uno de estos archivos *XML* se almacena la información relativa a un nivel del juego.

La idea principal que se quería plasmar en el diseño de este archivo de definición de la escena era que todos los elementos pudieran ser lo más configurables posibles, de esta forma se conseguiría que no fueran iguales todos los objetos de un determinado tipo. Por ejemplo; variando la configuración del *XML* para los enemigos, conseguiríamos que cada uno pudiera tener un tipo de arma o un modelo para representarlo en *Ogre 3D* diferente. Aunque algunas opciones de estas configuraciones no han sido posibles añadir definitivamente por falta de tiempo, en esta sección se explicará el *XML* completo.

Cada nivel contiene unas características o restricciones del mundo que conforman el escenario, una lista de armas, una lista de objetos y una lista de objetivos. Dentro de cada objetivo se tiene la información relativa a ese objetivo concreto, mientras que el resto de los secciones del *XML* del nivel contiene información perteneciente a todo el nivel. A lo largo de los siguientes apartados se irán describiendo todas las secciones de estos archivos.

Restricciones del nivel

El listado 3.1 muestra un ejemplo de las restricciones del escenario definido para un nivel, en el se pueden ver los siguientes campos:

- **radiusVisibility:** radio máximo de visibilidad.
- **radiusMax:** radio máximo del espacio.
- **timerRespawn:** tiempo de reaparición si el jugador se sale del escenario.
- **deactivacionDistance:** distancia de activación de los enemigos.
- **skybox:** nombre del material que se utiliza para el *skybox* de la escena.
- **position:** posición del origen de coordenadas del escenario, a partir de la cual se genera.

Listing 3.1: Restricciones de la escena.

```
<constraint radiusVisibility="50" radiusMax="45" timerRespawn="3"
   deactivationDistance="32" skybox="SkyboxMat">
 <position>
```

```
    <x>0.00000</x><y>0.00000</y><z>0.00000</z>
  </position>
</constraint>
```

Armas

En el *XML* definido inicialmente se incluyó la configuración de las armas, para generar múltiples tipos de armas modificando su configuración con rapidez. En el listado 3.2 se muestras las posibles opciones de configuración de las armas.

- **index:** identificador del arma.
- **mesh:** material que se utilizará en el juego para representar los disparos de este arma.
- **speed:** velocidad de movimiento del disparo.
- **damage:** daño del arma.
- **scope:** alcance del arma.
- **timerShots:** cadencia de disparo.
- **ammo:** munición máxima del arma.
- **type:** tipo de arma (láser, misil, . . .)
- **position:** posición relativa al jugador a partir de la cual sale el disparo.
- **rotation:** rotación del material utilizado como disparo.
- **scaled:** escalado del material disparado.

Listing 3.2: Definición de las armas.

```
<weapons>
  <weapon index="1" mesh="laser.material" speed="4.5" damage="42" scope="50"
     timerShots="0.6" ammo="200" type="1" >
    <position>
      <x>0.00000</x><y>0.00000</y><z>0.00000</z>
    </position>
    <rotation>
      <x>0.00000</x><y>0.00000</y><z>0.00000</z><w>1.00000</w>
    </rotation>
    <scaled>
      <x>1.00000</x><y>1.00000</y><z>1.00000</z>
    </scaled>
  </weapon>
  ...
</weapons>
```

Obstáculos y objetos

Al igual que sucede con las armas, se pretendía que fueran configurables en función de una serie de características definidas en el archivo *XML*, aunque en el listado 3.3 se muestran todas las opciones de configuración, por falta de tiempo no se han podido utilizar todas estas características.

- **index:** identificador del objeto.
- **mesh:** material que se utilizará en el juego para representar el objeto.
- **collision:** este atributo indica si el jugador puede colisionar con el objeto (0/1).
- **destroyable:** indica se el obstáculo explota y queda destruido cuando se colisiona con el.
- **weapon:** indica se el objeto con el que choca es un tipo de arma o munición para una ya existente. Por ejemplo; 0 el objeto no es una arma, 1 láser, 2 bomba, 3 misil, . . . Este número hace referencia a los identificadores de las armas definidas en el escenario.
- **ammo:** si el objeto es un arma, munición que proporciona.
- **damage:** daño que recibe la nave que colisione con el objeto, si fuera un arma el daño sería cero.
- **type:** el tipo del objeto.
- **position:** posición del objeto u obstáculo dentro de la escena *Ogre 3D*.
- **rotation:** rotación del modelo que representa al obstáculo dentro de *Ogre 3D*.
- **scaled:** escalado del modelo.

Listing 3.3: Definición de los objetos.

```
<objects>
  <object index="1" mesh="Sphere.mesh" collision="1" destroyable="1" weapon="0
      " ammo="40" damage="35" type="1">
    <position>
      <x>99999.089355</x> <y>829.413940</y> <z>996.598877</z>
    </position>
    <rotation>
      <x>0.000000</x> <y>0.000000</y> <z>-0.000000</z> <w>1.000000</w>
    </rotation>
    <scaled>
      <x>55000.369200</x> <y>55000.369200</y> <z>55000.369200</z>
    </scaled>
  </object>
  ...
</objects>
```

Objetivos

En la sección del *XML* referente a los objetivos es donde esta el grueso de la información, que en cada archivo se guarda un nivel completo y cada uno de estos niveles está formado por una lista de objetivos. Los objetivos contienen información propia de este y además el jugador y tres listas; una de enemigos, otra de aliados y finalmente otra de cargueros. Cualquiera de las listas podría estar vacía si el tipo de objetivo lo requiere. En el listado 3.4 se describe la configuración de los objetivos.

- **index:** identificador del objetivo.
- **radius:** radio de activación del objetivo.
- **idTextObjetive:** identificador del texto del información para el objetivo.
- **idTextAmbientacion:** identificador del texto de ambientación para iniciar este objetivo.
- **idTextWin:** identificador del texto de ambientación para si se ha conseguido terminar el objetivo.
- **idTextLose:** identificador del texto de ambientación para si no se completa el objetivo.
- **type:** referencia al tipo de objetivo; 1 hunter, 2 destroy y 3 protected.
- **route:** lista de posiciones en el espacio a las que tiene que acercarse el jugador.
- **times:** lista de tiempos para acercarse a los puntos.
- **enemies:** lista de enemigos que hay en este nivel.
- **allies:** lista de aliados que hay en este nivel.
- **carriers:** lista de cargueros que hay en este nivel.
- **player:** la información del jugador.

Listing 3.4: Definición de los objetivos.

```
<objectives>
  <objective index="3" radius="737832.099986225" idTextObjetive="3"
      idTextAmbientacion="1" idTextWin="1" idTextLose="1" type="3">
    <route>
      <position>
        <x>4004.592041</x> <y>1984.837402</y> <z>2268.858398</z>
      </position>
    </route>
    <times>
      <time t="10"></time>
    </times>
    <enemies> ... </enemies>
    <allies> ... </allies>
```

```
    <carriers> ... </carriers>
    <player ... ></player>
  </objective>
  ...
</objectives>
```

En los siguientes puntos se explicará más detenidamente la configuración de los enemigos, aliados, cargueros y jugador.

Enemigos

Los enemigos están definidos dentro de una lista en el objetivo al que pertenecen. A continuación se da una explicación de los atributos que definen a los enemigos, en el listado 3.5 puede verse la configuración de un enemigo en el *XML*.

- **index:** identificador del enemigo.
- **mesh:** material que se utilizará en el juego para representar al enemigo.
- **speed:** velocidad máxima.
- **ratioAceleration:** ratio de aceleración.
- **ratioDeceleration:** ratio de deceleración.
- **life:** vida del enemigo.
- **shield:** escudo del enemigo.
- **timerRegenerationShield:** temporizador de regeneración del escudo.
- **defaultWeapon:** arma con la que empieza por defecto.
- **activationDistance:** distancia a partir de la cual se activa el enemigo.
- **type:** define el tipo de enemigo; 0 soldado y 1 jefe.
- **position:** posición del enemigo dentro de la escena *Ogre 3D*.
- **rotation:** rotación del modelo que representa al enemigo dentro de *Ogre 3D*.
- **scaled:** escalado del modelo.
- **weapons:** lista de identificadores de armas disponibles para el enemigo, estos identificadores tiene que corresponder con un arma disponible en este nivel.

Listing 3.5: Definición de los enemigos.

```
<enemies>
  <enemy index="1" mesh="FeisharShip.mesh" speed="10" ratioAceleration="0.4"
      ratioDeceleration="0.6" life="100" shield="200" timerRegenerationShield=
      "1.25" defaultWeapon="1" activationDistance="50" type="0">
    <position>
      <x>0.00000</x><y>0.00000</y><z>0.00000</z>
    </position>
    <rotation>
      <x>0.00000</x><y>0.00000</y><z>0.00000</z><w>1.00000</w>
    </rotation>
    <scaled>
      <x>1.00000</x><y>1.00000</y><z>1.00000</z>
    </scaled>
    <weapons>
      <weapon w="1"></weapon>
      <weapon w="2"></weapon>
    </weapons>
  </enemy>
</enemies>
```

Aliados

Al igual que los enemigos, los aliados también están definidos dentro de una lista en el objetivo al que pertenecen. A continuación se da una explicación de los atributos que definen a los aliados, en el listado 3.6 puede verse la estructura de configuración de un aliado en el *XML*.

- **index:** identificador del aliado.
- **mesh:** material que se utilizará en el juego para representar al aliado.
- **speed:** velocidad máxima.
- **ratioAceleration:** ratio de aceleración.
- **ratioDeceleration:** ratio de deceleración.
- **life:** vida del aliado.
- **shield:** escudo del aliado.
- **timerRegenerationShield:** temporizador de regeneración del escudo.
- **defaultWeapon:** arma con la que empieza por defecto.
- **activationDistance:** distancia del jugador a partir de la cual se activa el aliado.
- **position:** posición del aliado dentro de la escena *Ogre 3D*.
- **rotation:** rotación del modelo que representa al aliado dentro de *Ogre 3D*.

- **scaled:** escalado del modelo.
- **weapons:** lista de identificadores de armas disponibles para el aliado, estos identificadores tiene que corresponder con un arma disponible en este nivel.

Listing 3.6: Definición de los aliados.

```
<allies>
  <ally index="1" mesh="DarkFighter.mesh" speed="10" ratioAceleration="0.4"
      ratioDeceleration="0.6" life="100" shield="200" timerRegenerationShield=
      "1.25" defaultWeapon="1" activationDistance="50">
    <position>
      <x>0.00000</x><y>0.00000</y><z>0.00000</z>
    </position>
    <rotation>
      <x>0.00000</x><y>0.00000</y><z>0.00000</z><w>1.00000</w>
    </rotation>
    <scaled>
      <x>1.00000</x><y>1.00000</y><z>1.00000</z>
    </scaled>
    <weapons>
      <weapon w="1"></weapon>
      <weapon w="2"></weapon>
    </weapons>
  </ally>
  ...
</allies>
```

Cargueros

Los cargueros también están definidos como una lista dentro del objetivo, en el listado 3.7 se pueden ver sus atributos de configuración dentro del *XML*.

- **index:** identificador del carguero.
- **mesh:** material que se utilizará en el juego para representar al carguero.
- **speed:** velocidad máxima.
- **ratioAceleration:** ratio de aceleración.
- **ratioDeceleration:** ratio de deceleración.
- **life:** vida del carguero.
- **shield:** escudo del carguero.
- **timerRegenerationShield:** temporizador de regeneración del escudo.

- **defaultWeapon:** arma con la que empieza por defecto. Este atributo esta puesto para hacer la herencia de la clase padre con todas las naves, pero a efectos prácticos del juego no se utiliza porque los cargueros no pueden disparar.
- **activationDistance:** distancia a partir de la cual se activa el carguero.
- **position:** posición del carguero dentro de la escena *Ogre 3D*.
- **rotation:** rotación del modelo que representa al carguero dentro de *Ogre 3D*.
- **scaled:** escalado del modelo.
- **weapons:** al igual que el atributo defaultWeapon, no tiene efectos prácticos para el juego.
- **route:** ruta de puntos que tiene que seguir el carguero hasta que transporte su carga.

Listing 3.7: Definición de los cargueros.

```
<carriers>
  <carrier index="1" mesh="Object026.mesh" speed="10" ratioAceleration="0.4"
      ratioDeceleration="0.6" life="100" shield="200" timerRegenerationShield=
      "1.25" defaultWeapon="1" activationDistance="50">
    <position>
      <x>0.00000</x><y>0.00000</y><z>0.00000</z>
    </position>
    <rotation>
      <x>0.00000</x><y>0.00000</y><z>0.00000</z><w>1.00000</w>
    </rotation>
    <scaled>
      <x>1.00000</x><y>1.00000</y><z>1.00000</z>
    </scaled>
    <weapons>
      <weapon w="1"></weapon>
    </weapons>
    <route>
      <position>
        <x>-330.801544</x> <y>233.650879</y> <z>8.266037</z>
      </position>
    </route>
  </carrier>
</carriers>
```

Jugador

La configuración inicial del jugador también se define dentro del archivo *XML*, junto con la configuración del jugador también está la configuración de la cámara, ya que esta tiene que apuntar en la misma dirección que el modelo que representa al jugador.

- **index:** identificador del jugador.

- **mesh:** material que se utilizará en el juego para representar al jugador.
- **speed:** velocidad máxima.
- **ratioAceleration:** ratio de aceleración.
- **ratioDeceleration:** ratio de deceleración.
- **life:** vida del jugador.
- **shield:** escudo del jugador.
- **timerRegenerationShield:** temporizador de regeneración del escudo.
- **defaultWeapon:** arma con la que empieza por defecto.
- **activationDistance:** en este caso este atributo no es necesario para el jugador, se puso para la herencia que contiene todos los tipos de naves dentro del juego.
- **position:** posición del jugador dentro de la escena *Ogre 3D*.
- **rotation:** rotación del modelo que representa al jugador dentro de *Ogre 3D*.
- **scaled:** escalado del modelo.
- **weapons:** lista de identificadores de armas disponibles para el jugador.
- **camera:** configuración de la cámara, requiere: identificador, posición, rotación y lookat.

Listing 3.8: Definición del jugador.

```
<player index="1" mesh="DarkFighter.mesh" speed="4.5" ratioAceleration="0.4"
    ratioDeceleration="0.6" life="30000" shield="33334"
    timerRegenerationShield="1.5" defaultWeapon="2" activationDistance="43">
  <position>
    <x>1.000000</x> <y>1.000000</y> <z>0.000000</z>
  </position>
  <rotation>
    <x>0.781600</x> <y>0.481707</y> <z>0.212922</z> <w>0.334251</w>
  </rotation>
  <scaled>
    <x>1.000000</x> <y>1.000000</y> <z>0.000000</z>
  </scaled>
  <weapons>
    <weapon w="2"></weapon>
  </weapons>
  <camera index="1">
    <position>
      <x>1.700824</x> <y>8.251286</y> <z>-24.790890</z>
    </position>
    <rotation>
```

```
      <x>-0.007464</x> <y>0.084401</y> <z>0.050675</z> <w>0.995115</w>
    </rotation>
    <lookat>
      <x>3.873697</x> <y>6.867293</y> <z>-6.338510</z>
    </lookat>
  </camera>
</player>
```

3.4.2. Exportador de Blender a XML

Para la realización de la escena se ha recurrido al programa de modelado en 3D *Blender* y utilizando para ello un script en *Python* para proceder a exportar la escena en formato *XML*.

El *script* se basa principalmente en seleccionar los objetivos etiquetados con un nombre específico el cual tiene implícito en el nombre una serie de propiedades que posteriormente serán explicadas.

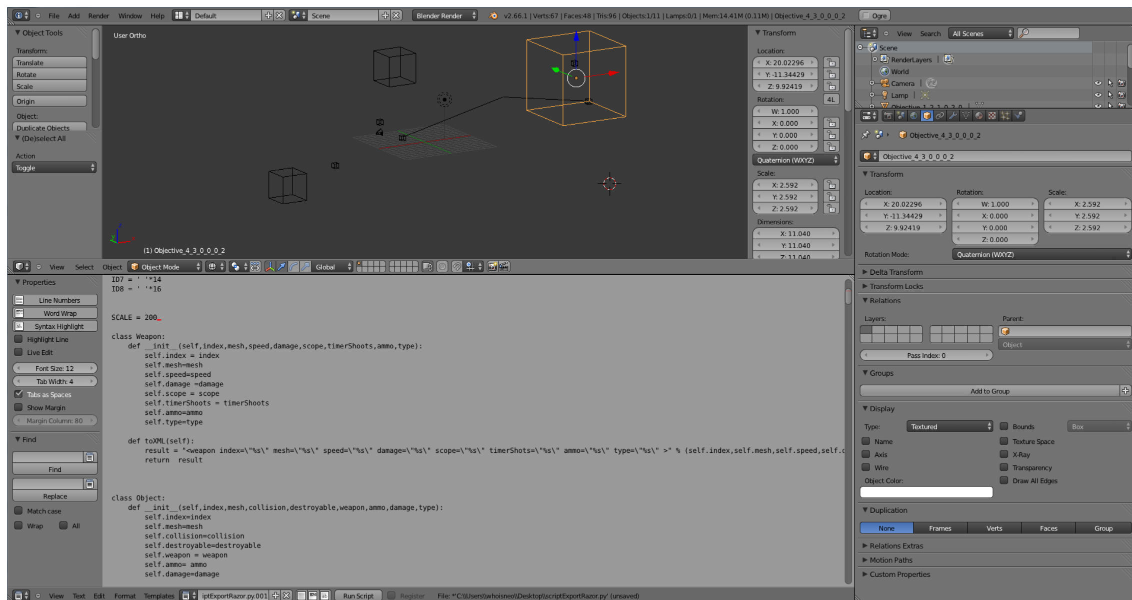


Figura 3.28: Captura del exportador en Blender.

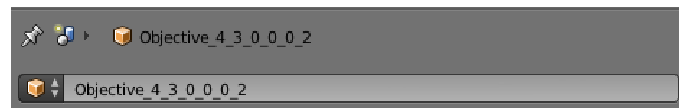
Existen 2 tipos básicos de nombres asignados a los objetos en *Blender* que son:

- **Objetos del escenario**, cuya estructura de nombre es Object_Tipo_NObjeto. Por ejemplo “Object_1_001”. Este nombre implica que es un objeto de escenario del tipo 1, y el NObjeto determina que puede haber más de un objeto de ese tipo. En la imagen 3.29 se muestra un ejemplo del nombre de los objetos.

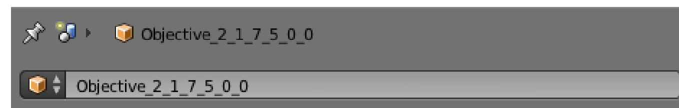
- **Objetivos de la misión**, cuya estructura de nombre de los objetivos es similar a la anterior, Objective_Indice_Tipo_NEnemigos_NAliados_NJefes_NCarrier. De esta forma con el índice se determina el número de prioridad dentro de los objetivos, el tipo determina el tipo de

Figura 3.29: Estructura del nombre del objeto.

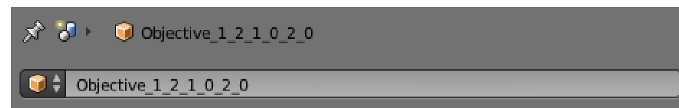
objetivo que puede ser hunter(1), destroy(2) y protected(3), el parámetro NEnemigos establece el número de enemigos que queremos que aparezcan en este objetivo, en NAliados establecemos el número de aliados que deseamos que aparezcan y que nos ayuden, en NJefes se establece el número de enemigos señalados como prioritarios para su eliminación, y por último en NCarrier se establecen el número de cargueros. En las capturas de la figura 3.30 se puede ver un ejemplo de cada uno de estos tipos de objetivos.



(a) Hunter.



(b) Destroy.



(c) Protected.

Figura 3.30: Estructura del nombre de los tres tipos de objetivo.

Dado que las escalas entre el juego y *Blender* varían mucho se ha establecido en el *script* una serie de parámetros que se encarga de regular el tipo de escala permitiendo crear un escenario en *Blender* y poder modificarlo para ajustarlo a la escala del juego.

3.4.3. Importador de XML a Ogre 3D

Los niveles del juego están almacenados en archivos *XML*, para procesarlos y cargarlos dentro de la escena generada en *Ogre 3D*, es necesario implementar un mecanismo que realice este proceso. Para realizar este importador se utilizó la biblioteca *Xerces-C++*, la cual proporciona una serie de funciones que facilitan este proceso. Este proceso consiste obtener la referencia a la raíz del documento *XML*, y a partir de este ir iterando sobre sus atributos y etiquetas hasta recuperar todo el contenido.

Cuando se ha recuperado todo el contenido de del archivo *XML* se tienen que generar los objetos que forman la escena en *Ogre 3D* a partir de esta información. Para ello se ha implementado el patrón *Factory Method*, de esta manera se resuelve el problema de crear todos los

objetos que conforman la escena de forma automática y abstrayéndose de como se crean.

En la figura 3.31 se muestra el diagrama *UML* de la utilización de este patrón para realizar la generación de la escena, en el caso de este proyecto la clase "factoría" que se encarga de generar los objetos es la clase *SceneLevel*.

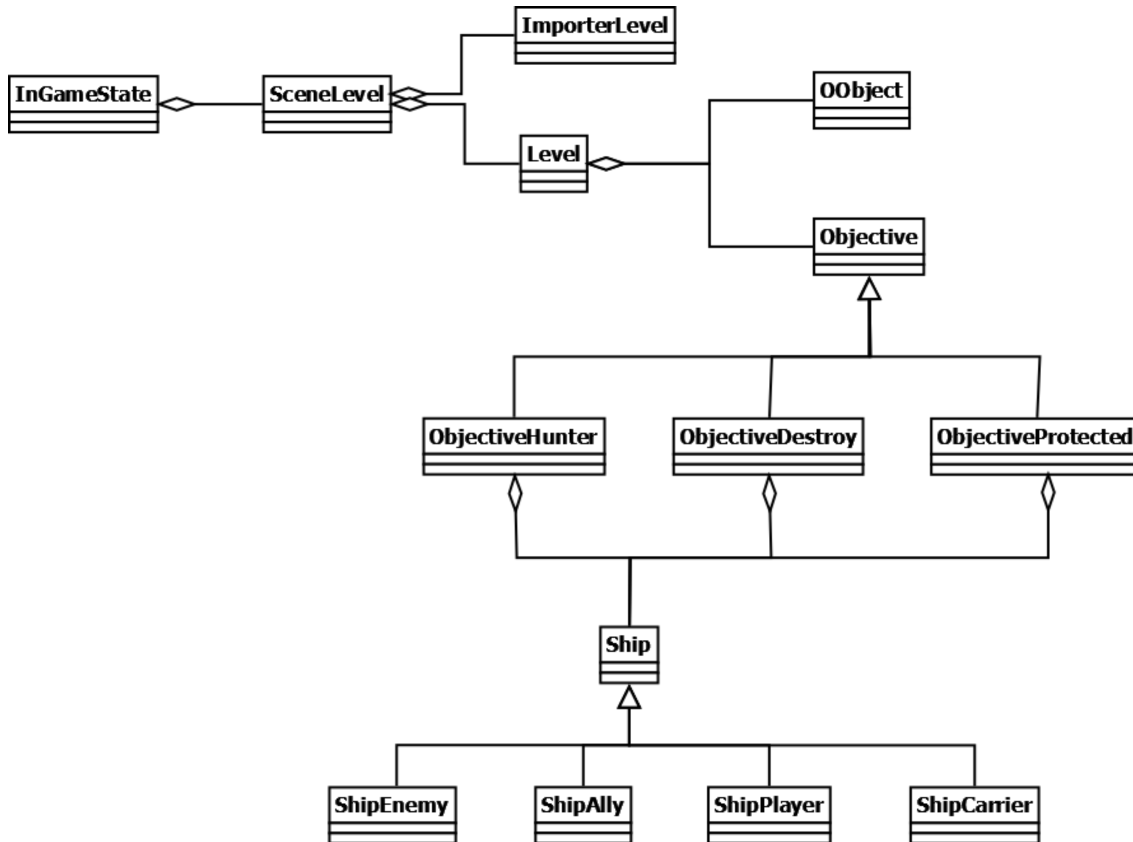


Figura 3.31: Diagrama *UML* del patrón *Factory Method* en nuestro juego.

3.5. Sistema de navegación del jugador

El sistema de navegación del jugador se basa en la rotación de la cámara a través del eje longitudinal, transversal y vertical de esta. Para este fin, se han utilizado cuaternios para determinar el vector de rotación y el ángulo a rotar, y aplicando esta serie de transformaciones a la orientación del sistema de cámara.

El jugador puede avanzar a través del espacio aumentando o disminuyendo el ratio de aceleración-deceleración que afecta al vector de desplazamiento aumentando o disminuyendo el tamaño de este, haciendo por lo tanto que la nave vaya más deprisa o más despacio.

El jugador puede orientarse en el espacio por medio de un sistema de brújula que será posteriormente explicado, el cual se orienta al enemigo más cercano alineando con este, posibilitando la orientación hacia los enemigos.

3.6. Sistema de disparo

Los disparos tanto del jugador como de los aliados y enemigos se basan en la creación y desplazamiento de una serie de *billboards*. En el caso particular del jugador, el *billboard* se crea a un cierto número de unidades debajo de la posición de la cámara, y se utiliza la dirección del vector de la cámara en el instante del disparo para desplazar el disparo a través del espacio. Una vez que el jugador presiona la tecla **T**, se calcula desde donde tiene que salir el *billboard* y la orientación de este respecto a la cámara.

En el caso de los enemigos y de los aliados se utiliza la misma idea que en el caso del jugador, pero en cambio en vez de la dirección de la cámara se utiliza el vector de dirección de la nave proporcionado por *OpenSteer*, y la posición actual para realizar la creación y desplazamiento de los disparos.

Respecto a los misiles, se utiliza un *billboard* orientado a cámara para el misil en cuestión y un *ribbonTrails* para la estela que va dejando atrás en su desplazamiento por el espacio.

El *ribbonTrails* es un efecto de estela implementado en *Ogre 3D* mediante una serie de *billboard* orientados a cámara que establecen una jerarquía como si fuesen eslabones de una cadena. De esta forma el elemento n de la cadena sigue al elemento n-1 y es seguido por el elemento n+1. Este tipo de comportamiento se sigue a lo largo de todos los eslabones de la cadena que se desplazan mediante una interpolación de sus posiciones. El efecto *ribbonTrails* se puede configurar una serie de atributos como por ejemplo el número de elementos que conforman la cadena, así como la longitud final de la estela, además de una serie de parámetros adicionales como material utilizado, color, etc.

3.7. Sistema de representación en pantalla (HUD)

El sistema *HUD (Heads-Up Display)* es un método de representación de información en forma gráfica mediante iconos o información textural de tal manera que al usuario le sea muy rápido obtener la información necesaria para el desarrollo de la actividad en cuestión.

En nuestro caso particular se ha utilizado el sistema *HUD* para la representación de información relevante como por ejemplo:

- Localización de los enemigos.
- Localización de las naves insignia a eliminar.
- Localización de los aliados.
- Nivel de escudo y vida de los enemigos.
- Distancia entre el jugador y las naves enemigas y aliadas.
- Orientación espacial al enemigo más próximo.
- Capacidad operativa del armamento del jugador.

- Representación de la vida y el escudo del jugador.

3.7.1. Sistema HUD

La idea principal en la cual se basa el *HUD* desarrollado para la representación de información en pantalla es la intersección entre rectas y planos. Inicialmente se crea un plano posicionado un cierto número de unidades delante de la cámara con el mismo vector normal que la dirección de la cámara, a continuación se recorren la lista de los objetos susceptibles de representación como son las naves enemigas y aliadas. Desde estas se crea una recta o en el caso particular de *Ogre 3D*, un rayo desde la posición del objeto a representar hacia la cámara, de tal forma que interseccione con el plano puesto delante de la cámara en un punto. En el punto obtenido, se posicionan los iconos o información que queramos representar, en nuestro caso particular posicionamos en ese punto los *billboard* del tipo de objetivo (aliado o enemigo), la información del escudo y la vida, distancia, . . . Dado que el *frustum* cercano de cámara puede eliminar los iconos representados en pantalla, se ha retrasado la distancia de recorte del *frustum* cercano. Se ha de tener en cuenta la eliminación de los iconos de representados en el *HUD* cuando el jugador no tiene visión directa con ellos.

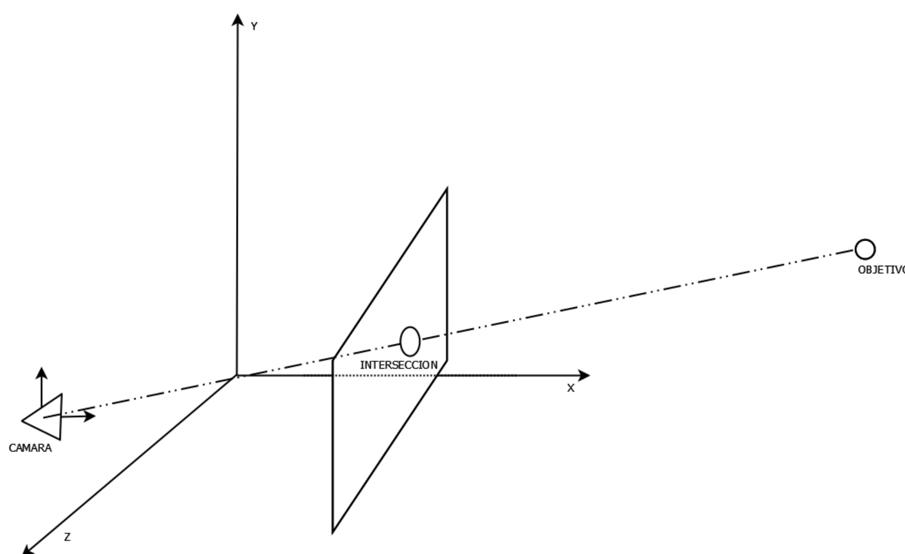


Figura 3.32: Sistema de representación del HUD.

3.7.2. Predicción del futuro posicionamiento enemigo

Para llevar a cabo esta tarea se utiliza el vector de dirección de la enemiga, así como su velocidad, la posición del jugador y la velocidad del disparo del jugador.

La idea principal se basa en predecir la futura posición enemiga para que nuestra ráfaga láser pueda incidir sobre las naves enemigas a una distancia determinada. Para ello se determina el tiempo teórico que tarda en recorrer una ráfaga láser desde la posición del jugador hasta la posición actual del enemigo. Determinado este tiempo se calcula según la dirección y velocidad

del enemigo cual será su posición futura. A partir de esta posición futura se proyecta sobre el *HUD* y se representa la retícula de predicción de disparo que nos ayudara a poder apuntar y hacer blanco en distancias largas.

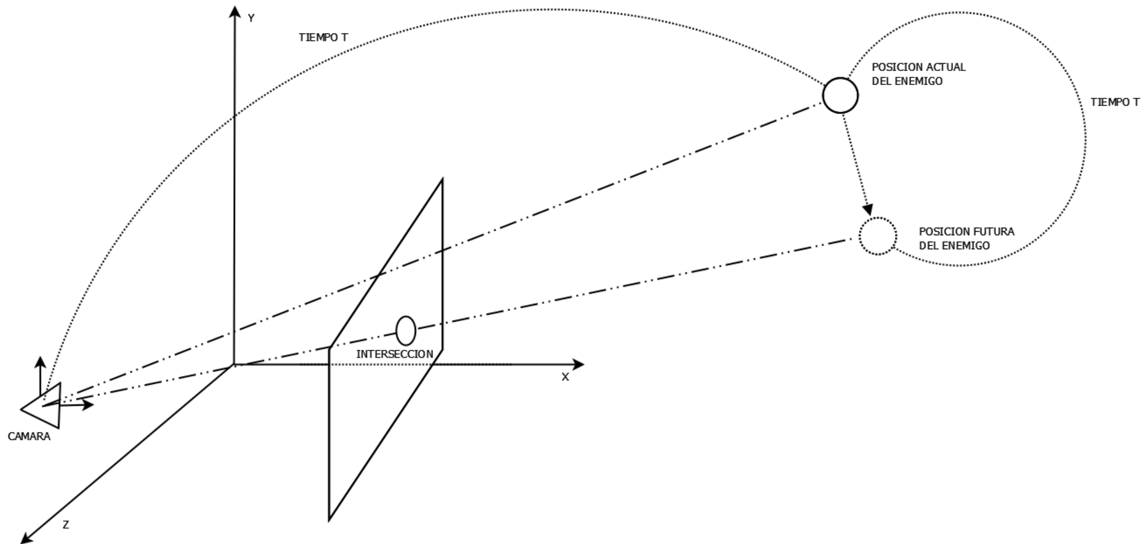


Figura 3.33: Proyección en el HUD de la posición futura del enemigo.

3.7.3. Orientación espacial hacia el enemigo más cercano

Este sistema se basa en encontrar el enemigo más próximo de nuestra posición actual, de tal forma que nos oriente hasta estar completamente alineados con nuestro enemigo y se pueda proceder a darle a caza. La forma de proceder es mediante un algoritmo voraz con complejidad lineal que determine cuál es la menor distancia. A partir de la posición del enemigo se genera un vector desde la posición del enemigo hasta la cámara, y teniendo en cuenta la orientación espacial de la cámara, se genera un cuaternio de rotación necesario para alinear la brújula de representación hacia el objetivo en cuestión.

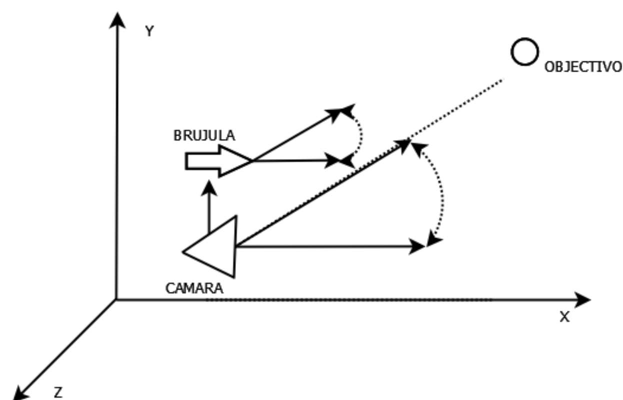


Figura 3.34: Orientación de la brújula hacia el enemigo mas cercano.

3.8. Sistema de lógica del juego

La lógica de juego se basa en la realización de una serie de objetivos. Estos objetivos están dispuesto de forma secuencial de tal forma que conforme avancemos en el juego y completando los objetivos se nos irán desbloqueando nuevos objetivos para completar la misión.

Por lo tanto, un nivel del juego esta formado una serie de objetivos. Estos objetivos pueden ser de los siguientes tipos:

- **Objetivo de tipo *Hunter:*** este tipo de objetivo se basa en eliminar todos los enemigos localizados en una zona determinada. Para ello se han utilizado una serie de contenedores que contienen los enemigos activos en la pantalla. Al ir eliminando los enemigos estos serán borrados del contenedor y cuando hallamos eliminado a todos los objetivos del contenedor habremos completado el objetivo en cuestión.
- **Objetivo de tipo *Destroy:*** este tipo de objetivo se basa en la eliminación de un cierto tipo de enemigos que será reflejado en el *HUD* principal Este tipo de enemigos señalados para su eliminación se almacenan en un contenedor y conforme se vayan eliminando serán borrados del contenedor. Si hemos eliminado todos los enemigos señalados para su eliminación del contenedor se considera que se ha completado el objetivo.
- **Objetivo de tipo *Protected:*** se basa en la protección de en un cierto grupo de aliados de un tipo espacial denominado carguero. Este tipo de aliado recorren una serie de rutas predeterminadas hasta alcanzar su último punto de su ruta sin que sean eliminadas por parte de las unidades enemigas. Estas naves son almacenadas en un contenedor y las unidades enemigas intentaran su eliminación, si al llegar al último punto de la ruta siguen estando vivas, es decir siguen estando dentro del contenedor se considera que se ha cumplido el objetivo.

Existen una serie de clases de tipo *manager* que son las encargadas de ir iterando sobre la estructura de objetivos y cuando se halla finalizado un objetivo pasar al siguiente objetivo si se cumplen todas las condiciones hasta llegar al final del juego.

3.8.1. Lógica de los objetivos

Como se ha comentando anteriormente, se va a disponer de forma más detallada la lógica particular de cada uno de los objetivos a la hora de determinar si se ha completado o no un objetivo concreto.

- **Objetivo de tipo *Hunter:*** se comprueba inicialmente si el jugador esta vivo o muerto, en el que caso de que estuviese muerto se determinaría que es final de juego en caso contrario es decir que siga vivo se comprueba que el contenedor donde están almacenados los enemigos sea igual a cero.
- **Objetivo de tipo *Destroy:*** se comprueba que el usuario siga vivo o muerto, en el caso que este muerto se determina final de juego, en caso contrario se comprueba la longitud del contenedor y si es igual a cero se determinará que se ha completado el objetivo.

- **Objetivo de tipo *Protected:*** en este tipo de objetivo se comprueba que el jugador este vivo o muerto, en el caso que este muerto se determina que sea fin del juego, en caso contrario se comprueba que el contenedor de cargueros sea distinto de cero y que los cargueros no hallan llegado al final de sus rutas. Si se satisface estas condiciones se determina que se ha completado el objetivo, en caso contrario se determina que no se ha completado el objetivo.

3.9. Sistema de sonido

En la actualidad la música en los videojuegos es muy importante, tanto como el diseño y modelo de los protagonistas de estos. El sistema de sonido es una parte importante para conseguir que el usuario disfrute y adquiera una experiencia óptima cuando juegue a cualquier videojuego. Por este motivo se ha puesto música durante todo el juego, de esta forma se atrae la atención del usuario. Además, se han introducido efectos de sonido en los menús del juego para aumentar la retroalimentación con el usuario y durante la ejecución de la partida para avisar al usuario de que le están disparando o que se ha destruido alguna nave en el escenario.

Para incluir la música y los efectos de sonido del juego se han utilizado las clases *TrackManager* y *SoundFXManager*, proporcionadas durante el Curso de Experto en Desarrollo de Videojuegos. Se pretendía simplificar y facilitara el acceso al sistema de sonido, para ello se decidió hacer una nueva interfaz que contuviera los elementos necesarios de las interfaces de las dos clases originales. Este proceso se realizó siguiendo los patrones *Adapter* y *Singleton*, en la figura 3.35 el diagrama *UML* de como quedó el diseño.

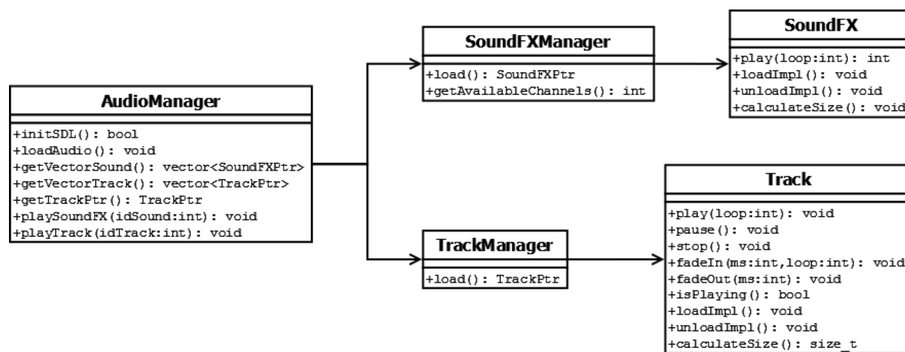


Figura 3.35: Diagrama *UML* del sistema de sonido.

4. Manual de usuario

Al iniciar el juego se nos presenta un menú principal 4.1(a) con las siguientes opciones:

- **Play**: Si deseamos jugar 4.2(a).
- **Help**: Para acceder al menú de ayuda 4.1(b).
- **Score**: Para acceder al menú de puntuación 4.1(c).
- **Credits**: Para acceder al menú de los créditos 4.1(d).
- **Exit**: Para salir del juego.

(a) Menú principal.

(b) Menú de ayuda.

(c) Menú de puntuación.

(d) Menú de créditos.

Figura 4.1: Menús del juego Razor Squadron.

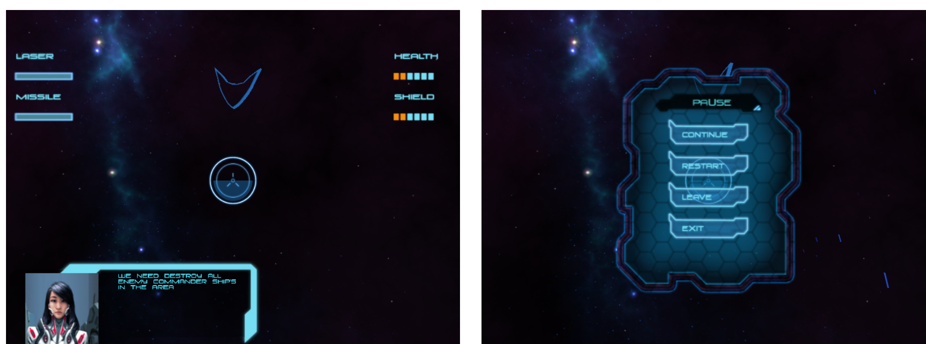
Si pulsamos sobre la opción Play, pasaremos a seleccionar el nivel de dificultad 4.2(a) y posteriormente tendremos a nuestra disposición una serie de naves para elegir así como el color de estas 4.2(b). Una vez seleccionada la nave entraremos en el juego 4.3(a).

(a) Menú de selección de dificultad. (b) Menú de selección de nave.

Figura 4.2: Menús del juego Razor Squadron (II).

Durante el juego podemos acceder al menú de pausa 4.3(b) que nos habilitará las siguientes opciones.

- **Continue**: Para continuar la partida.
- **Restart**: Para reiniciar la partida.
- **Leave**: Para abandonar la partida.
- **Exit**: Para salir del juego.



(a) Menú de juego. (b) Menú de pausa.

Figura 4.3: Menús del juego Razor Squadron (III).

Durante el transcurso del juego, el usuario dispone de una serie de teclas para poder navegar con su nave. A continuación se detalla el mapeo de teclas asignadas al usuario y sus acciones:

- **Tecla A**: Aceleración de la nave

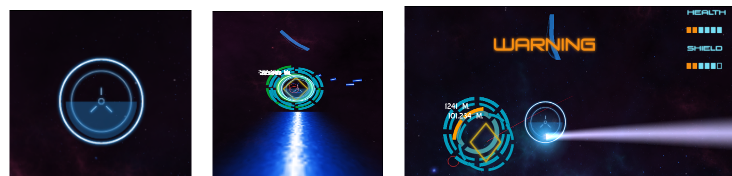
- **Tecla Z**: Deceleración de la nave
- **Tecla Q**: Movimiento de Tonel rápido hacia de derecha.
- **Tecla W**: Movimiento de Tonel rápido hacia de izquierda.
- **Tecla direccional Down**: Subir el morro de la nave.
- **Tecla direccional Up**: Bajar el morro de la nave.
- **Tecla direccional Left**: Girar hacia la izquierda la nave.
- **Tecla direccional Right**: Girar hacia la derecha la nave.
- **Tecla R**: Disparo del misil de seguimiento.
- **Tecla T**: Disparo de ráfaga láser.

El usuario dispone de unos indicadores en la parte izquierda 4.4(a) de la pantalla que indican el nivel de sobrecarga del láser, y el nivel de recarga de los misiles una vez disparados.

(a) Indicadores de las armas. (b) Indicadores de la salud.

Figura 4.4: Interfaz del juego ingame.

En la parte derecha 4.4(b) de la pantalla se encuentra el escudo y el nivel de vida del usuario, que irá descendiendo gradualmente el escudo conforme vayamos recibiendo daño, y una vez agotado el escudo iremos recibiendo daño en la vida del usuario.



(a) Mirilla. (b) Disparo láser. (c) Disparo misil.

Figura 4.5: Mirilla y disparos de las armas.

En la parte central 4.5(a) de la pantalla se encuentra la mirilla del usuario la cual es una ayuda para poder apuntar y disparar a los enemigos con las ráfagas láser 4.5(b).

Los misiles de seguimiento 4.5(c) siempre se son guiados hacia el enemigos más cercado y una vez que estén disponibles no es necesario apuntar ya que siguen automáticamente al enemigo.

El *HUD* también nos proporciona información relativa a la futura posición del enemigo lo que posibilita poder infringirle daño en distancia largas (figura 4.6). A partir de 150 unidades se activa la retícula de predicción de disparo y se representa por medio de un circulo de color rojo.

Figura 4.6: Retícula de predicción.

Los enemigos son posicionados en el *HUD* y presentan una serie de datos indicativos relativos a:

- **Nivel de distancia respecto al usuario.**
- **Nivel de escudo.** El escudo del enemigo se ubica en el anillo exterior. Cuando escudo está integro presenta un color azul 4.7(a), conforme se vaya bajando la vida presentará un color verde que se irá transformando en amarillo y naranja a medida que vaya perdiendo energía 4.7(b).

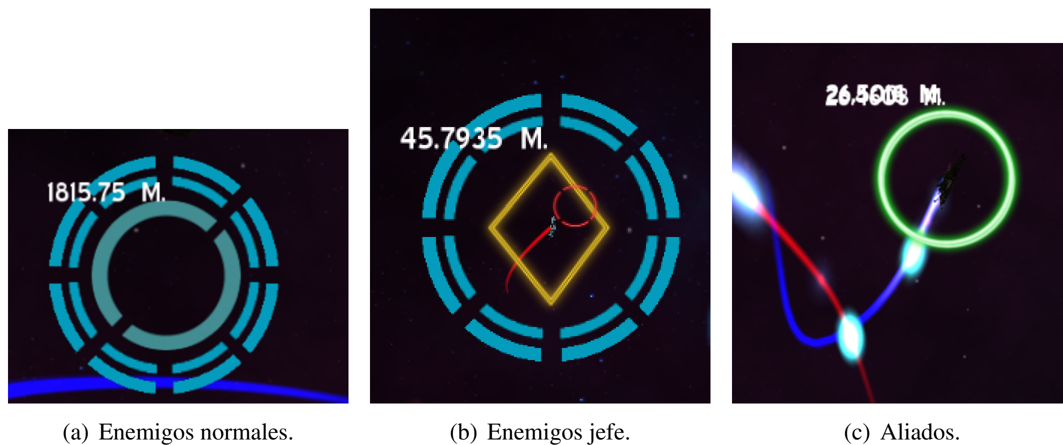
(a) Salud y escudo total. (b) Escudo dañado. (c) Vida parcial.

Figura 4.7: Porcentajes de salud y escudo.

- **Nivel de vida.** Al igual que el escudo del enemigo, la representación de la vida del enemigo se representan en el anillo interior, y presenta la misma configuración de colores que el anillo exterior del escudo 4.7(c).

La forma de representar a los enemigos en el *HUD* es mediante un icono representativo que marca al enemigo 4.8(a).

Existen una serie de enemigos que han sido marcados como prioritarios para su eliminación. Su representación en el *HUD* es la siguiente 4.8(b).



(a) Enemigos normales. (b) Enemigos jefe. (c) Aliados.

Figura 4.8: Iconos de las naves NPG.

Los aliados se proyectan en el *HUD* y presentan información asociada a su distancia relativa al usuario 4.8(c).

En la parte inferior izquierda 4.9 de la pantalla en el comienzo de cada nuevo objetivo se nos presenta información de lo que tenemos que hacer para poder completar el objetivo.

Figura 4.9: Menú de información de objetivo.

Una vez que hallamos completado las misiones nos saldrá un menú para poder poner el nombre y establecer un ranking con la puntuación obtenida 4.10.

Figura 4.10: Menú de fin de juego.

5. Conclusiones y trabajos futuros

5.1. Conclusiones 49
5.2. Trabajos futuros 49

5.1. Conclusiones

El desarrollo de este juego ha sido una experiencia positiva y pensamos que el resultado obtenido ha sido satisfactorio, debido a la falta de tiempo hay una serie de aspectos que nos hubiese gustado integrar en la versión definitiva del juego.

Hemos sufrido cierta inexperiencia en el desarrollo de la IA y su integración en el juego, esto ha comprometido el resto de partes del juego, gastando demasiado tiempo en esta tarea. Las representaciones de los mundos en *OpenSteer* y en *Ogre 3D* son significativamente distintas, debido a las velocidades de las naves y se tuvo que redimensionar y ajustar las escalas de todo el diseño.

La sensación final tras haber acabado el videojuego es que nos encontramos más preparados para realizar este tipo de desarrollos y que corrigiendo aquellos defectos que nos han surgido y que hemos ido aprendiendo a evitar a lo largo del curso podemos mejorar el nivel en posibles proyectos de futuro.

5.2. Trabajos futuros

En esta sección se listan las posibles mejoras que se pueden llevar a cabo en un futuro, algunas de estas tareas no se han podido llevar a cabo por falta de tiempo.

1. **Temporizadores:** poner temporizadores para terminar los objetivos y evitar que el usuario este dando vueltas alrededor del espacio sin completar a los objetivos.

2. **Mayor variedad de armas y obstáculos dentro del escenario:** este punto estaba planificado para realizarlo, el archivo XML de almacenamiento de los niveles está diseñado y preparado para almacenar armas u obstáculos con diferentes configuraciones.

3. **Selección de nave:** utilizar diferentes modelos de naves y que el usuario pueda elegirlos.

4. **Multijugador:** la idea inicial del juego era la inclusión el modo multijugador cooperativo y competitivo, pero después de valorar el tiempo del que se disponía para realizar este videojuego se desestimó la idea.

5. **Argumento:** desarrollo de un hilo argumental para darle más vida al juego y que no sea sólo destruir enemigos.

A. Imágenes del juego

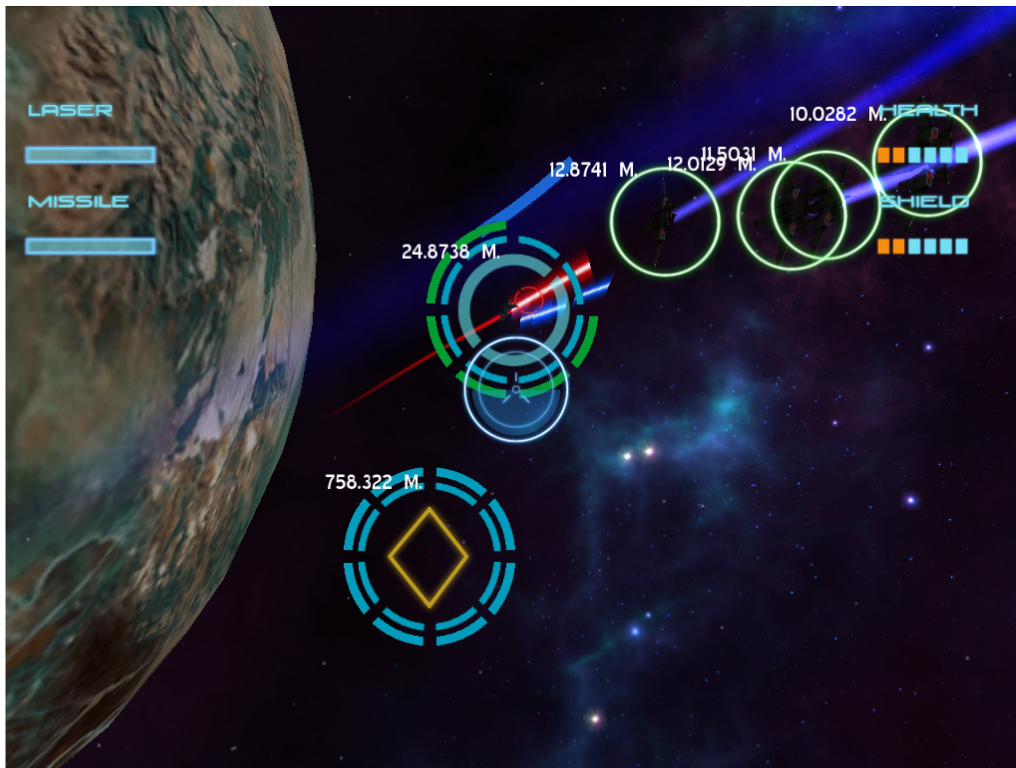


Figura A.1: Imagen del juego I.

Figura A.2: Imagen del juego II.

Figura A.3: Imagen del juego III.

Figura A.4: Imagen del juego IV.

Figura A.5: Imagen del juego V.

Figura A.6: Imagen del juego VI.

Figura A.7: Imagen del juego VII.

Bibliografía

[1] Ogre 3D. Ogre 3D. http://www.ogre3d.org.

[2] Alexander Feder. BibTEX. http://www.bibtex.org.

[3] Bernardo Cascales Salinas, Pascula Lucas Saorin, José Manuel Mira Ros, Antonio Pallarés Ruiz, and Salvador Sánchez-Pedreño Guillén. *LATEX: una imprenta en sus manos*. ADI, 2000.

[4] David Vallejo Fernández, Carlos González Morcillo, David Villa Alises, Francisco Jurado Monroy, and Francisco Moya Fernández et al. Desarrollo de videojuegos: Un enfoque práctico. http://www.cedv.es/descargas/cedv_3Ed.pdf, July 2014.

[5] Ltd Evolus Co. Pencil Project. http://pencil.evolus.vn/.

[6] Blender Foundation. Blender. http://www.blender.org.

[7] Eclipse Foundation. Eclipse. http://www.eclipse.org.

[8] The Apache Software Foundation. Xerces-C++ XML Parser. http://xerces.apache.org/xerces-c.

[9] Wrecked Games. OIS, Object-oriented Input System Library. http://sourceforge.net/projects/wgois.

[10] Sam Lantinga. SDL, Simple DirectMedia Layer. http://www.libsdl.org.

[11] William Jon McCann. Mercurial: The Definitive Guide.

[12] Bryan O'Sullivan. Dia. https://wiki.gnome.org/Apps/Dia.

[13] Colin Peters. MinGW, Minimalist GNU for Windows. http://www.mingw.org/.

[14] Alex C. Peterson. Spacescape. http://alexcpeterson.com/spacescape.

[15] Craig Reynolds. OpenSteer: Steering Behaviors for Autonomous Characters. http://opensteer.sourceforge.net.

[16] Richard M. Stallman, Roland McGrath, and Paul D. Smith. GNU Make. July 2010. http://www.gnu.org/software/make/manual/make.pdf.

[17] Richard M. Stallman et al. GNU Emacs Manual. 2012. http://www.gnu.org/software/emacs/manual/emacs_7x9.pdf.

[18] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with GDB. 2010. https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/Debugger.pdf.

[19] Roussel-Geoffrey. Ogre Particle Lab 0.999. http://roussel-geoffrey.blogspot.com.es/2011/09/ogre-particle-lab-0999.html.

[20] GIMP team. GIMP. http://www.gimp.org.

[21] Graphviz team. Graphviz - Graph Visualization Software. http://www.graphviz.org.

[22] The Free Software Foundation. Using gcc. https://gcc.gnu.org/onlinedocs/gcc-4.9.0/gcc.pdf.

[23] Yosoygames. Ogre Meshy. http://yosoygames.com.ar/wp/ogre-meshy.