Universidad
Carlos III de Madrid

Departamento de Informática

PROYECTO FIN DE CARRERA

Desarrollo del juego Sky Fighter mediante XNA 3.1 para PC

Autor: Íñigo Goicolea Martínez

Tutor: Juan Peralta Donate

Leganés, abril de 2011

Agradecimientos

Este proyecto es la culminación de muchos meses de trabajo, y de una carrera a la que llevo dedicando más de cinco años. En estas líneas me gustaría recordar y agradecer a todas las personas que me han permitido llegar hasta aquí.

En primer lugar a mis padres, Antonio y Lola, por el apoyo que me han dado siempre. Por creer en mí y confiar en que siempre voy a ser capaz de salir adelante y no dudar jamás de su hijo. Y lo mismo puedo decir de mis dos hermanos, Antonio y Manuel.

A Juan Peralta, mi tutor, por darme la oportunidad de realizar este proyecto que me ha permitido acercarme más al mundo de los videojuegos, algo en lo que querría trabajar. Pese a que él también estaba ocupado con su tesis doctoral, siempre ha sacado tiempo para resolver dudas y aportar sugerencias.

A Sergio, Antonio, Toño, Alberto, Dani, Jorge, Álvaro, Fernando, Marta, Carlos, otro Antonio y Javier. Todos los compañeros, y amigos, que he hecho y que he tenido a lo largo de la carrera y gracias a los cuales he podido llegar hasta aquí.

Por último, y no menos importante, a los demás familiares y amigos con los que paso mucho tiempo de mi vida, porque siempre están ahí cuando hacen falta.

Alumno: Íñigo Goicolea Martínez
Tutor: Juan Peralta Donate

Resumen

La industria de los videojuegos ha sufrido un crecimiento exponencial durante los últimos años. Con la llegada de la última generación de videoconsolas y, con ellas, los sistemas de distribución digital, la creación y distribución de videojuegos vuelve a estar al alcance de todos. Con una pequeña inversión es posible hacer llegar a muchos países un videojuego desarrollado por un pequeño grupo de personas.

Entre la cantidad de géneros que existen en el mundo de los videojuegos, hay uno que siempre ha destacado y ha estado presente a lo largo de toda la historia: los *shooters*. Se trata de juegos de mecánica sencilla, pero dificultad elevada que llevan décadas consiguiendo atraer la atención de los jugadores a lo largo y ancho del mundo.

El objetivo de este proyecto es desarrollar un *shooter* en 2D de *scroll* vertical, más concretamente del subgénero de los matamarcianos, destacando como punto principal la inclusión de un editor de niveles que alargue indefinidamente la vida del juego. El desarrollo se hará utilizando XNA, el API de desarrollo de videojuegos para XBOX 360, PC y Zune, distribuido por Microsoft.

A lo largo del desarrollo será necesario trabajar con aspectos importantes de un videojuego como es el control del mismo mediante teclado o ratón, el dibujo y animación de *sprites*, la detección de colisiones, la inclusión de música y efectos de sonido, o la construcción de un *scroll* de fondo utilizando un *tileset*.

Índice

Índice de tablas

Índice de figuras

Índice de código

CAPÍTULO 1: INTRODUCCIÓN

Este capítulo presenta los propósitos de este proyecto.

En primer lugar se muestra la motivación del proyecto, a continuación los objetivos que se plantean y, para terminar, una visión general de los distintos apartados que contiene este documento.

1.1 Motivación del proyecto

La industria de los videojuegos es el sector encargado de diseñar, desarrollar, distribuir y vender videojuegos. Abarca muchas disciplinas que van desde el dibujo a la programación, emplea a miles de personas a lo largo del mundo, y en la última década ha sufrido un crecimiento enorme.

Según los datos recogidos en una noticia del ABC [1], en el año 2009 los videojuegos se acercan cada vez más a la facturación de, nada más y nada menos, que la industria del cine de Hollywood.

Con una media de 60-70 dólares por juego frente a 10-20 por entrada y DVD/Blu-ray, los videojuegos hicieron una caja de 41.200 millones de euros (55.000 millones de dólares) en ingresos únicamente en *software* durante 2009. Además, se ingresaron 16.400 millones de euros (22.000 millones de dólares) en *hardware*. Estos 57.600 millones de euros (77.000 millones de dólares) se acercan cada vez más a la caja de Hollywood, que en 2009 estuvo en los 63.600 millones de euros (85.000 millones de dólares) en todo el mundo. Sin duda, la industria de los videojuegos está en alza.

Sin embargo, con las nuevas tecnologías, los costes de desarrollo de videojuegos se disparaban. Grand Theft Auto IV [2] se convirtió en uno de los juegos más caros de la historia, con un presupuesto de 100 millones de dólares. Aunque no todos los juegos requieren esa cantidad, sí es cierto que desarrollar videojuegos no estaba al alcance de todo el mundo hasta la última generación de consolas.

Gracias a sistemas de distribución digital como Steam [3] en PC, WiiWare [4] en Wii, Playstation Network [5] en Playstation 3 y PSP o Xbox Live [6] en XBOX 360; pequeñas empresas o desarrolladores independientes pueden lanzar sus juegos al mercado sin más coste que el de la suscripción al servicio.

Además, hoy en día hay muchos APIs y plataformas de desarrollo gratuitas disponibles para toda persona que quiera iniciar sus andaduras en este mundillo. Para este proyecto se pretende utilizar el API de Microsoft XNA Game Studio [7], que permite desarrollar videojuegos tanto para PC como para XBOX 360.

1.2 Objetivos del proyecto

La idea inicial del proyecto consistía en diseñar e implementar un videojuego para ordenador mediante la herramienta Microsoft XNA Game Studio, con un nivel de jugabilidad respetable.

Dada esta premisa, y después de debatir acerca del videojuego que se iba a desarrollar, los objetivos del proyecto son los siguientes:

- Desarrollar un *shooter* en 2D de *scroll* vertical con una nave como elemento controlable por el jugador, utilizando la herramienta Microsoft Visual Studio [8] y el API de desarrollo de videojuegos Microsoft XNA Game Studio.
- Incluir un editor de niveles que permita que la experiencia de juego no termine al finalizar el juego, sino que permita a los usuarios crear y configurar todos los elementos que conforman un nivel. Pudiendo jugar en dichos niveles posteriormente, de forma que el juego tenga una duración ilimitada.

Para ello, se deberán incluir en el videojuego todos los elementos posibles para hacer de él un juego entretenido y, sobre todo, adictivo, que no dé sensación de repetición o llegue a cansar pronto a los usuarios.

Además de los objetivos mencionados, también existen una serie de objetivos personales que se pretenden alcanzar realizando este proyecto:

- Participar en el desarrollo de un videojuego desde la concepción de la idea hasta el producto final, aprendiendo todos y cada uno de los pasos necesarios para completar dicho desarrollo.
- Realizar un primer acercamiento y profundizar en el trabajo con un API que permite desarrollar aplicaciones tanto para ordenador como para la consola XBOX 360, lo que permitiría distribuir el videojuego mediante sistemas de distribución digital tales como Steam en ordenador, o Xbox Live Indie Games en la XBOX 360, con el objetivo de obtener beneficios.

1.3 Visión general del documento

Este documento se organiza de la siguiente manera:

- En el primer apartado, Introducción, se aclara la motivación que ha hecho surgir el proyecto, así como los objetivos que se plantean.
- El segundo apartado, Estado de la cuestión, incluye una introducción a los videojuegos que explica cómo nació la industria. Posteriormente se centrará en la historia de los *shooters* en 2D, y terminará hablando de las herramientas disponibles para desarrollar videojuegos, profundizando en la alternativa elegida: XNA.
- El tercer apartado, Análisis, comienza con la definición del alcance del sistema. A continuación se incluirá la especificación de requisitos, tanto de usuario como

de software, y los casos de uso con sus correspondientes diagramas de secuencia. Finalmente, se adjunta el diagrama de actividad del sistema.

- El apartado cuatro, Diseño, describe el diagrama de las clases que se implementarán. Para cada una de las clases, se comentan sus atributos y métodos más importantes dentro del desarrollo del proyecto.
- En el apartado cinco, Implementación, se explican los aspectos más importantes del desarrollo, tales como el control mediante teclado o ratón, la detección de colisiones, etc. y como se han llevado a cabo.
- En el sexto apartado, Conclusiones, se recogen las conclusiones a las que se ha llegado tras el desarrollo del proyecto, así como un análisis para comprobar si se han cumplido los objetivos marcados.
- En el séptimo apartado, Líneas futuras, contiene las posibles mejoras que se podrían aplicar al videojuego en futuros desarrollos, tomando este proyecto como base.
- Por último, se recogen los anexos.
    - En primer lugar tenemos el manual de usuario, que acompañaría a cada copia del juego y explica el funcionamiento del mismo.
    - A continuación se incluye la planificación que se ha seguido a lo largo del desarrollo del proyecto.
    - Posteriormente se incluye el presupuesto derivado del desarrollo del videojuego.
    - Finalmente se incluye el anexo con las definiciones y las referencias.

CAPÍTULO 2: ESTADO DE LA CUESTIÓN

En lugar de plantear este capítulo desde el punto de vista de la historia de los videojuegos y, posteriormente, hablar de la historia de los matamarcianos, se ha considerado más apropiado contar como se inició una industria que actualmente genera grandes beneficios, y posteriormente enfocar la historia hacia el género mencionado.

En primer lugar se muestra un repaso al origen de los videojuegos. Posteriormente, un repaso a la historia de los matamarcianos desde el primer juego de dicho género hasta el día de hoy. Finalmente, se comentarán las distintas herramientas disponibles para programar videojuegos, para terminar explicando XNA Game Studio 3.1, la alternativa elegida.

2.1 El origen de los videojuegos

2.1.1 Los precursores de la industria

Mucha gente sitúa el inicio de los videojuegos en mayo de 1972, cuando salió al mercado el mítico Pong. Sin embargo, antes de ese hecho hubo numerosos aparatos que podrían considerarse como videojuegos prehistóricos [9]. En este apartado se van a repasar los más destacados.

1947: Lanzamiento de misiles

Después de la segunda guerra mundial, en el año 1947, se patentó un sistema de juego electrónico consistente en la utilización de misiles para eliminar un objetivo, denominado Lanzamiento de misiles. Fue creado por Thomas T. Goldsmith y Estle Ray Mann, que se inspiraron en las pantallas de radar utilizadas en la Segunda Guerra Mundial. Utilizaron válvulas y una pantalla de rayos catódicos para hacerlo funcionar, y permitía ajustar tanto la curva del lanzamiento como la velocidad de disparo. No obstante, no presentaba características propias de los videojuegos como el movimiento en pantalla, los objetivos venían sobre impresionados, por lo que no se le puede considerar un videojuego, pero sí uno de los muchos precursores de la industria.

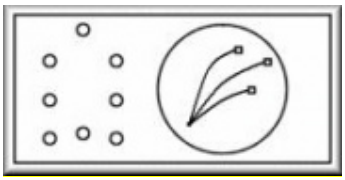


Figura 1. Lanzamiento de mísiles

1952: Tres en Raya

Tres en Raya, también conocido como “OXO”, “Nought and crosses”, “Ceros y cruces” o “Tic-Tac-Toe”, se convirtió en el primer juego de tablero en tener su versión digital. Funcionaba en una computadora ESDAC, y surgió de la tesis doctoral de Alexander Sandy Douglas en la Universidad de Cambridge.

El objetivo del juego es el mismo que en su versión de tablero, dos jugadores compiten para tratar de alinear tres círculos o cruces en horizontal, vertical o diagonal utilizando un dial telefónico integrado en la ESDAC, en una pantalla con una resolución de 35x16 píxeles. Como sucedía con el juego anterior, éste tampoco presentaba movimiento en pantalla por lo que más que un videojuego, se le puede considerar como un juego gráfico por ordenador. Aun así, hay mucha gente que lo considera el primer videojuego de la historia.

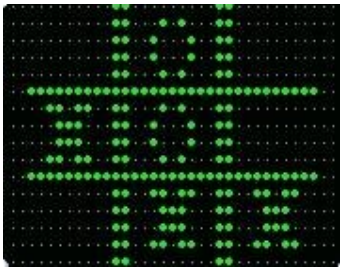


Figura 2. OXO

1958: Tennis for Two

En los laboratorios Brookhaven National trabajaba un físico llamado William Higinbotham, quién además de ocuparse de diseñar circuitos electrónicos, era aficionado a las máquinas de Pinball.

Figura 3. William Higinbotham

Se iba a celebrar un día de puertas abiertas en el laboratorio, por lo que mucha gente accedería a las instalaciones. Higinbotham decidió sorprender a los visitantes que se acercaran aquel día con un juego de entretenimiento. Con un osciloscopio, de ahí que la pantalla fuera circular, creó un juego utilizando un programa de cálculo de trayectorias conocido a día de hoy como “Tennis for Two”. Fue un éxito rotundo, los visitantes hacían largas colas para jugar una partida. Se podría decir que de este juego surgió el primer análisis de videojuegos de la historia, pues un estudiante que visitó el laboratorio aquel día, sería el fundador de la revista Creative Computing Magazine, donde escribiría sus impresiones sobre el aparato.

El juego presentaba una perspectiva lateral, a diferencia de la perspectiva aérea tan utilizada en los videojuegos de tenis, y representaba el campo de juego con una línea horizontal para determinar el suelo y una línea vertical en mitad de la pantalla que se correspondía con la red, como se puede ver en la figura 4. Dos jugadores tomaban el control y jugaban un partido en el que, además de golpear la pelota, podían elegir el ángulo en el que ésta saldría.

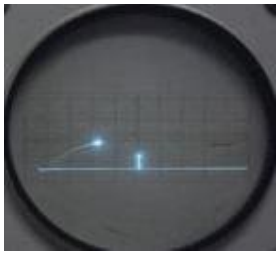


Figura 4. Tennis for Two

"Tennis for Two" fue la atracción principal de aquel día de puertas abiertas, sin embargo Higinbotham nunca se plateó patentar su invento y comercializarlo, ya que no lo consideró como un invento importante. Quizás si hubiera tenido otra perspectiva podría haberse convertido en una de las personas más ricas del mundo.

Varios años después de aquel evento, la máquina fue desmantelada. Dedicar tantos recursos a un simple juego de tenis de mesa era más de lo que se podían permitir en el laboratorio.

1961: Spacewar!

En el año 1961 el MIT (Instituto Tecnológico de Massachussets) recibe a modo de donación un ordenador último modelo, el PDP-1, con un tamaño tal que podría entrar tranquilamente en una cocina.

Tres miembros del equipo Wayne Witanen, Martin Graetz y Steve Russel comenzaron a debatir acerca de la aplicación que podrían diseñar con el ordenador que había llegado a sus manos. Finalmente optaron por desarrollar un juego que mezclara acción y habilidad, Spacewar!

El juego consistía en el enfrentamiento intergaláctico de dos naves, cada una controlada por un jugador, que dispondrían de un arma, un depósito de combustible y la capacidad de saltar al hiperespacio.

Figura 5. Steve Russell

Sin embargo, los problemas no tardaron en surgir. Witanen fue llamado por el ejército, y Graetz fue trasladado a otro departamento, por lo que Russel termina desmotivándose y abandonando el proyecto. Se disculpó aduciendo que no era capaz de obtener la función de seno/coseno necesaria para realizar el juego, pero los miembros del equipo pensaban que no era más que una excusa para dejar de lado el proyecto.

Cansado de excusas, Alan Kotok se presenta ante Russel con la función que éste necesitaba, o decía necesitar, para continuar el proyecto. Por ello, no le queda más remedio que ponerse manos a la obra en diciembre de 1961.

Un mes después ya tiene dos elementos en pantalla, conocidos como aguja y cuña por su aspecto, y comienza a trabajar en el control de armas y en el sistema de detección de colisiones, mientras Graentz, que pese a estar en otro departamento sigue ayudando a Russel, se ocupa de la función de hiperespacio. En febrero Russel dispone de una beta del juego, que utiliza para depurar errores. Además de las características mencionadas, dos miembros del equipo crearon un fondo de estrellas y un sol con gravedad propia que aparece en mitad del campo de batalla.

En abril de 1962, tras más de 200 horas de trabajo, Spacewar! Se puede considerar como un producto terminado. De cara a la jornada de puertas abiertas incluyeron un limitador de tiempo y una pantalla adicionar para que el público tuviera una buena visión de los enfrentamientos.

Figura 6. Spacewar!

Además de cuidarse del rival, las nuevas modificaciones habían provocado que los jugadores tuvieran que vigilar el sol, para no morir abrasados, además de usar con cuidado la función de hiperespacio que movía la nave a una zona aleatoria de la pantalla, incluyendo las proximidades del sol.

Spacewar! es considerado uno de los primeros videojuegos de la historia, e incluso puede considerársele como el primer matamarcianos, aunque ese privilegio se lo llevará siempre Space Invaders, juego del que se hablará más adelante. No tardó en circular de mano en mano mediante cintas magnéticas e, incluso, por la red de Arpanet, por lo que pocas universidades de la época se quedan sin su copia de Spacewar!

1971: Computer Space.

La importancia de este juego no radica en su mecánica o en sus ideas, sino en su creador: Nolan Bushnell. Nolan nació en 1943 y se licenció en ingeniería electrónica en la Universidad de Utah.

Durante su carrera, se encontró con una copia de Spacewar! y, a diferencia de Higinbotham, Bushnell si vio en este tipo de juegos un gran potencial y una gran oportunidad de hacer dinero. Antes de ser contratado por Ampex en 1969, creó sus propias versiones de juegos como el Tres en Raya o el Fox and Geese.

Más adelante comenzó a gestar su propia versión de Spacewar!, pero empezó centrándose en la presentación del juego en pantalla, aspecto que redujo considerablemente los costes en componentes necesarios para el videojuego. Dedicó varios meses a desarrollar el juego en sus ratos libres, hasta que en 1971 lo consiguió. Ahora, tenía que encontrar la forma de comercializarlo. Coincidió con Bill Nutting, fundador de Nutting Associates, una empresa que pasaba por graves problemas económicos y que buscaba cualquier idea atractiva que pudiera sacarla a flote, Bushnell estaba ahí para darle a Nutting la idea que necesitaba.

Emplearon la universidad de Stamford como terreno de pruebas, para ello diseñaron una cubierta de aspecto futurista y un manual de instrucciones más complicado de lo que se esperaba y situaron la máquina en la cafetería de dicha universidad. El juego fue un éxito entre los estudiantes, pero no funcionó a nivel global. Bushnell aprendió mucho de este experimento y aplicaría dichos conocimientos en el futuro, cuando fundara la compañía conocida como Atari.

2.1.2 Pong, el nacimiento de la industria

1972 es considerado como el año clave del comienzo de la industria de los videojuegos, y Pong el videojuego original. Para ello, se van a dar a conocer los hechos que llevaron a la fabricación tanto de la primera videoconsola [10], como del primer videojuego.

Ralph Baer y su Magnavox Odyssey.

Actualmente cada cierto número de años las empresas de hardware más importantes sacan nuevos modelos de videoconsolas al mercado. Una práctica que fue iniciada gracias a la ambición de Ralph Baer, considerado el padre de las videoconsolas.

Figura 7. Ralph Baer

Baer se licenció en ingeniería de televisión tras trabajar para la inteligencia militar durante la Segunda Guerra Mundial. Quería diseñar y construir un sistema que permitiera reproducir diversas aplicaciones en los televisores caseros, es decir, una videconsola.

Su periplo comenzó en 1966 cuando diseñó y construyó la primera videoconsola de la historia. Baer tenía los conocimientos y la ambición, y sabía lo que tenía que hacer. Solamente le faltaba el dinero, por lo que buscó empresas que creyeran en su proyecto, pero no podría conseguirlo hasta el año 1972.

En 1967 se unieron a su equipo dos personas, Bill Harrison y Bill Rusch. Este último resultó ser una gran incorporación puesto que tenía una visión clara de cómo debía ser un videojuego y que elementos debía contener para ser jugable. El 11 de noviembre de ese mismo año terminaron un juego de ping pong para dos jugadores. Cuando llegó la hora de distribuir y comercializar el juego, se pusieron en contacto con TelePrompter Corporation, la mayor compañía de cable de Estados Unidos. Querían ofrecer un canal de videojuegos por cable, algo que impresionó a los directivos, no obstante, la creciente recesión era tan fuerte que muchas compañías de telecomunicaciones, TelePrompter no fue una excepción, tuvieron que reducir plantilla para no entrar en bancarrota, por lo que el proyecto no llegó a realizarse.

Dejando la idea del canal de videojuegos por cable, Baer y su equipo volvieron a su idea principal: diseñar una videoconsola. En esta ocasión se pusieron en contacto con fabricantes de aparatos electrónicos, y trataron de convencerles de la buena idea que tenían entre manos. Sin embargo, aunque tuvieron contacto con muchas empresas como RCA, Zenith, Motorola o Sylvana, ninguna de ellas llegó a buen puerto.

En el año 1971, Magnavox se interesaría por la denominada “Caja Marrón” como se conocía a la máquina que reproducía el juego de ping pong, y cerraría un trato con el equipo de Baer para su desarrollo y comercialización.

Figura 8. Magnavox Odyssey

Así surgió la Magnavox Odyssey en 1972, la primera videoconsola de la historia. Era capaz de reproducir un juego de ping pong generando dos puntos por pantalla, una línea central y una pelota. El set de videojuegos venía acompañado de diversas láminas que debían ponerse en la televisión. La Magnavox Odyssey no tenía circuitos integrados, sino que consistía en una serie de transmisores y diodos, y los cartuchos más bien eran *jumpers*. La videoconsola era capaz de ofrecer hasta 12 juegos diferentes si se utilizaban de manera adecuada los elementos que venían en la caja.

La creación de Atari.

En el año 1972 se unieron las mentes de Nolan Bushnell, Ted Dabney y Larry Bryan para dar lugar a una de las compañías más influyentes en los orígenes de la industria de los videojuegos e, incluso, en la actualidad.

Los tres se pusieron de acuerdo para aportar a partes iguales el capital necesario para fundar la empresa, empero Bryan se echó atrás en el último momento, y Bushnell y Dabney tuvieron que cubrir su parte. El primer nombre elegido fue "Syzygy" que significa literalmente "cualquiera de los dos puntos opuestos en la órbita de un planeta o satélite, especialmente la luna, donde está en oposición o conjunción con el Sol". Un nombre que, finalmente, no pudieron utilizar porque una compañía de velas de cera lo utilizaba. Así pues, se decantaron por Atari [11] y fundaron la compañía el 1 de Junio de 1972. La palabra Atari proviene de un juego de mesa oriental, el Go, y su significado es similar al "jaque mate" del ajedrez, un nombre acertado para una buena filosofía de negocio.

Figura 9. Logo original de Atari

En sus comienzos, Atari se sustentaba gracias a las ganancias, bastante escasas, que habían obtenido con la comercialización de Computer Space. Su primer cliente fue una empresa dedicada a las máquinas de Pinball y a las tragaperras. Realizaron los diversos encargos y situaron las máquinas en locales cercanos, para que los costes de desplazamiento fueran prácticamente nulos.

El primer empleado de Atari fue una joven de 17 años que buscaba un empleo en verano para sacarse un dinero extra en su tiempo libre. Cynthia Villanueva terminó conviertiendose, además de en una empleada con contrato fijo, en la chica para todo de la compañía. No solo atendía a los clientes, sino que también se ocupaba de realizar diversos encargos de compra y distribución. Cynthia terminaría trabajando más tiempo en la empresa que sus propios fundadores.

Nace Pong.

Llegó a oídos de Atari la presentación de un sistema doméstico de videojuegos, que no era otro que la Magnavox Odyssey de Ralph Baer, por lo que enviaron a Bushnell para que echara un vistazo. Fue allí donde nació la idea de desarrollar Pong.

El segundo empleado de Atari fue Allan Alcorn, a quién Bushell puso a trabajar en un juego de ping pong con la idea de que empezara a manejarse en el campo de desarrollo de videojuegos. Allcorn fue una pieza clave del nacimiento de Pong, puesto que empezó a eliminar componentes, de forma que redujo los costes de fabricación, mientras le daba la forma que quería. Cambió las palas de ping pong por unos rectángulos divididos en varias secciones, cada una de las cuales generaba un ángulo de reflexión distinto. Además dotó a la pelota de aceleración, para que el partido fuera complicándose a medida que avanzaba.

Figura 10. Mueble de Pong

Después de tres meses de trabajo, Allcorn presentó a sus jefes el primer prototipo con el que Bushnell y Dabney quedaron asombrados. Decidieron llamarlo Pong, y comenzaron su experimento en alguno de los bares donde tenían instaladas las máquinas de pinball de Bally. El propietario de uno de esos bares, concretamente el Andu Capp's Tavern, llamó a Bushnell una noche diciéndole que viniera a arreglar la máquina porque se había estropeado. Al llegar, Bushnell se encontró con que el cajetín de monedas estaba colapsado, al no entrar ninguna moneda más, no se podía jugar.

Atari se dio cuenta del negocio que tenía entre manos, pero se encontró con un gran problema: tenía un acuerdo con Bally y Midway para desarrollar un juego creado por Atari. La solución consistió en un pequeño engaño, Bushnell explicó a Bally que Midway ya no estaba interesada en el proyecto y viceversa, de esta forma ambas empresas se desentendieron, y Atari tuvo luz verde para comercializar Pong por su cuenta.

Pong fue un éxito rotundo en aquella época y se convirtió en el juego que más dinero había generado. Cambiaría para siempre una industria que dura hasta el día de hoy.

2.2 Historia de los matamarcianos

2.2.1 Definición

De un modo general podemos definir este género como aquel en el que el jugador a los mandos de una nave espacial o sucedánea se enfrenta a numerosas oleadas de enemigos de carácter extraterrestre. No obstante, a lo largo de los años el género ha ido evolucionando y pasando por diversas fases. Así se puede encontrar mucha variedad y diferencias entre unos títulos y otros, desde los primeros juegos de limitadísimo movimiento hasta los más recientes que llenan la pantalla de disparos y enemigos cuyos patrones se deben memorizar si se quiere sobrevivir.

Se trata sin duda de uno de los géneros más populares de la historia de los videojuegos y de uno de los que más se identifica con el género. Hoy en día todavía hay gente que utiliza la expresión "jugar a los marcianitos" para referirse al hecho de jugar a los videojuegos.

2.2.2 Space Invaders, el origen de un género

A finales de la década de los 70, y después del nacimiento de Pong, se empezó a gestar la gran industria en la que se han convertido los videojuegos hoy en día.

Diversas compañías se interesan por este mundo [12], empiezan a surgir los clones de Pong, y nuevas novedades. Japón, de la mano de Namco, empresa fundada por Masaya Nakamura, empieza a interesarse por los videojuegos y termina siendo la distribuidora oficial de Atari en Japón.

Incluso Nintendo [13], en el año 1977, lanza al mercado su primera videoconsola. Además, un tal Shigeru Miyamoto empezó a trabajar ese mismo año como diseñador de *artwork* para arcades.

Figura 11. Nintendo Tv Game 6

A finales de la década, en 1979 surgen también varios hechos relevantes, tales como la comercialización del mítico Pac-Man [14], el comienzo de Capcom [15] en la industria de los videojuegos o la fundación de Activision [16], la primera *third-party* de la historia, de la mano de varios programadores de Atari que dejaron la empresa bastante descontentos: Larry Kapla, Alan Miller, Bob Whitehead, David Crane y Jim Levy.

Y es en medio de todo este tumulto donde surge el tema que importa: el lanzamiento de Space Invaders [17], el primer matamarcianos de la historia, y un absoluto éxito de ventas.

Taito [18] era una empresa que se dedicaba a fabricar máquinas tragaperras y pinballs, y no sería hasta 1973 cuando se introduciría en el mundo de las máquinas recreativas. En 1978 Toshihiro Nishikado, un programador de Taito, llegó a la empresa con la idea de un soldado que se ocultaba en una trinchera y disparaba a los ejércitos enemigos que se acercaban a su posición. Una idea que no gustó a la compañía por las connotaciones de violencia que tenía, por lo que se optó por cambiar a los humanos por alienígenas y naves espaciales.

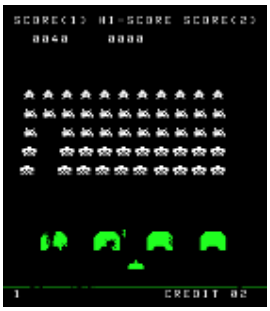


Figura 12. Space Invaders

Así surgió Space Invaders, la recreativa que, para mucha gente, dio a conocer los videojuegos al mundo. La idea inicial había cambiado, en lugar de tener un soldado que se ocultaba en una trinchera y disparaba al ejército enemigo, se tenía una nave espacial que podía cubrirse tras una serie de escudos destruibles, e iba eliminando oleadas de naves enemigas que se acercaban, cada vez más rápido, a su posición.

No obstante, la mayor innovación de Space Invaders fue la inclusión del *Hi-Score*. El ser humano tiene instinto de superación, y este aspecto provocó que la gente tratara de jugar la partida perfecta para situarse en lo más alto de la clasificación y dejar su huella en cada una de las recreativas. Las partidas de Space Invaders podían durar horas, o incluso días, puesto que las habituales limitaciones de tiempo habían sido sustituidas por tres vidas. De esta forma, los jugadores más diestros podían pasar mucho más tiempo jugando.

Space Invaders fue todo un éxito, no sólo en el país del sol naciente, sino en el resto del mundo. Todos querían tener un Space Invaders. A día de hoy la franquicia ha facturado más de 500 millones de dólares, y ha dejado tras de sí una larga serie de anécdotas como la chica que robó 5.000 dólares a sus padres para jugar en la recreativa, o Eric Furrer, un niño de 12 años que jugó durante 38 horas y media para lograr la mayor puntuación del momento: 1.114.000 puntos. Además, se cuenta que el gobierno japonés tuvo que triplicar la producción de monedas, ya que escaseaban debido a la cantidad de éstas que se empleaban en las recreativas.

Los clones no tardarían en aparecer, las empresas de desarrollo habían visto el negocio que estaba haciendo Taito y querían su parte. Como se verá en los puntos siguientes, cada nueva idea intentaría aportar conceptos y características nuevas para diferenciar su producto y hacerlo más atractivo de cara al usuario.

2.2.3 Definiendo el género de los matamarcianos

Después del increíble éxito cosechado por el juego de Taito, las demás empresas desarrolladoras de videojuegos no tardaron en subirse al barco y aportar sus obras a la industria [19]. Sería durante la década de los ochenta cuando se introducirían todo tipo de elementos y mejoras a este género y donde se convertiría en lo que es hoy en día.

La batuta la llevó Asteroids, un juego lanzado en 1979 y fuertemente inspirado en Spacewar! que incorporaba elementos tales como la rotación de la nave o la división de los enemigos grandes en otros más pequeños a medida que se destruían, hasta llegar al tamaño base desde el que eran finalmente eliminados.

Defender, lanzado al mercado en 1891, aportó su granito de arena incorporando el movimiento de *scroll*, de forma que el jugador podía desplazarse libremente por toda el área de combate. Fue una aportación interesante puesto que hasta ahora todos los juegos incorporaban un fondo estático. Atari no tardó en aparecer en escena con Tempest, también lanzado en 1981. Se trataba de un juego adelantado a su época que, gracias a su perspectiva, sentaría las bases de lo que serían los *shooters* 3D.

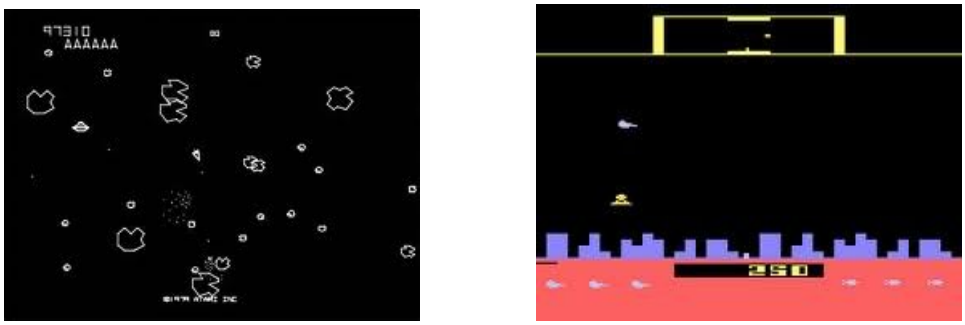


Figura 13. Primeros matamarcianos

El uso del *scroll* horizontal es una característica utilizada en muchos juegos, entre ellos grandes clásicos como R-Type o Gradius, pero el primer videojuego que lo emplearía sería Scramble, de Konami. Además de esta novedad, incluyó una barra de combustible que necesitaba ser recargada disparando a varios bidones que aparecían a lo largo de la pantalla. No solo había que cuidarse de sobrevivir, sino que también había que vigilar el combustible para no terminar perdiendo una vida.

Sin duda la perspectiva más utilizada y más característica de los matamarcianos es la perspectiva vertical forzada o *scroll* vertical. Pero ésta no se emplearía hasta 1982, año en el que salió al mercado Xevious [20] de Namco. Un juego que, además del mencionado scroll vertical, aportaría un arma secundaria que permitía disparar a enemigos terrestres. Toda una novedad, ahora llegaba el momento de diferenciar entre objetivos aéreos y terrestres, y disparar una u otra arma en función del enemigo que se quisiera destruir.

Figura 14. Matamarcianos con movimiento de *scroll*

Muchos fueron los juegos que salieron al mercado durante esta época, pues todas las empresas querían hacerse con su cuota de mercado. Sin embargo, los que se han mencionado se consideran los más relevantes y los que mayores mejoras aportaron en su momento.

2.2.3 Evolución y nacimiento de grandes sagas

El género de los matamarcianos estaba en auge. Las compañías sacaban más y más juegos, la gente estaba deseosa de probar las últimas novedades, bien en los salones recreativos, bien en la comodidad del hogar gracias a los sistemas domésticos. Había una gran pasión por este género, y no fueron pocos los videojuegos que aparecieron en esta década y que, aún a día de hoy, son considerados como auténticas obras de arte del entretenimiento.

Cada vez se iban incluyendo más elementos, o se adaptaban los ya existentes. Además, la evolución gráfica se iba haciendo notable como se pudo apreciar en el mencionado Xevious, que contaba con unos paisajes donde se podían diferenciar distintos elementos y edificaciones. En 1982 SEGA se atreve con Zaxxon [21] un juego con una perspectiva de lo más peculiar, la axonométrica, que ponía al jugador a los mandos de un caza que se aventuraba en el interior de una base enemiga. Incorporaba además un altímetro, pues había que sortear diversas estructuras a lo largo de las fases, por lo que el control de la altura era un elemento indispensable en el juego.

Figura 15. Dos nuevas propuestas: Zaxxon y 1942

Dos años más tarde, en 1984, llegarían Robotron 2084 y 1942. Capcom desarrolló 1942 [22], un juego ambientado en la Segunda Guerra Mundial, más concretamente en el año que daba el título al juego. El jugador debía ponerse a los mandos de un avión estadounidense y adentrarse en territorio japonés para llevar a cabo diversas misiones. Se convertiría en una de las primeras franquicias de Capcom, y tendría su secuela, 1943: La batalla de Midway, cinco años después. Robotron 2084 [23] por su parte apostaba por una ambientación más futurista y un sistema de juego que permitía disparar en cualquier dirección. Además, en lugar de manejar una nave, el jugador debía tomar el control de un robot y enfrentarse a multitud de androides.

Al año siguiente, en 1985, fueron lanzados al mercado dos de los juegos más recordados y más queridos por la comunidad de jugadores más fieles a este género: Space Harrier y Gradius.

Figura 16. Clásicos de la década de los 80

Space Harrier [24] fue otro gran éxito de SEGA [25]. Quizás no se trate de un matamarcianos habitual, pero es un juego que sentó las bases de los *shooters* sobre raíles que se crearían años más tarde. El juego ponía al jugador en la piel de un guerrero armado en medio de un mundo fantástico que debía avanzar esquivando o matando enemigos durante las fases, y acabando con los enormes y espectaculares jefes finales al final de cada una de esas fases. La principal característica de este juego era su perspectiva, el sprite del jugador estaba continuamente corriendo hacia delante y gracias al reescalado de *sprites* se generaba un efecto 3D que daba la impresión de estar adentrándose en la pantalla. Algo nunca visto hasta ahora.

Por su parte Gradius [26], desarrollado por Konami [27], incorporó por primera vez la variedad de armamento. Aunque anteriormente se había incluido un disparo secundario, como el que existía en Xevious, Gradius permitía cambiar el armamento principal e, incluso, almacenar las distintas armas que se recogían y alternar entre ellos en función de la situación. El jugador comenzaba disponiendo de una única arma y, a medida que iba destruyendo enemigos, recogía *power ups* que le proporcionaban armamento adicional. Todos ellos quedaban almacenados en una barra situada en la parte inferior para que fuera más sencillo alternar entre ellas y tener presente de qué armas se disponía. Esta característica aportó un toque estratégico que nunca antes había tenido un matamarcianos.

Los éxitos de 1985 no terminaron aquí. También fue un año caracterizado por la llegada al mercado de dos juegos que cambiarían el aspecto serio que se tenía hasta la fecha de los videojuegos. Se trata de Fantasy Zone y Twinbee.

Fantasy Zone [28] llegó de la mano de, nuevamente, SEGA y su protagonista, la famosa nave Opa-Opa sería considerada la mascota de la compañía hasta la llegada del Alex Kidd, no en vano recibiría una secuela dos años después: Fantasy Zone: The Tears of Opa-Opa. El juego presentaba un mundo fantástico y colorido, dentro de una temática desenfada y permitía al jugador moverse libremente por las distintas fases hasta que eliminaba todos los generadores de enemigos. Una vez acabada la tarea, era el turno de enfrentarse al característico y gigantesco jefe final de fase.

Figura 17. Matamarcianos de corte desenfadado

Twinbee [29] desarrollado por Konami, otra de las grandes empresas de la época, también presentaba un corte más desenfadado que la mayoría de los juegos de la época. A las características comunes que compartía con otros juegos como el arma secundaria para enemigos terrestres o el *scroll* horizontal, había que añadirle la inclusión de un modo multijugador donde, por primera vez, ambos jugadores aparecían en pantalla y cooperaban para superar el juego, en lugar de jugar por turnos como se había hecho hasta entonces. Además, el sistema de mejoras podía ser manipulado por el jugador para recoger lo que más le interesara en cada momento, puesto que cada vez que un *power up* era alcanzado por un disparo cambiaba de color, y dependiendo del color se adquiría una mejora u otra. Un toque distintivo y estratégico que hizo de Twinbee uno de los grandes juegos de la década.

Hubo que esperar hasta el año 1987 para recibir la primera entrega de una de las sagas de matamarcianos más reconocidas a nivel mundial: R-Type [30], desarrollado por Irem [31]. R-Type presentaba un matamarcianos de scroll horizontal caracterizado por un ritmo más pausado, en detrimento de la acción frenética que presentaban la mayoría de sus competidores. Sin embargo, la característica más importante fue la incorporación de la cápsula "Force", elemento distintivo que permanecería en todas y cada una de las secuelas que aparecerían en años posteriores. Esta cápsula ayudaba en tareas defensivas y ofensivas, y sus características variaban en función de los *power ups* que recogiera el jugador. El juego también se recuerda por la variedad y el diseño de los enemigos, y lo detallado de los escenarios.

2.2.4 Redefiniendo el género. Evolución hacia los Bullet Hell

A principios de la década de los 90 [32], el género vuelve a sufrir una nueva transformación. Aunque siguen apareciendo juegos más pausados y técnicos, la corriente empieza a girar en torno a juegos con acción frenética donde priman los reflejos y la memorización de fases enteras.

Uno de los juegos más reconocidos de esta época es Aero Fighters [33], desarrollado por Video System, que toma el frenesí como base del juego. Al comienzo se podía elegir la nacionalidad del piloto, lo que determinaba el tipo de armamento. Dicho armamento podía ir mejorándose recogiendo power ups durante la pantalla, y podía terminar abarcando una buena parte de ella. Pese a ello, eliminar a los enemigos no era una tarea fácil, y requería de buena capacidad de memorización para anticiparse a sus movimientos y acabar con ellos.

Figura 18. Matamarcianos en la década de los 90

Otros juegos destacados son Eco Fighters, de Capcom, que introdujo esta corriente de acción frenética en los juegos de *scroll* horizontal, o Samurai Aces de Banpresto.

Muchas eran las compañías que seguían apostando por el tirón que tenían estos juegos, pero el género que de verdad se estaba gestando son los conocidos a días de hoy como *Bullet Hell Shooters* o *Maniac Shooters*. Los *bullet hell* nacieron de la necesidad de los desarrolladores de matamarcianos bidimensionales de competir con la emergente popularidad de los juegos tridimensionales. Los desarrolladores de estos juegos querían demostrar que, aunque sus juegos no fueran en 3D, podían mostrar cosas espectaculares como pantallas llenas de disparos. Batsugun [34], desarrollado por Toaplan y lanzado en 1993, proporcionó la plantilla prototípica en la que se basarían todos estos juegos.

Desde la creación de los *bullet hell* las compañías japonesas ven negocio en un nicho de mercado que, si bien reducido, tiene una legión fiel de seguidores que no dejan escapar un solo videojuego. Gunbird [35], desarrollado por Capcom y lanzado al mercado en 1994, es uno de los baluartes del género. Unos gráficos coloridos y una temática desenfada acompañada de personajes afables esconden un juego con una dificultad endiablada que poca gente es capaz de superar en sus niveles más difíciles. Pese a la gran dificultad de estos juegos, resultan tremendamente adictivos y los seguidores de este género siempre tienen paciencia para un intento más. Todo sea por conseguir terminar el juego y ver la secuencia final.

Una característica que mantienen los *bullet hell* con respecto a los matamarcianos de siempre es el tamaño de los jefes finales. El jugador se encuentra al final de cada fase con monstruosas naves que pueden llegar a ocupar media pantalla, y cuyas secuencias de disparos son capaces de cubrir de balas la otra mitad.

Figura 19.Los primeros Bullet Hell

Sin embargo, no todo son *maniac shooters*, aún queda hueco para experiencias como Air Gallet [36], 1996. La experiencia de Banpresto se acerca más a los matamarcianos clásicos, y su principal exponente es la cadencia de fuego y la amplitud de los distintos armamentos que puede encontrar el jugador a lo largo de la partida.

Si hay una compañía que supo entender los *bullet hell* y sacar partido de ellos, esa es Cave [37]. Jugabilidad a prueba de bombas, efectos visuales espectaculares y fluidez en los movimientos son las cartas de presentación de sus juegos. Lo más destacados sin ninguna duda son DonPachi [38], desarrollado en 1995 y publicado por Atlus, y su secuela, lanzada dos años después, DoDonPachi [39]. La historia de la primera entrega se aleja de los cánones habituales, el jugador se pone en la piel de un piloto que quiere entrar en un escuadrón de élite y debe sobrevivir a un largo y duro entrenamiento donde deberá enfrentarse a sus camaradas. En la segunda parte, volviendo a los tópicos habituales, el piloto ha entrado en el escuadrón DonPachi y debe hacer frente a una invasión de alienígenas mecánicos.

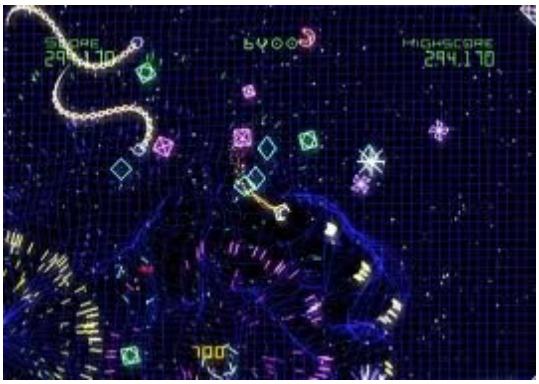
Muchos son los juegos lanzados al mercado en esta década. Los que se han mencionado se consideran los más relevantes y los mejores exponentes de un género que sigue vivo a día de hoy y cuenta todavía con un gran elenco de seguidores.

2.2.5 En la actualidad

En la actualidad, los matamarcianos siguen gozando de cierta popularidad. Pese a que durante unos años parecía que el género iba decayendo y no volvería a levantarse, está viviendo una nueva edad de oro gracias a los medios y tendencias actuales. Los sistemas de distribución digital actuales como Steam, XBOX Live o WiiWare, permiten a pequeñas empresas lanzarse al mundo de los videojuegos sin necesidad de realizar grandes inversiones, y desarrollar un matamarcianos es una opción tan buena como cualquier otra.

Los seguidores fieles de este tipo de juegos siguen recibiendo con gusto más y más títulos y una cierta tendencia actual a "lo retro" anima a nuevos usuarios a interesarse por juegos con una cierta apariencia arcaica. Space Invaders: Infinite Gene [40] es un ejemplo de esta tendencia. Coge la apariencia primitiva de Space Invaders, dos colores en pantalla, y modifica la jugabilidad para traernos un juego largo y tremendamente adictivo.

Otro ejemplo que sigue esta tendencia es Geometry Wars: Retro Evolved [41], desarrollado por Bizarre Games y publicado en XBOX Live Marketplace en 2005, que ha gozado de unas ventas tan buenas que el juego recibió una versión [42] tanto para Wii como para Nintendo DS.



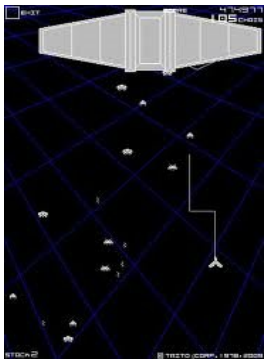


Figura 20. Matamarcianos actuales con estética retro

Sin embargo, pese a los altibajos, un género que no se ha visto resentido desde su nacimiento ha sido el de los *bullet hell*. Aunque es probable que más allá de las fronteras niponas no goce de mucha popularidad, en Japón es un género que tiene una legión fiel de seguidores. Buena muestra de ello es que todos los años se lanza una considerable cantidad de títulos.

Algunos destacables, además de por su calidad, porque han cruzado las fronteras japonesas son Deathsmiles, desarrollado por Cave y lanzado en 2007 o la Gundemonium Collection, colección de tres juegos desarrollada por Platine Dispositif en 2010.

Figura 21. Bullet Hell actuales

Deathsmiles [43] pone al jugador en la piel de una joven, a elegir entre cuatro, que debe usar sus poderes sobrenaturales para salvar el mundo de fantasía al que fue transportada cuando era pequeña. Una historia simple y llena de tópicos, que sirve de excusa para lanzar al usuario al juego en medio de una ingente cantidad de enemigos y disparos.

Por su parte la Gundemonium Collection [44] está compuesta por tres juegos: Gundemonium Recollection, Gundeadli Gne y Hitogata Happa. Los dos primeros son *bullet hell* de *scroll* horizontal, muy similares entre sí, en el que controlando a uno de los dos personajes disponibles, se debe librar una guerra contra las fuerzas infernales. Hitogata Happa por su parte, además de cambiar a un juego de *scroll* vertical, presenta la historia de una chica con la capacidad para controlar muñecas con poderes. Gracias a las muñecas, cada una con sus propias características, tanto en velocidad como en disparo, se puede hacer frente a un cruel imperio que sueña con la dominación del mundo. Pese a tratarse también de una historia muy tópica, el jugador se encuentra ante una colección de juegos divertida y adictiva.

Pese a los altibajos sufridos, se ha podido comprobar que el género de los matamarcianos aún está vivo, tiene mucho que decir y muchas horas de diversión que ofrecer. No siempre es tan importante realizar un producto innovador, como adaptar una fórmula existente y ofrecer un juego que sea capaz de divertir y animar a los jugadores para ver quién consigue el *Hi-Score*.

2.3 XNA Game Studio

Antes de profundizar en el API elegido para desarrollar el proyecto, se van a explorar las distintas alternativas disponibles.

2.3.1 Alternativas a XNA

Un motor de videojuego es un término que hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. La funcionalidad básica es proveer al videojuego de un motor de renderizado para gráficos en 2D y 3D, motor físico, detección de colisiones, sonidos, *scripting*, animación, inteligencia artificial, redes, *streaming*, administración de memoria y un escenario gráfico.

Hay motores compatibles tanto con videoconsolas, como con distintos sistemas operativos. Algunos son gratuitos y otros requieren licencia para ser utilizados. A continuación se enumeran y se describen brevemente los más importantes:

- Adventure Game Studio [45]: es una herramienta cuya principal funcionalidad es la creación de aventuras gráficas. El programa contiene una interfaz gráfica desde la que se puede crear la aventura gráfica sin ningún tipo de conocimiento de programación. No obstante, para los usuarios más experimentados se incluye un editor basado en *scripting* para programar dentro del juego.

- Allegro [46]: es una biblioteca para la programación de videojuegos desarrollada en C. Creada originalmente para Atari ST, ha evolucionado y hoy en día es compatible con DOS, Windows, Linux, y Mac OS X. Sus funcionalidades principales están orientadas a gráficos, sonidos, entrada de usuario y temporizadores.

- Blender 3D [47]: es una aplicación gratuita de diseño 3D. Su funcionalidad principal es el diseño de figuras y modelos en 3D, aunque también puede realizar simulación de fluidos, detección de colisiones o simulaciones físicas, por lo que en sus últimas versiones se ha utilizado para crear aplicaciones interactivas. Es compatible con Windows, Linux y Mac OS X.

- Crystal Space [48]: es un *framework* para el desarrollo de aplicaciones 3D escrito en C++. Crystal Space se usa típicamente como motor de juego pero el *framework* es más general y puede ser usado para cualquier tipo de visualización 3D. Crystal Space es portable y se ejecuta en Windows, Linux, UNIX y Mac OS X. Es *software* de código abierto licenciado bajo LGPL.

- dim3 [49]: es una herramienta de código libre orientada al diseño 3D y al desarrollo de videojuegos. Es compatible con Windows, Linux y Mac OS X. Utiliza OpenGL para el renderizado, OpenAL para el tratamiento de audio, JavaScript como lenguaje de *scripting*, XML como formato de datos y Simple Direct Media Layer como gestor de ventanas y vídeo.

- Gamebryo [50]: es un motor de gráficos 3D desarrollado en C++ y orientado al desarrollo de videojuegos. Una de sus mayores virtudes es que tiene soporte para casi todas las plataformas existentes, incluyendo Windows, Wii, Playstation 3, PSP o XBOX 360.

- Game Maker [51]: es una herramienta de desarrollo rápido de aplicaciones, basada en un lenguaje de programación interpretado y un paquete de desarrollo de software para desarrollar videojuegos, creado en el lenguaje de programación Delphi, y orientado a usuarios novatos o con pocas nociones de programación. La interfaz principal para el desarrollo de videojuegos de Game Maker usa un sistema de "arrastrar y soltar", que permite a los usuarios que no están familiarizados con la programación tradicional crear videojuegos intuitivamente organizando iconos en la pantalla. Game Maker viene con un conjunto de bibliotecas de acciones estándar, que cubren cosas como movimiento, dibujo básico, y control simple de estructuras. Para extender la funcionalidad de arrastrar y soltar de Game Maker, los usuarios pueden construir bibliotecas de acciones personalizadas para agregar nuevas acciones a sus videojuegos.

- Game Studio [52]: también conocido como 3D Game Studio, es una aplicación muy conocida para el desarrollo de juegos en 2D y 3D. El núcleo del programa se denomina A7, un motor gráfico que controla la imagen y el comportamiento del mundo virtual desarrollado. Es muy potente y trabaja de igual forma con espacios interiores y exteriores. También soporta sombras estáticas y dinámicas. A su motor 3D se le unen otros que complementan el programa y que permiten la producción de videojuegos con resultados muy espectaculares: motor de efectos y partículas, motor de física y colisiones, motor bidimensional, motor sonoro, motor de red, etc. La aplicación permite desarrollar un juego sin tener conocimientos de programación, pero para desarrolladores más expertos proporciona C-Script (también conocido como Lite-C), una versión simplificada de C++.

- MUGEN [53]: es un motor gráfico para crear juegos de lucha en 2D. Ha tenido mucho éxito entre los aficionados y hay una gran comunidad de usuarios que lo apoya debido a que con él, se pueden crear personajes, escenarios y barras de vida a partir de juegos existentes. Permite crear juegos a partir de personajes de distintos juegos y compañías; se puede desde simplemente añadir personajes de cualquier juego (que sean compatibles), escenarios, paquetes de visualización de pantalla o barras de vida; hasta crear un juego de peleas completo con todo un sistema establecido, personalizado y modificado al gusto del creador. Lo más destacado es la capacidad de crear juegos con personajes de cualquier otro juego del que se pueda sacar los *sprites* y programar sus movimientos.

- Ogre 3D [54]: es un motor de renderizado orientado al desarrollo de aplicaciones con modelados y elementos 3D, escrito en el lenguaje de programación C++. Sus bibliotecas evitan la dificultad de la utilización de capas inferiores de librerías gráficas como OpenGL y Direct3D, y además, provee una interfaz basada en objetos. El problema principal es que proporciona sólo funcionalidades relacionadas con la animación y el modelado 3D por lo que se queda muy corto (por ejemplo no incluye soporte audio ni de inteligencia artificial). El motor es libre, bajo licencia LGPL y con una comunidad muy activa. Ha llegado a usarse en algunos juegos comerciales.

- RPG Maker [55]: se trata de una herramienta que permite al usuario crear sus propios videojuegos de rol. Incluye un editor de mapas, un editor de eventos, y un editor de combates. Todas las versiones de RPG Maker de PC tienen el RTP (Run Time Package) que incluye materiales tales como gráficos para los mapas, personajes, música, efectos de sonido, etc. que pueden ser utilizados para crear nuevos juegos. Una característica interesante de las versiones de PC de RPG Maker es que el usuario puede agregar nuevos materiales gráficos y sonoros personalizados al proyecto. Muchos sitios y comunidades en la red se dedican a ayudar y compartir sus creaciones con los usuarios, también se dedican a compartir gráficos, sonidos, fondos y demás archivos que ayudan a la elaboración de una nueva creación, o la modificación de una ya creada anteriormente.

- Unreal Engine [56]: es un motor de juego de PC y consolas creado por la compañía Epic Games. Implementado inicialmente en el videojuego Unreal en 1998, también se ha utilizado en otros géneros como el rol y juegos de perspectiva en tercera persona. Está escrito en C++, siendo compatible con varias plataformas como PC (Microsoft Windows, GNU/Linux), Apple Macintosh (Mac OS, Mac OS X) y la mayoría de consolas (Dreamcast, Gamecube, Wii, Xbox, Xbox 360, PlayStation 2, PlayStation 3). Unreal Engine también ofrece varias herramientas adicionales de gran ayuda para diseñadores y artistas

- XNA: es un API de Microsoft orientado al desarrollo de videojuegos para PC, XBOX 360 y Zune. XNA pretende simplificar el desarrollo de videojuegos al programador, facilitando la gestión de gráficos, vídeo, sonidos, dispositivos, etc. Su problema es que carece de interfaz, por lo que no es apto para usuarios sin conocimientos de programación.

Estas son las herramientas más conocidas que se pueden encontrar en la red para desarrollar videojuegos. Antes de tomar una decisión, es importante analizar las ventajas y desventajas de cada una de ellas.

En primer lugar se van a descartar las herramientas Blender 3D, Crystal Space, dim3, Gamebryo y Ogre 3D; la razón no es otra que su perspectiva. Son herramientas creadas para diseñar juegos en 3D, y el objetivo de este proyecto es diseñar un shooter de naves en 2D, por lo tanto no puede emplearse ninguna de estas cinco herramientas.

Aventure Game Studio, MUGEN y RPG Maker también se descartan porque están orientados hacía un género distinto al que se quiere abordar. Se considera más apropiado utilizar una herramienta genérica.

Game Studio se descarta por tratarse de una herramienta de pago y algo similar ocurre con Unreal Engine. Si bien es gratis hacerse con el segundo, hay que pagar un porcentaje a la compañía si se utiliza para desarrollar alguna aplicación. Game Maker por su parte queda descartado por hacer uso de un lenguaje casi desconocido para el equipo de desarrollo.

Finalmente quedan dos opciones: Allegro y XNA. Sin embargo, la opción elegida será XNA. Esto se debe a que se trata de una herramienta más extendida, con mayor cantidad de documentación y soporte, y que permite programar para varios dispositivos.

En el apartado siguiente se explicará con más detalle que es XNA Game Studio, sus ventajas y la versión que se empleará.

2.3.2 ¿Por qué XNA?

Micrososft XNA Game Studio es un conjunto de herramientas con un entorno de ejecución proporcionado por Microsoft que facilita el desarrollo y gestión de juegos para las plataformas PC, XBOX 360 y Zune. Su nombre viene del juego de palabras *XNA is Not an Acronym* (XNA no es un acrónimo) elegido debido a que los productos y tecnologías de Microsoft tienen tantos acrónimos que decidieron crear un nombre que se reflejara como un acrónimo pero que en la realidad no lo fuera.

XNA se compone de una serie de librerías y aplicaciones construidas sobre el .NET Framework 2.0 de Microsoft que proveen a los lenguajes .NET de un conjunto de herramientas con el objetivo de programar videojuegos. Se compone de dos librerías que funcionan sobre el .NET Framework 2.0 y proporcionan una conexión entre el código manejado de los lenguajes .NET y la librería de sistema para multimedia por excelencia de Microsoft: DirectX.

El modelo de programación en XNA está basado en componentes y contenedores. El juego (clase *Game*) es en sí mismo el contenedor de todos los componentes (clases *GameComponent* y *DrawableGameComponent*) y se encarga de manejarlos automáticamente, lo que evita mucho trabajo de estructuración y programación al desarrollador. Todos los *GameComponent* tienen su método *Update* donde se encuentra la lógica del juego para ese componente. Los *DrawableGameComponent* además añaden el método *Draw*, que contiene las órdenes para renderizar el componente. Estos métodos se llaman automáticamente y pueden ser reescritos para crear componentes personalizados.

XNA Game Studio funciona con todas las versiones de Visual Studio 2008, incluida la versión gratuita. Está pensado para trabajar con el lenguaje C#. Al ser una biblioteca de ensamblados, nada impide que se puedan referenciar dichas bibliotecas en proyectos con otros lenguajes .NET, como por ejemplo Visual Basic. Sin embargo, el desarrollo para XBOX 360 y Zune sólo soporta C# ya que el Compact Framework de ambos no soporta otros tipos de ensamblado.

XNA fue creado con el objetivo de facilitar al programador la creación de juegos, evitando que éste se enfrente a problemas de integración con la plataforma, los dispositivos gráficos, etc. Aunque tiene la desventaja de carecer de interfaz como muchas otras herramientas, lo que lo excluye para usuarios sin conocimientos de programación, hay que destacar que desarrollar con XNA es realmente cómodo. Además desarrollar juegos o componentes con XNA es totalmente gratuito.

Hay varios motivos por los que decantarse por XNA. En primer lugar, como se ha mencionado, es una herramienta gratuita que permite programar para varios dispositivos, todos ellos con una buena base de usuarios, lo que permite que los juegos lleguen a mucha gente. Por ejemplo en PC, Steam, la mayor plataforma de distribución digital de videojuegos, permite programar en XNA y subir los juegos de forma gratuita. En el momento en el que aceptan el juego, se negocia un porcentaje que suele rondar el 70% de los beneficios para el desarrollador, y el 30% para el usuario.

Otra razón importante es la posibilidad de desarrollar aplicaciones tanto en 2D como en 3D. XNA tiene soporte para ambos tipos de juegos, lo que le da versatilidad para adaptarse a las necesidades del programador.

Además, es importante destacar la gran cantidad de documentación [57] y librerías que hay disponibles para XNA y a las que todo el mundo puede acceder. Hay multitud de comunidades y foros relacionados con XNA donde puedes solicitar ayuda o consultar documentación, y multitud de tutoriales [58] con una buena curva de aprendizaje para los que quieren iniciarse en la materia.

En conclusión, se trata de una herramienta potente, versátil y con mucho soporte, que destaca por su facilidad de uso al estar diseñada para desarrolladores *amateur*. Su robustez, el testeo ante fallos, las continuas revisiones y actualizaciones, y la facilidad de conversión, que permite desde un mismo código generar versiones para PC, XBOX 360 y Zune; también son buenas bazas para decantarse por esta herramienta.

2.3.3 XNA 3.1

Al comenzar el proyecto la última versión de XNA disponible era la versión 4.0. No obstante, dicha versión todavía estaba en fase de pruebas, por lo que se optó por utilizar la versión 3.1, que llevaba un tiempo disponible, y se trataba de una versión sólida y fiable del API de desarrollo. Por ello, fue la versión elegida para desarrollar el proyecto.

La versión 3.1 trajo consigo cuantiosas mejoras, unas más notables que otras. A continuación se enumeran algunas de ellas:

- Se incluye una librería de manejo de vídeo, que permite reproducir vídeos ya generados dentro del propio juego para utilizarlos como secuencias del mismo. Dentro de las funcionalidades de control de vídeos se incluyen: la reproducción a pantalla completa, control de la reproducción (P*lay*, P*ause* y S*top*), control sobre las propiedades del vídeo, etc.

- Una mejora en la librería de manejo de audio. En la versión 3.0 surgían problemas con algunas tarjetas de sonido, e instrucciones de carga de elementos de audio provocaban excepciones del tipo *InvalidOperationException*. Además de solventar este problema, arregla también un aspecto criticado en versiones anteriores que provocaba problemas a la hora de reproducir varios sonidos de forma simultánea.

- También se incluye la posibilidad de utilizar los avatares de XBOX Live como personajes de los juegos desarrollados. No obstante, esta funcionalidad sólo es compatible con juegos creados para XBOX 360.

Por último, es interesante destacar que la llegada de la última versión trajo consigo una reestructuración de la comunidad de XBOX Live. La comunidad antes conocida como XBOX Live Arcade que albergaba juegos tanto profesionales como *amateur*, se subdividió. XBOX Live Arcade queda para juegos desarrollados por empresas profesionales del sector, mientras que surge XBOX Live Indie Games, dedicado a desarrolladores noveles.

CAPÍTULO 3: ANÁLISIS

En este capítulo se va a realizar el análisis del sistema. Se empezará explicando el alcance del sistema, es decir, en que consiste, que funcionalidades tendrá, etc.

Posteriormente se incluirán los requisitos de usuario, divididos en requisitos de capacidad y de restricción, seguidos de los requisitos de software, funcionales y no funcionales.

Finalmente, se concluirá el capítulo con el análisis de los casos de uso, los diagramas de secuencia más relevantes y el diagrama de actividad del sistema.

3.1 Alcance del sistema

Como ya se comentó en el apartado 1.2 Objetivos del proyecto el videojuego a desarrollar consistirá en un *shooter* en 2D de *scroll* vertical. Dicho videojuego deberá contener, al menos, las siguientes características:

- Existirá un menú principal desde el que se accederá al juego y a un apartado de configuración.
- El usuario podrá elegir entre varias naves, cada una de ellas con distintas características.
- Habrá variedad de enemigos, tanto en el *sprite* como en el comportamiento.
- Se deben incluir jefes finales de fase.
- El *scroll* deberá desplazarse a medida que avanza la pantalla, hasta que se llegue al jefe final.
- Se deben incluir efectos de sonido y música de fondo.

Siempre que se incluya todo lo mencionado en la lista anterior, se dispone de plena libertad para añadir todas las mejoras que se consideren apropiadas para hacer más variado y entretenido el videojuego. Dichas mejoras pueden tratarse de aspectos como *powerups*, bombas, etc.

Adicionalmente, y para hacer más atractivo y duradero el producto, se acordó la realización de un editor de niveles que permita al usuario crear sus propias pantallas. De esta forma, una vez superado el juego inicial, el usuario podrá seguir disfrutando del mismo creando infinidad de niveles a su gusto, desde los más sencillos hasta los más complicados, teniendo a su disposición todos los elementos del juego.

Sin embargo, después de debatir sobre este apartado, se tomaron dos decisiones importantes:

- La primera hacía referencia al *scroll* de fondo. Debido a la cantidad de imágenes necesarias para mostrar un instante de juego, se decidió que era mejor tener una serie de *scrolls* predefinidos y dar al usuario la opción de elegir entre ellos, pues construirlo desde cero resulta ser una tarea larga y tediosa, a la que no estará dispuesta una gran cantidad de personas.
- La segunda hacía referencia a la inclusión de un editor más simple. Dicho editor será capaz de generar un nivel en base a una duración y una dificultad

establecidas por el usuario. De esta forma, se puede disponer de un nuevo nivel en apenas unos segundos.

3.2 Requisitos de usuario

En este punto del documento se pretende identificar, clasificar y catalogar los requisitos de usuario obtenidos a través del estudio del sistema que se demandó diseñar.

Los requisitos que se extraigan deben plasmar no sólo lo que el sistema debe hacer, sino también cómo va éste a desarrollar dichas funcionalidades. De esta manera se podrá realizar una clasificación de los requisitos en dos grandes bloques:

- Requisitos de capacidad: son aquellos que representan lo que el sistema debe hacer, de acuerdo a las necesidades indicadas. Se trata de requisitos que hacen referencia al problema en cuestión que se desea resolver.
- Requisitos de restricción: indican la manera que tendrá el sistema de resolver los problemas, la forma de interactuar con el sistema y las restricciones que habrá sobre éste.

A continuación se mostrará el formato que se va a emplear para plasmar los diversos requisitos extraídos, indicando el significado de cada uno de sus campos.

RUT-NN			
Prioridad		Necesidad	
Verificabilidad		Claridad	
Estabilidad			
Descripción			

Tabla 1: Tabla de definición de requisitos de usuario

Donde:

- RU: iniciales que denotan que se trata de un requisito de usuario.
- T: tipo de requisito de usuario. Existen dos: C, si es un requisito de capacidad y R si es un requisito de restricción.
- NN: identificador numérico y secuencial de los requisitos de cada categoría. Esto hace que se pueda diferenciar unívocamente cada requisito.
- Prioridad: indica en que fase del sistema se incluirá la funcionalidad especificada por el requisito. Valores: Alta, Media, Baja.

- Necesidad: Indica el interés de los participantes en que dicho requisito se cumpla. Valores: Esencial, Deseable, Opcional.
- Claridad: indica si un requisito es fácil de entender, o por el contrario su definición resulta ambigua. Valores: Alta, Media, Baja.
- Verificabilidad: Indica la facilidad de verificación que tiene el requisito una vez se ha diseñado el sistema. Valores: Alta, Media, Baja.
- Estabilidad: Indica el grado de variabilidad que puede tener un requisito a lo largo del proyecto.
- Descripción: como su propio nombre indica, describe el requisito.

3.2.1 Requisitos de capacidad

RUC-1			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Al iniciar, la aplicación mostrará el menú principal del juego.		

Tabla 2: Requisito de Capacidad RUC-1

RUC-2			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	Puede sufrir modificaciones durante el proyecto.		
Descripción	El menú principal dará acceso a: • Nueva partida. • Editor de niveles. • Opciones. • Salir del juego.		

Tabla 3: Requisito de Capacidad RUC-2

RUC-3			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	La música y el idioma serán parámetros configurables.		

Tabla 4: Requisito de Capacidad RUC-3

RUC-4			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Se permitirá elegir la nave antes de comenzar la partida.		

Tabla 5: Requisito de Capacidad RUC-4

RUC-5			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Se permitirá controlar el juego con el ratón.		

Tabla 6: Requisito de Capacidad RUC-5

RUC-6			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Se permitirá controlar el juego con el teclado.		

Tabla 7: Requisito de Capacidad RUC-6

RUC-7			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	El usuario podrá elegir el tipo de control que desea utilizar.		

Tabla 8: Requisito de Capacidad RUC-7

RUC-8			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	El usuario controlará el movimiento de la nave		

Tabla 9: Requisito de Capacidad RUC-8

RUC-9			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Existirá un botón que permita disparar.		

Tabla 10: Requisito de Capacidad RUC-9

RUC-10			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Una fase se compone de oleadas de enemigos y, opcionalmente, un jefe final.		

Tabla 11: Requisito de Capacidad RUC-10

RUC-11			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	El *scroll* avanzará durante las oleadas de enemigos.		

Tabla 12: Requisito de Capacidad RUC-11

RUC-12			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	El *scroll* se detendrá al llegar al jefe final.		

Tabla 13: Requisito de Capacidad RUC-12

RUC-13			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Se mostrará en pantalla la información del estado actual de la nave.		

Tabla 14: Requisito de Capacidad RUC-13

RUC-14			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Existirán dos pantallas de final de fase: • Fin del juego. • Misión cumplida.		

Tabla 15: Requisito de Capacidad RUC-14

RUC-15			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Existirá un editor que permitirá crear y configurar nuevas pantallas.		

Tabla 16: Requisito de Capacidad RUC-15

RUC-16			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Existirá un editor simple que permitirá generar una pantalla en base a: • Duración. • *Scroll*. • Dificultad.		

Tabla 17: Requisito de Capacidad RUC-16

RUC-17			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Habrá música de fondo en todo momento, ésta dependerá de la pantalla en la que esté actualmente el usuario.		

Tabla 18: Requisito de Capacidad RUC-17

RUC-18			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	No sufrirá modificaciones durante el proyecto		
Descripción	Habrá efectos de sonido para diversos eventos que se produzcan en el transcurso del juego.		

Tabla 19: Requisito de Capacidad RUC-18

3.2.2 Requisitos de restricción

RUR-1			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La interfaz de la aplicación tendrá un diseño sencillo e intuitivo.		

Tabla 20: Requisito de restricción RUR-1

RUR-2			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Existirá un manual de usuario.		

Tabla 21: Requisito de restricción RUR-2

RUR-3			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La aplicación funcionará correctamente en los sistemas operativos Windows XP/Vista/7.		

Tabla 22: Requisito de restricción RUR-3

RUR-4			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El idioma de la aplicación será, por defecto, el español.		

Tabla 23: Requisito de restricción RUR-4

RUR-5			
Prioridad	Alta	Necesidad	Deseable
Verificabilidad	Alta	Claridad	Alta
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Además del español, la aplicación contará con el idioma inglés.		

Tabla 24: Requisito de restricción RUR-5

RUR-6			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El tiempo de carga de cualquier pantalla será inferior a 5 segundos.		

Tabla 25: Requisito de restricción RUR-6

RUR-7			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El tiempo requerido por el editor para guardar un nivel será inferior a 2 segundos.		

Tabla 26: Requisito de restricción RUR-7

RUR-8			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La resolución de la aplicación será de 800x600.		

Tabla 27: Requisito de restricción RUR-8

RUR-9			
Prioridad	Alta	Necesidad	Esencial
Verificabilidad	Alta	Claridad	Alta
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Al cargar un archivo que la aplicación reconozca como una pantalla personalizada no deberán producirse errores, incluso aunque el archivo no se corresponda con una fase creada con el editor de niveles.		

Tabla 28: Requisito de restricción RUR-9

3.3 Requisitos de software

En este punto del documento se pretenden identificar, clasificar y catalogar los distintos requisitos de software obtenidos tanto de los requisitos de usuario, como de los casos de uso especificados en el siguiente apartado de este mismo documento.

Los requisitos deben plasmar, no sólo lo que el sistema va a ser capaz de realizar, sino también como va a hacerlo, las restricciones que tendrá y sus propiedades. De esta manera, tenemos dos grandes grupos en los que catalogar los requisitos de software:

- Requisitos funcionales: especifican que es lo que tiene que realizar el sistema.
- Requisitos no funcionales: especifican como hace las cosas el sistema.

A continuación se mostrará el formato que se va a emplear para plasmar los diversos requisitos extraídos, indicando el significado de cada uno de sus campos.

RST-NN			
Prioridad		Fuente	
Verificabilidad		Claridad	
Necesidad			
Estabilidad			
Descripción			

Tabla 29: Tabla de definición de requisitos de software

Donde:

- RS: iniciales que denotan que se trata de un requisito de software.
- T: tipo de requisito de usuario. Existen dos: F, si es un requisito de software funcional y NF si es un requisito de software no funcional.
- NN: identificador numérico y secuencial de los requisitos de cada categoría. Esto hace que se pueda diferenciar unívocamente cada requisito.
- Prioridad: indica en que fase del sistema se incluirá la funcionalidad especificada por el requisito. Valores: Alta, Media, Baja.
- Fuente: indica el requisito de usuario del que se ha obtenido.
- Necesidad: Indica el interés de los participantes en que dicho requisito se cumpla. Valores: Esencial, Deseable, Opcional.
- Claridad: indica si un requisito es fácil de entender, o por el contrario su definición resulta ambigua. Valores: Alta, Media, Baja.
- Verificabilidad: Indica la facilidad de verificación que tiene el requisito una vez se ha diseñado el sistema. Valores: Alta, Media, Baja.
- Estabilidad: Indica el grado de variabilidad que puede tener un requisito a lo largo del proyecto.
- Descripción: como su propio nombre indica, describe el requisito.

Además de la clasificación de los requisitos en funcionales y no funcionales, existen clasificaciones dentro de cada una. Los requisitos de software funcionales se dividirán en categorías según la funcionalidad. Para los requisitos de software no funcionales se tienen los siguientes tipos:

- Interfaz: son aquellos requisitos que hacen referencia a la interfaz del sistema a través de la cual el usuario deberá interactuar.
- Documentación: aquellos requisitos que estén relacionados con temas de ayudas y manuales de usuario.
- Portabilidad: son los requisitos que aporten al sistema la flexibilidad necesaria para que ésta sea compatible con diversos entornos de ejecución.
- Operacionales: son aquellos que hacen referencia al entorno de ejecución donde debe funcionar el sistema.
- Rendimiento: requisitos que garantizan el buen funcionamiento de la aplicación.
- Seguridad ante Fallos: requisitos que permiten a la aplicación recuperarse ante fallos u operaciones indebidas realizadas por los usuarios.

Antes de incluir los requisitos se hace necesario destacar que no se han encontrado requisitos de seguridad frente a amenazas externas, de ahí que no esté contemplada dicha clasificación.

3.3.1 Requisitos funcionales

Requisitos relacionados con el menú:

RSF-1			
Prioridad	Alta	Fuente	RUC-1
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Al iniciar la aplicación, se mostrará el menú principal del juego.		

Tabla 30: Requisito de software funcional RSF-1

RSF-2			
Prioridad	Alta	Fuente	RUC-2
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El menú principal dará acceso a: • Nueva partida. • Editor de niveles. • Elegir nave. • Ayuda y opciones. • Salir del juego.		

Tabla 31: Requisito de software funcional RSF-2

RSF-3			
Prioridad	Alta	Fuente	RUC-2
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El menú de ayuda y opciones dará acceso a: • Manual. • Control. • Configuración.		

Tabla 32: Requisito de software funcional RSF-3

RSF-4			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El apartado de manual contendrá una guía rápida del juego.		

Tabla 33: Requisito de software funcional RSF-4

RSF-5			
Prioridad	Alta	Fuente	RUC-7
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El apartado de control mostrará los controles y permitirá alternar entre teclado y ratón.		

Tabla 34: Requisito de software funcional RSF-5

RSF-6			
Prioridad	Alta	Fuente	RUC-3
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El menú de configuración permitirá modificar: • Música. • Efectos de sonido. • Idioma.		

Tabla 35: Requisito de software funcional RSF-6

RSF-7			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El menú de configuración permitirá restaurar los valores por defecto.		

Tabla 36: Requisito de software funcional RSF-7

Requisitos relacionados con el control del juego:

RSF-8			
Prioridad	Alta	Fuente	RUC-5
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Se permitirá controlar el juego utilizando el ratón.		

Tabla 37: Requisito de software funcional RSF-8

RSF-9			
Prioridad	Alta	Fuente	RUC-6
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Se permitirá controlar el juego utilizando el teclado.		

Tabla 38: Requisito de software funcional RSF-9

RSF-10			
Prioridad	Alta	Fuente	RUC-7
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El usuario podrá alternar entre ambos tipos de control.		

Tabla 39: Requisito de software funcional RSF-10

RSF-11			
Prioridad	Alta	Fuente	RUC-8
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El usuario controlará el movimiento de la nave.		

Tabla 40: Requisito de software funcional RSF-11

RSF-12			
Prioridad	Alta	Fuente	RUC-9
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Existirá un botón que permita disparar.		

Tabla 41: Requisito de software funcional RSF-12

RSF-13			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La nave podrá disparar un número limitado de bombas.		

Tabla 42: Requisito de software funcional RSF-13

RSF-14			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Existirá un botón que permita lanzar bombas.		

Tabla 43: Requisito de software funcional RSF-14

Requisitos relacionados con la pantalla de juego:

RSF-15			
Prioridad	Alta	Fuente	RUC-10
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Una fase se compone de oleadas de enemigos y, opcionalmente, un jefe final.		

Tabla 44: Requisito de software funcional RSF-15

RSF-16			
Prioridad	Alta	Fuente	RUC-10
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Los enemigos tendrán distintos *sprites* y comportamientos.		

Tabla 45: Requisito de software funcional RSF-16

RSF-17			
Prioridad	Alta	Fuente	RUC-10
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El jefe final no aparecerá hasta que no hayan desaparecido todos los demás enemigos de la pantalla.		

Tabla 46: Requisito de software funcional RSF-17

RSF-18			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Los enemigos pueden dejar caer *power ups* al ser destruidos.		

Tabla 47: Requisito de software funcional RSF-18

RSF-19			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Los posibles *power ups* son: • Potencia de disparo. • Recuperar vida. • Bombas adicionales. • Puntos adicionales.		

Tabla 48: Requisito de software funcional RSF-19

RSF-20			
Prioridad	Alta	Fuente	RUC-11
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El *scroll* de fondo avanzará de arriba a abajo durante las oleadas de enemigos.		

Tabla 49: Requisito de software funcional RSF-20

RSF-21			
Prioridad	Alta	Fuente	RUC-12
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El *scroll* se detendrá cuando aparezca el jefe final.		

Tabla 50: Requisito de software funcional RSF-21

RSF-22			
Prioridad	Alta	Fuente	RUC-13
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Durante la pantalla de juego se mostrará la siguiente información: • Vida restante. • Bombas restantes. • Puntuación actual.		

Tabla 51: Requisito de software funcional RSF-22

RSF-23			
Prioridad	Alta	Fuente	RUC-14
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Existirán dos pantallas de final de fase: • Fin del juego. • Misión cumplida.		

Tabla 52: Requisito de software funcional RSF-23

RSF-24			
Prioridad	Alta	Fuente	RUC-14
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La pantalla de fin del juego mostrará la puntuación obtenida, y permitirá: • Reiniciar la fase. • Salir al menú.		

Tabla 53: Requisito de software funcional RSF-24

RSF-25			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La pantalla de misión cumplida mostrará la puntuación acumulada, y permitirá: • Avanzar a la siguiente fase (si la hay). • Salir al menú.		

Tabla 54: Requisito de software funcional RSF-25

Requisitos relacionados con el editor de niveles:

RSF-26			
Prioridad	Alta	Fuente	RUC-15
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Habrá un editor avanzado que permitirá crear y configurar nuevas fases.		

Tabla 55: Requisito de software funcional RSF-26

RSF-27			
Prioridad	Alta	Fuente	RUC-15
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El editor avanzado requerirá los siguientes parámetros antes de permitir configurar los enemigos: • Duración. • *Scroll*. • Nombre de archivo.		

Tabla 56: Requisito de software funcional RSF-27

RSF-28			
Prioridad	Alta	Fuente	RUC-15
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El editor avanzado dará la opción de incluir, o no, un jefe final.		

Tabla 57: Requisito de software funcional RSF-28

RSF-29			
Prioridad	Alta	Fuente	RUC-15
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Para cada enemigo se puede configurar: • *Sprite*. • Comportamiento. • *Power up*.		

Tabla 58: Requisito de software funcional RSF-29

RSF-30			
Prioridad	Alta	Fuente	RUC-16
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Habrá un editor simple que permitirá crear una nueva pantalla de forma aleatoria.		

Tabla 59: Requisito de software funcional RSF-30

RSF-31			
Prioridad	Alta	Fuente	RUC-17
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El editor simple requerirá los siguientes parámetros antes de poder crear una nueva pantalla: • Duración. • *Scroll*. • Dificultad. • Nombre de archivo.		

Tabla 60: Requisito de software funcional RSF-31

Requisitos relacionados con el sonido:

RSF-32			
Prioridad	Alta	Fuente	RUC-17
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Habrá una melodía para cada una de las siguientes pantallas: • Menú principal. • Pantalla de juego durante las oleadas de enemigos. • Pantalla de juego durante el jefe final. • Fin del juego. • Misión cumplida.		

Tabla 61: Requisito de software funcional RSF-32

RSF-33			
Prioridad	Alta	Fuente	RUC-17
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La música cambiará automáticamente al pasar de una pantalla a otra.		

Tabla 62: Requisito de software funcional RSF-33

RSF-34			
Prioridad	Alta	Fuente	RUC-18
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Los efectos de sonido que se podrán escuchar serán: • Explosión. • Bomba. • Nave alcanzada por un disparo.		

Tabla 63: Requisito de software funcional RSF-34

3.3.2 Requisitos no funcionales

Requisitos de interfaz:

RSNF-1			
Prioridad	Alta	Fuente	RUR-1
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La interfaz se basará en dos aspectos importantes: • Fácil de aprender. • Fácil de usar.		

Tabla 64: Requisito de software no funcional RSNF-1

RSNF-2			
Prioridad	Alta	Fuente	RUR-1
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La interfaz presentará la información de forma legible.		

Tabla 65: Requisito de software no funcional RSNF-2

Requisitos de documentación:

RSNF-3			
Prioridad	Alta	Fuente	RUR-2
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Existirá una guía rápida del juego dentro de la aplicación.		

Tabla 66: Requisito de software no funcional RSNF-3

RSNF-4			
Prioridad	Alta	Fuente	RUR-2
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Existirá un manual de usuario, impreso, a disposición del jugador.		

Tabla 67: Requisito de software no funcional RSNF-4

Requisitos de portabilidad:

RSNF-5			
Prioridad	Alta	Fuente	RUR-3
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La aplicación funcionará correctamente en los sistemas operativos Windows XP/Vista/7.		

Tabla 68: Requisito de software no funcional RSNF-5

RSNF-6			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Si el ordenador donde se realiza la instalación no dispone de XNA Game Studio 3.1, el instalador incluirá las librerías necesarias para una correcta ejecución.		

Tabla 69: Requisito de software no funcional RSNF-6

Requisitos operacionales:

RSNF-7			
Prioridad	Alta	Fuente	RUR-4
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El idioma por defecto de la aplicación será el español.		

Tabla 70: Requisito de software no funcional RSNF-7

RSNF-8			
Prioridad	Alta	Fuente	RUR-5
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Además del español, la aplicación incluirá el idioma inglés.		

Tabla 71: Requisito de software no funcional RSNF-8

RSNF-9			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Se podrá salir del juego de la siguiente manera: • Desde la opción “Salir del juego” del menú principal. • Pulsando la tecla Esc en cualquier momento.		

Tabla 72: Requisito de software no funcional RSNF-9

RSNF-10			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La aplicación creará un archivo donde guardará la última configuración establecida.		

Tabla 73: Requisito de software no funcional RSNF-10

RSNF-11			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La aplicación reconocerá como fases personalizadas los archivos cuya extensión sea ".sf".		

Tabla 74: Requisito de software no funcional RSNF-11

Requisitos de rendimiento:

RSNF-12			
Prioridad	Alta	Fuente	RUR-6
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El tiempo de carga del juego al iniciar será inferior a 5 segundos.		

Tabla 75: Requisito de software no funcional RSNF-12

RSNF-13			
Prioridad	Alta	Fuente	RUR-6
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El tiempo de carga del menú principal y cualquiera de sus sub-menús será inferior a 1 segundo.		

Tabla 76: Requisito de software no funcional RSNF-13

RSNF-14			
Prioridad	Alta	Fuente	RUR-6
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El tiempo de carga de cualquier fase será inferior a 3 segundos.		

Tabla 77: Requisito de software no funcional RSNF-14

RSNF-15			
Prioridad	Alta	Fuente	RUR-6
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El tiempo de carga de las pantallas de final de fase será inferior a 3 segundos.		

Tabla 78: Requisito de software no funcional RSNF-15

RSNF-16			
Prioridad	Alta	Fuente	RUR-6
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El tiempo de carga del editor será inferior a 3 segundos.		

Tabla 79: Requisito de software no funcional RSNF-16

RSNF-17			
Prioridad	Alta	Fuente	RUR-7
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	El tiempo requerido para guardar el archivo del nivel personalizado será inferior a 2 segundos.		

Tabla 80: Requisito de software no funcional RSNF-17

RSNF-18			
Prioridad	Alta	Fuente	RUR-8
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	La resolución de la aplicación será de 800x600.		

Tabla 81: Requisito de software no funcional RSNF-18

Requisitos de seguridad frente a fallos:

RSNF-19			
Prioridad	Alta	Fuente	RUR-9
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Si la aplicación reconoce un archivo con extensión ".sf" que no se trata de una fase personalizada, cargará otra fase por defecto, en lugar de dar error.		

Tabla 82: Requisito de software no funcional RSNF-19

RSNF-20			
Prioridad	Alta	Fuente	-
Verificabilidad	Alta	Claridad	Alta
Necesidad	Esencial		
Estabilidad	El requisito no sufrirá modificaciones a lo largo del proyecto.		
Descripción	Si se introduce en el editor un nombre de archivo que sobrescribe un archivo existente, se avisará de ello.		

Tabla 83: Requisito de software no funcional RSNF-20

3.4 Casos de uso

3.4.1 Diagrama de casos de uso

Mediante el modelado de casos de uso se consigue identificar más funcionalidades del videojuego. En los casos de uso se especifican las respuestas del sistema a la interacción de agentes externos. Así, dada la naturaleza de un videojuego y sus funcionalidades, surgirán casos de uso que pueden parecer propios del sistema. Sin embargo se trata de interacciones, voluntarias o involuntarias, del jugador con el sistema. Un ejemplo de interacción involuntaria es la colisión de la nave con un enemigo.

Para el sistema que se está desarrollando, solamente se identifica un actor:

- Jugador.

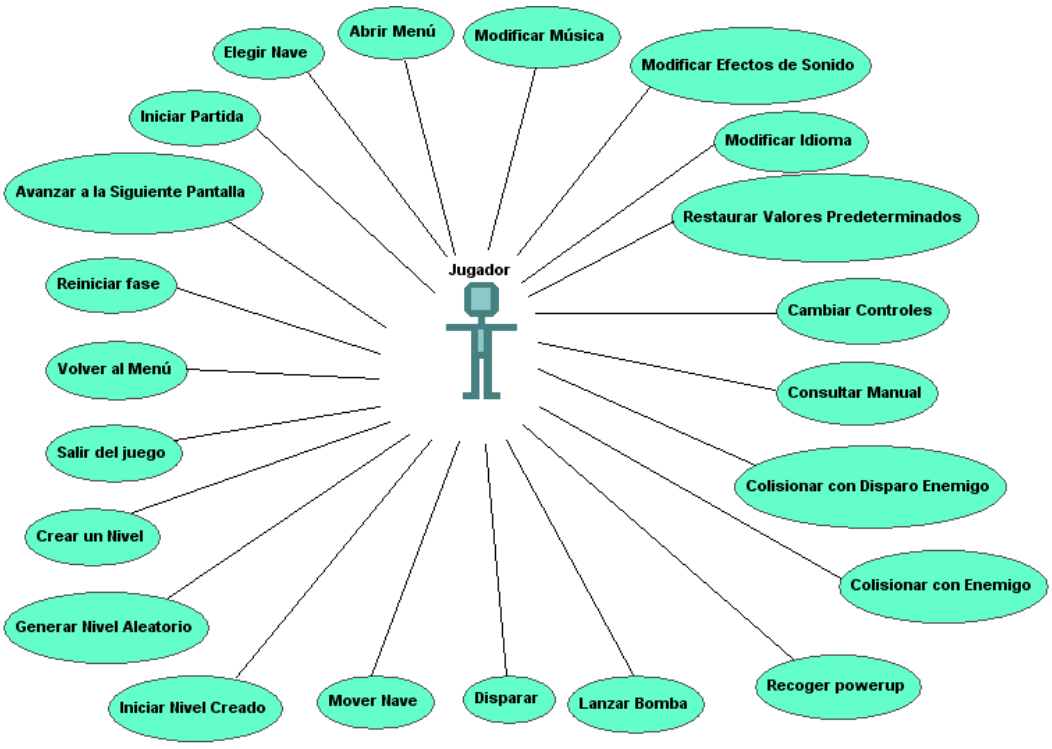


Figura 22. Casos de uso de Jugador

3.4.2 Especificación textual de los casos de uso

En este apartado se incluyen las tablas con la especificación textual de cada uno de los casos de uso identificados.

En primer lugar, se va a mostrar el formato que se va a emplear para representar los casos de uso, indicando el significado de cada uno de sus campos:

Nombre	CU X -
Actor	
Objetivo	
Precondiciones	
Postcondiciones	
Escenario básico	

Tabla 84: Tabla de definición de casos de uso

Donde:

- CU: iniciales que denotan que se trata de un caso de uso.
- X: identificador numérico y secuencial de los casos de uso. Esto hace que se pueda diferenciar unívocamente cada uno de ellos.
- Nombre: identificador textual del caso de uso.
- Objetivo: define la respuesta del sistema que se espera obtener ejecutando el caso de uso.
- Pre-condiciones: condiciones que deben cumplirse para poder ejecutar el caso de uso.
- Post-condiciones: condiciones que se darán en el sistema una vez haya finalizado la ejecución del caso de uso.
- Escenario básico: pasos necesarios para ejecutar el caso de uso.

Nombre	CU 1 – Abrir menú
Actor	Jugador.
Objetivo	Mostrar el menú al que desea acceder el usuario.
Precondiciones	Situarse en un menú que dé acceso al que se quiere acceder.
Postcondiciones	El jugador accede al menú seleccionado.
Escenario básico	1. El jugador está visualizando un menú que da acceso al que quiere llegar. 2. El jugador selecciona el menú al que quiere acceder.

Tabla 85: Caso de uso 1 – Abrir menú

Nombre	CU 2 – Iniciar partida
Actor	Jugador.
Objetivo	Iniciar una nueva partida con los niveles predefinidos.
Precondiciones	Estar en el menú principal.
Postcondiciones	Comienza la primera fase.
Escenario básico	1. El jugador está visualizando el menú principal. 2. El jugador selecciona la opción “Nueva Partida”.

Tabla 86: Caso de uso 2 - Iniciar partida

Nombre	CU 3 – Iniciar nivel creado
Actor	Jugador.
Objetivo	Iniciar una nueva partida en uno de los niveles creados por el usuario.
Precondiciones	• Estar en el menú “Niveles creados”. • Existe algún nivel creado.
Postcondiciones	Comienza la fase correspondiente al nivel creado.
Escenario básico	1. El jugador está visualizando el menú “Niveles creados”. 2. El jugador selecciona el nivel que desea iniciar. 3. El jugador selecciona la opción “Jugar”.

Tabla 87: Caso de uso 3 - Iniciar nivel creado

Nombre	CU 4 – Crear un nivel
Actor	Jugador.
Objetivo	Crear un nivel configurando el *scroll*, la duración y los enemigos.
Precondiciones	Estar en el menú “Editor de niveles”.
Postcondiciones	El nivel ha sido creado.
Escenario básico	1. El jugador está visualizando el menú “Editor de niveles”. 2. El jugador selecciona la opción “Editor avanzado”. 3. El jugador introduce los siguientes datos: 3.1. Duración. 3.2. *Scroll*. 3.3. Nombre del archivo. 4. El jugador pulsa el botón “Comenzar”. 5. El jugador configura el jefe final. 6. El jugador configura cada uno de los enemigos que aparecerán en la fase. 7. El jugador pulsa el botón “Guardar y salir”.

Tabla 88: Caso de uso 4 - Crear un nivel

Nombre	CU 5 – Generar un nivel aleatorio
Actor	Jugador.
Objetivo	Generar una nueva fase de forma aleatoria en base a unos parámetros.
Precondiciones	Estar en el menú “Editor de niveles”.
Postcondiciones	El nivel ha sido creado.
Escenario básico	1. El jugador está visualizando el menú “Editor de niveles”. 2. El jugador selecciona la opción “Editor simple”. 3. El jugador introduce los siguientes datos: 3.1. Duración. 3.2. *Scroll*. 3.3. Dificultad. 3.4. Nombre del archivo. 4. El jugador pulsa el botón “Guardar”.

Tabla 89: Caso de uso 5 - Generar un nivel aleatorio

Nombre	CU 6 – Elegir nave
Actor	Jugador.
Objetivo	Elegir la nave con la que se desea jugar.
Precondiciones	Estar en el menú “Elegir nave”.
Postcondiciones	La nave elegida se utilizará en la próxima partida.
Escenario básico	1. El jugador está visualizando el menú “Elegir nave”. 2. El jugador selecciona la nave que desea controlar. 3. El jugador pulsa la opción “Volver”.

Tabla 90: Caso de uso 6 - Elegir nave

Nombre	CU 7 – Consultar manual
Actor	Jugador.
Objetivo	Consultar la guía rápida del juego.
Precondiciones	Estar en el menú “Ayuda y opciones”.
Postcondiciones	Ninguna.
Escenario básico	1. El jugador está visualizando el menú “Ayuda y opciones”. 2. El jugador elige la opción “Manual”. 3. El jugador consulta las páginas del manual. 4. El jugador pulsa la opción “Volver”.

Tabla 91: Caso de uso 7 - Consultar manual

Nombre	CU 8 – Cambiar controles
Actor	Jugador.
Objetivo	Cambiar el método de control del juego.
Precondiciones	Estar en el menú “Controles”.
Postcondiciones	El método de control seleccionado está disponible.
Escenario básico	1. El jugador está visualizando el menú “Controles”. 2. El jugador cambia el método de control actual.

Tabla 92: Caso de uso 8 - Cambiar controles

Nombre	CU 9 – Modificar música
Actor	Jugador.
Objetivo	Ajustar el volumen de la música de fondo.
Precondiciones	Estar en el menú “Configuración”.
Postcondiciones	El volumen de la música de fondo se ajusta al valor seleccionado.
Escenario básico	1. El jugador está visualizando el menú “Configuración”. 2. El jugador aumenta o disminuye el volumen de la música.

Tabla 93: Caso de uso 9 - Modificar música

Nombre	CU 10 – Modificar efectos de sonido
Actor	Jugador.
Objetivo	Activar o desactivar los efectos de sonido.
Precondiciones	Estar en el menú “Configuración”.
Postcondiciones	Los efectos quedarán activados o desactivados.
Escenario básico	1. El jugador está visualizando el menú “Configuración”. 2. El jugador activa o desactiva los efectos de sonido.

Tabla 94: Caso de uso 10 - Modificar efectos de sonido

Nombre	CU 11 – Modificar idioma
Actor	Jugador.
Objetivo	Cambiar el idioma actual.
Precondiciones	Estar en el menú “Configuración”.
Postcondiciones	Se muestra el juego en el idioma seleccionado.
Escenario básico	1. El jugador está visualizando el menú “Configuración”. 2. El jugador cambia el idioma actual.

Tabla 95: Caso de uso 11 - Modificar idioma

Nombre	CU 12 – Restaurar valores predeterminados
Actor	Jugador.
Objetivo	Restaurar la configuración a los valores por defecto.
Precondiciones	Estar en el menú “Configuración”.
Postcondiciones	Los valores de música, efectos de sonido e idioma se ajustan a los valores por defecto.
Escenario básico	1. El jugador está visualizando el menú “Configuración”. 2. El jugador elige la opción “Predeterminado”.

Tabla 96: Caso de uso 12 - Restaurar valores predeterminados

Nombre	CU 13 – Salir del juego
Actor	Jugador.
Objetivo	Salir del juego y cerrar la aplicación.
Precondiciones	Estar en el menú principal.
Postcondiciones	La aplicación está cerrada.
Escenario básico	1. El jugador está visualizando el menú principal. 2. El jugador elige la opción “Salir del juego”.

Tabla 97: Caso de uso 13 - Salir del juego

Nombre	CU 14 – Avanzar a la siguiente pantalla
Actor	Jugador.
Objetivo	Avanzar a la fase 2.
Precondiciones	Haber completado con éxito la fase 1.
Postcondiciones	Se inicia la fase 2.
Escenario básico	1. El jugador está visualizando la pantalla de “Misión cumplida”. 2. El jugador elige la opción “Continuar”.

Tabla 98: Caso de uso 14 - Avanzar a la siguiente pantalla

Nombre	CU 15 – Reiniciar fase
Actor	Jugador.
Objetivo	Reiniciar la fase actual.
Precondiciones	Estar en la pantalla “Fin del juego”.
Postcondiciones	Se reinicia la fase no superada.
Escenario básico	1. El jugador está visualizando la pantalla “Fin del juego”. 2. El jugador elige la opción “Continuar”.

Tabla 99: Caso de uso 15 - Reiniciar fase

Nombre	CU 16 – Volver al menú
Actor	Jugador.
Objetivo	Regresar al menú principal después de jugar una fase.
Precondiciones	Haber terminado una fase superándola o muriendo.
Postcondiciones	El juego muestra el menú principal.
Escenario básico	1. El jugador está visualizando una pantalla de fin de fase. 2. El jugador elige la opción “Salir”.

Tabla 100: Caso de uso 16 - Volver al menú

Nombre	CU 17 – Mover nave
Actor	Jugador.
Objetivo	Desplazar la nave a lo largo y ancho de la pantalla.
Precondiciones	Haber iniciado una fase.
Postcondiciones	La nave se desplaza en el eje de coordenadas especificado.
Escenario básico	1. El jugador está jugando una fase. 2. El jugador desplaza la nave.

Tabla 101: Caso de uso 17 - Mover nave

Nombre	CU 18 – Disparar
Actor	Jugador.
Objetivo	Hacer que la nave dispare.
Precondiciones	Haber iniciado una fase.
Postcondiciones	La nave lanza un disparo.
Escenario básico	1. El jugador está jugando una fase. 2. El jugador pulsa el botón de disparo.

Tabla 102: Caso de uso 18 - Disparar

Nombre	CU 19 – Lanzar bomba
Actor	Jugador.
Objetivo	Hacer que la nave lance una bomba.
Precondiciones	• Haber iniciado una fase. • El número de bombas disponibles es mayor o igual a uno.
Postcondiciones	La nave lanza una bomba.
Escenario básico	1. El jugador está jugando una fase. 2. El jugador pulsa el botón de lanzar bomba.

Tabla 103: Caso de uso 19 - Lanzar bomba

Nombre	CU 20 – Recoger *power up*
Actor	Jugador.
Objetivo	Hacer que la nave recoja un *power up*.
Precondiciones	• Haber iniciado una fase. • Hay un *power up* flotando por la fase.
Postcondiciones	La nave adquiere una mejora.
Escenario básico	1. El jugador está jugando una fase. 2. La nave colisiona con el *power up*.

Tabla 104: Caso de uso 20 - Recoger *power up*

Nombre	CU 21 – Colisionar con un enemigo
Actor	Jugador.
Objetivo	La nave colisiona con una nave enemiga.
Precondiciones	• Haber iniciado una fase. • Hay al menos un enemigo en pantalla.
Postcondiciones	La nave sufre daños.
Escenario básico	1. El jugador está jugando una fase. 2. El jugador colisiona con el enemigo.

Tabla 105: Caso de uso 21 - Colisionar con un enemigo

Nombre	CU 22 – Colisionar con un disparo enemigo
Actor	Jugador.
Objetivo	La nave colisiona con un disparo realizado por un enemigo.
Precondiciones	• Haber iniciado una fase. • Hay al menos un disparo enemigo en pantalla.
Postcondiciones	La nave sufre daños.
Escenario básico	1. El jugador está jugando una fase. 2. El jugador colisiona con un disparo enemigo.

Tabla 106: Caso de uso 22 - Colisionar con un disparo enemigo

3.4.3 Diagramas de secuencia

A continuación se detallan los diagramas de secuencia de los casos de uso. En estos diagramas se muestran las interacciones que se producen entre los objetos durante la realización de los casos de uso.

Debido a la similitud, o a la poca información que aportarían algunos de los diagramas, se ha optado por hacer una selección de los casos de uso más importantes para representar sus diagramas de secuencia.

Iniciar partida:

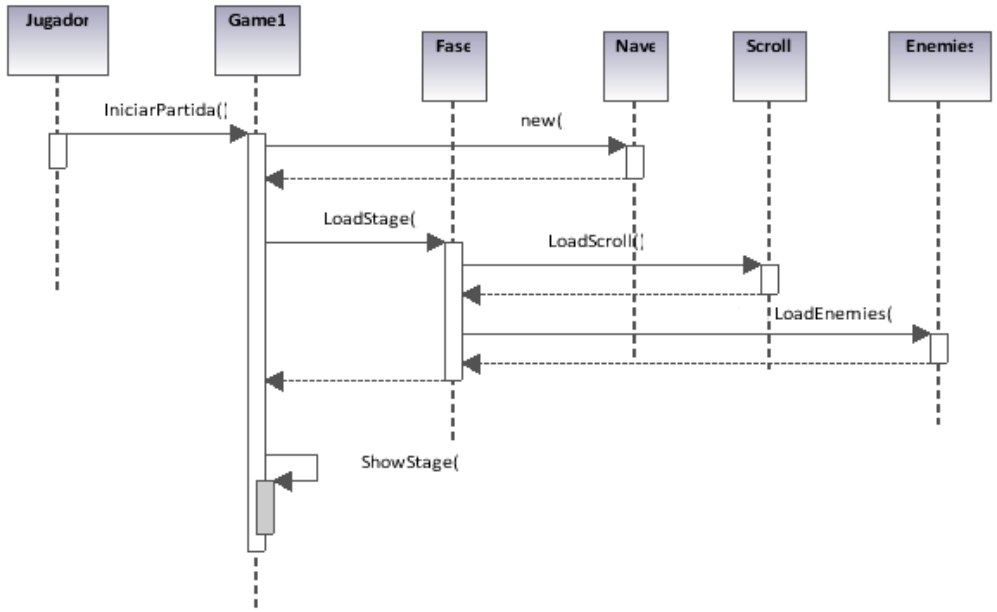


Figura 23. Diagrama de Secuencia de CU 2 - Iniciar partida

El diagrama de la figura 23 muestra el caso de uso correspondiente a iniciar partida. En el momento en el que el usuario decide iniciar una nueva partida, el juego debe cargar los elementos necesarios antes de mostrar la pantalla de juego. En primer lugar inicializará la nave, que es el elemento que controla el jugador, y posteriormente cargará la fase que contiene el *scroll* de fondo y las distintas oleadas de enemigos que irán apareciendo a lo largo de la pantalla.

Algo similar ocurriría a la hora de iniciar un nivel creado. La diferencia está en que el archivo al que se accede para cargar la fase sería distinto.

Avanzar de una fase a otra, o reiniciar una fase que no se ha completado tendrán procesos similares por lo que el diagrama de secuencia de dichos casos de uso no aporta información adicional.

Generar nivel aleatorio:

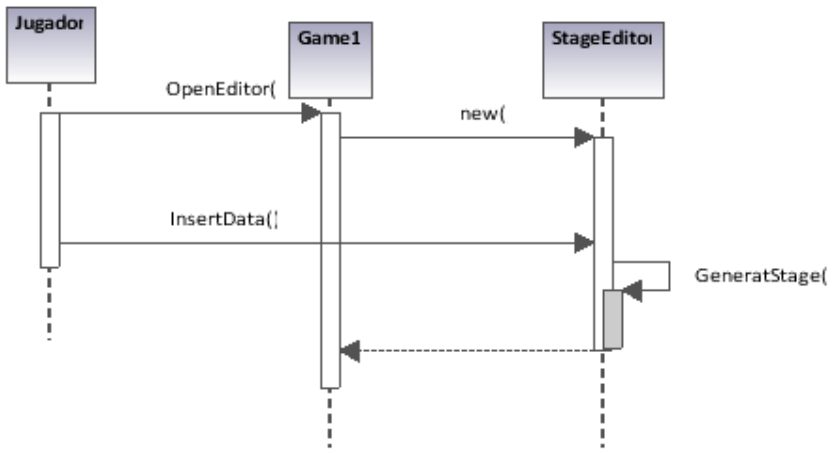


Figura 24. Diagrama de Secuencia de CU 5 – Generar nivel aleatorio

Cuando se desea generar un nivel aleatorio, lo que hace el juego es cargar el editor de niveles correspondiente. El usuario introduce los valores necesarios, y el editor genera el nivel. Posteriormente, informa al juego y se cierra.

La diferencia con la generación de un nivel configurando cada uno de sus parámetros, radica en que el jugador debería introducir además la configuración del jefe final y de cada uno de los enemigos de la pantalla.

Recoger *power up*:

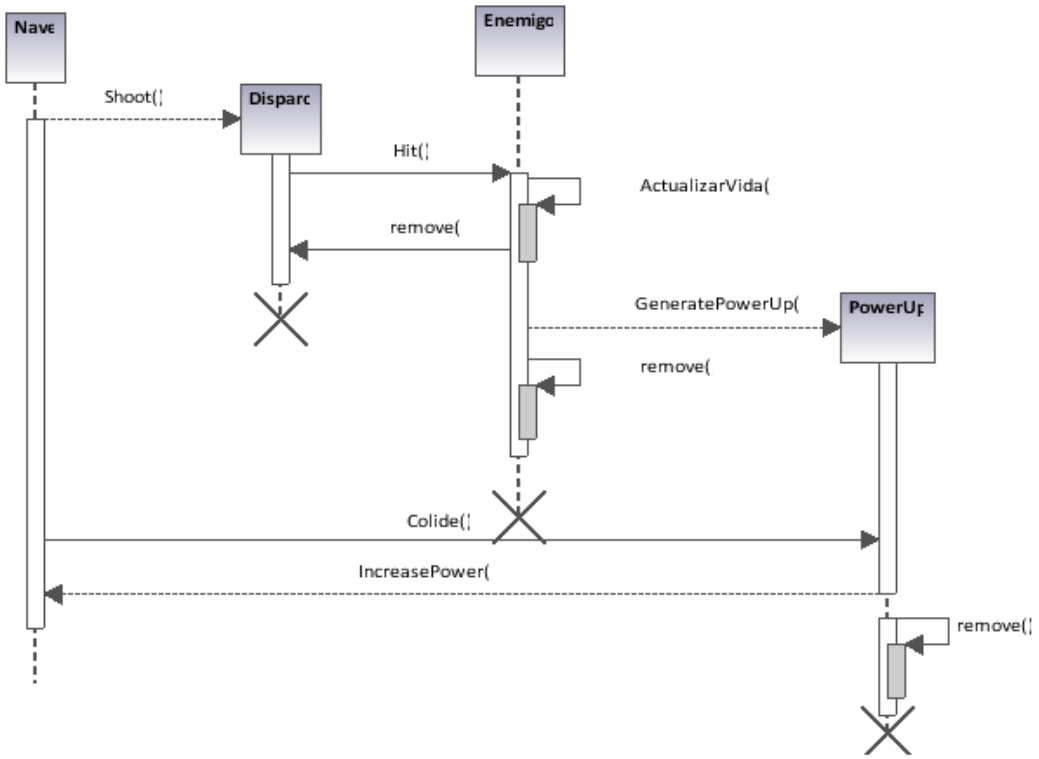


Figura 25. Diagrama de Secuencia de CU 20 – Recoger *power up*

Este diagrama muestra el proceso completo desde que la nave dispara, hasta que se recoge el *power up*. Se puede apreciar cómo es la nave la que genera el disparo. Posteriormente éste impacta con el enemigo, destruyéndose a su vez, quién reduce su vida por debajo de cero, quedando eliminado. Antes de eliminar el objeto enemigo, éste genera el *power up* correspondiente que aumentará los atributos de la nave cuando ésta lo recoja.

Gracias al diagrama de la figura 25 se puede apreciar cómo sería el proceso de disparo, de lanzamiento de bombas y de las distintas colisiones que se producen entre los elementos del juego a lo largo de una fase.

3.5 Diagrama de actividad del sistema

Un diagrama de actividad muestra los distintos estados de un sistema, así como las diferentes transiciones y eventos que provocan que el sistema cambie de un estado a otro.

A continuación se muestra el diagrama de actividad para el videojuego que se está desarrollando.

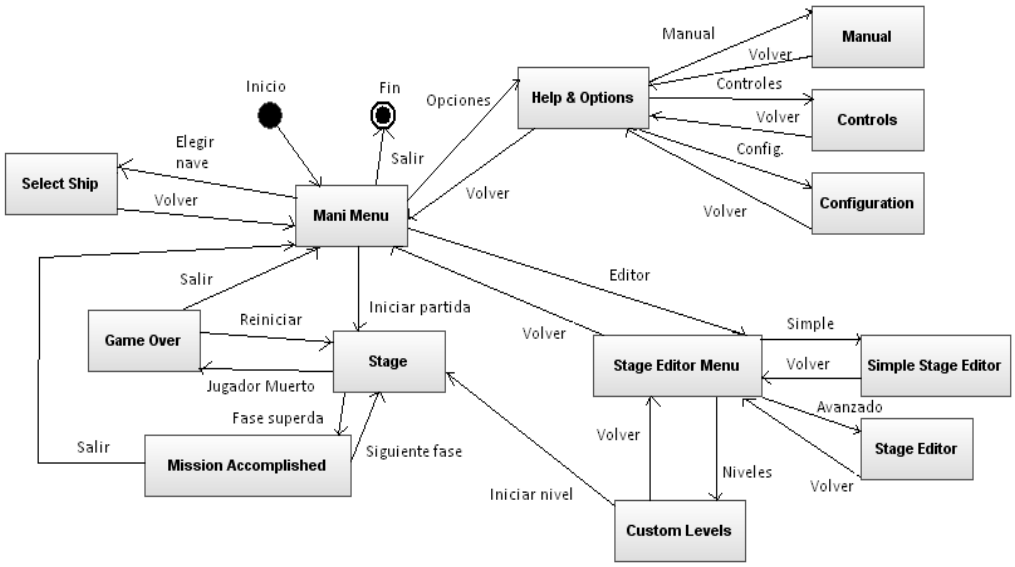


Figura 26. Diagrama de actividad del sistema

CAPÍTULO 4: DISEÑO

Una vez realizada la fase de análisis, se procederá a realizar un diseño que abarque todas las funcionalidades especificadas anteriormente. Dicho diseño, debe ser suficientemente exhaustivo y claro, de forma que no genere dudas durante la fase de implementación.

En primer lugar se va a incluir el diagrama de clases de la aplicación y, posteriormente, se irán explicando cada una de ellas por separado, pues explicarlo todo en conjunto puede resultar un tanto confuso.

4.1 Diagrama de clases

El diagrama de clases describe las clases y objetos que componen el sistema, así como las diversas relaciones que existen entre ellos. Además de determinar las clases que compondrán el programa, es necesario incluir los atributos y métodos de cada una de ellas, así como las diversas restricciones de visibilidad de cada miembro.

A continuación se muestra el diagrama de clases:

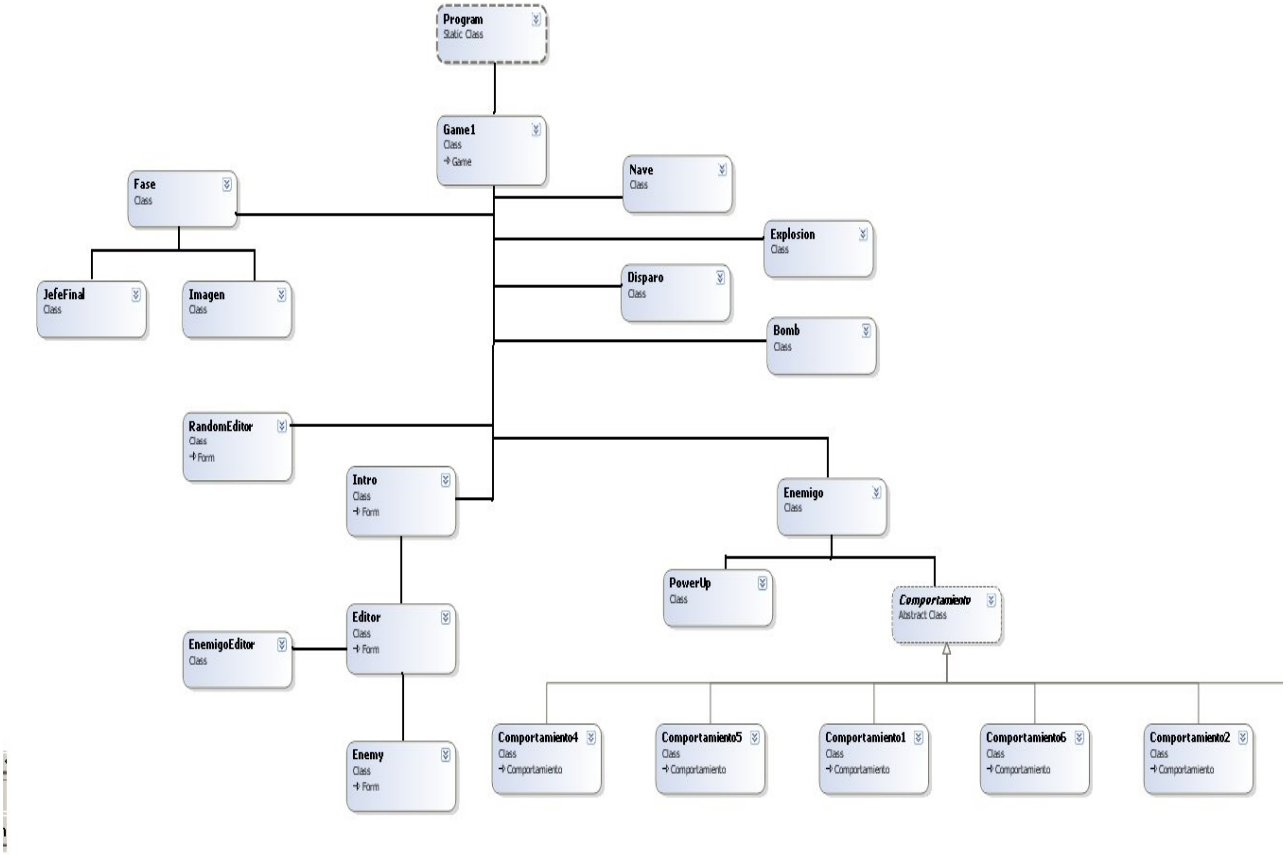


Figura 27. Diagrama de clases

4.2 Definición de las clases

Como se comentaba anteriormente, dada la cantidad de clases que componen el sistema, interpretar el diagrama en conjunto puede resultar complicado. Por ello se van a definir de forma individual, o en conjunto las que tengan una gran relación, de forma que quede clara la función de cada una.

Además, se va a hacer más hincapié en las clases más importantes, y se comentarán los atributos y métodos más relevantes de cada una de ellas.

Por último, resaltar que algunas clases no se explicarán en profundidad dado que hay similitudes entre ellas, como es el caso de los distintos comportamientos de los enemigos.

4.2.1 Program

Es una de las clases generadas automáticamente por XNA. Contiene el método *main* del proyecto, que instancia el juego (Game1) y lo ejecuta.

4.2.2 Game1

Hereda de *Microsoft.XNA.Framework.Game* y proporciona métodos que permiten inicializar y ejecutar el juego. Además, se trata de la clase central de la aplicación que se encarga de gestionar el estado de la pantalla y de los menús, de actualizar los elementos del juego, y de controlar que se dibujen todas las imágenes que deben estar presentes en cada momento. También gestiona el dispositivo de control, teclado o ratón, que se encuentre activo.

Los atributos más importantes son:

- currentScreen: se trata de un enumerado que gestionará la pantalla que debe mostrarse en cada momento.
- currentMenu: se trata de un enumerado utilizado en el menú principal, que se utilizará para determinar en qué menú se encuentra el usuario.
- teclado: una variable *booleana* cuyo valor indica si el control se realiza mediante teclado, o mediante ratón.
- graphics: objeto de tipo *GraphicsDeviceManager*. Es un manejador para la capa de gráficos.
- spriteBatch: objeto de tipo *SpriteBatch*, utilizado para dibujar los elementos 2D en pantalla.

- Los atributos nave, que se corresponde con la nave que utilizará el jugador; y pantalla, que se trata de un objeto de tipo Fase que controlará la fase que tratará de superar el jugador, también estarán incluidos ya que esta clase se ocupa de gestionar los elementos del juego.

- Los atributos enemigos, disparos, disparosEnemigos, bombas, explosiones y powerups, son elementos de tipo *List*<>, y cada una de ellas contiene objetos del tipo especificado. Se corresponden con el resto de elementos que habrá en pantalla durante la partida.

Los métodos más importantes son:

- Initialize: método proporcionado por XNA que se utiliza para cargar el contenido necesario antes de comenzar la ejecución.

- LoadContent: método proporcionado por XNA que se ejecuta una sola vez y permite cargar los recursos de los que hará uso el juego.

- Update: método proporcionado por XNA que se ejecuta sesenta veces por segundo. Contiene toda la lógica del juego.

- Draw: método proporcionado por XNA que se encarga de dibujar los distintos elementos en pantalla.

- ControlRaton: método que contiene la lógica necesaria para que el usuario pueda controlar el juego mediante el uso del ratón.

- ControlTeclado: método que contiene la lógica necesaria para que el usuario pueda controlar el juego mediante el uso del teclado.

- comprobarColisiones: método que se encarga de comprobar si se produce alguna colisión significativa entre los distintos elementos del juego.

- ActualizarElementos: método que se encarga de actualizar a cada instante de tiempo los distintos elementos que componen el juego.

4.2.3 Nave, Disparo y Bomba

La clase Nave se corresponde con el elemento que estará a disposición del jugador durante la partida, y contiene los elementos necesarios para su control y actualización.

Los atributos más importantes son:

- ancho y alto: indican la posición de la nave en la pantalla.
- sprite: controla cual de las imágenes de la nave se muestra en el momento actual.
- Las características de la nave vienen definidas por los atributos: tipo, que es el tipo de nave que se ha elegido; health, que indica la vida restante; power, que indica la potencia de disparo; bombs que indica el número de bombas restantes; y points, que indica la puntuación actual.

Los métodos más importantes son:

- actualizarSprite: en función de la posición anterior y la posición actual, determina si el valor de sprite debe cambiar o no.
- actualizarPosicion: como su propio nombre indica, actualiza la posición de la nave en la pantalla.
- impact: reduce la vida cuando la nave ha sufrido daños, y determina si el jugador ha muerto si la salud es menor o igual a 0.
- Cuando se controla la nave mediante teclado, los métodos: arriba, abajo, izquierda y derecha se ocupan de actualizar la posición de la nave.
- disparoTeclado, bombaTeclado, disparoRaton y bombaRaton: se utilizan para ejecutar la acción correspondiente desde el dispositivo de control que se esté utilizando.
- powerUp: incrementa las capacidades de la nave en función del *power up* que se ha recogido.
- dibujar y dibujarCursor: dibuja la nave en pantalla. Se utiliza uno u otro en función del tipo de control.

La clase Disparo representa los elementos que pueden disparar tanto jugador como enemigos. Su implementación es sencilla, dados dos valores, uno para cada eje de coordenadas, el disparo avanzará trazando una recta utilizando la aceleración especificada.

Los atributos más importantes son:

- alto y ancho: ubican el disparo en la pantalla.
- acx y acy: determinan la velocidad de movimiento del disparo para cada uno de los ejes de coordenadas.

- tipo: indica si es un disparo aliado o enemigo.
- power: determina la potencia del disparo.

Los métodos más importantes son:

- actualizarDisparo: determina la nueva posición del disparo en función de la posición anterior y la velocidad.
- dibujar: dibuja el disparo en la pantalla de juego.

La clase Bomba representa un recurso del que dispone el jugador para eliminar disparos y enemigos de la pantalla. Es un elemento que se lanza y tarda unos pocos segundos en explotar.

Los atributos más importantes son:

- alto y ancho: ubican la bomba en la pantalla de juego.
- status: determina si la bomba está lanzándose o explotando.
- contador: se utiliza para calcular el tiempo que tarda en lanzarse, y el tiempo que está presente la animación de la explosión.

Los métodos más importantes son:

- actualizarBomba: se ocupa de gestionar el estado de la bomba. En un primer momento la bomba se lanza e irá avanzando durante un tiempo fijo. Cumplido ese tiempo, la bomba explotará generando el efecto deseado: eliminar disparos y enemigos.
- dibujar: dibuja la bomba en pantalla.

4.2.4 Enemigo, PowerUp y Explosión

La clase Enemigo es el recipiente desde el que se controla la actividad de los enemigos que aparecen durante la partida.

Los atributos más importantes son:

- ancho y alto: ubican el elemento en la pantalla:
- sprite y tipo: determinan que imagen del enemigo se debe mostrar en el momento actual.
- powerUp: objeto que determina si el enemigo dejará caer un *power up* al morir. Sus características se explicarán a continuación.

- comp: objeto que indica el comportamiento del enemigo. Sus características se explicarán en el siguiente apartado.

Los métodos más importantes son:

- actualizarSprite: determina si el valor de sprite debe cambiar o no.
- mover: calcula la siguiente acción del enemigo. Depende del comportamiento.
- dibujar: dibuja al enemigo en la pantalla de juego.

La clase PowerUp representa las mejoras que pueden dejar caer los enemigos y que pueden ser recogidas por el jugador. Su implementación es sencilla, ya que desde el momento en que aparecen hasta que son recogidos, se mueve en zigzag.

Los atributos más importantes son:

- ancho y alto: ubican el elemento en pantalla.
- sprite: determina que imagen del powerup se muestra en el momento actual.
- contador y dir: se emplean para controlar el movimiento de zigzag del *power up*.

Los métodos más importantes son:

- actualizarPowerUp: calcula la posición y el sprite que tendrá el objeto en la siguiente iteración.
- dibujar: dibuja el *power up* en la pantalla de juego.

La clase Explosión representa un *sprite* que se muestra cada vez que un enemigo es destruido.

Los atributos más importantes son:

- ancho y alto: ubican la explosión en la pantalla.
- sprite: determina que imagen de la explosión hay que mostrar.

Los métodos más importantes son:

- actualizarExplosion: aumenta el valor del atributo sprite periódicamente.
- dibujar: dibuja el elemento en pantalla.

4.2.5 Comportamiento

La clase Comportamiento define las operaciones genéricas de cada uno de los posibles comportamientos que pueden tener los enemigos. Además, determina la vida y la potencia que tendrá dicho enemigo.

Los atributos más importantes son:

- x e y: determinan la velocidad de movimiento.
- desaparecer: indica si el enemigo debe desaparecer cuando se salga de la pantalla o, por el contrario, todavía le quedan cosas que hacer.
- Las características del enemigo vienen determinadas por los atributos: health, que indica la vida restante del enemigo; power, que indica la potencia de sus disparos; y points, que indica los puntos que sumará el jugador si destruye al enemigo.

Los métodos más importantes son:

- mover: determina la siguiente acción del enemigo.
- Ciertas acciones dependen de la cercanía o de la posición del enemigo respecto al jugador o respecto a la pantalla. Dado que la posición viene determinada por valores de tipo *float*, es necesario incluir los métodos aproxCoor y equalCoor, que se emplearán para calcular si el enemigo está a cierta distancia del enemigo, o si se encuentra a su altura.

Los distintos comportamientos que heredan de esta clase implementan el método mover, que determina la secuencia de movimiento del enemigo. Además de lo especificado anteriormente, pueden incluir los siguientes atributos:

- status: indica el movimiento que debe realizar en el momento actual.
- contador: aumenta con cada iteración y se utiliza para controlar los tiempos entre los distintos estados que componen el comportamiento.

4.2.6 Fase, JefeFinal e Imagen

La clase Fase se ocupa de gestionar los distintos elementos que tiene una fase. Proporciona los métodos necesarios para acceder a los ficheros de fondo y de enemigos que componen una fase, gestionar las oleadas de enemigos que aparecerán, determinar si aparece o no un jefe al final de la pantalla y mover las distintas imágenes que componen el *scroll* de fondo.

Los atributos más importantes son:

- matEnemigos: es una matriz que almacena una codificación para cada uno de los enemigos que componen la fase. Cada fila es una oleada.
- sigEnemigos: determina que oleada es la que vendrá a continuación.
- finalBoss: objeto del tipo JefeFinal que determina si la fase tiene un jefe final que aparecerá después de las oleadas de enemigos o no. Las características de la clase se explicarán a continuación.
- matFondo: es una matriz de objetos de tipo Imagen que determina el *scroll* de fondo de la fase. Las características de la clase se explicarán a continuación.
- Los atributos iniFondo, finFondo y contFondo calculan el rango de elementos de matFondo que se deben dibujar.

Los métodos más importantes son:

- getSiguiente: genera la siguiente oleada de enemigos, si aún quedan. En caso contrario, determina que la fase puede finalizar cuando se eliminen o desaparezcan los enemigos restantes.
- getFinalBoss: indica si la fase debe concluir, o debe aparecer un jefe final.
- dibujarFondo: en función de los valores correspondientes al fondo y de la matriz de imágenes, dibuja el *scroll* de fondo en la posición adecuada.

La clase JefeFinal proporciona los elementos necesarios para crear y gestionar los jefes finales de fase.

Los atributos más importantes son:

- ancho y alto: ubican el elemento en pantalla.
- tipo: indica de que jefe final se trata.
- Las características del jefe final vienen determinadas por los atributos: health, indica la salud restante; power, indica la potencia de los disparos; y points, indica la puntuación obtenida al vencerle.

Los métodos más importantes son:

- mover: determina la siguiente acción del jefe final.
- dibujar: dibuja el objeto en la pantalla de juego.

La clase Imagen representa cada una de las pequeñas imágenes que componen el fondo de la pantalla durante la partida.

Los atributos más importantes son:

- ancho y alto: determinan la posición de la imagen en la pantalla.
- coorx y coory: determinan que elemento del *tileset* se corresponde con este objeto.
- X_TAMANO e Y_TAMANO: son unas constantes que indican el ancho y el alto de las imágenes del *tileset*. Si se modifica el *tileset*, solamente hay que cambiar estos valores para que todo siga funcionando.
- FILES_TILESET y COLUMNS_TILESET: son unas constantes que representan el tamaño del *tileset* en imágenes. En caso de que se quiera cambiar, sólo hay que actualizar estos valores.

El método que contiene es:

- dibujar: dibuja la porción del *tileset* que representa el objeto, en la posición especificada.

4.2.7 Editor simple

Como se explicó en la fase de análisis, el editor simple permitirá generar niveles de forma rápida y sencilla. Este editor estará compuesto por un único formulario: RandomEditor.

El formulario deberá pedir, mediante objetos de tipo *comboBox* la duración, el *scroll* y la dificultad del nivel que se quiere generar. Además, permitirá insertar el nombre que se le quiere dar al nivel, restringiendo dicho nombre a letras y dígitos.

El aspecto del formulario será:

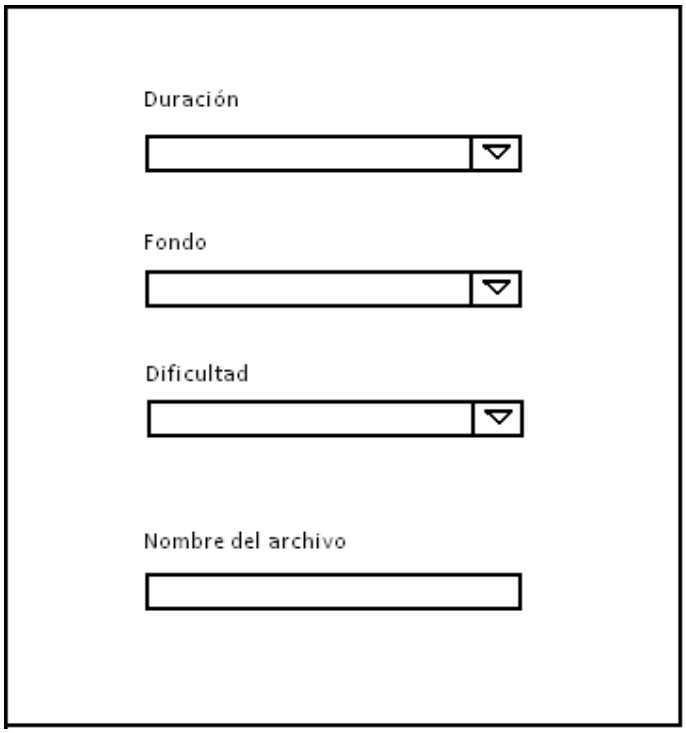


Figura 28. Editor simple: RandomEditor

4.2.8 Editor avanzado

El editor avanzado es más complejo y requerirá de tres formularios, cuyas características se exponen a continuación.

El formulario Intro será el primer en mostrarse. Pedirá, mediante objetos de tipo *comboBox*, la duración del nivel y el *scroll*. También permitirá introducir el nombre del nivel que se desea crear, restringiendo dicho nombre a letras y dígitos.

El aspecto del formulario será:

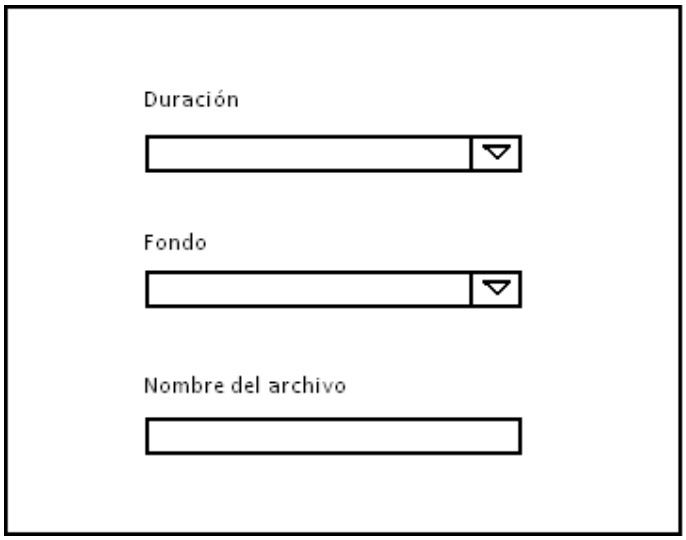


Figura 29. Editor avanzado: Intro

El formulario Editor permitirá configurar los enemigos y el jefe final. Los elementos estarán dispuestos de la siguiente forma:

- En primer lugar se dará la opción de elegir entre no incluir jefe final, o incluir alguno de los jefes finales implementados.
- A continuación habrá una serie de botones, dispuestos en filas, que se corresponderán con los enemigos. Cada enemigo se corresponde con un botón, y la cantidad dependerá de la duración estimada. El texto del botón será la codificación numérica del enemigo, o un guión en caso de que no haya nada.
- Finalmente, se incluirán dos botones que permitirán, previa comprobación de seguridad, guardar el nivel y salir del editor, o salir sin guardar.

El aspecto de la parte superior del formulario será:

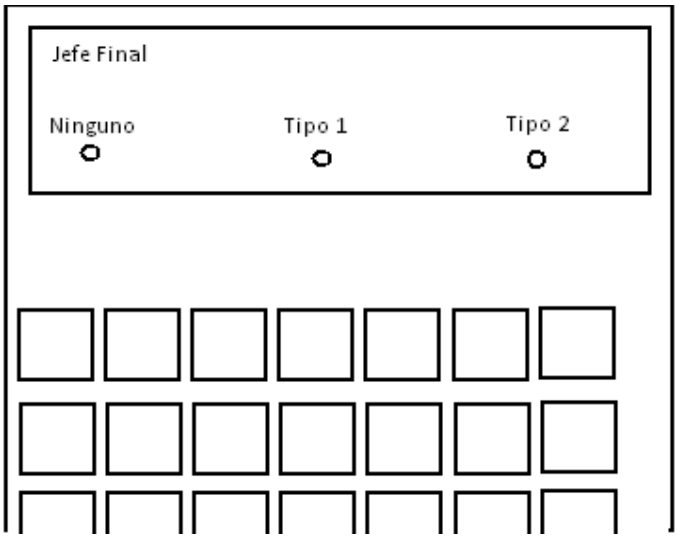


Figura 30. Editor avanzado: Parte superior de Editor

El aspecto de la parte inferior del formulario será:

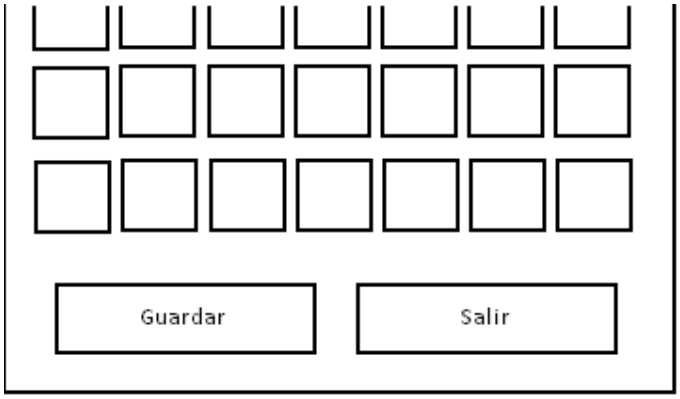


Figura 31. Editor avanzado: Parte inferior de Editor

Cada uno de los botones del formulario Editor se corresponderá con un elemento de la clase EnemigoEditor. Los objetos de dicha clase almacenarán la información correspondiente a la configuración del enemigo que se establezca en su posición.

Sus atributos más importantes son sprite, que indica el *sprite* seleccionado; behaviour, que determina el comportamiento elegido; y powerup que señala si el enemigo dejará caer un *power up* al ser destruido, y cual, o no.

El último formulario, Enemy, se utiliza para configurar los enemigos. Se abrirá cada vez que se pulse uno de los botones del formulario Editor. Su composición es la siguiente:

- En primer lugar y mediante un *radioButton*, permitirá determinar si en esa posición hay un enemigo o no.
- En caso afirmativo, se mostrarán las opciones para elegir el *sprite*, el comportamiento y el *power up*, si se desea, que dejará caer el enemigo al ser destruido.

El aspecto del formulario será:

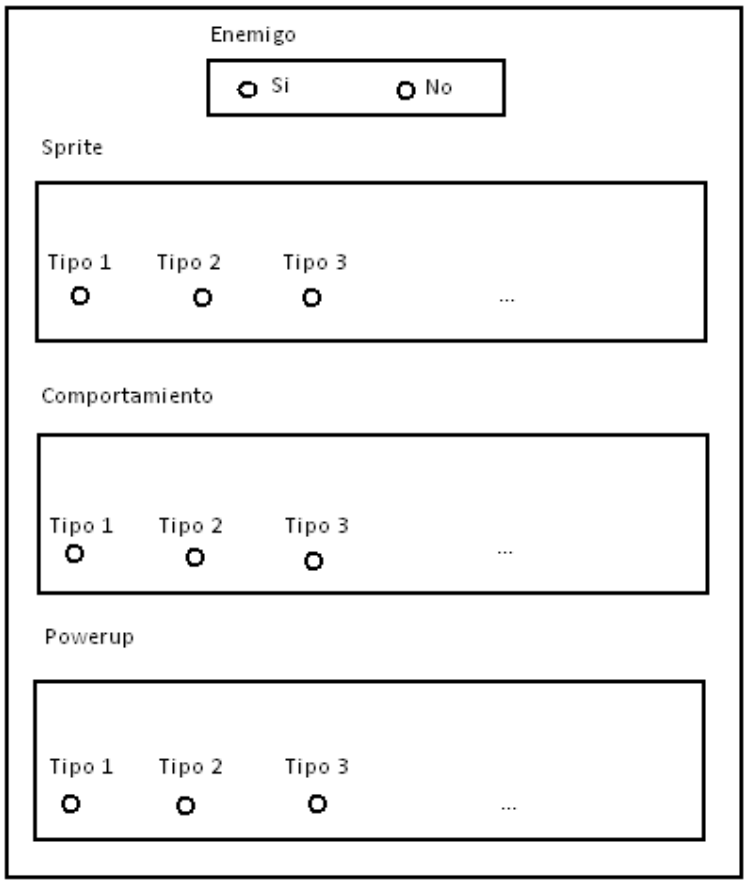


Figura 32. Editor avanzado: Enemy

CAPÍTULO 5: IMPLEMENTACIÓN

En este apartado se va a utilizar el análisis y el diseño para llevar a cabo la implementación del juego.

Se hace necesario destacar que el objetivo de este punto no es comentar en profundidad el código de cada clase que compone el juego, sino que lo que se pretende es ahondar en los aspectos más importantes del desarrollo.

Así, en este apartado se van a contar cosas importantes como la gestión del estado del juego, la implementación del jugador así como su control y sus acciones, la detección de colisiones, los enemigos y su comportamiento, como se han creado las fases, etc.

5.1 Gestión de estado

A la hora de implementar el juego se sabe que va a existir una cierta cantidad de pantallas tales como menús, pantallas de carga, fase de juego, etc. Por lo que es necesaria una gestión del estado de la pantalla para que el juego sepa en todo momento que información debe mostrar, y con qué elementos debe trabajar.

Pese a que hay muchas formas de llevar a cabo la gestión de estado [59], en este caso se ha optado por utilizar atributos de tipo enumerado.

```
enum ScreenState
{
      TitleIn,
      Title,
      TitleOutToStage,
      StageIn,
      Stage,
      StagePause,
      StageOut,
      StageBoss,
      StageBossPause,
      StageDeadIn,
      StageDead,
      StageDeadOut,
      StageClearedIn,
      StageCleared,
      StageClearedOut,
}
ScreenState currentScreen;
```

Código 1: ScreenState

El atributo currentScreen le permitirá al juego saber, en todo momento, en qué situación se encuentra. Así, el método *Update*, que se ejecuta sesenta veces por segundo, tendrá la siguiente arquitectura:

```
switch (currentScreen)
{
     case ScreenState.TitleIn:
           /* Código correspondiente */
     break;
     case ScreenState.Title:
           /* Código correspondiente */
     break;
     case ScreenState.TitleOutToStage:
           /* Código correspondiente */
     break;

     /* ... */
}
```

Código 2: Gestión de estado en *Update*

Con un simple *switch* y un *case* para cada uno de los estados, podemos gestionar el estado del juego y hacer que éste se comporte como debe. Para realizar la transición de un estado a otro basta con cambiar el valor del atributo currentScreen.

Como último apunte, en el método *Draw* hay que hacer algo semejante para que el juego sepa también, en todo momento, que elementos tiene que dibujar en pantalla. La arquitectura del método *Draw* será similar a la del método *Update*.

La gran mayoría de los estados que contiene el enumerado definido en la tabla de código 1 no requieren de una subgestión de estado. No obstante, esto es distinto para el estado *Title* que se corresponde con el menú principal del juego. El menú principal tiene a su vez varios submenús, y éstos a su vez pueden contender más submenús, por lo que necesita su propia gestión de estado.

El tratamiento que se le ha dado a la gestión del estado del menú principal es similar a la mencionada anteriormente: un enumerado. A continuación se incluye el código de dicho atributo:

```
enum MenuState
{
    Title,
    StageEditor,
    StageSelect,
    ShipSelect,
    HelpOptions,
    Manual,
    Controls,
    Configuration,
}
MenuState currentMenu;
```

Código 3: currentMenu

El atributo currentMenu le permite al juego determinar que menú debe dibujar en cada momento. Así, sabrá qué menú mostrar, y qué transiciones realizar en función de las opciones seleccionadas por el usuario. Recordar que esta gestión de estado sólo se encuentra activa cuando el juego se encuentra en el menú principal.

Con tan solo dos atributos enumerados es posible realizar la completa gestión del estado del videojuego que se está desarrollando, pues ninguno de los demás estados mencionados en la tabla de código 1 requiere de una subgestión propia.

5.2 Jugador

Para implementar el elemento jugador, que será la nave que controla el usuario, es necesario implementar la entrada, que podrá realizarse mediante teclado o mediante ratón; el movimiento y las acciones derivadas de dicho método de control, y la animación del *sprite* que representa la nave.

5.2.1 Control mediante teclado

XNA proporciona diversos recursos para trabajar con el teclado. El que se ha empleado en esta ocasión consiste en la utilización de una variable de tipo *KeyboardState* que permite determinar el estado actual del teclado. De esta forma, se pueden mapear las teclas para comprobar si alguna está siendo pulsada en el momento actual.

```
KeyboardState keybState;
keybState = Keyboard.GetState();
if (keybState.IsKeyDown(Keys.Space))
{
      disparoTeclado();
}
```

Código 4: Ejemplo de control mediante teclado

En este ejemplo se muestra como se detecta si la tecla espacio ha sido pulsada y, si es así, ejecuta la acción de disparo. Así, para realizar todos los movimientos permitidos en el juego, a saber: derecha, izquierda, arriba, abajo, disparar y lanzar bomba; el mapeo de teclas queda como se muestra en la tabla siguiente.

```
keybState = Keyboard.GetState();
if (keybState.IsKeyDown(Keys.Left))
{
    if (nave.getAncho() > limiteLeft)
    {
       nave.izquierda();
    }
}
if (keybState.IsKeyDown(Keys.Right))
{
    if (nave.getAncho() < limiteRight)
    {
       nave.derecha();
    }
}
if (keybState.IsKeyDown(Keys.Up))
{
    if (nave.getAlto() > limiteUp)
    {
       nave.arriba();
    }
}
if (keybState.IsKeyDown(Keys.Down))
{
    if (nave.getAlto() < limiteDown)
    {
       nave.abajo();
    }
}
if (keybState.IsKeyDown(Keys.Space))
{
    disparoTeclado();
}
if (keybState.IsKeyDown(Keys.Z))
{
    bombaTeclado();
}
```

Código 5: Mapeo de teclas

Se puede ver que cada una de las teclas contempladas en el código se corresponde con una de las acciones permitidas. Sin embargo, surge un problema debido a que este método lo que comprueba es que la tecla correspondiente esté pulsada. Dado que el método *Update* se ejecuta sesenta veces por segundo, una leve pulsación puede cubrir varios de esos instantes y provocar que se detecte la pulsación varias veces cuando sólo se pretendía realizar una acción.

Para el movimiento esto no supone un problema, pues la solución es tan sencilla como ajustar la velocidad a la que se mueve la nave, hasta conseguir la velocidad deseada.

Sin embargo, para el disparo o el lanzamiento de bombas esto supone un problema pues una sola pulsación puede provocar una oleada de disparos cuando sólo se pretendía lanzar uno. Dado que se pretende controlar la cadencia de fuego de la nave, la solución consiste en incluir un contador para que haya un tiempo de inactividad entre disparo y disparo.

```
void disparoTeclado()
{
    if (tiempoEntreDisparos > 10)
    {
        nave.disparoTeclado(disparos, disparoTexture);
        tiempoEntreDisparos = 0;
    }
}
```

Código 6: Controlar la cadencia de disparo

Empleando la variable entera tiempoEntreDisparos, que incrementa una unidad cada vez que se ejecuta el método *Update*, se permite que tan solo haya seis disparos por segundo en lugar de sesenta. Así, la cadencia de fuego de la nave queda, no sólo controlada, sino también realista.

5.2.2 Control mediante ratón

Al igual que sucedía con el teclado, también disponemos de distintos recursos para trabajar con el ratón [60] como método de control del juego. En este caso, utilizaremos varios objetos de tipo *MouseState* para controlar el estado del ratón y actuar en consecuencia.

A diferencia de lo que sucedía antes, al utilizar este método para controlar el ratón, los eventos se capturan de forma distinta.

```
MouseState mouseStateCurrent, mouseStatePrevious;
MouseState mouseState;

mouseStateCurrent = Mouse.GetState();

if (mouseStateCurrent.LeftButton == ButtonState.Pressed &&
mouseStatePrevious.LeftButton == ButtonState.Released)
{
      disparoRaton();
}

if (mouseStateCurrent.RightButton == ButtonState.Pressed &&
mouseStatePrevious.RightButton == ButtonState.Released)
{
      bombaRaton();
}

mouseStatePrevious = mouseStateCurrent;
```

Código 7: Disparo con el ratón

En este caso, se puede apreciar que se actúa cuando el botón correspondiente del ratón está presionado, pero no se vuelve a realizar la acción hasta que no se suelta dicho botón y se vuelve a pulsar. De esta forma, se evita el problema de los disparos múltiples que surgió trabajando con el teclado, pues una sola pulsación se corresponde con un único disparo. Y lo mismo sucede con el lanzamiento de bombas.

La gran diferencia con respecto al control anterior radica en el control del movimiento de la nave. Como bien se sabe, en un ordenador el ratón controla el cursor. Pues lo mismo ocurre con el juego. La diferencia es que, en este caso, el icono del ratón no va a ser una flecha, sino la propia nave. Así, al desplazar el ratón y mover el cursor, se actualizará la posición de la nave y con ello se tendrá el movimiento implementado.

```
mouseState = Mouse.GetState();
mousePosition.X = mouseState.X;
mousePosition.Y = mouseState.Y;
/* Se restringe el movimiento al los limites establecidos
*/
if (mousePosition.X < limiteLeft)
      mousePosition.X = limiteLeft;
if (mousePosition.X > limiteRight)
      mousePosition.X = limiteRight;
if (mousePosition.Y < limiteUp)
      mousePosition.Y = limiteUp;
if (mousePosition.Y > limiteDown)
      mousePosition.Y = limiteDown;
nave.setAncho(mousePosition.X);
nave.setAlto(mousePosition.Y);
```

Código 8: Movimiento con el ratón

Dicho método, que actualiza la posición de la nave en función de la posición actual del ratón, se actualiza cada vez que se ejecuta el método *Update*. Así se consigue una mayor precisión en el control.

En la implementación, determinar si el usuario ha elegido utilizar el ratón o el teclado, es tan simple como utilizar una variable cuyo valor indica si se está utilizando uno u otro método de control.

```
if (teclado)
{
      ControlTeclado()
}
else
{
      ControlRaton();
}
```

Código 9: Control del periférico que se utiliza

Como último apunte, se quiere recordar que el método de control puede ser seleccionado por el usuario en el menú “Controles”. Dicho menú informa del método de control actual, así como de los botones de disparo y lanzamiento de bombas.

5.2.3 Dibujo y animación de la nave

Una vez se tiene implementado el control del jugador es necesario representarlo en pantalla. No sólo hay que saber donde posicionarlo, sino también que imagen darle y como conseguir que provoque sensación de movimiento.

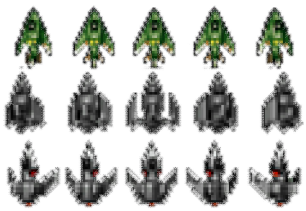


Figura 33. Texturas de la nave

Hay que tener en cuenta un aspecto importante a la hora de dibujar una imagen en pantalla utilizando XNA. A diferencia de lo habitual, en XNA el punto [0,0] está situado en la parte superior izquierda de la pantalla. Un incremento en el eje de abscisas implica que la imagen se desplazará a la derecha, y un incremento en el eje de ordenadas implica que la imagen se desplazará hacia abajo. Una vez se tiene claro este aspecto, se puede proceder a explicar el tratamiento de imágenes.

La nave se representa mediante un *sprite* en 2D. En la figura 33 se muestran los tres tipos de naves disponibles, y la tira de *sprites* de cada una de ellas.

```
Texture2D naveTexture;
naveTexture = Content.Load<Texture2D>("PlayerShip");
```

Código 10: Carga de una imagen

En primer lugar, es necesario cargar el archivo con la imagen y asignarlo a un objeto de tipo *Texture2D*, para que el juego sepa que va a trabajar con un objeto en dos dimensiones. Posteriormente, se debe crear el objeto de tipo Nave, que representa al jugador, y pasarle por parámetro la textura. El propio objeto contiene el método necesario para dibujar la nave cuando el juego se lo solicite.

```
spriteBatch.Draw(this.getTextura(), this.getPosicion(),
new Rectangle(sprite * X_TEXTURE, tipo * Y_TEXTURE,
X_TEXTURE, Y_TEXTURE),
new Color(255, 255, 255, (byte)MathHelper.Clamp(alphaValue,
0, 255)),
MathHelper.ToRadians(0), new Vector2(0, 0),
this.getScala(), SpriteEffects.None, 0);
```

Código 11: Dibujar la nave

El método de dibujo tiene varios aspectos importantes que hay que comentar. En primer lugar, se pasa la textura que se va a dibujar. En segundo lugar está la posición, que es un vector de dos valores que determinan la ubicación en pantalla del objeto. El tercer parámetro, y el más importante dado que estamos utilizando un archivo con múltiples imágenes, es el que determina que rectángulo del archivo se debe dibujar.

El constructor del objeto de tipo *Rectangle* que contiene cuatro parámetros que indican lo siguiente:

- El primero indica a partir de que píxel en el eje de abscisas de la imagen se debe dibujar. Para determinar la posición en la que se encuentra la nave, quieta o en movimiento, se utiliza la variable sprite. Más adelante se explicará la animación de la nave con más detalle.
- El segundo indica a partir de que píxel del eje de ordenadas de la imagen se debe dibujar. Para determinar que nave se está utilizando, se utiliza la variable entera tipo, que representa la nave seleccionada.
- El tercer parámetro indica cuantos píxeles se van a dibujar, hacia la derecha, a partir del punto indicado en el primer parámetro.

- El cuarto parámetro indica cuantos píxeles se van a dibujar, hacia abajo, a partir del punto indicado en el segundo parámetro.

Los siguientes parámetros del método de dibujo indican el color, la rotación, el desplazamiento respecto a la posición indicada en el segundo argumento y la escala. Los dos últimos parámetros no tienen relevancia pues no han sido utilizados en todo el proyecto.

Como se ha comentado anteriormente, posicionar la imagen no es lo único que hay que conseguir. También se le debe dar al jugador la impresión de que la nave está en movimiento [61] . Trabajando en un entorno 2D esto se consigue utilizando varias imágenes para un mismo elemento. En la figura 33 se muestra la tira de imágenes disponible para cada una de las naves, ahora se va a mostrar como determinar cuál de ellas dibujar.

```
int an = (int)this.getAncho();
int al = (int)this.getAlto();

if (an < spriteAncho)
{
      this.setSprite(-1);

else if (an > spriteAncho)
{
      this.setSprite(1);
}
else
{
      if (this.getSprite() < 2)
      {
            this.setSprite(1);
      }
      else if(this.getSprite() > 2)
      {
            this.setSprite(-1);
      }
}
spriteAncho = (int)this.getAncho();
spriteAlto = (int)this.getAlto();
```

Código 12: Variación de la variable *sprite*

Si la nave está en la misma posición, lo apropiado es que no se mueva, es decir, que se escoja la imagen del centro. Por el contrario si la nave está en movimiento, se debe determinar si se está desplazando hacia la derecha o hacia la izquierda, y cambiar la imagen que se debe mostrar. Esto se hace mediante la modificación del atributo sprite mostrado en la tabla de código 12.

Se compara la posición actual con la posición anterior. Si se está desplazando hacia la izquierda, decrece el valor de sprite, lo que indica que se van a utilizar las imágenes de giro a la izquierda. Si por el contrario, se está desplazando a la derecha, el valor de sprite crecerá.

Además, hay que controlar cuando la nave ha finalizado un movimiento para volver a la posición inicial. De esta manera, cuando se compruebe que la posición de la nave no ha variado, el valor de sprite se ajustará progresivamente hasta elegir la imagen central de la tira de sprites de la nave.

Por tanto, acabamos de ver como con una tira de imágenes y unos sencillos métodos, es posible provocar en el jugador la sensación de que la nave que controla se está moviendo a medida que se desplaza por la pantalla.

5.3 Enemigos

Una vez terminada la implementación del jugador, el siguiente paso trata de los enemigos. Los distintos tipos de naves, cada una de ellas con su correspondiente comportamiento, que aparecerán a lo largo de las fases para tratar de evitar que el jugador llegue hasta el final.

Figura 34. Texturas de los enemigos

Los enemigos se implementan gracias a las clases Enemigo y Comportamiento. Enemigo se ocupa de gestionar los aspectos relacionados con este tipo de objetos, y comportamiento determina sus características, así como sus acciones de movimiento y disparo. Si se desea incluir nuevos tipos de enemigos, sería tan sencillo como incluir una nueva clase que herede de Comportamiento.

No es necesario ahondar en la clase Enemigo, pues los aspectos más importantes como el posicionamiento en pantalla, el dibujo o la animación son similares a los vistos en el jugador.

5.3.1 Comportamiento

La forma de actuar de un enemigo viene determinada por su atributo “comp”, que es un objeto que hereda de la clase Comportamiento. Dicho comportamiento determina su vida, la potencia de sus disparos, los puntos que proporciona al jugador y, lo más importante, como se mueve y como dispara durante la vida del enemigo.

Pero antes de explicar los distintos tipos de enemigos que se van a implementar, hay un aspecto importante a destacar. A la hora de optimizar recursos, un enemigo que no ha sido destruido, pero que ha abandonado la pantalla, debe ser eliminado. Sin embargo, es posible que algunos comportamientos provoquen la circunstancia de que el enemigo abandone la pantalla para, posteriormente, volver a ella y atacar al jugador.

Para diferenciar estas dos situaciones, se ha utilizado una variable booleana “desaparecer” que determina si, cuando se hace la comprobación para determinar que enemigos deben ser eliminados, el enemigo debe desaparecer al abandonar la pantalla o no. En algunos casos, dicha variable estará tendrá valor *true* por defecto, mientras que en otros deberá actualizarse a dicho valor al cambiar de uno a otro estado de su comportamiento.

A continuación se van a indicar los distintos tipos de comportamientos implementados, con una breve explicación de cada uno de ellos.

Tipo 1.

El primer comportamiento es muy sencillo, pues tan solo tiene un estado. Se trata de un enemigo que recorre a velocidad constante la pantalla desde la parte superior hasta la parte inferior, mientras realiza un movimiento de zigzag.

Figura 35. *Sprite* asociado al Tipo 1

Siempre que va a iniciar el movimiento hacia la derecha, el enemigo realizará un disparo apuntado hacia la posición actual del jugador.

Tipo 2.

El segundo tipo de comportamiento es más complejo, y aborda el problema mencionado anteriormente acerca de salirse de la pantalla durante el tiempo de vida del enemigo.

Estos enemigos bajan en línea recta hasta situarse a la misma altura que el jugador. En ese momento, realizan un movimiento diagonal ascendente hasta posicionarse encima de la nave que controla el jugador, para descender a una velocidad ligeramente superior e intentar colisionar con el jugador.

Figura 36. *Sprite* asociado al Tipo 2

En el transcurso del movimiento ascendente, es posible que el enemigo salga de la pantalla por la parte superior. Esto no provocará que el enemigo desaparezca, simplemente el enemigo saldrá de la pantalla, pero seguirá ejecutando su movimiento. Dicha posibilidad supondrá una dificultad añadida, pues puede darse el caso de que el jugador lo pierda de vista, y luego sea sorprendido por el enemigo.

Finalmente, en el caso de que el jugador no destruya al enemigo, y éste desaparezca por la parte inferior de la pantalla, si podrá ser eliminado por el elemento del juego que controla a los enemigos.

Tipo 3.

El tercer comportamiento consiste en una nave enemiga que desciende en línea recta desde la parte superior de la pantalla. La peculiaridad de este comportamiento es que, cuando se aproxima al jugador, girará bruscamente en su dirección durante unos instantes con la intención de embestirle.

Figura 37. *Sprite* asociado al Tipo 3

Finalmente, continuará con su movimiento descendente en línea recta hasta que desaparezca de la pantalla. El enemigo desaparecerá tan pronto como abandone la pantalla por la parte inferior.

Tipo 4.

El cuarto comportamiento tiene similitudes con el mencionado anteriormente. En este caso, vuelve a tratarse de una nave que desciende en línea recta a velocidad constante desde la parte superior de la pantalla. Sin embargo, en este caso, cuando la distancia entre el enemigo y el jugador es de aproximadamente un tercio de la pantalla, deja caer un disparo. Este disparo también va en línea recta, mientras que la velocidad del enemigo aumenta y el movimiento se convierte en una diagonal descendente.

Figura 38. *Sprite* asociado al Tipo 4

El enemigo desaparecerá tan pronto como abandone la pantalla. Ya sea por la parte inferior o por alguno de los laterales.

Tipo 5.

El quinto comportamiento vuelve a implicar una serie de disparos. La nave aparece por la parte superior de la pantalla y avanza durante unos pocos segundos. Después detiene su movimiento, y realiza cuatro disparos consecutivos que van en dirección a la posición actual del jugador.

Figura 39. *Sprite* asociado al Tipo 5

Finalizada la tanda de disparos, desaparecerán de la pantalla con un movimiento diagonal descendente hacia la derecha o hacia la izquierda, según que borde de la pantalla le sea más próximo.

El enemigo desaparecerá tan pronto como abandone la pantalla. Ya sea por la parte inferior o por alguno de los laterales.

Tipo 6.

El sexto y último comportamiento, tiene dos variantes.

Si el enemigo aparece más cerca del centro de la pantalla que de los bordes, avanzará durante unos pocos segundos. Posteriormente se detendrá, y realizará una acción de disparo en abanico hacia abajo que dejará siete disparos en pantalla que se irán abriendo a medida que avancen.

Figura 40. *Sprite* asociado al Tipo 6

Si el enemigo aparece más cerca de alguno de los bordes de la pantalla que del centro, descenderá hasta situarse a la misma altura que el jugador en el eje de ordenadas, y realizará el mismo disparo en forma de abanico. La diferencia es que, en este caso, el disparo estará orientado en dirección al jugador.

Finalmente, e independientemente de la zona en la que haya aparecido, el enemigo realizará un movimiento ascendente en línea recta para tratar de escapar por la parte superior de la pantalla.

Como último apunte, es interesante comentar que en la primera versión de los comportamientos mencionados, la velocidad tanto de los enemigos como de los disparos, resultó ser demasiado elevada para la mayoría de las personas que probaron el prototipo. Por lo que al final se decidió realizar una serie de ajustes a la velocidad de movimiento, número de disparos y velocidad de los mismos.

5.3.2 Animación de sprites

Aunque la animación de *sprites* es algo que ya se ha tratado en el apartado 5.2.3, se ha considerado apropiado hacer énfasis en otra forma de animar un *sprite*.

En este caso se trata de *sprites* continuos, es decir, que la primera imagen de la derecha del archivo es la continuación de la última imagen de la izquierda. Se puede ver a que se hace referencia en las dos últimas tiras de *sprites* de la figura 34.

```
if (tipo == 5)
{
      sprite++;
      if (sprite > 7)
            sprite = 0;
}
```

Código 13: Animación de *sprites* continuos

Animar imágenes así es mucho más sencillo, especialmente si están en constante movimiento como sucede con los dos *sprites* mencionados. Sencillamente hay que aumentar de forma periódica el valor de la variable *sprite*, que es la que determina que imagen hay que mostrar, controlando que no se salga del rango.

5.4 Límites y colisiones

Los objetos se pueden desplazar por toda la pantalla, e incluso fuera de los límites de esta. De la misma forma, también pueden pasar unos por encima de otros sin que, por ello, suceda nada, a menos que esto se controle.

En este punto se va a comentar como restringir los movimientos de la nave a la pantalla de juego, y como realizar la detección de colisiones entre los distintos elementos del juego.

5.4.1 Límites de la pantalla

La pantalla tiene una resolución de 800x600, lo que quiere decir que, mientras la posición de la nave se encuentre entre cero y esos valores, se mostrará en pantalla. Dado que el movimiento puede llevarla fuera de ese rango, hay que establecer unos límites para impedirle abandonar la pantalla de juego.

```
private void setUpLimits()
{
      limiteUp = 200;
      limiteDown = screenHeight - (nave.getY() * nave.getScala());
      limiteRight = screenWidth - (nave.getX() * nave.getScala());
      limiteLeft = 0;
}
```

Código 14: Límites de movimiento de la nave

Este método, ejecutado al cargar el juego, restringe los movimientos de la nave al interior de la pantalla. Como se explicó anteriormente, el punto [0,0] se encuentra situado en la esquina superior izquierda de la pantalla, por lo que el límite izquierdo se corresponderá con el valor 0. El límite superior está acotado a 200, impidiendo al jugador subir más allá de esa frontera, para evitar que se sitúe en la parte superior de la pantalla y un enemigo le sorprenda sin posibilidad alguna de detectarlo.

Sin embargo, tanto el límite inferior, como el límite derecho, dependen del tamaño de la imagen que representa la nave. El cálculo realizado para determinar ambos límites consiste en restar al tamaño de la pantalla, el tamaño de la textura que representa la nave multiplicado por la escala de dicha imagen. De esta manera se obtiene el número de píxeles que ocupa la nave en el eje de abscisas para el límite derecho, y lo mismo sucede para el eje de ordenadas para el límite inferior.

Así, la nave podrá pasear por los bordes de la pantalla sin llegar nunca a salirse de ella, por mucho que el usuario lo intente.

5.4.2 Detección de colisiones

La detección de colisiones es uno de los aspectos más importantes de un videojuego. Impedir que un personaje atraviese una pared, evitar que dos coches se atraviesen o hacer que un disparo impacte en un enemigo son ejemplos de lo necesario que es esto en un videojuego.

Existen diversas formas de realizar la detección de colisiones y, dadas las características del juego que se está desarrollando, se ha elegido la detección de colisiones a nivel de píxel. Esto es, para que dos elementos colisionen, deben hacerlo dos píxeles no transparentes de cada una de sus imágenes.

Figura 41. Detección de colisiones a nivel de píxel

Si se toma como referencia el rectángulo de las imágenes, se podría decir que en la imagen izquierda de la figura 41 hay una colisión. Sin embargo, se puede ver que la nave pequeña está atravesando píxeles transparentes de la imagen de la nave grande, por lo que no existe dicha colisión.

En cambio, en la imagen derecha de la figura 41 se puede apreciar como sí existe una colisión. Los píxeles de color del disparo han alcanzado píxeles no transparentes de la nave grande, y se ha producido una colisión en el rectángulo marcado en rojo.

Una visión general de lo que es una colisión a nivel de píxel queda explicado en la siguiente tabla:

Imagen 1: Píxel Transparente	Imagen 2: Píxel transparente	Colisión
Si	Si	No
Si	No	No
No	Si	No
No	No	Si

Tabla 107: Visión general de la detección de colisiones

Como se deduce de la tabla anterior, para realizar la comprobación es necesario recorrer ambas matrices píxel a píxel para comprobar si existe alguna colisión entre las dos imágenes. El problema que deriva de utilizar este método es el coste de procesamiento. Sin embargo, en este proyecto se trabaja con imágenes pequeñas, y se ha incluido una mejora que eliminará decenas de comprobaciones innecesarias de manera que se optimice el rendimiento del juego. Dicha mejora se explicará más adelante.

Una vez explicado el método de detección de colisiones que se va a utilizar, lo siguiente es explicar su implementación.

En primer lugar, se necesita obtener la matriz de píxeles con su correspondiente color.

```
Color[] colors1D = new Color[texture.Width * texture.Height];
texture.GetData(colors1D);
```

Código 15: Obtención de la matriz unidimensional de píxeles

Con este sencillo método, creamos una matriz unidimensional de objetos de tipo Color que sea capaz de almacenar el color de cada uno de los píxeles que componen la imagen. No obstante, como queremos trabajar con matrices bidimensionales, hay que transformar dicha matriz.

```
Color[,] colors2D = new Color[texture.Width, texture.Height];
for (int x = 0; x < texture.Width; x++)
      for (int y = 0; y < texture.Height; y++)
            colors2D[x, y] = colors1D[x + y * texture.Width];
```

Código 16: Obtención de la matriz bidimensional de píxeles 1

Así, tendremos la matriz bidimensional de colores de una imagen. Pero esto no es suficiente para acometer la comparación de imágenes. Es necesario transformar la matriz obtenida, de forma similar a como lo hace XNA, para ubicar la imagen en la pantalla de manera que la detección de colisiones sepa en que posición se encuentra cada píxel.

```
Matrix disparoMat = Matrix.CreateTranslation(0, 0, 0) *
Matrix.CreateScale(disparos[i].getScala()) *
Matrix.CreateTranslation(disparos[i].getAncho(),
disparos[i].getAlto(), 0) *
Matrix.Identity;
```

Código 17: Obtención de la matriz bidimensional de píxeles 2

En primer lugar situamos la matriz en el punto [0,0]. Después ajustamos la escala que se especificó para la imagen a la hora de dibujarla. Posteriormente se realiza el desplazamiento de la imagen a la zona en la que está actualmente y, finalmente, se multiplica por la matriz identidad, que podría omitirse.

Ahora si tenemos la imagen en la posición deseada, tal y como la ha ubicado XNA en la pantalla. Con esta matriz, y realizando la misma transformación en la matriz con la que se quiere realizar la comprobación, se puede determinar si ambas imágenes colisionan. La idea principal del método es:

```
Para cada píxel de la primera imagen:
      Buscar la coordenada de la pantalla que ocupa.
      Buscar el píxel de la segunda imagen que ocupa esa misma
coordenada
      SI ambos no son transparentes
            COLISION
```

Código 18: Pseudo-código de la detección de colisiones

Traducido a código:

```
private Vector2 TexturesCollide(Color[,] tex1, Matrix mat1, int
width1, int height1, Color[,] tex2, Matrix mat2, int width2,
int height2)
{
      Matrix mat1to2 = mat1 * Matrix.Invert(mat2);

      for (int i = 0; i < width1; i++)
      {
            for (int j = 0; j < height1; j++)
            {
                  Vector2 pos1 = new Vector2(i, j);
                  Vector2 pos2 = Vector2.Transform(pos1,
mat1to2);

                  int x = (int)pos2.X;
                  int y = (int)pos2.Y;
                  if ((x >= 0) && (x < width2))
                  {
                      if ((y >= 0) && (y < height2))
                      {
                          if (tex1[i, j].A > 0)
                          {
                              if (tex2[x, y].A > 0)
                              {
                                  Vector2 screenPos =
Vector2.Transform(pos1, mat1);
                                  return screenPos;
                              }
                          }
                      }
                  }
            }
      }

      return new Vector2(-1, -1);
}
```

Código 19: Detección de colisiones entre dos matrices

Multiplicando la primera matriz por la inversa de la segunda obtenemos la matriz “mat1to2”. Transformando una coordenada de la primera imagen utilizando esta matriz, nos da automáticamente la coordenada en la imagen 2. Antes de comprobar si ambos píxeles colisionan, debemos asegurarnos de que la coordenada esté dentro de los rangos de la imagen. Una vez hecho, se comprueba si ambos píxeles no son transparentes. Si es así, se determina que ha habido una colisión, y se devuelve el punto donde se ha producido. En caso contrario, se sigue recorriendo la matriz hasta el final. Una vez tenemos el método, se van a analizar las distintas colisiones que pueden producirse.

Un disparo aliado alcanza a un enemigo. Una vez se produzca el impacto, se calculará la vida restante del enemigo reduciendo a la vida actual la potencia del disparo que ha impactado. Si la vida resultante es inferior a cero, el enemigo será destruido.

Figura 42. *Sprite* correspondiente a una explosión

Un disparo enemigo alcanza al jugador. Si un disparo enemigo alcanza al jugador, se comprueba la vida restante del jugador. Si esta es inferior a cero, el jugador será destruido y la partida finalizará. En caso contrario, se restará la cantidad correspondiente a la potencia del disparo de la barra de vida.

Figura 43. Barra de vida del jugador

Un enemigo alcanza al jugador. Si un enemigo colisiona con el jugador, éste quedará destruido. El daño que sufrirá la nave será igual a la vida restante del enemigo. Al igual que antes, si la salud resultante del jugador es inferior a cero, la partida finalizará.

Figura 44. Imágenes de los disparos aliados

El jugador alcanza un *power up*. Si el jugador alcanza un *power up*, se beneficiará de sus propiedades. El *power up* desaparecerá y el jugador podrá obtener un incremento en la potencia de disparo, una bomba adicional, recuperar vida, o sumar puntos adicionales.

Figura 45. *Power ups*

Como se ha comentado anteriormente, realizar la comprobación de colisiones sesenta veces por segundo para todas las imágenes que son susceptibles de colisionar entre sí puede resultar muy costoso. Por ello, teniendo en cuenta que el tamaño de las imágenes, tanto de los enemigos, como de la nave, de los *power ups* o de los disparos es inferior a 50 píxeles, se ha introducido una sencilla línea de código que comprueba la distancia entre ambos objetos. Si dicha distancia es inferior a 50 píxeles, tanto en el eje de abscisas como en el eje de ordenadas, se comprueba si se produce una colisión, en caso contrario no se hace nada. La excepción surge con los jefes finales, donde el rango de comprobación se ha ampliado para ajustarlo a su tamaño.

5.5 Fase

Una vez implementado el jugador, los enemigos y el sistema de colisiones, ya se tiene definido el motor del juego. Todos esos elementos deben unirse en conjunto para poder dar al usuario una experiencia jugable interesante. Esto se hace en una pantalla de juego.

La clase Fase, es la encargada de gestionar todos los elementos, salvo el jugador, que componen una pantalla del juego. Almacena todas las oleadas de enemigos, sabe cuántas quedan, gestiona el jefe final que aparecerá al final de la pantalla, y se ocupa de situar y dibujar el fondo de pantalla.

A continuación, se van a explicar con más detalle cada uno de los apartados que gestiona la clase Fase.

5.5.1 Enemigos

Ya se vio en el apartado 5.3 como se han implementado los distintos enemigos que aparecen en el juego, por lo que no va a volver a explicarse nada al respecto.

La clase Fase tiene diversos atributos para crear y lanzar las oleadas de enemigos de la pantalla.

```
String[,] matEnemigos;
Texture2D texEnemigos;
int sigEnemigos;
```

Código 20: Fase: Atributos relacionados con los enemigos

El atributo "matEnemigos", es una matriz de cadenas de caracteres que contienen la codificación de cada enemigo. Cada fila se corresponde con una oleada, y cada segundo se lanza una nueva oleada. El segundo atributo contiene las texturas de los enemigos, y el último se ocupa de determinar cuál es la siguiente oleada que deberá aparecer.

Para crear la matriz de enemigos, es necesario pasar por parámetro al constructor de la clase fase el nombre del archivo que contiene los datos pertinentes.

```
Mountains.txt
1
60 8
000 000 000 565 565 000 000 000
000 000 000 000 000 000 000 000
563 000 000 000 000 000 000 560
000 000 000 000 000 000 000 000
000 210 000 210 000 210 000 211
211 000 210 000 210 000 210 000
000 210 000 210 000 210 000 210
```

Código 21: Archivo de enemigos

La codificación del archivo de enemigos es la siguiente:

- La primera línea indica el nombre del archivo que contiene el *scroll* de fondo.
- La segunda línea indica la codificación del jefe final de fase, que se explicará más adelante.
- La tercera línea contiene el tamaño de la matriz de enemigos, siendo el primer número las filas, es decir, la cantidad de oleadas de enemigos, y el segundo número las columnas, es decir, la cantidad de enemigos que aparecerán en cada oleada.
- A continuación, y hasta el final del fichero, se incluyen tantas filas como se ha indicado, con la correspondiente codificación de los enemigos.

Cada enemigo viene codificado por un número de tres cifras. Si se trata de un "000" indica que no hay enemigo en esa posición, en caso contrario la codificación es la siguiente:

- El primer número indica la textura del enemigo. Es un número comprendido entre 0 y 5.
- El segundo número indica el comportamiento del enemigo, dichos comportamientos están definidos en el apartado 5.3.1. Es un número comprendido entre 1 y 6.

- El tercer y último número indica si el enemigo dejará caer un *power up*, y cual, o no al morir. Es un número comprendido entre 0 y 5.

```
if (sigEnemigos >= 0)
{
      for (int j = 0; j < matEnemigos.GetLength(1); j++)
      {
            aux = matEnemigos[sigEnemigos, j];
            if (aux.CompareTo("000") != 0)
            {
               /* Textura del enemigo */
               tipo = int.Parse(aux[0].ToString());
               /* Comportamiento del enemigo */
               com = int.Parse(aux[1].ToString());
               /* Power Up que suelta al morir */
               capsula = int.Parse(aux[2].ToString());

               /* Comportamiento */
               switch (com)
               {
                 case 1:
                        comp = new Comportamiento1();
                        break;
                 case 2:
                        comp = new Comportamiento2();
                        break;
                 case 3:
                        comp = new Comportamiento3();
                        break;
                 case 4:
                        comp = new Comportamiento4();
                        break;
                 case 5:
                        comp = new Comportamiento5();
                        break;
                 case 6:
                        comp = new Comportamiento6();
                        break;
                 default:
                        comp = new Comportamiento1();
                        break;
               }
               ancho = j * 100 + 50;
               Enemigo nuevo = new Enemigo(texEnemigos, ancho,
-30, tipo, comp, capsula, pows);
               enemigos.Add(nuevo);
            }
      }
      sigEnemigos--;
}
```

Código 22: Fase: Creación de enemigos

Cada vez que se debe lanzar una nueva oleada, la fase crea los objetos de la clase Enemigo necesarios, y los envía a la clase principal para que se ocupe de su gestión dentro de la pantalla de juego.

Cuando no hay más oleadas, el método expuesto en la tabla de código 22 devolverá el valor correspondiente para que el control del juego sepa que, una vez desaparezcan todos los enemigos de la pantalla, deberá invocar al jefe final, si lo hay.

5.5.2 Jefe final

Un *shooter* en 2D debe tener variedad de enemigos, pero no estará completo hasta que no se incluyan jefes finales de fase. Dichos jefes son naves mucho más grandes, resistentes y letales que el resto de enemigos, y suponen un reto para el jugador.

Como se ha mencionado en el apartado anterior, el jefe final es invocado una vez han desaparecido todos los enemigos de la pantalla. Dicho jefe final se controla mediante la clase JefeFinal.

Su arquitectura es similar a lo de los enemigos, con la diferencia, además de la vida, la puntuación y la cantidad de disparos que realiza, de que su ciclo de ejecución es infinito. Esto significa que seguirá en pantalla, moviéndose y disparando hasta que sea destruido, o hasta que derrote al jugador.

Para el juego que nos ocupa, se han implementado dos jefes finales, cada uno con su propia rutina de ejecución.

Tipo 1.

El primer jefe final se asocia a un nivel de dificultad normal, y es más sencillo de derrotar que el segundo.

Figura 46. Jefe final Tipo 1

Una vez desciende hasta situarse en la parte superior de la pantalla, comienza su rutina de ejecución. El jefe final se desplazará de extremo a extremo de la pantalla, realizando dos disparos buscadores de forma periódica cada medio segundo, y un disparo en abanico, similar al que realiza el enemigo de tipo 6, cada segundo.

Tipo 2.

El segundo jefe final se asocia a un nivel de dificultad elevado, y supone un reto para el jugador.

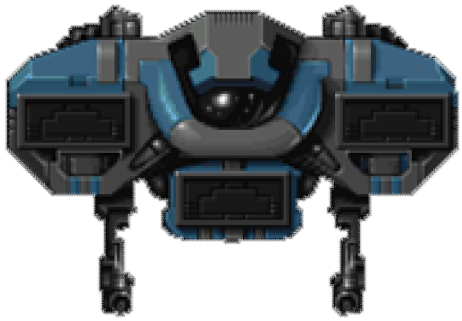


Figura 47. Jefe final Tipo 2

Una vez que desciende hasta situarse en la parte superior de la pantalla, comienza su rutina de ejecución. Este jefe final no tiene movimiento, se queda estático en la posición designada mientras realiza oleadas continuas de disparos.

Tres veces por segundo realiza dos disparos buscadores que irán en dirección al jugador, y una vez por segundo realiza tres disparos en abanico, desde las rejillas que se ven en la figura 47.

Aunque su resistencia es más elevada que la del primer tipo, su potencia de disparo es ligeramente inferior.

5.5.3 Scroll de fondo

La última parte que gestiona la clase Fase es el *scroll* de fondo. El *scroll* es el fondo de pantalla que se ve durante la partida y representa la superficie que están sobrevolando el jugador y los enemigos.

Sin embargo, el *scroll* no es una imagen estática que se va desplazando a medida que avanza la pantalla, sino que es un conjunto de pequeñas imágenes que, enlazadas, dan lugar a diversas formas y estructuras.

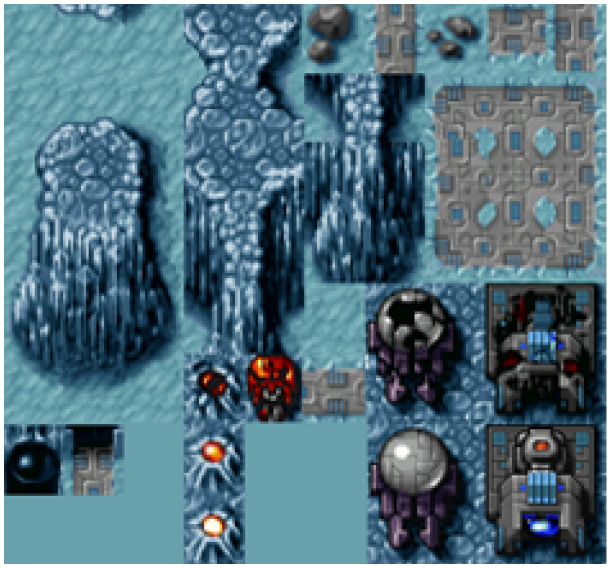


Figura 48. *Tileset* del juego

Llanuras, montañas, carreteras, edificios, etc. Muchos son los elementos que se pueden dibujar uniendo, y en ocasiones repitiendo, las imágenes que se muestran en la figura 48.

Por tanto, la gestión del *scroll* de fondo se corresponde con la gestión de un grupo de pequeñas imágenes que, en conjunto, forman una gran imagen que será la que se muestre durante la pantalla.

La clase Fase tiene diversos atributos para gestionar el fondo de pantalla.

```
Imagen[,] matFondo;
Texture2D fondo;
int iniFondo;
int finFondo;
int contFondo;
```

Código 23: Fase: Atributos relacionados con el scroll

Siendo matFondo la matriz de objetos Imagen que representará el *scroll*; fondo la textura que se corresponde con un *tiselet* similar al de la figura 48; y los últimos tres atributos, variables enteras que se utilizarán para determinar qué elementos hay que dibujar en cada momento.

La definición del *scroll* de fondo viene dada por el archivo al que se hace referencia en el archivo de enemigos, como se mostró en el apartado 5.5.1. La codificación del archivo que representa el fondo es la siguiente:

```
450 34
045 045 045 045 045 045 045 007 045 045 045 045 045 045
045 045 045 045 045 045 045 045 045 045 045 045 045 045
045 045 007 045 045 045 045 045 045 045 045 045 007 045
045 045 045 045 045 045 045 045 005 045 045 045 045 045
045 045 045 045 045 045 045 045 045 045 045 045 045 045
045 045 045 045 045 045 045 045 045 045 045 045 045 045
045 045 045 010 011 011 011 011 011 011 011 011 011 011
011 011 011 011 011 011 011 011 011 011 011 011 011 011
```

Código 24: Archivo de *scroll*

En primer lugar tenemos el tamaño de la matriz de objetos de clase Imagen que representarán el fondo, siendo el primer número las filas, y el segundo las columnas. A continuación se incluyen tantas filas como se haya especificado, cada una compuesta por una serie de números.

La codificación de las imágenes es sencilla, y se explicará mejor teniendo antes una idea de los atributos de la clase Imagen.

```
float ancho;
float alto;
int coorx;
int coory;
const int X_TAMANO = 24;
const int Y_TAMANO = 28;
const int FILES_TILESET = 7;
const int COLUMNS_TILESET = 10;
```

Código 25: Fase: Atributos relacionados con el *scroll*

Las variables ancho y alto sirven para ubicar la imagen en pantalla, las variables coorx y coory indican la codificación y sirven para determinar qué imagen del *tileset* hay que dibujar, y las constantes X_TAMANO e Y_TAMANO definen el tamaño de las imágenes del *tiselet*.

La parte importante para entender la codificación son las dos últimas variables, que representan el tamaño, en imágenes pequeñas, del *tiselet* que se está utilizando. Sabiendo el número de filas y el número de columnas, es fácil, a partir de un número entero, determinar qué imagen se quiere dibujar. Siendo 0 la primera imagen, situada arriba a la izquierda, 1 la de su derecha, y así sucesivamente.

Por tanto, tomando como ejemplo la figura 48, la codificación relativa al número 9 sería la imagen situada en la parte superior derecha de la imagen.

Figura 49. Imagen correspondiente a la codificación “9”

Una vez explicado cómo se construye un *tileset*, y como se codifica el archivo que lo representa, es hora de explicar cómo la clase Fase gestiona la imagen completa para mostrarla de forma fluida durante el transcurso de la partida.

Una primera aproximación consistía en posicionar todas las imágenes, desde la primera hasta la última en orden, la mayoría quedaría fuera de la pantalla, y hacer avanzar la imagen en cada iteración. Un método práctico y efectivo, pero que resulta costoso si tratamos con *scrolls* de fondo muy grandes, debido a que la actualización del *scroll* de fondo se hace en el método *Update*, es decir, sesenta veces por segundo.

Posteriormente se pensó en cómo trabajar con la menor cantidad de imágenes posibles, y este camino llevó a la solución final.

En una pantalla de 800x600 entran unas 34 imágenes a lo ancho y 22 a lo alto, cada una de ellas de 24x28 que es el tamaño de las imágenes del *tiselet* utilizado. Por tanto, se debe trabajar con al menos 22 filas para mostrar el fondo. Teniendo en cuenta que avanza de forma constante, hay que trabajar con dos filas más, dado que una será la que esté desapareciendo por la parte inferior, mientras que otra estará apareciendo por la parte superior.

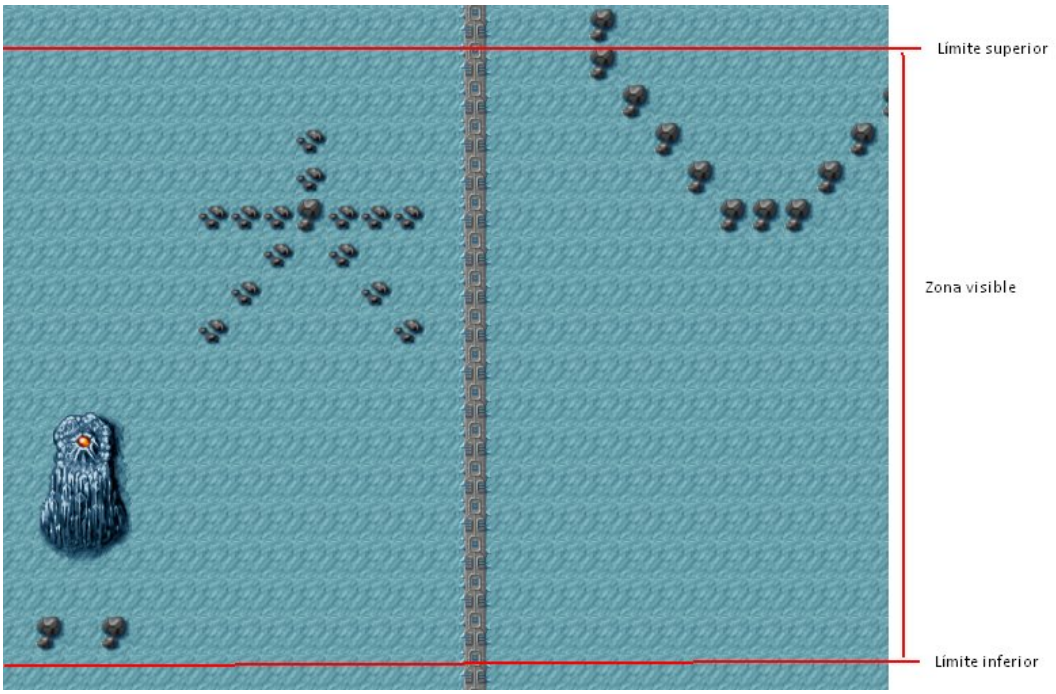


Figura 50. Porción de *scroll* con la que se trabaja

En la figura 50 se puede ver un ejemplo del rango de filas con el que trabaja el juego. Se trabaja en todo momento con las filas que deben aparecer en la pantalla de juego, incluida la que está desapareciendo, por debajo del límite inferior; y la que está apareciendo por encima del límite superior. Además, se trabaja con una fila adicional en la parte superior.

De esta manera, en el momento en el que una fila termine de desaparecer por debajo del límite inferior, la primera fila de la parte superior habrá hecho aparición, y habrá otra por encima del límite superior lista para mostrarse. Se podría decir que el programa tiene las imágenes enrolladas, y va desenrollándolas, y cortando las que sobran, a medida que la pantalla avanza.

```
public void dibujarFondo(SpriteBatch spriteBatch,
int alphaValue, int speed)
{
      for (int i = iniFondo; i > finFondo; i--)
      {
            for (int j = 0; j < matFondo.GetLength(1); j++)
            {
                matFondo[i, j].dibujar(spriteBatch, fondo,
alphaValue, speed);
            }
      }
      if (speed > 0)
      {
          contFondo++;
          if (contFondo == 28)
          {
              contFondo = 0;
              iniFondo--;
              finFondo--;
          }
      }
}
```

Código 26: Método que controla el dibujo del fondo

Todo esto lo conseguimos utilizando las variables “iniFondo”, “finFondo” y “contFondo” mostradas en la tabla de código 23. Las dos primeras variables indican el rango de filas de la matriz de imágenes que se deben dibujar y mover en cada iteración. La variable “contFondo” sirve para determinar cuando una fila ha desaparecido y hay que dar paso a la siguiente.

```
public void dibujar(SpriteBatch spriteBatch, Texture2D fondo,
int alphaValue, int speed)
{
      spriteBatch.Draw(fondo, new Vector2(ancho, alto),
new Rectangle(coorx * X_TAMANO, coory * Y_TAMANO, X_TAMANO,
Y_TAMANO),
new Color(255, 255, 255, (byte)MathHelper.Clamp(alphaValue, 0,
255)),
MathHelper.ToRadians(0), new Vector2(0, 0), 1.0f,
SpriteEffects.None, 0);

      alto += speed;
}
```

Código 27: Método que dibuja una de las imágenes del fondo

De esta manera, con cada iteración hacemos avanzar píxel a píxel las imágenes y, cada 28 iteraciones, pues ese es el alto de las imágenes, se avanza a la siguiente fila.

Como último apunte acerca del tratamiento del *scroll* de fondo hay que añadir que, una vez se acaban los enemigos y aparece el jefe final, el *scroll* de fondo se detiene, y permanecerá en la misma posición hasta que termine la pantalla, o el jefe final destruya al jugador.

5.5.4 Pantallas de final de fase

Se tiene implementado al jugador, los enemigos, las colisiones, y la fase. Pero falta por explicar las dos posibles formas de terminar una pantalla.

La primera forma de terminar una fase es cuando el jugador pierde todos sus puntos de vida y su nave es destruida. En ese momento se muestra la pantalla de final de juego y se le da la opción de reiniciar la partida o, si lo desea, volver al menú principal.

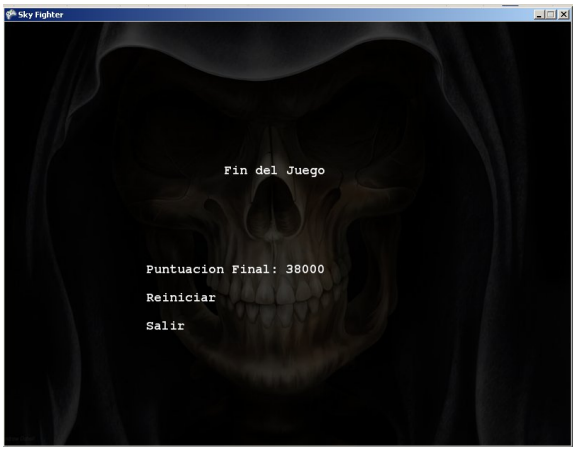


Figura 51. Pantalla de Fin del Juego

La segunda opción consiste en superar la fase. Esto ocurre después de vencer al jefe final o, si éste no existe, de resistir todas las oleadas de enemigos. La pantalla de misión cumplida puede dar dos opciones:

- Si se ha superado la primera fase que trae el juego, se tiene la opción de continuar a la siguiente.
- Si se ha superado la segunda fase que trae el juego, o cualquier fase creada por el editor de niveles, se dispone de la opción de volver al menú principal.

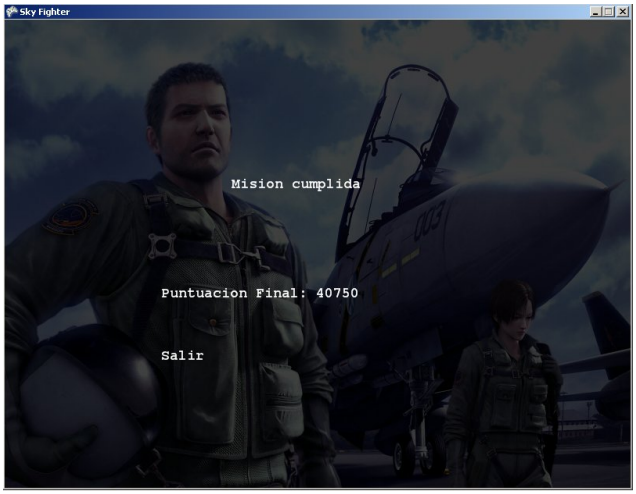


Figura 52. Pantalla de Misión Cumplida

5.6 Editor de niveles

Una de las características más destacadas del juego durante el proceso de desarrollo era la inclusión de un editor de niveles que permitiera al usuario generar cientos de pantallas, de forma que la duración del juego fuera infinita.

En este apartado se van a comentar los aspectos más relevantes, así como el aspecto, de los editores de niveles que se han desarrollado.

5.6.1 Editor simple

El editor simple consta de un único formulario en el que hay que introducir los siguientes datos:

- Duración del nivel. Un valor comprendido entre 30 segundos y 2 minutos y 30 segundos, en intervalos de medio minuto.
- Fondo, uno de los dos disponibles.

- Dificultad, que puede ser fácil, normal o difícil.

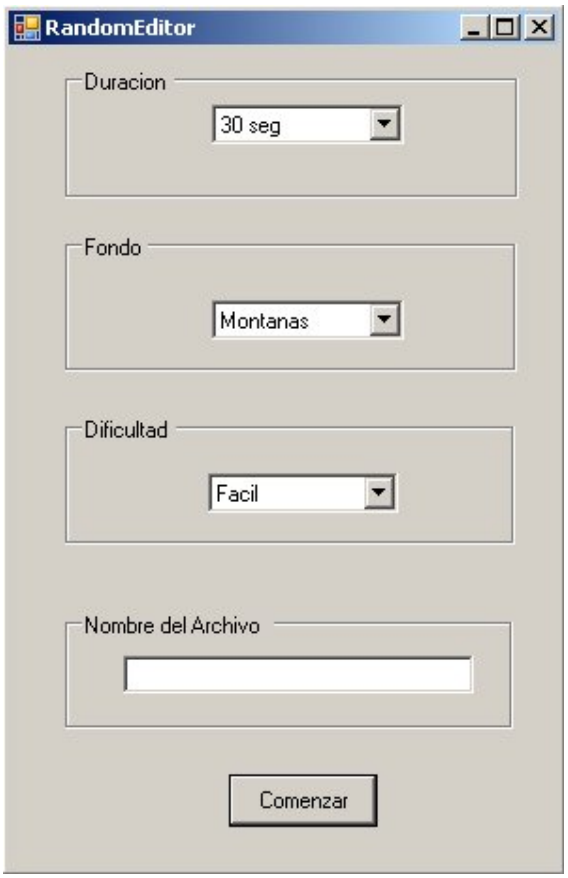


Figura 53. Editor simple: RandomEditor

En el caso de que el nombre del archivo sea igual a otro ya existente, el editor avisará al usuario y le dará la opción de cambiarlo, o de sobrescribirlo.

La dificultad es el parámetro que determina las características del nivel. Un nivel de dificultad fácil hace que se cree un nivel con las siguientes características:

- De cada 2 oleadas, una estará vacía, es decir, no contendrá ningún enemigo.
- En las oleadas que contienen enemigos habrá de 1 a 2 enemigos, con 1 *power up*.
- El jefe final será el de Tipo 1, el más sencillo.

Un nivel de dificultad normal hace que se cree un nivel con las siguientes características:

- De cada 3 oleadas, una estará vacía, es decir, no contendrá ningún enemigo.
- En las oleadas que contienen enemigos habrá de 2 a 4 enemigos, con la posibilidad de que al menos 1 contenga un *power up*.

- El jefe final se decidirá aleatoriamente.

Por último, seleccionar el nivel de dificultad difícil provoca que se cree un nivel con las siguientes características:

- Todas las oleadas contienen enemigos.
- En las oleadas que contienen enemigos, habrá de 4 a 8 enemigos, con la posibilidad de que hasta 2 de ellos dejen caer un *power up* al morir.
- El jefe final será el de Tipo 2, el más complicado.

5.6.2 Editor avanzado

El editor avanzado da libertad total para configurar los enemigos y sus características. En primer lugar se pide al usuario que introduzca la duración, el fondo y el nombre del archivo.

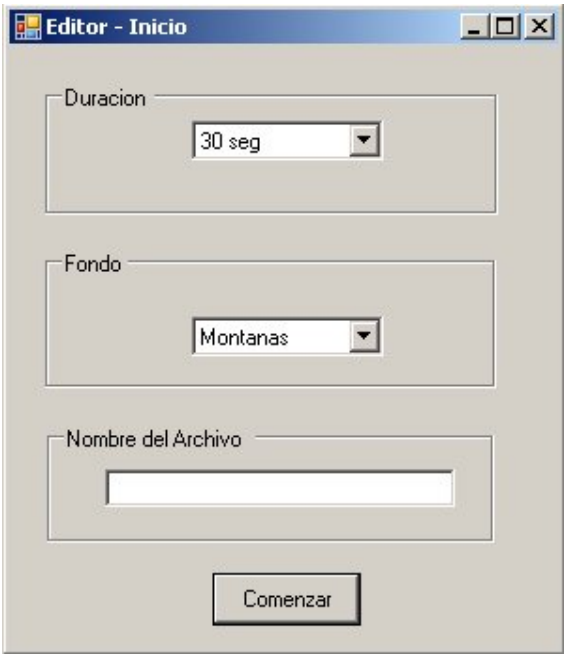


Figura 54. Editor avanzado: Intro

En función de la duración seleccionada, la siguiente ventana tendrá una fila de botones por cada segundo. De esta forma, el siguiente formulario tendrá el mismo aspecto que el archivo de enemigos.

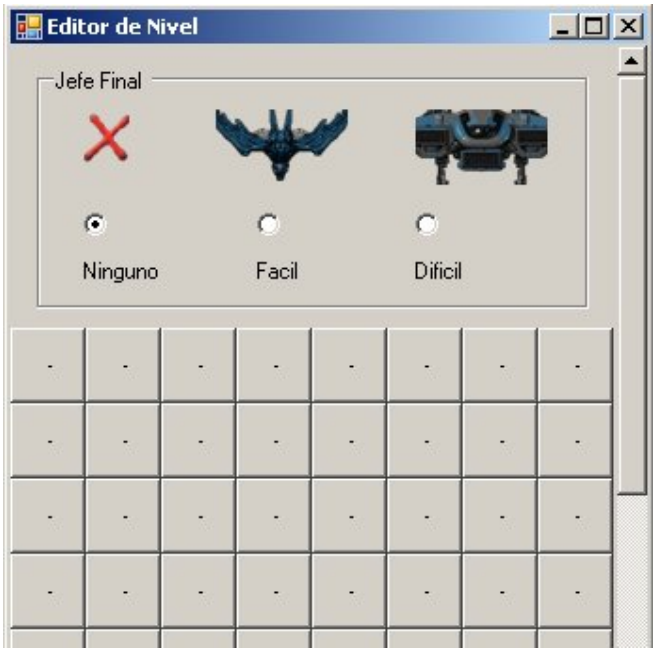


Figura 55. Editor avanzado: Parte superior de Editor

Además en la parte superior se incluye la elección del jefe final, y en la parte inferior se incluyen los botones de “Guardar y salir” y “Salir sin guardar”.

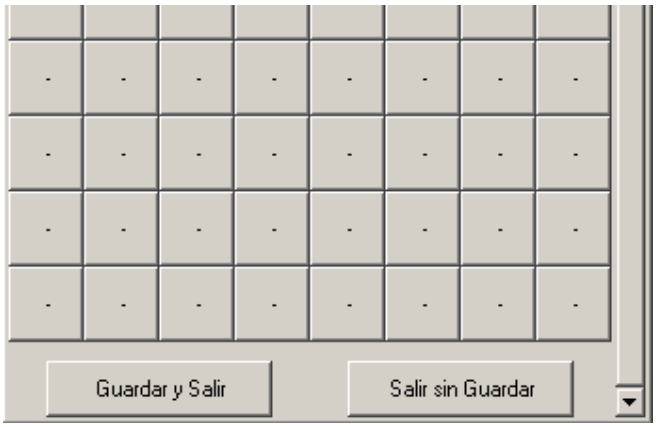


Figura 56. Editor avanzado: Parte inferior de Editor

Por defecto, todos los botones indican que no hay ningún enemigo en su posición. Al pulsar en cualquiera de los botones que llevan un guión como texto, se abre el formulario Enemy, que permite configurar el *sprite* del enemigo, el comportamiento y el *power up* que dejará caer si es destruido.

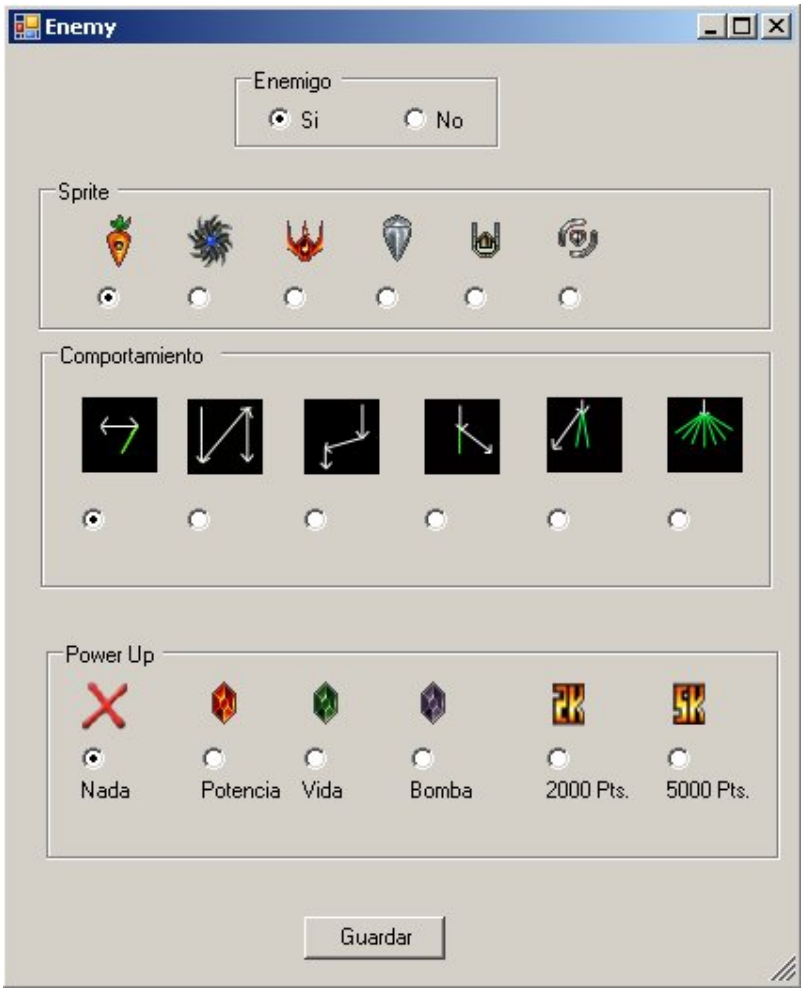


Figura 57. Editor avanzado: Enemy

Como se ha comentado anteriormente, el usuario dispone de plena libertad para configurar el nivel a su gusto.

La inclusión de los dos editores permite que tanto los usuarios pacientes y con ganas de crearse sus niveles, como los usuarios que quieran acción directa, puedan disfrutar de nuevas fases siempre que quieran.

5.7 Música y efectos de sonido

Un videojuego no estaría completo sin su banda sonora. Música de fondo, o efectos de sonido que se escuchan cuando se producen diversos eventos, son indispensables para que la inmersión en el juego sea mayor.

En este apartado veremos como incluir estos elementos en el juego resulta realmente sencillo gracias a los recursos proporcionados por XNA [62].

5.7.1 Música

La música de fondo puede tener varios formatos, en este caso se han elegido archivos con extensión .mp3.

```
introMusic = Content.Load<Song>("Intro");
stageMusic = Content.Load<Song>("Stage");
finalBattle = Content.Load<Song>("FinalBattle");
failedMusic = Content.Load<Song>("Failed");
victoryMusic = Content.Load<Song>("Victory");
```

Código 28: Música: Carga de canciones

Antes de cargar el elemento en el método *LoadContent*, es importante indicar, al incluir el archivo en el proyecto, que se procese como un objeto de tipo *Song*. Una vez cargada la canción, se pueden modificar diversos parámetros del reproductor.

```
MediaPlayer.IsRepeating = true;
MediaPlayer.Volume = volumen * 0.01f;
```

Código 29: Música: Parámetros del *MediaPlayer*

La primera línea de la tabla de código 29 indica que la canción volverá a repetirse cuando llegue al final. La segunda es el ajuste del volumen, que va desde el valor 0.0 hasta el valor 1.0. Dicho valor podrá ser modificado en el menú de configuración del juego.

```
MediaPlayer.Stop();
MediaPlayer.Play(stageMusic);
```

Código 30: Música: Detener una canción y reproducir la siguiente

La sentencia Play inicia la reproducción de la canción elegida. Si se está ejecutando una canción y se quiere cambiar a otra, es necesario utilizar antes la directiva *Stop*.

5.7.2 Efectos de sonido

El archivo que contiene el efecto de sonido que se quiere incluir debe tener la extensión .wav.

```
xplosion = Content.Load<SoundEffect>("ExploMuerte");
bomb = Content.Load<SoundEffect>("Bomba");
powerup = Content.Load<SoundEffect>("powerup");
hit = Content.Load<SoundEffect>("hit");
```

Código 31: Carga de efectos de sonido

Antes de cargar el elemento en el método *LoadContent*, es importante indicar, al incluir el archivo en el proyecto, que se procese como un objeto de tipo *SoundEffect*.

```
xplosion.Play();
```

Código 32: Reproducción de un efecto de sonido

Una vez cargado el efecto, se puede reproducir en cualquier momento con la directiva Play. Es importante destacar que la reproducción de un efecto de sonido es independiente de la reproducción de la música de fondo. Esto es, el sonido se escuchará por encima de la música sin que ésta se detenga.

En el menú de configuración es posible activar y desactivar los efectos de sonido del juego.

CAPÍTULO 6: CONCLUSIONES

Una vez concluido el desarrollo, se analizarán los resultados en función de los objetivos establecidos. Por tanto, en este capítulo se comentan las conclusiones extraídas del desarrollo del proyecto.

6. CONCLUSIONES

En este proyecto se ha llevado a cabo la realización de un *shooter* de naves en 2D para PC, utilizando la herramienta XNA Game Studio 3.1. El videojuego cuenta con todas las funcionalidades que se establecieron en la fase de análisis, y se puede afirmar que tiene un nivel de jugabilidad aceptable. El aspecto más destacado, los editores de niveles, cumple su función tanto para jugadores que busquen acción directa en pocos segundos, como para usuarios pacientes que quieran programar sus niveles.

Desarrollar este videojuego ha resultado ser una tarea diferente con respecto a la mayoría de las prácticas de la carrera. Al tratar con un videojuego, no sólo hay que tener nociones de diseño y programación, sino que también hay que ser capaz de crear los distintos componentes que dan vida a un juego como son los gráficos o el sonido. En cuanto a la programación, algunas partes han resultado más complicadas debido a que en un videojuego hay que cuidar hasta el más mínimo detalle para que no se produzcan errores o situaciones inverosímiles. Por el contrario, resulta más satisfactorio desarrollar un producto en el que se ve con más claridad la evolución, desde la primera nave, sin animación, que se movía sobre un fondo negro hasta el juego completo, y los resultados intermedios a lo largo del desarrollo.

Otro aspecto destacable es que, en un proyecto de estas características, hay que saber cuando decir “Basta”. Es fácil empezar a desarrollar con unas especificaciones y, a medida que avanza el proyecto, añadir más y más funcionalidades al juego. El desarrollo se puede alargar más de lo esperado si no se sabe cuando parar, porque siempre hay alguna idea más que introducir en el juego.

Finalmente, como se comentó en el apartado 2.3, XNA ha resultado ser una buena elección gracias a la cantidad de comunidades, foros, y tutoriales disponibles. Desde los primeros pasos, conociendo XNA, hasta las partes más complicadas del desarrollo, se ha podido disponer de muchos recursos para llevar el proyecto a buen puerto.

En lo personal, este proyecto me ha permitido acercarme al mundo del desarrollo de videojuegos y comprender mejor todas y cada una de las fases que lo componen, y a entender las diferencias que hay entre desarrollar una aplicación común y un videojuego. Espero que esta experiencia me sea útil de cara a futuros proyectos.

CAPÍTULO 7: LÍNEAS FUTURAS

Aunque el videojuego cuenta con las funcionalidades que se establecieron en un primer momento, además de con una jugabilidad aceptable, todo producto se puede mejorar. En este capítulo se comentan las posibles mejoras que podrían incluirse en el videojuego.

7. LÍNEAS FUTURAS

Todo proyecto tiene unos plazos y debe marcar unos límites. Hay multitud de opciones y mejoras que se pueden incluir en un juego, pero llega un momento en el que hay que decir "hasta aquí vamos a llegar" para poder cerrar el proyecto y no estar añadiendo mejoras cada día. En este punto se incluyen algunas ideas que se podrían incluir tomando este proyecto como base.

En primer lugar, surge la idea de aumentar el número de elementos que componen el juego. Esto es, añadir más naves para que el jugador tenga más variedad a la hora de elegir, incluir nuevos comportamientos y *sprites* para los enemigos, y aumentar la plantilla de jefes finales.

También se podría abordar la inclusión de un modo multijugador. La posibilidad de que dos jugadores estén en pantalla simultáneamente y cooperen para superar la fase. Si a esto se le añadiera un ajuste de dificultad en los niveles en función de si hay un jugador o dos, la mejora sería sustancial.

Las siguientes dos mejoras requerirían de un servidor que estuviera activo las 24 horas del día, al que se podría acceder desde el juego y que permitiera almacenar los siguientes elementos:

- En primer lugar, un marcador de puntuación para cada nivel, con una clasificación de los mejores jugadores en cada uno de los niveles. En estos juegos conseguir la puntuación máxima supone un reto, y la naturaleza competitiva del ser humano haría que muchos quisieran subir a lo más alto de la clasificación general.

- La segunda mejora consistiría en la posibilidad de acceder al servidor desde el juego para subir los niveles creados por el usuario y compartirlos con el resto de la comunidad. Además de subir niveles, se podrían descargar los niveles creados por los demás usuarios. De esta manera, no sólo se ampliaría la experiencia de juego en función de los niveles que crea el usuario, sino que tendría a su disposición miles de fases creadas por usuarios de todo el mundo.

La última mejora, sólo apta para los usuarios más dedicados, y que supondría ampliar una faceta ya desarrollada y no implementar algo desde cero, consistiría en ampliar el editor de niveles y permitir configurar todos los aspectos de la fase. Es decir, permitir al usuario que pueda configurar hasta el *scroll* de fondo como mejor le parezca. Sería una mejora interesante, aunque posiblemente utilizada por una menor cantidad de usuarios.

Anexo A: MANUAL DE USUARIO

En este primer anexo se incluye el manual de usuario que acompañaría a cada copia del juego, para poner a disposición del jugador los conocimientos necesarios para utilizar correctamente el juego y sus características.

A.1 Instalación

Para instalar el juego hay que hacer doble clic en el archivo *setup.exe*, y seguir los pasos del instalador.

Para ejecutar el juego, hay que acceder a la carpeta donde fue instalado y ejecutar el archivo *SkyFighter.exe*.

Figura 58. Icono "SkyFighter.exe"

El juego puede ser instalado en los sistemas operativos Windows XP / Vista / 7, sin necesidad de disponer de ningún software adicional.

A.2 Controles

Se puede controlar el juego tanto con el ratón, como con el teclado. Basta con elegir uno u otro método de control en el menú de Controles, explicado más adelante. El controlador establecido por defecto es el ratón.

Ratón

Pantalla de menú / Pantalla de Misión Cumplida / Pantalla de Fin del Juego:

- Botón izquierdo: seleccionar la opción marcada por el cursor.

Pantalla de juego:

- Botón izquierdo: Disparo.
- Botón derecho: Bomba.
- Desplazar el ratón: Mover la nave en la dirección indicada.
- Tecla P: Pausar / Reanudar el juego.

Teclado

Pantalla de menú / Pantalla de Misión Cumplida / Pantalla de Fin del Juego:

- Arriba: Desplazar el icono de selección hacia arriba.
- Abajo: Desplazar el icono de selección hacia abajo.
- Intro: Seleccionar opción.
- Derecha: En el Manual, ver la siguiente página. En el menú de configuración, aumentar el volumen de la música. En el menú de selección de nave, elegir la siguiente nave.
- Izquierda: En el Manual, ver la página anterior. En el menú de configuración, reducir el volumen de la música. En el menú de selección de nave, elegir la nave anterior.

Figura 59. Icono de selección

Pantalla de juego:

- Arriba: Mover la nave hacia arriba.
- Abajo: Mover la nave hacia abajo.
- Derecha: Mover la nave hacia la derecha.
- Izquierda: Mover la nave hacia la izquierda.
- Espacio: Disparar.
- Z: Bomba.
- P: Pausar / Reanudar el juego.

A.3 Pantallas de Menú

Al iniciar el juego, se mostrará el menú principal. Existen varios menús por los que se pueden navegar. A continuación se enumerarán los distintos menús y sus funcionalidades.

Menú Principal.

- Nueva Partida: Permite iniciar una partida con las fases predefinidas del juego.
- Editor de Niveles: Accede al menú de edición de niveles.
- Elegir Nave: Accede al menú para seleccionar la nave que se quiere utilizar.
- Ayuda y Opciones: Accede al menú de ayuda y opciones.
- Salir del Juego: Finaliza la ejecución y cierra el juego.

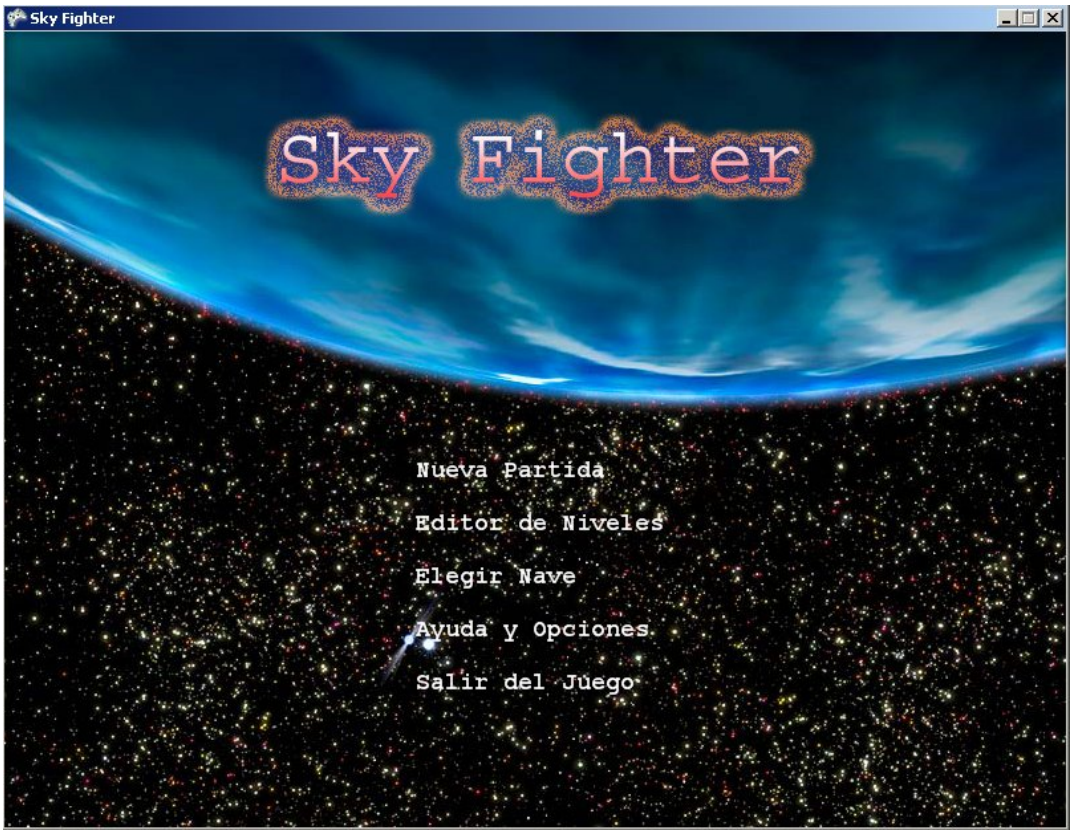


Figura 60. Menú Principal

Editor de Niveles.

Este menú permite acceder a las siguientes características:

- Editor simple. Abre el editor simple para crear una pantalla de forma rápida y sencilla. Las instrucciones para manejar el editor simple se pueden encontrar en el apartado “A.8 Editor de niveles”.
- Editor avanzado: Abre el editor avanzado para crear una pantalla, configurando todos los enemigos que saldrán en ella. Las instrucciones para manejar el editor avanzado se pueden encontrar en el apartado “A.8 Editor de niveles”.
- Niveles creados: Accede al menú Niveles Creados.

Figura 61. Menú: Editor de Niveles

Niveles Creados.

Este menú permite navegar entre los distintos niveles creados para elegir el que se desea jugar. Pulsando la opción "Jugar" se inicia la fase seleccionada. En el caso de que no exista ningún nivel personalizado, la opción "Jugar" estará deshabilitada.

Figura 62. Menú: Niveles Creados

Elegir Nave.

Este menú permite seleccionar la nave que se quiere utilizar entre las tres que se encuentran disponibles. En el apartado "A.5 Naves" se detallan las características de cada una.

Figura 63. Menú: Elegir Nave

Ayuda y Opciones.

El menú de Ayuda y Opciones da acceso a las siguientes opciones:

- Como se juega: Incluye una guía rápida de dos páginas para aprender a jugar.
- Controles: Accede al menú de Controles.
- Configuración: Accede al menú de Configuración.

Controles.

El menú controles permite alternar entre los distintos periféricos que se han implementado para controlar el juego: teclado y ratón. Además, indica los controles para disparar y lanzar bombas del periférico seleccionado.

Configuración.

El menú de configuración permite cambiar el valor de varios parámetros que afectan al juego.

- Música: Permite ajustar el volumen de la música entre 0 y 100, en intervalos de 10.
- Efectos de sonido: Permite activar o desactivar los efectos de sonido.
- Idioma: Permite alternar entre el idioma español e inglés.
- Predeterminado: Restaura los parámetros anteriores a los valores por defecto.

A.4 Pantalla de juego

A continuación se muestra la pantalla de juego, resaltando la información importante que se muestra en durante la partida.

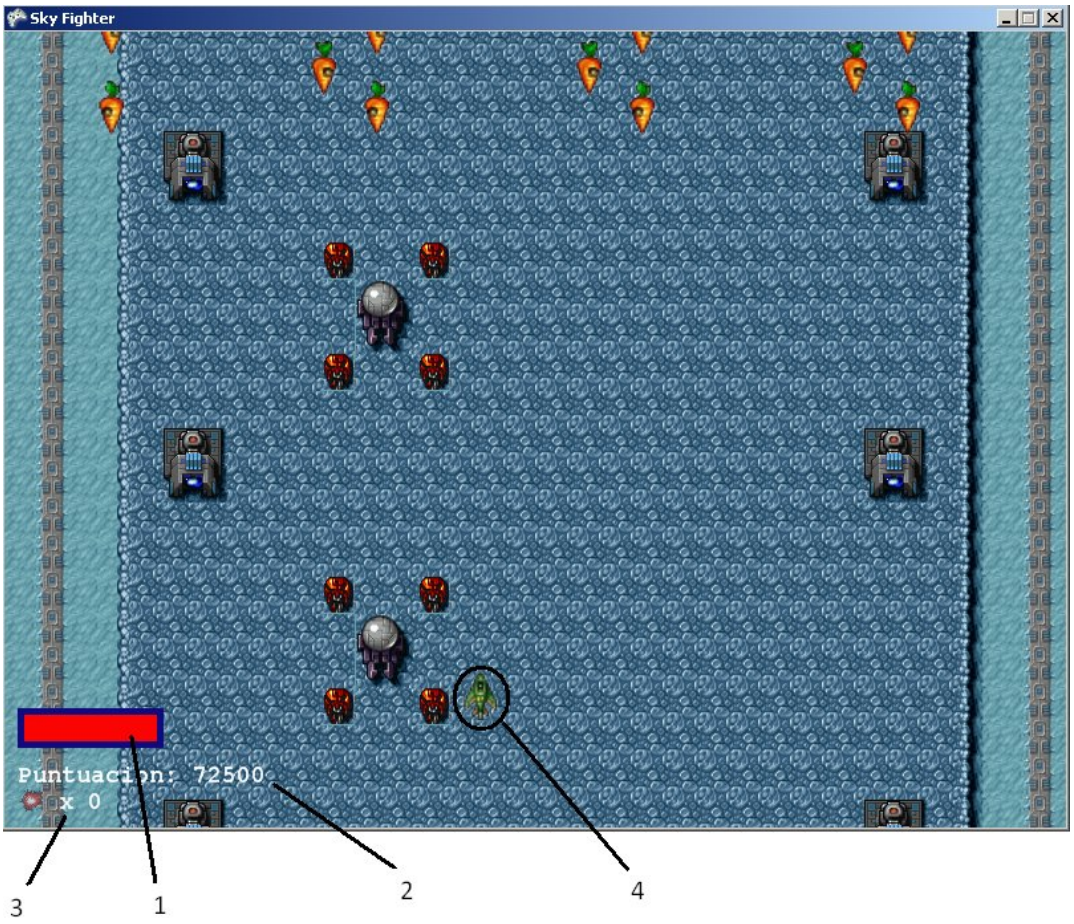


Figura 64. Pantalla de juego

- 1: Barra de vida.
- 2: Puntuación actual.
- 3: Número de bombas.
- 4: Jugador.

Si la vida del jugador llega a 0 durante el transcurso de la fase, la partida finalizará y se mostrará la pantalla de Fin del Juego, junto con la puntuación obtenida.

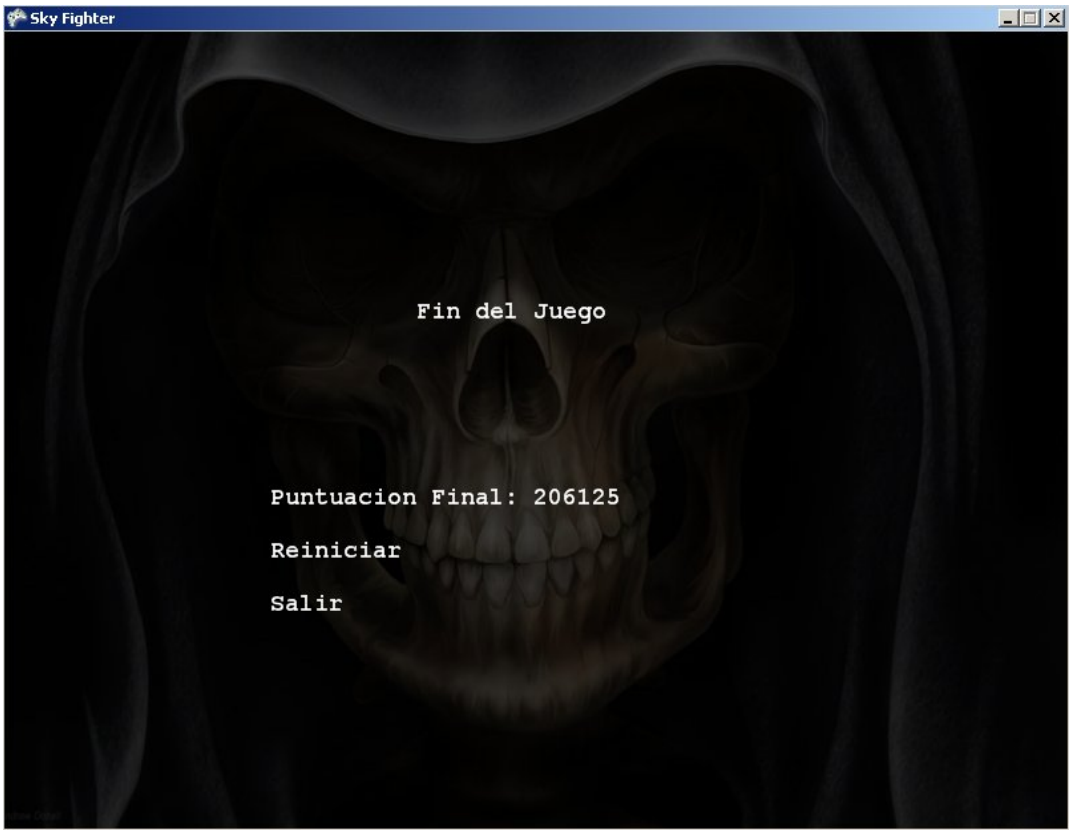


Figura 65. Pantalla de juego: Fin del Juego

Si se elige la opción “Reiniciar” se volverá a iniciar la fase no superada desde el principio. Por el contrario, si se elige la opción “Salir”, se volverá al menú principal.

Si el jugador derrota a todos los enemigos y al jefe final, si hay, superará la fase y se mostrará la pantalla de Misión Cumplida, junto con la puntuación obtenida.

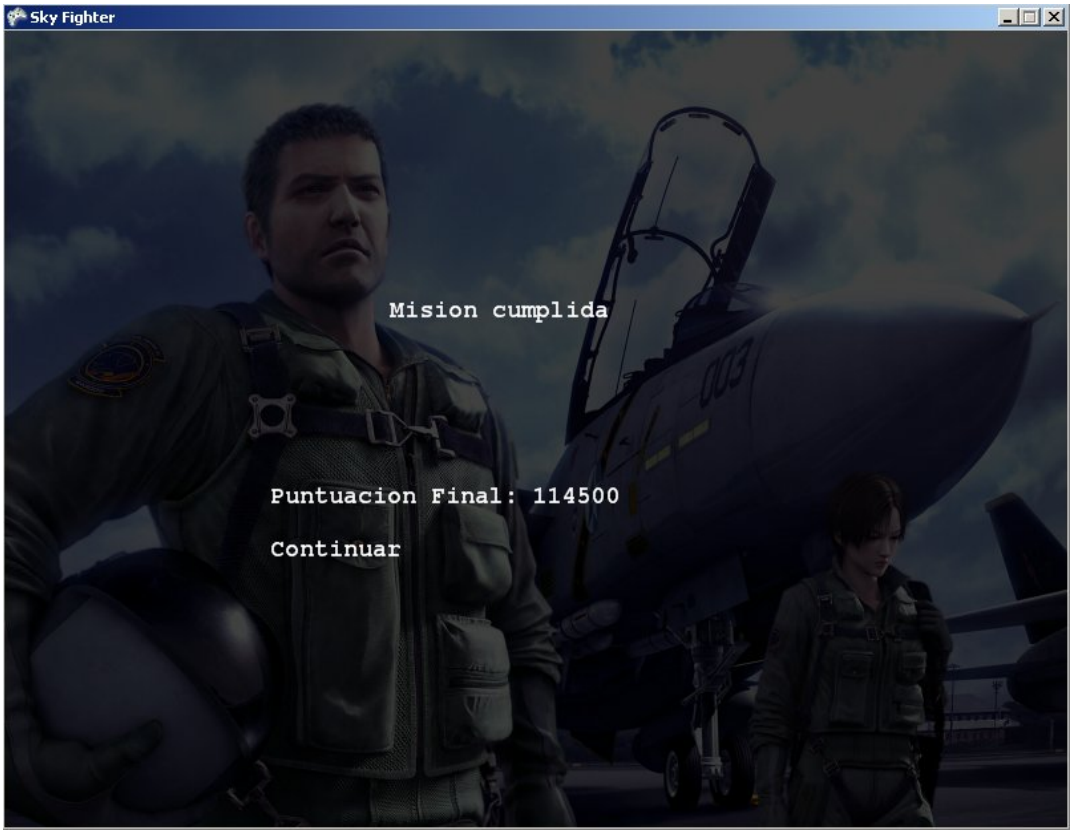


Figura 66. Pantalla de juego: Misión Cumplida

Si la fase superada es la primera del juego, la opción presente será “Continuar”, que permitirá avanzar a la siguiente fase. En caso contrario, la opción presente será “Salir”, que hará que el usuario regrese al menú principal.

A.5 Naves

Hay 3 naves disponibles en el juego. Cada una de ellas tiene sus propias características, las cuales se enumeran en la tabla siguiente:

	Tipo 1	Tipo 2	Tipo 3
Sprite			
Número de disparos	1	2	3
Potencia base	12	10	8
Incremento de potencia	6	4	2
Vida	100	100	100
Número inicial de bombas	2	1	1

Tabla 108: Comparativa de Naves

Al reiniciar una pantalla después de morir, se reinicia la potencia de disparo, la vida y el número de bombas de la nave. También se resetea el contador de puntos.

Al avanzar de la primera pantalla a la segunda, se conserva la potencia de disparo, la vida y las bombas. También se conserva la puntuación.

A.6 Enemigos

A lo largo de las fases pueden aparecer distintos tipos de enemigos. Cada uno de ellos tiene asociada una imagen y un comportamiento. El comportamiento es lo que define el resto de características del enemigo. Las características de los enemigos se enumeran en la siguiente tabla:

NOTA: En el esquema de comportamiento, la flecha blanca indica movimiento de la nave y la flecha verde, disparo.

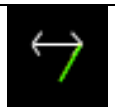
	Tipo 1	Tipo 2	Tipo 3	Tipo 4	Tipo 5	Tipo 6
Sprite						
Esquema de comportamiento						
Vida	25	40	25	15	25	50
Potencia de disparo	10	0	0	5	10	20
Puntos	2.000	1.000	1.000	500	2.500	4.000

Tabla 109: Comparativa de Enemigos

Aunque lo indicado en la tabla sea la imagen asociada por defecto a cada comportamiento, es posible que dichas imágenes y comportamientos se combinen de forma que un enemigo con el primer *sprite*, no siempre tiene porque comportarse como el tipo 1.

Además de los enemigos normales, existen dos jefes finales que pueden aparecer al final de la fase y suponen un reto para el jugador. Sus características son las siguientes:

	Tipo 1	Tipo 2
Sprite		
Vida	2.000	4.000
Potencia de disparo	25	10
Puntos	10.000	20.000

Tabla 110: Comparativa de Jefes Finales

A.7 Power ups

Durante el transcurso del juego es posible que los enemigos derrotados dejen caer unas cápsulas que contienen ventajas para el usuario. Los *power ups* que se pueden obtener, y su funcionalidad, son los siguientes:

-  : Aumenta la potencia de disparo.
-  : Recupera 10 puntos de vida. La vida no puede aumentar por encima de 100.
-  : Aumenta en 1 unidad el número de bombas disponibles.
-  : Proporciona 2.000 puntos.
-  : Proporciona 5.000 puntos.

A.8 Editor de niveles

Sky Fighter no termina cuando se superan todas las fases. El juego trae dos editores de niveles para prolongar la experiencia de juego de forma indefinida.

Editor Simple.

El editor simple permite generar una pantalla en cuestión de segundos. Sólo hay que elegir las opciones deseadas y ponerle un nombre a la fase.

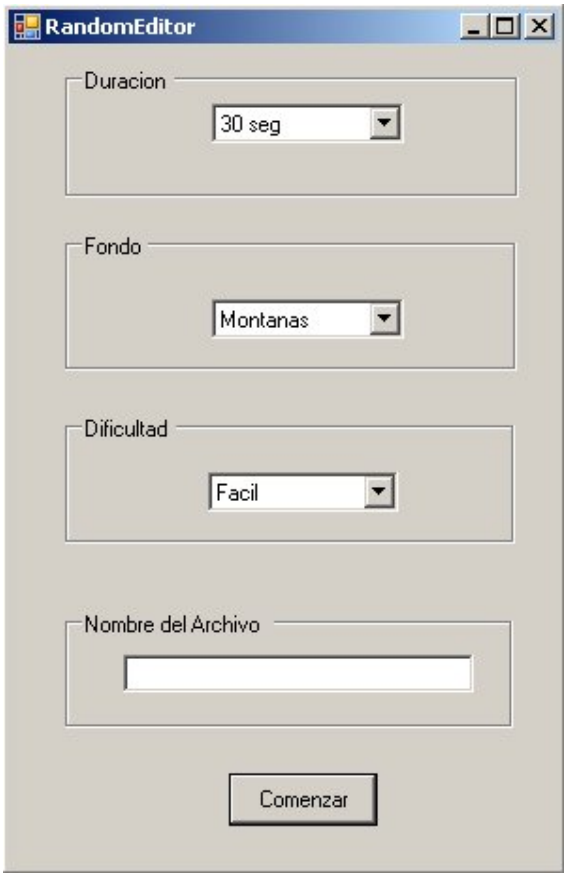


Figura 67. Editor Simple

Los *scrolls* de fondo disponibles son:

- Montañas.
- Ciudad.

Los posibles valores de duración son:

- 30 segundos.
- 1 minuto.
- 1 minuto y 30 segundos.
- 2 minutos.
- 2 minutos y 30 segundos.

La dificultad seleccionada puede ser:

- Fácil.
- Normal.
- Difícil.

El nombre del archivo sólo puede estar compuesto por letras y dígitos.

Editor Avanzado.

El editor avanzado proporciona total libertad para configurar las oleadas de enemigos, incluyendo el *sprite*, el comportamiento y los *power ups* que dejarán caer. Además permite elegir el jefe final de fase.

En primer lugar hay que rellenar una serie de campos:

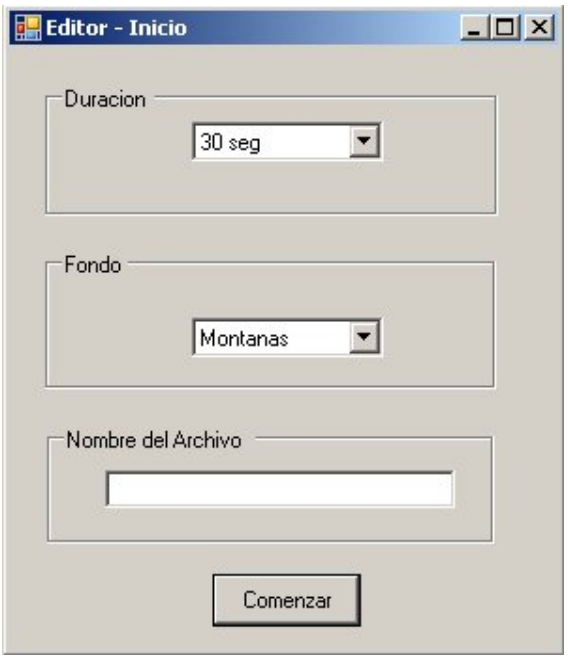


Figura 68. Editor avanzado: Inicio

Los posibles valores de duración son:

- 30 segundos.
- 1 minuto.
- 1 minuto y 30 segundos.
- 2 minutos.
- 2 minutos y 30 segundos.

Los *scrolls* de fondo disponibles son:

- Montañas.
- Ciudad.

El nombre del archivo sólo puede estar compuesto por letras y dígitos.

Una vez rellenados correctamente los campos y habiendo pulsado el botón "Comenzar", se abrirá una nueva ventana. El aspecto que tendrá es el siguiente:

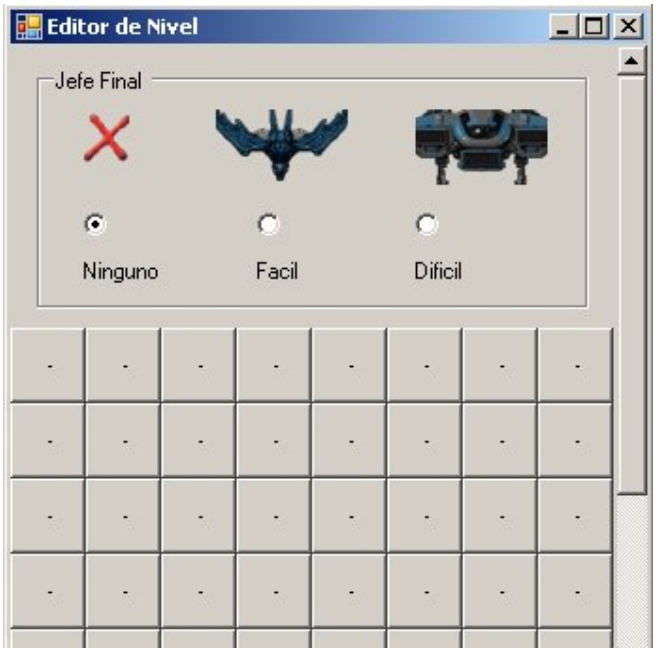


Figura 69. Editor avanzado: Configuración 1

Figura 70. Editor avanzado: Configuración 2

La parte superior permite elegir entre no incluir jefe final, o incluir cualquiera de los dos disponibles. Basta con seleccionar la opción apropiada. La parte central está compuesta por una fila de botones por cada oleada enemiga que habrá en la pantalla.

Pulsando uno de esos botones se abre el formulario de configuración de enemigos:

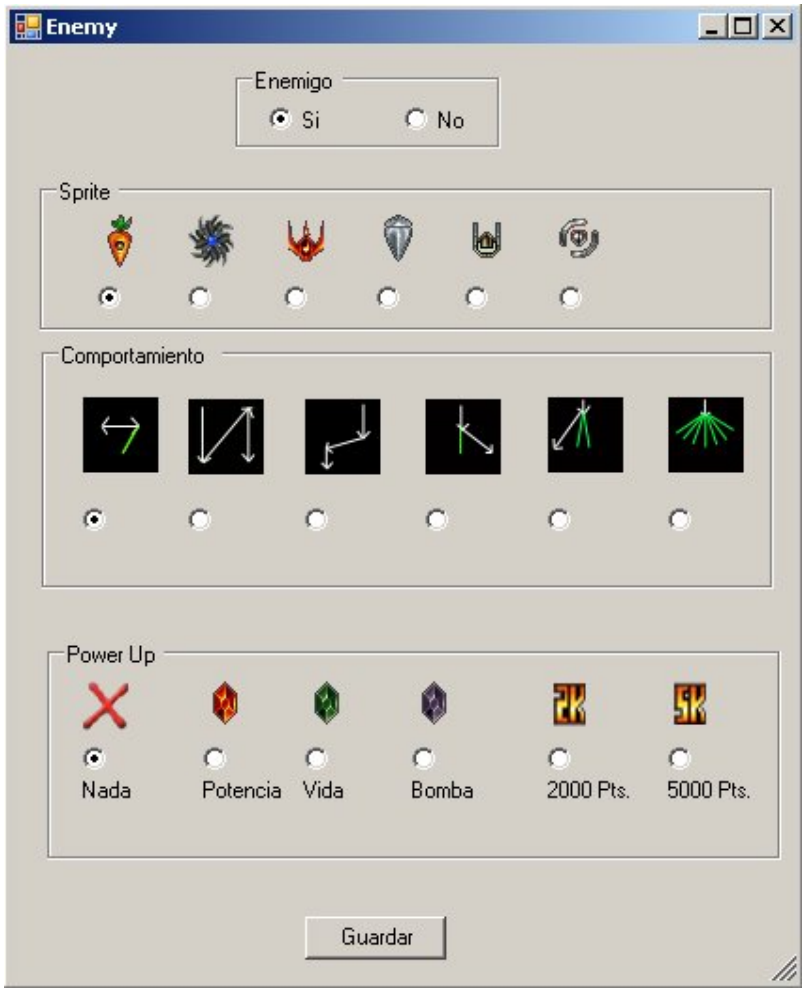


Figura 71. Editor avanzado: Configuración de enemigos

En primer lugar hay que decidir si se desea incluir un enemigo o no, en esa ubicación. En caso afirmativo, hay que seleccionar el *sprite*, el comportamiento y el *power up* que dejará caer al ser destruido. Repitiendo el proceso cuantas veces se desee, se pueden configurar todos los enemigos de la pantalla.

Hay que tener en cuenta que la fila inferior representa la primera oleada, y la fila superior la última.

Una vez finalizada la configuración, se debe pulsar el botón “Guardar y Salir” para guardar el archivo y finalizar la edición.

Si en algún momento se desea interrumpir el proceso de configuración y salir del editor, basta con pulsar el botón “Salir sin guardar”.

Anexo B: PLANIFICACIÓN

En este anexo se incluye la planificación del proyecto. En primer lugar se muestra la planificación establecida inicialmente y, posteriormente, la planificación final.

B.1 Planificación inicial

En este apartado se incluye la planificación inicial realizada antes de comenzar a desarrollar el proyecto. Se estimó que el proyecto comenzaría el 1 de julio de 2010, y terminaría el 28 de febrero de 2011. Lo que hace un total de 8 meses de trabajo.

En primer lugar se muestran los hitos más importantes del proyecto, en orden cronológico, y una breve explicación de cada uno:

- Documentación inicial. los primeros apartados de la memoria, la introducción y el estado de la cuestión.
- Repaso de Visual Studio. Tiempo dedicado a familiarizarse con un entorno de desarrollo que hace un tiempo que no se utiliza.
- Repaso de C# [63]. Tiempo dedicado a familiarizarse con un lenguaje con el que hace tiempo que no se programa.
- Estudio de XNA [64]. Tiempo dedicado a estudiar XNA con la profundidad suficiente para abordar el proyecto.
- Análisis. Fase de análisis donde se determinará el alcance del sistema y se procederá a extraer requisitos de usuario, casos de uso, etc.
- Diseño. Fase de diseño donde se determinará el diagrama de clases y la definición de cada una de ellas.
- Implementación. Fase de implementación donde se programará la aplicación, además de realizar diversas pruebas unitarias y de integración para cada uno de los módulos.
- Pruebas del sistema. Pruebas finales que se realizarán una vez que se haya dado por finalizada la fase de implementación.
- Documentación final. Última parte de la documentación que incluye, entre otras cosas, conclusiones y líneas futuras.
- Revisión final. Revisión final del proyecto y preparación para su entrega.

A continuación se incluyen unas capturas del programa Microsoft Project que reflejan la planificación, así como la fecha estimada para cada una de las tareas, y las relaciones de dependencia entre ellas.

Diagrama de Gantt

	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
1	**Documentación inicial**	**14 días?**	**jue 01/07/10**	**mar 20/07/10**		
2	Introducción	2 días	jue 01/07/10	vie 02/07/10		Jefe de Proyecto
3	**Estado de la cuestión**	**9 días?**	**lun 05/07/10**	**jue 15/07/10**		
6	Revisión	3 días?	vie 16/07/10	mar 20/07/10	5	Jefe de Proyecto
7	Repaso Visual Studio	14 días	jue 01/07/10	mar 20/07/10		Programador
8	Repaso C#	14 días	jue 01/07/10	mar 20/07/10		Programador
9	**Estudio de XNA**	**20 días?**	**mié 21/07/10**	**mar 17/08/10**		
10	Acercamiento a XNA	5 días?	mié 21/07/10	mar 27/07/10	6;7;8	Programador
11	Riemer's XNA Tutorial -> 2D	15 días	mié 28/07/10	mar 17/08/10	10	Programador
12	**Análisis**	**22 días**	**mié 18/08/10**	**jue 16/09/10**	**9**	
13	Alcance del sistema	2 días	mié 18/08/10	jue 19/08/10		Analista
14	Requisitos de Usuario	5 días	vie 20/08/10	jue 26/08/10	13	Analista
15	Casos de Uso	5 días	vie 27/08/10	jue 02/09/10	14	Analista
16	Requisitos de Software	5 días	vie 03/09/10	jue 09/09/10	15	Analista
17	Revisión	5 días	vie 10/09/10	jue 16/09/10	16	Jefe de Proyecto[50%];Ana
18	**Diseño**	**19 días?**	**vie 17/09/10**	**mié 13/10/10**	**12**	
19	Diagrama de clases	5 días	vie 17/09/10	jue 23/09/10	17	Diseñador
20	**Definición de clases**	**9 días?**	**vie 24/09/10**	**mié 06/10/10**	**19**	
40	Revisión	5 días	jue 07/10/10	mié 13/10/10	20	Jefe de Proyecto[50%];Dis
41	**Implementación**	**79 días?**	**jue 14/10/10**	**lun 31/01/11**	**18**	
42	**Gestión de estado**	**4 días?**	**jue 14/10/10**	**mar 19/10/10**		
46	**Jugador**	**10 días?**	**mié 20/10/10**	**mar 02/11/10**	**42**	
55	**Dibujar y animar la nave**	**6 días?**	**mié 03/11/10**	**mar 09/11/10**	**46**	
59	**Enemigos**	**10 días?**	**mié 10/11/10**	**mar 23/11/10**	**55**	
68	**Límites y Colisiones**	**12 días?**	**mié 24/11/10**	**jue 09/12/10**	**59**	
77	**Fase**	**17 días?**	**vie 10/12/10**	**lun 03/01/11**	**68**	
92	**Editor de niveles**	**12 días?**	**mar 04/01/11**	**mié 19/01/11**	**77**	
101	**Música y efectos de son**	**8 días?**	**jue 20/01/11**	**lun 31/01/11**	**92**	
110	Pruebas del sistema	5 días	mar 01/02/11	lun 07/02/11	41	Responsable de Pruebas[9
111	**Documentación final**	**5 días?**	**mar 08/02/11**	**lun 14/02/11**	**110**	
115	Revisión final	10 días	mar 15/02/11	lun 28/02/11	111	Jefe de Proyecto

Figura 72. Diagrama de Gantt – Tareas

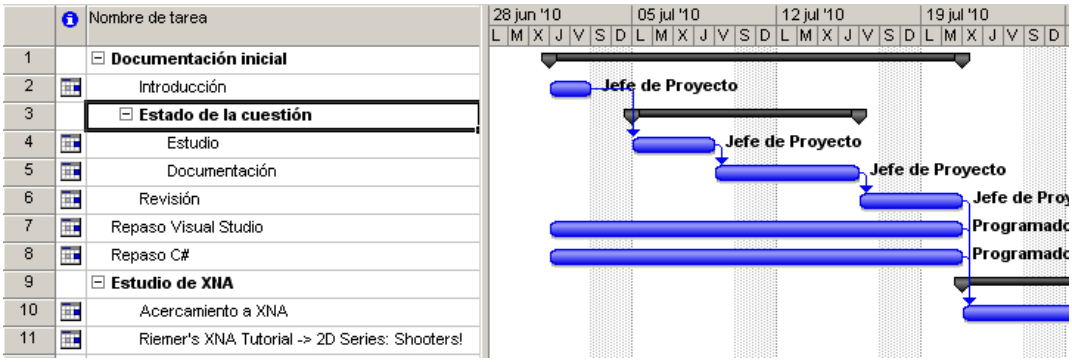


Figura 73. Planificación: Diagrama de Gantt - 1

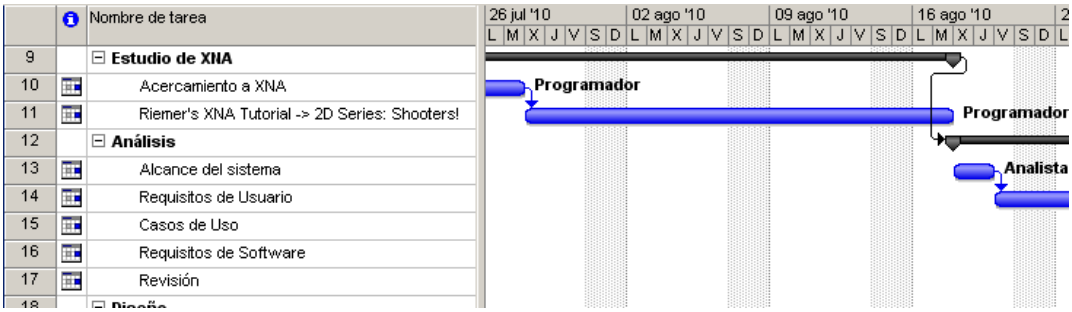


Figura 74. Planificación: Diagrama de Gantt - 2

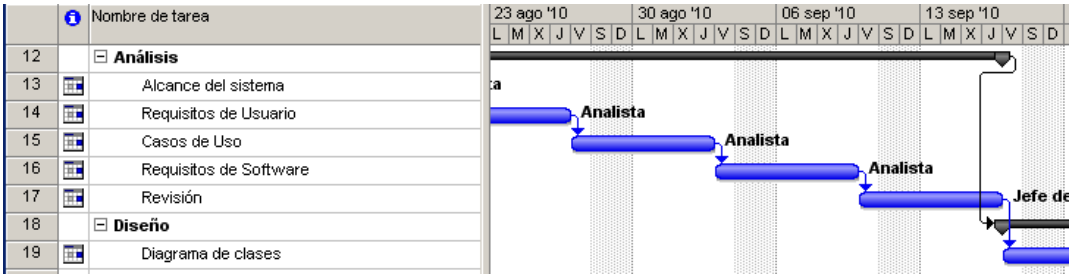


Figura 75. Planificación: Diagrama de Gantt - 3

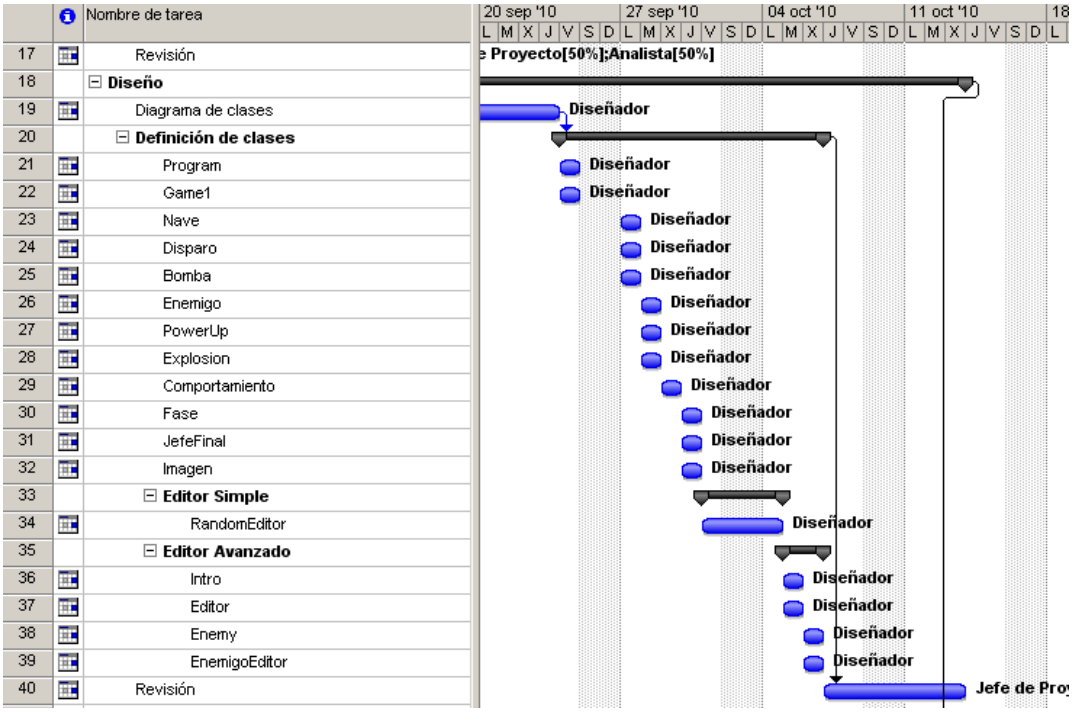


Figura 76. Planificación: Diagrama de Gantt - 4

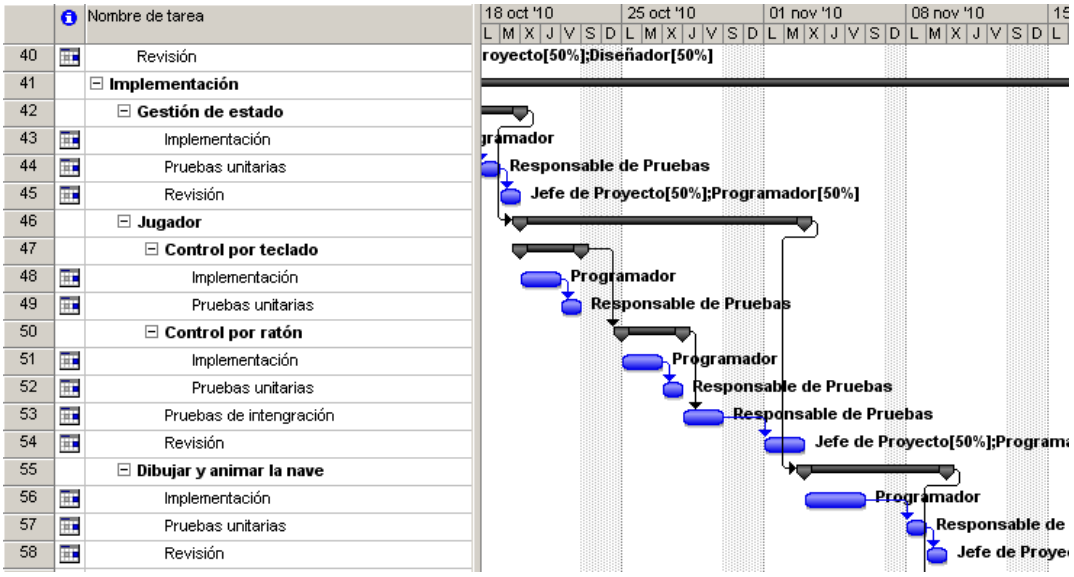


Figura 77. Planificación: Diagrama de Gantt - 5

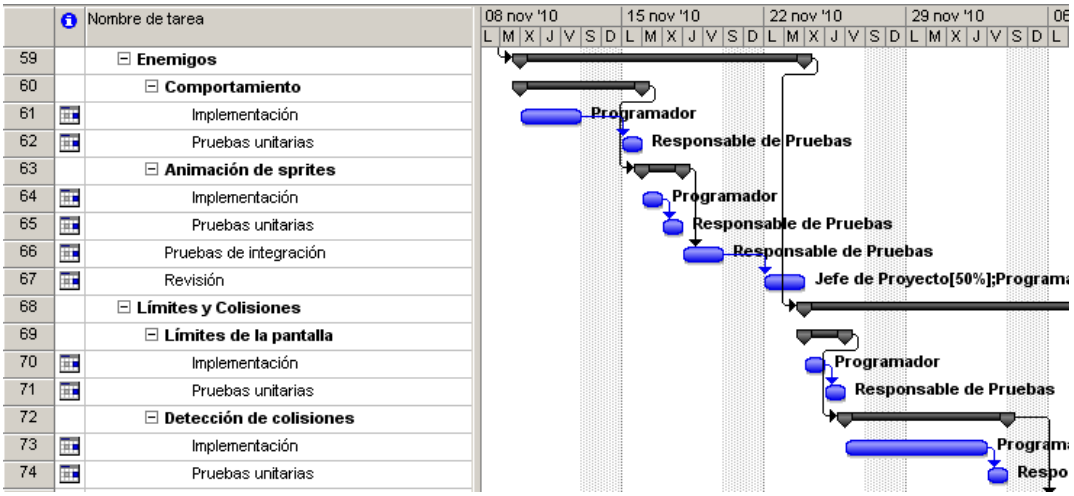


Figura 78. Planificación: Diagrama de Gantt - 6

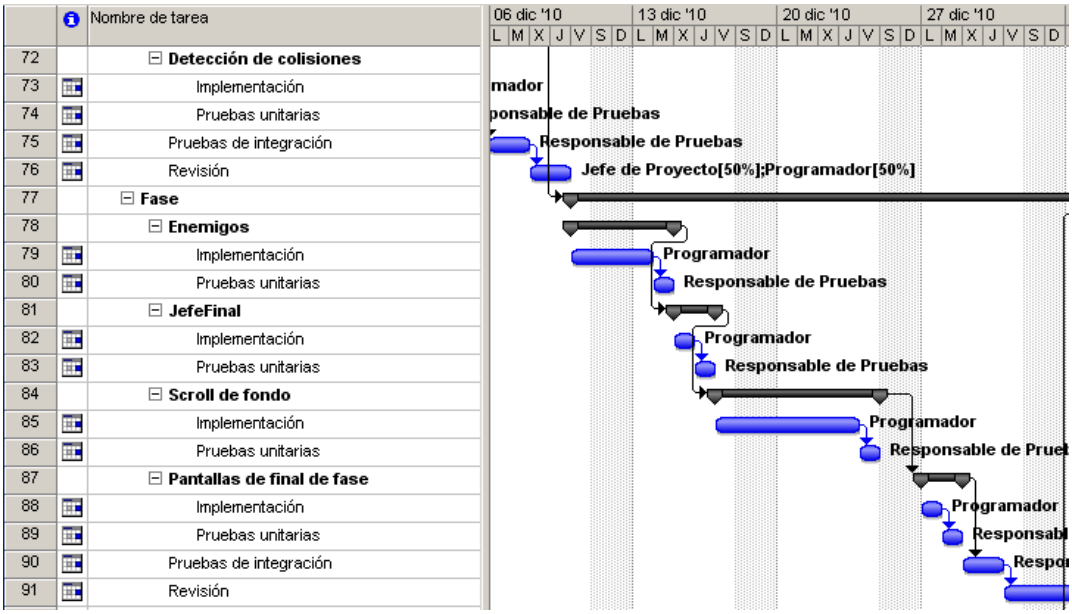


Figura 79. Planificación: Diagrama de Gantt - 7

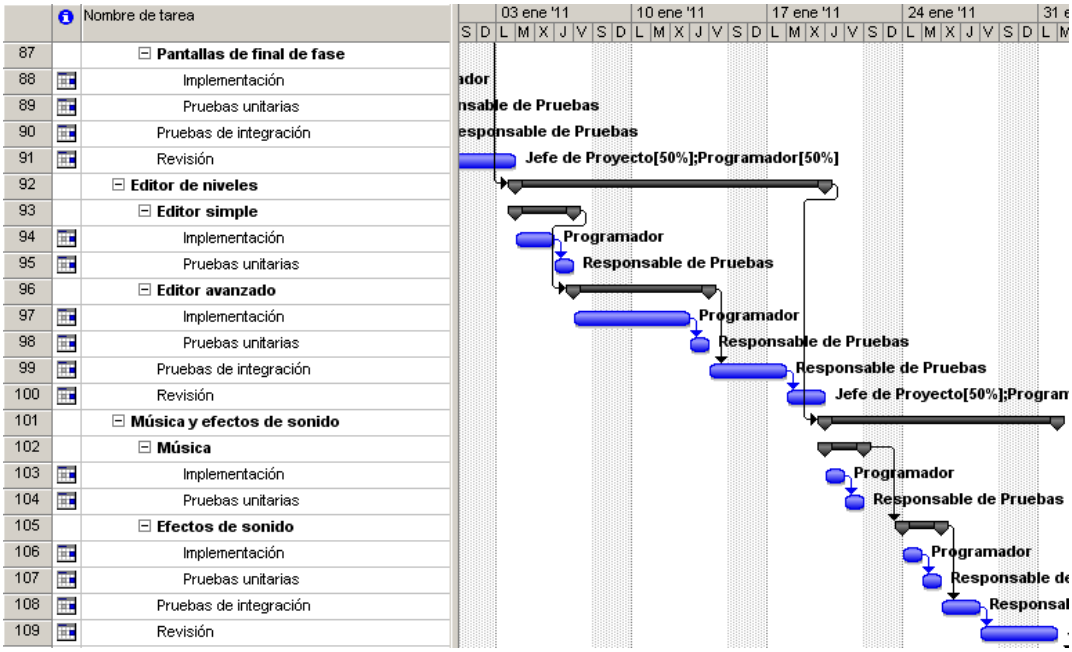


Figura 80. Planificación: Diagrama de Gantt - 8

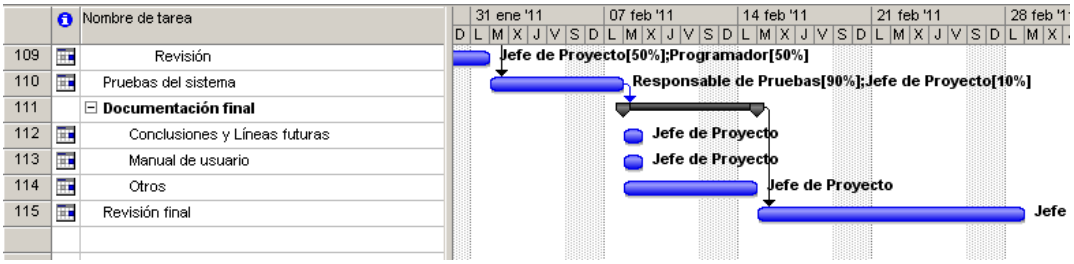


Figura 81. Planificación: Diagrama de Gantt - 9

B.2 Planificación final

La planificación inicial de un proyecto no siempre se corresponde con la realidad, y este proyecto no ha sido una excepción. A continuación se incluyen unas capturas del programa Microsoft Project que reflejan el estado final de la planificación. Posteriormente se explicarán brevemente los cambios acontecidos.

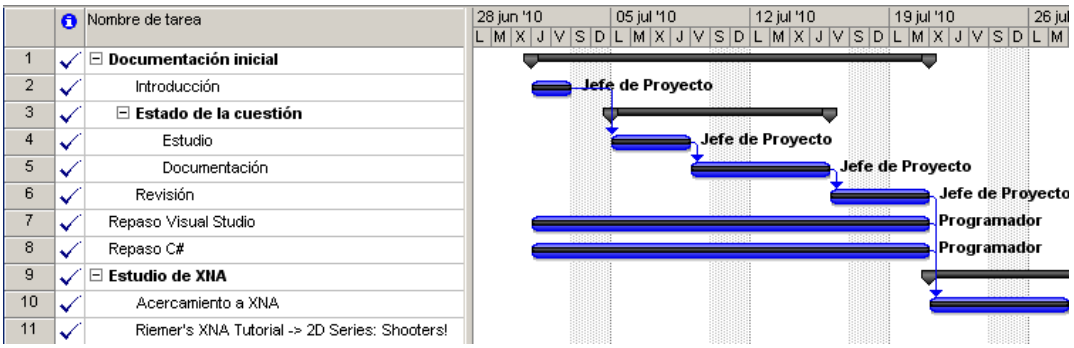


Figura 82. Planificación final: Diagrama de Gantt – 1

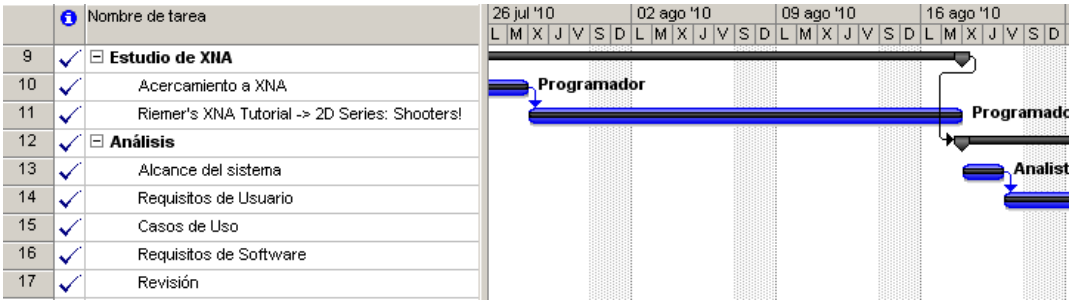


Figura 83. Planificación final: Diagrama de Gantt - 2

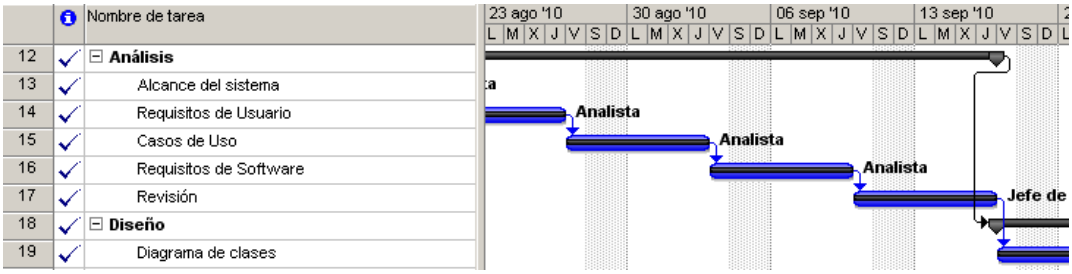


Figura 84. Planificación final: Diagrama de Gantt - 3

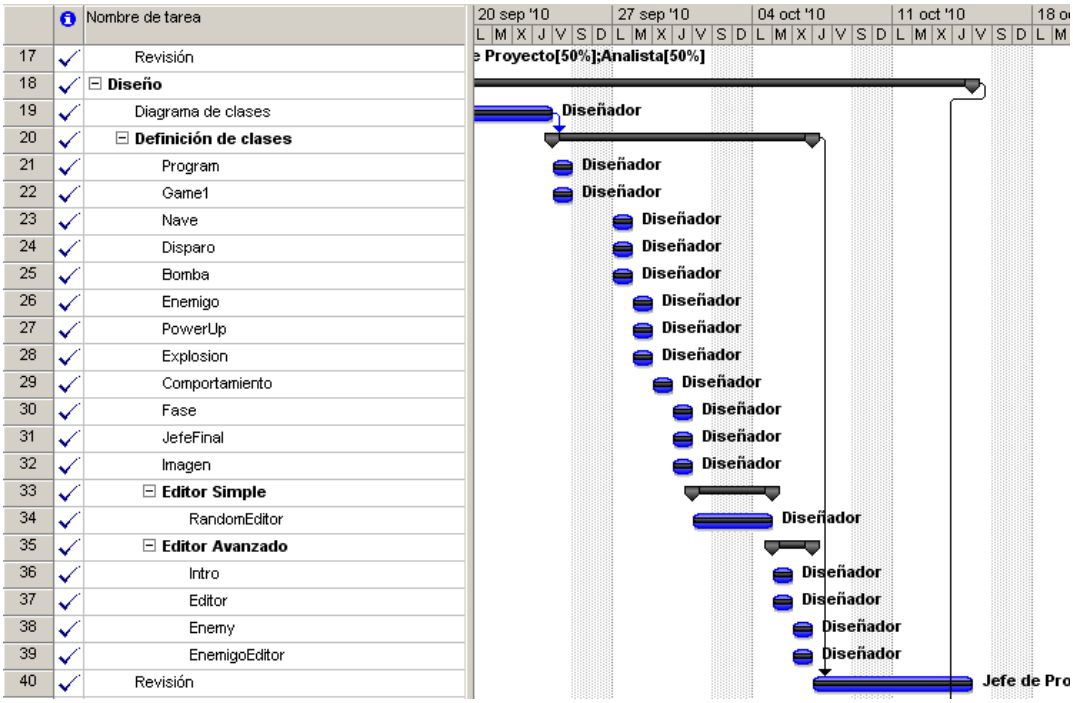


Figura 85. Planificación final: Diagrama de Gantt - 4

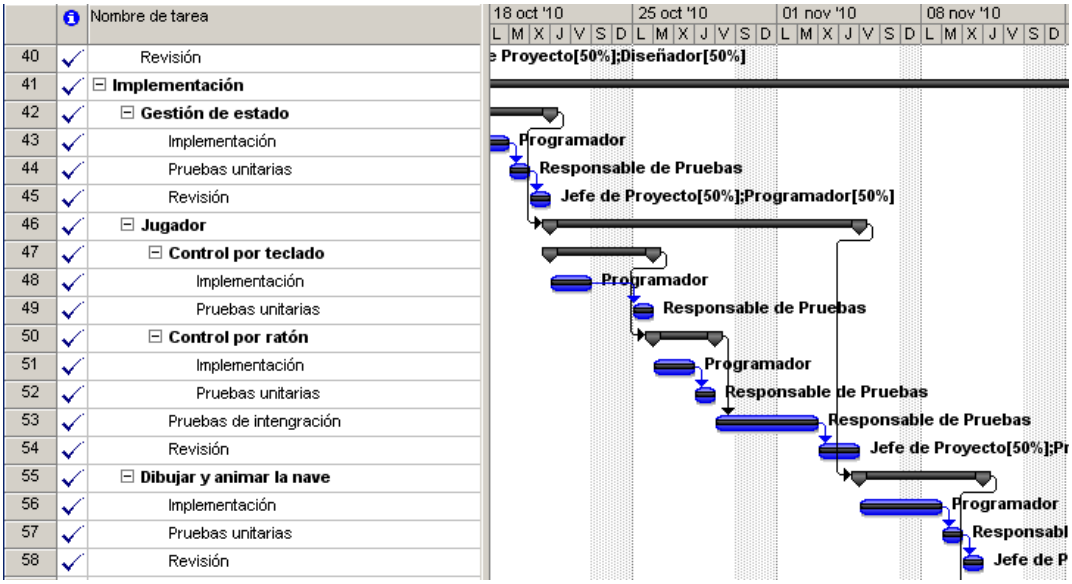


Figura 86. Planificación final: Diagrama de Gantt - 5

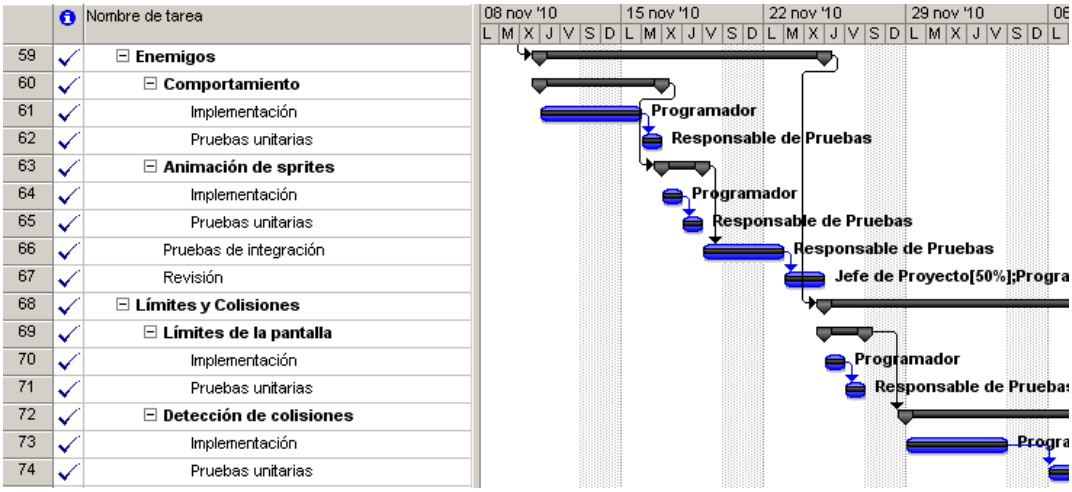


Figura 87. Planificación final: Diagrama de Gantt - 6

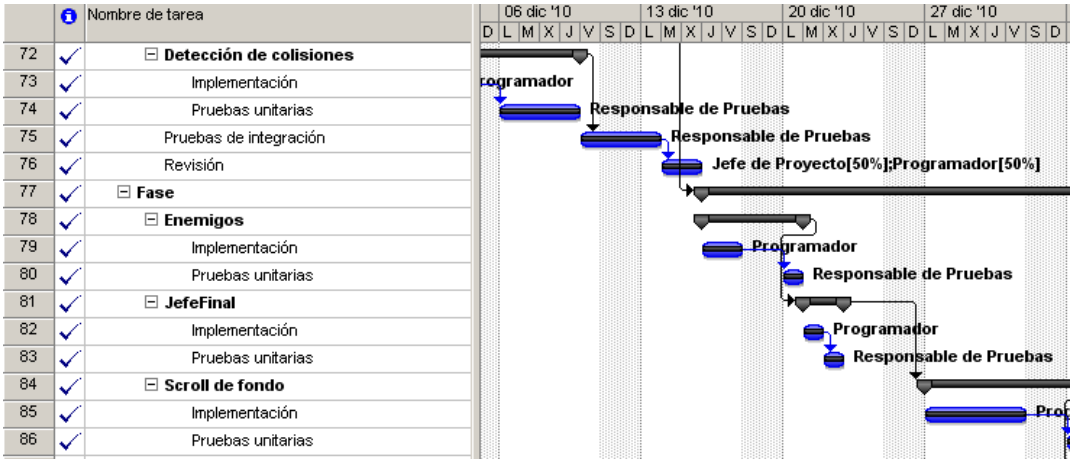


Figura 88. Planificación final: Diagrama de Gantt - 7

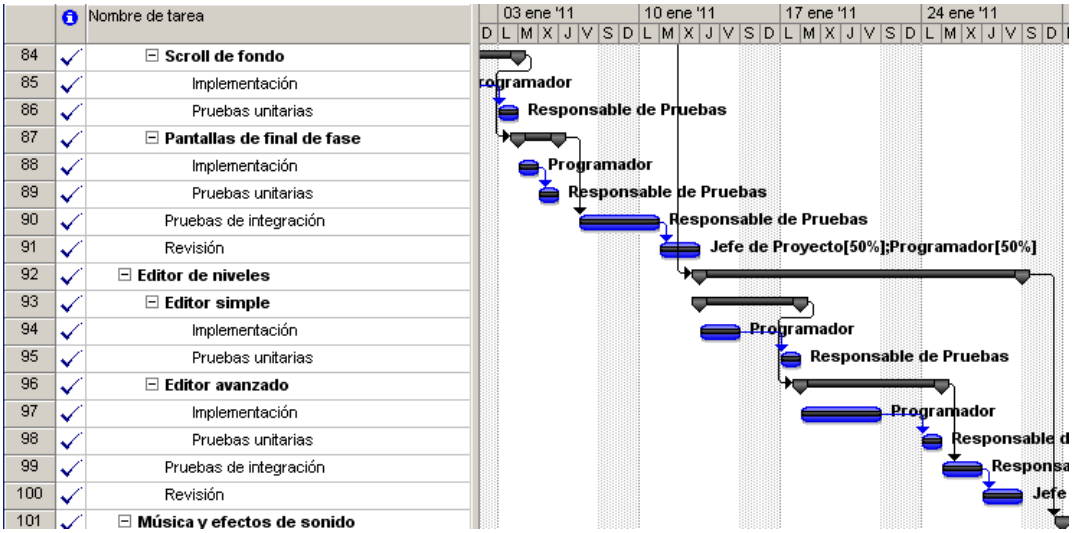


Figura 89. Planificación final: Diagrama de Gantt - 8

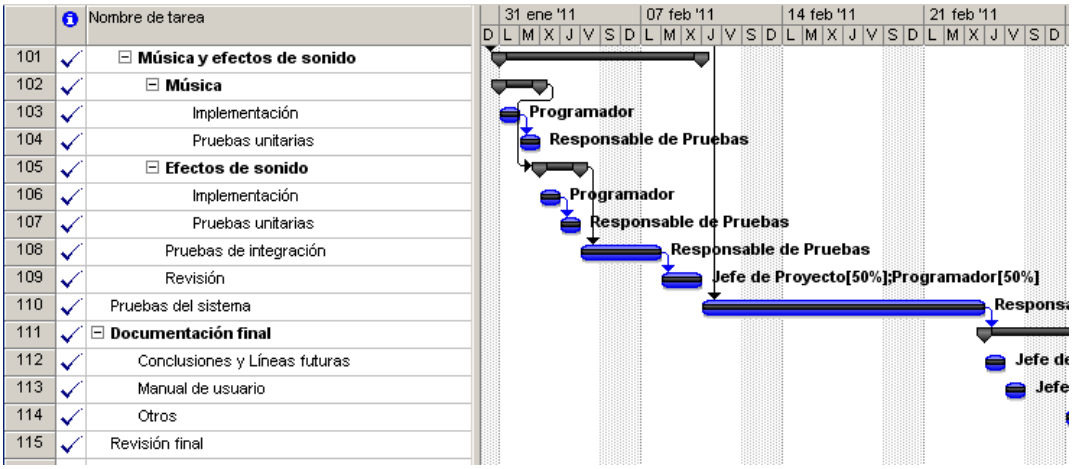


Figura 90. Planificación final: Diagrama de Gantt - 9

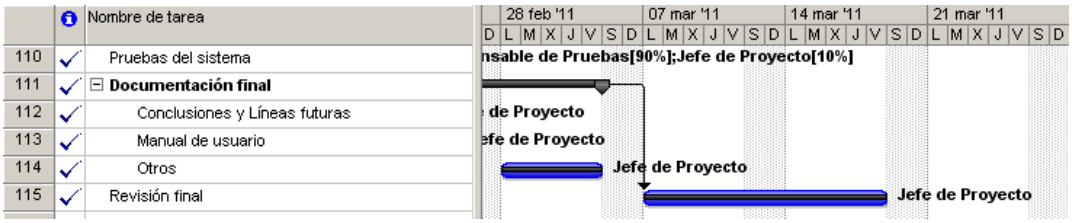


Figura 91. Planificación final: Diagrama de Gantt - 10

Como se puede apreciar en las capturas, los cambios no han sido muy significativos y la duración final del proyecto se ha excedido en aproximadamente medio mes.

En el aspecto externo, los principales retrasos se han debido a las fiestas. Aunque en la planificación inicial se tomaban como días laborables de lunes a viernes, y se dejaba el fin de semana de descanso, no se tuvo en cuenta las fiestas nacionales tales como el día de Navidad o el puente de la Constitución, que caían entre semana.

En cuanto al proyecto en sí, la fase de desarrollo apenas ha sufrido retrasos hasta la fase final. La planificación inicial exageró quizás algunos plazos ya que ciertos aspectos se terminaron más rápido de lo esperado. Sin embargo, ese tiempo extra del que se disponía se dedicó a aumentar el tiempo de revisión y a buscar la forma de optimizar el juego. Los retrasos más importantes se han producido en la fase final. Pese a que se realizó una importante batería de pruebas durante cada una de las fases de la implementación, hubo una serie de errores que no fueron detectados hasta las pruebas finales. Esto provocó que se tuviera que expandir el tiempo dedicado a ese aspecto, al tener que corregir todos y cada uno de los errores que se iban produciendo y comprobar que no se volvían a producir.

Anexo C: PRESUPUESTO

En el siguiente anexo se mostrarán los distintos gastos necesarios para llevar a cabo el desarrollo del videojuego. En cada uno de los apartados se especificarán, de forma desglosada, todos y cada uno de los gastos.

Se ha realizado un presupuesto a 8 meses tomando la planificación como referencia, y, posteriormente, un presupuesto a 3 meses que se ajusta más a la realidad.

Finalmente se incluyen los detalles de publicación del juego en la plataforma Steam.

C.1 Cálculo de costes a 8 meses

C.1.1 Personal a cargo del proyecto

En la primera tabla se muestran los gastos del personal por hora de trabajo. En la segunda se incluye la dedicación, en meses, de cada miembro al proyecto, el coste mensual que supone cada uno, y el coste total que cada empleado le supone a la empresa para el desarrollo del proyecto.

Personal a cargo del proyecto	
Puesto	Coste
Jefe de Proyecto / Analista	22,15 € / hora
Diseñador / Responsable de pruebas	14,43 € / hora
Programador	8,75 € / hora

Tabla 111: Coste/hora del personal a cargo del proyecto – 8 meses

Personal a cargo del proyecto			
Puesto	Dedicación (hombres mes)	Coste hombres mes	Coste
Jefe de Proyecto / Analista	8	2.658,00 €	21.264,00 €
Diseñador / Responsable de pruebas	5	1.731,60 €	8.658,00 €
Programador	4	1.050,60 €	4.200,00 €
TOTAL	17		34.122,00 €

Tabla 112: Coste total del personal a cargo del proyecto – 8 meses

C.1.2 Hardware y software

En la siguiente tabla se muestran los gastos por estaciones de trabajo, lo que incluye el ordenador y todo tipo de periféricos, así como conexión a Internet 24h. Además de los gastos derivados de la adquisición de las distintas licencias de software necesarias para el desarrollo del proyecto.

En función de la dedicación de cada elemento al proyecto y del coste de amortización, se calcula el coste imputable al proyecto de cada uno de los elementos.

Hardware y Software				
Descripción	Coste	Dedicación (meses)	Periodo de amortización (meses)	Coste imputable al proyecto
Estación de trabajo (x3)	595*3 = 1.785 €	8	60	238,00 €
Windows XP	199 €	8	60	26,53 €
Microsoft Office 2010	699 €	8	60	93,20 €
Altova UModel 2010	149 €	8	60	19,87 €
Adobe Photoshop 7.0	149 €	8	60	19,87 €
TOTAL	2.981 €			397,47 €

Tabla 113: Coste de los equipos – 8 meses

C.1.3 Material fungible

En apartado de material fungible se presupuestan principalmente los costes derivados de éste tipo de material. En este caso, material de oficina y consumibles.

El material de oficina son: bolígrafos, papel, cuadernos, grapadora, grapas, encuadernaciones, etc. se estima que el gasto no sobrepasará los 50 euros.

En cuanto a los consumibles se refieren a los CD´s o DVD´s que sean necesarios para almacenar o compartir datos. Se estima que el gasto no sobrepasará los 20 euros.

Material fungible	
Material	Coste
Material de oficina	50,00 €
Consumibles	20,00 €
TOTAL	70,00 €

Tabla 114: Material fungible – 8 meses

C.1.4 Viajes y dietas

También se cubrirán los gastos de comida, y los gastos de desplazamiento, en transporte público, de los empleados para llegar a su lugar de trabajo.

Viajes y Dietas	
Descripción	Coste
Dietas de comida	800 €
Desplazamientos	1.600 €
TOTAL	2.400 €

Tabla 115: Costes de viajes y dietas – 8 meses

C.1.5 Otros gastos

Es necesario cubrir una serie de gastos necesarios para el mantenimiento de los recursos tanto humanos como materiales, por lo que se incluyen a continuación los costes indirectos. Siendo éstos un 20% de la suma de los gastos de los apartados anteriores.

Otros gastos	
Descripción	Coste
Costes indirectos (20%)	7.397,89 €
TOTAL	7.397,89 €

Tabla 116: Otros gastos – 8 meses

C.1.6 Presupuesto final

El presupuesto final se corresponde con la suma de los costes especificados en cada uno de los puntos del apartado anterior.

Presupuesto Final	
Descripción	Coste
Personal	34.122,00 €
Hardware y Software	397,47 €
Material fungible	70,00 €
Viajes y dietas	2.400,00 €
Otros gastos	7.397,89 €
COSTE TOTAL	44.387,36 €

Tabla 117: Presupuesto final – 8 meses

El presupuesto final es de 44.387,36 (cuarenta y cuatro mil trescientos ochenta y siete coma treinta y seis) euros.

C.2 Cálculo de costes a 3 meses

El presupuesto final calculado en el apartado anterior es bastante elevado. Hay que tener en cuenta que se trata del cálculo de costes hecho en base a la planificación, que dura 8 meses, incluida en el anexo B.

Si se plantea el proyecto desde el punto de vista de una empresa que cuenta con trabajadores con experiencia en la materia, y no desde la perspectiva de una sola persona que aborda su primer proyecto, se estima que la duración del mismo no sobrepasaría los 3 meses.

Por ello, en este apartado se va a incluir el cálculo de costes para este mismo proyecto, ajustando a 3 meses el tiempo de desarrollo del juego. De esta manera, se obtendrá un presupuesto más acorde a la realidad.

C.2.1 Personal a cargo del proyecto

En la primera tabla se muestran los gastos del personal por hora de trabajo. En la segunda se incluye la dedicación, en meses, de cada miembro al proyecto, el coste mensual que supone cada uno, y el coste total que cada empleado le supone a la empresa para el desarrollo del proyecto.

Personal a cargo del proyecto	
Puesto	Coste
Jefe de Proyecto / Analista	22,15 € / hora
Diseñador / Responsable de pruebas	14,43 € / hora
Programador	8,75 € / hora

Tabla 118: Coste/hora del personal a cargo del proyecto – 3 meses

Personal a cargo del proyecto			
Puesto	Dedicación (hombres mes)	Coste hombres mes	Coste
Jefe de Proyecto / Analista	3	2.658,00 €	7.974,00 €
Diseñador / Responsable de pruebas	3	1.731,60 €	5.194,80 €
Programador	2	1.050,60 €	2.100,00 €
TOTAL	8		15.268,80 €

Tabla 119: Coste total del personal a cargo del proyecto – 3 meses

C.2.2 Hardware y software

En la siguiente tabla se muestran los gastos por estaciones de trabajo, lo que incluye el ordenador y todo tipo de periféricos, así como conexión a Internet 24h. Además de los gastos derivados de la adquisición de las distintas licencias de software necesarias para el desarrollo del proyecto.

En función de la dedicación de cada elemento al proyecto y del coste de amortización, se calcula el coste imputable al proyecto de cada uno de los elementos.

Hardware y Software				
Descripción	Coste	Dedicación (meses)	Periodo de amortización (meses)	Coste imputable al proyecto
Estación de trabajo (x3)	595*3 = 1.785 €	3	60	89,25 €
Windows XP	199 €	3	60	9,95 €
Microsoft Office 2010	699 €	3	60	34,95 €
Altova UModel 2010	149 €	3	60	7,45 €
Adobe Photoshop 7.0	149 €	3	60	7,45 €
TOTAL	2.981 €			149,05 €

Tabla 120: Coste de los equipos – 3 meses

C.2.3 Material fungible

En apartado de material fungible se presupuestan principalmente los costes derivados de éste tipo de material. En este caso, material de oficina y consumibles.

El material de oficina son: bolígrafos, papel, cuadernos, grapadora, grapas, encuadernaciones, etc. se estima que el gasto no sobrepasará los 30 euros.

En cuanto a los consumibles se refieren a los CD´s o DVD´s que sean necesarios para almacenar o compartir datos. Se estima que el gasto no sobrepasará los 10 euros.

Material fungible	
Material	Coste
Material de oficina	30,00 €
Consumibles	10,00 €
TOTAL	40,00 €

Tabla 121: Material fungible – 3 meses

C.2.4 Viajes y dietas

También se cubrirán los gastos de comida, y los gastos de desplazamiento, en transporte público, de los empleados para llegar a su lugar de trabajo.

Viajes y Dietas	
Descripción	Coste
Dietas de comida	300 €
Desplazamientos	600 €
TOTAL	900 €

Tabla 122: Costes de viajes y dietas – 3 meses

C.2.5 Otros gastos

Es necesario cubrir una serie de gastos necesarios para el mantenimiento de los recursos tanto humanos como materiales, por lo que se incluyen a continuación los costes indirectos. Siendo éstos un 20% de la suma de los gastos de los apartados anteriores.

Otros gastos	
Descripción	Coste
Costes indirectos (20%)	3.271,57 €
TOTAL	3.271,57 €

Tabla 123: Otros gastos – 3 meses

C.2.6 Presupuesto final

El presupuesto final se corresponde con la suma de los costes especificados en cada uno de los puntos del apartado anterior.

Presupuesto Final	
Descripción	Coste
Personal	15.268,80 €
Hardware y Software	149,05 €
Material fungible	40,00 €
Viajes y dietas	900,00 €
Otros gastos	3.271,57 €
COSTE TOTAL	19.629,42 €

Tabla 124: Presupuesto final – 3 meses

El presupuesto final es de 19.629,42 (diecinueve mil seiscientos veintinueve coma cuarenta y dos) euros.

C.3 Publicación del juego

El objetivo de desarrollar un videojuego culmina con su publicación. En este caso, se ha optado por utilizar la mayor plataforma de distribución digital de videojuegos del mercado: Steam.

Para poder publicar un juego en Steam, es necesario enviar el producto a la compañía para que sus empleados lo evalúen. Una vez lo han aceptado, se procede a incluir una serie de instrucciones en el videojuego para que éste soporte las características de Steam, tales como el inicio de sesión. Mientras tanto se negocian los porcentajes que se va a llevar cada una de las partes por cada descarga, puesto que Steam no establece un porcentaje fijo, lo negocia para cada juego.

Una vez se han acordado los porcentajes, el juego estará disponible en Steam para que cualquier usuario con acceso a internet y una cuenta de Steam pueda pagar por él y descargárselo.

Después de analizar juegos de similares características, se establece un precio de 2,99 euros. Suponiendo que los porcentajes queden repartidos de forma que Steam se quede con un 30% y la empresa desarrolladora con un 70%, se tendría un beneficio de 2,09 euros por cada descarga.

Por tanto, si el coste necesario para desarrollar el juego es de 19.629,42 euros, y se obtiene un beneficio de 2,09 euros por cada descarga; el número total de descargas necesarias para cubrir gastos asciende a 9.393 descargas.

Anexo D: DEFINICIONES Y REFERENCIAS

Este último anexo incluye el glosario de términos utilizados a lo largo de la memoria, y las referencias consultadas para escribirla.

D.1 Definiciones

- 2D: se refiere a aquellos videojuegos cuya representación se basa en dos dimensiones del espacio: anchura y longitud.
- 3D: se refiere a aquellos videojuegos cuya representación simula o se basa en las tres dimensiones del espacio: altura, anchura y longitud.
- API: una interfaz de programación de aplicaciones (del inglés *Application Programming Interface*) es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizada por otro software como una capa de abstracción.
- Arcade: es el término genérico de las máquinas recreativas de videojuegos disponibles en lugares públicos de diversión, centros comerciales, restaurantes, bares o salones recreativos especializados. Son similares a los *pinballs* y a las máquinas tragaperras de los casinos, no obstante debido a que no son juegos de azar ni de apuestas, ya que se basan en la destreza del jugador, no tienen las limitaciones legales de éstas.
- Artwork: en el mundo de los videojuegos, se refiere a las diversas imágenes que se utilizan para representar los elementos de un videojuego tales como el jugador, el fondo, etc.
- Avatar: en lo que a XBOX 360 se refiere, es la representación virtual del usuario en la plataforma XBOX Live.
- Axonométrica: la perspectiva axonométrica es un sistema de representación gráfica, consistente en representar elementos geométricos o volumétricos en un plano mediante proyección paralela o cilíndrica, referida a tres ejes ortogonales, de forma que conserven sus proporciones en cada una de las tres direcciones del espacio: altura, anchura y longitud.
- Bidimensional: ver 2D.
- Bullet Hell Shooter: subgénero de los matamarcianos caracterizado por un ritmo frenético y una pantalla llena de disparos.

- Framework: es una estructura conceptual y tecnológica de soporte definida, normalmente con artefactos o módulos de software concretos, con base en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado, entre otros programas para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

- Hardware: se refiere a todos los componentes tangibles de un ordenador, tales como la placa base, la tarjeta gráfica, etc.

- Hi-Score: término inglés que hace referencia al concepto de puntuación máxima. Este concepto revolucionó el mundo de los videojuegos, especialmente en máquinas recreativas, fomentando la competitividad entre jugadores para ver quién conseguía la puntuación máxima.

- Jugabilidad: se define como el conjunto de propiedades que describen la experiencia del jugador ante un sistema de juego determinado, cuyo principal objetivo es divertir y entretener de forma satisfactoria, ya sea solo o en compañía de otros jugadores.

- Jumper: hace referencia al dispositivo de los juegos de la videoconsola Magnavox Oddyssey. Fue la primera consola en distribuir sus cartuchos, conocidos como *jumpers*, porque en realidad no eran cartuchos programables sino tarjetas con *jumpers* que hacían diferentes contactos con la consola, consiguiendo así generar diferentes señales analógicas que se transmitían al televisor.

- Maniac Shooter: ver *Bullet Hell Shooter*.

- Matamarcianos: subgénero de los *Shoot'em Up* en el que el jugador, a los mandos de una nave espacial o sucedáneo, se enfrenta a hordas de enemigos, habitualmente de carácter extraterrestre.

- Multijugador: término que hace referencia a la posibilidad de que varias personas jueguen a la vez, o por turnos, a un videojuego durante la misma partida.

- Power Up: objeto que el jugador puede encontrar en el juego y que inmediatamente afecta de manera positiva, y a veces negativa, a alguna de sus cualidades, de forma temporal o definitiva.

- Renderizar: proceso por el cual un ordenador es capaz de generar una imagen desde un modelo, normalmente con la finalidad de mostrarla.

- Scroll: se denomina *scroll* al movimiento en 2D de los gráficos que conforman el escenario. Se habla de *scroll* horizontal cuando la acción se desarrolla horizontalmente, y de *scroll* vertical, cuando ésta se desarrolla verticalmente.
- Shooter: ver *Shoot'em Up*.
- Shoot'em Up: término utilizado para referirse a un género de los videojuegos en los que el jugador controla a un personaje, vehículo, nave u otro objeto, y dispara contra interminables hordas de enemigos que van apareciendo en la pantalla.
- Sistema de distribución digital: servicio *online* dedicado a la venta de videojuegos sin necesidad de mediadores o de que los juegos requieran ser grabados en algún soporte físico.
- Software: se refiere al equipamiento lógico de un ordenador digital. Comprende el conjunto de componentes lógicos necesarios que hacen posible la realización de tareas específicas.
- Sprite: pequeño mapa de bits que se dibuja en la pantalla, y representa algún elemento del videojuego. Es habitual en los videojuegos en 2D que los *sprites* tengan animación, especialmente los elementos importantes como el jugador.
- Tileset: conjunto de pequeñas imágenes que se agrupan en un mismo archivo y que, combinadas, sirven para representar diversos elementos.
- Tridimensional: ver 3D.

D.2 Referencias

- [1] HoyTecnologia. Los Videojuegos rivalizan con Hollywood con una caja de 57.600 millones de euros. [ref. de 2010] < http://www.hoytecnologia.com/noticias/videojuegos-rivalizan-Hollywood-caja/165569 >
- [2] Vandal. GTA IV podría ser el juego más caro de la historia. [ref. de 2008]: < http://www.vandal.net/noticia/32182/gta-iv-podria-ser-el-juego-mas-caro-de-la-historia/ >
- [3] Steamworks - Publishing Services. [ref. de 2010]: < http://www.steampowered.com/steamworks/publishingservices.php >
- [4] WiiWare – Nintendo. [ref. de 2010]: < http://www.nintendo.es/NOE/es_ES/systems/wiiware_8343.html >
- [5] Website Oficial de Playstation: Playstation Network, PSN. [ref. de 2010]: < http://es.playstation.com/psn/ >
- [6] XBOX Live. [ref. de 2010]: < http://www.xbox.com/es-ES/Live/Home >
- [7] XNA - Microsoft XNA. [ref. de 2010]: < http://create.msdn.com/en-US/ >
- [8] Visual Studio. [ref. de 2010]: < http://msdn.microsoft.com/es-es/vstudio/aa718325 >
- [9] ElOtroLado – Historia de los videojuegos: Los inicios. [ref. de 2009]: < http://www.elotrolado.net/wiki/Historia_de_los_videojuegos:_Los_inicios >
- [10] Wikipedia, la enciclopedia libre. Historia de los videojuegos. [ref. de 2010]: < http://es.wikipedia.org/wiki/Historia_de_los_videojuegos >
- [11] La Historia de Atari. [ref. de 2007]: < http://www.neoteo.com/la-historia-de-atari >
- [12] ElOtroLado – Historia de los videojuegos: Década de los 70. [ref. de 2009]: < http://www.elotrolado.net/wiki/Historia_de_los_videojuegos:_Decada_de_los_70 >
- [13] La Historia de la Gran N. [ref. de 2008]: < http://www.soynintendero.com/nintendo/historia/historia-de-nintendo.html >

- [14] Wikipedia, la enciclopedia libre. Pac-Man. [ref. de 2010]: < http://es.wikipedia.org/wiki/Pac-Man >
- [15] Capcom | History. [ref. de 2010]: < http://www.capcom.co.jp/ir/english/company/history.html >
- [16] Activision Inc. – Company History. [ref. de 2010]: < http://www.fundinguniverse.com/company-histories/Activision-Inc-Company-History.html >
- [17] Wikipedia, la enciclopedia libre. Shoot'em up. [ref. de 2010]: < http://es.wikipedia.org/wiki/Shoot_%27em_up >
- [18] Wikipedia, la enciclopedia libre. Taito Co. [ref. de 2010]: < http://es.wikipedia.org/wiki/Taito_Corporation >
- [19] Vandal. Breve Historia de los matamarcianos (I). [ref. de 2010]: < http://www.vandal.net/retro/breve-historia-de-los-matamarcianos-i >
- [20] Game Sanctuary. Historia de los SHMUPS – Xevious. [ref. de 2007]: < http://gamesanctuary.blogspot.com/2007/03/historia-de-los-shmups-xevious.html >
- [21] Wikipedia, the free encyclopedia. Zaxxon. [ref. de 2010]: < http://en.wikipedia.org/wiki/Zaxxon >
- [22] Wikipedia, the free encyclopedia. 1942 (video game). [ref. de 2010]: < http://en.wikipedia.org/wiki/1942_%28video_game%29 >
- [23] Robotron 2084 (1982) – Insert Coin. [ref. de 2009]: < http://jaimixx.lacoctelera.net/post/2009/02/19/robotron-2084-1982 >
- [24] Game Sanctuary. Historia de los SHMUPS – Space Harrier. [ref. de 2007]: < http://gamesanctuary.blogspot.com/2007/04/historia-de-los-shmups-space-harrier.html >
- [25] SEGA of America. [ref. de 2010]: < http://www2.sega.com/corporate/corporatehist.php >
- [26] Wikipedia, the free encyclopedia. Gradius. [ref. de 2010]: < http://en.wikipedia.org/wiki/Gradius >
- [27] Konami Co. – Corporation History. [ref. de 2010]: < http://www.konami.co.jp/en/corporate/history/ >
- [28] Wikipedia, the free encyclopedia. Fantasy Zone. [ref. de 2010]: < http://en.wikipedia.org/wiki/Fantasy_Zone >

- [29] Wikipedia, la enciclopedia libre. Twinbee. [ref. de 2010]: < http://es.wikipedia.org/wiki/TwinBee_%28videojuego%29 >
- [30] Brain of Cellsius. La historia de... R-Type. [ref. de 2009]: < http://blogs.gamefilia.com/cellsius/28-09-2009/26769/la-historia-de-r-type-pt1 >
- [31] Jap-sai – Irem. [ref. de 2009]: < http://www.jap-sai.com/Games/Irem/Irem.htm >
- [32] Vandal. Breve Historia de los matamarcianos (II). [ref. de 2010]: < http://www.vandal.net/retro/breve-historia-de-los-matamarcianos-ii >
- [33] Wikipedia, la enciclopedia libre. Aero Fighters. [ref. de 2010]: < http://es.wikipedia.org/wiki/Aero_Fighters >
- [34] Hombre Imaginario. Sega Saturn – Batsugun. [ref. de 2009]: < http://hombreimaginario.com/blog/2009/sega-saturn-batsugun/ >
- [35] Wikipedia, the free encyclopedia. Gunbird. [ref. de 2010]: < http://en.wikipedia.org/wiki/Gunbird >
- [36] Wikipedia, the free encyclopedia. Air Gallet. [ref. de 2010]: < http://en.wikipedia.org/wiki/Air_Gallet >
- [37] The Cave of Shooting – 1995 – DonPachi. [ref. de 2002]: < http://www.world-of-arcades.net/Cave/DonPachi/DonPachi.htm >
- [38] The Cave of Shooting – 2002 – DoDonPachi. [ref. de 2008]: < http://www.world-of-arcades.net/Cave/DdpDaiOuJou/DdpDaiOuJou.htm >
- [39] Cave Co. LTD. – Company History. [ref. de 2010]: < http://www.cave.co.jp/company_e/history.html >
- [40] Space Invaders: Infinite Gene. [ref. de 2010]: < http://dlgames.square-enix.com/siig/ >
- [41] Geometry Wars: Retro Evolved. [ref. de 2005]: < http://www.bizarrecreations.com/games/geometry_wars_retro_evolved/ >
- [42] Meristation. Geometry Wars: Galaxies – Avance. [ref. de 2007]: < http://www.meristation.com/v3/des_avances.php?id=cw4696980de2bea&idj=cw4656e466269cc&pic=GEN >
- [43] Wikipedia, la enciclopedia libre. Deathsmiles. [ref. de 2010]: < http://en.wikipedia.org/wiki/Deathsmiles >

- [44] Gundemonium Collection. [ref. de 2010]: < http://www.gundemonium.com/ >
- [45] Adventure Game Studio. [ref. de 2010]: < http://www.adventuregamestudio.co.uk/ >
- [46] Allegro. [ref. de 2010]: < http://www.talula.demon.co.uk/allegro/ >
- [47] Blender 3D. [ref. de 2010]: < http://www.blender.org/ >
- [48] Crystal Space 3D. [ref. de 2010]: < http://www.crystalspace3d.org/main/Main_Page >
- [49] Dim3 Documentation Wikie. [ref. 2010]: < http://klinksoftware.net/wiki/index.php?title=Main_Page >
- [50] GameBryo Game Engine. [ref. de 2010]: < http://www.gamebryo.com/Gamebryo/ >
- [51] YoYo Games - Game Maker. [ref. de 2010]: < http://www.yoyogames.com/gamemaker/ >
- [52] 3D Game Studio. [ref. de 2010]: < http://www.3dgamestudio.com/ >
- [53] Wikipedia, la enciclopedia libre - M.U.G.E.N. [ref. de 2010]: < http://es.wikipedia.org/wiki/M.U.G.E.N. >
- [54] Ogre – Open Source 3D Graphics Engine. [ref. de 2010]: < http://www.ogre3d.org/ >
- [55] RPG Maker. [ref. de 2010]: < http://rpgmaker.es/ >
- [56] Unreal Engine. [ref. de 2010]: < http://www.unrealengine.com/ >
- [57] XNA Game Studio 3.1. Documentación. [ref. de 2010]: < http://msdn.microsoft.com/en-us/library/bb200104%28v=XNAGameStudio.31%29.aspx >
- [58] Riemers XNA Tutorial – 2D Series: Shooters. [ref. de 2009]: < http://www.riemers.net/eng/Tutorials/XNA/Csharp/series2d.php >
- [59] The State of Things. [ref. de 2010]: < http://www.xnadevelopment.com/tutorials/thestateofthings/thestateofthings.shtml >
- [60] Custom Mouse Pointer in XNA. [ref. de 2008]: < http://www.virtualrealm.com.au/Blog/tabid/62/EntryId/2/Custom-Mouse-Pointer-in-XNA.aspx >

- [61] Escarbando código. Animación de sprites – XNA. [ref. de 2010]: < http://escarbandocodigo.wordpress.com/2010/02/19/animaciones-de-sprites-xna/ >
- [62] Aprendiendo XNA. Reproducir MP3 y más formatos de audio de forma sencilla. [ref. de 2009]: < http://aprendiendoxna.wordpress.com/tutoriales/tutoriales-varios/reproducir-mp3-y-mas-formatos-de-audio-de-forma-sencilla/ >
- [63] Fco. Javier Ceballos. *Microsoft C#*. 2006. RA-MA Editorial. 833 p.
- [64] Stephen Cawood, Pat MacGee. *Microsoft XNA Game Stidio Creator's Guide. An Introduction to XNA Game Programming*. 2007. The McGraw-Hill Companies. 482 p.