UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA EN INFORMÁTICA
PROYECTO FIN DE CARRERA

DISEÑO E IMPLEMENTACIÓN DE UN "BOMBERMAN" MEDIANTE XNA 3.1 APLICANDO INTELIGENCIA ARTIFICIAL

Autor: Juan Ramón Martínez Navarro

Tutor: Juan Peralta Donate

Abril de 2010

A Elena

Agradecimientos

Este documento pone punto y final a este Proyecto Fin de Carrera. En estas líneas me gustaría recordar a todas las personas que han conseguido que fuera posible.

Quiero comenzar dando las gracias a mis padres, Isabel y Juan, por el apoyo incondicional en todo lo que hago. Por haber creído siempre en mí, por preocuparse día a día y respaldarme cuando lo he necesitado. Gracias también a Daniel, mi hermano, por estar siempre ahí. Eres un jugón.

A Juan Peralta, mi tutor, por darme la oportunidad de realizar este proyecto. Por toda la ayuda y tiempo dedicados. Siempre ha estado disponible para resolver dudas y aportar sugerencias y conocimientos. Sin su colaboración, este proyecto no hubiera sido posible.

A mi amigos y compañeros de universidad y beca. En especial a Alberto y David, por conseguir que los dos años de segundo ciclo pasaran volando. También a Fernando, por los cafés y las cañas de los domingos que resumen diez años de buena amistad.

Al resto de mi familia, por su apoyo y cariño. Especialmente a Juanjo y Pablo, otro par de jugones.

A Carlos, por aceptar la invitación a ser parte del tribunal.

Por último y muy especialmente, a Elena. Por cada momento que ha compartido conmigo. Por tener siempre una sonrisa para mí y conseguir que afronte todo con ilusión. Sé que está y estará siempre ahí.

En definitiva, gracias a todos los que me han ayudado a llegar al final de este camino.

Resumen

En la última década los videojuegos se han consolidado como uno de los sectores más rentables de la industria del ocio, lo que ha llevado a que se produzca una gran expansión y diversificación del negocio. Un sector que hace menos de 5 años se basaba en el desarrollo de juegos comerciales[1] ha evolucionado introduciendo nuevos modelos como los videojuegos casuales[2], bajo demanda[3], arcade[4] o indie[5].

Este proyecto tiene como objetivo el desarrollo de un clon del juego ***Bomberman*** [1], destacando como puntos principales la inclusión de inteligencia artificial en la creación de enemigos y la generación automática de niveles. El desarrollo del juego se basará en ***XNA***[6], el API[7] para la implementación de videojuegos que proporciona ***Microsoft***; encajando una vez desarrollado en el grupo de los videojuegos denominados indie.

Índice general

Índice de figuras

Índice de código

Índice de tablas

Capítulo 1
Introducción

Un enfoque general de los propósitos de este proyecto se presenta en este capítulo. La motivación, los objetivos y los contenidos son los puntos principales.

El primer apartado de este capítulo, el 1.1 Motivación del proyecto, presenta la motivación, el por qué de este proyecto.

El segundo apartado, el 1.2 Objetivos, expone los objetivos a alcanzar.

Por último en el tercero, el 1.3 Contenidos de la memoria, se introducen los contenidos principales del proyecto aportando una visión general de este.

1.1. Motivación del Proyecto

La industria de los videojuegos es el sector económico encargado del diseño, desarrollo, distribución y venta de videojuegos, englobando decenas de disciplinas de trabajo y empleando a miles de personas alrededor del mundo. Esta industria ha experimentado en los últimos años altas tasas de crecimiento, debido al desarrollo de la computación, la capacidad de procesamiento, imágenes más reales, las redes o los dispositivos de almacenamiento.

Actualmente este sector está considerado uno de los más potentes dentro de la industria del ocio, generando en los últimos años más dinero que el cine o la música. El récord de la película más vista en el primer día de exhibición lo tiene *Batman: The Dark Knight*, con 66.4 millones recaudados el día de su estreno, nada comparado con los 310 millones que recaudó *Grand Theft Auto IV* [2] en su primer día de ventas, a pesar de ser un título no recomendado a menores de 18 años.

En cuanto a datos concretos en España, el crecimiento a lo largo de los últimos años ha sido constante a pesar de la crisis. Desde los 960 millones en ventas alcanzados en 2006, hasta los 1.432 millones de 2008, la industria ha crecido casi un 50% superando la suma de la taquilla de cine (644 millones), la venta de películas (362 millones) y la música grabada (284 millones) juntas (según los datos de la Asociación de Desarrolladores y Editores de Software de Entretenimiento [3] (aDeSe)).

El mundo de los videojuegos ha evolucionado y se ha expandido para evitar los descensos de ventas del antiguo modelo de negocio. Este se basaba en la salida de las nuevas consolas y caía demasiado rápido hasta la salida de una nueva generación. Ahora los nuevos dispositivos ponen al servicio del usuario juegos de muchos tipos, abriendo el mercado a usuarios que nunca habían jugado a los videojuegos.

Este nuevo modelo se ha visto impulsado principalmente por los juegos casuales, arcade e indie. Los casuales abren el mercado a nuevos usuarios que no se veían atraídos por los juegos tradicionales. Los indie además, lo abren a desarrolladores que no podían afrontar el desarrollo de un juego comercial.

El mundo de los juegos arcade (juegos no comerciales que se ofrecen a través de plataformas online), se ha potenciado gracias a los nuevos sistemas que presentan las consolas de última generación. Estas proporcionan API's y plataformas de desarrollo, puntos de venta y una lógica de negocio muy cercana al desarrollador y al jugador.

Este proyecto se puede considerar como el primer paso en el mundo del desarrollo de videojuegos amateur. Tomando el API de *Microsoft XNA* como base, y la consola *XBOX 360* y el PC como plataformas finales; se pretende diseñar y desarrollar un clon del videojuego "*Bomberman*". Los puntos a destacar del juego serán la inclusión de inteligencia artificial, mediante el desarrollo de distintos tipos de enemigos y la generación automática de niveles, lo que ofrecerá una experiencia distinta en cada partida.

"*Bomberman*" es un juego de estrategia basado en laberintos, desarrollado por ***Hudson*** [4] a mediados de los ochenta y del cuál siguen desarrollándose versiones hoy día. A lo largo de los años han aparecido muchas versiones del juego, que ha su vez incluían muchos modos de juego; pero la mecánica del modo original era la siguiente:

- El jugador debe ir abriéndose camino en un laberinto 2D formado por bloques rompibles e irrompibles.
- El jugador dispone de bombas que explotan de forma automática tras un tiempo y que le permiten derribar las paredes rompibles abriéndose camino por el laberinto.
- Al eliminar una pared el jugador puede encontrar ítems[8], los cuáles le afectarán de forma positiva, permitiéndole por ejemplo poner más bombas o detonarlas de forma manual.
- Dentro del laberinto existen enemigos que, si chocan con el jugador, lo matarán.
- El jugador puede acabar con estos enemigos utilizando las bombas.
- Se dispone de un tiempo determinado para eliminar a todos los enemigos de un nivel.
- Una vez que se han eliminado todos se puede avanzar al siguiente nivel.

1.2. Objetivos

Los objetivos principales planteados en este proyecto fin de carrera son tres:

1. El objetivo general de desarrollar un videojuego utilizando la herramienta *Microsoft Visual Studio .NET* y el API para desarrollo de videojuegos de *Microsoft XNA 3.1*, en concreto, un clon del videojuego "*Bomberman*".

2. La inclusión de inteligencia artificial en el juego a través del diseño de enemigos. Este objetivo surge porque ya se habían realizado proyectos previos a este que cubrían el desarrollo de un juego con las herramientas comentadas, pero ninguno incluía módulos de este tipo. Inicialmente los enemigos debían simular el comportamiento del jugador, pero posteriormente se optó por implementar varios tipos de enemigos similares a los que podemos encontrarnos en el modo de juego original de "*Bomberman*". Más adelante se explicará qué tipos de enemigos se han desarrollado, así como su comportamiento, detalles relacionados con su implementación etc.

3. El tercero es la inclusión de un generador de niveles. Su función será generar los niveles automáticamente, incrementando la dificultad de forma incremental según el jugador va avanzando en el juego.

Por otro lado, el desarrollo de este proyecto pretende cubrir los siguientes objetivos personales o secundarios:

1. El del desarrollo de un juego o videojuego, puesto que a lo largo de los cinco cursos de Ingeniería Informática, nunca se ha afrontado la implementación de ninguno en ninguna de las asignaturas cursadas.

2. La realización del proyecto proporciona la oportunidad de conocer varias herramientas, *Microsoft Visual Studio* y *XNA*, y un lenguaje C#, que hasta ahora se desconocían; así como la importante experiencia del desarrollo de un videojuego completo.

Respecto a los objetivos hay que aclarar que, cuando se acudió a solicitar este proyecto, se hizo con la idea de desarrollar un juego similar a *Super Gussun Oyoyo* [5]. Tras un primer análisis, se decidió que no encajaba con alguno de los objetivos principales del proyecto como la aplicación de inteligencia artificial, por lo que se desechó. Sin embargo este proyecto sirve como punto de introducción a la herramienta y proporciona muchos de los conocimientos necesarios para afrontar su implementación en un futuro.

Para terminar, se detallan una serie de puntos intermedios que deben alcanzarse para conseguir los objetivos marcados.

- **Profundizar en la herramienta Visual Studio .NET y el API XNA.**

Puesto que al inicio del proyecto la experiencia con *Visual Studio* es muy reducida (y nunca para el desarrollo de este tipo de aplicaciones) y con *XNA* nula, el primer paso es afianzar conceptos y profundizar en los conocimientos y la mecánica de trabajo de dichas herramientas.

- **Efectuar un análisis de requisitos.**

Se debe realizar un análisis de requisitos para el juego que determine las opciones básicas que incluirá, el comportamiento del jugador y de los enemigos, los ítems a los que puede acceder el jugador y cómo le afectarán, si habrá niveles de dificultad y cómo varía el juego en función de estos.

- **Desarrollo y validación.**

Durante el desarrollo se irán cubriendo una serie de hitos que generarán versiones del juego las cuales serán validadas por el tutor. Estas validaciones servirán de feedback en el proceso de desarrollo y ayudarán a que la versión final cubra todos los requisitos.

Los hitos más importantes que se deben llevar a cabo los siguientes:

- Conocer los métodos básicos y la estructura de un juego en XNA.
- Diseño del jugador, sus movimientos y selección del método de evaluación y detección de colisiones.
- Diseño del mapa, actualización de sus elementos e integración del jugador.
- Diseño de los enemigos, integración en el mapa y selección del método de evaluación y detección de colisiones.
- Diseño de pantallas, menús, elementos gráficos y de audio tales como animaciones, iconos, efectos de sonido etc.
- Evaluación del juego, su jugabilidad, el comportamiento de los enemigos, la generación de los niveles etc.

Estos son a grandes rasgos los hitos que deben cumplirse de forma que, si quedan todos cubiertos, se alcanzarán los objetivos propuestos del juego.

- **Viabilidad del proyecto**

Por último debemos destacar que se ha realizado una planificación previa para evaluar la viabilidad del proyecto. Para más información sobre esta planificación se remite al lector al Anexo A, Planificación.

1.3. Contenidos de la memoria

Los contenidos de los que se compone la presente memoria están organizados en forma de capítulos. Cada uno de ellos profundiza sobre un aspecto concreto perteneciente a este proyecto. A continuación se muestra un breve resumen del contenido de cada capítulo.

Capítulo 1, **Introducción**. Se ofrece una primera aproximación a la idea en torno a la cual gira el proyecto, el crecimiento y la evolución del mundo de los videojuegos y las nuevas posibilidades para un desarrollador en este mundo. Además, acerca al lector a los objetivos que han motivado la realización de este proyecto.

Capítulo 2, **Estado de la Cuestión**. Realiza un repaso a la historia de los videojuegos: empresas, juegos, consolas; ofreciendo una visión desde su nacimiento hasta nuestros días. También se incluye un repaso de la historia del videojuego "*Bomberman*", de forma que sirva de acercamiento al juego, su concepto, evolución, y como ha llegado a ser la insignia de *Hudson*, la empresa que lo desarrolló. Por último, se incluye un análisis de la herramienta utilizada para el desarrollo, así como una pequeña comparación con otras herramientas similares; de forma que queda justificada la elección de *Visual Studio 2008* y *XNA* en su versión 3.1 para el desarrollo del juego.

Capítulo 3, **Análisis, diseño e implementación**. Recoge el análisis de la arquitectura y el modelo de conocimiento del juego incluyendo el diseño de clases, la descripción de los distintos módulos con su funcionalidad, entradas y salidas; así como el detalle de los algoritmos seleccionados para la detección de colisiones, el comportamiento de los enemigos, la interfaz gráfica y la justificación de la doble visión en el diseño, "lógica" o "física", de algunos algoritmos en función de si se aplican al jugador o a los enemigos.

Capítulo 4, **Conclusiones**. Expone las reflexiones obtenidas tras la realización de este proyecto, analizando además si se han alcanzado los objetivos marcados en un principio.

Capítulo 5, **Líneas Futuras**. Hace referencia a las ideas e intenciones futuras que han surgido tomando este proyecto como base; puesto que este es sólo un punto de partida para aquel que quiera continuarlo.

Por último comentar que, al final de este documento, se encuentran los distintos **Anexos** que recogen la **Planificación** (Anexo A) y el **Manual de Usuario** (Anexo B). También un par de puntos adicionales: el **Diccionario de Términos y Acrónimos**, que recoge una definición de los términos que pueden resultar ambiguos al lector, pero que a la vez son básicos para la comprensión total del proyecto. Y las **Referencias**, que presenta todas las fuentes que se han consultado para profundizar en los distintos temas expuestos en el proyecto.

Capítulo 2
Estado de la cuestión

Un repaso a la historia de los videojuegos, profundizando en la historia de *Bomberman* y un análisis de XNA como herramienta de desarrollo componen este capítulo.

El primer apartado de este capítulo, el 2.1 Historia de los videojuegos, incluye un repaso a la evolución del mundo de los videojuegos.

El segundo apartado, el 2.2 Historia de Bomberman, analiza el juego y su mecánica en sus distintas versiones.

Por último el tercero, el 2.3 XNA Game Studio, analiza las distintas herramientas para el desarrollo de videojuegos y justifica la elección de XNA.

2.1. Historia de los videojuegos

Mucho antes de la aparición del primer videojuego, *Pong*, existían máquinas autómatas [6] que podemos considerar como prehistoria de los videojuegos actuales. Estas máquinas se basaban en principios eléctrico-mecánicos y suponen la primera versión de las actuales máquinas recreativas. Entre ellas se encuentran *Automated Skill Shooter*, *Erie Digger*, *Play the Derby*, *The Boxers*, o *Booz Barometer*.

No se profundizará en su funcionamiento ni su evolución, pero era necesario destacar su existencia, puesto que se consideran un punto de referencia en lo que a entretenimiento electrónico se refiere.

2.1.1. Las primeras empresas

Existen varias empresas que nacieron antes de la aparición de *Pong* en **1972**, y que posteriormente se involucraron en el mundo de los videojuegos ya sea como creadores, desarrolladores o simples distribuidores. Muchas de ellas se dedicaron en un principio a otros negocios y se introdujeron con el tiempo en los juguetes o el entretenimiento y posteriormente en los videojuegos. En este punto hablaremos de las más importantes, profundizando algo más en algunas que ya han abandonado el negocio y pasando algo más por encima de otras que se completarán en el apartado de consolas.

La primera empresa fue ***Nintendo*** [7], fundada por *Fusajiro Yamauchi* en Kyoto en **1889** con el nombre de *Marufuku Company* y cuyo producto principal eran unas cartas denominadas *Hanafuda* [8]. Posteriormente, en **1933** Yamauchi cambiaría el nombre de la empresa a *Yamauchi Nintendo & Co*. Nintendo significa en japonés NIN (responsabilidad), TEN (el cielo o los cielos), DO (templo, salón). Se puede interpretar su significado completo como "confía en lo que el cielo diga" o "en las manos del cielo". En **1951** *Hiroshi Yamauchi*, nieto de Fusajiro, asume la presidencia de la empresa y cambia de nuevo el nombre a *Nintendo Playing Card Co. Ltd*. En **1959**, cuando todavía era una empresa que se dedicaba a los juegos de cartas, llega a un acuerdo con *Disney* para crear cartas con

sus personajes, lo que le dio mucha popularidad. Por último en **1963** el nombre cambiaría de nuevo, quedando el actual *Nintendo Co. Ltd*.

Después de Nintendo, apareció ***Coleco*** [9] "*Connecticut Leather Company*" en **1932.** Creada por la familia *Greenberg*, la actividad principal durante sus primeros años fue el la venta de juguetes de plástico y cuero. Posteriormente bajo la dirección de *Arnold Greenberg*, la empresa entra en el negocio de las consolas de videojuegos con *Telstar* en **1975**. Durante varios años se dedica a desarrollar consolas propias y clones de otras como por ejemplo el de *Atari 2600*. Cuando el mercado del videojuego comenzó a caer en **1983**, y parecía que las videoconsolas desaparecerían debido al éxito de los ordenadores domésticos, *Coleco* lanza su versión; *Coleco Adam* [10]. Debido a que eran poco fiables, a finales de **1984** *Coleco* se retira completamente de la electrónica al borde de la bancarrota dedicándose a los juguetes hasta **1989**; año en el que quiebra y es adquirida por *Hasbro*. En **2005**, *River West Brands*, adquiere el nombre y los derechos de *Coleco*.

En **1945,** *Harold Matson* y *Elliot Handlet* fundan ***Mattel*** [11], el nombre surge de una fusión de los suyos "*Matt-El*" y en sus comienzos fue una fábrica de marcos para cuadros. Con los excedentes y restos de madera de la producción, se iniciaron en el negocio de los juguetes construyendo casas de muñecas. La prosperidad del negocio les llevó a crear más tarde una división de juguetes, que con el tiempo se ha convertido en la empresa juguetera más potente del mundo. En **1979**, se introducen en el mundo del ocio electrónico con la consola *Intellivision* [12], desarrollada en sólo un año como contrapunto de su principal competidor, *Atari 2600*. Fue la primera consola que significó una seria amenaza al dominio de *Atari* y su éxito llevo a *Mattel* a desarrollar periféricos como un teclado o un dispositivo de síntesis de voz, el *Intellivoice* [13]. Ambos periféricos resultaron un desastre aunque son precursores de algunos de hoy día como el micrófono de *Playstation 2*. En **1983** *Mattel* lanzó una versión remodelada de la consola, llamada *Intellivision II*, que incluía joysticks desmontables, un accesorio que permitía jugar con juegos de *Atari 2600* y un teclado musical opcional. La consola dejó de fabricarse con el tiempo pero hasta **1991** se siguieron lanzado juegos. Actualmente *Mattel* no desarrolla juegos directamente, pero si los produce vendiendo la imagen de algunos de sus juguetes como *Barbie* o *Hot-Wheels*.

En **1947** *Akio Morita* y *Masaru Ibuka* fundan ***Sony*** [14], inicialmente conocida como *Tokio Telecommunications Engineering Company*. En sus inicios se encargaba de reparar radios, y hacía

convertidores de onda corta o adaptadores, para convertir radios de onda media en radios portátiles. Posteriormente en **1958**, cambiaría el nombre por el cual se conoce hoy en día *Sony* (derivado de la palabra "*sonus*", sonido en latín). A lo largo de su trayectoria se ha convertido en una de las compañías electrónicas más importantes del mundo.

En **1952** *Stewart* y *Lemaire* fundan ***SEGA*** [15], aunque inicialmente lo harían con el nombre de *Service Games*, dedicada al mantenimiento y venta de máquinas de discos jukebox[9] en las bases militares de Japón. Más adelante, en **1965**, *Stewart* y *Daven Rosen* (antes dedicado a la importación de máquinas recreativas mecánicas de EEUU a Japón) unen sus compañías *Service Games* y *Rosen Enterprises* y cambian el nombre de la empresa por el de *SEGA Enterprises Ltd*. A partir de ahí *SEGA* empezó a fabricar juegos mecánicos como *Rifleman*, *Periscope* y *Jet Rocket* [16]; dedicándose por completo al negocio de juegos electrónicos y recreativas y entrando de lleno en el mundo de los videojuegos.

En **1953** se crea en Japón ***Taito*** [17]. Aunque la empresa era japonesa, su fundador fue *Michael Kogan*, un ruso judío. El nombre de *Taito* proviene del símbolo japonés que significa "*Judío del lejano oriente*". Su función inicial era la de distribución de máquinas expendedoras de todo tipo. Más tarde también se dedicaron a alquilar máquinas de discos para bares y pubs y empezaron a crear sus propias máquinas. En **1973** desarrolla su primer videojuego y cambia su nombre a *Taito Corporation.* En **1978** creó *Space Invaders*, que se convirtió en uno de los videojuegos más populares de la compañía. Posteriormente en **1986** desarrolló *Bubble Bobble* (en versiones posteriores *Puzzle Bobble* y *Bust-a-Move*), que introdujo nuevas ideas a los juegos de puzzles. En **2005**, Taito pasó a ser propiedad de *Square Enix*.

Un año después, en **1954,** se fundó ***Commodore*** [18]. *Jack Tramiel*, un ex-combatiente de la Segunda Guerra Mundial, fue su fundador y la actividad principal de la empresa era reparar máquinas de escribir. Con el tiempo *Tramiel* comenzó a fijarse en las sumadoras electrónicas enfocando el negocio en la electrónica. En **1970** absorbió *MOS Technology* e introdujo el *KIM-1* [19], un ordenador programable en lenguaje máquina. A partir de entonces tuvo un papel principal en el desarrollo de la industria del ordenador personal en la década de los 80. En **1982** *Commodore* desarrolló y comercializó el ordenador de sobremesa más vendido a nivel mundial, el *Commodore 64* [20], pero no le fue tan bien con el *Commodore Amiga* [21]. En **1994** se declaró en bancarrota, y

pasó por diversos propietarios. En **2005** llega a manos de *Yeahronimo Media Ventures*, que desde entonces ha centrado la compañía en los equipos con moddings orientados a jugadores.

Posteriormente, en **1968**, *Alan Sugar* fundó ***Amstrad*** [22] cuyo nombre surge de **"***Alan Michael Sugar Trading*". Inicialmente *Amstrad* construye televisores y radios y a partir de **1970** lidera el mercado. Este liderazgo permitió su expansión a la producción de amplificadores de audio y sintonizadores. En **1980**, *Amstrad* comenzó a comercializar sus propios ordenadores personales intentando competir con *Commodore* y *Sinclair*. La compañía presenta el *CPC 464* [23] en **1984**. En muchas zonas de Europa fue un éxito y aunque no logró desbancar a los competidores, consiguió que todos los videojuegos de la época se versionarán en sus sistemas. En **1985** introdujo el *Amstrad PCW* [24], que se comercializa como procesador de textos y que arrasa el mercado. *Amstrad* entró brevemente en el mercado de consolas de videojuegos con la *GX4000*, pero nunca alcanzó mucha popularidad. En **1986**, *Amstrad* adquiere *Sinclair Research* y los derechos del *ZX Spectrum*, lanzando posteriormente tres nuevas versiones. También en **1986**, la compañía produjo una gama de ordenadores personales económicos basados en *MS-DOS* y posteriormente en *Microsoft Windows* siendo todo un éxito. A partir de **1990** *Amstrad* empezó a centrarse en ordenadores portátiles más que en equipos de escritorio e intentó entrar otra vez en el mercado de las videoconsolas con la *Amstrad GX4000* [25]. Pero la máquina fue un fracaso, ya que sus competidores eran muy potentes (*SEGA Mega Drive* y *Super Nintendo*). A partir de entonces la compañía empezó a concentrarse más en comunicaciones y menos en ordenadores dedicándose a partir del año **2000** a los receptores de televisión digital. Desde **2006** el área de negocio principal de *Amstrad* es la fabricación de receptores interactivos de *Sky TV* [26].

Por último, en **1969**, nace ***Konami Corporation*** [27] como un negocio de reparación de jukeboxes en Japón. Fue fundada por *Kagemasa Kozuki*, quien todavía hoy es su presidente y CEO[10]. Su nombre significa *Kagemasa Kozuki, Yoshinobu Nakama*, *Hiro Matsuda* y *Shokichi Ishihara*, todos los fundadores de la compañía. En **1973**, *Kozuki* cambió el nombre de la empresa a *Konami Industry Co., Ltd*. y comenzó a trabajar en la construcción de máquinas arcade. La empresa comenzó a tener éxito con juegos como *Frogger*, *Scramble*, y *Super Cobra*. Entre **1982** y **1985**, *Konami* desarrolló y vendió juegos para PC y las consolas *MSX* y *Famicom* de *Nintendo* (en EEUU y Europa conocida como *Nintendo Entertainment System NES*). Los mejores juegos de *NES* fueron producidos por *Konami*, incluyendo las sagas *Castlevania* y *Metal Gear*. En **1992**, algunos

miembros de *Konami* formaron *Treasure Co. Ltd.* [28], conocida en el mundo de los videojuegos como creadora de juegos de acción. En el año **2005**, *Konami* se convirtió en el dueño de *Hudson Soft*. Actualmente *Konami* es el cuarto desarrollador de videojuegos más grande en Japón después de *Nintendo Co, Ltd.*, *SEGA Sammy Holdings*, y *Namco Bandai Holdings* y posee licencias tan productivas como *Pro Evolution Soccer* o *Metal Gear Solid*.

Puesto que se considera **1970** como la década en la que nacieron los videojuegos, sólo se tienen en cuenta las empresas anteriores a esta fecha como verdaderas precursoras. Por ese motivo este punto finaliza aquí. Muchas otras empresas han aparecido desde esa fecha en el mundo de los videojuegos e incluso han llegado a dominarlo, como por ejemplo *Microsoft*, que en muchos sentidos lidera el mercado con *XBOX 360* [29]; pero sólo las comentadas anteriormente se pueden considerar como las verdaderas pioneras.

Una vez cerrado este tema, se hará referencia al concepto propio de videojuego. Se profundizará en los antecedentes del *Pong,* publicado en **1972** y considerado por todos como primer videojuego de la historia.

2.1.2. Los precursores de los videojuegos

Como ya se ha comentado en el punto anterior, el año **1972** es el que marca la aparición del primer videojuego, pero la idea y los primeros intentos surgen casi 30 años antes. En este punto se hace referencia a todas las creaciones previas al ya mítico *Pong* y que se consideran precedentes de lo que más adelante serían los primeros videojuegos de la historia.

Quizá la primera piedra en la historia de los videojuegos se puso en **1947**. Es cierto que hasta este momento de la historia la funcionalidad y la evolución de las máquinas estaba condicionada a su utilización en conflictos armados (la Segunda Guerra Mundial duró hasta **1945**); pero a partir de entonces la inversión en tecnología da un espaldarazo al desarrollo de otras posibilidades, cuyo potencial real sería explotado dos décadas después.

Así en **1947**, aparece el denominado como "*Primer Experimento Electrónico de Simulación en Pantalla*". *Thomas T. Goldsmith* y *Estle Ray Mann* patentaron ***Lanzamiento de Misiles*** [30], un sistema electrónico de juego que consistía en simular el lanzamiento de misiles contra un objetivo. Se trataba de una adaptación de un rádar como los que se usaban en la época en los buques armados, con válvulas que se proyectaban sobre una pantalla de rayos catódicos. Las funciones que tenía eran ajustar curva y velocidad del disparo. No presentaba movimiento en pantalla ya que los objetivos estaban sobre impresionados; por lo que en realidad no se le puede considerar un videojuego. No queda ninguna representación gráfica de él, sólo alguna descripción por anotaciones de la época, que queda representada el la figura 1.

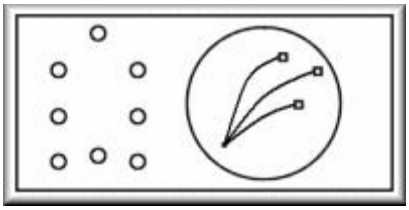

Figura 1: Diagrama de Lanzamiento de Misiles

Siete años después de la aparición de *Lanzamiento de Misiles*, en **1952**, *Alexander Sandy Douglas* presenta su tesis de doctorado en matemáticas en la Universidad de Cambridge sobre la interactividad entre seres humanos y computadoras. Esta tesis incluye el código del que se considera primer juego gráfico computerizado denominado ***OXO*** [31]. Dicho juego era una versión del “Tres en Raya" escrito para la computadora *EDSAC* [32] (el primer calculador electrónico con capacidad para almacenar algunos programas). *OXO* tomaba decisiones en función de los movimientos del jugador, que transmitía las órdenes a través de un dial telefónico integrado en el sistema. Existen discrepancias a la hora de considerar a *OXO* como el primer videojuego de la historia, ya que no contaba con ningún tipo de animación de vídeo. No fue un juego muy conocido ni llegó a ser comercializado.

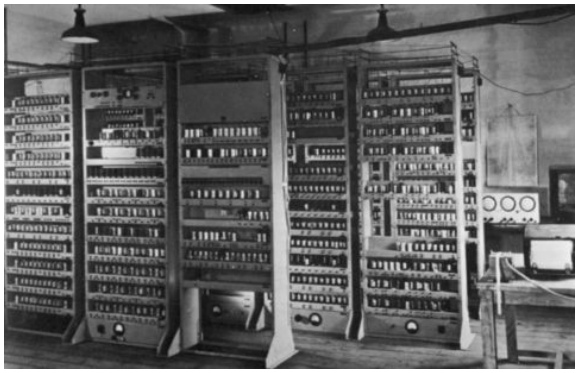

Figura 2: computadora EDSAC

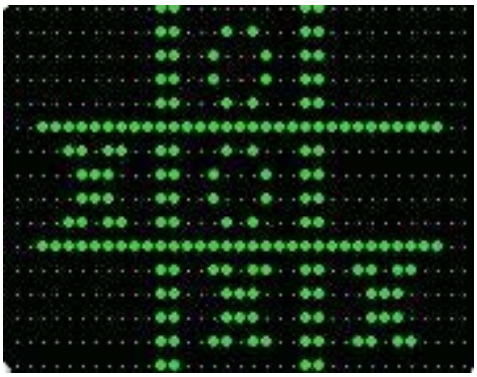

Figura 3: Pantalla de OXO

En **1958** *William Higinbotham* programa ***Tennis for Two*** [33], el primer juego con animación gráfica y que permitía la interacción entre dos jugadores. Se jugaba en la pantalla de un "Osciloscopio" de laboratorio conectado a un computador, donde se mostraban dos líneas perpendiculares que simulaban el suelo y una red de pista de tenis. El objetivo era simple: jugar al tenis con dos raquetas (dos líneas móviles) que interceptaban una pelota (un punto en oscilación) en la pantalla. Se presentó como una curiosidad científica y nunca se patentó, ya que su autor lo creó para entretener a los visitantes del *Brookhaven National Laboratory*. Quince años más tarde, en **1972**, *Atari* lo comercializó con el nombre de *Pong* y un formato algo más simple. Probablemente *William Higinbotham* nunca llegó a pensar en la transcendencia de su invento en el mundo futuro de los videojuegos.

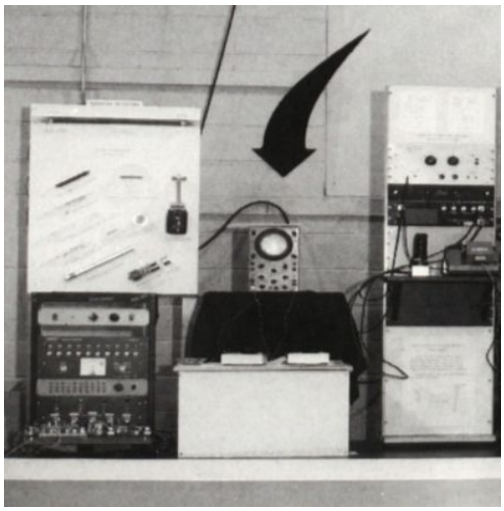

Figura 4: Ociloscopio

Figura 5: Pantalla de Tennis forTwo

En el verano de **1961**, *Digital Equipment* donó uno de sus ordenadores más potentes, el *PDP-1* [34] al *MIT* "*Instituto Tecnológico de Massachussets*". Varios miembros del *MIT Wayne Witanen*, *Martin Graetz* y *Steve Russell* fueron los encargados de decidir sobre qué aplicación podrían diseñar para aprovechar las posibilidades del nuevo ordenador. Decidieron que debían crear algo que mezclara acción y habilidad y pensaron en diseñar un juego en el que se pudiera controlar objetos moviéndose por la pantalla. Finalmente diseñaron ***Spacewar*** [35] [36], un juego para dos jugadores, en el que cada uno manejaba una nave espacial e intentaba disparar a la otra. Tenían un depósito de combustible y varios tipos de armas. El escenario también implicaba ciertos riesgos, como no calcular de forma precisa el giro y morir quemado en el sol, cuya gravedad atraía a las naves hasta destruirlas si las alcanzaba, o usando la función "*hiperespacio*"; que hacía aparecer la nave en un lugar aleatorio de la pantalla pudiendo también terminar destruida si aparecía demasiado cerca del sol. El código de *Spacewar* llegó a muchos ordenadores circulando por la red de *Arpanet*[11] (precursora del actual *Internet*) y de mano en mano en cintas programables, de manera que se jugaba en casi todas las universidades que contaban con un ordenador con pantalla. *Spacewar* es considerado como el primer videojuego para ordenador de la historia.

Figura 6: Imagen de PDP-1

Figura 7: Pantalla de Spacewar

Una década después, en **1971**, apareció el que se puede considerar como primer videojuego comercial de la historia: ***Galaxy Game*** [37]. Esto no es una contradicción con respecto a considerar a *Pong* como el primer videojuego, ya que Pong fue comercializado y *Galaxy Game* sólo apareció en recreativas. *Galaxy Game* fue creado por dos estudiantes de la *Universidad de Stanford*, *Bill Pitts* y *Hugh Tuck*, a partir del código original de *Spacewar*. La máquina se instaló dentro del complejo del

campus de la universidad y se considera por tanto como la primera máquina de videojuegos en ser instalada en un sitio público. El éxito fue enorme. Hoy día está expuesto en el *Museo de la Computación* [38].

Figura 8: Recreativa Galaxy Game

Un par de meses después ese mismo año, **1971**, se produjo el siguiente paso lógico y apareció ***Computer Space*** [39], la primera máquina en ser producida en serie. En realidad el juego no es conocido por la gran mayoría de la gente, pero lo importante no tanto el juego como su creador: *Nolan Bushnell* quien posteriormente fundó *Atari*. *Bushnell* fue contratado por *Ampex* [40] en **1969** y en sus ratos libres trató de hacer su propia versión de *Spacewar* que no necesitara de un ordenador para mostrarse en pantalla. Esto redujo mucho los costes en componentes para el juego. Tras varios meses de trabajo consiguió terminarlo y se asoció con *Nutting Associates* fabricando alrededor de 1.000 recreativas de *Computer Space* (incluso se llegó a fabricar una versión para dos jugadores). Aunque en un principio funcionó bastante bien, el intento global fue un fracaso del que *Bushnell* dijo aprender mucho. Según él, todo este proceso le dio las directrices básicas para comenzar a gestar su verdadera idea, que más tarde llevaría acabo fundando *Atari* [41].

Figura 9: Recreativa Computer Space

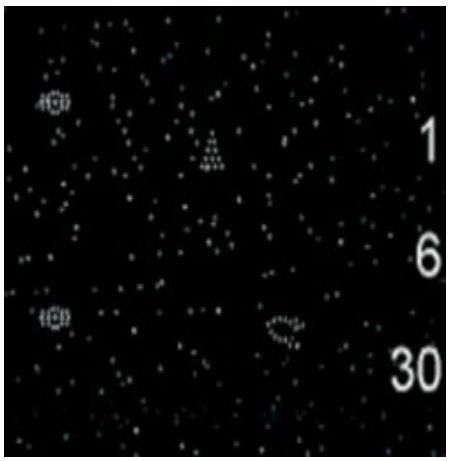

Figura 10: Pantalla de Computer Space

Aquí termina este punto puesto que los siguientes grandes hitos en la historia de los videojuegos *Magnavox Odyssey* y *Atari* llegaron de la mano de *Pong* en **1972**. En los puntos siguientes se trata cada una de las décadas desde los 70 hasta nuestros días, comentando los hechos más importantes relacionados con la historia de los videojuegos.

2.1.3. Los videojuegos en los setenta

Ya se ha hecho referencia en puntos anteriores al año **1972** y que se considera un año clave en la historia de los videojuegos debido a la aparición de la *Magnavox Odyssey*, *Atari* y *Pong*. En esta sección se detallará lo que pasó en la década de los 70 en el mundo de los videojuegos.

Tan importante como la aparición de *Pong* fue la aparición de la compañía ***Magnavox*** [42] y su ***Magnavox Odyssey*** [43], en **1972**. Se considera *Magnavox Odyssey* como la primera consola de la historia, y su creador fue *Ralph Baer*. Este hecho hace que *Ralph Baer* sea considerado por muchos como el inventor de los videojuegos tal y como se conocen hoy en día, además de ser el inventor de las videoconsolas.

La intención de *Baer* era desarrollar un sistema de videojuegos para jugar en casa. A partir de esta idea propuso a su jefe (dueño de una fábrica de televisores) agregar a uno de los televisores un sistema de juego pero su jefe rechazó la propuesta tajantemente. En **1966** desarrolló por su cuenta la

primera consola doméstica de videojuegos pero durante años no encontró empresas o inversores que apostaran por su idea.

En **1967**, *Bill Harrison* se unió él y diseñó un sistema que permitía utilizar pistolas ópticas como periférico para disparar a la pantalla. Más tarde ese mismo año se unió al grupo de desarrollo *Bill Rusch*, quien según ellos fue fundamental por su concepto de jugabilidad y su capacidad para llevarla a los juegos. A finales de **1967** consiguieron hacer funcionar un juego de ping pong para dos jugadores.

Una vez desarrollada esta primera versión llegó el momento de la venta. La primera compañía interesada fue *TelePrompter Corporation* (una compañía de televisión por cable) que en **1968** tenía la idea de sacar un canal de juegos en su paquete. A pesar de ver demostraciones y quedar muy contentos el proyecto nunca se llevó a cabo. Tras este intento fallido desecharon la idea de los juegos por cable y ofrecieron la máquina a todos los fabricantes de productos eléctricos que pudieron llegando a negociar un contrato con *RCA* que finalmente no se firmó. Posteriormente *Bill Enders* (de *RCA*) dejó la empresa y comenzó a trabajar para *Magnavox*, donde les habló del invento de *Baer* y su grupo. *Magnavox* se acercó a ver la máquina y cerró el trato a finales del año **1971**.

Finalmente en **1972** se lanzó *Magnavox Odyssey*. Era una consola poco potente que sólo era capaz de dibujar un punto móvil y barras verticales en la pantalla, por lo que era necesario jugar con plantillas que la empresa también proporcionaba con la consola. No tenía ni un solo circuito integrado, estaba compuesta por 40 transistores y 40 diodos, y los cartuchos eran jumpers[12]. Ofrecía 12 juegos diferentes, entre ellos de dados y cartas, combinando los elementos que traía en la caja. La consola sólo era compatible con televisores de *Magnavox*.

El 1 de Junio de **1972** nació ***Atari***, *Nolan Bushnell* junto a *Ted Dabney* y *Larry Bryan*, ambos de *Ampex*, idearon el proyecto poniendo cada uno 750 $. *Bryan* se echó atrás en el último momento y *Bushnell* y *Dabney* se quedaron solos al frente. Eligieron *Syzygy* como nombre para la empresa pero una compañía local ya usaba ese nombre, así que finalmente lo descartaron y se decidieron por *Atari* que proviene del juego de mesa *Go* [44]. *Atari* es la palabra utilizada cuando se consiguen rodear las piedras del oponente y se va a ganar la partida.

Los primeros proyectos de *Atari* se hicieron gracias al dinero que habían conseguido con *Computer Space*, pero poco tuvieron que ver con los videojuegos. El primer cliente fue *Bally*, una empresa dedicada a las máquinas tragaperras y los pinballs. Pero cuando *Bushnell* trabajaba con *Nutting*, todo cambia puesto que se enteraron de que *Magnavox* estaba promocionando públicamente un sistema doméstico de videojuegos. *Bushnell* fue a verlo y vio el juego de ping pong de *Baer*, lo que le llevó a crear un juego similar para *Atari*.

Se le encargó el desarrollo de juego de ping pong a *Allan Alcorn*, un ingeniero de *Ampex*. Mientras *Bushnell* y *Bally* comenzaron a diseñar un juego de carreras. *Alcorn* eliminó componentes, abarató el producto y le dio una nueva forma cambiando las palas de ping pong por rectángulos divididos en ocho secciones diferentes en los cuales la bola rebotaba de diferente manera y añadiendo aceleración a la pelota. *Alcorn* tardó unos tres meses en conseguir crear un prototipo, al cual lo llamó ***Pong*** [45], y para probarlo lo introdujeron en los bares donde tenían instalados pinballs de *Bally*. *Pong* fue un éxito rotundo y *Bushnell*, con gran visión comercial, quiso atarlo para *Atari*. Él había apalabrado con *Bally/Midway* un juego creado por *Atari* pero hizo creer a cada una de las partes, *Bally/Midway* por separado, que a los otros no les interesaba el proyecto *Pong*. Finalmente, ninguna de las dos partes lo quisieron y *Bushnell* consiguió que *Atari* comercializase el juego en solitario. *Pong* fue el mayor éxito de los videojuegos hasta ese momento, y cambió para siempre esta industria.

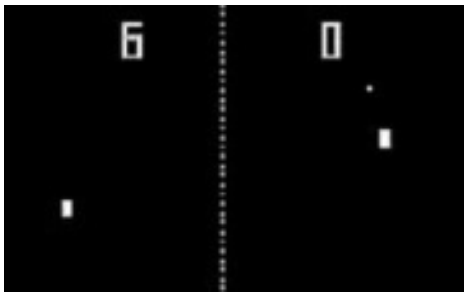

Figura 11: Pantalla del juego Pong

1973 es el año en que ***Taito*** lanza su primer juego, ***Elepong*** [46], un clon de *Pong,* y supone un año de evolución y crecimiento para *Atari*. *Bushnell* pensaba que había que expandir el mercado para que *Pong* se vendiera bien, y sobre todo para que los juegos que vinieran después de *Pong* pudieran seguirse vendiendo a buen ritmo. Las cosas le iban tan bien a *Atari* que era imposible

mantener la demanda. A pesar de los primeros problemas, *Pong* se había convertido en la recreativa más rentable de la historia. Se estima que cada recreativa recogía más de 200 dólares a la semana, cuando una máquina normal llegaba a 40 o 50 dólares como mucho.

Según iba creciendo *Atari*, *Bushnell* se iba rodeando de personas que compartiesen su visión, despidiendo a quien no lo hacía. La primera baja fue *Ted Dabney*, amigo suyo y cofundador de la compañía. Al principio *Dabney* no quería abandonar la compañía, pero al ver los beneficios obtenidos en la recaudación de *Pong*, y quedarse con algunas acciones, terminó marchándose. Años después, *Dabney* vendería sus acciones y se haría millonario. *Bushnell* fue contratando a gente de la plantilla de *Ampex*, todos jóvenes y con mucha ambición. En esta época incorporó a *Gene Lipkin*, lo que se considera uno de sus grandes aciertos.

Aún en **1973**, *Atari* lanzó su segundo juego llamado ***Space Race*** [47], donde dos cohetes competían por llegar a lo más alto de la pantalla mientras esquivaban asteroides que cruzaban la pantalla de lado a lado y *Midway* lanzó ***Asteroid***, un clon de *Space Race*. *Atari* continuó lanzando juegos en 1973 como ***Pong Doubles*** (versión de *Pong* para dos jugadores) y ***Gotcha*** [48], un juego con poco éxito pero que se consideró precursor de ***Pac Man***.

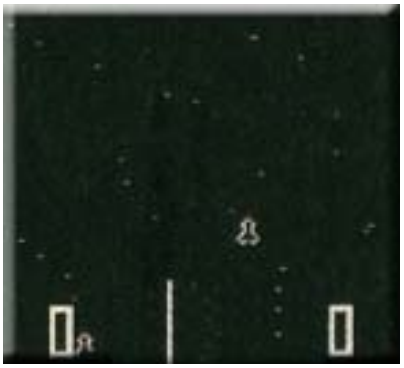

Figura 12: Pantalla del juego Space Race

En año **1974** marca el comienzo de la verdadera competencia a *Atari* con multitud de compañías que copian su estrategia. *Atari* había creado una nueva industria que comenzaba a verse repleta de compañías que querían subirse al carro del éxito. El problema de *Atari* era la imposibilidad de sacar una máquina sin que sus competidores le robasen la idea, puesto que las patentes para sus nuevas recreativas tardaban muchísimo tiempo en llegar. *Bushnell* tomó una

decisión drástica, se centró en sacar nuevos juegos y en investigar ideas revolucionarias que marcasen de forma contundente a *Atari* como la compañía referente de la época. Así comenzó a lanzar en **1974** un juego cada mes. Pero las nuevas ideas no llegaban y el mercado se colapsó.

La situación *Atari* era complicada y para solucionarlo *Bushnell* decidió crear una empresa fantasma. Dicha empresa es ***Kee Games*** [49], fundada junto a *Joe Keenan*, vecino suyo y que había formado parte de *Atari*. Desde el primer momento, las relaciones entre las dos compañías fueron muy tensas e incluso *Atari* acusó a *Kee Games* de espionaje industrial. Pero en realidad todo esto era una farsa, la idea de *Bushnell* fue hacer ver al consumidor que había competencia, darse publicidad y de paso, incrementar el porcentaje de sus máquinas en los salones recreativos.

Con esta estrategia, *Atari* consiguió consolidarse en el mercado, pero también hubo tropiezos. *Steve Bristow* desarrolló un innovador juego llamado ***Tank*** [50], en el que los jugadores controlaban un tanque blanco y otro negro, y se enfrentaban entre sí en un mapa laberíntico. *Tank* representó un avance importante en la industria, al ser el primer videojuego de la historia en utilizar una memoria ROM para almacenar los gráficos. Fue el juego del año sin duda y no era de *Atari*, o al menos por culpa de *Bushnell*, es lo que pensaba la gente.

Este mismo año, otra idea revolucionaria llegó de un pequeño estudio llamado ***Mayer and Emmons***. Comenzó a desarrollar el primer videojuego de carreras de la historia, ***Gran Trak 10*** [51], en el que el jugador utilizaba un volante como mando. Fue muy costoso de desarrollar y aún más de distribuir, pero también fue el juego más vendido de **1974**. A finales de este año ***Taito*** presentó otro juego de carreras, ***Speed Race*** [52].

Al igual que en Estados Unidos, en Japón las recreativas se habían convertido en un buen negocio. ***Namco*** [53], fundada por *Masaya Nakamura*, era la sexta compañía más importante del país. *Atari* creó una rama en Japón para distribuir sus máquinas allí, y *Nakamura* quiso comprar algunas de ellas. Poco después *Bushnell* vendió la rama de la compañía a *Nakamura*. Estos negocios se consideran el prime paso serio en la industria japonesa de los videojuegos.

A finales de **1974** se produjo uno de los hechos más importantes en la industria del videojuego, el prototipo del primer chip que contenía el ***Home Pong*** [54]. *Harold Lee*, un ingeniero

de *Atari*, propuso la idea: una máquina que pudiera conectarse al televisor para jugar a *Pong* en casa y a *Bushnell* le pareció un negocio genial. *Home Pong* costó menos que la *Odyssey*, se veía mejor y solo necesitaba un control para jugar. Pero sólo se podía jugar al *Pong*, mientras que con la máquina de *Magnavox* se podía jugar a 12 juegos. Una vez desarrollada la máquina, trataron de venderla, pero todas las jugueterías y grandes almacenes se negaban pensando que tendría la misma pobre aceptación que la *Odyssey*. Después de recorrer medio país con la máquina, la única salida que le quedaba a *Atari* era el *Toy Show* [55], una feria de juguetes. Allí *Tom Quinn,* representante de *Sears*, ve una demostración y encarga 150.000 unidades (la producción de *Atari* hasta la fecha era de 75.000 y tuvo que recurrir a un crédito para financiar el pedido). Con el dinero obtenido Atari continuó su expansión y alcanzó el mejor momento en toda su historia.

Figura 13: Consola Home Pong

Por último destacar que durante **1974**, *Bill Gates* y *Paul Allen* crean ***Microsoft*** [56]. Al principio se dedican a hacer intérpretes de *BASIC* y más tarde harán compiladores de *Fortran* y *COBOL*. Durante muchos años estuvieron fuera del mundo de los videojuegos, pero más tarde entrarían de lleno en el negocio de las consolas y han llegado a consolidarse como una de las compañías más fuertes. Por otra parte, este año supuso también la inclusión de una nueva compañía en la industria: ***Coleco***, fundada por Maurice Greenberg.

En **1975** ***Nintendo*** da su primer paso en el mundo de los videojuegos. *Yamauchi* observó los avances tecnológicos que había en la época y cómo se aplicaban al entretenimiento, investigando sobre el uso que le habían dado en los EEUU empresas como *Atari* y *Magnavox*. *Yamauchi* negoció una licencia para vender el sistema de videojuegos *Odyssey* en Japón. *Nintendo* no tenía máquinas ni conocimiento sobre los microprocesadores usados en estas máquinas, así que le sugirieron que se aliara con una compañía electrónica. Y así fue, *Nintendo* se alió con *Mitsubishi Electric* [57].

Con el éxito de *Home Pong*, muchas compañías se lanzaron a crear consolas y dispositivos electrónicos para jugar en casa que resultaron clones de los que ya existían. Surgió así un nuevo rival de *Atari*: *Coleco*. *Coleco* diseñó una consola durante todo **1975**: ***Telstar*** [58] y la lanzó a mediados de **1976**. Era un clon más de *Home Pong* más, pero tuvo un éxito enorme ya que *Coleco* era el principal cliente de *General Instrument*, la empresa que suministraba los chips para los juegos de consola, y el resto de empresas que lo pedían se quedaban sin existencias.

Durante este mismo año *Al Alcorn* contrató a *Steve Jobs*. Este comenzó a diseñar un videojuego, ***Breakout*** [59], que en realidad era una idea original de *Nolan Bushnell* desarrollada a partir de *Pong*. En esta ocasión sólo había una pala y en el otro extremo una serie de bloques que había que ir destruyendo con una pelota que iba rebotando en ellos. Además de *Breakout*, *Atari* presentó ***Night Driver*** [60] en él que el jugador conducía por diferentes circuitos que iban apareciendo de repente mientras esquivaba a los demás corredores. Esta formula revolucionaria ha sido explotada durante años hasta la aparición de las 3D en los juegos de carreras.

A finales de **1976** se produce otro hito en la historia de los videojuegos, la primera vez que se hace referencia a la violencia en un videojuego. Por problemas de licencias ***Destruction Derby***, un juego en el que el jugador debe destruir los otros vehículos de la carrera, se convierte en ***Death Race*** [61]; un juego en el que el jugador atropella esqueletos que salen de un cementerio. *Death Race* tuvo muchísimos problemas de distribución en la época.

Hay dos hechos más destacables en **1976**. El primero es la aparición de la ***Channel F*** de *Fairchild* [62], en la que se podía jugar a diferentes juegos guardados cada uno de ellos en un cartucho independiente. El segundo, la revolución que esta consola supuso a pesar de no tener éxito. Los cartuchos[13] pasaron a ser el formato dominante y *Atari* decidió apostar por el diseño de una nueva consola que se basara en este sistema. Esta fue la ***VCS*** "*Video Computer System*" [63]. Como consecuencia de la producción de esta nueva máquina *Bushnell* entró en graves problemas económicos y acabó decidiendo que la mejor forma de obtener el capital necesario era poner a la venta *Atari*. Finalmente la compañía fue vendida a *Warner Communications*, dirigida por *Steve Ross*.

Con la venta de Atari se pasa a **1977,** año marcado por la entrada de *Nintendo* en el mercado**.** Junto con *Mitsubishi Electrics* sacó a la venta el ***Nintendo Tv Game 6*** [64]. Tenía 6 versiones del

juego ***Light Tennis***, una versión mejorada y a cuatro colores del *Pong*. La consola no llevaba cartuchos.

Salto a **1978**, año en el que *Bushnell* anuncia su salida de **Atari** por discrepancias con *Warner*. Años también del lanzamiento de ***Space Invaders*** [65] de *Taito*. La idea fue de *Toshihiro Nishikado*, uno de los programadores de *Taito*, que para uno de sus proyectos se le ocurrió la idea de un soldado parapetado detrás de unas trincheras, que disparaba a filas de enemigos acercándose hacia él. Se cambiaron los enemigos soldados por naves espaciales intentando huir del problema de violencia generado por *Death Race*. Así nace *Space Invaders*, con la siguiente mecánica: el jugador detrás de una serie de escudos debe vencer a 4 filas de enemigos alienígenas que cada vez se aproximaban a él más y más rápido. La idea fue un éxito y además el juego trajo consigo dos innovaciones más que, a la larga, serían fundamentales en el mundo de los videojuegos: el Hi-Score[14] y el juego basado en vidas y no en tiempo. Las dos se basaban en ese instinto de superación que lleva a intentar alcanzar la partida perfecta, batir el record y colocar la puntuación máxima en la máquina.

Fue tal el éxito de *Space Invaders* que llegó a provocar que el gobierno nipón cuadriplicara la producción de monedas de yen, por su escasez a causa del juego. Pero no solo fue éxito en Japón, sino que a través de *Midway* llegó a todo el mundo. Según Taito, ha facturado más de 500 millones de dólares a lo largo de su historia.

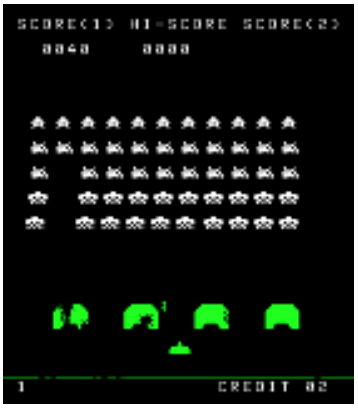

Figura 14: Pantalla del juego Space Invaders

Durante **1978** *Nintendo* continúa su evolución y desarrolla y presenta una nueva consola y un nuevo juego. La consola fue *Nintendo Tv Game 15* [66] que ya presentaba los dos controles con cables. El juego, *BlockBuster* que fue un juego de carreras cortas de gran éxito.

También se lanza en el 78 ***Magnavox Odyssey 2*** [67], convirtiéndose en la segunda generación de consolas *Odyssey*. La tecnología que incluía era bastante similar a la de su competidora, *Atari 2600*, incorporando además un teclado, novedad entre las consolas de 8 bits. Tuvo buena aceptación, pero nunca llegaría a alcanzar la popularidad de *Atari.*

Otra de las pequeñas curiosidades que más tarde se convertirían en un hito es la aparición del primer ***easter egg***[15]**,** o elemento oculto dentro de un juego y que hoy en día son mucho más comunes. El primero llegó en el juego de *Atari* ***Adventure*** [68] y se produjo cuando el programador principal, *Warren Robinett*, introdujo una sala en la que salía su nombre. La razón principal fue que por aquella época, el nombre de los desarrolladores no aparecía en los juegos, sólo el de las compañías lo hacía. Hoy en día son muchísimos los *easter eggs* que podemos encontrar, ya sea como elementos escondidos o que se desbloquean con logros o trucos.

Así se llega a **1979**, último año de la década y año de juegos míticos como ***Asteroids*** [69] de *Atari*. Fue una idea de *Lyle Rains* y programado por *Ed Logg*. Se considera uno de los éxitos más importantes de la historia de los videojuegos en Estados Unidos, convirtiendo a *Atari* en la mejor compañía hasta ese momento. El objetivo del juego consistía en disparar y destruir los asteroides sin que ningún fragmento se llegase a estrellar contra la nave del jugador. Los jugadores contaban con un botón de disparo y con la posibilidad de girar la nave sobre su propio eje. Cada pantalla enfrentaba al jugador a cuatro asteroides gigantes. Al disparar, se separaban en dos medianos y éstos a su vez en dos pequeños. *Asteroids* fue un éxito, destronando a *Space Invaders* de *Taito*.

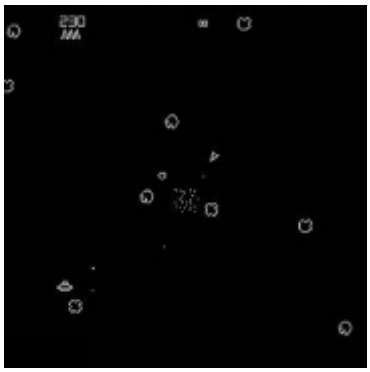

Figura 15: Pantalla del juego Asteroids

Tras el lanzamiento y la consagración de *Asteroids*, *Namco* lanzó ***Pac-Man*** [70] que de nuevo reinventó la forma de jugar. Diseñado por *Toru Iwantani* supuso una revolución total en los videojuegos que probablemente no vuelva a suceder. Tuvo tanto éxito, que se le dedicó una portada en la revista *Time Magazine* [71]. El nombre del juego fue en un principio Puck-Man, derivado de la palabra japonesa "paku-paku", que significa comer, pero al pasar al mercado americano se le cambió el nombre por las similitudes de las palabras puck y fuck.

Uno de sus puntos a favor era que en el momento de su llegada, eran los “marcianitos” en todas sus variantes los reyes de las salas recreativas, por lo que, aunque el juego era bastante sencillo, el cambio de temática y el hecho de ser tremendamente adictivo lo llevaron al éxito. La acción consistía básicamente en ir comiendo puntos a través de un laberinto, en el que había que evitar a los fantasmas que podían matar a *Pac-Man*. Se considera al *Pac-Man* como el videojuego más influyente de la historia junto al pionero *Pong*.

Figura 16: Pantalla del juego Pac-Mac

Para cerrar la década de los setenta, hay que destacar la creación a finales de **1979** de dos compañías que hoy día siguen liderando el mundo de los videojuegos. La primera es ***Capcom*** [72], fundada de *Japan Capsule Computers*, su función inicial era la de fabricación y distribución de máquinas recreativas. *Capcom* rompería un día el mercado con el famoso ***Street Figther II.*** Por otro lado, fruto de los problemas internos de *Atari*, surge la primera third-party[16] de la historia. *Larry Kapla*, *Alan Miller*, *Bob Whitehead*, *David Crane* y *Jim Levy* fundan ***Activision*** [73]. La compañía lanzó más de 50 juegos en la década de los ochenta y revolucionó en concepto de jugabilidad a finales de los noventa con su ***Tony Hawk Pro Skater***.

2.1.4. Los videojuegos en los ochenta

Se pueden considerar los ochenta como la década en la que *Nintendo* empieza a crecer, afianzándose en el mercado y empezando a dominarlo. De hecho, en los primeros años ya da pasos que empiezan a convertir a la empresa en el gigante que es hoy día.

El primer paso de *Nintendo* en **1980** es presenta la serie ***Game & Watch*** [74], un solo juego que se podía jugar en pantalla LCD, además de ser un reloj y alarma. Estas consolas fueron las primeras que usaron el pad[17], que posteriormente apareció en la mayoría de los controles del resto de consolas. Algunos de los juegos más conocidos de este dispositivo fueron *Donkey Kong*, *The Legend of Zelda* o *Mario Bros*. Aunque en pocos meses muchas compañías fabricaron clones sin licencia que llevaron a *Nintendo* a perder muchos ingresos, el éxito fue tal que aparecieron cerca de sesenta modelos.

Ese mismo año, ***Atari*** patentó *Space Invaders*, siendo esta la primera licencia de una recreativa y dándole la exclusiva de explotación evitando así los clones. *Atari* ganó unos 100 millones de dólares. A la vez ***Mattel*** distribuía su primera consola, ***Intellivision***. Fue la primera consola que le pudo causar serios problemas a *Atari*.

1981 trae consigo otro par de éxitos recordados hoy día. El primero fue ***Frogger*** [75] de *Konami* que supuso un gran éxito para esta compañía. Posteriormente apareció ***Donkey Kong*** [76] de *Nintendo*, de la mano de *Shigeru Miyamoto*. En el juego podíamos ver por primera vez a un

fontanero llamado *Jumpman* (más adelante rebautizado como ***Mario***). El objetivo del juego es rescatar princesa secuestrada por un gorila llamado *Donkey Kong*. El juego fue un éxito total.

Figura 17: Pantalla del juego Donkey Kong

En año **1982** se convierte en un año de presentación de consolas y también de la creación de dos nuevas compañías que entrarían fuerte en el mercado y se mantienen hoy día.

Apareció ***Commodore 64*** [77] de la compañía *Commodore*, una máquina potente y barata. Utilizaba unidad de casete y de disquetera. Tenía expansiones para cartucho, aplicaciones de juegos, gráficos, una paleta de 16 colores y permitía utilizar el lenguaje *BASIC*. *Coleco* lanzó ***ColecoVision*** [78] que se podía transformar en un ordenador añadiéndole un teclado y un lector de discos. Era técnicamente muy superior a todas las consolas de la competencia y emulaba bastante bien las máquinas recreativas de la época. Además, fue famosa por ser el primer sistema en tener un adaptador para jugar con los juegos de la competencia y por incluir *Donkey Kong* con la consola; juego por el que *Coleco* pagaba 2$/copia a *Nintendo*.

Emerson lo intentó con ***Arcadia 2001*** [79], desgraciadamente para esta compañía, debía competir con sistemas de bastante más calidad. Solo se le prestó atención cuando salió al mercado, pero en seguida cayó en el olvido de la gente. Durante este año *Atari* presenta el rediseño de la *Atari 2600* cambiando el juego que venía incluido por *Pac-Man*. Más tarde presenta la *Atari 5200* [80], la consola tuvo menos éxito que su antecesora (debido a su incompatibilidad), pero era una máquina muy potente y por primera vez los controles de una consola tenían un botón de pausa para detener el juego. Para potenciar su venta, se acabó lanzando un adaptador para las dos consolas de *Atari*.

Por último, en lo que a lanzamiento de consolas se refiere *Sinclair* lanza el ***ZX Spectrum*** [81] y ***Vectrex*** [82] llega a las tiendas. El *Zx Spectrum* tenía muy pocos recursos, los juegos venían en cintas. Se convirtió en el ordenador más pequeño y barato. Acabó siendo el ordenador más popular de la década de los ochenta en Europa. De *Vectrex* hay que destacar que basaba sus juegos en vectores. Este hecho le daba cierta ventaja sobre sus competidoras al ser capaz de recrear entornos 3D, algo muy novedoso para la época. Debido a la gran competencia se dejó de fabricar en poco tiempo.

En cuanto a compañías, se crean ***Electronic Arts*** [83] y ***Ocean*** [84]. *Electronics Arts* es fundada por *William M. Hawkin III*, bajo el nombre de *Amazin' Software*, con una inversión inicial de 2 millones de dólares. Revolucionó el marcado siendo la pionera a la hora de ofrecer sus productos directamente a los minoristas. Hoy día es una de las compañías más grandes. Por otro lado también se crea *Ocean Software*. Durante muchos años *Ocean* sería la compañía europea más importante.

1983 y **1984** son recordados por ser años en los que el mercado de los videojuegos sufrió una grave crisis, tanto en los ordenadores como en las recreativas y consolas domésticas. Muchas compañías se hundieron, algunas se tuvieron que a retirar a Japón, para volver años después.

Sin embargo de este par de años se pueden destacar 5 hechos principales. El primero el lanzamiento de ***Famicom*** [85] en Japón por parte de *Nintendo* (en estados unidos y algunas zonas europeas se llamará ***N.E.S.*** - *Nintendo Entertainment System*). La idea principal era crear un nuevo sistema capaz de tener muchos juegos diferentes, almacenados en cartuchos. Su lanzamiento no fue todo lo bien que se esperaba ya que la consola muchísimos errores que hacían que se colgara y su catálogo de juegos se limitaba a conversiones como *Donkey Kong*. Tras unos cambios en su placa base se lanzaron nuevas unidades que fueron un tremendo éxito y la convirtieron en la consola más vendida.

Figura 18: Consola Famicom

Figura 19: Consola N.E.S.

El segundo hecho a destacar es el lanzamiento de *SEGA* de su primera videoconsola, la *SG-1000* [86]. Podía utilizar cartuchos o tarjetas. Los cartuchos serían compatibles con las consolas posteriores de *SEGA* como ***Master System*** u ***Othello Multi Vision***.

El tercero está relacionado con la parición de dos personajes que hoy día siguen protagonizando juegos prácticamente cada año. El primero es ***Mario Bros*** [87], que aunque ya había aparecido en *Donkey Kong*, aparece por primera vez en su propio juego. También aparece por primera vez su hermano *Luigi*. El objetivo del juego era limpiar las tuberías de las diferentes plagas que había golpeando el juego de debajo de ellas. El segundo es ***Bomberman***, creado por *Hudson Soft*. El juego consistía en ir colocando estratégicamente bombas en cada nivel con el objetivo de quitar obstáculos o eliminar enemigos. Las bombas tardaban un tiempo en explotar y tenían un cierto alcance, por lo tanto había que vigilar para no ser alcanzado por ellas. El personaje principal, que aún no tenía el diseño actual, se ha convertido con el tiempo en la imagen de *Hudson*.

Figura 20: Pantalla del juego Mario Bros

Figura 21: Pantalla del juego Bomberman

El cuarto hecho importante llega de España y es que aparece la primera compañía dedicada exclusivamente a los videojuegos. Fue ***Indescomp*** [88], que inicialmente distribuía los primeros *ZX Spectrum* y *Amstrad CPC* y que más tarde sería la primera en distribuir videojuegos de creación

española. Por último, este par de años están marcados por la creación de ***Squaresoft*** [89], una de las compañías más importantes en la historia de los videojuegos. La compañía alcanzó la bancarrota en 4 años y estuvo a punto de desaparecer. No lo hizo porque creó el primer ***Final Fantasy*** [90] como un último intento de salir a flote (de ahí lo de "Final" en el nombre del juego). *Final Fantasy* alcanzó un éxito enorme y llevó a la empresa a lo más alto.

1985 se puede considerar como el año en el que se supera la crisis. *Nintendo* lanza el ***Super Mario Bros.*** [91] Creado para la *Famicom* por *Shigeru Miyamoto*. Este juego fue el que llevó a *Mario* al estrellato hasta convertirlo en la mascota principal de la compañía. Miyamoto empezó creando el sprite de Mario y las primeras pantallas, pero más adelante se le ocurrió la idea de reducir el tamaño y pensó en variaciones de la mecánica de juego. En la versión final, si Mario se comía una seta, crecería de tamaño y si se comía una flor, lanzaría bolas de fuego. Son probablemente los primeros *ítems* o *power ups*[18] de la historia de los videojuegos y con toda seguridad de los más reconocibles. Creó un antes y un después en el mundo de los videojuegos, considerándosele como el juego más vendido de todos los tiempos (ayudado en parte por ir incluido con la consola).

Posteriormente ese mismo año se crea ***Tetris*** [92], uno de los juegos más importantes de la historia y versionado hasta la saciedad. Fue inventado por *Alexey Pazhintov* que se inspiró en un juego de pentominós[19] que vio mientras trabajaba en la *Academia de Ciencias de Moscú*. Su primera versión incluía siete piezas formadas a partir de bloques cuadrados. Tetris se hizo mundialmente famoso cuando se hicieron versiones para *IBM PC*, *Apple II* y *Commodore 64*.

Figura 22: Pantalla del juego Super Mario Bros

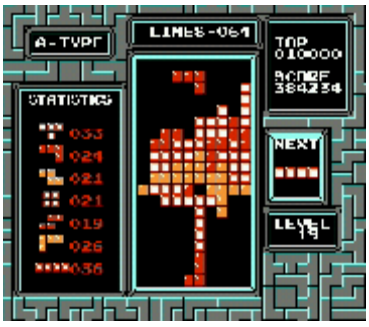

Figura 23: Pantalla del juego Tetris

Por último, ese mismo año, *SEGA* lanza ***Master System*** [93] en Japón. Posteriormente, guiada por el éxito de la *NES* en EEUU, decidió mejorarla y lanzarla en el mercado americano.

El año **1986**, se recordará por que fueron lanzados una gran cantidad de juegos que a día de hoy aún se recuerdan, como ***Out Run*** y ***Wonderboy*** de *SEGA* o ***Arkanoid*** y ***Bubble Bobble*** de *Taito*. Pero son tres los que realmente han quedado en la memoria de todos. El primero ***Legend of Zelda*** [94] de *Famicom*, otro juego creado por *Shigeru Miyamoto*. La historia se ambienta en la tierra de *Hyrule* donde un joven llamado *Link* tiene que rescatar a la princesa *Zelda*. El videojuego tenía varias innovaciones técnicas hoy en día básicas, la más importante, la posibilidad de guardar los progresos aunque apagaras la máquina. Su éxito hizo que se haya convertido en una de las grandes franquicias de *Nintendo*. El segundo es ***Metroid*** [95], también de *Nintendo* en NES y que aportó como novedades la inclusión de una protagonista femenina y el hecho de tener un desarrollo no lineal. *Metroid* acabaría convirtiéndose en otro de los videojuegos insignia de *Nintendo*.

Por último, *Taito* lanzó el mítico ***Arkanoid*** [96] aprovechando que la gente ya se había olvidado de *Breakout* de *Atari*. *Arkanoid* es una versión renovada del juego que incluye 33 fases, un enemigo final y una jugabilidad totalmente revisada con ladrillos de varios tipos, algunos que soltaban cápsulas para dar mejoras a nuestra nave, bolas múltiples, etc.

Figura 24: Pantalla del juego Zelda

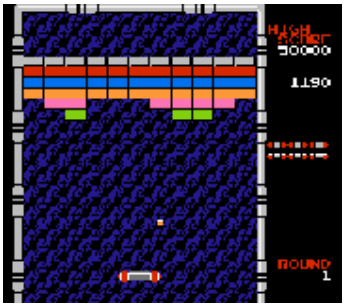

Figura 25: Pantalla del juego Arkanoid

El año siguiente, 1987, de nuevo está marcado por la aparición históricos. En primero de ellos es ***Final Fantasy***. Ya se ha comentado anteriormente lo que supuso para *Squaresoft* la publicación del juego. Es el primer gran RPG[20] de la historia. Tanto la saga como el género alcanzaron su apogeo con la salida de ***Final Fantasy VII***, casi una década más tarde.

Poco después apareció ***Maniac Mansion*** [97] de *Lucasfilm Games*. Revolucionó el mercado por ser la primera gran aventura gráfica[21] y por su motor gráfico: *SCUMM* (*Script Creation Utility for Maniac Mansion*), que supuso una gran innovación en los interfaces de juego en la época. Otra

de las grandes novedades del juego estaba en la posibilidad de escoger a varios personajes para que acompañaran al protagonista lo que daba lugar a distintos finales. Supuso toda una revolución y provocó que salieran versiones para *PC*, *Atari ST*, *Apple I*, *Amiga* y *NES*.

El siguiente clásico en aparecer fue ***MegaMan*** [98] de la mano de *Capcom*. Su aporte al mundo de los videojuegos fue que cada uno de los jefes finales era vulnerable a ciertos tipos de armas especiales y ataques. Ha quedado para la posteridad como uno de los juegos de plataformas más famosos de la época.

Por último aparece ***Metal Gear*** [99] de la mano de *Konami*. Diseñado por *Hideo Kojima*, apareció en principio para *MSX*, pero poco después s publicó una versión para *NES*, con varios cambios (ciertas pantallas, los escenarios, el argumento). Este juego presentó dos nuevas ideas al público, el sigilo o infiltración como modo de juego, y el *codec*; comunicador que utilizaba el protagonista *Solid Snake* para mantener el contacto con sus aliados. La consagración llegó años más tarde en *PlayStation*, dando a conocer el juego a toda una generación y convirtiendo la saga y a *Kojima* en un autentico mito.

Figura 26: Pantalla del juego Final Fantasy

Figura 27: Pantalla del juego Maniac Mansion

Figura 28: Pantalla del juego Megaman

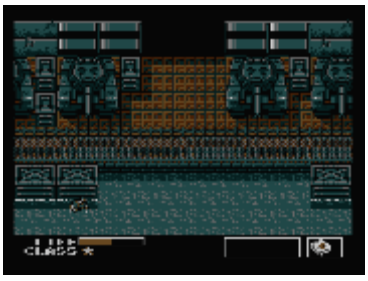

Figura 29: Pantalla del juego Metal Gear Solid

En los años restantes de década un par de consolas y de juegos marcan la historia. **1988** se convierte en el año de *SEGA* y su ***Mega Drive*** [100]. Fue el intento de *SEGA* de romper el mercado americano en el que no tenía éxito. Se decidió a sacar una consola de 16 bits que sería tan potente como los mejores ordenadores del momento. Un hardware avanzado y un completo catálogo de juegos hicieron que comiera el terreno a *Nintendo* convirtiéndose en la consola más vendida hasta la aparición de la ***Super Nintendo*** [101]. Entre su catálogo destacan juegos como *Sonic*, *Golden Axe*, *Outrun* o *Street Fighter 2*.

En España, el año está marcado por la aparición de ***La Abadía del Crimen*** [102], considerado por muchos como el mejor videojuego español durante años. Estaba basado en la novela "*El nombre de la rosa"* y presentaba una alta dificultad. Sus controles eran bastante innovadores, el juego usaba un punto de vista isométrico[22] y las teclas izquierda y derecha servían para girar al personaje, quedando la superior para avanzar y la inferior para retroceder.

1989 comienza la batalla de consolas portátiles[23]. *Nintendo* lanza ***Game Boy*** [103] que se convierte en éxito inmediato. Su objetivo era conseguir una consola pequeña, ligera, barata y con muchos videojuegos. Los alcanzó todos, incluso el hasta entonces impensable de permitir el uso de cartuchos en una portátil. Se convirtió en la portátil más famosa de la historia hasta la aparición de la ***PSP*** [104] de *Sony*, con más de 100 millones de unidades vendidas. Al mismo tiempo *Atari* presentó su ***Lynx***, que en España tuvo una buena acogida pero que falló, tal y como ocurrió en Estados Unidos, porque *Atari* encontró muy pocos desarrolladores que programaran para la máquina. Más tarde, con la aparición de la ***Game Gear*** [105] de *SEGA*, acabaría olvidada.

Se cierra la década con la aparición de otro juego mítico de la mano de *Capcom*, en este caso ***Pang!*** [106]. Cada fase empieza con un número de globos de distintos tamaños que, al tocar al personaje, le quitan una vida. El jugador debe usar un arma para disparar a las bolas y hacer que se dividan en otros trozos más pequeños y así hasta que desaparezcan. Tanto en recreativa como en versiones para consolas domésticas fue un éxito con gran aceptación entre los jugadores.

2.1.5. Los videojuegos en los noventa

Las década de los noventa comienza con la llegada de ***Super Famicom*** o ***SuperNES*** de *Nintendo*, una de las consolas que más ha triunfado en la historia. *Nintendo* desarrolló la consola con mucho tiempo y paciencia, y en el momento de su lanzamiento era superior a sus competidoras. Su procesador de 16 bits, su potencia y su enorme catálogo de juegos hicieron que triunfara. Dentro del catálogo de *Super NES* había juegos como *Super Mario World*, *Super Mario Kart*, *The Legend of Zelda*, *Pilotwings*, *Mario Paint*, *Super Metroid*, *Starfox*, *Donkey Kong Country*, o *Killer Instinct* entre otros. En poco tiempo plantó cara a *Mega Drive* de *SEGA* (que ya estaba bien implantada en Estados Unidos) y comenzaron una rivalidad que duró años.

Viendo el éxito de la *Game Boy* de *Nintendo SEGA* decide probar suerte en el mundo de las portátiles y lanza su propia consola, la ***Game Gear***. Era una versión portátil de la *Master System*, y tuvo buena aceptación gracias a que salió con todo su catálogo. Su mayor problema era la escasa duración de la batería. La pantalla a color y las buenas ventas hicieron que se fabricaran periféricos como un adaptador para ver la TV en la consola o una lupa para mejorar la resolución de la pantalla.

Este mismo año *SNK* pone a la venta ***Neo Geo*** [107]. La estrategia de *SNK* fue muy novedosa, puesto que el sistema de recreativas era prácticamente idéntico al doméstico proporcionando a usuario una experiencia lo más cercana posible al videojuego comercial. La tecnología era muy superior a las de la época y pronto se impuso en salones recreativos, pero el elevado precio lo hacía inalcanzable para el usuario medio. El mercado era muy reducido pero rentable para *SNK*. La *Neo Geo* fue la primera consola con dos procesadores, uno de 16 bits y otro de 8 bits. Fue una consola al alcance de muy pocos.

Durante este año, *Amstrad* lanza el ***Amstrad GX4000***. La consola tenía buenos gráficos pero su catálogo tenía sólo 40 juegos. Además era difícil adaptar los juegos de la época porque empleaba tecnología de 8 bits, frente a los 16 bits de la *Super Nintendo* y la *Mega Drive*. Fue un fracaso.

El año **1990** se cierra con la llegada de dos juegos míticos. El primero ***Super Mario World*** [108] de *Nintendo*. Suponía una evolución en gráficos, sonido y jugabilidad. Además presentó un

concepto que *Microsoft* ha vuelto a poner de moda con su ***XBOX 360***: los *logros*[24]; siendo uno de los primeros juegos que compensaba al jugador al completarlo al 100% descubriendo todos los secretos. Fue un completo éxito en todo el mundo y es considerado uno de los mejores juegos de la historia. El segundo fue ***The Secret of Monkey Island*** [109]**.** Fue el quinto juego en utilizar el motor *SCUMM*. Este juego supone una revolución en las aventuras gráficas, ya que introducía novedades como las famosas batallas de insultos o la imposibilidad de que mataran al jugador.

Figura 30: Pantalla del juego Super Mario World

Figura 31: Pantalla del juego The Secret of Monkey Island

La tendencia a partir de estos años se acercó más al software y menos al hardware. Los usuarios habían apostado por un sistema y demandaban juegos. Cada vez era más difícil lanzar un sistema que fuera capaz de competir con la *Super NES* y la *Mega Drive*. Por eso los siguientes años destacan por la creación de sagas que continúan hoy día. El mejor ejemplo es ***Sonic*** [110] de *SEGA*. Tan importante fue que sustituyó a *Alex Kidd* como mascota de la compañía. La diferencia con *Super Mario Bros* fue la velocidad y el dinamismo y fue un gran éxito mundial. *Nintendo* contraatacó ese mismo año publicando ***The Legend of Zelda: A Link to the past*** [111]. Se convirtió en uno de los juegos más vendidos de *Super Nintendo* y es considerado por muchos como uno de los mejores juegos de la saga.

Figura 32: Pantalla del juego Sonic

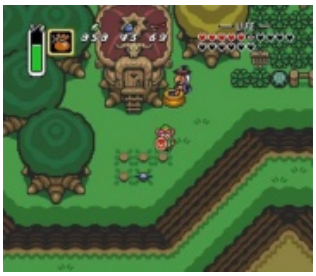

Figura 33: Pantalla del juego The Legend of Zelda

El siguiente juego publicado marcó un antes y un después en los juegos de lucha. Es ***Street Fighter II*** [112], desarrollado por *Capcom* y que revoluciona la saga y el género. Proponía un plantel de 8 personajes a elegir, 4 jefes finales y una historia y final diferente para cada personaje. Cada luchador tenía distintas características de fuerza, velocidad, técnicas y movimientos especiales. El juego se extendió por todo el mundo en su versión recreativa y doméstica. El éxito del juego de *Capcom* hizo que aparecieran una gran cantidad de juegos de lucha, entre ellos destacó *Midway* con su ***Mortal Kombat*** [113]. El juego constaba de siete luchadores, había una gran variedad de golpes y el juego era tremendamente violento, pudiendo realizar técnicas que alcanzaban un nivel de violencia y gore nunca visto. Otra novedad fue que, para el diseño de personajes, se utilizaron actores reales. Fue un juego muy criticado por su violencia, pero esto en lugar de afectar negativamente, hizo que aumentaran sus ventas y popularidad. Hoy en día se siguen haciendo continuaciones de la saga.

Figura 34: Pantalla del juego StreetFighter II

Figura 35: Pantalla del juego Mortal Kombat

Cuatro juegos más marcaron el mercado antes de que en **1994** se produjera un cambio de ciclo. El primero fue ***Wolfenstein 3D*** [114], primer juego en utilizar con éxito la perspectiva tridimensional. El jugador es un espía que es capturado por los nazis y encerrado en un castillo. Cuya misión será escapar. Gráficamente era muy pobre: no había techo, ni suelo y los objetos eran sprites perpendiculares al jugador pero el concepto y el uso de las 3d fue totalmente revolucionario y lo llevaron al éxito. Fue el predecesor de los juegos de disparo en primera persona[25].

Si *Wolfenstein 3D* fue el predecesor de los juegos en primera persona ***Alone in the Dark*** [115] lo fue del género *Survival Horror*[26]. El jugador se encontraba en una mansión llena de zombies y criaturas y podía afrontar dos modos de juego: sobrevivir y salir con vida, o entrar en la mansión e investigar sobre la muerte del dueño. Este sistema ha sido utilizado posteriormente en

juegos tan aclamados como *Resident Evil 2* o la saga *Silent Hill*. Además, *Alone in the Dark* fue de los primeros juegos en utilizar un entorno 3D real.

Figura 36: Pantalla del juego Wolfenstein 3D

Figura 37: Pantalla del juego Alone in the Dark

El tercer juego aparece en **1993** y llegó aprovechando el tirón de los FPS (First Person Shooter o juegos de disparo en primera persona). El juego es ***Doom*** [116] y todas las mejoras que presentaba respecto a *Wolfenstein 3D* han servido de base a los FPS que salieron durante años. La mecánica consistía en salir de un laberinto enorme que representaba cada nivel aniquilando alienígenas. Técnicamente era muy avanzado para la época y tenía muy buenos modelados además de destacar sus enormes mapas. El éxito fue tan grande que, además de salir versiones para todas las consolas de la época, se ha ido versionando para todas y cada una de las consolas posteriores hasta la misma *XBOX 360* (en versión arcade).

Por último, pero no menos importante, el año **1993** acogió el primer juego de una saga que perdura hasta hoy y que abrió las puertas a otras muchas franquicias deportivas. El juego no es otro que el primero de los ***FIFA*** [117] de *Electronic Arts,* conocido más tarde como ***FIFA 94.*** La novedad se centraba en la vista isométrica ya que antes la mayoría de los juegos de deporte presentaban perspectiva aérea o lateral y no manejaban bien la simulación de las 3D. *FIFA* rompió esta tendencia y aportó varias novedades más al género deportivo: gran cantidad de animaciones, un novedoso sistema de repeticiones, una gran inteligencia artificial muy precisa y una infinidad de equipos y competiciones. Se lanzó para todos los soportes y las ventas fueron altísimas. En septiembre de **2009**, 16 después del original, se ha publicado ***FIFA 2010***.

Figura 38: Pantalla del juego Doom

Figura 39: Pantalla del juego FIFA94

Tras esta tendencia hacia el desarrollo de juegos, varias empresas intentan lanzar nuevas consolas con poco o ningún éxito. Algunos ejemplos son ***Jaguar*** de *Atari*, la ***Amiga CD32*** (siendo la última máquina de tecnología *Amiga*), la ***3DO*** de *Panasonic*, ***Pionner LaserActive*** de *Pionner* o la ***32X*** de *SEGA*. Ninguna aportó nada nuevo por lo que no se profundizará en su historia ni características.

Aunque no había tenido éxito en los últimos intentos *SEGA* volvió a probar suerte y en **1994** lanzó la ***Saturn*** [118], una consola con dos procesadores de 32 bits, 2Mb de RAM y juegos en formato CD. La estrategia de *SEGA* fue malísima, puesto que al enterarse de la salida inminente de la consola de *Sony*, la ***PlayStation*** [119]; intentó rediseñar su máquina para hacerla más competitiva. El resultado fue que el catálogo de juegos inicial se redujo por incompatibilidades y el resto de juegos aún en desarrollo fueron acabados con prisa para compensar. Esta política hizo que, aunque en potencia *Saturn* era superior a la máquina de *Sony* (*Saturn* era la plataforma ideal para juegos 3D de la época, muchos de ellos imposibles de ver en *PlayStation*) acabo fracasando. De hecho, con la llegada de la ***Nintendo 64*** [120] quedó relegada al tercer lugar.

También en este año se produce la llegada de la consola que para muchos reinventó el mercado de los videojuegos, la ***PlayStation*** de *Sony*. En defensa de *Saturn* o *Nintendo 64* hay que decir que compitieron con una de las consolas más vendidas de la historia de los videojuegos y que en muchos sentidos lanzó el negocio más que nuca. Antes de hablar de la consola, hay que destacar algo que pocos saben. Años antes de la salida de *PlayStation*, *Sony* estaba negociando con *Nintendo* para desarrollar un accesorio llamado *SNES PlayStation* que diera soporte CD a la *Super Nintendo*. Estas negociaciones fueron canceladas con el tiempo y *Sony* decidió entrar en el negocio de forma independiente lanzando el proyecto *PlayStation*, una consola con una CPU de 32 bits a 33,8 Mhz y juegos en formato CD.

Por esta consola han pasado prácticamente todas las sagas ya sean actuales o retro y muchas han dado sus primeros pasos en ella. El catálogo de juegos era enorme y aunque incluía muchas mediocridades, se pudieron jugar joyas como *Metal Gear Solid*, *Winning Eleven* (en la zona europea conocido como *PES*), *Resident Evil* (los tres primeros de la saga) *Tomb Raider* (1 y 2), *Tekken* (1 y 2), *Gran Turismo* (hasta el 4), *ISS Pro Evolution*, *Oddworld: Abe's Oddysee*, *Medievil*, *Final Fantasy* (VII, VIII y recopilaciones de los cinco primeros).

Fue una consola que tuvo éxito en todos los sentidos, acabó con toda su competencia y se convirtió en una de las más vendidas de la historia. Si algo hay que achacarle, es su facilidad para ser pirateada, debido a la poca o nula protección de su software. Esto para muchos fue el impulso definitivo para su rotundo éxito y el motivo principal por el que aplastó tan claramente a la competencia.

En **1995** se produce un hecho claro que demuestra que los videojuegos ya se habían convertido en un negocio mundial a gran escala. Este año se celebra en los Ángeles el primer ***E3*** [121]**.** La *Asociación del Software Digital Interactivo* inaugura la primera ***Exposición de Entretenimiento Electrónico*** (*Electronic Entertainment Expo*), más conocida como *E3*, que desde entonces se celebra cada tercera semana de Mayo. Esta exposición reúne a muchas de las principales compañías de los videojuegos de todo el mundo para que muestren sus futuros lanzamientos, ya sean consolas, periféricos o videojuegos.

1996 es el año de ***Nintendo 64*** y ***Super Mario 64*** [122]. Nintendo llevaba años haciendo acuerdos con empresas como *Silicon Graphics* o *Rambus Inc*. para fabricar hardware y con *Rare* o *Williams* para el software y con la ayuda de todas lanzó un proyecto con el nombre de ***Ultra 64*** (cambiado finalmente por el de ***Nintendo 64***). Tenía un CPU de 64 bits a 93,75 Mhz y juegos en formato cartucho (un error grave como se demostró más tarde).

La consola trajo consigo muchas innovaciones, como por ejemplo el mando. Fue el primero en incluir botón de gatillo, analógicos[27], botones de acción dispuestos en forma de cruz o de proporcionar vibración gracias al accesorio *Rumble Pack*. Muchas de estas innovaciones fueron aplicadas casi inmediatamente a la consola de *Sony* que poco después lanzó los famosos *Dual Shock*, mandos con dos controles analógicos y vibración sin accesorios adicionales. A su vez los dos

controles analógicos de los mandos de *Sony* supusieron una revolución y posteriormente se incluyeron en los mandos de *XBOX* y *XBOX 360*.

Por último destacar que el catálogo de juegos de *Nintendo 64* era muy completo, entre los juegos más famosos se puede destacar *Super Mario 64*, *The Legend of Zelda: Ocarina of Time*, *GoldenEye 007*, *Mario Kart 64*, *The Legend of Zelda: Majora's Mask*, *Super Smash Bros* o *F-Zero X*. De entre todos los juegos que se cancelaron destaca *Final Fantasy VII*, que posteriormente salió de forma exclusiva en *PlayStation* y que muchos creen cambió el rumbo de ambas consolas.

De entre los juegos de *Nintendo 64*, además de los dos *Zelda* que destacan por su cuidadísima producción, hay que resaltar *Super Mario 64*, ya que supone un avance en lo que a las 3D se refiere. Fue el primer plataformas 100% 3D y se adaptaba al mando de *Nintendo 64* como un guante. De hecho se comenta que primero se creo el juego y luego se desarrolló un mando que cubriera todos los movimientos y acciones que Mario podía hacer. Vendió casi 13 millones de copias y se considera como uno de los mejores juegos de la historia.

Antes del final de la década, en **1999**, *SEGA* lanzó lo que a la postre sería su última consola, la ***Dreamcast*** [123]. Fue una consola adelantada a su tiempo, contaba con una CPU de 128 bits a 200 Mhz y utilizaba como soporte el GD-ROM. Fue la primera consola en contar con juego on-line al llevar un modem incluido, explotaba la competición mediante logros y planeaba un sistema de avatares[28] similar al de las consolas de hoy en día. Era la más potente de la generación de 128 bits y como su predecesora, la *Saturn,* estaba hecha para mover polígonos en 3D a velocidades mucho mayores que sus competidoras. Pero sin duda estaba marcada por el fracaso de *Saturn* y la *PlayStation 2*[124] venía avalada por el éxito de su antecesora. Hubo muchos factores y hay muchas hipótesis explicando que pasó para que la *Dreamcast* fracasara de la manera que lo hizo. Lo cierto es que hoy día, se considera a *Dreamcast* como una de las mejores consolas que han existido siendo base de las posteriores *XBOX* y *XBOX 360*. De su catálogo de juegos se puede destacar *Crazy Taxi*, *The House of the Dead 2*, *Marvel vs Capcom 2*, *Sega Rally 2*, *Shenmue*, *Shenmue 2*, *Sonic Adventure* o *Soul Calibur*. Hay que hacer mención especial a ***NBA 2K*** [125], franquicia de *SEGA* que comienza en *Dreamcast* y que le ha arrebatado a *NBA Live* [126] de *EA* el primer puesto en juegos de baloncesto.

Para cerrar la década y comenzar con lo que se puede considerar la época actual de los videojuegos, hay que hacer referencia a varias sagas que dieron sus primeros pasos entre **1998** y **1999.** La primera de ellas es ***Residente Evil*** [127], saga que reinventó el género survival horror. Llega de las manos de *Capcom* a *PlayStation*. Buen nivel gráfico, una historia envolvente y tensión continua son las características de este juego basado en un mundo futuro lleno de zombies. Es una saga de las más conocidas y que más éxito ha tenido en el mundo (más de 30 millones de unidades vendidas), llegando a sacar numerosas secuelas (más de diez títulos para diversas plataformas, incluidas las de última generación).

Poco después apareció ***Final Fantasy VII*** [128] de la mano de *Squaresoft*, juego que para muchos decidió la guerra entre *PlayStation* y *Nintendo 64.* El problema fue el formato de los juegos de la consola de *Nintendo*, ya que el juego de *Squaresoft* no cabía en uno de sus cartuchos (de hecho necesito 3 CD's en su versión PlayStation). De ahí que muchos consideren que el gran error de la máquina de *Nintendo* fue el formato de sus juegos. *Final Fantasy VII* supuso un gran éxito, dio a conocer a muchos usuarios los RPG's y vendió cerca de 11 millones de unidades. Fue el primer título de la saga con los personajes modelados en 3D y secuencias de video. Muchos lo consideran como una película hecha videojuego y de hecho se han realizado un par de películas sobre el mundo de *Final Fantasy VII*. Han pasado muchos años desde el lanzamiento de este juego y hoy en día es considerado como uno de los mejores de la historia.

Tres juegos más dieron sus primeros pasos en PlayStation a finales de década y hoy continúan sus sagas con éxito. El primero fue ***Metal Gear*** desarrollado por *Konami*. Su creador, *Hideo Kojima* fue fiel a la versión original publicada en *NES* (comentada anteriormente) pero aprovechó todas las posibilidades técnicas de la consola de *Sony*. El resultado es un juego de infiltración, espionaje y acción difícil de superar. El juego presentaba un diseño cuidadísimo así como una historia que enganchaba desde el primer momento. Las posibilidades eran enormes a pesar de tener escenarios cerrados y los logros invitaban a jugar varias veces el juego para desbloquear todos sus finales y secretos. El éxito fue tan rotundo que *Metal Gear* se ha convertido en una de las sagas más rentables. Este mismo año se ha publicado ***Metal Gear Solid 4: Guns of the Patriots*** en *PlayStation 3* y para 2010 se espera la primera versión de la saga para *XBOX 360*.

El siguiente fue ***Gran Turismo*** [129], juego que aprovechó como pocos el ir incluido en un paquete promocional con la consola. Llegó de manos de *Polyphony Digital* y fue un juego basado en la simulación de la conducción. Cambió el concepto de los juegos de conducción, el jugador debía ir superando licencias que le daban la posibilidad de participar en nuevas competiciones y de acceder a vehículos y accesorios mucho mejores. Para muchos es la saga de conducción/simulación por excelencia. Su última secuela, la quinta, se espera para *PlayStation 3* a finales del 2009.

Por último se presenta de manos de *DMA Games* (posteriormente la actual *RockStar Games* [130]) el primer ***GTA*** [131] (*Grand Theft Auto*). El juego no contaba con grandes gráficos, es más devolvía al jugador a la vista aérea típica de las 16 bit pero tuvo un excelente éxito. Este vino marcado sobre todo por un factor, el juego era un *sandbox*[29] (se pueden ignorar los objetivos principales del juego) en su máxima expresión. Proporcionaba libertad total de movimientos y acciones a lo largo de grandes ciudades, conduciendo distintos vehículos. *GTA* permitía desplazarte libremente por ciudades y jugar sin avanzar absolutamente nada en la historia. Además incluía una serie de minijuegos (carreras de coches, localización de elementos escondidos, retos de habilidad) que se han ido manteniendo en el resto de secuelas y que alargan la vida del juego. El juego se convertía en una combinación de acción, aventura, conducción, y rol ocasional, con elementos de sigilo y carreras al que costaba no engancharse a pesar de su naturaleza adulta y su carácter violento.

El tremendo éxito hizo que antes de finales de **1999** salieran tres versiones más del juego ***Grand Theft Auto: London 1969,*** una expansión que tiene lugar en Londres en el año 1969, ***Grand Theft Auto: London 1961***, una expansión descargable gratuitamente para PC y ***Grand Theft Auto 2*** que incluye varios elementos nuevos pero mantiene la esencia del original.

Estas primeras versiones de *GTA* sentaron las bases de un juego que, a partir de su tercera secuela ***Grand Theft Auto III*** ha roto moldes. *GTA* se ha convertido en la saga con más éxito de la actualidad tras vender más de 100 millones de unidades.

2.1.6. Los videojuegos en la actualidad

Esta última década supone en el mundo de los videojuegos una vuelta a la inversión en el hardware y un giro hacia el jugador más que hacia el propio videojuego. Con tres décadas a sus espaldas, las empresas cada vez encuentran más complicado innovar en los juegos y deciden centrarse más en las plataformas y en el propio jugador, proporcionando así nuevas formas de jugar no explotadas. Por tanto este último punto se centrará más en las consolas y menos en los juegos, ya que la repercusión de estos últimos ha sido menor.

Se puede decir que la década comienza con el lanzamiento de tres grandes consolas anteriores a la última generación [132]: ***PlayStation 2*** de *Sony*, ***Game Cube*** de *Nintendo* y ***XBOX*** de *Microsoft*. *PlayStation 2* (*PS2*) apareció a finales del **2000** con las mejoras de una CPU a 299 Mhz y juegos en formato DVD. Incorporaba puertos USB (siendo la primera) y era retrocompatible con la anterior consola de *Sony*. Las ventas al principio no fueron nada buenas, sobre todo por que los primeros juegos del catálogo tampoco lo eran, pero *Sony* pronto utilizó sus licencias para cambiar el rumbo y con las secuelas de juegos como *Gran Turismo*, *Grand Theft Auto*, *Metal Gear Solid*, *Pro Evolution Soccer* cambió la tendencia. Terminaría marcando un antes y un después en el mundo de los videojuegos, barriendo a todos sus rivales (*Dreamcast*, *Game Cube* y *XBOX*) y se convirtió en una de las consolas más vendidas de la historia. A día de hoy, es la única de su generación que sigue teniendo un gran soporte por parte de la compañía.

2001 es el año de *Nintendo* y *Microsoft*. Nintendo lanzó *Game Cube* [133], una consola que no usaba DVD sino mini-DVD con una CPU a 485 Mhz. A la hora de salir la ventaja que llevaba *PS2* ya era demasiado grande y quedó relegada a un segundo lugar de ventas junto con la *XBOX* [134]. Aún así aparecieron grandes juegos en esta consola, como las correspondientes secuelas de las sagas *Resident Evil*, *Zelda* y *Metal Gear Solid*. A finales de año *Microsoft* se une y anuncia el lanzamiento de *XBOX*, fue la primera consola de la historia que venía equipada con un disco duro y poseía una CPU a 733 Mhz. Tendría bastante éxito y en bastantes mercados se situaría en el segundo lugar de las más vendidas, pero nunca pudo superar a la PS2.

Aunque se ha comentado al principio de este punto que los juegos no han sido tan relevantes, en esta década en **2001** comienzan dos sagas que han marcado el mercado en años posteriores además de continuar una que lo ha dominado. La primera es ***Halo*** [135], saga exclusiva de *XBOX* que alcanzó 5 millones de unidades vendidas, convirtiéndose en una de las sagas más conocidas del mundo de los videojuegos. Ese mismo año *Konami* saca a la venta el primer ***Pro Evolution Soccer*** [136]. Destaca sobre todo porque un juego sin licencias consiguió superar al gigante de la época, *FIFA* de *EA*, comenzando una batalla que se sigue dado hoy día año a año por ver que simulador de futbol domina el mercado.

Por último, en **2001** aparece ***Grand Theft Auto III*** de la mano de *Rockstar Games.* Manteniendo la misma mecánica que sus predecesores acabó revolucionando el estilo del juego con una vista en 3ª persona desde atrás y una ciudad completa hecha en 3D, la ya famosa *Liberty City* (basada en la ciudad de Nueva York del año 2001). Marcó un antes y un después en este tipo de juegos y su modelo ha sido copiado por muchos otros videojuegos del género.

2003 es un año marcado por el sistema ***PEGI***[30] (*Pan European Game Information*), sistema actual de clasificación de videojuegos según la edad y su contenido, y que se usa para la clasificación de los videojuegos y otro software de entretenimiento, dentro de Europa.

Salto a **2004**, año de aparición de la portátil de última generación de *Nintendo*, la ***Nintendo DS*** *(Dual Screen)* [137]. La compañía quería innovar en las consolas y ofrecer una nueva manera de jugar, así que diseñaron esta consola con dos pantallas (una de ellas táctil), un micrófono con reconocimiento de voz, una conexión wifi para poder jugar partidas online, etc. Se ha convertido en la consola más vendida de todos los mercados hasta hoy en día; aunque muchos achacan este éxito a la facilidad con que se puede piratear. Ya sea por un motivo u otro ha superado los 107 millones de unidades vendidas.

Esta buena acogida de la portátil de *Nintendo* hizo que *Sony*, ya acomodada en el mercado de los videojuegos, se decidiera a lanzar su primera portátil: la ***PSP*** [138]. Técnicamente es una consola impresionante, CPU de 128 bits a 333 Mhz y juegos en formato *UMD*[31] (*Universal Media Disk*) que además permite ver películas en la misma consola con gran calidad. . La consola tenía una pantalla enorme para lo que acostumbraban las portátiles que proporcionaba gran calidad de

visionado. Otra innovación fue el stick analógico, siendo la primera portátil de la historia en incluirlo. Hoy en día hay una gran comunidad de usuarios en torno a esta consola, que desarrollan emuladores, *imports*[32] de *Playstation 1* y otras muchas aplicaciones. Pese a todo, ha sido relegada al segundo lugar del mercado de las portátiles a mucha distancia de la consola de *Nintendo*.

Llega **2005**, el año que marca la generación actual de consolas [139]. Primero aparece la ***XBOX 360***, en dos versiones *Core* (sin disco duro) y *Premium* (con disco de 20 GB). Posee un procesador *Xeon* con 3 núcleos de procesamiento paralelos los cuales corren a 3,2 GHz cada uno. Es una de las consolas más vendidas actualmente, en pugna con la ***Wii*** [140] de *Nintendo.* Más que de una consola, se puede hablar de una plataforma de ocio, *Microsoft* ha apostado por un sistema de juego on-line de pago que supere en calidad al de sus competidores y por una red de usuarios que proporcione de forma sencilla acceso a todo tipo de contenido, ya sean demos, películas, discos. Además se amplían las formas de jugar gracias a los arcade, los indie, los contenidos extra de cada juego o los juegos bajo demanda. Por último, *Microsoft* ha explotado la competitividad de los jugadores con su sistema de logros, que más tarde ha sido imitado en *Wii* y *PlayStation 3*. Es curioso como un sistema que nació para alargar la vida de los juegos (se obtienen puntos al completar el juego varias veces o al desbloquear secretos) se ha convertido en uno de los principales métodos de competición de la comunidad de jugadores de *XBOX 360*. Estas nuevas formas de jugar, que también han desarrollado *Sony* y *Nintendo*, han abierto nuevos mercados potenciando el negocio de los videojuegos y convirtiéndolo en uno de los más potentes de la industria del ocio.

Para completar el trío de consolas de última generación hay que esperar a **2006**, año en el que aparecen *Wii* y ***PlayStation 3*** *(PS3)* [141]. La *PS3*, consola de *Sony*, aparece también en dos versiones, una con un disco duro de 20 GB y otra con uno de 60, además de ciertas diferencias poco importante entre ellas. Más adelante se lanzó una versión con 80 GB de disco duro. La consola utiliza un microprocesador con 7 núcleos a 3.2GHz cada uno. Al igual que la máquina de *Microsoft*, una característica la *PS3* de sus predecesoras es su servicio de juegos en línea, siendo en el caso de *Sony* un servicio totalmente gratuito. Otras características importantes de la consola son sus capacidades sólidas de multimedia, los puertos HDMI y Ethernet, la conectividad con la *PSP* y el formato de disco óptico de alta definición, *Blu-ray Disc*, como su principal medio de almacenamiento. La *PS3* también da soporte al *Blu-ray perfil 2.0*, gracias a ello se puede interactuar de manera online con contenidos extras de películas y juegos. Actualmente no goza de buenas

ventas, pese a ser una gran máquina, cosa que se achaca sobre todo a su alto precio y a un catálogo todavía algo escaso en grandes títulos.

Para cerrar el ciclo de las consolas llega la más importante de la última generación, la ***Wii***. Si hay una palabra que define a esta consola es ***revolución***, debido a lo que ha supuesto en la forma de jugar. *Nintendo*, con esta consola, apostó por una nueva manera de jugar con su mando revolucionario, en vez de por la potencia de la consola. La principal característica es el control inalámbrico que lleva incorporado, denominado *Wiimote*, capaz de detectar el movimiento y rotación en un espacio de tres dimensiones; además de incluir vibración y un altavoz. Al igual que sus compañeras de generación, dispone de una plataforma para recibir mensajes y actualizaciones a través de Internet. Puede sincronizarse con la *Nintendo DS*, e incluso hay aplicaciones que aprovechan la pantalla táctil de la portátil.

Desde su lanzamiento, la consola ha recibido premios por la innovación de su mando y la popularidad que ha generado rápidamente. También por el gran número de ventas que ha tenido, siendo actualmente la más vendida. Ha descubierto una nueva forma de jugar y ha abierto el mundo de los videojuegos nuevos colectivos como mujeres, personas mayores, niños de corta edad etc., grupos antes minoritarios que encuentra en esta consola juegos a su medida. Por otro lado, algunos desarrolladores no han apostado por ella debido a su escasa capacidad técnica, o si lo han hecho; ha sido lanzando juegos de baja calidad, cosa que ha perjudicado un poco al sistema de cara a usuarios más especializados.

Por último hay que hacer referencia a tres juegos publicados en esta última etapa y que representan un cambio de ciclo en lo que al mundo del ocio se refiere. Son ***GTA San Andreas*** (de la saga *Grand Theft Auto*), ***Halo 3*** (de la saga *Halo*) y ***GTA IV*** (nuevamente de la saga *Grand Theft Auto*). Los tres han batido el record anterior de juego más vendido en el día de su publicación, *Halo 3* supero a *GTA San Andreas* y *GTA IV* a su vez superó a *Halo 3*. Esto sería sólo un hecho anecdótico si no fuera porque las ventas de cada uno superaron a las de la película que más recaudó, y al disco y libro más vendidos (en su día de publicación) en su año correspondiente, los tres juntos. Este hecho designa los videojuegos como reyes del ocio y ha hecho que incluso las productoras de cine empiecen a programar sus estrenos en fechas libres de lanzamientos de videojuegos importantes para evitar combatir con ellos.

Por tanto, se puede concluir, que los videojuegos son ya el referente en el mundo del ocio, gracias a que los sistemas existentes incluyen a cualquier tipo de jugador posible proporcionando formas de jugar inimaginables hace unos años. De aquel mercado inicial de máquinas recreativas ha surgido un gigante que hoy en día domina el ocio por encima de cine y audio y que ha evolucionado tan rápido, que es difícil preveer como será dentro de diez años.

2.2. Historia de Bomberman

Como ya se ha comentado en el punto **2.1.4 Los videojuegos en los ochenta**, la historia de *Bomberman* [142] comienza en **1983** cuando *Hudson* lanza la primera versión del juego. En Europa y Estados Unidos, el nombre del primer juego de la saga fue ***Eric and the Floaters*** [143]. Sin embargo en Japón ya salió con el nombre actual. Las primeras plataformas fueron *ZX Spectrum* y *MSX*.

Hoy día siguen saliendo clones, secuelas y versiones de Bomberman para casi todas las plataformas. Prácticamente todas las consolas han tenido una versión de *Bomberman*, sea cual sea la generación a la que pertenecen, y con los años la mecánica del juego ha variado poco. Es cierto que existen versiones del juego que pertenecen a otros géneros como la aventura (***Bomberman Hero*** de *Nintendo 64*), los puzzles (***Panic Bomber*** de *PSP*) o los juegos de carreras (***Bomberman Kart*** de *PS2*); pero no son las más recordadas.

La lista total de versiones incluye más de cien títulos. Puesto que la evolución del juego ha sido escasa en estos años, en este punto se comentarán cuatro de las versiones, las que se consideran las más importantes. De esta manera se pretende aportar una visión general del juego pero también una primera aproximación a los conceptos e ideas que se han desarrollado en la versión del juego que recoge este proyecto.

Antes de entrar en detalle sobre cada una de las versiones hay que aclarar cuál es la mecánica del juego. Se puede decir que *Bomberman* es un juego de estrategia basado en laberintos. Cada nivel del juego lo representa un laberinto (normalmente entre 11x13 y 11x17 casillas aunque puede variar

según la versión) y, aunque según se han ido publicando versiones del juego se han ido añadiendo modos de juego, el original era el siguiente: el jugador debe abrirse paso a través del laberinto, el cual está formado por dos tipos de muro: rompible e irrompible. El jugador dispone de bombas (inicialmente una) que tienen un determinado alcance (inicialmente el espacio de una casilla contigua en cada dirección) y un tiempo de detonación (una vez puestas explotan de forma automática pasados unos segundos) y que podrá usar para hacer explotar los bloques rompibles del laberinto y abrirse camino. Dentro del laberinto se encontrará enemigos, que si alcanzan al jugador le matarán y con los que él podrá acabar usando las bombas. Una vez eliminados todos los enemigos de un nivel se supera dicho nivel.

2.2.1. Mecánica de juego

La primera versión del juego a la que se hará referencia es, como es lógico, el primer *Bomberman* de **1983**, también conocido como *Eric and the Floaters*. Esta versión disponía de un único modo de juego con la mecánica descrita anteriormente. El jugador afrontaba cada nivel con la capacidad de soltar una bomba de forma simultánea (el número de bombas totales es infinito) y con la misión de acabar con todos los enemigos del nivel. Una vez logrado pasaba al siguiente.

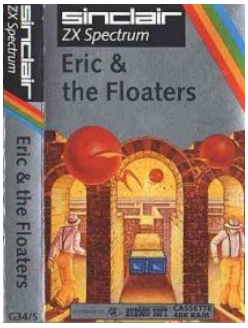

Figura 40: Portada de Eric & the Floaters

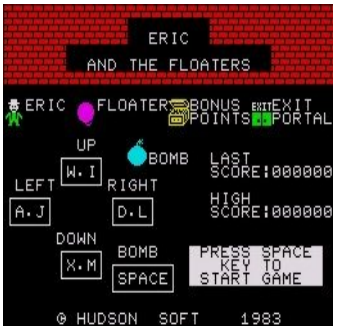

Figura 41: Pantalla del juego Eric & the Floaters

En **1985** salio la versión para la *NES, en Japón* con el nombre de *Bomberman* y en Europa con el nombre de *Dynablaster*. Esta versión se considera el origen del juego tal como lo conocemos hoy día, cambió la imagen del protagonista a una muy similar a la que actual y añadió distintos enemigos. Posteriormente en **1990,** *Hudson* mejoro el juego sobre todo en lo que se refiere a la

imagen e incluyó por primera vez el modo de juego *Battle*. La imagen actual de los *Bomberman* proviene de esta versión. Se hizo tan famosa que el protagonista pasó a ser la mascota e imagen de la compañía *Hudson* manteniéndose hoy día.

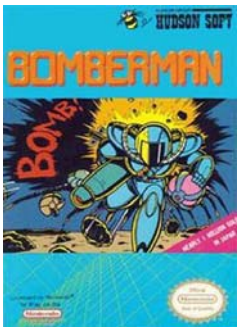

Figura 42: Portada de Bomberman de NES de 1985

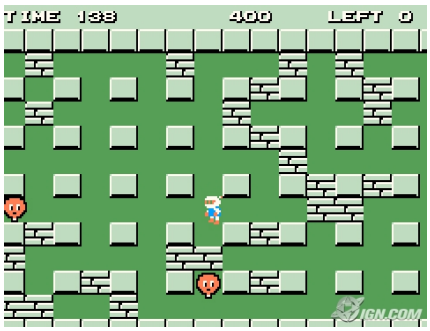

Figura 43: Pantalla del juego Bomberman de NES de 1985

Figura 44: Portada de Bomberman de1990

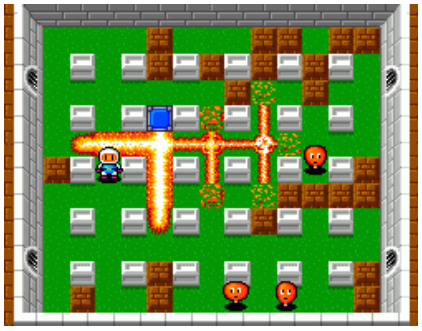

Figura 45: Pantalla del juego Bomberman de 1990

La secuelas de **1985** y **1990** son las que más cambios aportaron al juego y la mayoría de las versiones que salen hoy día son clones de estas versiones añadiendo modos de juego adicionales. La de **1985** daba, además de una imagen renovada al juego y al protagonista, una historia algo más profunda, añadían nuevos enemigos además de los globos del juego original y por primera vez incluía *power ups* que mejoraban temporalmente las capacidades del jugador. La de **1990** añadía por primera vez el modo de juego *Battle*, modo en el que el jugador debía enfrentarse a otros Bomberman con las mismas capacidades que él. Puesto que estas versiones son las más importantes se detallará a continuación que enemigos y power ups se incluían y en qué consistían el modo normal de juego y el modo *Battle*.

El modo de juego *Normal* es muy similar al que ya se ha comentado. Cada nivel estaba representado en una pantalla sin *scroll*[33], es decir, se veía el nivel completo en todo momento. Cada

nivel estaba representado por un laberinto de 11x13 casillas y el jugador sólo podía desplazarse horizontal o verticalmente. Existen dos tipos de bloques que componen el nivel, rompibles e irrompibles. Los irrompibles siempre tienen la misma disposición en todos los niveles ocupando posiciones pares de las filas pares. Los bloques irrompibles se disponen en el mapa de forma aleatoria evitando que el jugador aplique una estrategia determinada en todos los niveles y haciendo cada partida distinta.

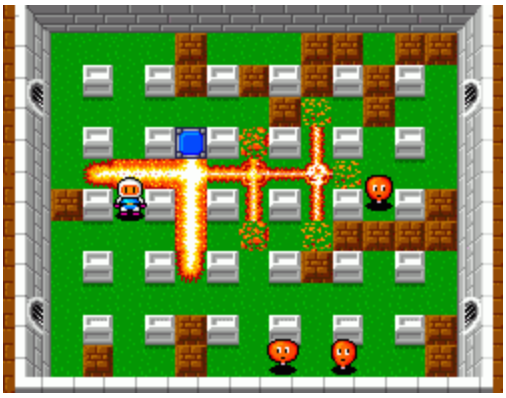

Figura 46: Pantalla del juego Bomberman

En se observa la imagen anterior pueden distinguirse los bloques rompibles (en marrón y formados por ladrillos) y los irrompibles (en gris, de piedra). La disposición de los bloques irrompibles es, como se ha dicho, fija; ocupando posiciones pares (2, 4, 6 etc. si se empieza a contar en 1) de las filas pares (2, 4, 6 etc. si empezamos a contar en 1).

Continuando con la mecánica, el jugador podía depositar bombas, que al soltar, se colocaban en la casilla en la que estaba el jugador. Las bombas tenían un alcance determinado, inicialmente una casilla y que podía incrementarse con el power up adecuado, y un tiempo de detonación determinado. El número de bombas simultáneas que el jugador puede poner es inicialmente uno, pero también puede incrementarse mediante power ups. Si el efecto de la explosión de un jugador alcanza una bomba sin detonar esta explosiona automáticamente provocando un efecto cadena.

Con las bombas el jugador podía eliminar bloques rompibles lo que le permitía avanzar por el laberinto. De forma aleatoria, dentro de los bloques rompibles se podían encontrar power ups que aparecían una vez destruidos los bloques que los almacenaban. Las bombas servían también para

eliminar a los enemigos del nivel. El jugador debía tener cuidado puesto que las bombas también le infringían daño a él.

Para pasar de nivel, el jugador debía acabar con todos los enemigos antes de que terminara el tiempo asignado. Además debía localizar la salida del nivel, que estaba escondida bajo uno de los bloques rompibles. Si se localizaba la salida antes de terminar con todos los enemigos esta estaba cerrada y se abría al morir el último enemigo del nivel.

El modo ***batalla*** o *Battle* es un modo multijugador, en el que el jugador debe enfrentarse a otros Bomberman con las mismas capacidades que él. Estos podían estar manejados por otros jugadores o por la máquina. El objetivo del juego no es ahora pasar de nivel si no acabar con el resto de jugadores. Es necesario hacerlo antes de un tiempo determinado. Si el tiempo llega a cero, el nivel empezará a cubrirse de bloques no rompibles que cierran las casillas (empezando por la casilla 1 de la fila 1 y cubriendo casilla a casilla cada fila). Si un jugador se encuentra en una casilla que se está cerrando morirá automáticamente. El último jugador vivo es el ganador.

2.2.2. Enemigos y Power Ups

En cuanto a los enemigos, en el juego original sólo había 1 con forma de globo y que se limitaba a perseguir al jugador si lo detectaba en el mapa (si estaba dentro de radio de acción del enemigo este perseguía al jugador, si no se limitaba a moverse aleatoriamente por el mapa). La versiones de **1985** y **1990** añadieron nuevos enemigos que además tenían distintos comportamientos, lo que aumentaba la dificultad del juego. Los enemigos incluidos, y que se consideran los originales de la saga son los siguientes:

- **Ballom:** basado en los enemigos de la primera versión, tienen forma de globo (su nombre viene del inglés *ballon*) y es el enemigo más lento del juego. Su comportamiento es el más simple, su *radar*[34] tiene poco alcance, detectando al jugador si está muy cerca de él. Su valor es 100 puntos.

- **Oneal:** basados en los enemigos de tipo *slime* del juego *Dragon Warrior*. Más rápido que el anterior y con un radar más amplio. Su valor es 200 puntos.
- **Doll:** más lento que *Oneal* pero con un radar más amplio. Su valor es 400 puntos.
- **Minvo:** más rápido y con un radar más amplio que *Doll*, es una moneda giratoria. Su valor es 800 puntos.
- **Kondoria:** el enemigo más lento del juego pero también de los más inteligentes. Su radar es más amplio que el de los enemigos anteriores y una vez detectado persigue al jugador. Además tiene la capacidad de atravesar los bloques rompibles. Su valor es 1000 puntos.
- **Ovapi:** creado como homenaje a los enemigos de *Pacman* (en concreto a *Blinky*), es lento y su radar no es muy amplio pero también puede atravesar los bloques rompibles. Su valor es 2000 puntos.
- **Pass:** muy rápido y con un radar muy amplio. También puede atravesar los bloques rompibles. Su valor es 4000 puntos.
- **Pontan:** versión avanzada de *Minvo*, es el enemigo más rápido del juego y el más inteligente. Su radar es el mayor y una vez detectado persigue al jugador. Su valor es 8000 puntos.

Figura 47: Diseño original de los enemigos del juego Bomberman

Estos son los enemigos de la versión de **1985**, considerados los originales (si exceptuamos el globo del primer juego). En posteriores versiones han sufrido distintas evoluciones como por ejemplo tener más de una vida, o modificar su velocidad y comportamiento una vez que han sido alcanzados por la primera bomba. También se han añadido nuevos enemigos no relacionados con estos en otras versiones, que no se comentarán por no pertenecer a la versión original.

En cuanto a los power ups, podían ser de tres tipos: temporales, afectando al jugador durante un tiempo limitado. De una vida, perdiéndose el efecto al morir; o definitivos manteniéndose el efecto hasta el final del juego. Los que se incluyeron en las versiones comentadas son:

- **Speed Up:** aumenta la velocidad de Bomberman. La velocidad se reiniciaba al perder una vida.
- **Bomb Up:** suma uno al número de bombas que el jugador puede poner de forma simultánea. El número de bombas se mantiene hasta el fin de la partida.
- **Fire Up:** suma uno a la potencia de las bombas, lo que significa mayor alcance. El alcance se mide en casillas por lo que por cada uno la bomba llegará una casilla más lejos. El alcance de las bombas se mantiene hasta el fin de la partida.
- **Wall Pass:** permitía a Bomberman comportarse como alguno de sus enemigos y atravesar los bloques rompibles. El efecto se mantenía hasta que se perdía una vida.
- **Remote Bomb:** este ítem permite detonar las bombas cuando el jugador quiera sin tener que esperar un tiempo determinado. Se pierde la capacidad si el jugador muere.
- **Bomb Pass:** permite a Bomberman pasar por encima de las bombas evitando que le bloqueen el camino. Su efecto desaparece al perder una vida.
- **Fire Pass:** permite a Bomberman ser alcanzado por las explosiones sin morir y sin perder corazones. Desaparece al perder una vida.
- **Heart/Shield:** permite a Bomberman ser alcanzado por una explosión sin morir. Se acumulan hasta un número máximo y se van restando si el jugador es alcanzado por una explosión.

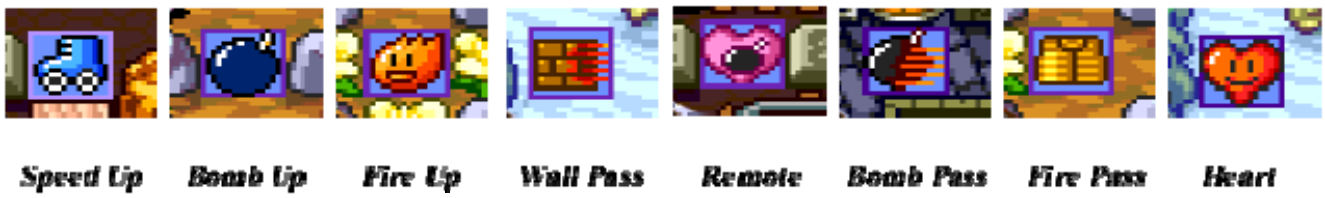

Figura 48: Iconos de los Power Ups del juego Bomberman

Estos son los power up originales. Posteriormente han indo apareciendo otros que se han hecho muy famosos como el ***Power Bomb*** , bombas de potencia máxima que alcanzan una fila y una columna completa, el ***Bomb Kick***, que permite golpear una bomba y desplazarla una distancia determinada o el ***Block Wall Pass*** que evita que el alcance de una explosión se detenga por un bloque rompible.

Por último hay que hacer referencia a dos juegos más, ***Super Bomberan*** y ***Super Bomberman 3***, ya que ambos introdujeron innovaciones que posteriormente se han mantenido en

muchos juegos de la saga y que se han utilizado como base en este proyecto. *Super Bomberman*, la primera secuela del juego lanzada para *Super NES*, fue la primera en incluir jefes finales (aunque en la versión de *Turbo Grafx 16* ya se aplicó una idea similar). Los niveles estaban agrupados en mapas o mundos que tenían en común los enemigos y el diseño de los elementos (si el mapa era "*Ice*", se destruían iglús y el decorado era nevado; si era "*Jungle*", se destruían palmeras y el decorado era la selva). Al superar todos lo niveles incluidos en un mapa (en el caso de *Bomberman* eran seis mapas con diez niveles cada uno) aparecía el jefe correspondiente. Todos tenían una característica común, su tamaño, bastante superior al de los objetos en pantalla llegando a ocupar 1/3 de esta. Su comportamiento, y por tanto la estrategia a seguir para derrotar a cada uno, era distinta.

Figura 49: Jefe final del mapa "Slammin' Sea" del juego Super Bomberman

En cuanto a ***Super Bomberman 3***, fue la primera vez que se vio en la saga a los *Rooeys*, personajes similares a los canguros, que aparecían de un huevo de un determinado color. Los huevos aparecían, al igual que los power up, dentro de los bloques rompibles. Cuando Bomberman encontraba un huevo podía montar un *Rooeys*, que según su color, le proporcionaba una habilidad determinada (efecto similar al de los power up). A partir de entonces, estos personajes han aparecido en muchas versiones del juego, variando las habilidades que proporcionaban a Bomberman.

Figura 50: Bomberman subido en un Rooey azul

La historia de Bomberman es mucho más completa de lo que se recoge en este punto, pero resultaría pesado repasar cada versión del juego ya que las variaciones son escasas. Bomberman sigue siendo un juego muy famoso hoy día y cada año años aparecen versiones para las consolas existentes.

Actualmente están en el mercado ***Bomberman Act Zero*** (en 3D con una estética futurista) y ***Bomberman Live*** (juego arcade del bazar) para *XBOX 360*, ***Bomberman PSP*** y ***Bomberman Land PSP*** para *PSP*, ***Bomberman Land DS*** para *Nintendo DS* y ***Bomberman Land Wii*** para *Wii*, además de existir una versión multijugador online para *PC* (con versión gratuita y extensiones de pago) llamada ***Bomberman Online***. Todo esto convierte a *Bomberman* en uno de los juegos más famosos de la historia.

2.3. XNA Game Studio

Este punto del documento recoge información relacionada con la plataforma utilizada para el desarrollo del proyecto. Se describe qué es *XNA* [144], sus funcionalidades principales, su estructura y los módulos que lo componen, además de justificar porqué se ha seleccionado *XNA* en su versión 3.1 [145] sobre otras herramientas disponibles para el desarrollo de videojuegos.

Antes de profundizar en la plataforma *XNA*, se hará referencia a otras herramientas o plataformas que permiten desarrollar videojuegos y con las que también se podría haber afrontado la realización de este proyecto.

2.3.1. Alternativas a XNA

Un *Game Engine*[35] [146] es una plataforma software diseñada para la creación y desarrollo de videojuegos. Actualmente existe gran cantidad de "*Game Engines*" en el mercado, tanto libres como de pago, orientados a las distintas plataformas de juego (*Wii*, *XBOX 360*) y sistemas operativos existentes *Windows*, *Linux* y *Mac OS X*. Este tipo de plataformas [147] proporcionan una serie de funcionalidades que incluyen un motor de renderización o "*render*" para gráficos 2D y 3D, un motor de física de cuerpos también conocido como motor de colisiones o motor de detección de colisiones, soporte para audio y sonidos, scripts, animaciones, inteligencia artificial, juego en red, streaming de contenidos, manejo de memoria, multijugador etc.

Con el tiempo estas plataformas se han convertido en verdaderos motores, reutilizándose en múltiples videojuegos. Un ejemplo es el famoso ***Unreal Engine***, motor de juego de PC y consolas creado por la compañía ***Epic Games***. Está escrito en *C++* y es compatible con varias plataformas como *Microsoft Windows*, *GNU/Linux*, *Mac OS*, *Mac OS X* y la mayoría de consolas *Dreamcast*, *Gamecube*, *Wii*, *XBOX*, *XBOX 360*, *PlayStation 2* y *PlayStation 3*. Ofrece además varias herramientas adicionales de gran ayuda para diseñadores y desarrolladores.

La historia de estos motores es larga, algunos piensan que comenzó con la Atari 2600 y su motor de desarrollo conocido por muchos hoy día como "*kernel*" (dando a entender que fue la primera herramienta que se puede considerar núcleo o motor de una serie de videojuegos). La primera generación de motores estuvo dominada por tres que hoy día siguen evolucionando y son base de juegos muy importantes. Estos son ***BRender*** de *Argonaut Software*, ***Renderware*** de *Criterion Software Limited* y ***Reality Lab*** de RenderMorphics. *Reality Lab* era el más rápido de los tres, por lo que poco después de aparecer fue comprado por *Microsoft* y utilizado como base para ***Direct3D***. ***Renderware*** fue adquirido más tarde por *EA*, siendo antes de esta adquisición motor de juegos tan importantes como los de las sagas *Grand Theft Auto* y *Burnout*.

La evolución del mundo de los videojuegos es tal, que ha hecho que hoy día existan motores similares a las versiones comentadas orientadas a desarrolladores amateur y que proporcionan muchas de las características de los motores comerciales. A continuación se hace referencia a los más importantes.

- **Adventure Game Studio (AGS)** [148]: plataforma de creación de aventuras gráficas. Proporciona una interfaz gráfica desde la que es posible crear la aventura gráfica sin ningún tipo de conocimiento de programación. Para desarrolladores más avanzados incluye un editor de scripting. Se distribuye en versión gratuita. A partir de la versión 2.72 el lenguaje de script está basado en un lenguaje orientado a objetos. Una vez dominado el lenguaje de script las limitaciones del programa son muy pocas e incluso se pueden hacer juegos de otros estilos, como *RPG*. La nueva versión 3.0, usa *Visual Studio .NET* sustituyendo al scripting. Una de sus desventajas es que, hasta la versión 3.0, los juegos desarrollados consumen muchos recursos, pero con la conversión a *VS .NET* se espera solucionar este problema y potenciar la herramienta.

- **Allegro** [149]**:** es una biblioteca para la programación de videojuegos desarrollada en C. Creada originalmente para *Atari ST*, con el tiempo ha ido evolucionando y hoy día es compatible con plataformas como *DOS*, *Linux*, *Windows*, *QNX*, y *MacOS X*. Sus funcionalidades principales están orientadas a los gráficos, sonidos, entrada de usuario (teclado, ratón y mandos de juego) y temporizadores. También tiene funciones matemáticas en punto fijo y coma flotante y funciones 3D. Está escrita en *C*. Para explotar al máximo sus funcionalidades se recomienda usar la biblioteca en conjunto con *OpenGL* mediante la extensión *AllegroGL*, que expande su funcionalidad hacia la de *OpenGL* y por lo tanto, al hardware.

- **Build** [150]**:** es un motor para juegos de disparo en primera persona tipo *Doom*. Creado por *Ken Silverman* para *3D Realms*, representa todos sus elementos en una malla 2D, utilizando formas 2D y sprites planos. Se considera un motor 2.5D, ya que a la geometría básica 2D le añade la componente de la altura. La herramienta, escrita en *C*, fue liberada en el año 2000.

- **Blender 3D** [151]**:** es una aplicación de diseño 3D. Es gratuita y se distribuye bajo la licencia *GNU General Public License*. Su funcionalidad principal es la de diseño 3D de figuras y modelos, pero también es posible realizar simulaciones de fluidos, detección de colisiones o

simulaciones físicas gracias por lo que en sus últimas versiones la potencia del motor se ha aprovechado para crear aplicaciones interactivas 3D incluidos juegos. Es compatible con *Linux*, *Mac OS X*, y *Microsoft Windows*. Todo el diseño se realiza a partir de su interfaz gráfica pero además incluye soporte de script en *Phyton*. Sus detractores critican la complejidad de su interfaz pero lo cierto es que es una herramienta muy potente tanto gráficamente como en físicas y colisiones y en su última versión, la 2.5, se ha proporcionado una nueva interfaz, y nuevos sistemas de animación y de entrada de usuario que aumentan la potencia de la herramienta.

- **Crystal Space** [152]**:** es un framework para el desarrollo de aplicaciones 3D. Se suele usar como motor de videojuegos, pero el framework es más general y puede ser usado para cualquier tipo de diseño 3D. Diseñado para ser totalmente portable, se ejecuta en *Microsoft Windows*, *Linux* y *Mac OS X*. *Crystal Space* es gratuito y se distribuye bajo la licencia *LGPL*. Es compatible con *OpenGL*, *SDL*, *X11* y *SVGALib*. Está programado en *C++* usando un diseño orientado a objetos.

- **dim3** [153]**:** también conocido como ***Dimension3***, es una herramienta de código libre orientada al diseño 3D y desarrollo de videojuegos. La herramienta es compatible con *Mac OS X*, *Microsoft Windows* y *Linux*. *dim3* se apoya en otra serie de herramientas y estándares para proporcionar sus funcionalidades, utiliza *OpenGL* para el renderizado, *OpenAL* para el audio, *JavaScript* como lenguaje de scripting, *XML* como formato de datos y *Simple DirectMedia Layer* como gestor de ventanas y video. Su primera versión estable es de **2007** pero ya ha conseguido el apoyo de *Apple* y se considera una de las herramientas que más crecerá en los próximos años.

- **DIV** [154]**:** *DIV Games Studio* es un lenguaje de programación basado en *C*, nacido para la creación de juegos en *MS-DOS*. Existen dos paquetes distribuidos: *DIV* y *DIV2*. Ambas incluían un compilador, un programa de diseño de sprites y un programa de edición de archivos de sonido, es decir. Gracias al interfaz, los usuarios no deben ser expertos programadores. Actualmente existe ***Fénix***, un lenguaje derivado de *DIV*, que dispone de compilador e intérprete para múltiples plataformas y sistemas operativos *Microsoft Windows*, *Linux* y *Mac OS X* etc. También ***Gemix Studio***, una herramienta profesional de desarrollo de videojuegos que incorpora un interfaz muy completo y una serie de lenguajes *DIV* con diferentes evoluciones; se le

considera el nuevo *DIV Games Studio 3* y la versión final será un producto comercial de pago con distintas versiones y licencias.

- **Doom Engine** [155]**:** es el motor grafico que se usó para los videojuegos *Doom* y *Doom II*. Originalmente desarrollado en computadoras *NeXT*, fue portado a *DOS* y posteriormente a otras consolas y sistemas operativos. El código fuente se hizo público en 1997 bajo licencia GNU en 1999. No es un verdadero motor 3D, ya que no hay componente de altura. Está algo obsoleto aunque se sigue usando para desarrollar clones de *Doom*.

- **Gamebryo** [156]**:** es un motor de gráficos 3D escrito en C++ y dirigido además al desarrollo de videojuegos. Su gran ventaja es que da soporte a todas las plataformas existentes actualmente *Windows DirectX 9* y *DirectX 10*, *Wii* / *WiiWare*, *PlayStation 3*, *PSP* y *XBOX 360* (incluyendo *XBOX Live Arcade*). Su principal inconveniente, que es de pago.

- **Game Maker** [157]**:** es una herramienta de desarrollo rápido de aplicaciones, basada en el paquete de desarrollo de software *SDK* para desarrollar videojuegos, creado en el lenguaje de programación *Delphi*, y orientado a usuarios novatos o con pocas nociones de programación. El programa es gratuito, aunque existe una versión comercial ampliada con características adicionales. Actualmente se encuentra en su versión 7.0. La interfaz principal usa un sistema de "arrastrar y soltar", que permite a los usuarios que no están familiarizados con la programación tradicional crear juegos. Incluye un conjunto de bibliotecas de estándar, que cubren movimiento, dibujo básico, y control simple de estructuras. Para extender la funcionalidad, los usuarios pueden construir bibliotecas de acciones personalizadas. Además usa su propio lenguaje de programación, el ***GML*** (*Game Maker Language*), con el que se pueden conseguir grandes mejoras en los juegos.

- **Game Studio** [158]: 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++*. Existen cuatro versiones y todas tienen actualizaciones gratuitas.

- **Glk/Glulx** [159]**:** *Glk* es un API con interfaz de usuario basada en texto, creada para el desarrollo de aventuras conversacionales sobre el lenguaje *Inform*. *Glulx* es una máquina virtual que utiliza la API *Glk*. El lenguaje de desarrollo de aventuras conversacionales *Inform* (y su biblioteca *InformATE*) puede ser compilado para esta máquina virtual. *Glulx* tiene límites mucho menos restrictivos en el tamaño del juego que otras herramientas similares y ofrece características de entrada y salida muy potentes gracias a *Glk*, permitiendo a desarrolladores de crear juegos más grandes, con interfaz de usuario más complejas. Quizá el gran problema sea su interfaz sólo textual y el hecho de tener que dominar un lenguaje poco conocido como *Inform*.

- **GtkRadiant** [160]**:** es un software desarrollado por *id Software* y *Loki Software*, utilizado para crear mapas de una cierta variedad de videojuegos. Nació como la herramienta de diseño de niveles de *Quake III Arena* sólo disponible para *Windows* y evolucionó (basado en GTK+) lanzándose versiones para *Linux* y *Mac OS*. Es independiente del motor de juego y semi-libre, el núcleo original es propietario y su uso en videojuegos comerciales sin el permiso de *id Software* no está permitido. Es una herramienta potente pero limitada a la creación de niveles y terrenos.

- **Havok** [161]**:** *Havok Game Dynamics SDK*, es un motor físico de simulación dinámica utilizado en videojuegos que recrea las interacciones entre objetos y personajes del juego. No es una herramienta ni una plataforma, se puede considerar un API de físicas y colisiones. Entre sus funcionalidades principales se encuentra detectar colisiones, gravedad, masa y velocidad en tiempo real llegando a recrear ambientes mucho más realistas y naturales. Se apoya en las liberias de *Direct3D* y *OpenGL* compatibles con *Shader Model 3.0*. Las últimas versiones incluyen *The Havok Behavior*, editor y compilador para desarrolladores que quieran implementar todo con *Havok*. Destaca su potencia y le avala la cantidad enorme de juegos comerciales que lo utilizan como motor gráfico. Sin embargo muchos piensan que es demasiado denso para usuarios no profesionales.

- **Irrlicht** [162]**:** es un motor 3D gratuito y de código abierto, escrito en *C++*, que puede ser usado tanto en *C++* como con lenguajes *.NET*. Sus características principales incluyen compatibilidad *Windows*, *Linux* y *MacOS*, soporte *Pixel Shader* y *Vertex Shader*, capacidad de manejo de interiores y exteriores, sistema de animaciones *skeletal*[36] y *morph*[37], sartículas, mapas de luces *enviromment mapping*[38] y sombras *stencil buffer*[39], sistema para interfaces 2D, rápido y fácil sistema de colisiones, infinidad de formatos 3D, texturas y lenguajes de programación soportados, así como un API totalmente documentada con ejemplos y tutoriales. Es una herramienta muy potente pero debido a que está orientada al desarrollo desde código es poco conocida.

- **LWJGL** [163]**:** *Lightweight Java Game Library* es una solución dirigida a programadores tanto amateurs como profesionales, destinada a la creación de juegos comerciales desarrollados en *Java*. Proporciona acceso a diversas bibliotecas multiplataforma, como *OpenGL* y *OpenAL*, permitiendo la creación de juegos de alta calidad con gráficos y sonido 3D. Por otro lado, LWJGL permite además acceder a controladores de juegos como *GamePads* y volantes. Todas estas funcionalidades están integradas en una sola API que facilita enormemente la creación de videojuegos en Java. Su objetivo no es proporcionar un engine gráfico que permita crear juegos de forma inmediata, sino dar acceso a los programadores *Java* a una tecnología y unos recursos que muchas veces se implementan incorrectamente. Por tanto, debe entenderse como un API. Está disponible bajo licencia BSD, y por lo tanto es de libre distribución.

- **M.U.G.E.N.** [164]**:** 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 con el sistema M.U.G.E.N.), escenarios varios, paquetes de visualización de pantalla (llamados screenpacks) o barras de vida; hasta crear un juego de peleas completo con todo un sistema establecido, personalizado y modificado al gusto del creador. Lo que más destacan sus usuarios 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** [165]**:** (*Object-Oriented Graphics Rendering Engine*) 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, proveen 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 llagado a usarse en algunos juegos comerciales.

- **Panda3D** [166]: es una librería de subrutinas para el renderizado y desarrollo de juegos en 3D. Está basado en los lenguajes *C++* y *Python*. Dispone de herramientas de creación de animaciones, tolerancia a errores, es rápido y se pueden crear juegos de resultado comercial. Es de licencia gratuita con algunas restricciones en ciertas librerías, y además dispone de bastante documentación.

- **Pygame** [167]**:** es un conjunto de módulos del lenguaje *Python* que permiten la creación de videojuegos en dos dimensiones de una manera sencilla. Está orientado al manejo de sprites. Gracias al lenguaje, se puede diseñar y desarrollar rápidamente. Los resultados pueden llegar a ser profesionales y cada vez es más usado, aunque está orientado a usuarios con altos conocimientos de programación. Funciona como interfaz de las bibliotecas *SDL*.

- **Quest3D** [168]**:** fusión de un motor de videojuego con una plataforma de desarrollo. Se usa para arquitectura, diseño de producto, videojuegos, software de entrenamiento y simuladores. Los datos y animaciones son importados de paquetes *CAD* a *Quest3D* donde son utilizados para la creación de aplicaciones interactivas 3D en tiempo real. Una de las características más importantes de *Quest3D* es que su entorno de desarrollo, pudiéndose modificar la aplicación mientras esta se ejecuta, pero sin la posibilidad de compilación de código como en los entornos de programación habituales. Este tipo de desarrollo lo hace accesible a más usuarios pero también lo limita con respecto a los más expertos. Existen varios tipos de licencias y las aplicaciones finalizadas pueden ser publicadas en diferentes formatos: fichero ejecutable ".exe" y visor WEB basado en control *ActiveX*.

- **RPG Maker** [169]**:** herramienta para *Windows* 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* necesitan el *RTP* (*Run Time Package*) que incluye gráficos para mapeados, personajes, música, efectos de sonido, etc. que pueden ser utilizados para crear nuevos juegos. Una característica muy destacada por los usuarios es que en las versiones de PC, es posible agregar nuevos materiales gráficos y sonoros personalizados a las librerías de proyecto. Hay una gran comunidad de usuarios que comparten librerías de gráficos, sonidos, fondos y demás archivos que ayudan a la elaboración de una nueva creación, o la modificación de una ya creada anteriormente. Como inconveniente a destacar, que sólo está orientado a juegos de rol.

- **Source** [170]**:** es un motor de videojuego desarrollado por la empresa *Valve* para las plataformas *Windows*, *XBOX*, *XBOX 360* y *PlayStation 3*. Utilizado por primera vez con el videojuego *Counter-Strike: Source* y después con *Half-Life 2*. *Source* fue creado para ir evolucionando poco a poco mientras la tecnología avanza, al contrario que los cambios de versión bruscos de sus competidores. Esto se vuelve especialmente relevante cuando se considera que esta ligado a *Steam*, el cual baja las actualizaciones automáticamente lo que hace que nuevas versiones del motor puedan llegar a toda la base de usuarios instantáneamente. Es muy potente en cuanto al manejo de animaciones, modelos 3D, físicas y colisiones. Está orientado a desarrolladores comerciales y sus licencias son de pago, aunque existen licencias especiales para modders[40].

- **Verge** [171]**:** *Vecna's Extraordinary Roleplaying Game Engine* es un motor que permite desarrollar juegos de plataformas en 2D**.** Inicialmente fue pensado, tal y como indica su nombre, para desarrollar juegos de rol, pero con el tiempo y las nuevas versiones los usuarios han ido orientando sus desarrollos hacia otros géneros 2D. *Verge* proporciona un interfaz gráfico desde el que es posible realizar todo el proceso de creación, aunque para usuarios más avanzados también soporta programación en *C#*. Es compatible con *Windows*, *Mac OS X* y *Linux*.

- **Wonderland** [172]**:** también conocido como *Proyecto Wonderland*, es un toolkit 3D para crear mundos virtuales colaborativos. Dentro de estos mundos, los usuarios pueden comunicarse por voz y compartir aplicaciones en vivo como navegadores web, documentos de texto y juegos. Está

construido sobre el *Proyecto Darkstar* y *Java 3D*. El motor gráfico fue reemplazado por *jMonkeyEngine* en su versión 0.5 a finales de **2008**.

- **XNA:** es un API de *Microsoft* orientado al desarrollo de videojuegos para las plataformas *XBOX 360*, *Zune* y PC. Técnicamente es un Marco de Trabajo (Framework), basado en *.NET Framework 2.0*; aunque en una implementación que proporciona un manejo optimizado para la ejecución de videojuegos. *XNA* pretende simplificar el desarrollo de juegos al programador y a través de su estructura facilita la gestión des gráficos, de sonido, de dispositivos, etc. Su problema principal es que carece de interfaz, lo que lo excluye para usuarios sin conocimientos de programación.

Estas son sólo algunas de las herramientas y plataformas que es posible encontrar en la red y que permiten diseñar y desarrollar videojuegos. Muchos de ellos tienen el inconveniente de las licencias, sobre todo los más avanzados y potentes. Otro añaden el problema del desarrollo, resultando muy complejo en algunos casos en los que la única manera de desarrollar es mediante código; o muy simple si se limitan a proporcionar un interfaz y no dar soporte adicional de código. De la misma manera muchas de estas herramientas se apoyan en librerías externas, o lenguajes de programación que les permitan proporcionar funcionalidades adicionales, sobre todo las que se centran en las 3D, de forma que es necesario manejar varias librerías, herramientas y lenguajes para conseguir un resultado óptimo. Por último resaltar que la mayoría proporciona una mínima documentación y escasos ejemplos, lo que hace aún más complicado el desarrollo.

Casi todos estos problemas quedan solventados en mayor o menor medida en *XNA*, API utilizado para el desarrollo de este proyecto. A continuación se detalla en qué consiste *XNA* y se justifica el porqué de la elección de *XNA* en su versión 3.1.

2.3.2. Qué es XNA y por qué usarlo

Microsoft XNA Game Studio es un conjunto de herramientas con un entorno de ejecución proporcionado por Microsoft que facilita el desarrollo y gestión de 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* un conjunto de herramientas con el objetivo de programar videojuegos. Se compone de dos librerías (*dll*) 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 Framework está concebido como una capa de abstracción sobre *DirectX* para permitir a los lenguajes *.NET* acceder a las funciones de *DirectX*. Además de proveer una capa de abstracción, provee de un modelo de programación intuitivo y fácil de utilizar, orientado a la programación de videojuegos y basado en componentes. Aunque *XNA* utiliza las funciones de *DirectX* por debajo, proporciona un acceso más intuitivo con clases y funciones más fáciles de usar desde arriba. *XNA* no sustituye a *DirectX*, pero sí a *Managed DirectX* (*MDX*), la primera aproximación de *Microsoft* para proveer de un acceso manejado desde los lenguajes *.NET* a *DirectX* (un mapeo tal cual de las clases y funciones). *XNA* extiende esta función de *MDX* y no solo provee un acceso a *DirectX*, sino que proporciona un verdadero sistema de trabajo orientado a la programación de videojuegos. Oculta las funciones gráficas complicadas de *DirectX* en clases y funciones más intuitivas y provee al

desarrollador de un modelo de programación robusto, sencillo y versátil. Por tanto *XNA* sí sustituye a *MDX*, que no tendrá más actualizaciones más allá de *MDX 2.0*.

XNA Game Studio funciona con todas las versiones de *Visual studio 2008*, incluida la versión gratuita (*Express*). Esta pensado para trabajar con el lenguaje *C#*. Al ser una biblioteca de ensamblados (*dll*), nada impide que se puedan referenciar estas dll 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 (es el programador el que decide si incluye ese ensamblado o no, pero hay consecuencias relativas a la velocidad y uso de la memoria al no usarlo).

XNA fue creado con el objetivo de facilitar al programador la creación de juegos, evitando que este se enfrente a problemas de integración con la plataforma, los dispositivos gráficos, etc. Aunque tiene el hándicap 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 gratis.

El modelo de negocio de *XNA* es claro, desarrollar es gratis, incluso publicar juegos para PC o *Zune* es totalmente gratuito; sólo en el caso de *XBOX 360* es necesaria una suscripción al servicio de *Creator's Club* de *Microsoft* [173]. Esta suscripción (que puede ser anual por 99$ o cuatrimestral por 49$), nos da la posibilidad de acceder a todo el material para desarrolladores y además publicar el juego creado (aunque antes dicho juego deberá pasar un control de calidad y contenidos a manos de la comunidad de desarrolladores). El precio del juego vendrá definido por el tamaño del mismo (entre otras variables) y varía entre 80, 200, 400 y 800 *MPoints*[41], ocupando como máximo 50 Mb los de 80 y 200 *MPoints* y como máximo 150 Mb los de 400 y 800 *MPoints*. A cambio de permitir que el juego pase a formar parte del catálogo de juegos de *Xbox Live Arcade*[42] (posteriormente conocido como *Xbox Live Community Games*[43] y actualizado en Octubre de 2009 a *Xbox Live Indie Games*[44]), *Microsoft* se queda con el 30% de las ventas que genere dicho juego. Es una oportunidad para los nuevos desarrolladores para introducirse dentro de la industria. De hecho, en la lista de los diez juegos más jugados de *XBOX Live* siempre suelen aparecer un par de juegos i*ndie*, por lo que una buena idea bien llevada a cabo suele tener su merecido premio dentro de este sistema.

Por otro lado, hay que destacar que los requisitos de *XNA* no son tan estrictos como los de muchos de sus competidores (sobre todo los motores 3D). Los aspectos técnicos a tener en cuenta para desarrollar con *XNA* son los siguientes:

- Visual C# Express Edition
- XNA Game Studio Express Edition
- .Net Framework 2.0 (se instala con Visual C# Express Edition)
- Una tarjeta gráfica que soporte shaders 1.1 (aunque se recomienda 2.0 o superior)

Si lo que se quiere es jugar y no desarrollar, para poder jugar a juegos desarrollados con *XNA* es necesario que el PC (además de *Windows*) tenga instalado:

- El .Net Framework 2.0
- El XNA Framework redistribuible
- DirectX instalado (8.1 o superior, preferiblemente 9.0c)
- Una tarjeta gráfica con shaders 1.1 o superior

Por último destacar que, la gran cantidad de documentación disponible así como las numerosas comunidades de usuarios y foros relacionados con *XNA*, compensan la falta de experiencia en programación con *.NET*, *C#*, o en programación de videojuegos. La curva de aprendizaje de *XNA* permite afianzar conocimientos de forma rápida, pudiendo realizar desarrollos complejos en poco tiempo. Actualmente es la plataforma para desarrollo amateur más usada.

Como conclusión, hay varios motivos por los que decantarse por *XNA,* entre los que destacan su facilidad de uso, al estar la plataforma diseñada para que desarrolladores amateur programen juegos con ella. Su robustez, basada en su Framework, el testeo ante fallos y las continuas revisiones y actualizaciones. La facilidad de conversión, que permite desde un mismo código generar versiones para PC, *XBOX 360* y *Zune*; y de publicación, al proporcionar una plataforma de lanzamiento robusta para desarrolladores amateur como es *Xbox LIVE Indie Games.* La amplia comunidad de usuarios, de foros y de grupos que la apoya y de ejemplos y tutoriales disponibles que hace que su curva de aprendizaje sea mucho menor. Y por último, el hecho de ser

totalmente gratuita y propiedad de *Microsoft*, lo que le permite proporcionar más funcionalidad que muchas otras herramientas con versiones de pago.

2.3.3. Arquitectura XNA

En este punto se detallará en qué consiste *XNA* técnicamente hablando y se describirá su núcleo y las librerías que lo componen así como su funcionamiento.

Como ya se ha comentado, el *Framework XNA* es un conjunto de librerías diseñadas y destinadas específicamente para el desarrollo de videojuegos. A continuación se muestra una imagen del modelo de capas de *XNA*, indicando donde está situado dicho Framework.

Figura 51: Modelo de capas de XNA

XNA se ejecuta bajo *.NET* y se puede apreciar que, según en la plataforma donde esté en ejecución, el framework *.NET* será diferente, debido a que *XBOX 360* y *Zune* utilizan el *Compact Framework* (que además no es el mismo de *Windows Mobile*). Por tanto *XNA* funcionará tanto en PC con *Windows* (*XP* o *Vista*), *XBOX 360* o *Zune*, pero no en *Windows Mobile*.

Desde el punto de vista del desarrollador XNA tiene dos ventajas evidentes:

- La primera es la facilidad de conversión a las tres plataformas (*Windows*, *XBOX 360* y *Zune*), gracias a las API que proporciona el *Framework XNA*, que en su mayoría sirven para todas las plataformas. Evidentemente hay funcionalidades que únicamente se pueden utilizar para una determinada plataforma, pero en general, muchos juegos son 100% compatibles con las tres.

- La otra, es la simplificación del proceso completo de desarrollo de un videojuego. Desarrollar un juego no es fácil, pero gracias al *Framework XNA*, se pueden evitar muchos problemas. El desarrollador no tendrá que preocuparse de crear la ventana del juego, ni de capturar los mensajes de error, ni de comprobar los eventos de inactividad, etc. Tampoco de enumerar los adaptadores gráficos o modos de pantalla, ni de crear dispositivos de Direct3D o gestionarlos cuando se minimiza el juego, o cambiar el tamaño de la pantalla. Todo esto y más, lo gestionará *XNA*.

A continuación, se muestra un diagrama con el modelo de capas que componen el *Framework XNA*. Como se puede ver en la figura 52, el modelo está compuesto por cuatro grandes capas: Plataforma, Núcleo del Framework, Framework extendido y Juegos.

Figura 52: Capas que componen el Framework XNA

Seguidamente, se detallan las características de cada capa.

- **Plataforma:** es la capa más baja, contiene las API nativas de *XNA* que son utilizadas mas arriba, en otro nivel, por las clases administradas. Algunas de las API que incluye esta capa son *Direct3D 9*, *XACT*, *XINPUT* o *XCONTENT*.

- **Núcleo del framework:** el núcleo, en realidad, es la primera capa del *Framework XNA*. Proporciona las funcionalidades básicas sobre las que las otras capas trabajan. Las áreas agrupadas en el núcleo son las siguientes:

  - **Gráficos**: como se ha visto anteriormente, la API gráfica (*Graphics*) que utiliza *XNA* es *Direct3D 9*. Un dato importante es que, a diferencia de *DirectX9*, *XNA* no tiene soporte para la pipeline de función fija (*Fixed-Function Pipeline* o *FFP*. Aún así, con XNA es posible programar Shaders y efectos propios, así como programar juegos sin tener que desarrollar los shaders. Existen clases que encapsulan shaders, como *BasicEffect* para definir el detalle final del renderizado de la escena y modelos 3D, o *SpriteBatch* para el manejo de sprites en 2D (*SpriteBatch* ha sido usada en este proyecto para el manejo de los sprites que componen las distintas animaciones).

  - **Audio**: *XNA* admite los siguientes formatos de archivos de sonido: .xap, .wav, .wma y .mp3. El sonido en *XNA*, antes de la versión 3.0 sólo se reproducía a través de *XACT* (*Cross-platform Audio Creation Tool*), que es una librería de alto nivel para audio diseñada por Microsoft para XBOX. A partir de la versión 3.0, además de importar archivos .xap, también es posible añadir directamente ficheros .mp3, .wav y wma. La gestión del audio ha sido uno de los puntos débiles de las versiones *XNA* (en la versión 3.0 era imposible cargar efectos de sonido con determinadas placas y tarjetas de sonido) hasta la 3.1 en la que el API de audio ha sido completamente remodelado.

  - **Entradas**: el API de entradas (*Input*) del usuario, es *XINPUT* de *DirectX*. Es de acceso inmediato, no requiere ningún tipo de inicialización previa. Hay clases para los controladores necesarios: el teclado (*Keyboard*), el ratón (*Mouse*) y por supuesto el

mando de la *XBOX 360* (*GamePad*). Uno de los problemas que más se critican de *XNA* es su deficiente entrada de teclado para el tratamiento de cadenas, teniendo que mapear tecla a tecla el teclado si se quiere disponer de esta funcionalidad.

- **Matemáticas:** el API de matemáticas (*Maths*), proporciona una gran colección de clases y métodos para el cálculo matemático. Incluye tipos de datos que muy usados en la programación de videojuegos para la carga de sprites, texturas y modelos 2D y 3D como *Vector2*, *Vector3*, *Vector4*, *Matrix*, etc. También incluye tipos de volúmenes como *BoundingBox*, *BoundingSphere* y *BoundingFrustum***,** que poseen métodos usados para la detección de colisiones**.** Las matemáticas en *XNA* usan por defecto el sistema de coordenadas basadas en la regla de la mano derecha.

- **Almacenamiento:** el API de almacenamiento (*Storage*), proporciona formas para leer y escribir los datos de los juegos, como por ejemplo, las partidas guardadas o las puntuaciones. *XNA* emula que el juego se ejecuta en una plataforma, por lo que se usa el mismo código para guardar datos en *XBOX* 360, *Windows* o *Zune*, de forma que no hace falta controlar de forma manual la asignación de perfil para dispositivo de almacenamiento o el estado del juego (obligatorios en *XBOX 360*).

- **Framework extendido:** el objetivo principal de esta capa es facilitar el desarrollo al programador. Existen dos elementos principales en esta capa, el *Modelo de aplicación*, cuyo propósito es apartar al programador de los problemas con la plataforma en la que se ejecuta el videojuego (crear una ventana, controlar los eventos, contadores de tiempo etc.). También proporciona la clase *GraphicsDevice*, que se encarga de la creación y gestión de los dispositivos gráficos. Además, en esta parte también se proporciona un modelo de componentes, que permite crear *GameComponent*. Esto permite crear librerías de componentes propias y reutilizarlas en diferentes proyectos o compartirlas con la comunidad. El segundo elemento es el *Administrador de Contenido* (*Content Pipeline*), que permite a los desarrolladores incorporar contenidos multimedia a los proyectos de *XNA*, tales como imágenes, sonido, modelos 3D, efectos, etc. Facilita el acceso a estos

archivos y proporciona una interfaz unificada sin una excesiva complejidad que permite una amplia gama de formatos de archivos diferentes.

- **Juegos:** es la capa más alta del *XNA*. Aquí se encuentra el código del propio juego, incluyendo los siguientes elementos:

  - **Kits de inicio:** existen multitud de kits de inicio para crear videojuegos, así no se tiene que programar el juego desde el principio. Se pueden utilizar estos kits como punto de partida o como complemento (existen kits de gestión de ventanas, de entrada de usuario, de emulación de teclado etc.).

  - **Código:** como su propio nombre indica es el código fuente que compone cada una de las clases del juego. Gracias al *Framework XNA*, el código será el mismo sea cual sea la plataforma para la que se trabaje (*XBOX 360*, PC o *Zune*), con ciertas excepciones en el apartado gráfico relacionadas con el alcance del hardware de cada plataforma.

  - **Contenido:** se refiere a los archivos de imagen, sonido, texturas etc. que deben ejecutarse en el juego. La gestión de contenidos en *XNA* se realiza gracias al *Content Pipeline*, un componente que permite añadir contenido a los juegos de una manera fácil y automática, indiferentemente del formato del archivo. El content pipeline se encargará de importar, compilar, cargar y gestionar todo el contenido en memoria.

  - **Componentes**: los componentes son librerías creadas por el desarrollador o importadas. Estas librerías suelen estar agrupadas por funcionalidades (control de entrada de usuario, manejo de pantallas, gestión de partidas guardadas) y su característica principal es la sencillez de integración. La función principal de un componente es la reusabilidad, proporcionando una funcionalidad concreta de forma sencilla e integrándose en el código sin necesidad de realizar grandes cambios.

2.3.4. XNA 3.1

Una vez justificada la elección de XNA y analizada su arquitectura, es necesario indicar porqué trabajar con la versión 3.1.

Cuando comenzó este proyecto, a finales de Febrero de 2009, la última versión de *XNA* estable era la 3.0. Con esta versión se estuvo trabajando hasta aproximadamente Septiembre. El problema principal que se encontró fue que, con ciertas configuraciones hardware de placa base y tarjeta de sonido (en particular de las *Realtek*), las instrucciones de carga de archivos de audio como "content.Load<SoundEffect>" generaban excepciones no controladas del tipo "InvalidOperationException". Este problema era bastante grave, puesto que hacían imposible la carga y reproducción de archivos de audio en el juego que no fueran .mp3, lo que impedía casi por completo la utilización de efectos de sonido (debido a que el tratamiento de ficheros .mp3 está asociado a la clase *Song,* con funcionalidades más reducidas que *SoundEffect*).

La versión 3.1, aparecida en Junio de 2009, además de otras mejoras; proporcionaba una gestión y tratamiento de audio totalmente renovadas, por lo que se optó por actualizar a dicha versión para resolver el problema. En cuanto al proyecto, la actualización a la versión no causó más problemas que su instalación y la ejecución del comando ***Upgrade***. Esto solucionó el problema de las excepciones no controladas en la carga de ficheros de audio como objetos *SoundEffect*.

Puesto que la versión definitiva de *XNA* usada para el desarrollo del proyecto es la 3.1, se detallan a continuación las mejoras introducidas.

La primera y más comentada, pero que posteriormente ha causado más decepción, es posibilidad de usar los *avatares* de *XBOX Live* como personajes de los juegos desarrollados. Se proporcionan nuevas funciones que permiten cargar los modelos 3D, utilizar animaciones predeterminadas así como generar nuevas animaciones. El mayor problema es que esta opción sólo es compatible con juegos desarrollados para *XBOX 360*.

Otra de las nuevas mejoras está relacionada con *Xbox LIVE Party Support*. Esta nueva librería proporciona a los desarrolladores funciones de red que permite que los jugadores se comuniquen cuando estén jugando en sesiones multijugador pero también cuando no estén jugando al mismo juego. Esta nueva funcionalidad permite ocho jugadores simultaneous en la misma conversación y se apoya en la mecánica de sesiones de *XBOX Live*. De nuevo, sólo compatible con juegos desarrollados para *XBOX 360*.

Una de las nuevas mejoras más aplaudida, sobre todo porque es común a todas las plataformas, es la inclusión de una librería de manejo de video, que permitirá utilizar videos ya generados como secuencias del propio juego. El nuevo API añade, entre otras, la funcionalidad de reproducir video a pantalla completa, de utilizar videos como texturas precargadas del juego, control total de reproducción (play/pause/stop), control de propiedades del video (tiempo de reproducción, tamaño, framerate), reproducción múltiple de varios elementos etc.

Otra de las mejoras más esperadas era la ya comentada sobre el API de audio. Además de la solución a las excepciones no controladas, el API mejora la gestión de memoria, algo muy criticado en versiones anteriores y que provocaba problemas en la reproducción de varios audios de forma simultanea.

Por último, la versión 3.1 incluye mejoras relacionadas con el Administrados de Contenidos (*Content Pipeline*), orientadas sobre todo a facilitas la creación de nuevos tipos, así como de atributos propios de tipos ya existentes.

Para terminar, hay que destacar que, con la nueva versión, se produjo una reestructuración en la comunidad de *XBOX Live*. Lo que en un principio era *XBOX Live Arcade* y recogía juegos tanto comerciales como amateur (aunque con unas características muy concretas como su tamaño o su precio en *Microsoft Points*), evolucionó más tarde surgiendo también *Xbox Live Community Games*. Esta nueva sección incluía juegos de desarrolladores amateur, dejando *Live Arcade* para juegos arcade de tipo comercial desarrollados por empresas. Por último, con la versión 3.1, ha aparecido una nueva sección, *Xbox LIVE Indie Games*, que incluye juegos de nuevos desarrolladores y quedando *Comunity Games* para desarrolladores arcade consolidados.

Capítulo 3
Análisis, diseño e implementación

Este capítulo incluye cada una de las tres grandes fases del desarrollo de un proyecto.

El primer apartado, el 3.1 Análisis, incluye la captura de requisitos y los diagramas previos al diseño.

El segundo apartado, el 3.2 Diseño, presenta el diagrama de clases del sistema profundizando en cada una de ellas.

Por último el tercero, el 3.3 Implementación, comenta detalles importantes en el proceso de codificación, justificando decisiones importantes como los métodos de detección de colisiones.

3.1. Fase de Análisis

Aquí comienza la segunda parte de la memoria de este proyecto. La primera parte, ya explicada, abarca todos los aspectos que se podrían considerar como introducción, aportando los conocimientos y herramientas necesarias para llevar a cabo un desarrollo completo; que en este caso es la elaboración de un juego.

Este punto se centra en la fase de análisis, fundamental en cualquier elaboración que contenga una arquitectura. Se parte de una necesidad y a partir de esta se toman las decisiones necesarias para llevar a la práctica un proyecto real. Como se trata de un desarrollo en el que el autor de la idea es la misma persona que realiza el diseño y el desarrollo final, se va a comenzar desde la base, es decir, desde la decisión de qué idea se quiere convertir en juego.

En primer lugar se va a plantear qué tipo de juego se va a realizar y por qué, a continuación comienza la fase de análisis especificándose los requisitos (menús, opciones, movimientos del jugador, enemigos, número de niveles, elementos que lo componen…). Otra parte importante del análisis, después de modelar el concepto en los distintos objetos que se puede descomponer, consiste en analizar los estados en los que se pueden encontrar estos y cómo transitar de unos a otros. Para ello se usarán técnicas de modelado como UML, para la elaboración de diagramas de casos de uso.

Una vez esté analizada y definida la idea se pasará al siguiente punto, la fase de diseño, que abarcará cada elemento por separado de la estructura y lo acercará más al desarrollador; aunque sin llegar todavía a la implementación.

3.1.1. Idea inicial

Como ya se comentó en el punto ***1.2 Objetivos***, la primera idea planteada para la realización del juego fue desarrollar un juego similar a *Super Gussun Oyoyo*. Este juego, por su mecánica sencilla (una mezcla de *Tetris* y *Lemmings* en el que el jugador debe construir un camino con piezas que van

cayendo para conseguir que el personaje principal alcance una salida), hubiera sido una buena manera de iniciarse en el desarrollo de videojuegos. Sin embargo, tras un primer análisis, se decidió que no encajaba con alguno de los objetivos principales del proyecto como la inclusión de inteligencia artificial; por lo que finalmente se desechó.

Una vez descartada la idea inicial, se inició un proceso de selección pensando en cuál sería la mejor opción. Se conocen muchos tipos diferentes de juegos, prácticamente puede haber tantas categorías como ocurrencias distintas se puedan tener. En este paso, tampoco es necesario justificar cada planteamiento con un motivo, a parte de la necesidad de dar algo de libertad a las ideas, de que la mecánica del juego suponga un crecimiento paulatino de la dificultad para el desarrollador (ya que inicialmente la experiencia en este tipo de desarrollos era casi nula) y que incluyera un módulo de inteligencia artificial.

Tras analizar unas cuantas ideas se decidió finalmente desarrollar un clon de *Bomberman*, ya que cumplía todos los requisitos expuestos. Su mecánica permitía ir afrontando el desarrollo con hitos de dificultad incremental (implementación básica del personaje, controles, colisiones, implementación de niveles y finalmente de enemigos) y permitía además la inclusión de IA en el diseño de los enemigos.

3.1.2. Identificación de requisitos

El propósito de este punto consiste en la identificación de los **requisitos de usuario** y los **requisitos software**. Una descripción del sistema a desarrollar a nivel de capacidades, restricciones, características del usuario, entorno operacional y dependencias.

Los requisitos de usuario reflejan las necesidades que debe cubrir el sistema desde el punto de vista del usuario. A partir de estos requisitos se podrá entender lo que espera el usuario del sistema a construir y llevarlo a cabo con éxito. Dentro de los de usuario se distinguen dos tipos: **requisitos de capacidad**, que representan una necesidad o servicio requerida por el usuario. Y **requisitos de restricción**, que son las restricciones impuestas por los usuarios sobre cómo se debe resolver el problema o cómo se debe alcanzar un objetivo.

Dentro de los requisitos software, se encuentra los **requisitos funcionales** que describen lo que debe hacer el sistema, y los **requisitos de rendimiento**, **interfaz**, **operación**, **recursos**, **comprobación**, **documentación**, **seguridad**, **calidad**, **mantenimiento**, **daño** y **aceptación de pruebas** que limitan la forma en que debe llevarse a cabo.

Estos requisitos tienen un alto grado de importancia en cuanto a verificación y seguimiento del juego, ya que este deberá cubrir todos y cada uno de los requisitos identificados para afirmar que está completo.

A continuación se definen los requisitos de capacidad:

IDENTIFICADOR: RUC-01	
Descripción	*Ejecutable*: Lanzar el juego a través de un ejecutable.

IDENTIFICADOR: RUC-02	
Descripción	*Jugar*: Comenzar una nueva partida.

IDENTIFICADOR: RUC-03	
Descripción	*Configurar Opciones*: Configurar los parámetros ajustables del juego. Los parámetros configurables son: - Seleccionar dificultad - Seleccionar tecla para soltar las bombas - Activar/desactivar el control analógico - Configurar la sensibilidad del analógico - Ajustar el volumen de la música y los efectos sonoros - Activar/desactivar la música

IDENTIFICADOR: RUC-04	
Descripción	*Activar los valores por defecto*: Aplicar los valores por defecto a los parámetros ajustables del juego.

IDENTIFICADOR: RUC-05	
Descripción	*Guardar opciones*: Guardar los valores seleccionados para los parámetros ajustables del juego.

IDENTIFICADOR: RUC-06	
Descripción	*Mostrar puntuaciones*: Mostrar las puntuaciones más altas alcanzadas en el juego.

IDENTIFICADOR: RUC-07	
Descripción	*Salir*: Cerrar la ventana de juego.

IDENTIFICADOR: RUC-08	
Descripción	*Mover el personaje*: Desplazar al personaje de forma horizontal y vertical.

IDENTIFICADOR: RUC-09	
Descripción	*Manejar bombas*: El personaje podrá depositar bombas en el mapa y detonarlas si se tiene la capacidad.

IDENTIFICADOR: RUC-10	
Descripción	*Power ups*: El personaje podrá recoger ítems que le afectarán de forma temporal o continua. Los power ups pueden ser de cuatro tipos: - Heart - Bomb Up - Remote - Fire Up

IDENTIFICADOR: RUC-11	
Descripción	*Pausa*: Detener una partida temporalmente manteniendo su estado actual para reanudarla más tarde sin cerrar el juego.

IDENTIFICADOR: RUC-12	
Descripción	*Puntuación máxima*: El jugador podrá introducir un nombre de usuario para registrar su puntuación y así identificarse en la lista de puntuaciones máximas.

A continuación se definen los requisitos de restricción identificados:

IDENTIFICADOR: RUR-01	
Descripción	*Dificultad*: El jugador podrá seleccionar dos niveles de dificultad: Fácil y Normal.

IDENTIFICADOR: RUR-02	
Descripción	*Soltar Bombas*: El jugador podrá seleccionar entre dos teclas para activar la acción de soltar una bomba: Enter y Espacio. La tecla que quede libre se asigna de forma automática a la acción detonar bomba remoto. Enter corresponde al botón A en *XBOX 360* y Espacio al botón B.

IDENTIFICADOR: RUR-03	
Descripción	*Activar analógico*: El jugador podrá activar/desactivar los controles analógicos en *XBOX 360*, de forma que sólo se pueda controlar al jugador con la cruceta.

IDENTIFICADOR: RUR-04	
Descripción	*Sensibilidad del analógico*: El jugador podrá seleccionar la sensibilidad de los controles analógicos en *XBOX 360*, en un rango de 1 a 10.

IDENTIFICADOR: RUR-05	
Descripción	*Volumen*: Es posible seleccionar el volumen de la música y los efectos sonoros. El volumen varía en un rango de 1 a 10, aunque si se quiere desactivar por completo se puede seleccionar la opción Off.

IDENTIFICADOR: RUR-06	
Descripción	*Música*: Es posible activar/desactivar la reproducción de la música desde el Menú de Opciones.

IDENTIFICADOR: RUR-07	
Descripción	*Puntuación máxima*: El juego muestra las cinco puntuaciones más altas.

IDENTIFICADOR: RUR-08	
Descripción	*Nombre de jugador*: El jugador puede introducir un nombre de tres letras que le identifique en la lista de puntuaciones máximas.

IDENTIFICADOR: RUR-09	
Descripción	*Dificultad fácil*: Al seleccionar fácil como nivel de dificultad varían los siguientes parámetros del juego: - El tiempo para superar cada nivel es de 240 segundos - El ratio de aparición de Power Ups es de un 10% - Al perder una vida el jugador perderá una unidad de potencia en el alcance de sus bombas si es mayor que uno - Al perder una vida el jugador perderá una bomba en el máximo de bombas que puede soltar de forma simultánea - El número de corazones iniciales es dos - El número de vidas iniciales es cinco

IDENTIFICADOR: RUR-10	
Descripción	*Dificultad normal*: Al seleccionar normal como nivel de dificultad varían los siguientes parámetros del juego: - El tiempo para superar cada nivel es de 210 segundos - El ratio de aparición de Power Ups es de un 5% - Al perder una vida, la potencia de alcance de las bombas del jugador se reiniciará a uno - Al perder una vida el jugador el número máximo de bombas que el jugador puede soltar de forma simultánea se reiniciará a uno - El número de corazones iniciales es uno - El número de vidas iniciales es cuatro

IDENTIFICADOR: RUR-11	
Descripción	*Formatos*: El formato de los ficheros de sprites debe ser .JPG o .PNG, el formato de los ficheros de audio para música .MP3 y el formato de los ficheros de audio para efectos sonoros .WAV.

IDENTIFICADOR: RUR-12	
Descripción	*Entorno operativo*: El juego compilado será compatible con los sistemas operativos Windows XP y Vista que tengan instalado XNA Framework Redistributable 3.1.

A continuación se enumeran los requisitos software:

IDENTIFICADOR: RSF-01	
Fuente	RUC-01
Descripción	*Lanzar el juego*: El juego, en su versión PC, se lanzará mediante la ejecución de un fichero .EXE generado al compilar.

IDENTIFICADOR: RSF-02	
Fuente	RUC-01
Descripción	*Mostrar Menú Principal*: Una vez lanzado se mostrará la pantalla de Menú Principal que incluye el logo del juego y las siguientes entradas: - Jugar - Opciones - Puntuaciones - Salir

IDENTIFICADOR: RSF-03	
Fuente	RUC-03
Descripción	*Mostrar Menú Opciones*: Para mostrar el Menú Opciones hay que seleccionar la entrada correspondiente. Dicho menú muestra las siguientes entradas con sus respectivos valores: - Dificultad: Fácil/Normal - Soltar Bombas: Espacio/Enter - Activar analógico: Activado/Desactivado - Sensibilidad del analógico: 1..10 - Volumen: Off/1..10 - Música: On/Off - Valores por defecto - Salir y guardar

IDENTIFICADOR: RSF-04	
Fuente	RUR-01
Descripción	*Seleccionar Dificultad*: En el Menú Opciones el usuario podrá seleccionar el nivel de dificultad del juego, eligiendo entre **Fácil** y **Normal**.

IDENTIFICADOR: RSF-05	
Fuente	RUR-02
Descripción	*Seleccionar tecla bombas*: En el Menú Opciones el usuario podrá seleccionar qué tecla usar para la acción de soltar las bombas, eligiendo entre **Enter** y **Espacio**. La tecla que no seleccione se asignara automáticamente a la acción detonar las bombas remoto. En el caso de *XBOX 360*, Enter está mapeado como el botón A y Espacio como el B.

IDENTIFICADOR: RSF-06	
Fuente	RUR-03
Descripción	*Seleccionar activación de analógico*: En el Menú Opciones el usuario podrá seleccionar si quiere o no activar los controles analógicos del mando de *XBOX 360*. Si están activados, el jugador podrá manejar al personaje con el analógico izquierdo y con la cruceta; si están desactivados, sólo podrá hacerlo con la cruceta.

IDENTIFICADOR: RSF-07	
Fuente	RUR-04
Descripción	*Seleccionar sensibilidad del analógico*: En el Menú Opciones el usuario podrá seleccionar el grado de sensibilidad de los controles analógicos del mando de *XBOX 360*. A mayor sensibilidad, más rápido responderá el personaje a un toque más leve del control analógico.

IDENTIFICADOR: RSF-08	
Fuente	RUR-05
Descripción	*Ajustar volumen*: El usuario puede ajustar el volumen de la música y los efectos sonoros desde el Menú Opciones eligiendo un valor entre 1 y 10. El valor **Off** hace que de dejen de escuchar tanto la música como los efectos sonoros. Seleccionando un valor de 1 a 10 se incrementa gradualmente el volumen de ambos elementos.

IDENTIFICADOR: RSF-09	
Fuente	RUR-06
Descripción	*Activar música*: En el Menú Opciones el jugador podrá activar o desactivar la música. Si se selecciona **Off** la música quedará desactivada pero los efectos sonoros se seguirán escuchando.

IDENTIFICADOR: RSF-10	
Fuente	RUC-04
Descripción	*Valores por defecto*: Es posible aplicar los valores por defecto a las entradas del Menú Opciones. Una vez aplicados, los valores de las entradas configurables serán los siguientes: - Dificultad: Normal - Soltar Bombas: Espacio - Activar analógico: Desactivado - Sensibilidad del analógico: 5 - Volumen: 7 - Música: On

IDENTIFICADOR: RSF-11	
Fuente	RUC-05
Descripción	*Salir y guardar*: Es posible salir del Menú Opciones guardando las modificaciones mediante la opción **Salir y guardar**. Esto mantendrá los cambios realizados en la partida actual y en las futuras, hasta que alguna de las opciones vuelva a ser modificada y se vuelva a guardar.

IDENTIFICADOR: RSF-12	
Fuente	RUC-06
Descripción	*Puntuaciones*: Desde el Menú Principal se podrán mostrar las cinco puntuaciones máximas seleccionando la opción **Puntuaciones**. El Menú Puntuaciones muestra el nombre de jugador, la puntuación y el nivel alcanzados.

IDENTIFICADOR: RSF-13	
Fuente	RUC-07
Descripción	*Cerrar juego*: Desde el Menú Principal el jugador podrá cerrar la ventana de juego seleccionando la opción **Salir**.

IDENTIFICADOR: RSF-14	
Fuente	RUC-02
Descripción	*Jugar partida*: Desde el Menú Principal el jugador podrá comenzar una nueva partida seleccionando la opción **Jugar**.

IDENTIFICADOR: RSF-15	
Fuente	RUC-02
Descripción	*Mostrar información*: Una vez iniciada la partida, antes de iniciar cada nivel, puede aparecer una pantalla con información importante para el jugador relativa al juego.

IDENTIFICADOR: RSF-16	
Fuente	RUC-11
Descripción	*Pausar partida*: Una vez iniciada la partida el jugador puede pausarla en cualquier momento pulsando **Escape** durante 2 segundos. Desde la pantalla de Pausa el jugador puede reanudar la partida seleccionando **Reanudar juego**, o abandonar la partida seleccionando **Finalizar Juego**.

IDENTIFICADOR: RSF-17	
Fuente	RUC-08
Descripción	*Controlar el personaje*: El jugador puede desplazar al personaje de forma vertical y horizontal. El desplazamiento vertical responde a las fechas arriba y abajo del teclado y el horizontal a las teclas izquierda y derecha. En caso de jugar en *XBOX 360* el movimiento responde a la cruceta y el analógico.

IDENTIFICADOR: RSF-18	
Fuente	RUC-09
Descripción	*Utilizar bombas*: El jugador puede depositar las bombas utilizando la tecla que haya definido en el Menú de Opciones. También podrá detonar bombas remoto (si tiene la capacidad) con la tecla que quede libre del par Enter/Espacio.

IDENTIFICADOR: RSF-19	
Fuente	RUC-09
Descripción	*Explotar bombas*: El jugador puede eliminar bloques rompibles del mapa y enemigos si la explosión de una bomba les alcanza.

IDENTIFICADOR: RSF-20	
Fuente	RUC-10
Descripción	*Recoger Power Ups*: El jugador puede recoger Power Ups que aparezcan en el mapa. Los efectos son los siguientes: - Heart (Corazón): permite ser alcanzado por una explosión o tocado por un enemigo. Son acumulables hasta un máximo de tres. - Bomb Up (Bomba): aumenta en uno el máximo de bombas que se pueden depositar de forma simultánea. - Remote (Bomba remoto): permite detonar las bombas depositadas cuando el jugador quiera. - Fire Up (Potencia): aumenta la potencia de las bombas en uno.

IDENTIFICADOR: RSF-21	
Fuente	RUC-02
Descripción	*Avanzar nivel*: El jugador puede avanzar al siguiente nivel si elimina a todos los enemigos del nivel actual. Para avanzar al nivel siguiente deberá además encontrar la puerta de salida que se encuentra escondida bajo uno de los bloques rompibles del nivel. La puesta permanecerá cerrada hasta que todos los enemigos del nivel mueran.

IDENTIFICADOR: RSF-22	
Fuente	RUC-02
Descripción	*Morir*: Si el jugador es alcanzado por un enemigo o por una explosión perderá un corazón. Si sólo dispone de un corazón perderá una vida. Si sólo dispone de una vida finalizará la partida actual.

IDENTIFICADOR: RSF-23	
Fuente	RUC-02
Descripción	*Reiniciar nivel*: Si el jugador es alcanzado por un enemigo o por una explosión y sólo dispone de un corazón perderá una vida. Si tiene más de una vida aparecerá la pantalla ¡Has muerto!, en la que el jugador podrá optar por **Reiniciar el nivel** o **Finalizar el juego**. Si finaliza el juego volverá al Menú principal, si Reinicia el nivel volverá al jugar el nivel donde perdió la última vida.

IDENTIFICADOR: RSF-24	
Fuente	RUC-02
Descripción	*Fin del juego*: Si el jugador es alcanzado por un enemigo o por una explosión y sólo dispone de un corazón perderá una vida. Si era su última vida aparecerá la pantalla ¡Game Over!, en la que el jugador podrá introducir sus iniciales y **Finalizar el juego**. Al finalizar volverá al Menú principal.

IDENTIFICADOR: RSF-25	
Fuente	RUC-02
Descripción	*Controlar enemigos*: Los enemigos del mapa serán controlados por el módulo de IA del juego. Detectarán al jugador a una distancia determinada e intentarán alcanzarlo y huir de sus ataques.

IDENTIFICADOR: RSF-26	
Fuente	RUC-02
Descripción	*Ganar*: Cuando el jugador supere todos los niveles de los que consta el juego aparecerá la pantalla ¡Has ganado! Desde la que podrá introducir sus iniciales. Esta es la pantalla final del juego y alcanzarla supone ganar. Una vez introducido el nombre se redirige al jugador al Menú Principal.

IDENTIFICADOR: RSF-27	
Fuente	RUC-12
Descripción	*Introducir puntuación máxima*: Desde las pantallas de ¡Game Over! y ¡Has ganado!, el jugador puede introducir su nombre o sus iniciales para que su puntuación quede registrada en el histórico de puntuaciones. Aparecerá en las puntuaciones máximas si es una de las cinco más altas.

Una vez enumerados los requisitos software funcionales se exponen los requisitos software de interfaz:

IDENTIFICADOR: RSI-01	
Fuente	RUR-09
Descripción	*Iniciales*: Las iniciales o nombre que el jugador puede introducir para identificarse en la lista de puntuaciones máximas tendrá un máximo de tres caracteres.

IDENTIFICADOR: RSI-02	
Fuente	RUR-09
Descripción	*Formato de puntuaciones máximas*: El Menú Puntuaciones mostrará las cinco puntuaciones máximas con el formato Iniciales, Puntuación Nivel.

IDENTIFICADOR: RSI-03	
Fuente	RUR-12
Descripción	*Formato de archivos*: Los archivos de sprites incluidos en el proyecto serán del formato .JPG o .PNG, la música estará en formato .MP3 y los efectos de sonido en formato .WAV.

IDENTIFICADOR: RSI-04	
Fuente	RUR-09
Descripción	*Formato de puntuaciones máximas*: El Menú Puntuaciones mostrará las cinco puntuaciones máximas con el formato Iniciales, Puntuación Nivel.

IDENTIFICADOR: RSI-05	
Fuente	NA
Descripción	*Niveles*: Los niveles son laberintos rectangulares de dimensiones 11x17. 40 posiciones están ocupadas por bloques no rompibles colocados en las posiciones pares de las filas pares (si se empieza a contar en uno). Los bloques rompibles se colocan de forma aleatoria.

IDENTIFICADOR: RSI-06	
Fuente	NA
Descripción	*Software y formato de ficheros*: Se utilizará Visual Studio .NET 2008 y XNA 3.1 como herramientas de desarrollo. Los ficheros que almacenan las opciones y las puntuaciones estarán en formato XML.

En cuanto a los requisitos de operación:

IDENTIFICADOR: RSO-01	
Fuente	NA
Descripción	*Tamaño*: Una vez finalizado el desarrollo, el juego, incluyendo todo el contenido de animaciones, música y efectos sonoros no excederá de 50Mb, máximo tamaño de los juegos con valor de 200 *MPoints.*

En lo referente a los requisitos de recursos:

IDENTIFICADOR: RSRe-01	
Fuente	NA
Descripción	*Desarrollo*: Para el desarrollo del juego es necesario un equipo con Windows XP o Vista con las siguientes características: - Visual Studio .NET 2008 - XNA Game Studio 3.1 - Una tarjeta gráfica que soporte shaders 1.1 (recomendado 2.0)

IDENTIFICADOR: RSRe-02	
Fuente	NA
Descripción	*Ejecución*: Para la ejecución del juego es necesario un equipo con las siguientes características: - Windows XP o Vista - XNA Framework Redistributable 3.1. - DirectX instalado (8.1 o superior, preferiblemente 9.0c) - Una tarjeta gráfica que soporte shaders 1.1 (recomendado 2.0)

No se han identificado requisitos esenciales de comprobación ni seguridad. En cuanto a los requisitos de documentación y calidad vienen marcados por la exigencia propia de un proyecto fin de carrera por lo que tampoco se reflejan requisitos adicionales de este tipo.

3.1.3. Especificación de casos de uso

Una vez expuestos los requisitos de la aplicación, se deben elaborar los distintos diagramas de casos de uso resultantes de los requisitos definidos, así como la especificación textual de cada caso de uso

para su mejor entendimiento. A diferencia de otras aplicaciones de escritorio, los videojuegos, por su naturaleza; permiten agrupar los casos de uso en varios grupos que son comunes a casi todos los videojuegos. Así se tienen los casos de uso de tipo ***Player Input***, que agrupan las funcionalidades relacionadas con la entrada de usuario, los de tipo ***View*** o ***Display***, que agrupan las funcionalidades relacionadas con menús y gráficos; las de tipo ***Game Object Interaction***, que recoge la interacción del juego con los distintos objetos que lo componen y por último las de tipo ***Miscellaneous***, que como su propio nombre indica recoge acciones variadas que no se han encajado en los otros grupos.

En los casos de uso se especifica qué hace el sistema en respuesta a una interacción de un usuario externo, por tanto no se tendrá en cuenta al propio sistema como actor (sólo sistemas externos que interactúen con el propio se pueden considerar actores). De esta forma, y por la naturaleza propia de algunas funcionalidades típicas de los videojuegos, surgirán casos de uso que quizá parezcan propios del sistema como la detección de colisiones o ser alcanzado por un enemigo; pero serán representados con el jugador como actor.

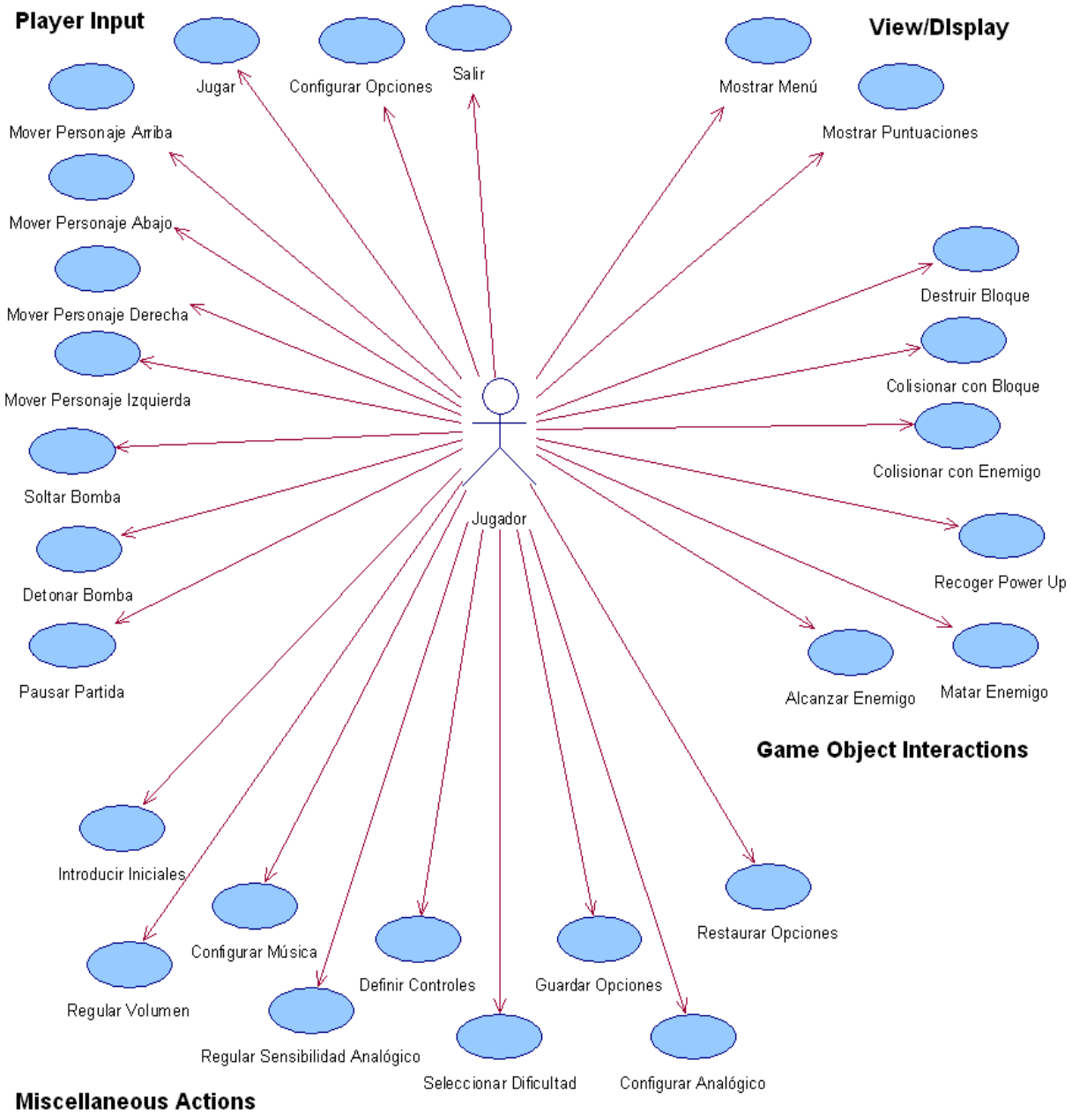

Figura 53: Diagrama de Casos de Uso

Como se ve en la figura 53, los casos de uso quedaría agrupados de la siguiente manera, en **Player Input**: Salir, Configurar Opciones, Jugar, Mover Personaje Arriba, Mover Personaje Abajo, Mover Personaje Derecha, Mover Personaje Izquierda, Soltar Bomba, Detonar Bomba y Pausar Partida. En **View/Display**: Mostrar Menú y Mostrar Puntuaciones. En **Game Object Interaction**: Destruir Bloque, Colisionar con Bloque, Colisionar con Enemigo, Recoger Power Up, Alcanzar

Enemigo y Matar Enemigo. Por último en **Miscellaneous Actions**: Introducir Iniciales, Regular Volumen, Configurar Música, Regular Sensibilidad Analógico, Definir Controles, Seleccionar Dificultad, Guardar Opciones, Configurar Analógico, Restaurar Opciones.

A continuación, se adjuntan las tablas con la especificación textual de cada caso de uso, para mayor comprensión. De cada caso de uso se especificarán los siguientes términos:

- **Identificador**: representa de forma unívoca cada caso de uso. La sintaxis de nombrado será la siguiente: CU-XX, donde XX serán 2 dígitos que numeran ordenadamente los casos de uso.
- **Nombre**: título corto que se le da a cada caso de uso, de acuerdo al diagrama anterior.
- **Actores**: son los usuarios que intervienen directamente en la realización del caso de uso.
- **Objetivo**: se refiere al objetivo concreto del caso de uso.
- **Precondiciones**: condiciones que deben darse previamente en el sistema para que el caso de uso pueda efectuarse.
- **Postcondiciones**: indican el estado del sistema después de la ejecución del caso de uso.

IDENTIFICADOR: CU-01	
Nombre	Jugar
Actores	Jugador
Objetivo	Comenzar una partida desde el primer nivel
Precondiciones	- Haber arrancado el juego
Postcondiciones	- El sistema carga el primer nivel - El usuario comienza la partida

IDENTIFICADOR: CU-02	
Nombre	Configurar Opciones
Actores	Jugador
Objetivo	Acceder al Menú de Opciones para configurarlas a medida del jugador
Precondiciones	- Haber arrancado el juego
Postcondiciones	- El sistema carga el Menú de Opciones

IDENTIFICADOR: CU-03	
Nombre	Salir
Actores	Jugador
Objetivo	Cerrar la ventana de juego
Precondiciones	- Haber arrancado el juego
Postcondiciones	- Se sale de la aplicación

IDENTIFICADOR: CU-04	
Nombre	Mover Personaje Arriba
Actores	Jugador
Objetivo	Desplazar al personaje en el eje de coordenadas X en una unidad positiva
Precondiciones	- Haber comenzado una partida - Que la posición de destino no esté ocupada por un bloque no rompible
Postcondiciones	- El personaje se desplaza en el eje de coordenadas X de forma positiva

IDENTIFICADOR: CU-05	
Nombre	Mover Personaje Abajo
Actores	Jugador
Objetivo	Desplazar al personaje en el eje de coordenadas X en una unidad negativa
Precondiciones	- Haber comenzado una partida - Que la posición de destino no esté ocupada por un bloque no rompible
Postcondiciones	- El personaje se desplaza en el eje de coordenadas X de forma negativa

IDENTIFICADOR: CU-06	
Nombre	Mover Personaje Izquierda
Actores	Jugador
Objetivo	Desplazar al personaje en el eje de coordenadas Y en una unidad negativa
Precondiciones	- Haber comenzado una partida - Que la posición de destino no esté ocupada por un bloque no rompible
Postcondiciones	- El personaje se desplaza en el eje de coordenadas Y de forma negativa

IDENTIFICADOR: CU-07	
Nombre	Mover Personaje Derecha
Actores	Jugador
Objetivo	Desplazar al personaje en el eje de coordenadas Y en una unidad positiva
Precondiciones	- Haber comenzado una partida - Que la posición de destino no esté ocupada por un bloque no rompible
Postcondiciones	- El personaje se desplaza en el eje de coordenadas Y de forma positiva

IDENTIFICADOR: CU-08	
Nombre	Soltar Bomba
Actores	Jugador
Objetivo	Depositar una bomba en el mapa en la posición actual del personaje
Precondiciones	- Haber comenzado una partida
Postcondiciones	- Una bomba sin detonar se deposita y se inicia el contador de detonación

IDENTIFICADOR: CU-09	
Nombre	Detonar Bomba
Actores	Jugador
Objetivo	Detonar una bomba depositada en el mapa
Precondiciones	- Haber comenzado una partida - Haber depositado una bomba en el mapa - Haber recogido el Power Up Remote Bomb
Postcondiciones	- La bomba que más tiempo lleve depositada en el mapa explota

IDENTIFICADOR: CU-10	
Nombre	Pausar Partida
Actores	Jugador
Objetivo	Pausar temporalmente una partida activa guardando el estado del juego
Precondiciones	- Haber comenzado una partida
Postcondiciones	- Se muestra el Menú de Pausa - El jugador podrá volver a la partida o salir

IDENTIFICADOR: CU-11	
Nombre	Introducir Iniciales
Actores	Jugador
Objetivo	Introducir un nombre que identifique al jugador en el Menú de Puntuaciones
Precondiciones	- Haber comenzado una partida - Haber muerto o finalizado el juego
Postcondiciones	- Se muestra el menú correspondiente - El jugador podrá introducir un nombre de tres caracteres como máximo

IDENTIFICADOR: CU-12	
Nombre	Regular Volumen
Actores	Jugador
Objetivo	Seleccionar el nivel del volumen de la música y los efectos
Precondiciones	- Abrir el Menú de Opciones
Postcondiciones	- Se establece el volumen al nivel seleccionado por el usuario

IDENTIFICADOR: CU-13	
Nombre	Configurar Música
Actores	Jugador
Objetivo	Activar o desactivar la reproducción de música
Precondiciones	- Abrir el Menú de Opciones
Postcondiciones	- Se activa o desactiva la reproducción de música

IDENTIFICADOR: CU-14	
Nombre	Regular Sensibilidad Analógico
Actores	Jugador
Objetivo	Seleccionar el nivel de sensibilidad del control analógico de *XBOX 360*
Precondiciones	- Abrir el Menú de Opciones
Postcondiciones	- Se establece el grado de sensibilidad del control analógico izquierdo

IDENTIFICADOR: CU-15	
Nombre	Definir Controles
Actores	Jugador
Objetivo	Asignar las teclas para depositar bombas y detonarlas
Precondiciones	- Abrir el Menú de Opciones
Postcondiciones	- Se establecen las teclas para depositar bomba y detonarla

IDENTIFICADOR: CU-16	
Nombre	Seleccionar dificultad
Actores	Jugador
Objetivo	Seleccionar el nivel de dificultad del juego en fácil o normal
Precondiciones	- Abrir el Menú de Opciones
Postcondiciones	- Se estable el tiempo máximo para superar cada nivel - Se estable el ratio de aparición de Power Ups - Se establece el número de vidas y de corazones iniciales - Se establece la política de efecto de los Power Ups al perder una vida

IDENTIFICADOR: CU-17	
Nombre	Guardar Opciones
Actores	Jugador
Objetivo	Almacenar la configuración de opciones actual
Precondiciones	- Abrir el Menú de Opciones
Postcondiciones	- Se guarda la configuración de opciones actual - La configuración actual se mantiene en futuras partidas - Se sale al Menú Principal

IDENTIFICADOR: CU-18	
Nombre	Configurar Analógico
Actores	Jugador
Objetivo	Activar o desactivar el uso del control analógico en XBOX 360
Precondiciones	- Abrir el Menú de Opciones
Postcondiciones	- Se activa o desactiva el uso del control analógico izquierdo en XBOX 360

IDENTIFICADOR: CU-19	
Nombre	Restaurar Opciones
Actores	Jugador
Objetivo	Inicializar los parámetros de configuración a sus valores por defecto
Precondiciones	- Abrir el Menú de Opciones
Postcondiciones	- Se establecen los valores a o Dificultad: Normal o Soltar Bombas: Espacio o Activar analógico: Desactivado o Sensibilidad del analógico: 5 o Volumen: 7 o Música: On

IDENTIFICADOR: CU-20	
Nombre	Colisionar con Bloque
Actores	Jugador
Objetivo	El jugador, en un desplazamiento, choca con un bloque del mapa
Precondiciones	- Haber comenzado una partida - Haber iniciado un movimiento - La posición de destino del movimiento está ocupada por un bloque
Postcondiciones	- El personaje no realiza el movimiento

IDENTIFICADOR: CU-21	
Nombre	Colisionar con Enemigo
Actores	Jugador
Objetivo	El jugador, en un desplazamiento, choca con un enemigo del mapa
Precondiciones	- Haber comenzado una partida - Haber iniciado un movimiento - La posición de destino del movimiento está ocupada por un enemigo
Postcondiciones	- El personaje pierde un corazón si tiene más de uno - El personaje pierde una vida si sólo tiene un corazón

IDENTIFICADOR: CU-22	
Nombre	Recoger Power Up
Actores	Jugador
Objetivo	El jugador recoge un Power Up del mapa y se le aplica su efecto
Precondiciones	- Haber comenzado una partida - Haber destruido un bloque que contenía un Power Up
Postcondiciones	- Se aplica el efecto correspondiente al Power Up al personaje - El Power Up desaparece del mapa

IDENTIFICADOR: CU-23	
Nombre	Alcanzar enemigo
Actores	Jugador
Objetivo	El jugador deposita una bomba y su explosión alcanza un enemigo
Precondiciones	- Haber comenzado una partida - Haber depositado una bomba - Que la bomba haya explotado
Postcondiciones	- El enemigo pierde una vida - Se aplica un cambio de comportamiento al enemigo

IDENTIFICADOR: CU-23	
Nombre	Matar enemigo
Actores	Jugador
Objetivo	El jugador deposita una bomba y su explosión alcanza un enemigo
Precondiciones	- Haber comenzado una partida - Haber depositado una bomba - Que la bomba haya explotado
Postcondiciones	- El enemigo muere y desaparece del mapa

IDENTIFICADOR: CU-24	
Nombre	Mostrar Menú
Actores	Jugador
Objetivo	Mostrar cualquiera de los menús que componen el juego
Precondiciones	- Haber arrancado el juego
Postcondiciones	- Se muestra el menú correspondiente a la entrada seleccionada

IDENTIFICADOR: CU-25	
Nombre	Mostrar Puntuaciones
Actores	Jugador
Objetivo	Mostrar las cinco puntuaciones máximas en el histórico del juego
Precondiciones	- Haber seleccionado la entrada Puntuaciones
Postcondiciones	- Se muestran las cinco puntuaciones máximas - Se muestra además el jugador que las hizo y el nivel que alcanzó

3.1.4. Diagrama de actividad del sistema

Una vez especificados los casos de uso, los cuales representan la interacción entre usuario y sistema, quedan más claras las distintas funcionalidades que el usuario puede realizar. Para que quede más claro el comportamiento del sistema se mostrará a continuación su diagrama de actividad, que recoge las transiciones y eventos.

Los diagramas de actividad describen la secuencia de las actividades en un sistema. En ellos, las transiciones entre estados se producen como consecuencia de eventos y pueden tener un procesamiento asociado. Representan los sucesos que influyen en el comportamiento y evolución del sistema, describiendo tanto su comportamiento normal (ejecución típica), como excepcional (errores o excepciones). A continuación, en la figura 54, se muestra el diagrama actividad del sistema:

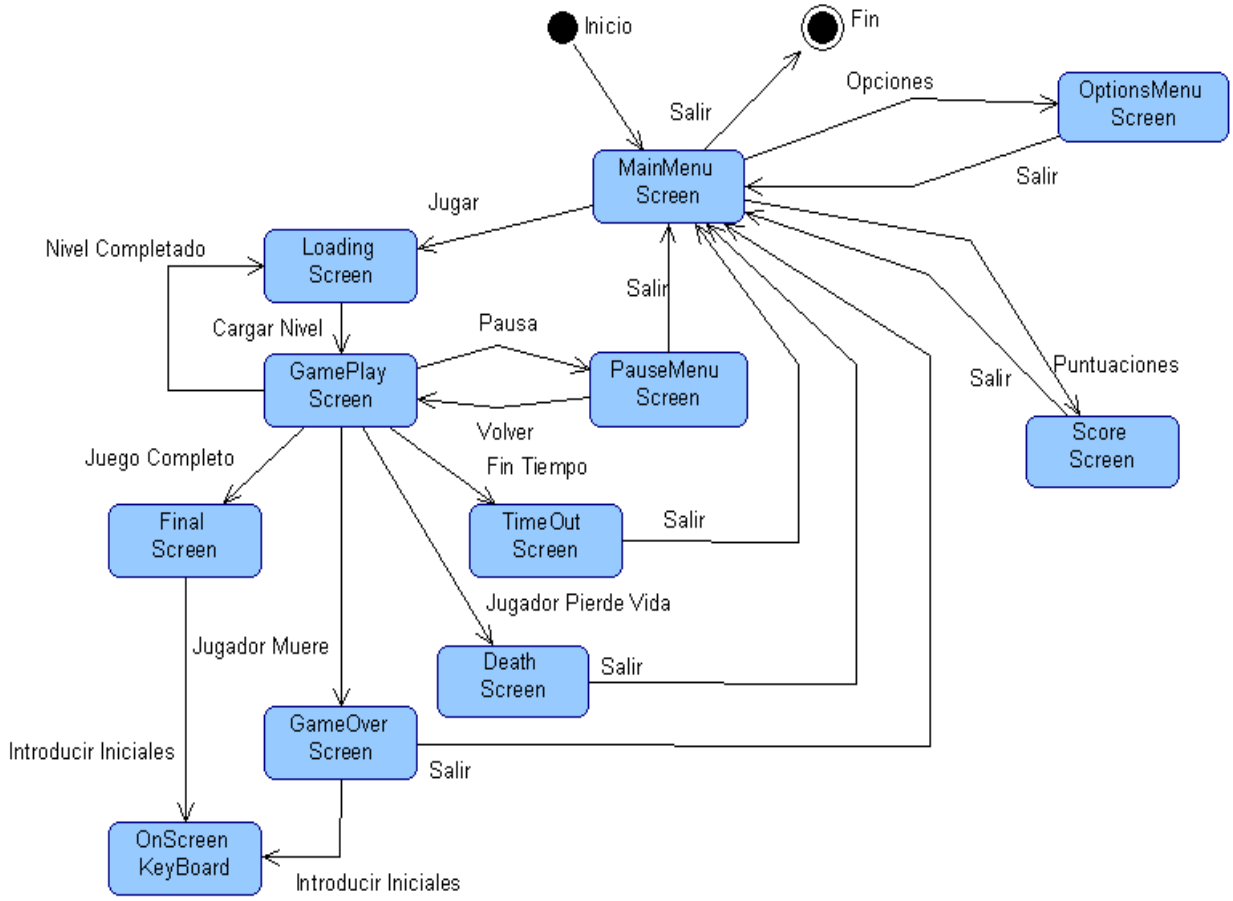

Figura 54: Diagrama de Actividad del sistema

3.1.5. Diagramas de secuencia

La especificación de los casos de uso ha permitido dar una idea clara de cómo el usuario puede utilizar el sistema a través de sus distintas funcionalidades. De la misma manera, el diagrama de actividad del sistema ha aclarado cuál es el comportamiento del sistema ante los eventos más típicos.

Los diagramas de secuencia, permiten representar cómo se comunican entre sí varios objetos para la realización de los casos de uso. Si el diagrama de casos de uso permite el modelado de la vista o escenario del sistema, el diagrama de secuencia contiene detalles de implementación del escenario, incluyendo los objetos y clases que se usan para implementar el escenario, y mensajes intercambiados entre los objetos; de manera que se puede determinar qué objetos son necesarios para

la implementación del escenario. Los diagramas de secuencia facilitarán la tarea de creación del modelo de clases en la fase de diseño.

Sería muy pesado representar el diagrama de secuencia de cada caso de uso ya que por su similitud muchos no aportarían ninguna información de interés. Por esto, se ha decidido que sólo se van a recoger en este punto algunos diagramas de secuencia que hagan referencia a funcionalidades importantes del sistema.

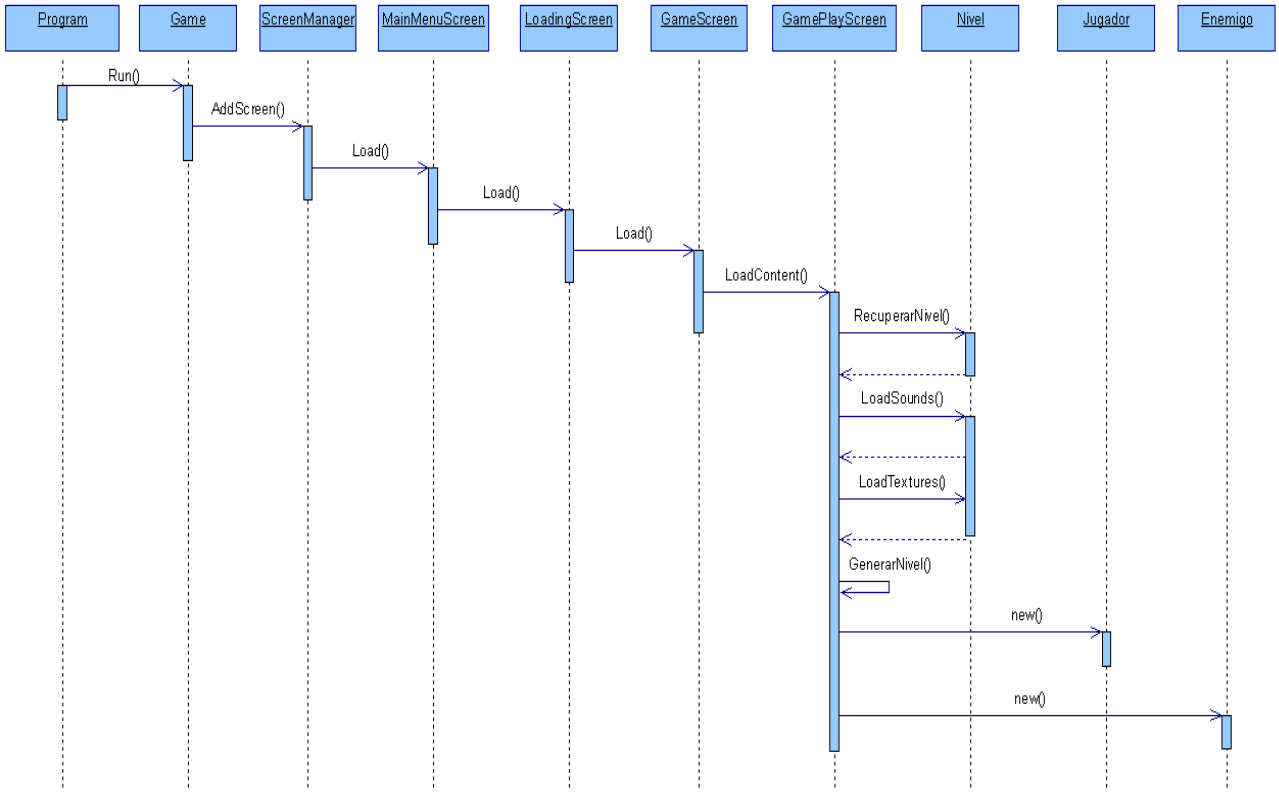

Figura 55: Diagrama de Secuencia Jugar

El diagrama de la figura 55 representa la funcionalidad Jugar, en la que el jugador lanza el juego y comienza una nueva partida. Se puede observar la interacción de la clase *ScreenManager*, que actúa como gestor de pantallas de la aplicación, cargando el Menú Principal. Al seleccionar Jugar desde dicho menú, se carga una pantalla de loading que, cuando termina el proceso de carga, lanza un *GameScreen* o pantalla de juego genérica. Esta mediante el método LoadContent lanza la pantalla de juego de *Bomberman*, que tras cargar sonidos y texturas y generar el nivel inicializa un nuevo jugador y los enemigos correspondientes.

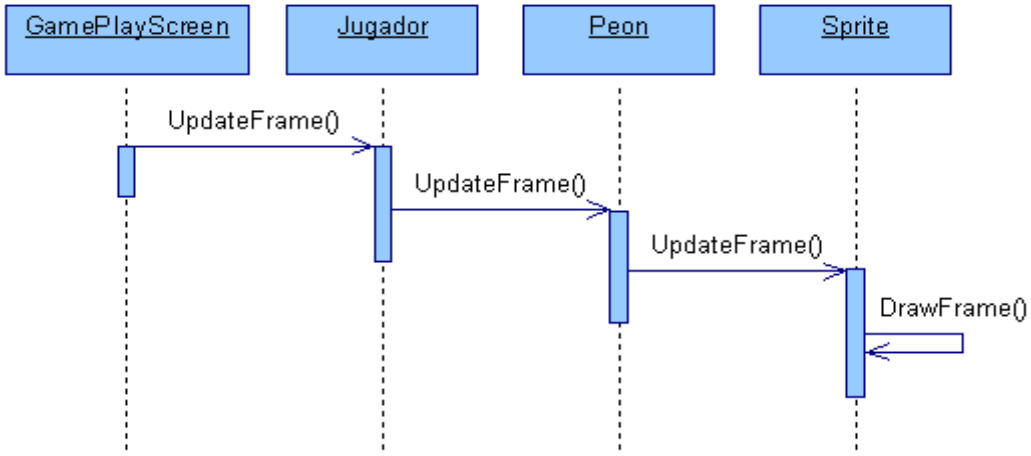

Figura 56: Diagrama de Secuencia Pintar Frame

El diagrama de secuencia de la figura 56 representa la funcionalidad Pintar Frame, que actualiza el frame de la animación después de un movimiento. La secuencia de llamadas es muy simple, tras realizar un movimiento la clase *GamePlayScreen* llama al método UpdateFrame de Jugador. Tanto el jugador como los enemigos heredan de la clase *Peon* (se puede considerar como un elemento animado del juego) que también implementa el método UpdateFrame. Por último *Peon*, al ser un elemento animado, hereda de *Sprite* que es la clase que ejecuta DrawFrame; método en el que se actualiza al siguiente frame de la animación correspondiente para que posteriormente se muestre por pantalla cuando la clase *Game* ejecute el método Draw.

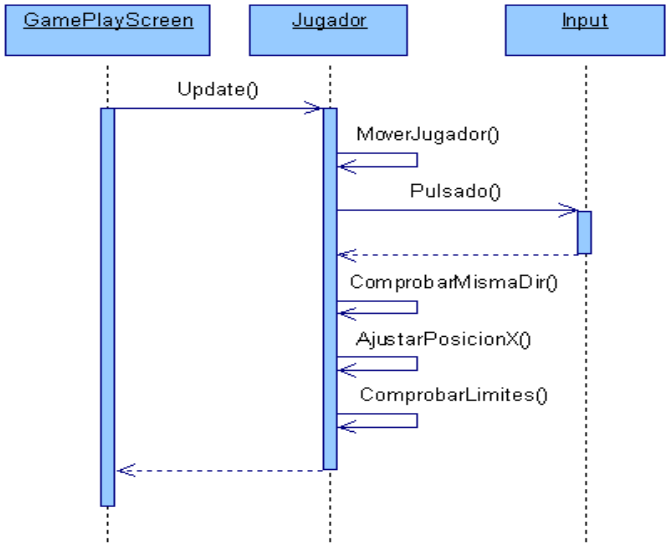

Figura 57: Diagrama de Secuencia Mover Personaje

En la figura 57 se representa el diagrama de secuencia Mover Personaje, en este caso en un desplazamiento horizontal. *GamePlayScreen* detecta que se ha producido un movimiento y hace Update de *Jugador*. La clase *Jugador* ejecuta MoverJugador, método que ejecuta Pulsado de la clase Input para saber qué tecla fue la que originó el movimiento. Con ComprobarMismaDir se detecta si el movimiento es el mismo que el anterior para saber si se debe continuar con el siguiente frame de una animación o lanzar una nueva. AjustarPosicion detecta, en el caso de producirse una colisión, si esta es mínima (se explicará más detenidamente este concepto en puntos posteriores) se desplaza al jugador aplicando un movimiento adicional. Por último será necesario comprobar si dicho movimiento adicional está dentro de los límites del tablero con ComprobarLimites. Tras esta serie de llamadas *GamePlayScreen* ejecutará el método comentado en el anterior diagrama de secuencia (UpdateFrame), actualizando así la animación correspondiente.

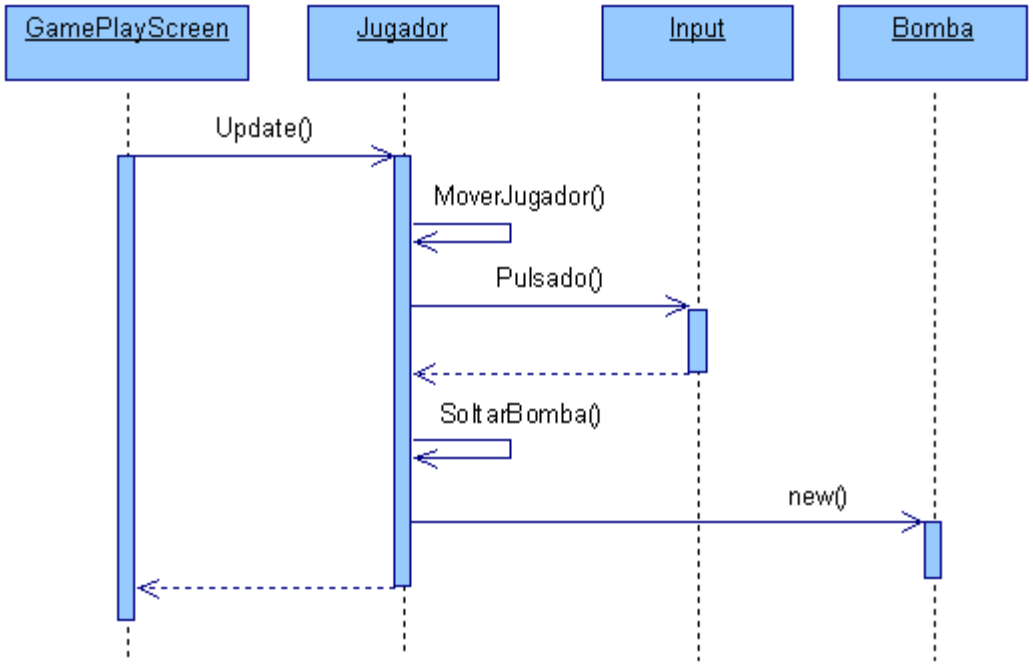

Figura 58: Diagrama de Secuencia Soltar Bomba

La figura 58 representa el diagrama de secuencia SoltarBomba. Muy similar al caso anterior salvo que no se debe detectar si el jugador lleva la misma dirección porque no se ejecutó un movimiento de desplazamiento. En este caso *Jugador* ejecutará SoltarBomba, en el cuál se crea una nueva instancia del objeto *Bomba* que pasará a la lista de bombas activas del mapa.

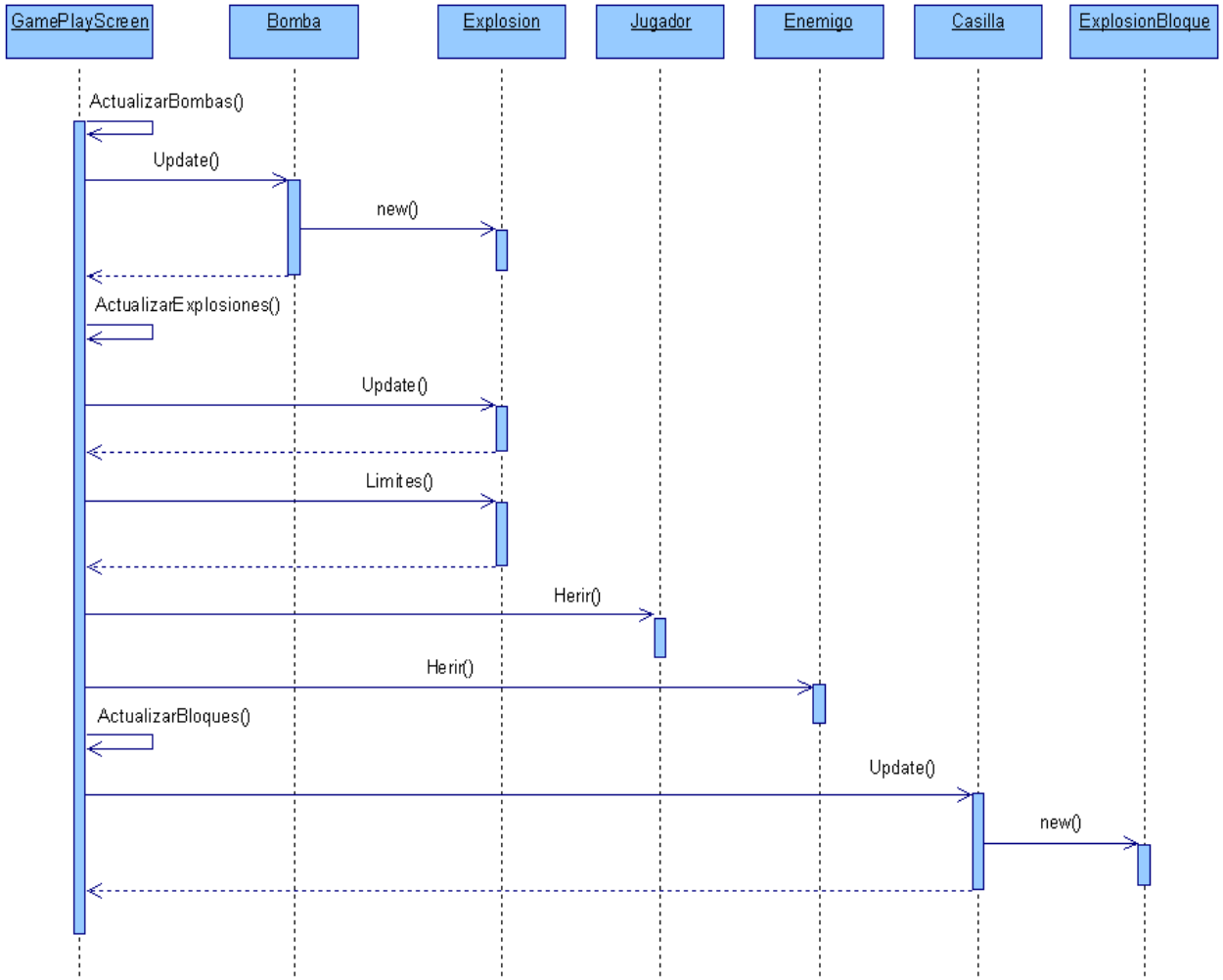

Figura 59: Diagrama de Secuencia Explosión Bomba

Por último, la figura 59, muestra un diagrama de secuencia en el que se recogen varias funcionalidades, pero la principal es Explotar Bomba. El diagrama muestra la cadena de llamadas que se produce al explotar de forma automática, una bomba depositada por el jugador en el mapa. *GamePlayScreen* ejecuta ActualizarBombas, que detecta que ha saltado el timer de explosión y elimina la bomba de la lista de bombas activas, generando después un nuevo objeto *Explosion*. En este caso la bomba explota de forma automática por lo que no hay que comprobar entrada de usuario. *GamePlayScreen* ejecuta ActualizarExplosiones y con los valores de potencia la función Limites, que marca el alcance de la explosión. Si los límites de la explosión alcanzan al jugador o a un enemigo se ejecutará el procedimiento Herir respectivo. De la misma manera si alcanza un bloque rompible se ejecuta ActualizarBloques que hará Update de la casilla que contiene el bloque y creará un nuevo objeto ExplosionBloque generando un Power Up si el bloque lo contenía.

3.2. Fase de Diseño

En la fase de diseño se debe tomar el análisis de la etapa anterior y realizar un diseño exhaustivo de los componentes, para que después en la etapa posterior de implementación esté todo tan claro como para no tener que replantear ningún aspecto. Por tanto en esta fase, hay que centrarse en obtener un diseño que se adapte a las necesidades de la aplicación y una definición exhaustiva de las clases, así como su respectivo diagrama que sirva para ver la colaboración e interacción de éstas.

La primera tarea a llevar a cabo es definir las clases a utilizar y establecer las relaciones entre éstas, que reflejen la estructura a seguir en la Fase de Implementación.

3.2.1. Diagrama de clases

El diagrama de clases describe las clases y objetos que debe tener el sistema y las diversas relaciones de carácter estático que existen entre estas. Es necesario además, para mayor claridad, señalar los atributos y operaciones más importantes de las distintas clases; así como las restricciones de visibilidad a la que se verán sujetas.

Una vez estudiado los requisitos funcionales del sistema, las funcionalidades y el comportamiento del sistema, mediante el análisis en el punto anterior, se está en disposición de comenzar a definir las diferentes clases que compondrán el modelo de diseño, de forma que cubran las necesidades y funcionalidades del software.

En esta sección del capítulo se muestra en primer lugar, el diagrama de clases, la descripción de las clases más importantes y su relación con otras clases y objetos a través de diagramas para posteriormente ir profundizando en las mismas, incluyendo los métodos y propiedades más importantes.

La figura 60 muestra el diagrama de clases del sistema.

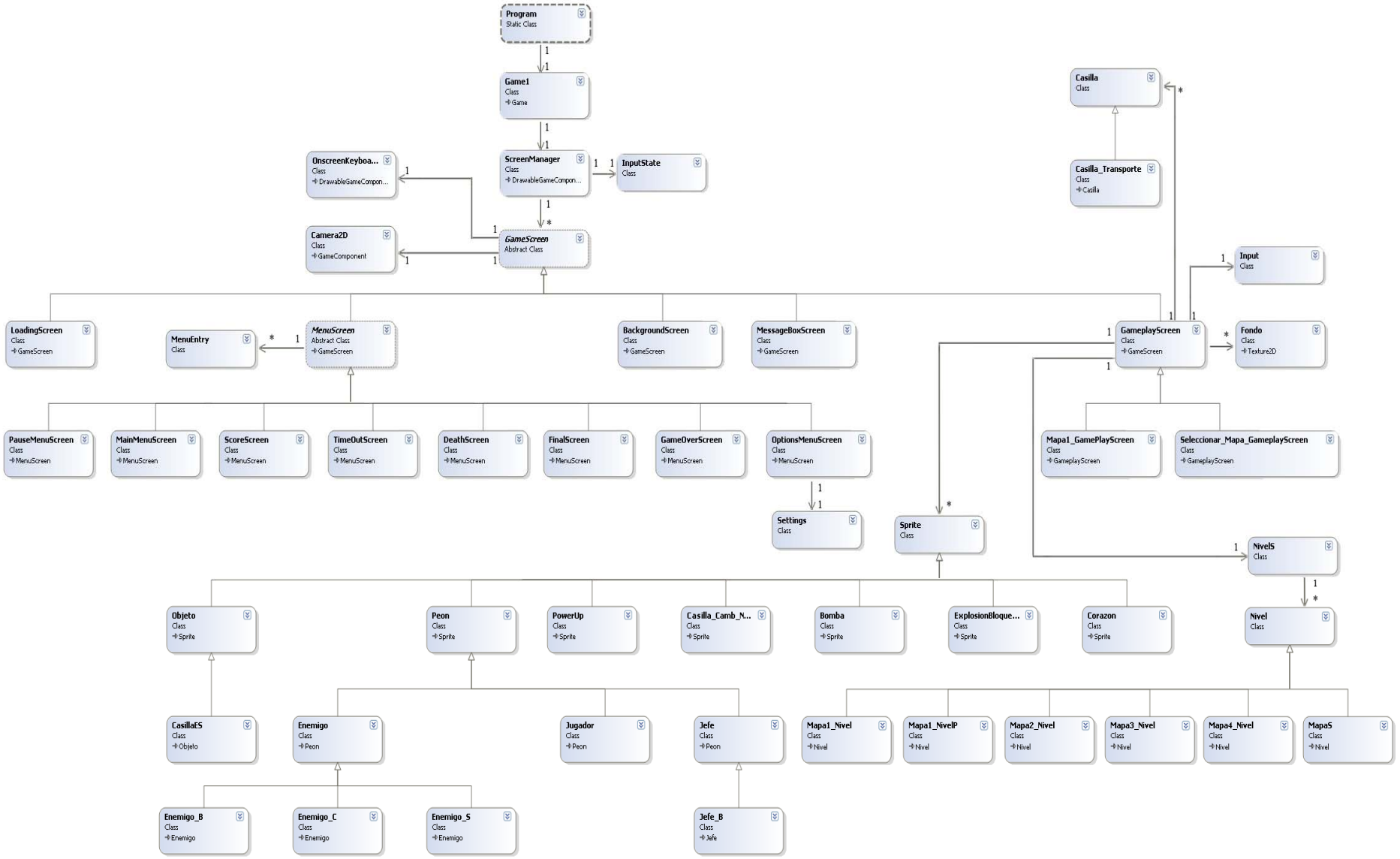

Figura 60: Diagrama de clases

3.2.2. Definición de las clases

Como se puede observar, el sistema está compuesto por más de 30 clases, lo que hace que el diagrama de clases sea algo difícil de interpretar en su conjunto. Por ello, a continuación, se explicarán las clases más relevantes con detalle, haciendo referencia a sus atributos y métodos más importantes. Se hará referencia a todas las clases, pero no se describirán en profundidad algunas por dos motivos, el primero es que existen clases redundantes en lo que al diseño se refiere (como por ejemplo las clases que representan cada tipo de pantalla, que son en esencia iguales) y explicando una quedan representadas las demás. El segundo, que durante el desarrollo del proyecto se han utilizado librerías de apoyo de *Microsoft* (como el gestor de pantallas *ScreenManager*), las cuales han sido modificadas para adecuarlas a la funcionalidad del juego. En el caso de estas últimas se explicará su funcionalidad y los métodos añadidos o modificados con respecto a la librería original.

- **Program**

  Contiene el Main del proyecto, instancia el juego (clase *Game1*) y llama a su método Run().

- **Game1**

  Clase que hereda de *Microsoft.Xna.Framework.Game* y proporciona una serie de métodos para inicializar el juego, así como el bucle principal que lo controla. Los atributos más importantes son:

  - graphics: este objeto, de tipo *GraphicsDeviceManager*, es el handler o manejador para la capa de gráficos.
  - spriteBatch: objeto de tipo *SpriteBatch* usado para dibujar texto e imágenes en 2D.

  En cuanto a los métodos más importantes:

  - Constructor: método en el que se asigna la carpeta Content, directorio raíz del proyecto donde se guardan todos los contenidos del juego, es decir, sonidos, gráficos, modelos 3D.

- Initialize: dentro de este método se inicializa cualquier cosa que no sea recursos gráficos.
- LoadContent: es el lugar indicado para cargar los recursos gráficos.
- UnloadContent: en este método se liberan los recursos cargados.
- Update: es el bucle del juego, sirve para actualizar los cálculos de los objetos en función de las posiciones y de los inputs del usuario, redibujar la pantalla y tiene una condición de salida que finaliza el juego (el usuario pulsa salir, ha muerto o ha ganado).
- Draw: en este método es donde se pintan los sprites a nivel de render de objetos por pantalla.

- **ScreenManager, InputState y GameScreen**

Para el correcto funcionamiento del juego, es necesario un componente que administre la transición entre pantallas, así como la actualización y el dibujado de cada pantalla individual. Para ello se ha tomado como base una implementación propuesta por *Microsoft* en *XNA Creators Club*[45], llamada Game State Management. Dicha implementación proporciona un sistema de pantallas que facilita la transición entre cada una de ellas, generando su lógica y contenido de manera independiente. La clase *ScreenManager* esta implementada como un componente dentro del paradigma de *XNA*. Su función es administrar la presentación, superposición y actualización de las pantallas, así como determinar cual de ellas es la pantalla activa. Además incluye una instancia de la clase *InputState* la cual permite hacer una abstracción de toda señal de entrada, ya sea teclado, mando de *XBOX 360* o *Zune*; pudiendo ser usada por cada pantalla que herede de la clase GameScreen, con la que se proporcionará la funcionalidad básica de inicialización, actualización, manejo de datos de entrada y dibujo.

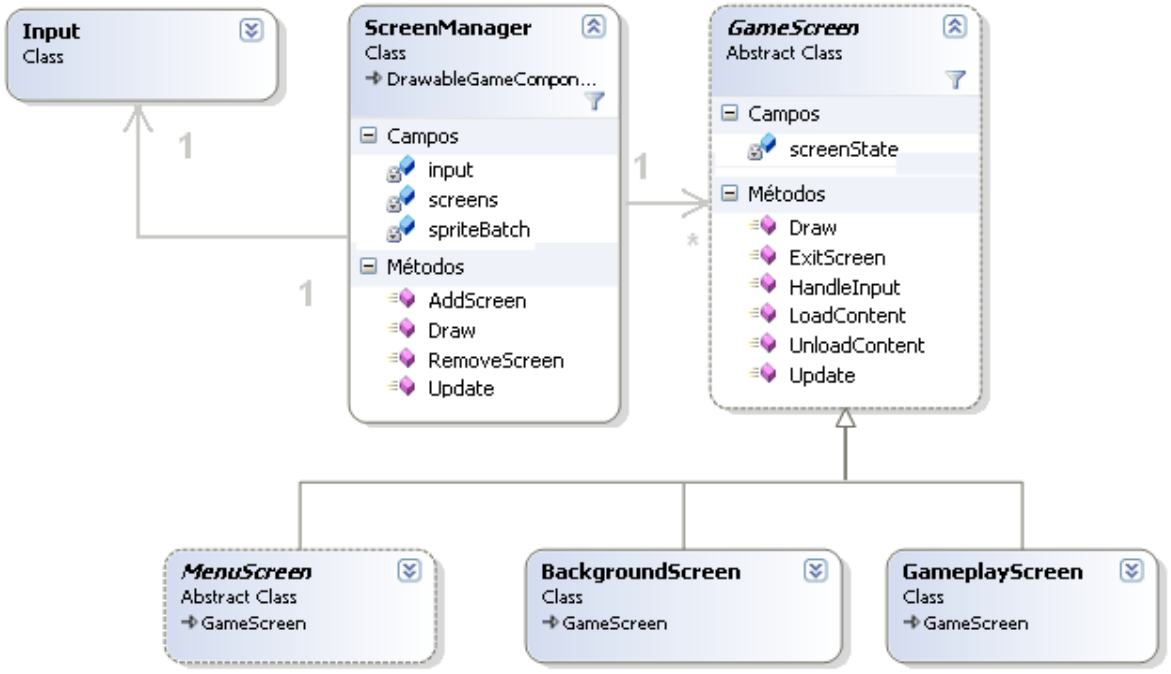

Figura 61: Detalle de las clases del paquete ScreenManager

- **Screens:**

Junto con el paquete de clases *ScreenManager*, el componente Game State Management proporciona un paquete adicional denominado Screens, con modelos básicos de pantallas que heredan de la clase *GameScreen*. Cada una de las clases del paquete Screens representa un tipo de pantalla con funcionalidades muy concretas. Los distintos modelos de pantallas son los siguientes:

- BackgroundScreen**:** esta clase permite gestionar elementos como fondo de pantalla, de manera que se puedan cargar imágenes o animaciones que sustituyan la pantalla básica de fondo.
- GameplayScreen**:** contiene toda la lógica del juego, en esta clase se manejan las texturas, los efectos de sonido y las listas con los distintos elementos que componen el juego.
- LoadingScreen: pantalla de transición que muestra mensajes de espera durante la carga de otras pantallas. Permite homogeneizar los procesos de espera en la carga de pantallas o la generación de niveles.
- MenuScreen: pantalla básica de menú que permite cargar un fondo determinado, manejar las distintas entradas así como controlar los eventos que activan cada una.

- MainMenuScreen: pantalla de menú principal, extiende de *MenuScreen* y no aporta ninguna funcionalidad destacable.
- MessageBoxScreen: aunque su aspecto en pantalla es idéntico al del resto, en realidad es una pantalla de pop up maximizada. Se utiliza para mostrar mensajes de tipo "estas seguro de…" y una acción. Permitiendo gestionar acciones como salir del juego manteniendo el estado en memoria hasta que se confirma la acción.
- PauseMenuScreen: al igual que *MainMenuScreen*, es una pantalla de menú sin más.

El paquete Screens añade además dos clases adicionales, *PlayerIndexEventsArgs* que permite gestionar la entrada en juegos con más de un jugador simultáneo, por lo que en este proyecto no se ha usado; y *MenuEntry*, que permite gestionar cadenas de texto como entradas de menú, configurando el color, posibles animaciones o el tipo de letra. Las clases del paquete son sólo esqueletos que deben completarse para mostrar los menús y las pantallas con las opciones y los elementos correspondientes.

Además de estas clases, se han desarrollado otras que heredan de la clase MenuScreen y son:

- DeathScreen: pantalla de menú que se muestra cuando el jugador pierde una vida.
- FinalScreen: pantalla final del juego, muestra la puntuación total acumulada y permite al jugador introducir sus iniciales para registra la puntuación.
- GameOverScreen: pantalla de menú que se muestra cuando el jugador pierde todas las vidas.
- OptionsMenuScreen: pantalla que permite modificar las opciones.
- ScoreScreen: pantalla que muestra la lista de puntuaciones máximas.
- TimeOutScreen: pantalla que se muestra cuando el jugador pierde una vida al acabarse el tiempo máximo para superar un nivel.

Se incluyen dos clases adicionales que extienden de *GameplayScreen*:

- Mapa1_GameplayScreen: al heredar de *GameplayScreen*, contiene la lógica del juego, se utiliza para cargar y gestionar los niveles estándar del juego (con enemigos, tiempo máximo, power up etc.)

- Seleccionar_Mapa_GameplayScreen: sirve para cargar el nivel inicial de entrenamiento en el que cambia la lógica del juego (no hay enemigos, ni tiempo, ni power up, ni el personaje puede morir etc.).

Una vez aclarado como se lleva a cabo la gestión de pantallas, es necesario profundizar un poco más en las clases *MenuScreen* y *GameplayScreen* puesto que todas las demás, como se puede observar en la figura 60, extienden de estas.

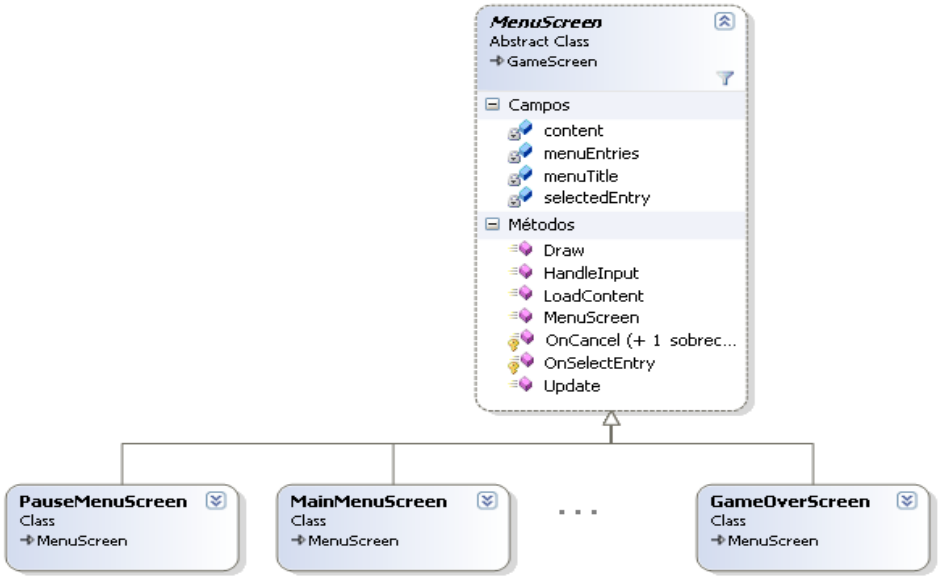

Figura 62: Detalle de la clase MenuScreen

- **MenuScreen**

Como ya se ha comentado, la clase hereda de *GameScreen* y sirve para crear pantallas de menú. Los atributos más importantes son:

- content: instancia de ContentManager, es la entidad encargada de gestionar la carga de archivos externos tales como modelos, imágenes , fuentes, etc
- menuEntries: de tipo *MenuEntry*, es una lista que almacena todas las entradas que contiene la pantalla de menú.
- selectedEntry: hace referencia a la entrada activa del menú.

En cuanto a los métodos más importantes:

- Constructor: inicializa la pantalla y asigna un tiempo mínimo de transición en la interacción con *LoadingScreen*.
- HandleInput: maneja la entrada de usuario y marca la opción correspondiente del menú.
- OnSelectedEntry: aplica la acción correspondiente cuando una entrada de menú es seleccionada.
- OnCancel: se lanza al abandonar la pantalla.
- Update: actualiza el estado de la pantalla en cada ciclo de reloj del juego
- Daw: en este método es donde se pinta.

- **GameplayScreen**

Esta es una clase con 1600 líneas de código aproximadamente. Su extensión se debe a que, como ya se ha indicado, contiene toda la lógica del juego. Una vez que se selecciona la opción "Jugar" del menú principal se instancia *GameplayScreen*, y se cargan las texturas, las animaciones, la música, los efectos sonoros, se genera el nivel correspondiente y se inicializan todos los elementos del juego, el jugador y los enemigos. La mayoría de estos elementos, como las texturas de jugador y enemigos, o los efectos sonoros, son comunes a todos los niveles por lo que no se vuelven a cargar. Los elementos que varían como la música, las texturas de los bloques o la disposición de los bloques y los enemigos en los niveles se actualizan en los métodos correspondientes de las clases *Mapa1_GameplayScreen* y *Seleccionar_Mapa_GameplayScreen*.

Describir esta clase de forma detallada sería muy complejo debido a su gran número de atributos y métodos. Para evitar ser redundantes, se agruparán por funcionalidades. Destacar por otro lado que, el diagrama detalle que se muestra a continuación, sólo contiene parte de los atributos y de los métodos de la clase pero da una idea de su funcionalidad. Los atributos más importantes de la clase son:

- Las listas: casillasNivel, con la disposición de casillas del nivel actual; listaCasillasCamb con las casillas que contenía bloques rompibles y han generado un power up o la salida;

listaEnemigos con los enemigos activos del mapa; o listaExplosiones con las explosiones generadas a partir de una bomba detonada de listaBombas.

- Jugador: el objeto que hace referencia al personaje, se maneja el número de corazones, de vidas, los puntos acumulados y el efecto de los power up.
- Objetos SoundEffect: sirven para cargar los distintos efectos de sonido.
- Objetos Song: sirve para cargar la música de fondo correspondiente a un nivel.
- Objetos Texture 2D: cargas las tiras de sprites correspondientes a las animaciones del jugador y los enemigos, además de los sprites que representan los elementos del nivel.
- Objetos Vector2: necesarios para el proceso de detección de colisiones.
- Por último elementos adicionales como timers, contadores, auxiliares necesarios para el correcto funcionamiento del juego.

En cuanto a los métodos, también se pueden agrupar por funcionalidad

- Constructor: inicializa los atributos principales, vidas, nivel actual, puntuación etc.
- De carga: es más importante es LoadContent, en él se genera la disposición del nivel correspondiente y se completa mediante llamadas a LoadTextures, que carga los sprites; o LoadSounds, que carga música y efectos de sonido.
- De actualización: en cada ciclo de reloj del juego se actualizan los distintos elementos. El método Update realiza llamadas a ActualizarEnemigos, ActualizarExplosiones o ActualizarBombas; que como su nombre indica actualizan las listas de enemigos, explosiones y bombas activas respectivamente. Estos métodos mantienen la lógica del juego, borran una bomba activa cuando explota y generan un objeto explosión, que puede alcanzar a un enemigo o al jugador.
- Draw: pinta los elementos en pantalla en su estado actual tras haber sido actualizados.
- De control: por último los métodos de control, como HandleInput que maneja la entrada de usuario o UpdateSpiffyTimer que maneja los temporizadores.

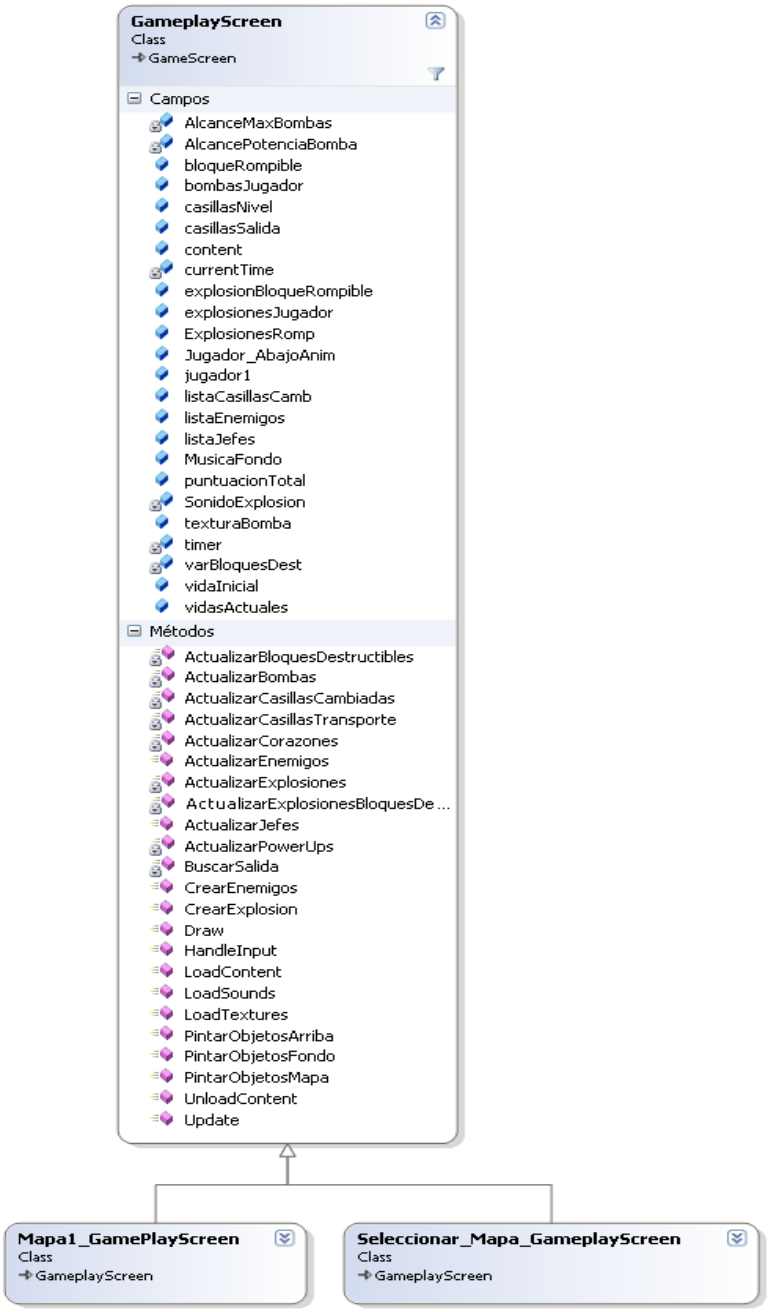

Figura 63: Detalle de la clase GameplayScreen

Una vez detallada la clase *GameplayScreen*, que soporta toda la lógica, se pasa a los elementos que componen el juego. En este sentido las clases más importantes son *Sprite*, *Casilla* y *Nivel*.

- **Sprite**:

Esta clase representa cualquier elemento que se pueda pintar en pantalla. Al ser una clase básica en cualquier videojuego existen muchas implementaciones propuestas, como por ejemplo la que proporciona *Microsoft* a través de *XNA Creators Club*. Sin embargo en este proyecto se ha optado por desarrollar la clase desde cero. Los atributos más importantes de la clase hacen referencia a propiedades necesarias para la gestión de dibujado de un objeto en pantalla y son:

- posicion: de tipo Vector2 almacena la posición del sprite y es esencial para la detección de colisiones.
- textura: de tipo Texture 2D, almacena la imagen que representa el objeto a pintar.
- Centro: de tipo Vector2, define el centro del sprite a partir del rectángulo que se puede obtener rodeando la textura.
- MargenX, MargenY: sirven para no tener que evaluar la textura completa en la detección de colisiones.
- Frame y contFrames: almacenan el frame actual que se está pintando y el número de frames totales que contiene la textura correspondiente.
- Escala: representa la escala de un objeto con respecto a los demás, permite aumentar o disminuir el tamaño de un sprite en pantalla.
- Rotación: permite rotar una textura de forma que la animación resultante produce un giro.

En cuanto a los métodos, los más importantes son:

- Limites: genera un rectángulo calculando el tamaño de un sprite individual en una tira de sprites, de manera que se pueda pintar cada elemento de la tira por separado. Además este rectángulo se utiliza para la evaluación de colisiones.
- LimitesFis: a partir de los márgenes se genera un nuevo rectángulo que mejora el proceso de detección de colisiones.
- UpdateFrame: maneja la tira de sprites desplazándose según los movimientos. Permite pintar el sprite correspondiente a cada movimiento de la animación.
- DrawFrame: pinta el sprite correspondiente.

De la clase *Sprite*, extienden varias clases: *Objeto*, *Peon*, *Power Up*, *Casilla_Cambio_Nivel*, *Bomba*, *Explosion_Bloque_Rompible* y *Corazón*.

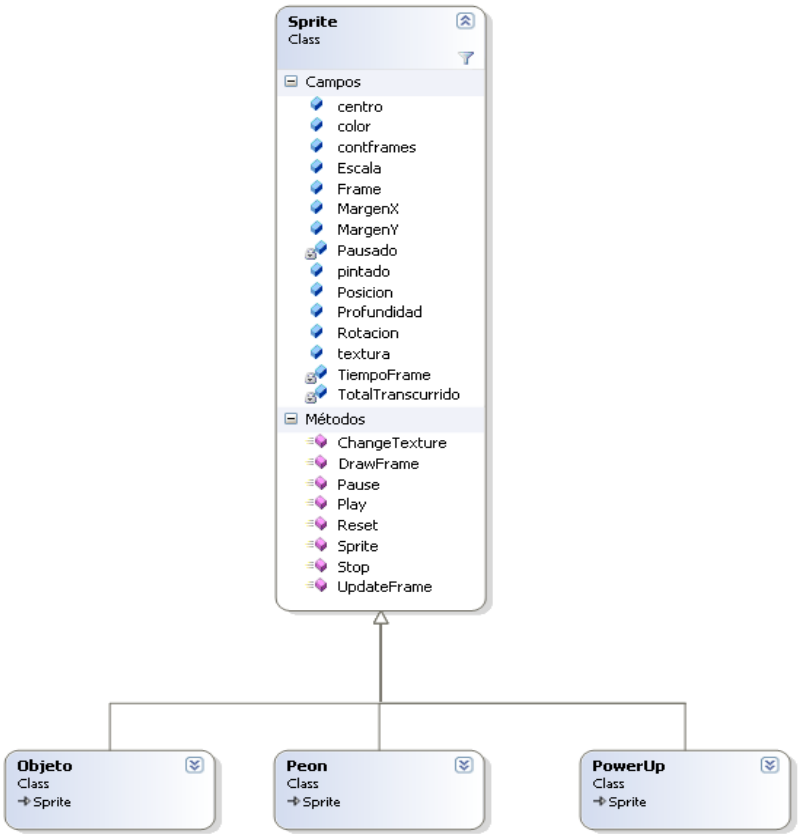

Figura 64: Detalle de la clase Sprite

- **Objeto**:

Es una clase sencilla que representa pintar objetos temporales en el escenario. De ella extiende *CasillaES*, que sirve para pintar el rastro temporal que deja un *Enemigo_S* al ser alcanzado por una bomba.

- **Power Up, Casilla_Cambio_Nivel, Bomba, Explosion_BloqueRompible, Corazon**:

Extienden de *Sprite*, implementando su propio Update con animaciones diferentes para cada uno.

- **Peon**:

Es una clase muy importante, puesto que representa los elementos con animación. De ella extienden *Enemigo*, *Jugador* y *Jefe*. Este diseño puede parecer complejo al heredar los enemigos y el jugador de la misma clase, pero en realidad soluciona muchos problemas. La idea es que todos los personajes del juego, ya sean el jugador o los enemigos, tengas las mismas características básicas; acotando luego cada uno según su condición, pero permitiendo de forma sencilla en un futuro añadir nuevos enemigos o modos de juego sin grandes modificaciones. También facilita tareas como la detección de colisiones. Se hará más hincapié en este tema en el punto de implementación. Los atributos más importantes, se agrupan por funcionalidad y son:

- Atributos relacionados con las bombas: OrigenBomba indica la casilla en la que está depositada y desde la que generar la explosión, VelocidadBomba que representa la velocidad con la que se extiende la explosión y ColorBomba que sirve para colorear las bombas en el caso de haber varios jugadores.
- Atributos de los movimientos: DireccionUltimoMovimiento y MismaDireccion, que sirven para controlar que sprite se debe cargar de una tira y, en el caso de los enemigos, para no repetir un mismo movimiento en el caso de estar bloqueado.
- Atributos propios de la lógica del juego: como PotenciaBomba que indica el alcance de la explosión, estaHerido que indica si ha sido alcanzado por una bomba, Vidas que almacena el número de vidas restante o tieneBombas que indica si el personaje puede o no soltar bombas.

En cuanto a los métodos, los más importantes son:

- Herir: detecta si el personaje ha sido alcanzado por una explosión, actualizando el estado a herido o muerto según el caso.
- DepositarBomba: añade una bomba a la lista de bombas activas del personaje en el mapa, posicionando la bomba en el centro de la casilla en la que se encuentra el jugador.
- Update: actualiza el estado en cada ciclo de reloj además de controlar el timer de explosión de las bombas del jugador.

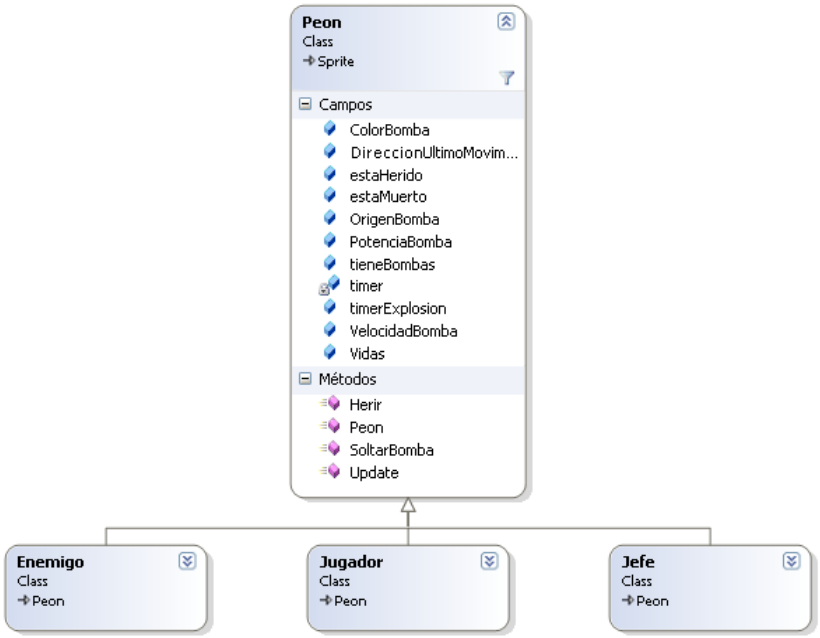

Figura 65: Detalle de la clase Peon

Las clases *Enemigo* y *Jefe* son muy similares, sus diferencias se basan más en la implementación que en el propio diseño puesto que, como se comentará más tarde en el apartado de implementación, la clase *Jefe* fue la primera versión de la clase *Enemigo* que se diseñó. Por tanto se comentará la clase *Enemigo*, dado que las diferencias son mínimas.

- **Enemigo**

La clase *Enemigo* contiene todos los atributos necesarios para que el comportamiento de los enemigos sea correcto, de manera que generen caminos, huyan de las bombas o persigan al personaje sin entrar en bucles y sin realizar movimientos incoherentes. Los atributos más importantes son, agrupados por funcionalidad:

  - Atributos de movimiento: DireccionesDisponibles almacena las direcciones a las que el enemigo puede moverse, Velocidad almacena la velocidad del movimiento, distanciaRecorrida representa la distancia recorrida en el último movimiento realizado y casillasAlrededor almacena un número de casillas que representan el radar o alcance del enemigo, siendo en función del contenido de esas casillas con el que se determina si el enemigo actúa de una manera u otra.

- Atributos de comportamiento: se detallarán en la parte de implementación. timerEspera, limiteTimerEspera y timerEsperaRandNum controlan las acciones del enemigo, marcando el tiempo máximo en el que se genera un nuevo camino o se cambia de estrategia.
- Atributos adicionales: relacionados con los estados y los movimientos y que completan la implementación.

Los métodos más importantes son:

- Update: actualiza el estado del enemigo determinando si está vivo, herido o muerto. Gestiona los temporizadores, comprobando si se debe generar un nuevo camino, se debe huir de una bomba o se debe cambiar de dirección.
- ObtenerDireccionRandom: genera un nuevo camino si el enemigo no ha detectado al jugador en el radar y no tiene una casilla destino.
- ComprobarCasillasAlrededor: evalúa las casillas del radar, determinando las acciones del enemigo.
- ComprobarCasillasDisponibles: comprueba si las casillas de un camino está ocupadas, evitando desplazamientos que acaben en un camino bloqueado.
- Movimiento: desplaza al enemigo hasta una casilla destino en concreto.

De la clase Enemigo heredan los tres tipos de enemigos incluidos en el juego: *Enemigo_B*, *Enemigo_C* y *Enemigo_S*. Las clases sobrescriben el valor de algunos atributos y la implementación del método Update para reflejar el comportamiento de cada enemigo.

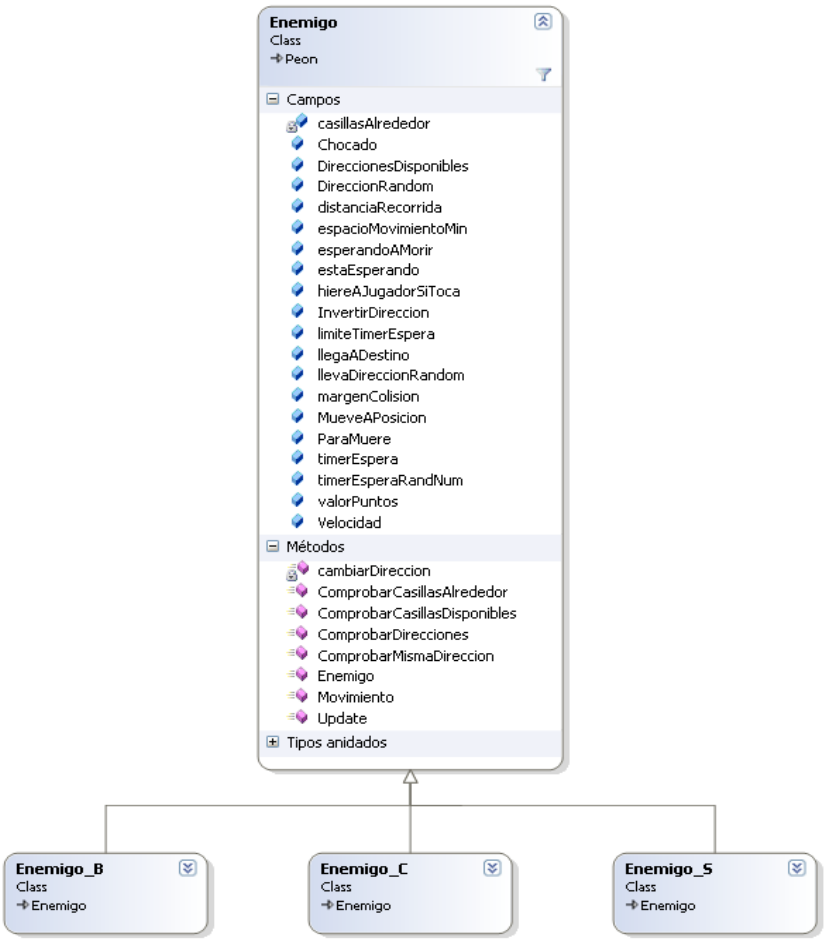

Figura 66: Detalle de la clase Enemigo

- **Jugador**:

Representa al personaje del juego que maneja el usuario y tanto los atributos, como los métodos que añade la clase, están relacionados de alguna manera con la interacción del usuario. Los atributos principales son:

- ajustarJugadorX, ajustarJugadorY: corrige la posición del jugador cuando la colisión con un bloque es menor que un margen determinado.
- casillaColisionaX, casillaColisionY: detecta la colisión en la dirección correspondiente.
- Velocidad: representa la velocidad de desplazamiento del jugador.
- estaEnCasillaEs: para reducir la velocidad de desplazamiento si el jugador está en una casilla con rastro del Enemigo_S.

- PosicionJugador: almacena la posición actual del jugador en el mapa.

En cuanto a los métodos:

- AjustarPosX, AjustarPosY: corrigen la posición del jugador en un desplazamiento en el que la colisión con un bloque es menor que un margen determinado. Se detallará en el apartado de implementación.
- ComprobarLimites: comprueba que el desplazamiento y el destino asociados a un movimiento son válidos.
- SoltarBomba: añade una bomba a la lista de bombas activas del jugador.
- DetonarBombas: detona una bomba si el jugador tiene la capacidad de hacerlo.
- ComprobarMismaDireccion: evalua el movimiento y comprueba si debe mostrarse un nuevo frame del mismo movimiento o cargar una nueva animación.
- MoverJugador: desplaza al jugador en función de la entrada de usuario.
- Update: actualiza el estado del jugador, vivo, muerto, herido etc.

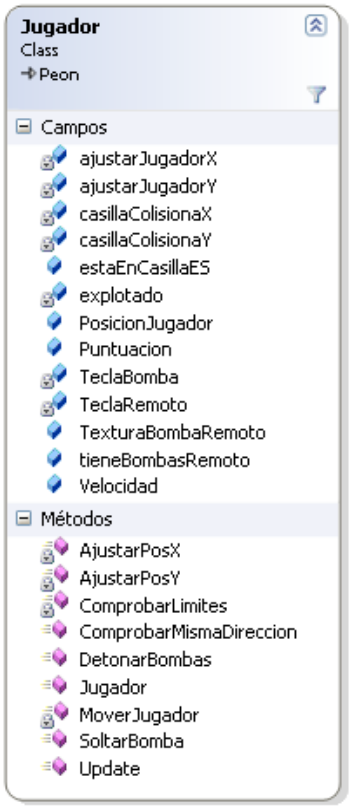

Figura 67: Detalle de la clase Jugador

Una vez analizada la clase *Sprite*, la especificación de las clases continúa *Casilla* y *Nivel*.

- **Casilla**:

La clase casilla es muy importante en el diseño puesto que es el elemento principal del que se componen los niveles del juego. Los atributos de la clase casilla permiten representar todos los estados y situaciones posibles. Los más importantes son:

- bloqueaJugador: si la casilla bloquea la posición de destino de un movimiento del jugador.
- haExplotado: permite gestionar el estado de la casilla entre la detonación de una bomba y la creación de la explosión.
- Margen: representa el margen mínimo que debe tener una colisión para que se corrija la posición del jugador.
- Muerto: permite gestionar el estado de la casilla durante el proceso de muerte de un personaje.
- Origen: de tipo Vector2, almacena las coordenadas reales de origen de la casilla en la pantalla.
- Posicion: representa la posición de la casilla en el mapa del nivel.
- Textura: la textura que se muestra en la casilla.
- tieneBomba, tienePowerUp, tieneSalida: si la casilla contiene una bomba, un Power Up en ese ciclo o contiene la salida.

Respecto a los métodos:

- Constructor: el constructor inicializa la casilla según el tipo indicado en la matriz que representa el mapa asignando valores a los diversos atributos de estado.
- Update: actualiza el estado de la casilla en función de las acciones ocurridas.
- Draw: pinta el contenido de la casilla en función del estado.

De Casilla hereda la clase *Casilla_Transporte*. Esta clase representa las casillas que permiten saltar de un nivel a otro, en este proyecto la casilla de salida. Su diseño da más posibilidades de implementación que se comentarán más adelante.

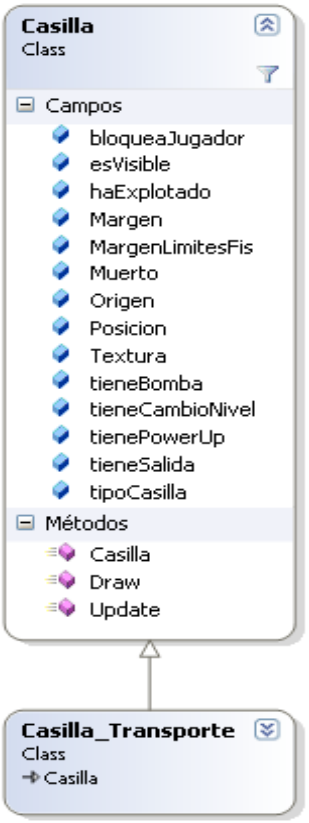

Figura 68: Detalle de la clase Casilla

- **Nivel**

En cuanto a *Nivel*, es una clase bastante sencilla que representa cada nivel del juego a nivel lógico. El nivel queda definido con una matriz de dimensiones NxM, en cuyas posiciones están almacenados los códigos correspondientes a las casillas de un determinado nivel. Los atributos más importantes son:

  - Alto, Ancho: representan el alto y el ancho en casillas del nivel. En este proyecto todos los mapas de nivel son de 11 casilla de alto y 17 de ancho si sólo tenemos en cuenta las casillas andables, 13x19 en el casi de contar las que hacen de muros frontera.

- TrazadoNivel, matriz de dimensiones Alto x Ancho que contiene los códigos de las casillas que forman el nivel.
- ConjuntoCasillas: nombre que sirve para cargar el paquete de texturas correspondiente al mapa.
- ColoFondo: color del fondo del nivel.
- noEnemigos, noBloques, noSalida, noPowerUp: indican la presencia en el nivel de enemigos, bloques rompibles, salida y power ups respectivamente.
- mensajeBienvenida, mensajeSalida: mensajes que se muestran al cargar y al superar el nivel respectivamente.
- nivelJefe: indica la presencia de un jefe final en el nivel.

En cuanto a los métodos, son get y set de los distintos atributos de la clase.

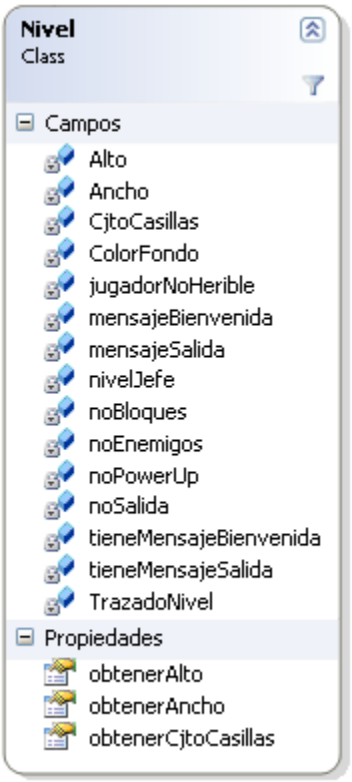

Figura 69: Detalle de la clase Nivel

Las clases ya comentadas son las más importantes en la ejecución del juego, pero existen además una serie de clases complementarias que se comentan a continuación.

- **Camera2D**

Implementación básica de una cámara 2D que proporciona *Microsoft* a través de *XNA Creators Club*. Si el juego se ejecuta en *Zune*, el sistema lo detecta y carga la cámara pasando a una visión reducida que sigue los movimientos del jugador.

- **Explosion**

Representa las explosiones. Muy similar a *Sprite*, no hereda de esta clase porque las explosiones son dinámicas, es decir, su tamaño varía durante el juego y no cumple con las restricciones de límite de la clase *Sprite*.

- **Fondo**

Genera un fondo animado que sustituye al fondo básico de un color determinado. El código genera punto en movimiento simulando estrellas y es una versión muy básica de un código del problema de los N-Cuerpos.

- **Input**

Gestiona la entrada de usuario de los dispositivos teclado, *Zune* y mando de *XBOX 360* en los movimientos del jugador.

- **NivelS**

Carga los niveles correspondientes a cada mapa.

- **OnscreenKeyboard**

Emula el teclado virtual de *XBOX 360*. A partir de modificaciones del código que proporciona *Microsoft* en *XNA Creators Club* es posible leer cadenas de teclado. Permite el usuario introducir sus iniciales para registrar una puntuación máxima.

- **Settings**

Gestiona los ficheros en los que se guardan las configuraciones y las puntuaciones máximas. Sus métodos son de lectura y escritura de ficheros XML.

3.3. Fase de Implementación

En este punto se mostrará cómo se va a utilizar el análisis y diseño llevados a cabo en los apartados 3.1 y 3.2 respectivamente, implementando las clases propuestas y teniendo en cuenta que la aplicación refleje las funciones presentadas en los Casos de Uso. Por otro lado, es necesario destacar que la finalidad del punto no es la de comentar el código de las clases, si no profundizar en todos los procesos que componen la fase de desarrollo. Por tanto, se especificará que sistema se ha utilizado para la detección de colisiones, tanto del jugador como de los enemigos; cómo se han diseñado e implementando los enemigos y cuál es su comportamiento, se detallarán los sprites que se han usado para las animaciones del jugador, los enemigos y los elementos de los mapas, se especificará el diseño de los niveles y se explicará como el diseño permitiría añadir nuevas funcionalidades de forma sencilla. En definitiva cualquier aspecto relacionado con el desarrollo que se considere importante

En cuanto a la organización del capítulo, será paralela al desarrollo real del juego, de forma que se comenzará por comentar aspectos de la implementación del jugador, posteriormente del entorno y los niveles, después de los enemigos y por último de otros elementos adicionales de relevancia.

3.3.1. Jugador

Para la implementación del juego se tomó como punto de partida el jugador, tendiendo en mente tres objetivos principales:

1. Conseguir un desplazamiento fluido del jugador en pantalla.
2. Animar los movimientos de forma correcta.
3. Una vez implementados los movimientos, marcar sus límites y detectar las colisiones del jugador con los distintos elementos del mapa.

- **Sprites**

La funcionalidad completa del jugador se consigue gracias a tres clases, *Sprite*, *Peon* y *Jugador*. El primer paso es mostrar al jugador en la pantalla y se realiza al través de los métodos de la clase *Sprite*. Para conseguir mostrar un elemento en pantalla es necesario cargar la imagen correspondiente usando el método Load en un objeto de tipo Texture2D.

La gestión de contenido gráfico y sonoro de un proyecto en *XNA*, se realiza a través del *Content Pipeline* [174] (es un *API* que permite a los desarrolladores y diseñadores incorporar contenidos multimedia en los proyectos creados con *XNA* framework). El *Content Pipeline* o CPL proporciona el importador (es el encargado de obtener los datos y normalizarlos) y el procesador (recibe los datos que el importador ha generado y crea un objeto para su uso en tiempo de ejecución). En este caso el importador será el objeto de tipo ContentManager y los procesadores dependen de los datos que se quieran cargar. Existen varios tipos en función del contenido:

- Model: sirven para cargar objetos 3D con texturas incorporadas.
- Texture2D: permiten cargar tiras de sprites en animaciones 2D y texturas para modelos 3D que no las llevan incorporadas.
- Effect: sirve para gestionar efectos relacionados con los materiales de modelos 3D, como reflejo, brillo, transparencia etc.

La tabla de código 1, muestra cómo cargar una tira de sprites en un objeto Texture2D.

```
Public ContentManager content;
content = new ContentManager(ScreenManager.Game.Services, "Content");
public Texture2D Jugador_AbajoAnim;
Jugador_AbajoAnim =
content.Load<Texture2D>("AnimacionesJugador\\Jugador_AbajoAnim");
```

Código 1: Carga de una tira de sprites

Serán necesarios tantos objetos Texture2D como animaciones se quieran cargar. En este caso, para un único jugador, serán necesarias las siguientes animaciones:

- Animaciones de desplazamiento correspondientes a cada dirección en la que el jugador se puede mover: arriba, abajo, izquierda y derecha.
- Una animación de muerte.
- Una que represente al jugador parado en cada una de las cuatro direcciones en que se puede desplazar.

Los sprites utilizados para el jugador son los siguientes:

Figura 70: Tira de sprites Jugador_AbajoAnim

Figura 71: Tira de sprites Jugador_ArribaAnim

Figura 72: Tira de sprites Jugador_DerechaAnim/ Jugador_IzquierdaAnim

Figura 73: Tira de sprites Jugador_MuertoAnim

Figura 74: Sprite Jugador_ParadoArriba

Figura 75: Sprite Jugador_ParadoAbajo

Figura 76: Sprite Jugador_ParadoDerecha/ Jugador_ParadoIzquierda

Los sprites Jugador_IzquierdaAnim y Jugador_DerechaAnim son iguales aplicando una transformación horizontal. La misma acción es aplicable a los sprites Jugador_ParadoDerecha y Jugador_ParadoIzquierda. Hay que destacar que, en un principio se pensó diseñar los sprites desde cero llegando incluso a hacer un diseño de los movimientos del jugador. Más tarde se optó por utilizar los sprites del juego *Bomberman Online* [175] de la compañía *Hudson*. *Hudson* permite utilizar los sprites de *Bomberman Online*, proporcionando herramientas de exportación a través de su foro, siempre y cuando no se utilicen en un juego comercial. Por tanto, si este proyecto se publicara en algún momento, sería necesario cambiar los sprites.

- **Entrada**

Para manejar la entrada de usuario se utilizan los métodos de la clase *Input*. *XNA* no proporciona una gestión demasiado eficiente de la entrada de usuario por lo que se deben mapear todas las teclas del teclado que se quieran usar como controles. Esto significa que en todo momento es necesario preguntar qué tecla en concreto se está pulsando para "convertirla" en una acción del juego. Par esto se define un enumerado con los controles permitidos en el juego y posteriormente se definen los tipos de entrada que se quieren manejar. Posteriormente, el método Pulsado, detecta si se ha pulsado alguna tecla, controlando que teclas de cada uno de los dispositivos están pulsadas y convirtiéndolas a un elemento del enumerado; lo que devuelve una única acción sea cual sea el dispositivo de entrada. En la tabla de código 2, se incluye la conversión de la acción izquierda.

```
Public enum TeclaInput{ Izquierda, Derecha, Arriba, Abajo, A, B, Atras, Otros }
KeyboardState[] kbs;
GamePadState[] gps;
ZunePadState[] zps;
public bool Pulsado(TeclaInput Tecla, PlayerIndex index){
    switch (Tecla){
         case TeclaInput.Izquierda:
              if (kbs.IsKeyDown(Keys.Left)||gps.IsButtonDown(Buttons.DPadDown)||...
                   pulsado = true;
              break;
         case TeclaInput.Derecha:
```

Código 2: Detección de la entrada de usuario

- **Movimientos**

Una vez cargadas las animaciones se deben implementar los movimientos. Para realizar correctamente el movimiento hay que controlar dos factores, el primero la entrada de usuario y el segundo el desplazamiento del jugador en respuesta a esa entrada.

En el caso del jugador, el movimiento se controla en el método MoverJugador de la clase *Jugador*. El personaje debe responder a tres entradas distintas: desplazamiento, que se realiza con las teclas de dirección; soltar una bomba que se realiza con las teclas Enter o Espacio y detonar una bomba que se realiza con la tecla no asignada a soltar bomba del par Enter y Espacio.

Después de detectar si ha habido entrada de usuario, es necesario mapearla y convertirla en una acción del jugador. El método MoverJugador de la clase *Jugador* comprueba a qué elemento del enumerado pertenece la entrada que ha activado el método Pulsado y lo convierte en la acción correspondiente: desplazar al jugador, soltar una bomba o detonarla; ejecutando en cada caso el método correspondiente. En el ejemplo se muestra el mapeo de la acción cuando se pulsa Izquierda:

```
void MoverJugador(GameTime gameTime, Input GameInput, ...){
    if (GameInput.Pulsado(TeclaInput.Izquierda, index)){
        ComprobarMismaDireccion("Izquierda");
        AjustarPosY(-16, -6, TeclaInput.Izquierda, anchoCasilla, ...);
        if (ajustarJugadorY == false)
            Posicion.X += -Velocidad;
        if (GameInput.Pulsado(TeclaBomba, index))
            SoltarBomba(bombas, casillasNivel);
```

Código 3: Mapeo y conversión de la entrada de usuario

- **Animaciones**

El método MoverJugador desplaza al sprite del personaje en la dirección correcta en pantalla o ejecuta las acciones relacionadas con las bombas, pero no controla las animaciones. Es necesario combinar la entrada de usuario y el manejo de sprites para realizar las animaciones de forma correcta. Una vez detectada la entrada se debe convertir además de en movimiento, en uno de los elementos que compone la animación. Para hacerlo de forma correcta se utilizan los métodos ComprobarMismaDireccion de la clase *Jugador* y UpdateFrame y DrawFrame de la clase *Sprite*.

ComprobarMismaDirección permite controlar si la última entrada de usuario es igual a la inmediatamente anterior. Si es así, la acción que se debe realizar es la misma, y en el caso de ser de desplazamiento se debe continuar una acción de animación ya iniciada. UpdateFrame actualiza el elemento de la tira de sprites que se debe pintar y DrawFrame se encarga de pintarlo en pantalla.

```
Public void ComprobarMismaDireccion(string Direccion){
    if (DireccionUltimoMovimiento == Direccion){
        MismaDireccion = true;
        DireccionUltimoMovimiento = Direccion;
    }
    else if (DireccionUltimoMovimiento != Direccion){
        MismaDireccion = false;
        DireccionUltimoMovimiento = Direccion;
 }
```

Código 4: Comprobación de acción continuada

```
Public void ComprobarMismaDireccion(string Direccion){
    if (ComprobarCasillasDisponibles(auxc, Direccion)){
        llevaDireccionRandom = true;
        MueveAPosicion = auxc.Posicion;
        MismaDireccion = true;}
}
```

Código 5: Comprobación de acción continuada

```
Public void UpdateFrame(float transcurrido){
    if (Pausado) return;
    TotalTranscurrido += transcurrido;
    if (TotalTranscurrido > TiempoFrame){
        Frame++;
        Frame = Frame % contframes;
        TotalTranscurrido -= TiempoFrame;
    }
```

Código 6: Actualización del frame de una animación

El cálculo del frame correspondiente a la animación se realiza de forma sencilla dividiendo el ancho total de la tira por el número de sprites que contiene y actualizando un contador de movimiento, que se reinicia al llegar al último para convertir la tira en un bucle. Cuando se sabe que frame se debe mostrar, se genera un rectángulo que permite pintar sólo ese frame en una tira completa.

```
Public virtual void DrawFrame(SpriteBatch batch, int frame){
    int AnchoFrame = textura.Width / contframes;
    Rectangle rectorigen = new Rectangle(AnchoFrame * frame, 0, textura.Height);
    batch.Draw(textura, Posicion, rectorigen, color, Rotacion, centro, Escala,
SpriteEffects.None, Profundidad);
    pintado = true;
}
```

Código 7: Pintado del frame de una animación

- **Colisiones**

La detección de colisiones es uno de los aspectos más importantes de cualquier juego. Un mal sistema de detección de colisiones puede acabar con la mecánica de un juego, especialmente en juegos en los que la velocidad de movimientos es un factor determinante. Hay diferentes formas de manejar la detección de colisiones y la elección del método correcto depende de muchos factores. Por ejemplo, si el juego es 2D o 3D o si es un juego de carreras, aventuras o de pelea.

Para el caso específico de los juegos en 2D, aunque hay diferencias entre unos estilos de juego y otros, existen tres métodos fundamentales para detección de colisiones: detección de colisión con figuras geométricas, detección por celdas y detección por píxel. Los tres métodos son aplicables en el caso de este proyecto por lo que se debe justificar el por qué de la elección de uno en concreto.

En primer lugar se debe destacar que tipo de detección de colisiones se quiere realizar. En el caso de este proyecto se ha optado por una detección de colisiones de tipo “físico” en lo que respecta al jugador, entendiendo por “físico” el proceso de detección que utiliza los sprites de los elementos en pantalla. El método “lógico” evalúa la posición de los elementos del juego como si estuvieran en una matriz imaginaria, de forma que dos elementos colisionan cuando al moverse alguno de ellos, ambos ocupan la misma posición de la matriz. Este método de detección, aunque parezca que se ajusta bien al caso de *Bomberman*, es demasiado simple y no se puede aplicar en este proyecto. El

problema principal es que el jugador puede realizar movimientos cortos que le hacen ocupar más de una posición de la matriz de forma simultánea, por lo que haría falta otro método adicional para controlar estos casos.

Descartada la detección de colisiones de forma lógica, se pasa a analizar los métodos de tipo "físico". Como ya se ha comentado, estos métodos utilizan los sprites de los distintos elementos para evaluar la colisión. En el caso de la detección de colisiones por figura geométrica, se basa en el concepto de envolver cada sprite con una figura, ya sea círculo o rectángulo, y aprovechar las funciones matemáticas de *XNA* para determinar si se produce una intersección entre dos figuras, lo que equivaldría a una colisión. La elección de la figura geométrica a utilizar depende de cuál se ajusta mejor a la forma del sprite.

El sistema de detección por celdas es muy similar al método lógico, salvo por el hecho de que se pueden tener distintas matrices que evalúen las colisiones. La idea es dividir la pantalla en celdas, pero no de forma uniforme, si no a partir de las posiciones de los objetos. Se recorren las celdas buscando grupos de objetos comunes a cada celda y si se quiere saber si uno o más objeto están en una celda determinada se calculan las posiciones: Celda_X(Y)_actual = Posicion_X(Y)_actual_objeto / Ancho_celda, siendo Celda_X(Y)_actual de tipo flotante y Posicion_X_actual_objeto de tipo entero y generando de una a cuatro posiciones. El problema principal es que las búsquedas se hacen demasiado pesadas y que en determinados casos, es necesario aplicar métodos de detección adicionales.

Y por último la detección por píxeles. La idea de este método es aprovechar la información del canal Alpha o de transparencias de los sprites. Para ello se separa la información de las texturas de los objetos y se recorren punto a punto buscando una colisión con ayuda del canal alpha. Si los objetos son transparentes en los puntos en los que se sobreponen no hay colisión, por el contrario si los objetos se sobreponen en puntos de color hay colisión. El problema principal de este método es su procesamiento. Para dos sprites de 64x64 píxeles, cada comparación ocuparía 64x64 x 2 bytes = 8192 bytes x 2 sprites = 16384 bytes; requiriendo 64x64=4096 comprobaciones, es decir una por cada píxel.

En el caso de este proyecto se ha optado por un método mixto de detección por figuras geométricas y detección lógica, aplicando además el movimiento típico de los juegos de *Bomberman* cuando la colisión es menor que un margen determinado; mejorando así el proceso de detección, la velocidad de los movimientos y el resultado final.

El proceso de detección se realiza en dos fases. La primera, llamada fase general o *Broad Phase* sirve para determinar si dos pares de figuras deben ser evaluadas para comprobar una colisión y surge como necesidad de reducir los cálculos innecesarios, evitando comprobar siempre posibles colisiones entre todos los elementos del juegos. Posteriormente se aplica la fase específica o *Narrow Phase*, que sirve para determinar el resultado de la colisión de un par de figuras.

Para la primera fase se aplica el método "lógico". Para cada elemento se sabe la posición que ocupa en la matriz del mapa. Se almacenan en listas los enemigos, los bloques rompibles, los no rompibles, las bombas, las explosiones; en definitiva todos los elementos con los que puede colisionar el jugador y para cada elemento se almacena también su posición y se ordenan las listas en función de esta. Así sólo se evaluará la colisión con los objetos que, en función del movimiento de jugador y su posición puedan bloquearle. De esta manera se simplifica el proceso y se ahorran cálculos.

Para la segunda fase se aplica el método de detección por figuras geométricas. En principio el método de detección por píxel es mejor, pero cuando se tienen muchos elementos en pantalla puede afectar a la velocidad del juego si varios de estos elementos coinciden en la misma zona y hay que evaluar todas las posibles combinaciones de colisión. Por tanto el sistema que mejor se ajusta es el de figuras geométricas. Este sistema, como ya se ha comentado, consiste en rodear los sprites con una o varias figuras geométricas como se muestra en la figura 77. Posteriormente se evalúa la colisión mediante las funciones matemáticas de intersección o inclusión. El problema es que si la figura no se ajusta bien al sprite, se pueden detectar colisiones cuando no las hay; como en el ejemplo de la figura 78.

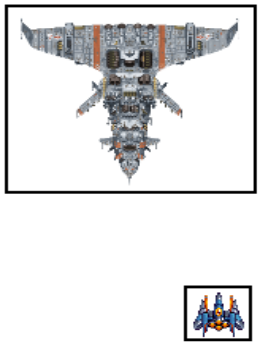

Figura 77: Detección de colisiones con rectángulos

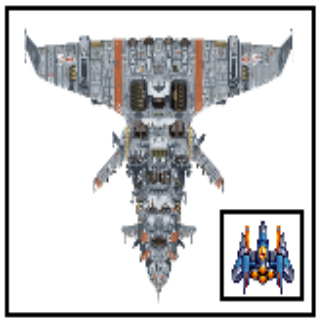

Figura 78: Ejemplo de detección errónea

Para el proceso de detección se ha escogido el rectángulo como figura geométrica, ya que se ajusta bien a la mayoría de los elementos del juego. Pero esto no significa que no se produzca el problema de la detección errónea antes señalado. En la figura 79 se muestra el problema en la primera versión del juego implementada. El rectángulo se ajusta bien al bloque, pero no al personaje, detecta colisión porque la parte izquierda del sprite del personaje es más alta que la derecha, impidiendo el movimiento.

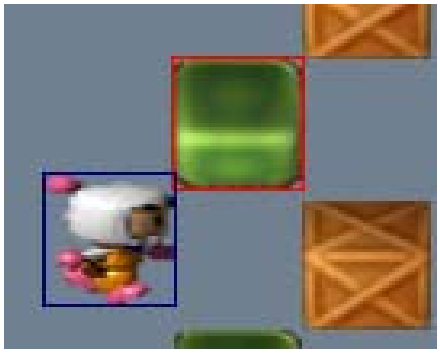

Figura 79: Detección de colisiones errónea en la primera versión del juego

En un principio la solución escogida fue implementar un segundo método de detección, eligiendo el de detección por píxel. Sin embargo cuando muchos elementos se desplazaban en una misma zona de la pantalla la velocidad del juego se veía afectada. Como solución a este problema se mejoró el algoritmo de detección por píxel, evaluando sólo ciertas partes del sprite. Si se toma la figura 79 como ejemplo, al ser el movimiento hacia la derecha y al estar el bloque por encima del personaje, serviría con evaluar sólo los píxeles de la zona superior derecha del personaje y la inferior

izquierda del bloque para detectar la colisión, tal y como se ve en la figura 80. Esta modificación mejoró mucho el proceso de detección y permitió solucionar el problema de ralentización del juego.

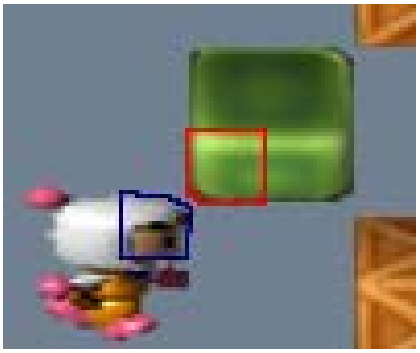

Figura 80: Solución aplicando detección de colisiones por píxel de forma parcial

Aunque esta última implementación resolvía el problema, viendo vídeos de versiones antiguas y actuales de *Bomberman*, se puede observar que la solución implementada no reflejaba bien la mecánica de juego. En realidad en el juego no se evalúan todas las colisiones de la misma forma, si no que dependen de un margen previamente fijado, por lo que se tendrían dos tipos de colisiones una básica y otra avanzada. Este margen se aplica al rectángulo de colisión, que ya no cubre por completo el sprite del jugador ni del bloque. La diferencia entre ambos tipos de colisiones es difícil de explicar sin imágenes, así que se utilizarán las figuras 81, 82 y 83 para hacerlo.

- En la figura 81 el personaje quiere hacer un desplazamiento hacia la derecha pero colisiona con un bloque y el desplazamiento no se produce. Como se puede observar, el rectángulo ya no cubre por completo al jugador, pero intersecciona con el rectángulo del bloque y la colisión se produce dentro de los márgenes mínimos. Es lo que se denomina colisión básica.

- En la figura 82 el personaje quiere hacer un desplazamiento hacia la derecha y colisiona con el bloque. Esta vez, sin embargo, el rectángulo del sprite del jugador no intersecciona con el del bloque, ya que se encuentra fuera del margen de colisión básica. El caso en que los sprites de jugador y bloque colisionan, pero los rectángulos no interseccionan se denomina colisión avanzada. Una vez detectada, se aplica un desplazamiento automático del personaje. Este se desplazará primero hacia abajo y luego hacia la derecha, quedando colocado debajo del bloque al igual que en la figura 83.

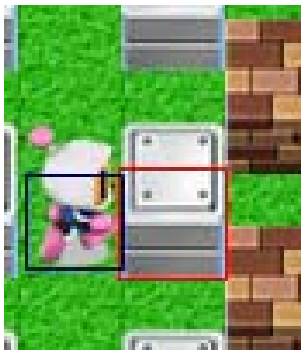

Figura 81: Colisión básica

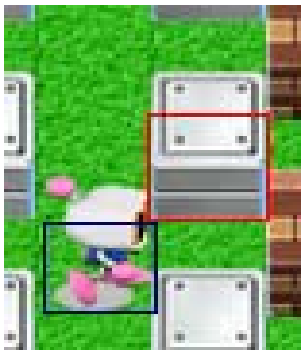

Figura 82: Colisión avanzada

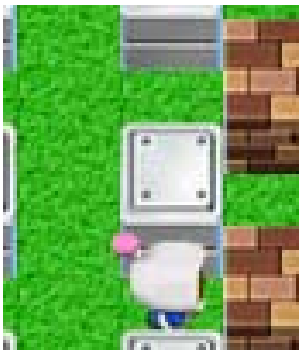

Figura 83: Desplazamiento automático

Por último aclarar dos detalles:

- El primero, por qué los rectángulos de los bloques utilizados para la detección tampoco cubren los sprites por completo. El motivo es que con esto se consigue simular un efecto 3D aunque los modelos de jugador, bloques, enemigos etc. estén en realidad en 2D.

- Y segundo que, en el caso de la detección de colisiones del jugador con otros elementos distintos a los bloques (como las explosiones o los enemigos), estas se evaluarán sin tener en cuenta ningún margen. En el proceso se utilizan rectángulos que cubren por completo al jugador y al elemento correspondiente, de forma que se detecte la colisión sea cual sea el punto en que se produzca.

Una vez analizado y explicado el método de detección de colisiones, hay que detallar qué métodos de qué clases intervienen en el proceso. En primer lugar, en la clase *Sprite*, se definen los rectángulos que se utilizan para evaluar las colisiones. Limite define el rectángulo que rodea por completo al sprite y LimiteFis define el rectángulo con los márgenes X e Y aplicados.

```
MargenY = -20;
MargenX = -5;

public virtual Rectangle Limites{
    get{
        return new Rectangle((int)(Posicion.X - centro.X),
        (int)(Posicion.Y - centro.Y), (int)textura.Width / contframes,
        (int)textura.Height);
    }
}
```

```
public virtual Rectangle LimitesFis{
    get{
        return new Rectangle((int)(Posicion.X - centro.X - (MargenX / 2)),
        (int)(Posicion.Y - centro.Y - MargenY),
        (int)(textura.Width / contframes) + MargenX,
        (int)(textura.Height) + MargenY - 5);
    }
}
```

Código 8: Generación de los rectángulos de colisión

Una vez definidos los rectángulos, en el método MoverJugador de la clase *Jugador*, habrá que comprobar si cada movimiento que implique un desplazamiento provoca colisión, y si es así, identificar además si es básica o avanzada. En el caso de ser básica el movimiento no se realizará. En el caso de ser avanzada, habrá que aplicar un desplazamiento automático y corregir la posición final del movimiento, acción que se lleva a cabo en los métodos AjustarPosX y AjustarPosY.

```
void AjustarPosX(int MargenCasilla, List<Casilla> casillasNivel ...){
    ajustarJugadorX = false;
    casillaColisionaY = false;
    foreach (Casilla cas in casillasNivel){
        if (cas.LimitesFis.Contains((int)Posicion.X )){
            if (!cas.bloqueaJugador && casillaColisionaY == false){
                if (Posicion.X - 1 > cas.Posicion.X){
                    Posicion.X += -(Velocidad / 2);
                }
                else if (Posicion.X < cas.Posicion.X){
                    Posicion.X += (Velocidad / 2);
                }
                ajustarJugadorX = true;
                casillaColisionaY = true;
            }
            else if (cas.bloqueaJugador && casillaColisionaY == false)
                ajustarJugadorX = true;
            }
            else if (cas.LimitesFis.Contains((int)Posicion.X))
                ajustarJugadorX = true;
}
```

Código 9: Ajuste de posición en el eje X después de una colisión avanzada

Respecto al jugador, hay pocas cosas más que destacar. Las acciones relacionadas con las bombas, soltar y detonar, se evalúan a nivel de entrada de usuario como los desplazamientos y ejecutan las acciones correspondientes. En el caso de soltar una bomba, se añade una bomba nueva al mapa colocándola en el centro de la casilla para evitar colisiones erróneas, siempre que la casilla sea válida. Inmediatamente después se añade la lista de bombas activas del jugador. En el caso de la

detonación, se elimina una bomba de las bombas activas del jugador y a partir de esta acción se generará una explosión.

```
Public override void SoltarBomba(List<Bomba> bombas, List<Casilla> casillasNivel){
    if (timerExplosion <= 0f){
        timerExplosion = alcanceExplosion;
        Bomba auxBomba = null;
        if (!tieneBombasRemoto)
            auxBomba = new Bomba(TexturaBomba, 5, 30, false);
        else if (tieneBombasRemoto)
            auxBomba = new Bomba(TexturaBombaRemoto, 1, 1, true);
        bool casillaValida = true;
        foreach (Casilla cas in casillasNivel){
            if (cas.LimitesFis.Contains((int)Posicion.X, (int)Posicion.Y + 10)){
                if (cas.tipoCasilla == 0 || 6 || 8){
                    auxBomba.Posicion = cas.Posicion;
                    casillaValida = true;
                    if (bombas.Count < MaxBombas)
                        cas.tieneBomba = true;
                }
            else if (cas.tipoCasilla != 0)
                casillaValida = false;
        }
    }
    auxBomba.Velocidad = VelocidadBomba;
    auxBomba.Color = ColorBomba;
    foreach (Bomba b in bombas)
        if (b.Posicion == auxBomba.Posicion)
            casillaValida = false;
        if (casillaValida && bombas.Count < MaxBombas) {
            bombas.Add(auxBomba);
            if (OptionsMenuScreen.nivelSonidoActivo > 0)
                SonidoSoltarBomba.Play();
        }
    }
}
```

Código 10: Acción soltar bomba

```
Public void DetonarBombas(List<Bomba> bombas){
    if (bombas.Count > 0){
        Bomba b = bombas[0];
        b.explotada = true;
    }
}
```

Código 11: Acción detonar bomba

Por último se hace referencia a la acción herir, que afecta al jugador cuando es alcanzado por una explosión o un enemigo, actualizando su estado a herido si tiene más de un corazón; o muerto si sólo tenía uno.

```
Public void Herir(int danio, Vector2 PosicionHerido){
    if (!estaHerido)
        Vida -= danio;
    if (Vida <= 0){
        Vida = 0;
        estaHerido = false;
        estaMuerto = true;
    }
    if (!estaMuerto)
        estaHerido = true;
}
```

Código 12: Acción herir jugador

3.3.2. Mapas, niveles y casillas

El siguiente paso, después de implementar el jugador, era desarrollar el entorno de juego en cuanto a mapas y niveles. Recordar que el concepto de nivel, hace referencia a cada uno de los laberintos de los que tiene que escapar el personaje. El de mapa, hace referencia a una agrupación de niveles con una temática común: mismo diseño de bloques, misma música etc.

- **Niveles**

La primera idea consistía en implementar los niveles de la misma forma que al jugador. Cada nivel estaría definido por una serie de sprites que representarían los elementos del nivel en la pantalla: bloques rompibles y no rompibles, ítems etc. Este sistema permitía añadir los elementos de los niveles como clases que heredaban de *Sprite*, encajando perfectamente en la mecánica ya implementada de movimientos y detección de colisiones de jugador. Pero por otro lado, mostraba un diseño poco trabajado, difícilmente modificable y muy difícil de gestionar. Los niveles se convertían en un conjunto enorme de listas de elementos que se podían pintar en pantalla y que había que actualizar de forma constante, creando nuevas listas para ciertos elementos, gestionándolas en clases distintas. En definitiva, el diseño era poco acertado y complicaba la implementación.

La solución a este problema consistía en diseñar una estructura de clases que reflejara de forma lógica el concepto de nivel y casilla y que, apoyándose en los elementos ya implementados

que heredaban de *Sprite*, completaran un diseño correcto. Este segundo y definitivo diseño es el que se muestra en el diagrama de clases de la figura 60.

Los atributos de la clase *Nivel* permiten definir de forma que la gestión de las listas de elementos que lo componen sea después mucho más sencilla. Así, se define en la clase el alto y el ancho del nivel, la disposición de las casillas en forma de matriz de dos direcciones cuyas entradas almacenan los códigos de las casillas que componen el nivel. Un identificador que permite recuperar los elementos del nivel en función del mapa al que corresponden facilitando la carga de sprites, efectos, etc. También ciertas características especiales como si tiene enemigos, si tiene ítems, si el jugador puede ser herido o si es un nivel con jefe final además de los mensajes de inicio y fin de nivel.

```
class Nivel
{
    int Ancho = 0;
    int Alto = 0;
    int[,] TrazadoNivel = new int[0, 0];
    string CjtoCasillas = "Mapa1";
    Color ColorFondo = new Color(230, 240, 255, 0);
    bool noEnemigos = false;
    bool noBloques = false;
    bool noSalida = false;
    bool noPowerUp = false;
    bool jugadorNoHerible = false;
    bool tieneMensajeBienvenida = false;
    bool tieneMensajeSalida = false;
    bool nivelJefe = false;
    string mensajeBienvenida = "BIENVENIDO! :)";
    string mensajeSalida = "FELICIDADES!";
}
```

Código 13: Atributos de la clase Nivel

De esta manera quedan definidos los niveles, que se completan con la clase *Casilla*.

- **Casillas**

En esta clase se definen los tipos de casillas que puede tener un nivel además de reflejar mediante los atributos los distintos estados por los que puede pasar una casilla. Así se simplifica más aún la gestión de las casillas, puesto que cada objeto casilla reflejará su posición en la matriz, las

coordenadas en la pantalla, su tipo, si ha generado un ítem o la salida, o si contiene una bomba o ha generado una explosión; evitando la gestión tan pesada de listas comentada al principio del punto.

```
Public class Casilla
{
    public int Margen = 0;
    public Vector2 Origen;
    public int tipoCasilla = 0;
    public bool Muerto = false;
    public bool bloqueaJugador = false;
    public bool tieneBomba = false;
    public bool haExplotado = false;
    public bool tieneSalida = false;
    public bool tienePowerUp = false;
    public bool esVisible = true;
    public Vector2 Posicion;
}
```

Código 14: Atributos de la clase Casilla

Como se ha dicho, la distribución del nivel se representa por una matriz de dos dimensiones que almacena los códigos de las casillas. Los tipos de casilla definidos, con su código correspondiente, son los siguientes:

```
0 = Suelo
1 = Pared, bloque no rompible
2 = Punto de salida de jugador
3 = Bloque rompible
4 = Casilla donde no se permiten enemigos ni bloques
5 = Bloque sin enmigos.
6 = Salida para saltar de nivel.
7 = Bloque vacio
8 = Casilla PowerUp
9 = Nada
10 = Salto al mapa de entrenamiento
11 = Salto al mapa 1
12 = Salto al mapa 2
13 = Salto al mapa 3
14 = Salto al mapa Jefe
21 = Enemigo
51-91 = Casilla salto nivel superado
```

Código 15: Atributos de la clase Casilla

El tipo 0 representa el suelo, casillas por las que el personaje puede andar. El tipo 1 representa un bloque no rompible del nivel. El tipo 2 es el punto de salida inicial del jugador al comenzar un nivel, por lo que genera una casilla de tipo 0 desde la que el jugador empieza a moverse. El tipo 3 representa un bloque rompible. El tipo 4 representa casillas que no pueden estar

ocupadas por enemigos ni bloques, permitiendo así generar los niveles de forma automática pero con la seguridad de que el personaje no quedará encerrado en su posición inicial ni estará rodeado por enemigos. El tipo 5 es una variación del anterior pero sólo impide la aparición de enemigos. El tipo 6 representa la casilla de salida. El tipo 7 es un bloque vacío que sirve para evaluar colisiones en momentos de cambio de estado como la conversión de una bomba en explosión. El tipo 8 representa las casillas con ítem. El 9 representa casillas vacías y sirve para diseñar niveles que no tengan forma rectangular o cuadrada. Del 10 al 14 son casillas utilizadas para facilitar la depuración del juego, de forma que permite saltar entre los distintos mapas implementados sin necesidad de superar sus niveles. La casilla tipo 21 representa el punto de salida de un enemigo en el mapa. El rango restante representa casillas de salto, similares a las del rango 10..14, que permitirían al jugador saltar a cada uno de los niveles en concreto. Se incluyeron pensando en poder desbloquear los mapas y niveles una vez superados, accediendo a ellos a través de casillas de este tipo desde un mapa especial pero no se implementó finalmente.

- **Sprites y generación visual**

En el constructor de la clase se inicializarán los valores de los atributos en función del tipo de casilla leído en la matriz. Al igual que para el jugador, para las casillas se calculan dos rectángulos que permitan evaluar las colisiones. El método Limite genera un rectángulo que contiene el sprite completo, de forma que se puedan evaluar las colisiones con las explosiones y las colisiones básicas con el jugador. El método LimiteFis genera un rectángulo reducido por los márgenes, de forma que se puedan evaluar las colisiones avanzadas con el jugador. El código de estos métodos es muy similar al de los métodos homónimos de la clase Jugador por lo que no se incluye en este documento.

El método RecuperarNivel de la clase *NivelS* es el que se encarga de controlar el nivel que se debe cargar en cada momento. Cuando se supera un nivel se llama a RecuperarNivel cargando el siguiente, se lee la matriz de definición del nivel y se generan los elementos de forma que se van creando una serie de listas con los elementos actualizables: bloques, ítems, enemigos, etc. Mediante el identificador de mapa se pueden cargar los sprites correspondientes a un mapa concreto.

Se muestra a continuación el código de una de las matrices de nivel incluidas en el juego, en este caso, la de los niveles del Mapa1. Se ha utilizado la misma matriz para cada uno de los diez niveles de cada mapa, pero esto no significa que los niveles sean iguales, ya que se generan de forma aleatoria en base a un algoritmo que distribuye los bloques rompibles partiendo de los códigos de la matriz del nivel. Sólo se mantendrá la posición de los bloques no rompibles y la de salida del personaje; el resto, se genera durante la creación del mapa. Por otro lado, destacar que las matrices de nivel no incluyen casillas de tipo 21, excepto en el nivel del jefe final. Estas casillas son puntos fijos de salida de enemigos y para evitar que el jugador conozca estos puntos se generan de forma aleatoria. Sólo el jefe final aparece siempre desde el mismo lugar: la casilla central de su nivel.

```
private static int[,] TrazadoNivel1 = new int[,]
{
    {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
    {1,2,4,5,3,5,5,5,5,4,0,4,4,5,5,5,4,0,1},
    {1,4,1,3,1,5,1,5,1,5,1,5,1,5,1,5,1,4,1},
    {1,3,5,5,5,5,5,5,5,5,5,4,0,4,5,5,4,4,1},
    {1,5,1,5,1,5,1,4,1,5,1,5,1,5,1,5,1,5,1},
    {1,5,5,5,5,5,5,0,4,5,5,5,5,4,4,0,4,5,1},
    {1,5,1,5,1,5,1,4,1,5,1,5,1,5,1,5,1,5,1},
    {1,4,5,5,5,5,5,5,5,5,4,5,5,5,5,5,5,4,4,1},
    {1,0,1,5,1,5,1,5,1,0,1,5,1,5,1,5,1,0,1},
    {1,5,5,5,4,5,5,5,5,4,4,5,5,0,4,4,5,5,1},
    {1,4,1,5,1,5,1,5,1,5,1,5,1,5,1,5,1,4,1},
    {1,0,4,5,5,5,5,4,0,4,5,5,5,5,5,5,4,0,1},
    {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
};
```

Código 16: Matriz de los niveles del Mapa 1

La generación de un nivel se lleva a cabo en el método LoadContent y es automática, de forma que sólo los bloques no rompibles ocupan el mismo lugar. La cantidad de bloques rompibles disminuye cuanto más alto es el nivel que se alcanza complicando la partida. De la misma manera, los enemigos no siempre parten de los mismos puntos y su número aumenta cuanto más alto es el nivel por el mismo motivo que disminuye el número de bloques rompibles. Tal y cómo se indicaba en el párrafo anterior, y se puede observar en la tabla de código 16, en el mapa no se incluyen posiciones de salida de los enemigos (código 21), si no que se generan desde las casillas con valor 0, o desde casillas contiguas a estas que no tengan valor 4 o 5 (generación de enemigos prohibida).

```
Public List<Bomba> bombasJugador = new List<Bomba>();
public List<Explosion> explosionesJugador = new List<Explosion>();
public List<ExplosionBloqueRompible> expRomp = new List<ExplosionBloqueRompible>();
public List<Casilla> casillasNivel = new List<Casilla>();
public List<Casilla> casillasSalida = new List<Casilla>();
public List<Enemigo> listaEnemigos = new List<Enemigo>();
public List<Jefe> listaJefes = new List<Jefe>();
public List<Peon> peonesActivos = new List<Peon>();
public List<PowerUp> powerUpsActuales = new List<PowerUp>();
```

Código 17: Listas necesarias para la gestión de un nivel

El código de generación de los niveles es demasiado largo para incluirlo en este punto, pero básicamente refleja el proceso comentado. Una vez leído el nivel y cargadas las listas, se pintan los elementos correspondientes al mapa, generando el nivel de forma visual. Los sprites usados para cada uno de los niveles son los siguientes:

Figura 84: Suelo Mapa 1

Figura 85: Bloque no rompible Mapa 1

Figura 86: Bloque rompible Mapa 1

Figura 87: Suelo Mapa 2

Figura 88: Bloque no rompible Mapa 2

Figura 89: Bloque rompible Mapa 2

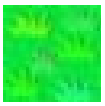

Figura 90: Suelo Mapa 3

Figura 91: Bloque no rompible Mapa 3

Figura 92: Bloque rompible Mapa 3

Figura 93: Suelo Mapa Jefe

Figura 94: Bloque no rompible Mapa Jefe

Figura 95: Casilla de salida cerrada

Figura 96: Casilla de salida abierta


Figura 97: Bomb Up


Figura 98: Remote Bomb


Figura 99: Fire Up


Figura 100: Heart

Las figuras 97, 98, 99 y 100, representan los sprites del los Power Up definidos. *Bomb Up* incrementa en uno el número de bombas simultáneas que el jugador puede soltar en el mapa. *Remote Bomb* proporciona la capacidad de detonar las bombas. *Fire Up* aumenta en uno el alcance de las explosiones y *Heart* añade un corazón más al jugador, permitiéndole ser alcanzado por una explosión o un enemigo. Al igual que en el caso del personaje, los sprites de los niveles, así como los de los ítems de *Bomb Up* y *Remote Bomb,* han sido extraídos del juego *Bomberman*. Los sprites que representan la casilla de salida, y los ítems *Fire Up* y *Heart* han sido diseñados para este proyecto.

3.3.3. Enemigos

Una vez implementados jugador y entorno, el siguiente paso es el desarrollo de los enemigos. Los enemigos a implementar se consideran *NPCs*. Un *NPC* es un personaje de un juego controlado automáticamente por inteligencia artificial u otra técnica. En un principio, los enemigos a implementar debían simular el comportamiento del jugador. Con esta intención se diseñó la clase *Enemigo* como una clase que extendía de *Peon*. Esta relación de herencia permitía utilizar la infraestructura de clases ya creada para el jugador en lo que se refiere a colisiones, posicionamiento en pantalla y pintado. Sólo se debería añadir en la nueva clase el código que determinara el comportamiento de los enemigos a implementar.

- **Estructura básica**

Observando la implementación de la clase *Peon*, se puede ver de forma sencilla que atributos son comunes al jugador y a los enemigos. Los atributos relacionados con las bombas, como la velocidad o el origen permiten relacionar las bombas con el personaje que la soltó ya sea jugador o enemigo. Los atributos relacionados con los movimientos, permiten gestionar de forma sencilla las animaciones de los enemigos.

Atributos como estaHerido o tieneBombas representa el estado del personaje. Por último aniadidoAPuntuacion almacena el valor en puntos del personaje al ser alcanzado por una explosión y morir.

```
Public Vector2 VelocidadBomba = Vector2.UnitY;
public Vector2 OrigenBomba = Vector2.Zero;
public string DireccionUltimoMovimiento = "Abajo";
public bool MismaDireccion = false;
public int PotenciaBomba = 1;
public int MaxBombas = 1;
public int Vida = 1;
public int Vidas = 1;
public bool estaMuerto = false;
public bool estaHerido = false;
public bool tieneBombas = false;
public bool aniadidoAPuntuacion = false;
```

Código 18: Atributos de la clase Peon

Tras estos primeros pasos en la implementación de los enemigos, se decidió que el comportamiento que tendrían no sería similar al del jugador, si no que se diseñarían e implementarían varios tipos de enemigos que aportaran más posibilidades al juego. Para el diseño de estos enemigos se tomaron como base los enemigos del juego original de *Bomberman*.

Puesto que la clase *Peon* ya proporciona la estructura y los métodos necesarios para gestionar las colisiones, el posicionamiento en pantalla y el pintando de los enemigos, la clase *Enemigo* será la encargada de reflejar su comportamiento. Los enemigos no responden a la entrada de usuario, si no que deben tener un comportamiento propio, reaccionando ante las situaciones que se producen en el juego, siendo además sus acciones lo más “lógicas” posibles. Esto significa que el comportamiento de los enemigos debe reflejar desde un principio su función principal, que es acabar con el jugador; pero siempre reaccionando a las distintas situaciones que se plantean durante el juego de la mejor forma posible. Por ejemplo, se debe evitar repetir de manera continuada un mismo comportamiento, evitar la repetición de caminos de forma reiterativa o huir de una posible explosión.

- **Diagramas de estado**

Para el diseño e implementación de los enemigos, se optó por las máquinas de estado. Las máquinas de estados finitos (FSM), también conocidas como autómatas de estados finitos (FSA), son modelos de comportamiento de un sistema o un objeto, con un número limitado de estados o condiciones predefinidos, donde existen transiciones de estado. Están compuestas por 4 elementos principales:

- Estados que definen el comportamiento y pueden producir acciones
- Transiciones de estado que implican cambio de un estado a otro
- Reglas o condiciones que deben cumplirse para permitir un cambio de estado
- Eventos de entrada que permiten las transiciones

Existen varios motivos por los que elegir las máquinas de estado para diseñar e implementar los enemigos. El primer motivo es que se adaptan muy bien al proyecto. Los enemigos son relativamente sencillos, por lo que su diseño mediante FSM será sencillo y claro. También la implementación lo será, ya que los diseños mediante FSM son fáciles de implementar.

Por otro lado, el modelado de *NPC* en los videojuegos con máquinas de estado es una técnica muy usada gracias a su flexibilidad, de manera que el diseño puede ir evolucionando sin provocar grandes cambios. A continuación, se mostrarán cada uno de los diagramas de estado que se han realizado durante el proyecto, intentando reflejar la evolución en el diseño de los enemigos. El primero, hace referencia a los enemigos propuestos inicialmente, cuyo comportamiento intenta simular al del jugador humano.

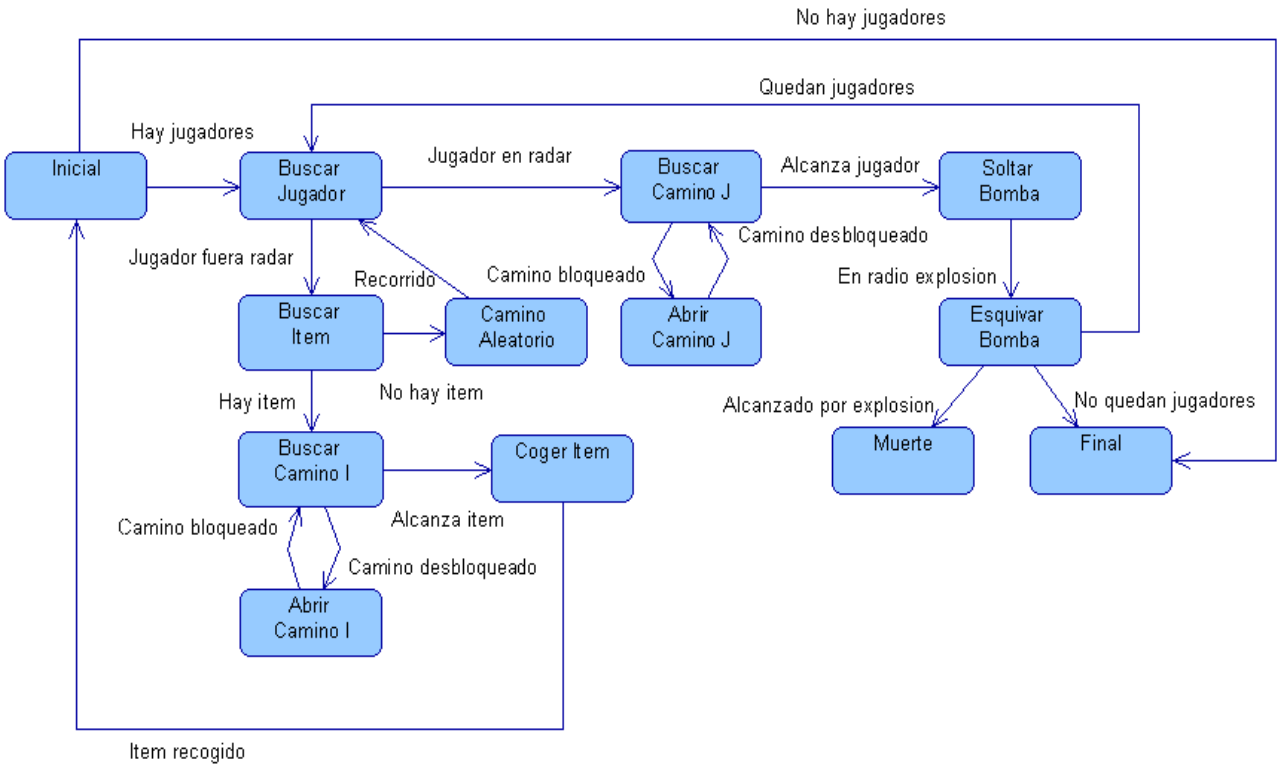

Figura 101: Diagrama de estados de enemigo v1

Como se ve en la figura 101, se parte del estado **Inicial**, si no hay jugadores en el mapa se ha acabado la partida y se transita al estado **Final**. Si hay jugadores, se busca en el mapa su posición y se evalúa si están dentro del *radar* del enemigo, estado **Buscar Jugador**.

Antes de continuar se debe aclarar el concepto de *radar*. El radar es un radio de acción o alcance del enemigo que determina si detecta o no la posición del jugador. Si el radar de un enemigo tiene un alcance de X casillas, este y el jugador deben estar separados por menos de X casillas para que pueda detectar su posición. Si el jugador está dentro del límite de casillas del radar de un enemigo, este detectará su posición y buscará un camino hasta él.

El proceso de búsqueda del camino hasta el jugador por parte de un enemigo está representado por el estado **Buscar Camino J**. Puede que el camino generado esté bloqueado por bloques rompibles, si es así, se deben romper los bloques, transición al estado **Abrir Camino J**. Si el enemigo ha recorrido el camino y tiene al jugador lo suficientemente cerca debe poner una bomba para intentar matarle, transición al estado **Soltar Bomba**. Una vez depositada la bomba el enemigo

debe huir de su explosión, evitando así ser alcanzado, estado **Esquivar Bomba**. Si no encuentra una salida y es alcanzado morirá, transición a estado **Muerte**. Si esquiva la bomba y tras la explosión no hay más jugadores habrá ganado el juego, estado **Final**. Si se esconde de la explosión y tras esta quedan jugadores, debe ver si se encuentran en su radar de alcance, transición al estado **Buscar Jugador**. Vueltos a este estado, se explica la rama alternativa, si el jugador está fuera del radar de alcance, se busca un posible ítem en el mapa, estado **Buscar item**. Si hay algún ítem en el mapa se genera un camino hasta él, estado **Buscar Camino I**. Si el camino generado esté bloqueado por bloques rompibles se deben romper, transición al estado **Abrir Camino I**. Si el camino está libre se coge el ítem, estado **Coger Item**, y se vuelve al estado **Inicial**, comenzando el ciclo otra vez. Por último, volviendo al estado **Buscar Item**, si no hay ningún ítem que recoger en el mapa se generará un camino aleatorio que desplace al enemigo a una nueva posición.

Este es el primer diagrama de estados generado. Hay que destacar que, para un enemigo concreto, tanto el jugador como el resto de enemigos son objeto de su ataque. Esto significa que el modo de juego plantea un "todos contra todos", en el que un personaje debe acabar con todos los demás, ya sean personajes manejados por el usuario o por el juego. Por otro lado, también hay que comentar que en el diagrama de estados se detectaron varios casos no contemplados, como la posibilidad de encontrar los caminos bloqueados por bombas, en cuyo caso el enemigo no huiría. También el caso en que el enemigo abre un camino bloqueado pero el jugador ya no está en el radar, por lo que seguiría un camino incorrecto; o la posibilidad de que en el camino generado el enemigo tenga en el radar al jugador, pero no lo detecte porque no ha alcanzado el punto final del camino.

Durante el proceso de solución de estos errores, se decidió que los enemigos implementados no simularían el comportamiento del jugador, si no que se comportarían como los enemigos del *Bomberman* original. Sin embargo, muchos aspectos del comportamiento eran similares en ambos casos, por lo que la solución a los problemas de la primera versión del diagrama afectaría también a la nueva versión. Se muestra primero la nueva versión del diagrama de estados, que refleja el nuevo comportamiento de los enemigos. Posteriormente ya sobre el diagrama, se explicará cómo se resolvieron los problemas comentados. Señalar por último que, con el nuevo diseño, los enemigos no pondrán bombas ni podrán coger ítems del mapa. Su objetivo será ahora alcanzar al jugador, ya que al contactar con él harán que pierda un corazón o una vida.

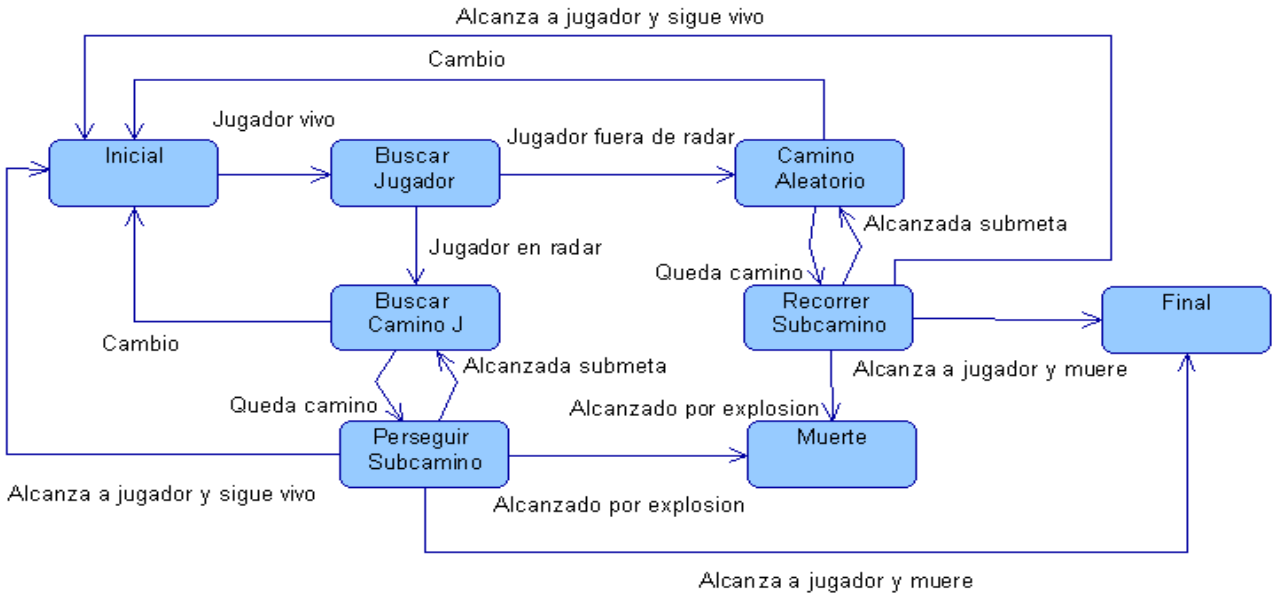

Figura 102: Diagrama de estados de enemigo v2/Jefe

La figura 102 corresponde al diagrama de estados que representa el comportamiento de los nuevos enemigos. Se parte del estado **Inicial**. Si el jugador está vivo, se busca su posición, estado **Buscar Jugador**. Si está dentro del alcance del radar se genera un camino hasta él, estado **Buscar Camino J**. Del camino completo generado hasta el jugador, se recorrerá sólo una parte, transición a **Perseguir Subcamino**. Una vez que se ha alcanzado el punto final del subcamino, se evalúa si el jugador es alcanzable con esta misma ruta. Si es así, se recorre otro subtramo, nueva transición a **Perseguir Subcamino**. Si el jugador ya no es alcanzable con la ruta generada, se ha producido un cambio y desde **Buscar Camino J** hay una transición al estado **Inicial**. Por otro lado, durante el tiempo en que el enemigo está recorriendo un subcamino puede ser alcanzado por una bomba, lo que llevaría al estado **Muerte**; o puede alcanzar al jugador y matarlo, lo que le llevaría al estado **Final**; o alcanzar al jugador pero que a este le queden corazones, lo que le llevaría al estado **Inicial.** Volviendo al estado **Buscar Jugador**, si este no está dentro del alcance del radar, se transita al estado **Camino Aleatorio**, estado en el que se genera un nuevo camino con dirección aleatoria. Se empieza a recorrer el camino por subtramos, estado **Recorrer Subcamino**, comprobando cada vez que se alcanza una submeta si el camino sigue siendo válido. En caso de no serlo, desde el estado **Camino Aleatorio** se volvería al estado **Incial**. Mientras está en el estado Recorrer Subcamino, el enemigo puede ser alcanzado por una bomba, lo que le llevaría al estado **Muerte**; o puede alcanzar al jugador, lo que le llevaría al estado **Final** si este muere o al estado **Inicial** si le quedaban más corazones.

Este nuevo diagrama de estados soluciona varios problemas que tenía la versión uno. Por ejemplo, que el enemigo recorría caminos completos aunque le llevaran a un destino que ya no era correcto porque el jugador había cambiado de posición. En esta nueva versión, mediante los estados **Perseguir Subcamino** y **Recorrer Subcamino**, los caminos generados se recorren de forma parcial, permitiendo al enemigo cambiar su estrategia de movimiento más rápido. Sin embargo, con el nuevo diseño, los enemigos se comportan de forma demasiado agresiva; causando esto dos problemas principales. El primero, que en determinadas situaciones en las que muchos enemigos rodean al jugador escapar se hace casi imposible, puesto que una vez que los enemigos detectan al jugador en el radar siguen sus movimientos hasta alcanzarlo. Por otro lado, si el jugador tiene más tiempo y espacio para pensar el ataque, es sencillo acabar con los enemigos de forma individual debido a que persiguen al jugador a pesar de que en el camino haya bombas; de manera que en muchas ocasiones, en su intento de alcanzar al jugador, acaban muriendo en una explosión. El siguiente paso lógico en el diseño de los enemigos es añadir nuevos estados que compensen de alguna manera el comportamiento en ataque, con desplazamientos defensivos que les permitan evitar las bombas aunque el jugador pueda ser alcanzado; combinando así este comportamiento conservador con el más agresivo antes comentado.

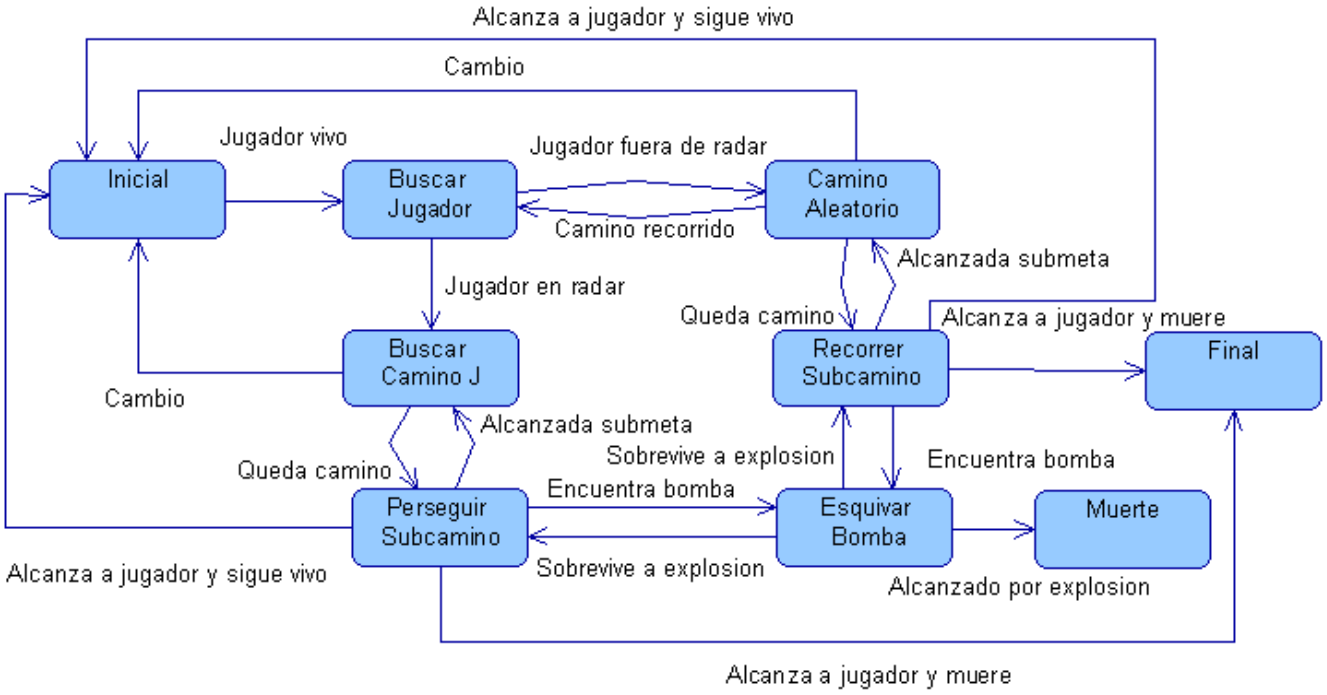

Figura 103: Diagrama de estados de enemigo v2.1A

La nueva versión del diagrama de estados, representada en la figura 103, añade pocas modificaciones con respecto a la anterior. Cuando el enemigo se encuentra en los estados **Perseguir Subcamino** o **Recorrer Subcamino**, y detecta una bomba, no mantiene siempre su estrategia. Por ejemplo, el enemigo se encuentra en el estado **Perseguir Subcamino** porque ha detectado al jugador en el radar de casillas. Por tanto, ha generado un camino y está recorriendo parte de él. Si durante este proceso, detecta una bomba, pasará al estado **Esquivar Bomba**, en el que recorrerá el camino a la inversa para intentar evitar la explosión. Si lo consigue volvería al estado **Perseguir Subcamino**. Si no lo consigue, se transita al estado **Muerte**. De esta manera se consigue evitar el comportamiento agresivo de los enemigos descrito con anterioridad, solucionando los problemas comentados. Por otro lado, habrá que tener en cuenta que, cuando se tenga que trasladar a código el diagrama, habrá que compensar de alguna manera el grado de ataque/defensa, de forma que el comportamiento no sea repetitivo.

El diagrama de la figura 103 es el definitivo en lo que a diseño de enemigos se refiere. A partir de él, aplicando diversas modificaciones en el código para lograr comportamientos de ataque/defensa diferentes, se iban a implementar todos los enemigos del juego. Sin embargo, durante el proceso de implementación, y tras observar videos de varias de las versiones de *Bomberman*, se decidió añadir un tipo de enemigo adicional. El nuevo diseño, está basado en los enemigos tipo *Slime* (punto ***2.2.2 Enemigos y Power Ups*** y figura 47) del *Bomberman* original y sus variaciones en versiones posteriores. Este tipo de enemigos tenía más de una vida. El comportamiento antes de ser alcanzado por una explosión será similar al de los otros enemigos, basado en el diagrama v2.1A, con las correspondientes modificaciones para variar las tendencias de ataque/defensa. Pero al recibir el impacto de una explosión, variará su comportamiento, haciéndolo más agresivo y aumentando su velocidad y las tendencias de ataque. Adicionalmente, en algunas versiones comerciales de *Bomberman*, el enemigo dejaba rastros en las casillas por la que pasaba haciendo que la velocidad del jugador disminuyera al pasar por ellas. Este aspecto también se ha incluido en el diseño.

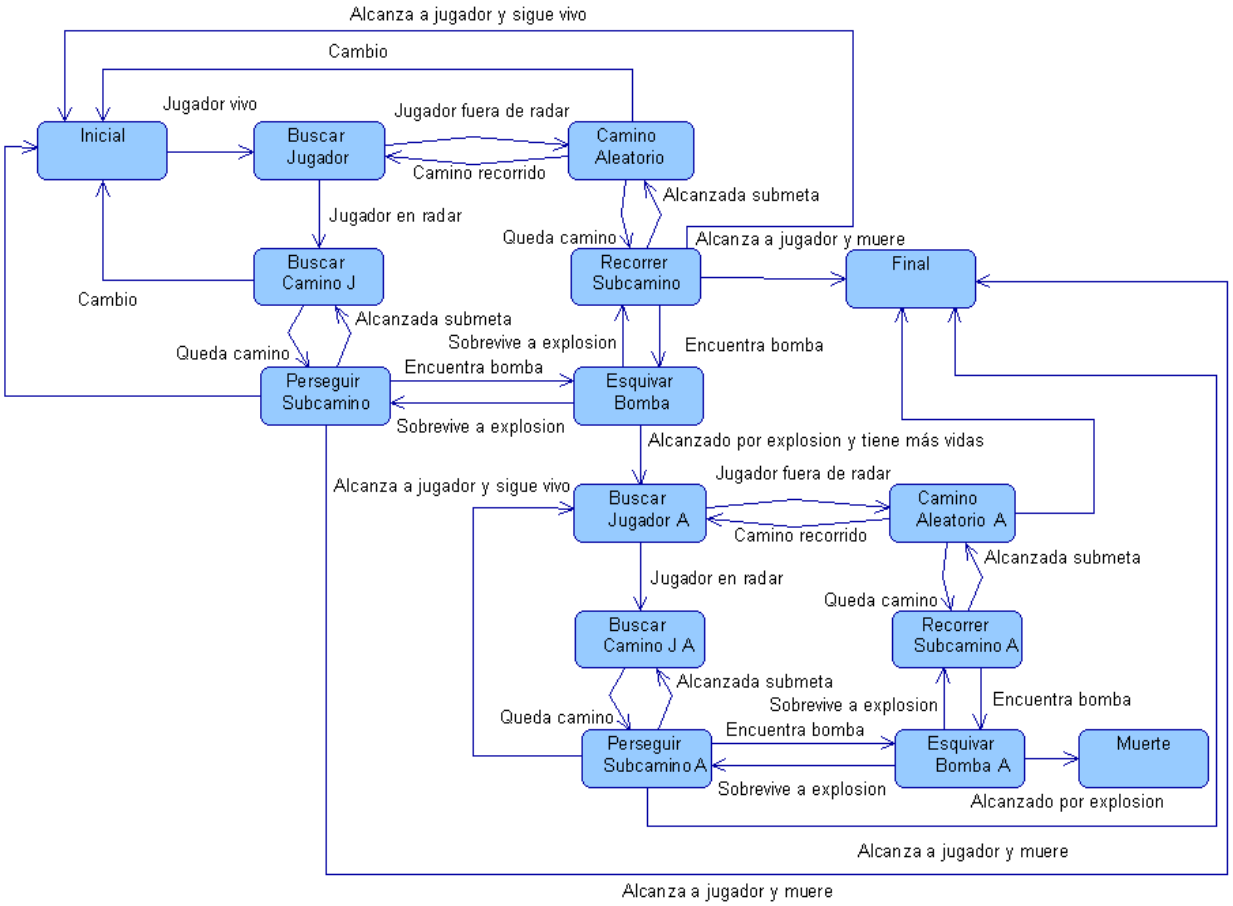

Figura 104: Diagrama de estados de enemigo v2.1B

El diagrama de la figura 104 representa el diseño del nuevo enemigo. Como se puede ver, el diagrama es idéntico en estados y transiciones al de la versión 2.1A hasta que el enemigo es alcanzado por una explosión. Si es alcanzado, transitará a **Buscar Jugador A**, que se corresponde con el nuevo comportamiento en el que el enemigo varia sus tendencias de ataque siendo más agresivo. A partir de aquí los estados y transiciones son equivalentes a los ya explicados en la versión 2.1A, con la salvedad de que el enemigo mantendrá ese cambio de comportamiento que lo hace más agresivo. Si el jugador no está en el radar generará un camino aleatorio que recorrerá por segmentos, estados **Camino Aleatorio A** y **Recorrer Subcamino A**. Si el jugador está en el radar lo perseguirá, generando un camino que también recorrerá por segmentos, estados **Buscar Camino J A** y **Perseguir Subcamino A**. Cuando se encuentre recorriendo estos subcaminos podrá detectar bombas que el jugador deposite y esquivarlas, estado **Esquivar Bomba A**. Si es alcanzado por la explosión morirá, estado **Muerte**. En el caso de alcanzar el enemigo al jugador, le quitará un corazón o una vida si no dispone de más corazones, estado **Final**.

- **Sprites**

Una vez analizados los distintos diagramas de estado, se explican los conceptos relacionados con la implementación. Para empezar se han implementado tres tipos de enemigos además de un jefe final, a saber: *Enemigo_B* (B de ballon, por la forma de globo del enemigo), *Enemigo_C* (C de coin, por la forma de moneda del enemigo), *Enemigo_S* (S de slime, por el rastro que deja el enemigo cuando es alcanzado) y *Jefe*. Todos los enemigos están basados en enemigos del *Bomberman* original, *Enemigo_B* en *Ballon, Enemigo_C* en *Minvo* y *Pontan* y *Enemigo_S* en *Oneal*. El jefe final no se basa en ninguno del juego original, puesto que en la versión original no habías jefes finales.

En cuanto a la codificación, hay que recordar que la clase *Enemigo* hereda de la clase *Peon*, que a su vez hereda de la clase *Sprite*. Esto permite utilizar la infraestructura ya desarrollada de métodos para los procesos de gestión de animaciones y detección de colisiones. Puesto que estos métodos ya están explicados en el punto ***3.3.1 Jugador*** (en los apartados que hacen referencia a las animaciones y las colisiones), no se volverán a comentar ahora. Antes de empezar con la implementación de los movimientos de los enemigos, se muestran los sprites utilizados.

Figura 105: Sprite Enemigo_C_ParadoAnim

Figura 106: Tira de sprites Enemigo_C_MuertoAnim

Figura 107: Tira de sprites Enemigo_C_AbajoAnim

Figura 108: Tira de sprites Enemigo_C_ArribaAnim

Los desplazamientos a derecha e izquierda de los enemigos de tipo *Enemigo_C* se realizan rotando el sprite *Enemigo_C_ParadoAnim*, mostrado en la figura 105.

Figura 109: Tira de sprites Enemigo_B_ArribaAnim

Figura 110: Tira de sprites Enemigo_B_AbajoAnim

Figura 111: Tira de sprites Enemigo_B_LadoAnim

Figura 112: Tira de sprites Enemigo_B_MuertoAnim

Los movimientos laterales a derecha e izquierda de los enemigos de tipo *Enemigo_B* se realizan con el mismo sprite, *Enemigo_B_LadoAnim*, mostrado en la figura 111.

Figura 113: Tira de sprites Enemigo_S_ArribaAnim

Figura 114: Tira de sprites Enemigo_S_AbajoAnim

Figura 115: Tira de sprites Enemigo_S_LadoAnim

Figura 116: Tira de sprites Enemigo_S_MuertoAnim

Los movimientos laterales a derecha e izquierda de los enemigos de tipo *Enemigo_S* se realizan con el mismo sprite, *Enemigo_S_LadoAnim*, mostrado en la figura 115. Todos los sprites utilizados en las animaciones de los enemigos pertenecen al juego *Bomberman World* y han sido descargados de una base de datos [176] de sprites. Los de *Enemigo_B* y *Enemigo_C* corresponden al diseño original de *Ballon* y *Minvo* respectivamente. No se han encontrado sprites que representen el diseño original de *Oneal*, utilizándose una serie de sprites alternativos. En cuanto al jefe final, sus sprites se corresponden con los de *Enemigo_C*, con la salvedad de que el tamaño se ha aumentado.

Figura 117: Correspondencia de tamaño entre Jefe y Enemigo_C

- **Implementación de los diagramas de estado**

Una vez aclarados los conceptos relacionados con las animaciones y los sprites, se pasa a explicar con más detalle cómo se han pasado a código en la implementación de los enemigos. La base de la implementación de los enemigos es la clase *Enemigo*. En ella quedan representados los diagramas de estado diseñados. En el caso del jugador, los movimientos del personaje respondía a acciones del jugador, pero en este caso los enemigos deben moverse de forma automática por el mapa según los diagramas diseñados. Para ello, la clase contiene una serie de atributos que permiten controlar el estado del enemigo, los movimientos realizados y posibles, así como los subcaminos recorridos y por recorrer.

```
Public enum Direccion{ Arriba, Abajo, Izquierda, Derecha }
public struct Disponible{
    public bool Arriba;
    public bool Abajo;
    public bool Izquierda;
    public bool Derecha;
    public bool Ninguna;
```

Código 19: Atributos de control de los movimientos del enemigo

Los enumerados Direccion y Disponible representan la dirección actual del enemigo y las direcciones disponibles en el siguiente movimiento respectivamente. Gracias a Dirección se puede controlar cuál es el frame que se debe mostrar en una animación o que dirección lleva el enemigo para no entrar en un bucle. Por otro lado, Disponible permite evaluar que direcciones se pueden tomar desde un punto concreto, evitando que el enemigo tome caminos bloqueados. La clase *Enemigo* incluye, además de un atributo de cada uno de los tipos comentados en la tabla 19, otra serie de atributos que determinan su comportamiento y estado. Como se dijo en el punto de los diagramas, es necesario que los enemigos se comporten de manera lógica, evitando repetir de manera constante los mismos movimientos o eligiendo en cada momento la estrategia de ataque defensa más correcta. Se puede decir que el comportamiento de un enemigo viene determinado de forma principal por tres temporizadores, los cuales marcan la velocidad con la que el enemigo toma decisiones y cambia de estrategia o estado. Así mismo, es necesario evaluar que parte se ha recorrido de un camino, los subcaminos que quedan por recorrer, si el camino que se está recorriendo es de huida o el mínimo de casillas que se debe retroceder para evitar una explosión. Todos estos factores quedan determinados en los siguientes atributos:

```
Public Disponible DireccionesDisponibles;
public Direccion DireccionUltimoMovimiento;

public float Velocidad;
public float distanciaRecorrida;
public float timerEspera;
public float limiteTimerEspera;
public float timerEsperaRandNum;

public Direccion DireccionRandom;
public bool Chocado;
public bool InvertirDireccion;
public bool estaEsperando;

public bool esperandoAMorir;
public bool ParaMuere;
public bool hiereAJugadorSiToca;

public bool cambio;
public Vector2 MueveAPosicion;
public bool llegaADestino;
public bool llevaDireccionRandom;
public int espacioMovimientoMin;

List<Casilla> casillasAlrededor = new List<Casilla>();
```

Código 20: Atributos de control de comportamiento del enemigo

Como se ha comentado, DireccionUltimoMovimiento y DireccionesDisponibles permiten controlar la dirección del último movimiento y las que se pueden tomar desde la posición actual respectivamente. Velocidad, como su propio nombre indica, representa la velocidad de los desplazamientos del enemigo. distanciaRecorrida, almacena el número de casillas que se han recorrido persiguiendo al jugador. Este atributo permite evitar situaciones en las que varios enemigos persiguen al jugador hasta acorralarlo; cambiando el enemigo de estrategia tras un máximo de casillas de persecución. timerEspera, LimiteTimerEspera y TimerEsperaRandNum son los tres temporizadores que marcan el comportamiento del enemigo. timerEspera indica el tiempo que lleva un enemigo realizando una acción determinada, ya sea buscando al jugador, generar un camino o evaluar los obstáculos de un subcamino. LimiteTimerEspera almacena el tiempo máximo en milisegundos que el enemigo puede estar realizando una acción sin llegar a completarla, por ejemplo el tiempo máximo en recorrer un camino, en generarlo o en buscar al jugador. Por último TimerEsperaRandNum, almacena el tiempo que el enemigo puede dedicar a recorrer un camino aleatorio. Cada vez que se genera un nuevo camino aleatorio el rango de valores posibles se reduce de forma que, cuanto más tiempo pasa desde que se inició el nivel, menos tiempo se puede dedicar a recorrer un camino aleatorio sin buscar al jugador.

DireccionRandom almacena la última dirección aleatoria tomada. Chocado, sirve para determinar si un camino está bloqueado, impidiendo al enemigo continuar en la misma dirección. InvertirDireccion permite que el jugador haga un cambio rápido de dirección para esquivar una bomba, de forma que si su valor es true, el enemigo dará prioridad a escapar de la explosión. EstaEsperando sirve para controlar los saltos entre estados, de manera que si su valor es true, no se generarán nuevos movimientos porque todavía tiene alguno pendiente. EsperandoAMorir y paraMuere indican que el enemigo ha sido alcanzado por una explosión y no tiene más vidas, por lo que debe lanzarse la animación de muerte y detener la generación de movimientos. El valor de hiereAJugadorSiToca será false en este caso, impidiendo que hiera al jugador si lo toca mientras se lanza la animación de muerte. Cambio almacena que, durante el tiempo que se ha tardado en recorrer un camino, el jugador ha cambiado de posición y no está en el radar, de forma que se debe aplicar una nueva estrategia. Los últimos atributos hacen referencia al recorrido de los caminos, MueveAPosicion, almacena la posición final de un recorrido determinado. llegaADestino, indica que el enemigo ha completado un recorrido. llevaDireccionRandom, que el camino que se está recorriendo en este instante es aleatorio, y espacioMovimientoMin, el número de casillas que contiene un subcamino. Por último casillasAlrededor almacena un número concreto de casillas alrededor del enemigo que permite evaluar los caminos generados en busca de bloques, bombas etc.

El proceso de actualización del estado del enemigo se realiza en cada ciclo en el método Update de la clase *Enemigo*. Dentro de este método se comprueba se el enemigo está vivo, si es así, se genera un nuevo listado con las casillas que tiene alrededor y se evalúa su estado actual. En el caso de estar recorriendo un subcamino que no ha completado, seguirá recorriéndolo y comprobando la lista de casillas. Si ha llegado al destino y llevaba una dirección aleatoria, comprobará el radar para buscar al jugador. Si no la llevaba, es que perseguía al jugador y evaluará si este ha cambiado de posición durante la persecución. En el caso de estar esperando, realizará la acción en espera siempre y cuando no haya saltado el timer. En el caso de no estar esperando y no haber llegado al destino el enemigo ejecuta el desplazamiento. Por último, si está muerto actualiza el estado para evitar herir al jugador.

El esqueleto básico del método Update queda representado en la tabla de código 21. En la tabla sólo se representan algunos de los estados en forma de ejemplo y no se incluye el código de

cada estado puesto que sería demasiado pesado. Los métodos a los que se llama en cada estado se explican a continuación de la tabla 21.

```
if (!estaMuerto){
    List<Casilla> casillasAlrededor = new List<Casilla>();
        if (llegaADestino && ParaMuere){
            ...
        }
        else if (llegaADestino && llevaDireccionRandom){
            ...
        else if (!llevaDireccionRandom){
            ...
        if (estaEsperando){
            if (timerEspera > limiteTimerEspera){
                ...
        }
        if (!estaEsperando && !llegaADestino){
                    Movimiento();
                    base.Update();
        }
}
else if (estaMuerto){
    hiereAJugadorSiToca = false;
}
```

Código 21: Esqueleto del método Update de la clase Enemigo

Una vez evaluado el estado, es necesario ejecutar las acciones correspondientes. En el primer estado se comprueba si el jugador está en el radar. Esto se realiza en el método ObtenerDireccion, que evalúa la posición del jugador en función del alcance del radar, generando una dirección aleatoria si está fuera o la dirección del jugador si está dentro de forma que el camino a generar sea de persecución. En la tabla 22 se puede ver como se comprueba la distancia entre el jugador y el enemigo en función del radar. La variable Num almacenará un número que representa la dirección que debe tomar el enemigo para perseguir al jugador si este está dentro de los límites. El enemigo ataca tres de cada cinco veces, compensando así un comportamiento demasiado agresivo.

```
while (direccionValida == false){
    auxRand = rand.Next(5);
    if ((!esRandom) && ((auxRand == 0) || (auxRand == 1) || (auxRand == 2))){
        if (Jugador.PosicionJugador.X < Posicion.X - Radar)
            Num = 3;
        else if (Jugador.PosicionJugador.X > Posicion.X + Radar)
            Num = 4;
    ...
    }
    else Num = RandomNum(1, 5);
}
```

Código 22: Atributos de control de comportamiento del enemigo

En el caso de no encontrarse el jugador dentro de los límites del radar, el enemigo debe determinar una dirección aleatoria para generar un camino en dicha dirección.

En el método DireccionesDisponibles se comprueba si la dirección generada está o no disponible, es decir, si algo bloquea el camino a generar. Si fuera así, se volvería al estado anterior para determinar una nueva dirección eliminando la anterior de entre las disponibles.

Una vez determinada la dirección, esta se marcará como dirección válida. Se debe generar ahora el camino en función del contenido de las casillas cercanas. El enemigo posee una lista (de un tamaño máximo determinado) de casillas que están a su alrededor y según sean estas se generará el camino, intentando evitar los bloques.

```
if (Num == 1 && DireccionesDisponibles.Arriba == true){
    direccionValida = true;
    if (DireccionUltimoMovimiento == " Arriba "){
        foreach (Casilla auxCasilla in casillasNivel){
            if (ComprobarCasillasDisponibles(auxCasilla, Direccion.Arriba)){
            ...
        }
}
else if (Num == 2 && DireccionesDisponibles.Abajo == true){
    direccionValida = true;
    if (DireccionUltimoMovimiento == "Abajo"){
        foreach (Casilla auxCasilla in casillasNivel)
            if (ComprobarCasillasDisponibles(auxCasilla, Direccion.Abajo)){
            ...
        }
}
```

Código 23: Generación del camino en función de la dirección

Si en el proceso de recorrido de un subcamino el enemigo se ha quedado bloqueado porque hay una bomba, se debe generar un nuevo camino en dirección contraria para huir de la explosión.

```
if (Chocado){
    switch (Num){
        case 1: //Mueve Arriba
        if (DireccionUltimoMovimiento == "Izquierda" || ...){
            if (cambio == false) break;
        }
        if (puedeIr){
            foreach (Casilla auxCasilla in casillasNivel){
                if (ComprobarCasillasDisponibles(auxCasilla, Direccion.Arriba)){
                    MueveAPosicion = auxCasilla.Posicion;
                    DireccionRandom = Direccion.Arriba;
                    if (DireccionUltimoMovimiento == "Arriba") MismaDirec = true;
                    else{
                        DireccionUltimoMovimiento = "Arriba";
                        MismaDireccion = false;
                         if (distanciaRecorrida > espacioMovimientoMin)
                             distanciaRecorrida = 0;
```

Código 24: Cambio de dirección y camino de huida

Como se puede ver en la tabla de código 24, una vez detectada la bomba se comprueba la dirección del movimiento actual para evitar que el de huida acerque al enemigo a la explosión. Se genera un nuevo camino actualizando la dirección actual y controlando la distancia que se ha recorrido. Si se evita la explosión, al superar el número de casillas del subcamino se inicializa la distancia recorrida y se vuelve al estado inicial. La tabla 24 sólo recoge el código del movimiento si el jugador se desplazaba hacia arriba, pero el método incluye la comprobación para cada una de las cuatro direcciones.

El método ComprobarCasillasAlrededor es el encargado de generar la lista de casillas que el enemigo tendrá en cuenta al generar un camino en función de un radio determinado.

```
Public virtual ComprobarCasillasAlrededor(){
    List<Casilla> auxCasillasAlrededor = new List<Casilla>();
    Rectangle Radio = LimitesFis;
    Radio.X = (int)Posicion.X - (MaxRX / 2);
    Radio.Y = (int)Posicion.Y - (MaxRY / 2);
    Radio.Height = MaxRY;
    Radio.Width = MaxRX;
    foreach (Casilla auxCasilla in casillasNivel){
        if (Radio.Contains((int)auxCasilla.Posicion.X, (int)auxCasilla.Posicion.Y){
                    auxCasillasAlrededor.Add(auxCasilla);
        }
    }
return auxCasillasAlrededor;
}
```

Código 25: Método para generar la lista de casillas

El método ComprobarCasillasDisponibles genera el camino en función de la dirección y el contenido de las casillas.

```
Public bool ComprobarCasillasDisponibles(){
    switch (direccion){
        case Direccion.Arriba:{
        if (auxCasilla.Limites.Contains(Posicion.X, (Posicion.Y - 30 - Velocidad)){
            if (!auxCasilla.bloqueaJugador && !auxCasilla.tieneBomba){
                return true;
            }
        ...
        }
        break;
...
```

Código 26: Método para generar un camino en función del contenido de las casillas

La tabla 26 sólo contiene parte del código del método, en el que se comprueba para el contenido de una única casilla si la dirección del camino que se está generando es arriba.

Una vez comprobado el estado del enemigo, determinada la dirección y generado el camino con sus correspondientes subcaminos, el enemigo se desplaza mediante el método Movimiento. El método mueve al enemigo hasta un destino determinado. Si el enemigo rota en algún desplazamiento, como en el caso de *Enemigo_C*, este método marca el grado de rotación.

```
Public virtual void Movimiento(){
    if (MueveAPosicion.X < Posicion.X - Velocidad){
        DireccionRandom = Direccion.Izquierda;
        Posicion.X -= Velocidad;
        if (Rota)
            Rotacion -= Velocidad / 10;
    }
    else if (MueveAPosicion.X > Posicion.X + Velocidad){
        DireccionRandom = Direccion.Derecha;
        Posicion.X += Velocidad;
        if (Rota)
            Rotacion += Velocidad / 10;
    }
    else if (MueveAPosicion.Y < Posicion.Y - Velocidad){
        DireccionRandom = Direccion.Arriba;
        Posicion.Y -= Velocidad;
    }
    else if (MueveAPosicion.Y > Posicion.Y + Velocidad){
        DireccionRandom = Direccion.Abajo;
        Posicion.Y += Velocidad;
    }
     else llegaADestino = true;
```

Código 27: Código del método Movimiento de la clase Enemigo

De la clase *Enemigo* heredan las clases *Enemigo_B*, *Enemigo_C*, *Enemigo_S* y *Jefe*. En estas clases se agregan métodos, se añaden otros y se asignar valores a los atributos de forma que se consigue un comportamiento diferente para cada enemigo. La implementación de *Enemigo_B* y *Enemigo_C* se corresponde con el diagrama de estados v2.1A mostrado en la figura 103. La de *Enemigo_S* con el diagrama 2.1B de la figura 104. Por último, la implementación de la clase *Jefe*, se corresponde con el diagrama v2, de la figura 102. A continuación se profundizará en la implementación de las clases *Enemigo_B*, *Enemigo_C*, *Enemigo_S* y *Jefe*, explicando cómo condicionan el comportamiento de cada enemigo.

Enemigo_B es el enemigo más sencillo del juego, es el más lento en desplazamientos y en la toma de decisiones. Sólo tiene una vida y proporciona 100 puntos al jugador cuando muere. No añade ningún método nuevo pero sí modifica el método Update, modificando su funcionalidad para ajustar la posición del enemigo en función de los márgenes. Los valores de los atributos que condicionan su comportamiento son los siguientes:

```
MargenX = -5;
MargenY = -5;
limiteTimerEspera = 600;
timerEsperaRandNum = 600;
posicionAjustada = false;
valorPuntos = 100;
Vida = 1;
Velocidad = .5f;
```

Código 28: Valores de los atributos de Enemigo_B

Enemigo_C es una versión mejorada del anterior. Es más rápido y sus temporizadores son más bajos, lo que le permite cambiar de estrategia más rápido que a *Enemigo_B*. Ni sobrescribe ni añade métodos nuevos. Los valores de los atributos que condicionan su comportamiento son:

```
MargenX = -5;
MargenY = -8;
limiteTimerEspera = 250;
timerEsperaRandNum = 250;
valorPuntos = 250;
Rota = true;
Vida = 1;
Velocidad = 1f;
```

Código 29: Valores de los atributos de Enemigo_C

Enemigo_S es una versión mejorada de los dos anteriores. Sus temporizadores son más bajos que los de *Enemigo_C*, lo que permite que cambie de estrategia y que ejecute los movimientos más rápido. En su primera vida la velocidad es la misma que la de *Enemigo_B* pero una vez alcanzado, dobla la velocidad y reduce más aún los temporizadores. Es su segunda vida además deja un rastro en las casillas por las que pasa haciendo que el jugador se mueva más despacio si las pisa. Es rastro mantendrá su efecto en 5 casillas de forma simultánea como máximo. La clase sobrescribe el método Update para ajustar la posición del enemigo en función de los márgenes y añade el método ActualizarCasillaES que gestiona las casillas con el rastro que deja el enemigo.

```
//Valores primera vida
MargenX = -5;
MargenY = -5;
limiteTimerEspera = 225;
timerEsperaRandNum = 225;
valorPuntos = 500;
Vida = 2;
Velocidad = .5f;
//Valores segunda vida
Velocidad = 1f;
limiteTimerEspera = 150;
frameRate = 45;
```

Código 30: Valores de los atributos de Enemigo_S

```
foreach (CasillaES s in listaCasillaES){
    if (s.Limites.Contains((int)auxc.Posicion.X, (int)auxc.Posicion.Y)){
        existeEnemigo_S = true;
        s.contPermanece = 0;
    }
    else s.contPermanece++;
}
if (!existeEnemigo_S && Vida <= 1){
    CasillaES auxES = new CasillaES(TexturaEnemigo_S, 1, 1);
    auxES.Posicion = auxc.Posicion;
    listaCasillaES.Add(auxES);
    existeEnemigo_S = true;
}
break;
```

Código 31: Creación del rastro de Enemigo_S

```
void ActualizarCasillaES(){
    velocidadModificada = false;
    foreach (CasillaES s in listaCasillaES){
        s.Update();
        Rectangle rectCasillaES = new Rectangle();
        if (rectCasillaES.Contains(Jugador.X, JugadorY) && !velocidadModificada){
            Jugador.estaEnCasillaES = true;
            velocidadModificada = true;
        }
        if (s.contPermanece > 5) s.desaparece = true;

        if (s.Alpha <= 0) casillasESABorrar.Add(s);
    }
    if (!velocidadModificada) Jugador.estaEnCasillaES = false;

    foreach (CasillaES sR in casillasESABorrar) listaCasillaES.Remove(sR);
}
```

Código 32: Actualización del rastro de Enemigo_S

Por último, hay que comentar la implementación de la clase *Jefe*. La implementación se corresponde con el diagrama de estados v2, de la figura 102. La clase añade dos atributos nuevos, TimerMovimiento y TimerCambio que permiten controlar el comportamiento del jefe de forma más estricta. Además sobrescribe el método Update casi por completo, puesto que este tipo de enemigo no esquiva las bombas ni combina ataque/defensa; si detecta al jugador lo persigue hasta matarlo o ser alcanzado por una explosión. Además posee un radar más amplio que el del resto de enemigos que alcanza quince casillas en cualquier dirección. En la implementación también se debe modificar la detección de colisiones puesto que el tamaño es cuatro veces mayor que el de los enemigos o el jugador. La última característica importante a destacar es el comportamiento tras ser alcanzado por una explosión. Si el jefe es alcanzado desaparecerá generando dos enemigos de la mitad de tamaño. Este comportamiento se repetirá tres veces. , hasta que los enemigos generados tengan el tamaño de *Enemigo_C*. Una vez eliminados todos los enemigos generados el jefe morirá.

```
limiteTimerEspera = 600;
timerEsperaRandNum = 600;
posicionAjustada = false;
valorPuntos = 1000;
Vida = 1;
Velocidad = .5f;
```

Código 33: Valores de los atributos de Jefe

```
Public override Rectangle Limites{
    get{
        return new Rectangle(
            (int)(Posicion.X - centro.X),
            (int)(Posicion.Y - centro.Y),
            (int)textura.Width / contframes,
            (int)textura.Height);
    }
}

public override Rectangle LimitesFis{
    get{
        return new Rectangle(
            (int)(Posicion.X - (centro.X * Escala) - (MargenX / 2)),
            (int)(Posicion.Y - (centro.Y * Escala) - MargenY),
            (int)(Escala * textura.Width / contframes) + MargenX,
            (int)(Escala * textura.Height) + MargenY - 5);
    }
}
```

Código 34: Generación de los rectángulos de colisión de Jefe

```
int num = RandomNum(1, 5);
    if (!esRandom){
        if (Jugador.PosicionJugador.X < Posicion.X - 15)
            num = 3;
        else if (Jugador.PosicionJugador.X > Posicion.X + 15)
            num = 4;
        else if (Jugador.PosicionJugador.Y < Posicion.Y - 15)
            num = 1;
        else if (Jugador.PosicionJugador.Y > Posicion.Y + 15)
            num = 2;
    }
switch (num){
    case 1:
        if (DireccionUltimoMovimiento != "Arriba" || esRandom == false)
            foreach (Casilla auxCasilla in casillasNivel)
                if (auxCasilla.Limites.Contains(Posicion.X, (Posicion.Y)) {
                    if (!auxCasilla.bloqueaJugador && !auxCasilla.tieneBomba){
                            if (DireccionUltimoMovimiento != "Arriba")
                                MismaDireccion = false;
```

Código 35: Método ObtenerDireccion de la clase Jefe

Como se puede observar el método ObtenerDireccion de la clase *Jefe* es muy similar al de la clase *Enemigo*, con la salvedad de que sólo se cambia de dirección al detectar una bomba si bloquea directamente el camino del enemigo. En cuanto a la determinación de la dirección y la posterior generación del camino, la tabla de código 35 muestra que es similar a la de la clase *Enemigo* aumentando el alcance del radar.

3.3.4. Otros

En este último punto del apartado, se comentarán otros detalles relativos que se consideran importantes. Las tres clases que se comentarán son *Camera2D*, *OnScreenKeyboard* y *Settings*.

Camera2D es una implementación básica de una cámara 2D que proporciona *Microsoft* a través de *XNA Creators Club*. Si el juego se ejecuta en *Zune*, el sistema lo detecta y carga la cámara, pasando a una visión reducida ajustada a la pantalla del dispositivo que sigue los movimientos del jugador. El problema principal es que, por no disponer de este dispositivo, sólo se ha podido probar con versiones muy iniciales del juego. Por tanto, aunque se incluye el código de la cámara, el código de detección de dispositivo que lanza la cámara está comentado.

Por otro lado, *OnScreenKeyboard*, es una implementación que emula el teclado virtual de *XBOX 360*. Con esta clase se soluciona el problema de la gestión de las cadenas de teclado, permitiendo al usuario introducir sus iniciales para registrar una puntuación. La clase mapea el teclado completo y lo muestra en pantalla. Se han realizado modificaciones del código que proporciona *Microsoft* en *XNA Creators Club*, eliminando la parte visual del teclado y manteniendo sólo la lectura de teclado.

```
Public string Text{
    get { return oskb_EntryString; }
    set {
        oskb_EntryString = value;
        if (oskb_EntryString == "")
            oskb_EntryString = "   ";
        Settings.nombre = oskb_EntryString;
}
```

Código 36: Lectura de la entrada de teclado

```
Private char[] cKeyValues = new char[] {
    'a', 'b', 'c', 'd', 'e', 'f', 'g', '1', '2', '3',
    'h', 'i', 'j', 'k', 'l', 'm', 'n', '4', '5', '6',
    ...
    'A', 'B', 'C', 'D', 'E', 'F', 'G', '1', '2', '3',
    'H', 'I', 'J', 'K', 'L', 'M', 'N', '4', '5', '6',
    ...
    ',', ';', ':', '\'', '"', '!', '?', '1', '2', '3',
    '[', ']', '{', '}', '`', '$', '%', '4', '5', '6',
};
```

Código 37: Caracteres que se quieren leer de teclado

Una vez controlados los caracteres que se quieren leer, se debe mapear la entrada, asignando cada carácter que se quiere a la tecla correspondiente del teclado. Este proceso se lleva a cabo en el método Update_TextEntry. A continuación se muestra un ejemplo del proceso de mapeo.

```
if(KBoard.IsKeyDown(Keys.A)){AddKey = true; if(Shifted) key = "A"; else key = "a";
if(KBoard.IsKeyDown(Keys.B)){AddKey = true; if (Shifted) key = "B"; else key = "b";
...
if(KBoard.IsKeyDown(Keys.D1){AddKey = true; if (Shifted) key = "!"; else key = "1";
if(KBoard.IsKeyDown(Keys.D2){AddKey = true; if (Shifted) key = "@"; else key = "2";
...
if(KBoard.IsKeyDown(...)){AddKey = true; if (Shifted) key = "~"; else key = "`";
if (KBoard.IsKeyDown(...)){AddKey = true; if (Shifted) key = ":"; else key = ";";
```

Código 38: Mapeo del teclado

Por último, la clase *Settings* gestiona los ficheros en los que se almacenan las opciones guardadas y las puntuaciones máximas. El proceso de lectura y escritura es sencillo ya que se realiza a través de las funciones de lectura y escritura de fichero XML que proporciona *Visual Studio*. Para gestionar de forma automática los datos como ficheros XML se deben crear estructuras, que posteriormente se pasaran a objetos de tipo XmlSerialize. Para leer se utiliza el método deserialize y para escribir el método serialize.

```
[Serializable]
public struct HighScoreData{
    PlayerName = new string[count];
    Score = new int[count];
    Level = new int[count];
    Count = count;
}
```

Código 39: Estructura utilizada para gestionar las puntuaciones máximas

Capítulo 4
Conclusiones

Una vez finalizado el proyecto se analiza el resultado teniendo en cuenta los objetivos y las expectativas que se fijaron en un principio.

Por tanto, en este capítulo se comentan que conclusiones se han extraído de la realización de este proyecto.

4.1. Conclusiones

Una vez finalizado el proyecto, se puede afirmar que el trabajo realizado ha cumplido de manera satisfactoria la mayoría de los objetivos que se habían fijado. En primer lugar, hay que destacar que uno de los factores más importantes para llevar a cabo un desarrollo de este tipo de aplicaciones, es que están muy vinculadas a las comunidades que le dan soporte. Esto es algo a lo que se hizo referencia en puntos anteriores, evaluando las distintas posibilidades existentes y resaltando que el uso de *XNA* era una ventaja. Una vez finalizado el proyecto se puede recalcar que sin duda lo es, ya que *XNA* está respaldado por *Microsoft* y una comunidad enorme que le colabora en foros y blogs. Seguramente hace unos años, sería impensable poder participar en colectivos agrupados por un interés como desarrollar aplicaciones para una videoconsola determinada, pero las herramientas que existen hoy día en la red (blog, wiki, foros,…) permiten que colectivos de este tipo puedan intercambiar ideas, formación, opiniones e incluso recursos; que facilitan mucho las cosas.

En cuanto al propio videojuego, sin duda se han cubierto todos los objetivos y requisitos planteados a lo largo del desarrollo. El clon de *Bomberman* es bastante fiel al original en cuanto a movimientos, desarrollo, o enemigos; manteniendo la mayoría de las características que han hecho de este juego uno de las más versionados. Por otro lado, añade algunas de las mejoras que las versiones posteriores fueron añadiendo como la inclusión de los enemigos finales; así como algunos aspectos propios añadidos por las distintas personas involucradas en el proyecto, como el diseño aleatorio de los mapas o el comportamiento de los enemigos. En cuanto a la inclusión de inteligencia artificial, se puede decir que el resultado ha sido satisfactorio, consiguiendo unos enemigos variados con un comportamiento correcto, integrándose bien con el mapa y respondiendo de forma lógica a las acciones del jugador.

En lo referente a la programación, el API *XNA* proporciona un gran número de soluciones a cada problema que se ha planteado. En el caso de este proyecto, los primeros pasos se llevaron a cabo con los movimientos de los personajes y su posterior animación. En este aspecto el API es muy completo proporcionando acceso a dispositivos de teclado, ratón y mandos de *XBOX 360* (con librerías disponibles en la comunidad *Creator's Club* es posible emular los mandos aunque no sean

los de *XBOX 360*). En cuanto a la animación, *XNA* proporciona soluciones para la carga de texturas 2D así como de modelos 3D. El siguiente paso fue el manejo de colisiones, aspecto de sobra explicado en este documento. Las funciones matemáticas del API permiten manejar este aspecto sin problemas. La última parte, relacionada con los mapas y los enemigos, están más relacionadas con las decisiones de diseño tomadas y su implementación. Dependía principalmente de la capacidad de los programadores para plasmar correctamente el diseño, puesto que no se basaban en ninguna librería del API, si no en la correcta programación de los diagramas de estado y las funciones de control. Hay que resaltar un último aspecto relacionado con *XNA*, ya que la dimensión del juego no explota por completo la librería. Faltan por explorar las funciones de juego en red, aspecto que queda pendiente para desarrollos futuros.

Por último, en lo que a motivación y objetivos personales se refiere, este proyecto ha cubierto las expectativas que me había planteado. Debía suponer un primer paso en el desarrollo de los videojuegos y así ha sido, siendo además un proceso muy entretenido. Me ha permitido conocer aspectos básicos como la animación de personajes, el control de movimientos, de detección y evaluación de colisiones, la interacción con distintos dispositivos y periféricos; el funcionamiento del *Creator's Club* de *Microsoft*.

Para concluir, espero que todos estos objetivos cumplidos me permitan afrontar también con éxito, en un futuro a corto plazo, el desarrollo de un clon de *Super Gussun Oyoyo*; videojuego que se propuso como base en los primeros días de realización de este proyecto.

Capítulo 5
Líneas futuras

Una vez realizado el proyecto, se analiza en el capítulo 5 que trabajos futuros es posible desarrollar a partir de este pequeño paso.

Sin duda hay muchos aspectos que se podían haber incluido en el juego y que no se han tratado en este proyecto.

En este capítulo se comentarán algunos de estos posibles trabajos.

5.1. Líneas futuras

Puesto que todo proyecto debe de ser terminado en un plazo de tiempo prefijado, de forma que no se estén incluyendo mejoras prácticamente cada día, hay objetivos que no se han podido incluir en esta versión pero que pueden servir como mejoras para el futuro. He aquí algunas líneas futuras:

En primer lugar, el juego multijugador. El clon de *Bomberman* realizado en este proyecto se basa en el modo de juego original, pero uno de los modos de juego más jugados por los fans de la saga es el multijugador, un "todos contra todos" por ver quien es el último jugador que queda vivo en el mapa. En lo relacionado con los controles y la lógica del juego, la inclusión sería sencilla, puesto que en el diseño ya se han tenido en cuenta la mayor parte de los factores que afectan a este modo. Por otro lado, sería necesario revisar el diseño y la implementación de los enemigos, desarrollando un nuevo tipo que interactúe con las bombas y los ítems de la misma forma que lo hace el jugador. Además el nuevo diseño debe tener en cuenta las acciones del resto de enemigos, puesto que la mecánica ahora es diferente. Por tanto, la parte más importante a tener en cuenta en esta mejora es el rediseño de los enemigos, de forma que el multijugador sea jugable aunque no haya otros jugadores humanos a los que enfrentarse.

Por otro lado, y derivado del modo multijugador, la siguiente mejora a plantear consistiría en permitir partidas en red. La mayoría de los juegos indie del catálogo de *XBOX 360* no cuentan con juego en red, puesto que el soporte *XNA* de versiones anteriores a la 3.1 no era demasiado amplio. Esta nueva versión ha remodelado la librería de funciones de juego en red, simplificando muchos aspectos y añadiendo nuevas funcionalidades. Por tanto, una clara mejora estaría en el desarrollo de un modo multijugador en red.

En cuanto a mejoras en lo ya implementado, surgen rápidamente tres ideas que completan las funcionalidades existentes:

1. Permitir guardar la partida. En principio esta funcionalidad es muy sencilla de añadir, bastaría con guardar en un fichero los datos de nivel que se almacenan durante la partida,

pero el tipo de juego está orientado al mercado arcade y lo que prima es llegar lo más lejos y alcanzar la mayor puntuación posible en una partida.

2. Por otro lado se podría haber incluido un modo en el que se desbloqueen lo niveles. De hecho, en el diseño se incluyeron casillas que permiten saltar a un nivel concreto o a un mapa determinado. Pero finalmente no se utilizaron por la mecánica del juego.

3. Por último, una funcionalidad muy utilizada en las últimas versiones de *Bomberman* es el editor de niveles. En el caso de leer los niveles desde fichero la mejora sería casi instantánea, puesto que los niveles se generan a partir de matrices. Sin embargo, este método es algo arcaico y lo correcto sería implementar un editor que permitiera seleccionar la extensión y el aspecto del nivel, así como la disposición de las casillas desde el propio juego. Para ello, sería necesario implementar la entrada de ratón, que por no tener funcionalidad alguna en el proyecto actual no se ha incluido. De la misma forma, habría que desarrollar un "parseador" que transformara el diseño en pantalla del nivel en la matriz correspondiente.

Anexo A

Planificación

A.1. Introducción

Un proyecto no debe ser empezado sin antes haberse llevado a cabo una planificación previa. Uno de los fines de esta planificación es lograr que el proyecto quede estructurado en etapas, lo cual facilita mucho los cálculos de los plazos necesarios para alcanzar los objetivos marcados. La división del proyecto en fases también tiene una gran utilidad a la hora de asignar recursos materiales y humanos a las distintas actividades. Todo esto sumado, da como resultado un mayor control sobre el proyecto y una mayor precisión a la hora de calcular su viabilidad, ya que al tener una noción de los plazos y recursos necesarios para llevar a cabo las distintas partes, se tendrá una idea aproximada de su coste total. En primer lugar, antes de detallar las fases del proyecto, se hará una pequeña referencia a las fases típicas de un proyecto comercial, para poder comparar su planificación y desarrollo con el de este proyecto.

A.2. Ciclo de vida de un juego comercial

En este punto se detallarán las fases que típicamente componen el diseño y desarrollo de un videojuego comercial. Esto no significa que sean las que se han seguido en este proyecto, si no que servirá para comparar después ambos procesos.

El desarrollo de un videojuego comercial se compone (aunque por supuesto puede variar según el proyecto), desde su inicio, de las siguientes fases:

- **Concepto**: en esta fase, como su propio nombre indica se define el concepto del juego. A partir de una idea de partida o una tormenta de ideas, se crea una propuesta del jugo y el arte conceptual, que servirá como primera base y ejemplo de cómo serán el juego y su historia. En esta fase se definirán conceptos como el género, las características generales, la ambientación (incluyendo arte conceptual), plataformas de desarrollo, cronograma estimado, presupuesto y análisis de riesgos. En esta fase participan diseñadores, analistas y artistas conceptuales.

- **Pre-Producción**: en esta etapa se evalúa la viabilidad del juego y se determina si el proyecto puede ser realizado por el equipo. Posteriormente se define una guía de proyecto, que indica como va a ser construido el juego. Se define el juego describiéndolo en forma clara, detallando la mecánica, gameplay, vistas, niveles, personajes, las distintas pantallas, interfaz de usuario, historias, assets, etc. Se define la estética definitiva del juego en un documento que define los objetos y personajes que deben ser creados. Por último se detalla un plan de hitos y prototipos.

- **Producción**: en esta fase comienza la construcción del juego. Se escribe el código, se crea el arte gráfico y los sonidos. Se definen y diseñan los niveles del juego. Además el grupo de testeadores comenzará a trabajar en cuanto algo de lo desarrollado pueda ser jugado. En esta fase hay que tener en mente que el juego puede cambiar o evolucionar, apareciendo nuevas características o quitando otras, manteniendo siempre actualizada la planificación.

- **Alfa**: este es el punto del desarrollo en el que el juego puede ser jugado de principio a fin. Esto no significa que el proceso haya finalizado, ya que pueden quedar detalles por agregar o arreglar, e incluso que no todas las funcionalidades estén terminadas; pero motor, interfaz de usuario y subsitemas están completos. Se comienza la verificación en busca de errores. Las tareas principales cambian, enfocándose en terminar las funcionalidades y arreglar los problemas que surjan del testeo.

- **Beta**: punto en el que todas las funcionalidades están desarrolladas. Las funcionalidades pendientes son integradas y el desarrollo se detiene, lo único que se hace es arreglar errores surgidos de las pruebas. El objetivo en esta etapa es estabilizar el proyecto y eliminar la mayor cantidad de errores posible antes de liberar el juego

- **Congelación de código**: una vez solucionados los errores encontrados en *Beta* (o al menos los mas críticos) se obtiene el código para la liberación final. En esta etapa se congela el código y queda pendiente de aprobación para pasar a ser la versión final.

- **Liberación**: cuando la versión del código se aprueba, el juego esta completo, verificado y listo para ser enviado a los canales de distribución.

- **Parches**: después de la distribución surge una nueva fase de testeo, realizada esta vez por los usuarios. En la mayoría de los juegos actuales, se ha hecho casi inevitable la publicación de parches posteriores al lanzamiento, sobre todo en lo que al juego en red se refiere.

- **Actualizaciones**: surgidas principalmente de los nuevos canales de distribución y de las nuevas plataformas de las consolas de última generación, la mayoría de los juegos actuales presentan poco después de su publicación contenido adicional creado para mejorar el videojuego. Estos contenidos, que pueden ser gratuitos o de pago, alargan la vida del juego y están generando un nuevo mercado llegando a funcionar y a comercializarse de forma independiente al juego original.

A.3. Planificación y etapas del proyecto

En primer lugar, antes de definir las etapas que componen el proyecto, qué modelo de ciclo de vida se ha aplicado. En el caso de este proyecto, se ha optado por un **Modelo de Desarrollo Evolutivo** (también conocido como prototipado evolutivo). Este es un modelo de desarrollo incremental en el que se construyen una serie de grandes versiones sucesivas del producto, en este caso del juego. Sin embargo, mientras que el modelo incremental, parte desde el principio con el conjunto completo de requisitos, el modelo evolutivo asume que los requisitos no son completamente conocidos al inicio del proyecto.

Este modelo se ajusta muy bien al este proyecto, debido a que el desarrollo se enfocó desde un principio a la creación de prototipos que fueran cubriendo los requisitos marcados y las distintas funcionalidades. Por otro lado, el hecho de que el conjunto de requisitos no tenga que ser conocido desde el principio también se ajusta, ya que tras la primera fase de análisis de este proyecto tampoco se establecieron todos. En la fase inicial se marcaron los más importantes y en las fases posteriores se detallaron el resto dependiendo de las funcionalidades que se fueran a desarrollar en el siguiente prototipo.

Por tanto, siguiendo el modelo evolutivo, los requisitos son cuidadosamente examinados, y sólo los más importantes son seleccionados para el primer incremento; construyendo una implementación parcial del sistema que recibe sólo estos requerimientos.

Esta versión parcial desarrollada se evalua, dando retroalimentación a los desarrolladores. El siguiente incremento incluye esta retroalimentación y la especificación de requisitos actualizada, dando lugar a una segunda versión del videojuego que será desarrollada y desplegada. El proceso se repite hasta completar una versión del juego que incluye todos los requisitos.

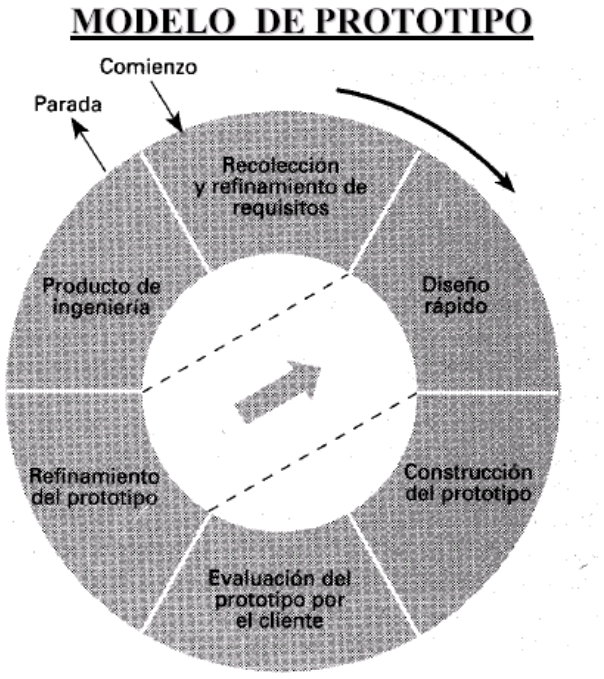

Figura 118: Modelo de Desarrollo Evolutivo o de Construcción de Prototipos

Una vez establecido el modelo del ciclo de vida se pasan a definir las fases que componen el proyecto. Como ya se ha explicado, todas las fases del modelo se aplican a cada de las versiones del prototipo que se desarrollan. En el caso de este proyecto, se han desarrollado cuatro versiones, en las que la recolección de requisitos y desarrollo de funcionalidades seguía la siguiente evolución: Generales, Jugador, Mapas, Enemigos y Otros. En primer lugar se definieron una serie de requisitos generales, que representaban aspectos básicos e importantes que el juego debía tener. Posteriormente se centró el proyecto en el personaje, controles, animación, movimientos. Una vez implementado se pasó a los mapas, casillas y diseño de niveles; siempre teniendo en cuenta además la integración con

lo ya desarrollado y las mejoras a incluir surgidas de las pruebas de los prototipos. Posteriormente se incluyeron los enemigos y por último ciertas funcionalidades adicionales como el teclado virtual o el manejo de ficheros XML. Por tanto, las tareas que componen el proyecto, indicado además su duración en días, quedan como sigue:

1. Formación previa en las tecnologías de desarrollo	**29 días**
1.1 Estudio documentación Visual Studio.NET	29 días
1.2 Estudio documentación XNA [177] [178]	29 días
2. Desarrollo de la documentación y la memoria	**130 días**
3. Fase 1 – Estructura básica del juego	**20 días**
3.1 Requisitos Fase 1	5 días
3.1.1 Aspecto básico y dimensiones	4 días
3.1.2 Desplazamiento básicos	5 días
3.1.3 Eventos simples	5 días
3.1.4 Interrupción y salida del juego	5 días
3.2 Diseño	5 días
3.3 Implementación	8 días
3.4 Evaluación	2 días
4. Fase 2 – Jugador	**25 días**
4.1 Requisitos Fase 2	5 días
4.1.1 Diseño gráfico y creación de sprites	2 días
4.1.2 Control y desplazamientos	5 días
4.1.3 Sincronización de animaciones	2 días
4.1.4 Detección y evaluación de colisiones	5 días
4.2 Diseño	5 días
4.3 Implementación	12 días
4.4 Evaluación	3 días
5. Fase 3 – Mapas y niveles	**25 días**
5.1 Requisitos Fase 3	5 días
5.1.1 Diseño gráfico y creación de sprites	2 días
5.1.2 Lógica de niveles	5 días

5.1.3 Lógica de casillas	5 días
5.2 Diseño	5 días
5.3 Implementación	12 días
5.4 Evaluación	3 días
6. Fase 4 – Enemigos	**37 días**
6.1 Requisitos Fase 4	10 días
6.1.1 Diseño gráfico y creación de sprites	4 días
6.1.2 Comportamiento y diagramas de estado	10 días
6.1.3 Detección y evaluación de colisiones	10 días
6.2 Diseño	5 días
6.3 Implementación	15 días
6.4 Evaluación	5 días
7. Fase 5 – Otros	**20 días**
7.1 Requisitos Fase 5	5 días
7.1.1 Aspecto final de pantallas	2 días
7.1.2 Teclado virtual	5 días
7.1.3 Música y efector sonoros	2 días
7.1.4 Gestión de ficheros XML	5 días
7.2 Diseño	5 días
7.3 Implementación	5 días
7.4 Evaluación	5 días
7.5 Evaluación y pruebas finales	**5 días**

Tabla 1: Tareas del proyecto

Cada una de las tareas señaladas en la tabla anterior debe tener necesariamente al menos un recurso asignado, ya sea este de tipo material o humano. La falta de disponibilidad de estos recursos puede provocar que se vea retrasado el comienzo de ciertas tareas, por lo que a menudo, para evitar estos retrasos, se debe aumentar el número de recursos, pudiendo así llevar a cabo múltiples tareas de forma paralela. Además de las tareas anteriormente nombradas a lo largo del proyecto, se llevarán a cabo reuniones periódicas de seguimiento que marcarán los hitos del proyecto (aproximadamente cada 3 o 4 semanas, según la importancia de la fase).

A continuación se muestra un pantallazo de Microsoft Project, indicando las tareas y su fecha de comienzo y fin correspondientes; además de los recursos asignados a cada una.

Diagrama de Gantt

	Nombre de tarea	Duración	Comienzo	Fin	Nombres de los recursos
1	**Formación previa en las tecnologías de desarrollo**	**29 días**	**mar 24/02/09**	**vie 03/04/09**	**PC[1];Programador**
2	Estudio documentación Visual Studio.NET	29 días	mar 24/02/09	vie 03/04/09	Programador
3	Estudio documentación XNA	29 días	mar 24/02/09	vie 03/04/09	Programador
4	Desarrollo de la documentación y la memoria	130 días	lun 06/04/09	vie 02/10/09	Programador[10%];PC[1]
5	**Fase 1 – Estructura básica del juego**	**20 días**	**lun 06/04/09**	**vie 01/05/09**	**PC[1]**
6	**Requisitos Fase 1**	**5 días**	**lun 06/04/09**	**vie 10/04/09**	
7	Aspecto básico y dimensiones	4 días	lun 06/04/09	jue 09/04/09	Artista conceptual;Adobe Photoshop 7.0[1]
8	Desplazamiento básicos	5 días	lun 06/04/09	vie 10/04/09	Analista
9	Eventos simples	5 días	lun 06/04/09	vie 10/04/09	Analista
10	Interrupción y salida del juego	5 días	lun 06/04/09	vie 10/04/09	Analista
11	Diseño	5 días	lun 13/04/09	vie 17/04/09	Analista
12	Implementación	8 días	lun 20/04/09	mié 29/04/09	Programador
13	Evaluación	2 días	jue 30/04/09	vie 01/05/09	Testeador
14	Reunión	2 horas	vie 01/05/09	vie 01/05/09	Analista;Jefe de proyecto
15	**Fase 2 – Jugador**	**25 días**	**lun 04/05/09**	**vie 05/06/09**	**PC[1]**
16	**Requisitos Fase 2**	**5 días**	**lun 04/05/09**	**vie 08/05/09**	
17	Diseño gráfico y creación de sprites	2 días	lun 04/05/09	mar 05/05/09	Artista conceptual;Adobe Photoshop 7.0[1]
18	Control y desplazamientos	5 días	lun 04/05/09	vie 08/05/09	
19	Sincronización de animaciones	2 días	lun 04/05/09	mar 05/05/09	
20	Detección y evaluación de colisiones	5 días	lun 04/05/09	vie 08/05/09	Analista
21	Diseño	5 días	lun 11/05/09	vie 15/05/09	Analista
22	Implementación	12 días	lun 18/05/09	mar 02/06/09	Programador
23	Evaluación	3 días	mié 03/06/09	vie 05/06/09	Testeador
24	Reunión	2 horas	vie 05/06/09	vie 05/06/09	Analista;Jefe de proyecto
25	**Fase 3 – Mapas y niveles**	**25 días**	**lun 08/06/09**	**vie 10/07/09**	**PC[1]**
26	**Requisitos Fase 3**	**5 días**	**lun 08/06/09**	**vie 12/06/09**	
27	Diseño gráfico y creación de sprites	2 días	lun 08/06/09	mar 09/06/09	Artista conceptual;Adobe Photoshop 7.0[1]
28	Lógica de niveles	5 días	lun 08/06/09	vie 12/06/09	
29	Lógica de casillas	5 días	lun 08/06/09	vie 12/06/09	Analista
30	Diseño	5 días	lun 15/06/09	vie 19/06/09	Analista
31	Implementación	12 días	lun 22/06/09	mar 07/07/09	Programador
32	Evaluación	3 días	mié 08/07/09	vie 10/07/09	Testeador
33	Reunión	2 horas	vie 10/07/09	vie 10/07/09	Analista;Jefe de proyecto
34	**Fase 4 – Enemigos**	**35 días**	**lun 13/07/09**	**vie 28/08/09**	**PC[1]**
35	**Requisitos Fase 4**	**10 días**	**lun 13/07/09**	**vie 24/07/09**	
36	Diseño gráfico y creación de sprites	4 días	lun 13/07/09	jue 16/07/09	Adobe Photoshop 7.0[1];Artista conceptual
37	Comportamiento y diagramas de estado	10 días	lun 13/07/09	vie 24/07/09	
38	Detección y evaluación de colisiones	10 días	lun 13/07/09	vie 24/07/09	Analista
39	Diseño	5 días	lun 27/07/09	vie 31/07/09	Analista
40	Implementación	15 días	lun 03/08/09	vie 21/08/09	Programador
41	Evaluación	5 días	lun 24/08/09	vie 28/08/09	Testeador
42	Reunión	2 horas	vie 28/08/09	vie 28/08/09	Analista;Jefe de proyecto
43	**Fase 5 – Otros**	**20 días**	**lun 31/08/09**	**vie 25/09/09**	**PC[1]**
44	**Requisitos Fase 5**	**5 días**	**lun 31/08/09**	**vie 04/09/09**	
45	Aspecto final de pantallas	2 días	lun 31/08/09	mar 01/09/09	Artista conceptual;Adobe Photoshop 7.0[1]
46	Teclado virtual	5 días	lun 31/08/09	vie 04/09/09	
47	Música y efector sonoros	2 días	lun 31/08/09	mar 01/09/09	
48	Gestión de ficheros XML	5 días	lun 31/08/09	vie 04/09/09	Analista
49	Diseño	5 días	lun 07/09/09	vie 11/09/09	Analista
50	Implementación	5 días	lun 14/09/09	vie 18/09/09	Programador
51	Evaluación	5 días	lun 21/09/09	vie 25/09/09	Testeador
52	Reunión	2 horas	vie 25/09/09	vie 25/09/09	Analista;Jefe de proyecto
53	Evaluación y pruebas finales	5 días	lun 28/09/09	vie 02/10/09	Programador;Testeador
54	Reunión final	2 horas	vie 02/10/09	vie 02/10/09	Analista;Artista conceptual;Programador;Testeador;Jefe de proyecto

Figura 119: Diagrama de Gantt – Tareas

A continuación se muestra el diagrama de Gantt mostrando cada una de las tareas, indicando su evolución en el tiempo y sus dependencias.

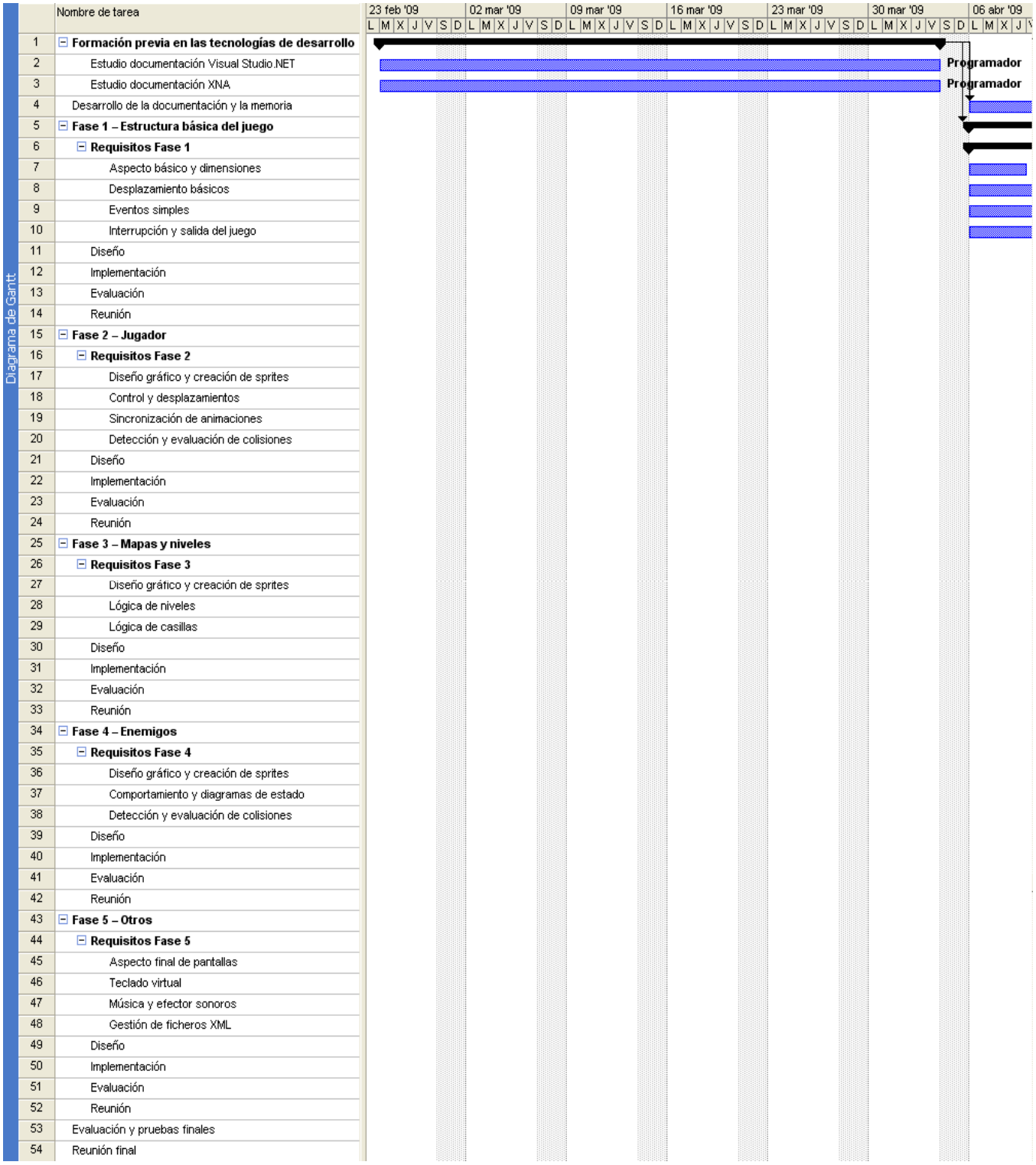

Figura 120: Diagrama de Gantt 1

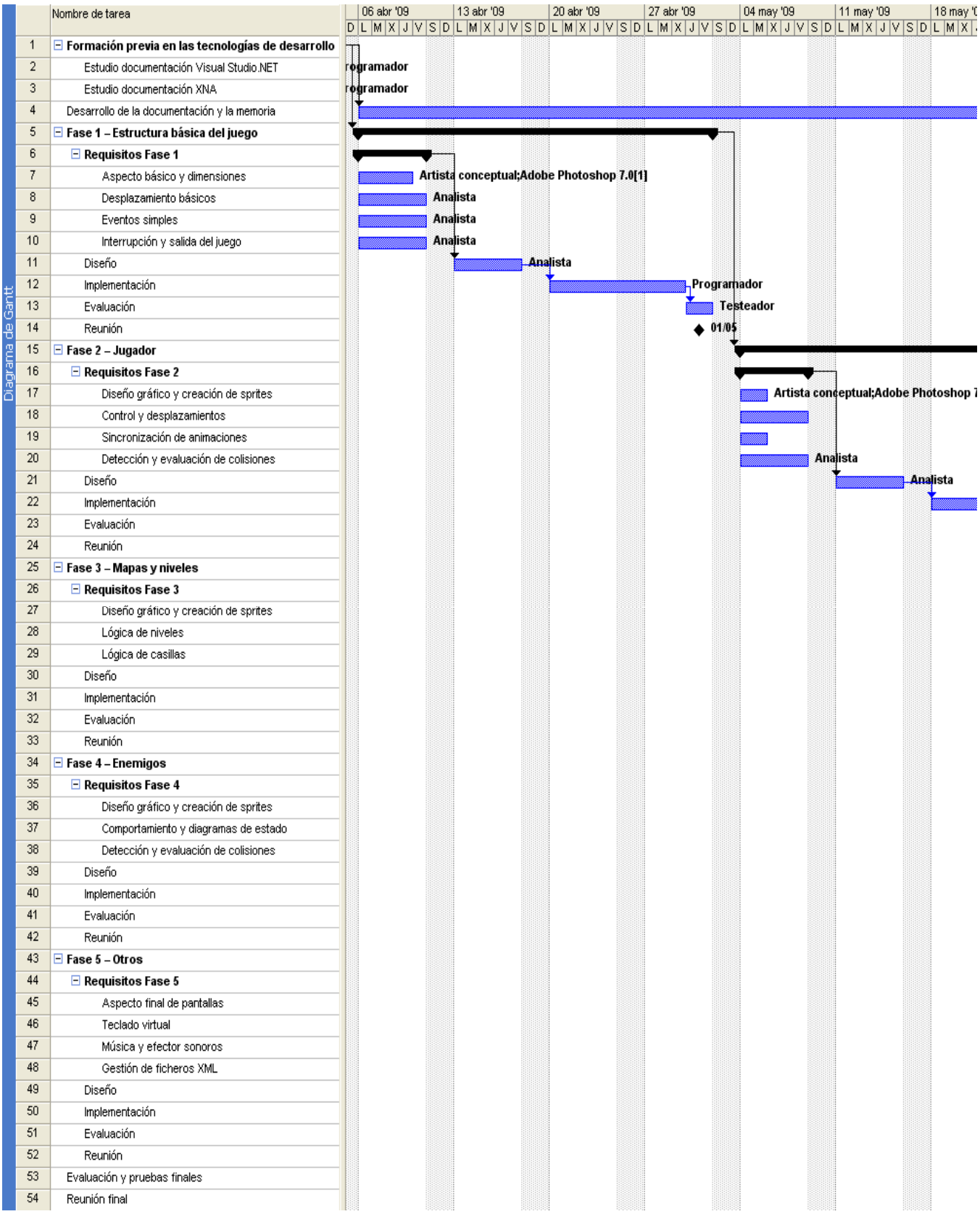

Figura 121: Diagrama de Gantt 2

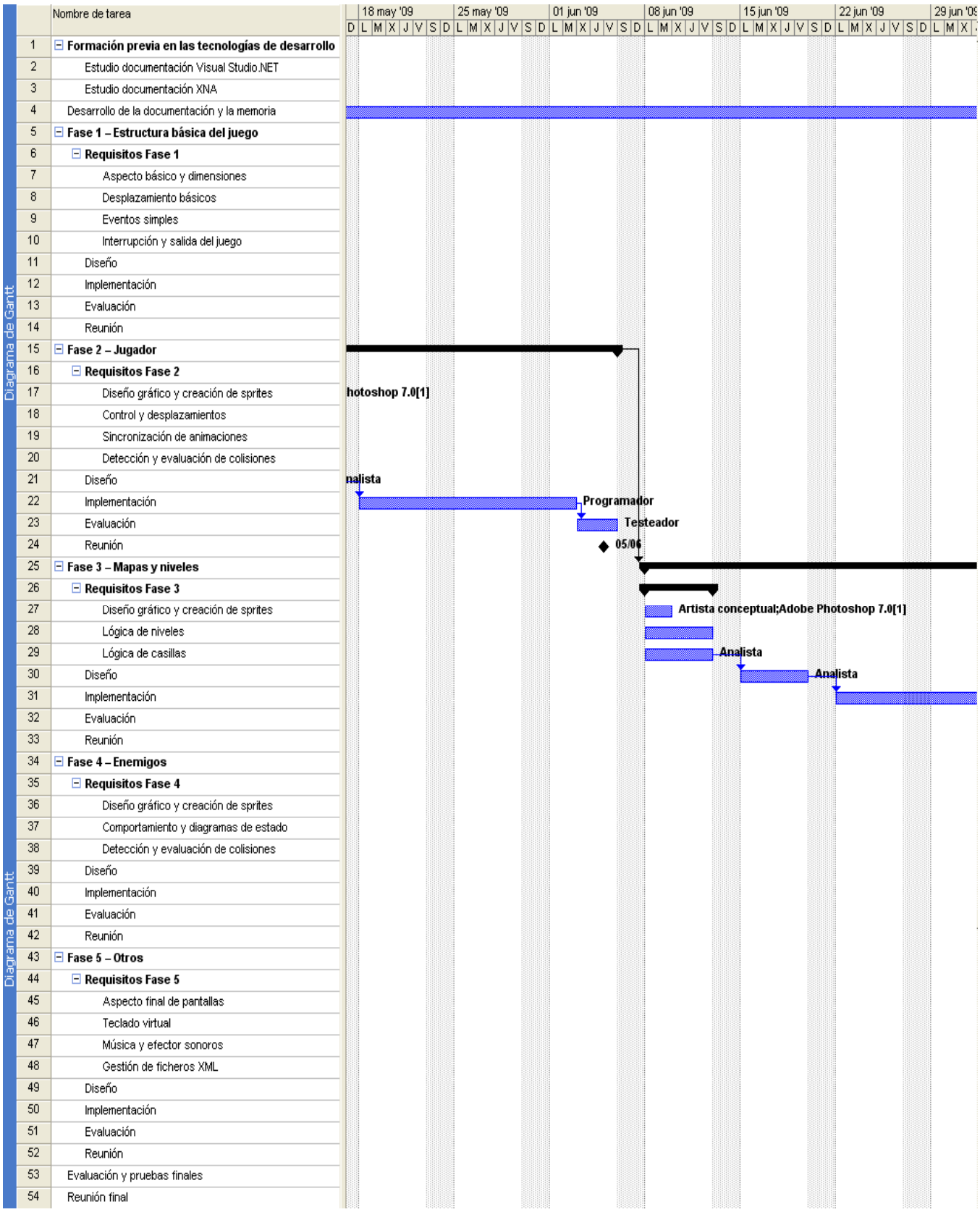

Figura 122: Diagrama de Gantt 3

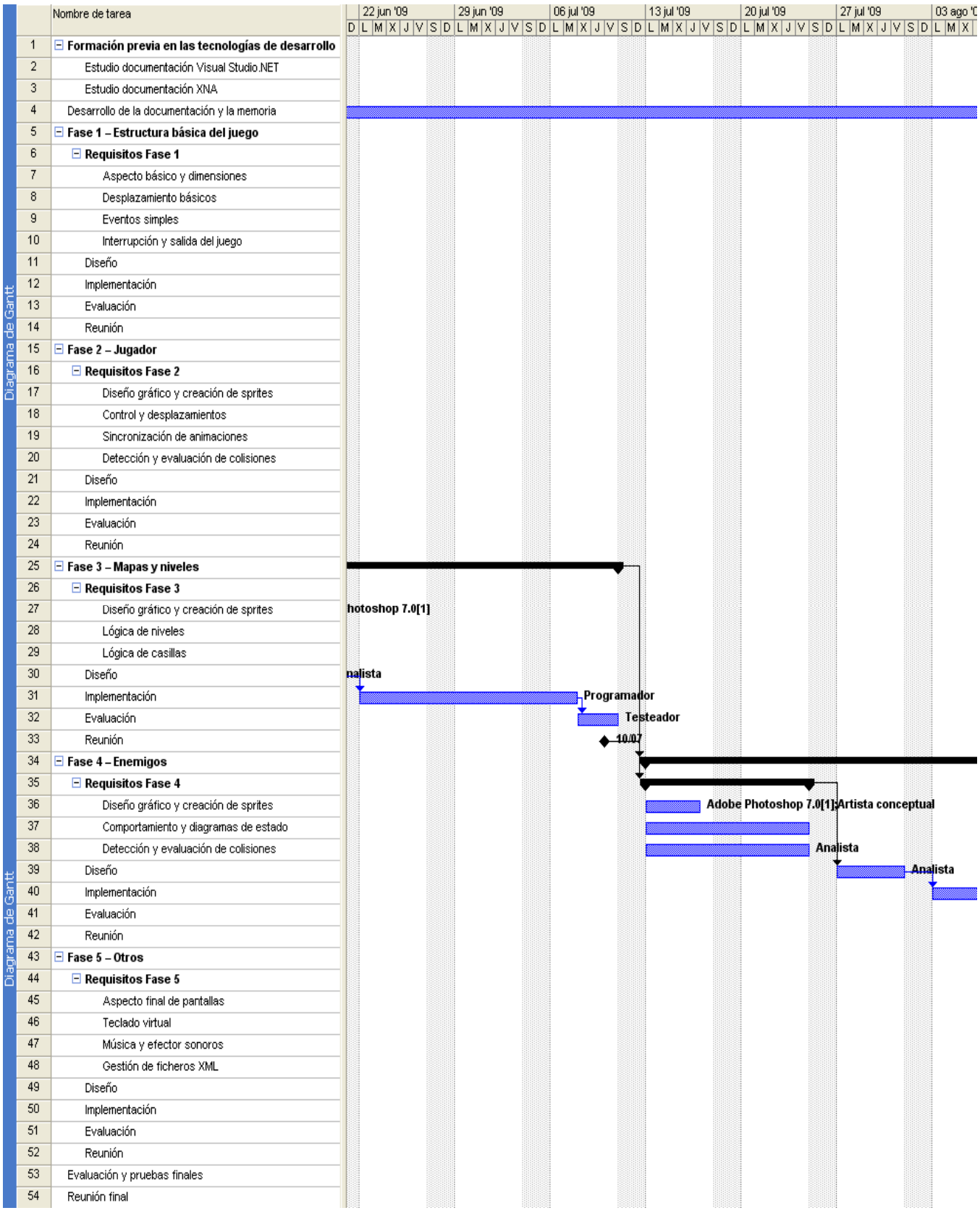

Figura 123: Diagrama de Gantt 4

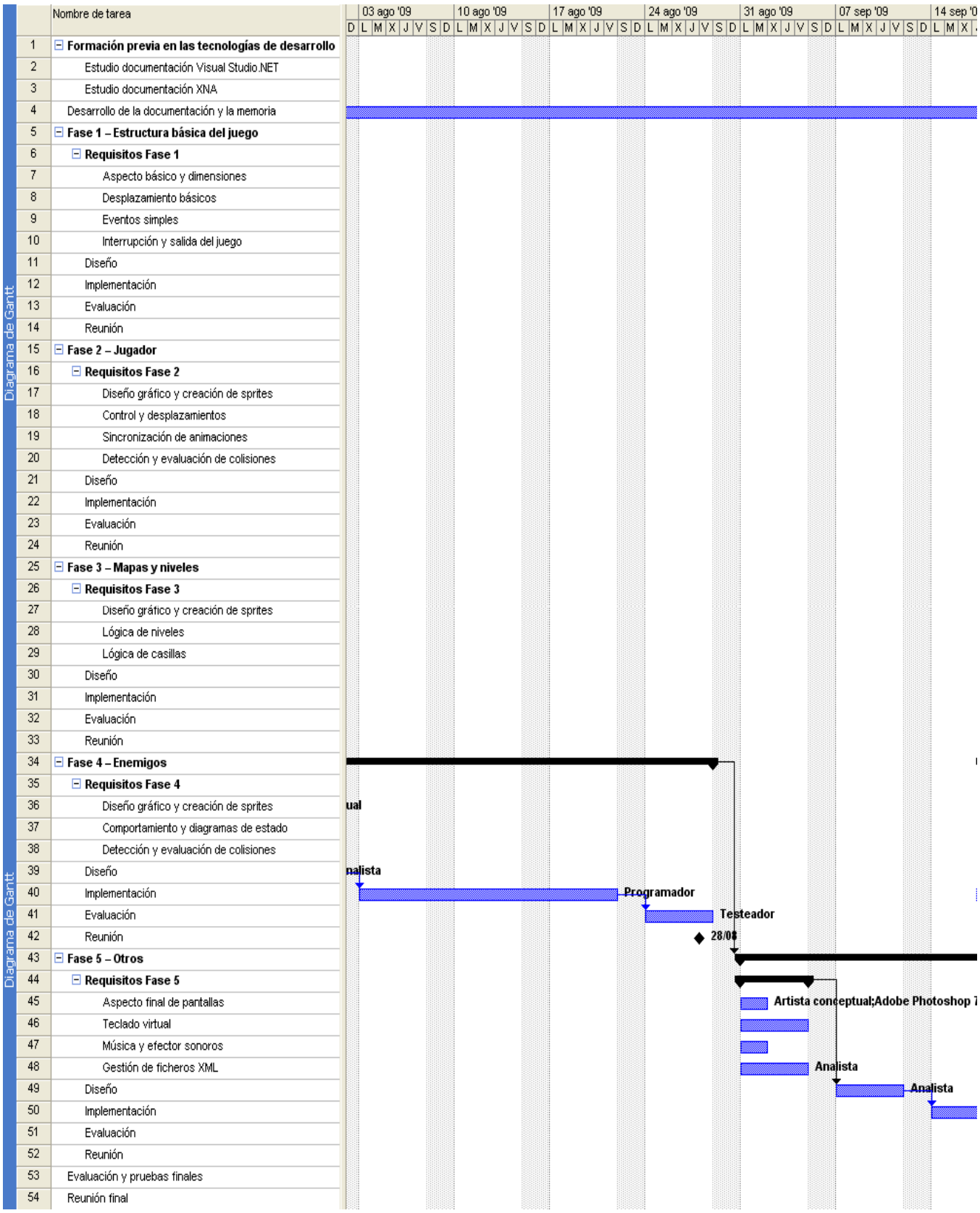

Figura 124: Diagrama de Gantt 5

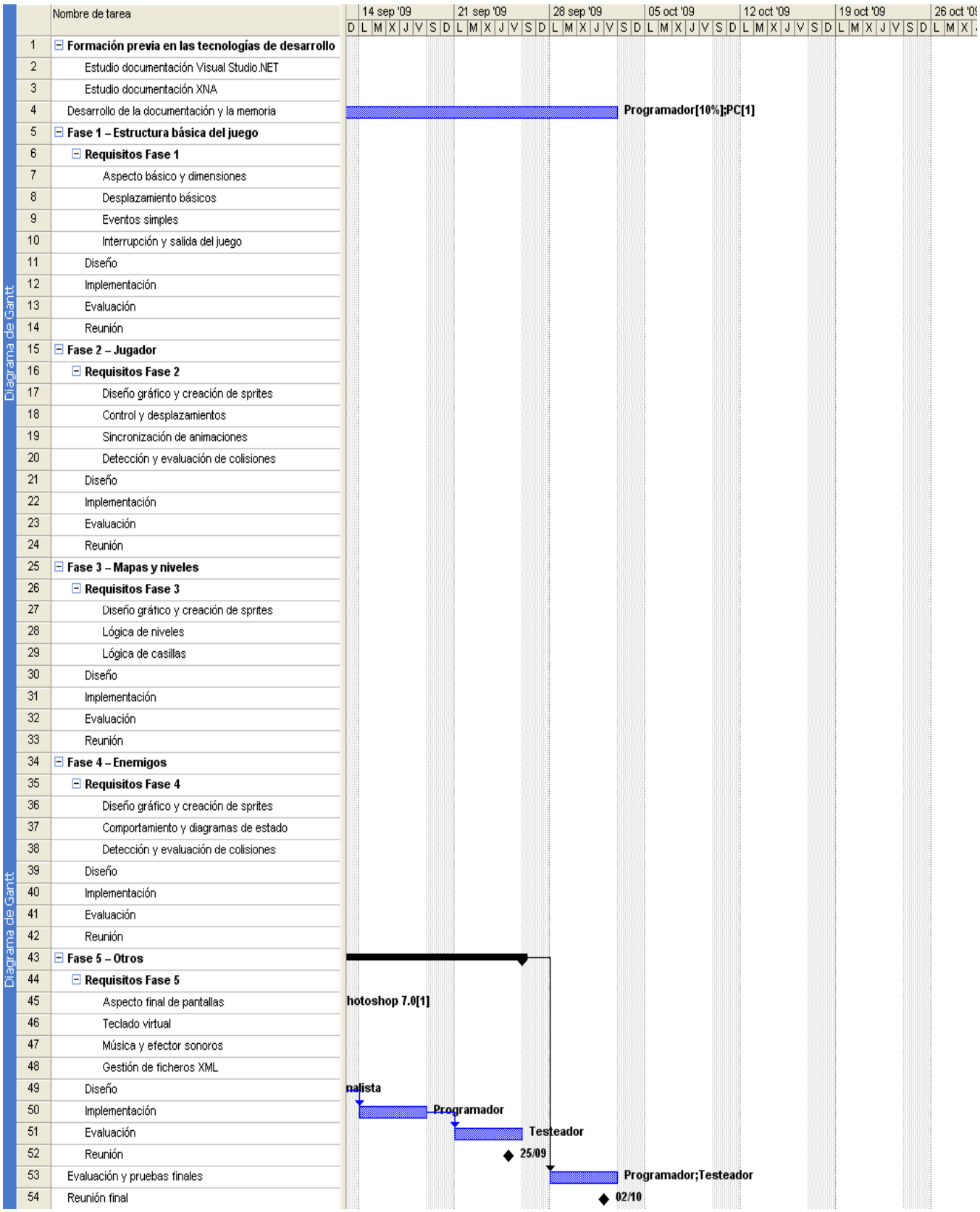

Figura 125: Diagrama de Gantt 6

A.4. Presupuesto del proyecto

A continuación se hará una estimación del proyecto en función de los recursos humanos y materiales necesarios para llevarlo a cabo. El presupuesto es un factor clave para el estudio de la viabilidad. El presupuesto se realiza asumiendo unas fechas y condiciones determinadas, éstas se suponen similares a las que realmente se va a encontrar quien realiza el proyecto durante el desarrollo del mismo. No se debe olvidar que cualquier variación de estas condiciones reales respecto de lo planeado puede suponer una modificación importante del coste inicialmente previsto.

A.4.1 Estimación

La estimación recoge de forma ordenada los recursos necesarios para la ejecución del proyecto, ya sean recursos materiales o recursos humanos. En el caso de los recursos humanos, se proporciona el número de horas de trabajo de cada persona necesarias para llevar a acabo la ejecución del proyecto. En el caso de loa materiales o equipamiento se proporciona el número de unidades necesarias. La siguiente tabla muestra los datos correspondientes.

Recurso Humano	**Trabajo**
Jefe de proyecto	12 horas
Analista	272 horas
Artista conceptual	58 horas
Programador	630 horas
Testeador	94 horas
Recurso Material	**Unidades**
PC con Windows XP	1
Suscripción XNA Creators Club	1
Adobe Photoshop 7.0	1

Tabla 2: Recursos asociados al proyecto: horas y número de unidades

Además debemos tener en cuenta el precio de cada recurso, reflejando en la estimación el coste de la hora de trabajo del empleado, o en el caso de recursos tangibles, los precios de los materiales o equipos utilizados en el proyecto. La siguiente tabla hace referencia a estos datos.

Recurso Humano	€ Brutos	€/Hora
Jefe de proyecto	48.406,00	23,08
Analista	34.758,00	16,71
Artista conceptual	19.526,00	9,31
Programador	20.911,00	10,05
Testeador	20.848,00	10,02
Recurso Material	**-**	**€/Unidad**
PC con Windows XP	-	750,00
Suscripción XNA Creators Club	-	72,50
Adobe Photoshop 7.0	-	17,00

Tabla 3: Recursos asociados al proyecto: costes

La suscripción a XNA Creators Club es necesaria para poder publicar el juego en el Bazar de XBOX 360. Este proceso se comentará al final de este punto.

A.4.2 Presupuesto final

Supone la valoración definitiva del coste del proyecto. A continuación se muestra una tabla en la que se recoge el coste asociado a cada fase del proyecto y el coste total.

Fases del proyecto	Coste €
Formación previa en las tecnologías de desarrollo	3.497,40
Desarrollo de la documentación y la memoria	522,60
Fase 1 – Estructura básica del juego	1.967,10
Fase 2 – Jugador	1.425,10
Fase 3 – Mapas y niveles	1.425,10
Fase 4 – Enemigos	2.034,54
Fase 5 – Otros	1.223,86
Evaluación y pruebas finales	401,40
Reunión final	138,34
Total fases €	12.635,44
Total recurso materiales €	839,50
Total del Proyecto €	13.474,94

Tabla 4: Recursos asociados al proyecto: costes

El coste total del proyecto asciende a 13.474,94 € y su duración se extiende un total de 159 días.

A.5. Publicación del juego

El último paso de este proyecto es la publicación del juego. En este caso se ha seleccionado el bazar de *XBOX 360* como plataforma óptima para el lanzamiento, puesto que con el pago de la suscripción de *XNA Creators Club* es posible colgar todos los juegos que se desee en el bazar durante un año, proporcionando una forma relativamente rápida de distribución.

La publicación no es inmediata. Una vez desarrollado el juego el siguiente paso es enviar una versión para revisión. El juego debe estar correctamente etiquetado, si no lo está será rechazado automáticamente. Los datos necesarios para subir una revisión son los siguientes: información del juego, un binario del juego, los países en los que se publicará, posibles comentarios y una indicación de si el juego es una versión total o una demo. Los juegos con archivos binarios inferiores a 50 MB de tamaño se pueden adquirir con 80, 240, 400 u 800 puntos. Los juegos con archivos binarios iguales o mayores a 50 MB de tamaño se pueden adquirir con 240, 400 u 800 puntos

El proceso de revisión lo pueden llevar a cabo testeadores de Microsoft u otros usuarios del Creators Club. Una vez que el juego haya pasado las pruebas pertinentes (que consisten en un testeo del código para evitar que contenga errores, una evaluación del material para ver que no incumple derechos de autor y una evaluación de contenido para comprobar que no es ofensivo) el juego pasa a formar parte del bazar.

A partir de aquí el juego empezará a generar beneficios pero siempre bajo la normativa de *Microsoft*, a saber: todos los cánones se calculan tomando el valor Puntos del juego y multiplicándolo por el número de unidades vendidas. Se aplica una división porcentual a esa base, y se realizan los ajustes promocionales. En ese momento, los puntos se convertirán en dólares estadounidenses. Cuando un creador alcanza el límite mínimo de pago, se produce una conversión de moneda a la divisa local del creador. Esto significa que el juego debe vender un número mínimo de copias para empezar a generar dinero, ya que el valor de todas las copias anteriores a ese número pertenece a Microsoft. Por otro lado *Microsoft* se queda con un porcentaje de cada copia vendida a partir del mínimo en concepto de derechos de promoción.

El impuesto promocional exacto será de entre un 10 y un 30 por ciento para todos los juegos, aunque no hay establecido un porcentaje exacto, ya que se fija el primer trimestre de cada año. Por otro lado, el límite de pago mínimo que establece *Microsoft* es de 150 dólares estadounidenses. Si no se alcanza este mínimo, no se ganará nada. Según el precio que se elija para el juego, el mínimo se establecerá por cuatrimestres, semestres o por años.

Teniendo en cuenta estos factores, lo recomendable es fijar un valor inicial de 240 *Microsoft Points* (equivalentes a 2,40 dólares) para el primer semestre y posteriormente reducirlo a 80 Microsoft Points (equivalentes a 1 dólar) una vez que el juego haya alcanzado algo de fama en la comunidad. Por otro lado se puede utilizar el sistema de ***Tokens***[46] (códigos para adquirir el juego de manera gratuita) para hacer publicidad en revistas y blogs gastando lo mínimo posible.

Según los datos de *Microsoft*, cerca del 70% de los jugadores de *XBOX 360* descargan juegos arcade. La media de descargas por usuario es de 7. En cuanto a los datos de descarga por juego, la media de un juego de éxito es de 350.000 en los dos primeros meses, recuperando un 156% de la inversión en el primer año.

A continuación, se muestran las cifras mínimas de descarga necesarias para rentabilizar el juego en un periodo de dos años (cuatro semestres).

- El coste total del juego es de 13.474,94 €, equivalentes a 18.279,81 $. Para realizar unos cálculos aproximados, se pretende recuperar ¼ de la inversión en cada semestre, es decir, 4579,95 $.

- El primer semestre el juego tiene un valor de de 240 *Microsoft Points* (equivalentes a 2,40 $)

    - Para superar el límite de pago mínimo (150 $) se deben realizar 63 descargas, de las que no se obtiene nada.
    - Una vez superado el límite de pago mínimo, del resto de copias se debe restar aproximadamente un 15% del valor que se queda *Microsoft* en derechos de promoción. De cada copia se obtendrán 2,04 $.
    - Para alcanzar los 4579,95 $ deben producirse 2246 descargas más.

- A partir del segundo semestre, el juego costará 80*Microsoft Points* (equivalentes a 1 $)

    - Para superar el límite de pago mínimo (150 $) se deben realizar 177 descargas.
    - Restando el 15% de derechos, de cada copia se obtendrán 0,85 $.
    - Para alcanzar los 4579,95 $ deben producirse 5390 descargas más.

Esto es sólo una estimación de cómo se rentabilizaría en juego en dos años. Las cifras varían bastante si el número de copias aumenta el primer semestre, puesto que el precio del juego es casi el triple en comparación con los semestres posteriores. De hecho según *Microsoft*, los primeros dos meses se suele recuperar el 35% de la inversión total en un juego arcade. De modo que, en el primer semestre, se podría recuperar gran parte de la inversión realizada obteniendo beneficios mucho antes.

Anexo B

Manual de usuario

B.1. Instalación

El juego *UC3M Bomber* no precisa de instalación, pero es necesario tener instalados una serie de componentes para su correcto funcionamiento. Es posible jugar al juego en cualquier PC Windows que tenga instalados los siguientes componentes:

- .Net Framework redistribuible (última versión disponible)
- XNA Framework redistribuible (última versión disponible)
- DirectX instalado (8.1 o superior, preferiblemente 9.0c)
- Una tarjeta gráfica con shaders 1.1 o superior

Una vez instalados, bastará con ejecutar el archivo *UC3M Bomber.exe* para lanzar el juego.

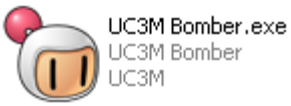

Figura 126: Ejecutable UC3M Bomber.exe

B.2. Pantallas

Una vez lanzado *UC3M Bomber* aparecerá el **Menú principal**. Para desplazarse por los menús se deben utilizar las *flechas de desplazamiento*. Para seleccionar una opción de un menú se utiliza la tecla *Enter*. Para salir de cualquier menú se utiliza la tecla *Escape*. Las opciones que se presentan en el **Menú Principal** son:

- Jugar
- Opciones

- Puntuaciones
- Salir

La opción Jugar lanza una nueva partida. Opciones abre el **Menú de Opciones**. Puntuaciones muestra el **Menú de Puntuaciones** y Salir cierra el juego. A continuación se muestra la pantalla de **Menú Principal**.

Figura 127: Pantalla de Menú Principal

Desde el **Menú de Opciones** es posible modificar los siguientes parámetros del juego:

- **Dificultad**: pudiendo seleccionar entre Fácil y Normal.
- **Soltar Bombas**: pudiendo seleccionar entre la tecla Espacio y Enter.
- **Activar Analógico**: para activar el control analógico en un mando de XBOX 360.
- **Sensibilidad del Analógico**: asigna la sensibilidad del analógico con un valor entre 1 y 10.
- **Volumen**: para asignar un valor al volumen entre 1 y 10, o Off para desactivar (afecta a la música y los efectos sonoros).
- **Música**: seleccionando On/Off para activar y desactivar (afecta sólo a la música).
- **Valores por defecto**: asigna los valores por defecto del juego a las opciones anteriores.
- **Salir y guardar**: salir guardando los cambios (para salir sin guardar se pulsa *Escape*).

A continuación se muestra la pantalla de **Menú de Opciones**.

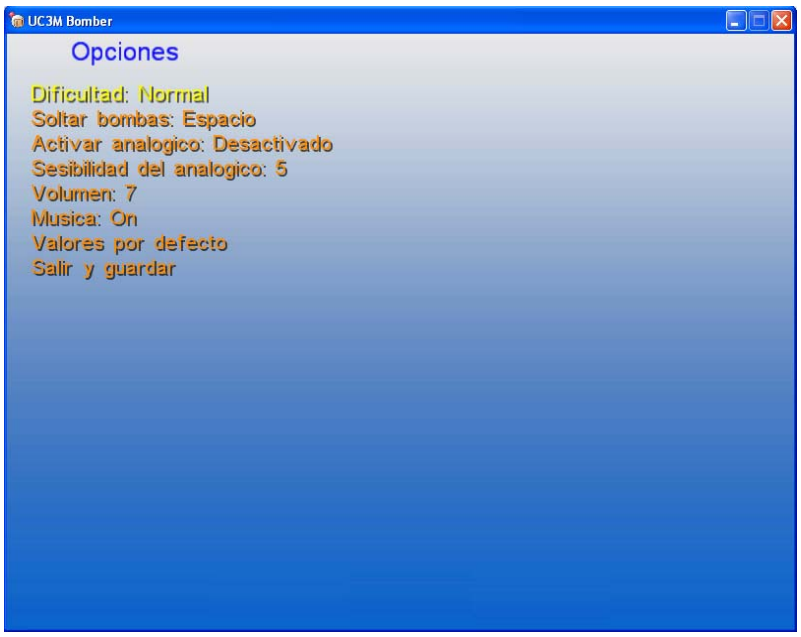

Figura 128: Pantalla de Menú de Opciones

Desde el **Menú de Puntuaciones** es visualizar un histórico con las cinco puntuaciones más altas obtenidas en el juego, el nivel en el que se alcanzaron y el usuario que las consiguió. A continuación se muestra la pantalla de **Menú de Puntuaciones**.

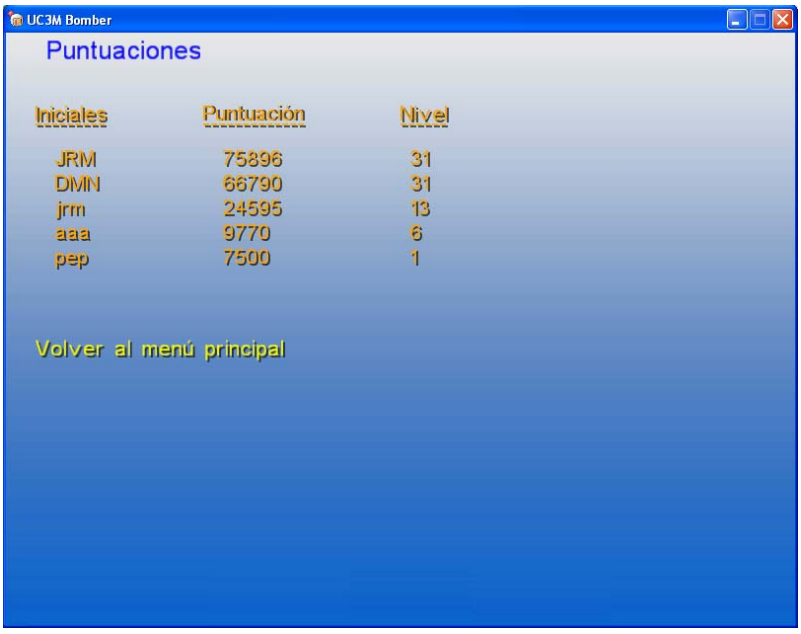

Figura 129: Pantalla de Menú de Puntuaciones

Si se selecciona la opción Jugar, desde el **Menú Principal**, comienza una nueva partida.

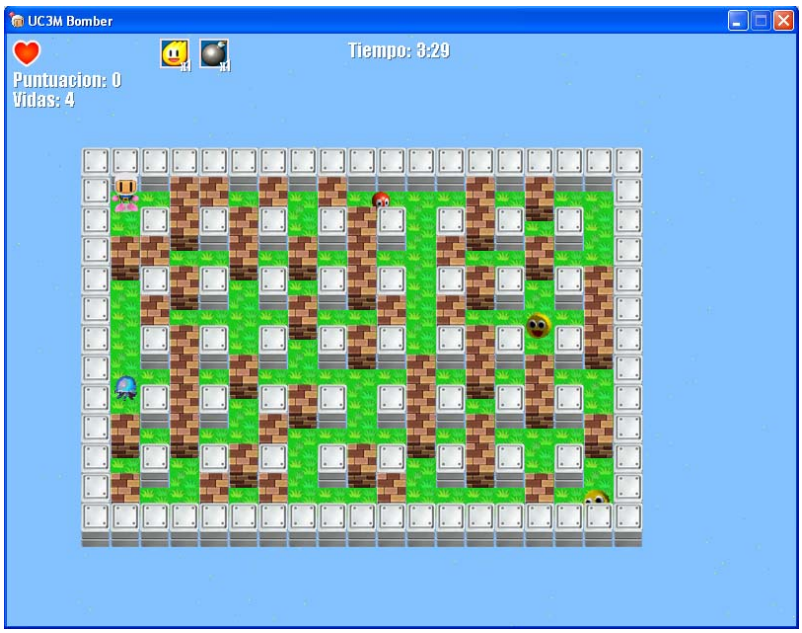

Figura 130: Pantalla de juego

Si estando en la en la pantalla de juego, se pulsa más de tres segundo Escape, se mostrará la pantalla de Pausa, desde la que es posible seguir jugando o salir al **Menú Principal**.

Figura 131: Pantalla de juego

Si durante el juego, el jugador pierde una vida, aparecerá la siguiente pantalla, desde la que se puede retornar al juego o volver al **Menú Principal**.

Figura 132: Pantalla de muerte

Si durante el juego, el jugador pierde todas las vidas vida, aparecerá la pantalla de game over, desde la que se introducen las iniciales de jugador y se retornar al **Menú Principal**.

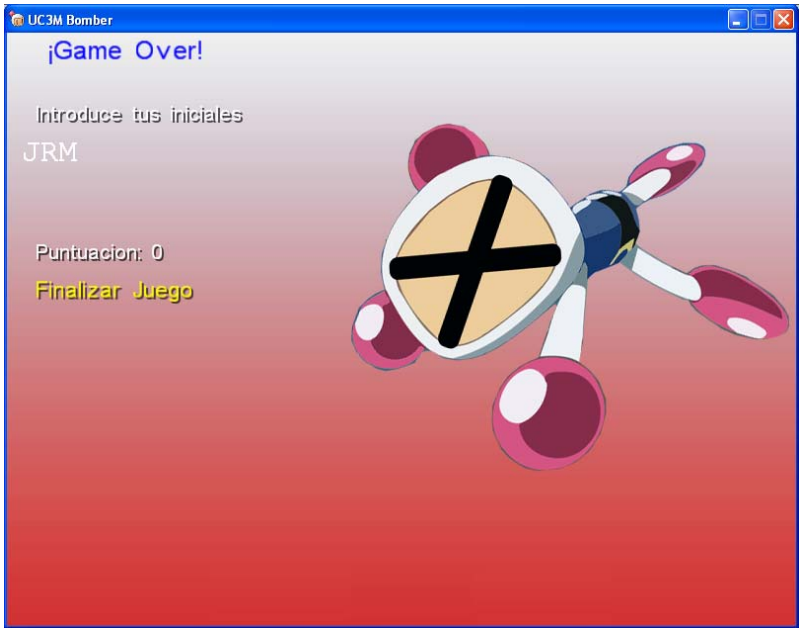

Figura 133: Pantalla de Game Over

Si durante el juego, el jugador pierde una vida porque se acaba el tiempo, aparecerá la siguiente pantalla, desde la que se puede retornar al juego o volver al **Menú Principal**.

Figura 134: Pantalla de Time Out

Si se consiguen superar todos los niveles, aparecerá la pantalla de victoria, en la que el jugador podrá introducir sus iniciales para almacenar su record.

Figura 135: Pantalla de Victoria

B.3. Pantalla de juego y niveles

Como ya se ha indicado, si se selecciona la opción Jugar, desde el **Menú Principal**, comienza una nueva partida. El nivel cero es de entrenamiento. Si se dominan los controles del juego se recomienda utilizar la salida y pasar al primer nivel. La pantalla de juego muestra al jugador en el nivel, indicando la información relevante para el jugador durante el juego. A continuación se muestra la pantalla de juego, señalando con número cada uno de los valores mostrados.

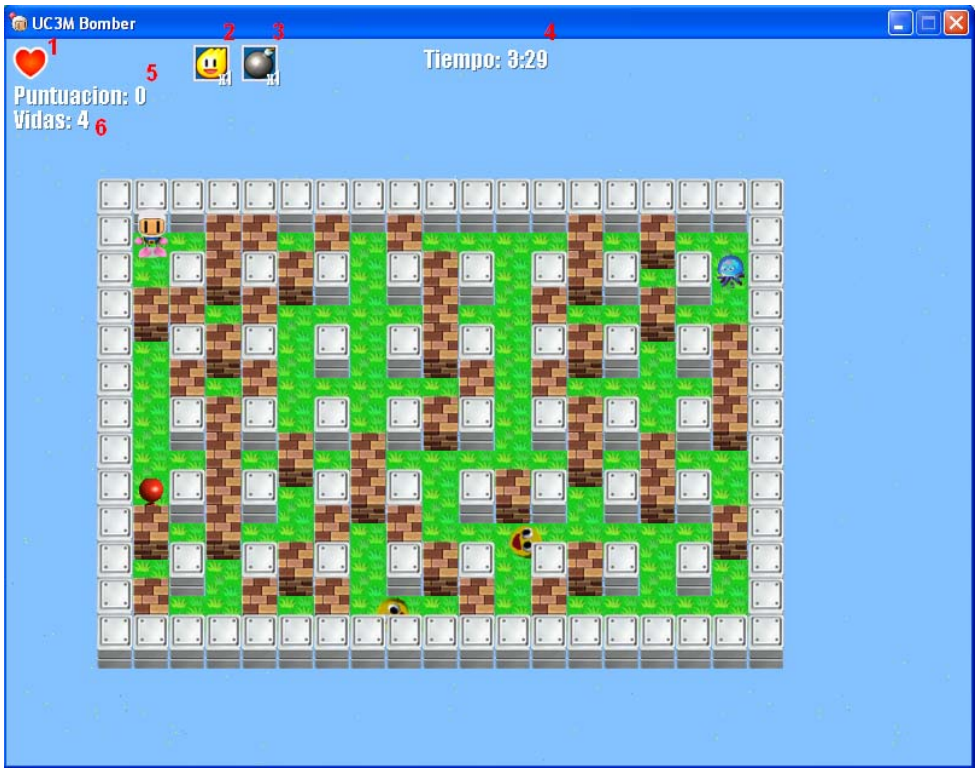

Figura 136: Pantalla de juego

1. Corazones actuales: número de veces que puede ser alcanzado el jugador antes de perder la vida actual.
2. Potencia actual: alcance en casillas de las bombas.
3. Número de bombas: número de bombas que el jugador puede soltar de forma simultánea en el mapa.
4. Tiempo: tiempo restante para encontrar la salida.

5. Puntuación: puntuación acumulada.
6. Vidas: número de vidas restantes.

Como se ha comentado, el objetivo del juego es encontrar la salida para poder escapar del laberinto de cada nivel. Para ello debe buscarla destruyendo los bloques rompibles de cada nivel. Si hay enemigos en el nivel, la salida estará cerrada; por lo que para desbloquearla, el jugador debe acabar con todos los enemigos del mapa antes de finalizar el tiempo. A continuación se muestra el conjunto de casillas correspondiente a cada mapa (existen tres tipos de mapas con 10 niveles cada uno, más uno adicional con el nivel del jefe final).

Figura 137: Suelo Mapa 1

Figura 138: Bloque no rompible Mapa 1

Figura 139: Bloque rompible Mapa 1

Figura 140: Suelo Mapa 2

Figura 141: Bloque no rompible Mapa 2

Figura 142: Bloque rompible Mapa 2

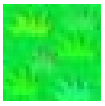

Figura 143: Suelo Mapa 3

Figura 144: Bloque no rompible Mapa 3

Figura 145: Bloque rompible Mapa 3

Figura 146: Suelo Mapa Jefe

Figura 147: Bloque no rompible Mapa Jefe

Figura 148: Casilla de salida cerrada

Figura 149: Casilla de salida abierta

Recordar que la casilla de salida se encuentra escondida bajo uno de los bloques rompibles de cada nivel. Además de la salida, dentro de cada bloque se pueden encontrar varios tipos de Power Up que proporcionan mejoras, temporales o permanentes al jugador. A continuación se muestran los ítems que se pueden encontrar en el juego.

Figura 150: Bomb Up

Figura 151: Remote Bomb

Figura 152: Fire Up

Figura 153: Heart

- **Bomb Up**: incrementa en uno el número de bombas simultáneas que el jugador puede soltar.
- **Remote Bomb**: proporciona la capacidad de detonar las bombas.
- **Fire Up**: aumenta en uno el alcance de las explosiones.
- **Heart**: añade un corazón más al jugador, permitiéndole ser alcanzado por una explosión o un enemigo.

B.4. Enemigos

En *UC3M Bomber* hay tres tipos de enemigos, además del jefe final del último nivel. A continuación se muestra cada tipo de enemigo, su valor en puntos, su velocidad y su nivel de dificultad.

Enemigo				
Nombre	Enemigo_B	Enemigo_C	Enemigo_S	Jefe
Valor	100	250	500	1000
Velocidad	0.5	1	0.5/1	0.5/1.5
Vidas	1	1	2	4
Dificultad	Baja	Media	Alta	Muy Alta

Tabla 5: Enemigos

El Jefe final, cada vez que es alcanzado por una bomba, genera dos nuevos enemigos de la mitad de tamaño. Cada uno de estos dos enemigos nuevos generará a su vez dos enemigos de la mitad de tamaño al ser alcanzados; así hasta perder cuatro vidas.

B.5. Controles

Por último se muestran los controles en juego de *UC3M Bomber*. Si el jugador maneja al personaje con el teclado los controles serán los siguientes:

- **Flechas de dirección**: mueven al jugador arriba/abajo/izquierda/derecha.
- **Enter**: deposita bomba o la detona, según se seleccione en las opciones.
- **Barra de espacios**: deposita bomba o la detona, según se seleccione en las opciones.
- **Escape**: si se pulsa durante más de tres segundos, muestra la pantalla de pausa.

Si el jugador maneja al personaje con un mando de XBOX 360:

- **Cruceta**: mueven al jugador arriba/abajo/izquierda/derecha.
- **Analógico:** mueven al jugador arriba/abajo/izquierda/derecha, si se activa en las opciones.
- **Botón A**: deposita bomba o la detona, según se seleccione en las opciones.
- **Botón B**: deposita bomba o la detona, según se seleccione en las opciones.
- **Botón Start**: si se pulsa durante más de tres segundos, muestra la pantalla de pausa.

B.6. Niveles de dificultad

Desde el Menú de Opciones es posible seleccionar el nivel de dificultad del juego de entre dos posibles: Fácil y Normal. Los parámetros que se modifican quedan como sigue según el nivel:

- **Nivel de dificultad Fácil**

  - El tiempo para superar cada nivel es de 240 segundos
  - El ratio de aparición de Power Ups es de un 10%
  - Al perder una vida el jugador perderá una unidad de potencia en el alcance de sus bombas si es mayor que uno
  - Al perder una vida el jugador perderá una bomba en el máximo de bombas que puede soltar de forma simultánea
  - El número de corazones iniciales es dos
  - El número de vidas iniciales es cinco

- **Nivel de dificultad Normal**

  - El tiempo para superar cada nivel es de 210 segundos
  - El ratio de aparición de Power Ups es de un 5%
  - Al perder una vida, la potencia de alcance de las bombas del jugador se reiniciará a uno
  - Al perder una vida el jugador el número máximo de bombas que el jugador puede soltar de forma simultánea se reiniciará a uno
  - El número de corazones iniciales es uno
  - El número de vidas iniciales es cuatro

Diccionario de términos y acrónimos

[1] **Videojuego comercial.** Se consideran videojuegos comerciales aquellos publicados, producidos o simplemente respaldados por grandes firmas dedicadas al ocio electrónico como *Nintendo*, *SEGA*, *Sony*, *Microsoft*, *Konami* etc y cuyo desarrollo es totalmente profesional. [↑]

[2] **Videojuego casual.** El videojuego casual u ocasional es aquel que está dirigido al grupo de jugadores no tradicionales, usuarios relativamente nuevos, que dedican pocas horas al juego, o que conciben los videojuegos como una forma adicional de ocio. En cuanto a temática, los juegos casuales suelen ser juegos deportivos, sociales o de lógica mental. Presentan reglas simples y jugabilidad fácil, siendo el objetivo principal brindar una experiencia del tipo "*pick up and play*" (poner y jugar), orientándose a usuarios de cualquier edad y nivel de habilidad. [↑]

[3] **Videojuego bajo demanda.** Sistema de publicación de videojuegos basado en la descarga. Al usuario no se le proporciona un dispositivo físico, si no que tras adquirir el juego lo descarga y lo almacena en un dispositivo (generalmente disco duro) asociado a la consola. Suelen ser juegos con varios años, o de consolas de generaciones anteriores. [↑]

[4] **Videojuego arcade.** Arcade es el término genérico que se aplicaba a las antiguas máquinas recreativas de videojuegos disponibles en lugares públicos de ocio o salones recreativos. Actualmente, se ofrecen muchos de estos juegos en las plataformas online de descarga de las consolas de última generación, por lo que dichos juegos también son identificados con el término arcade. A su vez, y por compartir espacio de descarga, los juegos no comerciales que se ofrecen a través de estas plataformas online también son conocidos como juegos arcade. Muchos, son nuevas versiones de estos juegos antiguos, con mejoras técnicas, gráficas o de jugabilidad. [↑]

[5] **Videojuego indie.** Este término está asociado a la comunidad de desarrolladores de *XBOX 360* y hace referencia a los juegos publicados por desarrolladores amateur a través de la plataforma online de dicha consola. Están sujetos a condiciones especiales de publicación y venta ya comentadas en este documento. [↑]

[6] **XNA.** *Microsoft XNA* es un conjunto de herramientas con un entorno de ejecución administrado proporcionado por *Microsoft* que facilita el desarrollo de videojuegos compatibles con las plataformas *XBOX 360*, PC y *Zune*. [↑]

[7] **API.** Un interfaz de programación de aplicaciones o API (del inglés *application programming interface*) es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. [↑]

[8] **Ítem.** Un ítem es un elemento típico en los videojuegos cuya función es otorgar una mejora temporal o definitiva al jugador en alguna de sus cualidades, ya sea positiva o negativamente. Un ejemplo típico es el aumento de vida o de velocidad, la obtención de un nuevo arma o la capacidad temporal de ser invulnerable. [↑]

[9] **Jukebox.** Es el término inglés que se refiere a las máquinas de discos, también conocidas como rockola. Un jukebox es un dispositivo parcialmente automatizado que reproduce música pregrabada y que normalmente se acciona con monedas. [↑]

[10] **CEO.** Siglas de "*Chief Executive Officer*", CEO hace referencia al puesto de presidente-director general o máximo responsable de una empresa y sirve para referirse a la persona tiene las máximas responsabilidades ejecutivas dentro de una empresa. [↑]

[11] **Arpanet.** La red *ARPANET* (*Advanced Research Projects Agency Network*) fue creada por encargo del Departamento de Defensa de los Estados Unidos como medio de comunicación para los diferentes organismos del país. Fue la espina dorsal de Internet hasta **1990**, tras finalizar la transición al protocolo TCP/IP iniciada en **1983**. [↑]

[12] **Jumper.** El término Jumper hace referencia al dispositivo físico de los juegos de la consola Magnavox. Fue la primera consola en distribuir sus juegos mediante cartuchos, conocidos como "jumpers" porque en realidad los cartuchos no eran programables, sólo eran unas tarjetas con "jumpers" que hacían diferentes contactos con la consola consiguiendo así generar diferentes señales analógicas que se trasmitían al televisor. [↑]

[13] **Cartucho.** El término cartucho hace referencia a cualquier aparato extraíble sujeto a otro que lo contiene. Se puede considerar sinónimo de casete. Un cartucho es un recipiente que contiene un tipo de memoria externa rígida y que se conecta a un dispositivo electrónico como por ejemplo una videoconsola. [↑]

[14] **Hi-score.** Término inglés que hace referencia al concepto de puntuación máxima. Este concepto revolucionó el mundo de los videojuegos, sobre todo en lo que a máquinas recreativas se refiere fomentando la competitividad entre los jugadores por ver quien ostentaba la mayor puntuación en un determinado juego. [↑]

[15] **Easter-egg.** Literalmente "huevo de pascua virtual", en el mundo de la informática y los videojuegos, los huevos de pascua son mensajes, gráficos, efectos de sonido, o cambios inusuales en el comportamiento de los programas que se producen después de introducir un comando o una combinación de botones. [↑]

[16] **Third-party.** El término hace referencia a los equipos de desarrollo de videojuegos externos a las productoras/distribuidoras. [↑]

[17] **Pad.** Un pad, gamepad o joypad, es un tipo de controlador o mando de juego. Diseñado para jugar con las dos manos, suelen tener una serie de botones de acción (manejados con el pulgar derecho) y una serie de botones de dirección (manejados con el pulgar izquierdo). [↑]

[18] **Power up.** Término similar a ítem, hace referencia a los elementos que el jugador puede encontrar en el juego y que afectan de manera positiva o negativa a alguna de sus cualidades, ya sea temporal o definitivamente. [↑]

[19] **Pentominó.** Un pentominó es una poliforma de la clase poliominó que consiste en una figura geométrica compuesta por cinco cuadrados unidos por sus lados. Existen doce pentominós diferentes, que se nombran con diferentes letras del abecedario. *Alekséi Pázhitnov* se inspiró en ellos al crear el videojuego *Tetris*. [↑]

[20] **RPG.** Siglas de "*role playing game*", hace referencia al género de los videojuegos que usa elementos de los juegos de rol tradicionales. [↑]

[21] **Aventura gráfica.** Subgénero de los videojuegos de aventura, cuya dinámica consiste en ir avanzando a través de la resolución de diversos rompecabezas, planteados como situaciones que se suceden en la historia, interactuando con personajes y objetos a través de un menú de acciones o interfaz similar, utilizando un cursor para mover al personaje y realizar las distintas acciones. [↑]

[22] **Isométrico.** Una proyección isométrica es un método gráfico de representación. Constituye una representación visual de un objeto tridimensional en dos dimensiones, en la que los tres ejes ortogonales principales, al proyectarse, forman ángulos de 120º, y las dimensiones paralelas a dichos ejes se miden en una misma escala. Muchos videojuegos utilizaban un punto de vista isométrico, o en "perspectiva 3/4", en la década de los noventa porque permitía desplazar los elementos gráficos sin modificar el tamaño; limitación inevitable para máquinas con baja capacidad gráfica. [↑]

[23] **Videoconsola portátil.** Un videojuego o videoconsola portátil es un dispositivo electrónico ligero que permite jugar a los videojuegos y que, a diferencia de una videoconsola clásica, proporciona los controles, la pantalla, los altavoces y la alimentación (ya sean pilas o baterías) integrados en la misma unidad y todo ello con un pequeño tamaño; de forma que se pueda transportar y jugar en cualquier lugar o momento. [↑]

[24] **Logro.** También conocido con el término ingles "*achievement*" un premio obtenido en un videojuego por alcanzar un determinado objetivo de carácter complejo. Cuanto mayor sea la complejidad del logro, mayor será la recompensa. El sistema de logros, recuperado en la comunidad de jugadores de *XBOX 360*, ha conseguido alargar la vida de los juegos aplicando

un sistema en el que cada juego proporciona una cantidad fija de puntos obtenidos cumpliendo los logros. Dichos puntos se acumulan y muestran en el perfil público del jugador, fomentando la competitividad de los jugadores por alcanzar la mayor cantidad posible de puntos asociados a los logros. [↑]

[25] **Videojuego de disparos en primera persona.** También conocido como FPS, de inglés "*first person Shooter*", o simplemente Scooter; es un género de videojuegos y subgénero de los videojuegos de disparos que se desarrolla desde la perspectiva del personaje protagonista y en la que, típicamente, sólo se ven las manos del personaje. [↑]

[26] **Survival horror.** Conocido también como "terror y *supervivenciar"* es un género de videojuegos que utiliza distintos elementos para crear una atmósfera de terror psicológico en el jugador. Destacan por la poca libertad de movimientos (caminar, correr, apuntar y atacar) y la escasez de recursos, por presencia de puzzles o acertijos y por requerir una capacidad de investigación y observación detallada por parte del jugador. [↑]

[27] **Mando Analógico.** El término hace referencia a los controles añadidos a los gamepad de las consolas similares a los joystick. Fueron introducidos en los mandos con el auge de las 3D, puesto que permiten controlar a los personajes en escenarios en tres dimensiones con mucha mayor precisión que las crucetas. La mayoría de los mandos actuales incluyen dos analógicos: uno que sustituye la funcionalidad de la cruceta y otro que sustituye parte de la funcionalidad de los botones de acción. [↑]

[28] **Avatar.** En lo que a XBOX 360 se refiere, un avatar una representación virtual y "caricaturizada" del usuario en la plataforma XBOX Live. [↑]

[29] **Sandbox.** Género de los videojuegos, también conocido como acción-aventura, que se caracterizan por permitir que el jugador pueda hacer lo que él quiera, viajar libremente por el mapa e interactuar con casi todo lo que este a su disposición sin tener que seguir una línea argumental prefijada. Estos juegos son una mezcla de géneros, entre ellos disparos, luchas y carreras. El representante más famoso de este género es la saga *Grand Theft Auto*. [↑]

[30] **Sistema PEGI.** PEGI, siglas de "*Pan European Game Information*", es un sistema europeo para clasificar el contenido de los videojuegos y otro tipo de software de entretenimiento. Fue desarrollado por la *ISFE* y entró en práctica en abril del **2003**. Se aplica en 25 países. [↑]

[31] **Formato UMD**. Siglas de "*Universal Media Disc***"**, es un disco óptico desarrollado por *Sony* conocido sobre todo por su uso en la consola *PSP*. Puede almacenar 800 Mb de datos, 1,8 GB en doble capa. Puede incluir juegos, películas, música, o combinaciones de estos elementos. [↑]

[32] **Import**. En el ámbito de la consola *PSP* de *Sony*, el término import hace referencia a las conversiones de juegos de otras consolas, emuladores, o software en general no comercial, desarrollados por la comunidad de usuarios de forma amateur aumentando las funcionalidades de la consola. Por ejemplo, reproductores multimedia, conversores de archivos, versiones de juegos de *Playstation* o emuladores de consolas como *SNES*. [↑]

[33] **Scroll**. En un videojuego, se denomina Scholl al desplazamiento en 2D de los gráficos que conforman el escenario. Se puede hablar de "*scroll horizontal*" cuando la acción se desarrolla horizontalmente, "*scroll vertical*" cuando se desarrolla verticalmente y "*scroll parallax*" cuando se mueven dos o más planos de scroll para dar una cierta sensación de profundidad al videojuego. [↑]

[34] **Radar**. En el caso de este proyecto, radar hace referencia al alcance en casillas que tienen los enemigos para detectar la posición del jugador en el mapa, de forma que, si el jugador está muy alejado de los enemigos será indetectable para ellos. En otros videojuegos el radar de los está representado por los sentidos humanos influyendo en el proceso de detección el alcance visual y el auditivo de los enemigos. [↑]

[35] **Game Engine**. Es una plataforma software diseñada para la creación y desarrollo de videojuegos. Proporciona una serie de funcionalidades que incluyen un motor de renderización, un motor de física de cuerpos, soporte para audio y sonidos, scripts, animaciones, inteligencia artificial, juego en red, streaming de contenidos, manejo de memoria y multijugador etc. [↑]

[36] **Skeletal**. También conocido como "*Skeletal Mesh*", es un modelo dinámico de animación basado en los huesos, es decir, todos estos modelos representados tienen una base o esqueleto que determina sus movimientos. [↑]

[37] **Morph**. Es un modelo dinámico de animación basado en puntos de inflexión, utilizado principalmente para animaciones faciales y de expresión. [↑]

[38] **Environment mapping**. Es una forma de mapeado de texturas en la cual las coordenadas de la textura son dependientes de la vista. Por ejemplo, simular reflejo en un objeto con brillo. [↑]

[39] **Stencil buffer**. Es una técnica aplicada para generar sombras y brillos en aplicaciones en 3D basada en el uso de buffers de memoria para procesar la información de los píxeles en tiempo real. [↑]

[40] **Modder**. Un mod en informática es cualquier tipo de cambio a algún programa, mejorándolo o modificándolo respecto a la forma original del mismo. Un modder es aquel que realiza dicho cambio. En los videojuegos, sobre todo en los de PC, son muy típicos los mods que modifican la funcionalidad del juego, por ejemplo añadiendo nuevas capacidades a los personajes. [↑]

[41] **Microsoft Points**. Microsoft Points es la moneda de la tiendas online XBOX Live y Zune. Los puntos permiten que los usuarios compren el contenido sin una tarjeta de crédito agilizando las gestiones online. [↑]

[42] **XBOX Live Arcade**. Es un servicio online de *Microsoft* que permite descargar videojuegos en *XBOX 360*. Apareció a mediados de **2004**, como servicio desde el cual se podían descargar videojuegos para *XBOX*. Incluía versión de juegos antiguos y juegos de desarrollo amateur. Actualmente sólo incluye proporciona descargas de juegos bajo demanda, juegos retro y juegos arcade desarrolladores por grandes empresas. [↑]

[43] **XBOX Live Comunity Games**. Una vez que *XBOX Live Arcade* dejó de incluir juegos de desarrollo amateur en su catálogo se creo esta plataforma para dar servicio a este tipo de

juegos. Actualmente *Comunity Games* sólo incluye la descarga de juegos arcade desarrollados por compañías pequeñas. [↑]

[44] XBOX Live Indie Games. Una vez que *XBOX Comunity Games* dejó de incluir juegos de desarrollo amateur en su catálogo se creo esta plataforma para dar servicio a este tipo de juegos. Para publicar juegos en esta plataforma hay que ser miembro de "*XNA Creators Club*". [↑]

[45] XNA Creators Club. *XNA Creators Club* es una plataforma online creada por *Microsoft* para dar apoyo a los programadores de juegos en *XNA*. A través de la plataforma se proporcionan tutoriales, ejemplos de buenas prácticas, foros de discusión etc. Para poder publicar un juego en el bazar online de *XBOX 360* es necesario ser miembro *Premium* de *XNA Creators Club*, lo que supone una cuota de 49$ por un cuatrimestre o 99$ por un año. [↑]

[46] Token. En el ámbito de *XBOX 360*, un token es un código único que permite a su poseedor descargar un determinado contenido son gastar *Microsoft Points*. El contenido descargado suele asociarse con el número de serie de la consola por lo que no podrá ser distribuido evitando la copia. Los token suelen utilizarse para distribuir betas de juegos a revistas o blogs especializados o para proporcionar contenidos exclusivos a los usuarios. [↑]

Referencias

[1] Hudson – The Bomberman Super Site! [en línea]. [ref. de 2007]. Disponible en World Wide Web: < http://www.hudsonentertainment.com/ >

[2] Guinness – Guinness World Records Gamers' Edition [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://gamers.guinnessworldrecords.com/ >

[3] Asociación de Desarrolladores y Editores de Software de Entretenimiento (aDeSe). [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.adese.es/ >

[4] Hudson Entertainment [en línea]. [ref. de 2007]. Disponible en World Wide Web: < http://www.hudsonentertainment.com/ >

[5] YouTube – Super Gussun Oyoyo Walkthrough [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://www.youtube.com/ >

[6] El otro lado – Historia de los videojuegos: Prehistoria [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://www.elotrolado.net/wiki/ >

[7] Zona Nintendo – Historia de la gran NINTENDO [en línea]. [ref. de 2005]. Disponible en World Wide Web: < http://zonanintendo.blogcindario.com/ >

[8] Club Nintendo – Hanafuda Cards [en línea]. [ref. de 2008]. Disponible en World Wide Web: < https://club.nintendo.com/ >

[9] Wikipedia, la enciclopedia libre – Coleco [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Coleco >

[10] Museo 8bits – Coleco Adam [en línea]. [ref. de 2000]. Disponible en World Wide Web: < http://www.museo8bits.com/ >

[11] Mattel Inc. – History [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://corporate.mattel.com/ >

[12] Intellivision Lives – Intellivision [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.intellivisionlives.com/ >

[13] Intellivision Lives – Intellivoice [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.intellivisiongames.com/ >

[14] Sony – Una historia de Innovación [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.sony.es/ >

[15] Gamerfilia – SEGA, la historia de un mito [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://blogs.gamefilia.com/ >

[16] System 16 – SEGA Mechanical Hardware [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.system16.com/ >

[17] Wikipedia, la enciclopedia libre – Taito Corporation [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Taito_Corporation >

[18] Computer Emuzone – La historia de Commodore [en línea]. [ref. de 2001]. Disponible en World Wide Web: < http://computeremuzone.com/ >

[19] KIM-1 – The KIM-1 Enthusiasts Page [en línea]. [ref. de 2001]. Disponible en World Wide Web: < http://www.kim-1.com/ >

[20] C64 – Commodore 64 [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.c64.com/ >

[21] Amiga – Commodore Amiga Community [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.amiga.org/ >

[22] Amstrad ESP – Historia [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.amstrad.es/ >

[23] Old Computers Museum – CPC 464 [en línea]. [ref. de 2007]. Disponible en World Wide Web: < http://www.old-computers.com/ >

[24] Old Computers Museum – Amstrad PCW [en línea]. [ref. de 2007]. Disponible en World Wide Web: < http://www.old-computers.com/ >

[25] Amstrad ESP – Amstrad GX4000 [en línea]. [ref. de 2007]. Disponible en World Wide Web: < http://www.gx4000.amstrad.es/ >

[26] Sky TV – Digital Satellite TV [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.sky.com/ >

[27] Konami Digital Entertainment – Corporate History [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.konami.co.jp/ >

[28] Treasure Co. Ltd – Treasure Company [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.treasure-inc.co.jp/ >

[29] Microsoft España Internacional – Xbox [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.xbox.com/es-ES/ >

[30] Infoconsolas – Historia de los videojuegos: los inicios [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.infoconsolas.com/ >

[31] Wikipedia, the free encyclopedia – OXO [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/OXO >

[32] Tecnotopia – EDSAC [en línea]. [ref. de 2004]. Disponible en World Wide Web: < http://www.tecnotopia.com/ >

[33] Abadía DIGITAL – Tennis for Two: El primer videojuego de la historia [en línea]. [ref. de 2007]. Disponible en World Wide Web: < http://www.abadiadigital.com/ >

[34] Computer History Museum – PDP-1 [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://pdp-1.computerhistory.org/pdp-1/ >

[35] El Complejo Lambda – Físicas en los videojuegos: ENIAC y Spacewar! [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://complejolambdaweb.blogspot.com/ >

[36] Spacewar! Original 1962 game code running on PDP-1 emulator in Java [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://spacewar.oversigma.com/ >

[37] Stanford University Infolab – The Galaxy Game [en línea]. [ref. de 1997]. Disponible en World Wide Web: < http://infolab.stanford.edu/ >

[38] Computer History Museum – The Galaxy Game [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://www.computerhistory.org/ >

[39] Abadía DIGITAL – Computer Space: La primera máquina recreativa de la historia [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.abadiadigital.com/ >

[40] Wikipedia, the free encyclopedia – Ampex [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Ampex >

[41] Retro Games – Historia de Atari [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://www.retrogames.cl/ >

[42] Wikipedia, the free encyclopedia – Magnavox [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Magnavox >

[43] Wikipedia, la enciclopedia libre – Magnavox Odyssey [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Magnavox_Odyssey >

[44] AEGO – Asociación Española de Go [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://aego.biz/ >

[45] Pong Story – Pong [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.pong-story.com/ >

[46] System 16 – Taito Discrete Logic Hardware Elepong [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.system16.com/ >

[47] Arcade History – Space Race [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.arcade-history.com/ >

[48] n-1 – The Atari Gotcha [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://n-1.nl/gotcha/ >

[49] Everything2 – Kee Games [en línea]. [ref. de 2003]. Disponible en World Wide Web: < http://everything2.com/ >

[50] Wikipedia, the free encyclopedia – Tank [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Tank >

[51] Old Computers Museum – Gran Trak 10 [en línea]. [ref. de 2007]. Disponible en World Wide Web: < http://www.old-computers.com/ >

[52] Melbourne Pinball Restorations – Taito Speed Race [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.melbournepinballrestorations.com/ >

[53] Wikipedia, the free encyclopedia – Namco History [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Namco >

[54] Atari Museum – Atari Home Pong [en línea]. [ref. de 2007]. Disponible en World Wide Web: < http://www.atarimuseum.com/ >

[55] Toy Show – The International resource for the toy industry [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.toyshow.com/ >

[56] Microsoft [en línea]. [ref. de 2010]. Disponible en World Wide Web: < www.microsoft.com/spain/ >

[57] Mitsubishi Electric [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://global.mitsubishielectric.com/ >

[58] Gamefilia blogs – Coleco Telstar Alpha [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://blogs.gamefilia.com/ >

[59] Wikipedia, the free encyclopedia – Breakout [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Breakout >

[60] Atari Guide – Night Driver [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.atariguide.com/ >

[61] Insert Coin Clásicos – Death Race [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://jaimixx.lacoctelera.net/ >

[62] Wikipedia, the free encyclopedia – Fairchild Channel F [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Fairchild_Channel_F >

[63] Atari Museum – Atari VCS [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.atarimuseum.com/ >

[64] Nintendo Wikia – Color TV Game 6 [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://nintendo.wikia.com/wiki/Color_TV_Game_6 >

[65] Space Invaders Online and game history [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.spaceinvaders.de/ >

[66] Nintendo Wikia – Color TV Game 15 [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://nintendo.wikia.com/wiki/Color_TV_Game_15 >

[67] Odyssey2 – Magnavox Odyssey 2 [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://videopac.org/o2site/ >

[68] Atari – Atari Adventure [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.atari.com/arcade/adventure >

[69] Atari – Atari Asteroids [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.atari.com/arcade/asteroids >

[70] Wikipedia, la enciclopedia libre – Pac - Man [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Pac-Man >

[71] Time Magazine – Pac-Man Fever [en línea]. [ref. de 2001]. Disponible en World Wide Web: < http://www.time.com/ >

[72] Wikipedia, la enciclopedia libre – Capcom [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Capcom >

[73] Feel The Byte – Grandes momentos de la historia de los videojuegos: Activision [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://www.feelthebyte.com/ >

[74] Wikipedia, the free encyclopedia – Game & Watch [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Game_&_Watch >

[75] Wikipedia, la enciclopedia libre – Frogger [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Frogger >

[76] Wikipedia, la enciclopedia libre – Donkey Kong [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Donkey_Kong >

[77] Neo Teo – Historia de la Commodore 64 [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://www.neoteo.com/ >

[78] Classic Gaming – ColecoVision [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://classicgaming.gamespy.com/colecovision/ >

[79] Museo 8bits – Emerson Arcadia [en línea]. [ref. de 2000]. Disponible en World Wide Web: < http://www.museo8bits.com/fever.htm >

[80] Atari Gaming Headquarters – Atari 5200 [en línea]. [ref. de 2002]. Disponible en World Wide Web: < http://www.atarihq.com/5200/ >

[81] Wikipedia, la enciclopedia libre – Sinclair ZX Spectrum [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Sinclair_ZX_Spectrum >

[82] Vectrex [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.vectrex.com/ >

[83] Gamasutra – A history of Electronic Arts [en línea]. [ref. de 2006]. Disponible en World Wide Web: < http://www.gamasutra.com/ >

[84] Wikipedia, the free encyclopedia – Ocean Software [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Ocean_Software >

[85] Famicom World – Nintendo Famicom [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://famicomworld.com/ >

[86] Museo 8bit – SEGA SG-1000 [en línea]. [ref. de 2000]. Disponible en World Wide Web: < http://www.museo8bits.com/ >

[87] Wikipedia, the free encyclopedia – Mario Bros [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Mario_Bros. >

[88] MSDOX – Historia del software español de entretenimiento [en línea]. [ref. de 2004]. Disponible en World Wide Web: < http://www.msdox.com/ >

[89] Square Enix Japan – Corporate History [en línea]. [ref. de 2006]. Disponible en World Wide Web: < http://www.square-enix.com/ >

[90] Universo Final Fantasy – Final Fantasy [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.universoff.es/FF1/ >

[91] Wikipedia, la enciclopedia libre – Super Mario Bros. [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Super_Mario_Bros. >

[92] Tetris [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.tetris.com/ >

[93] Museo 8bits – Master System [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.museo8bits.com/sega8_ms.htm >

[94] Nintendo Zelda Universe – The Legend of Zelda [en línea]. [ref. de 2006]. Disponible en World Wide Web: < http://www.zelda.com/ >

[95] Metroid Headquarters – Metroid [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.metroidheadquarters.com/ >

[96] Wikipedia, la enciclopedia libre – Arkanoid [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Arkanoid >

[97] Wikipedia, the free encyclopedia – Maniac Mansion [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Maniac_Mansion >

[98] The Mega Man Home Page – Mega Man [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.mmhp.net/ >

[99] Kojima productions – Metal Gear Series [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.konami.jp/ >

[100] Museo 8bits – SEGA Mega Drive [en línea]. [ref. de 2000]. Disponible en World Wide Web: < http://www.museo8bits.com/megadrive1.htm >

[101] Nintendo – Super Nintendo [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.nintendo.es/ >

[102] La abadía del crimen [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.abadiadelcrimen.com/ >

[103] Neo Teo – La historia de Game Boy [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.neoteo.com/ >

[104] 3D Juegos – Historia de la PSP [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.3djuegos.com/ >

[105] Museo 8bits – SEGA Game Gear [en línea]. [ref. de 2000]. Disponible en World Wide Web: < http://www.museo8bits.com/sega8_gg.htm >

[106] Wikipedia, la enciclopedia libre – Pang [en línea]. [ref. de 2000]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Pang >

[107] Neo-Geo – Neo Geo [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.neo-geo.com/ >

[108] Wikipedia, the free encyclopedia – Super Mario World [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Super_Mario_World >

[109] Wikipedia, la enciclopedia libre – The Secreto f Monkey Island [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Monkey_Island >

[110] Wikipedia, la enciclopedia libre – Sonic [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Sonic_the_Hedgehog >

[111] Nintendo Zelda Universe – The Legend of Zelda: A Link to the past [en línea]. [ref. de 2006]. Disponible en World Wide Web: < http://www.zelda.com/ >

[112] Wikipedia, la enciclopedia libre – Street Fighter II [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Street_Fighter_II >

[113] Mortal Kombat Universe – Mortal Kombat [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.mortal-kombat.org/ >

[114] 3D Realms Site – Wolfenstein 3D [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.3drealms.com/ >

[115] Wikipedia, the free encyclopedia – Alone in the Dark [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Alone_in_the_Dark >

[116] Wikipedia, la enciclopedia libre – Doom [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Doom >

[117] Wikipedia, the free encyclopedia – FIFA Series [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/FIFA >

[118] Sega Saturn [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.segasaturn.co.uk/ >

[119] ING – Playstation History [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://psx.ign.com/ >

[120] Nintendo – Nintendo 64 [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.nintendo.es/ >

[121] E3 – Electronic Entertainment Expo [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.e3expo.com/ >

[122] Wikipedia, the free encyclopedia – Super Mario 64 [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Super_Mario_64 >

[123] Dreamcast – SEGA Dreamcast [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.dreamcast.es/ >

[124] Sony Playstation – Playstation 2 [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.playstation.com/ps2/ >

[125] Hoopedia – NBA 2K Series Overview [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://hoopedia.nba.com/ >

[126] Wikipedia, the free encyclopedia – NBA Live Series [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/NBA_Live_series >

[127] Capcom – Resident Evil [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.residentevil.com/ >

[128] Universo Final Fantasy – Final Fantasy VII [en línea]. [ref. de 2006]. Disponible en World Wide Web: < http://www.universoff.es/FF7/ >

[129] Polyphony Digital – Gran Turismo [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.gran-turismo.com/ >

[130] RockStar Games [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.rockstargames.com/ >

[131] Wikipedia, The free encyclopedia – Grand Theft Auto [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Grand_Theft_Auto >

[132] El otro lado – Historia de los videojuegos: Era moderna [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://www.elotrolado.net/ >

[133] Nintendo – Nintendo game Cube [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.nintendo.es/ >

[134] Wikipedia, The free encyclopedia – Microsoft XBOX [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/Xbox >

[135] Halo [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://halo.xbox.com/en-us >

[136] Konami – Konami PES Club [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.pesclub.es/ >

[137] Nintendo – Nintendo DS [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.nintendo.es/ >

[138] Sony – Sony PSP [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.playstation.com/psp/ >

[139] El otro lado – Historia de los videojuegos: Era moderna – Última generación de consolas [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://www.elotrolado.net/ >

[140] Nintendo – Nintendo Wii [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.nintendo.es/ >

[141] Sony – Sony Playstation 3 [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.playstation.com/ps3/ >

[142] Segasaturno – Historia y listado de juegos de Bomberman [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.segasaturno.com/ >

[143] Eric and the Floaters [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://www.therubberbeermat.co.uk/ >

[144] XNA – Microsoft XNA [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.xna.com/ >

[145] XNA Creators Club –XNA Game Studio 3.1 [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://creators.xna.com/es-ES/ >

[146] Wikipedia, the free encyclopedia – List of Game Engines [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://en.wikipedia.org/wiki/List_of_game_engines >

[147] Wikipedia, la enciclopedia libre – Herramientas de desarrollo de videojuegos [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://es.wikipedia.org/Herramientas_de_desarrollo_de_videojuegos >

[148] Adventure Game Studio [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.adventuregamestudio.co.uk/ >

[149] Allegro – A game programming library [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.talula.demon.co.uk/allegro/ >

[150] Build Engine [en línea]. [ref. de 2002]. Disponible en World Wide Web: < http://advsys.net/ken/build.htm >

[151] Blender 3D [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.blender.org/ >

[152] Crystal Space 3D [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.crystalspace3d.org/ >

[153] Dim3 [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://dim3wiki.site88.net/ >

[154] Div Game Studio [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.gemixstudio.com/ >

[155] Wikipedia, la enciclopedia libre – Dom Engine [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Doom_Engine >

[156] Emergent – Gamebryo [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.emergent.net/ >

[157] YoYo Games – Game Maker [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.yoyogames.com/ >

[158] 3D Game Studio [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.3dgamestudio.com/ >

[159] Glk/Glulx [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.eblong.com/ >

[160] Trac – GtkRadiant [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.qeradiant.com/ >

[161] Wikipedia, la enciclopedia libre – Havok [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Havok >

[162] Irrlicht Engine [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://irrlicht.sourceforge.net/ >

[163] The Lightweight Java Game Library (LWJGL) [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://lwjgl.org/ >

[164] M.U.G.E.N. [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://mugenhispania.org/ >

[165] OGRE 3D [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.ogre3d.org/ >

[166] Panda3D [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.panda3d.org/ >

[167] Pygame [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://www.pygame.org/ >

[168] Quest3D [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://quest3d.com/ >

[169] RPG Maker [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://rpgmaker.es/ >

[170] Valve Software – Source [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://source.valvesoftware.com/ >

[171] Verge – 2D Game Engine [en línea]. [ref. de 2004]. Disponible en World Wide Web: < http://verge-rpg.com/ >

[172] Wikipedia, la enciclopedia libre – Wonderland [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Proyecto_Wonderland >

[173] XNA Creators Club Online [en línea]. [ref. de 2010]. Disponible en World Wide Web: < http://creators.xna.com/es-ES/ >

[174] XNA Content Pipeline [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://xnafactory.blogspot.com/ >

[175] Bomberman Online [en línea]. [ref. de 2009]. Disponible en World Wide Web: < http://bmoworld.com/ >

[176] Sprite Data Base [en línea]. [ref. de 2008]. Disponible en World Wide Web: < http://randomhoohaas.flyingomelette.com/Sprites/ >

[177] Stephen Cawood, Pat MaGgee. *Microsoft XNA Game Studio Creator's Guide. An Introdution to XNA Game Programming*. 2007, The McGraw-Hill Companies. 482 p.

[178] Benjamin Nitschke. *Professional XNA Game Programming: For Xbox 360 and Windows*. 2007. Wrox Press. 504 p.