TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA POLITÉCNICA SUPERIOR Y FACULTAD DE CIENCIAS EXPERIMENTALES

"Booster: Un videojuego de plataformas en 2D con C#"

Curso 2013/2014

Alumno/a:

Andrés Vicente Linares

Director/es:

Antonio Leopoldo Corral Liria

Agradecimientos

En primer lugar me gustaría agradecer a mi tutor, Antonio Corral, por el apoyo, supervisión y consejos que me he dado.

A mi familia por todo su apoyo durante el desarrollo del proyecto y la carrera.

A mis amigos y compañeros de clases, los que, además de apoyarme y darme ideas, me han ayudado a encontrar errores de los que no me había dado cuenta, en especial a Alejandro Rodríguez, Juan José Riado, Carlos Ridao, con los que he pasado muy buenos momentos haciendo actividades tanto relacionadas como ajenas a la universidad.

Finalmente, a la Universidad de Almería y sus profesores, los cuáles me han enseñado las bases que han permitido que haya podido realizar este proyecto.

Contenido

1. Introducción 9
1.1. Motivación 10
1.2. Justificación profesional 11
1.3. Objetivos 11
1.4. Estructura del documento 12
2. Videojuegos 13
2.1. Videojuego 13
2.1.1. Controlador de videojuegos 13
2.1.2. Plataformas 15
2.1.3. Géneros 16
2.1.4. Impacto en la sociedad 23
2.2. Desarrollo de videojuegos 25
2.2.1. Roles en un equipo de desarrollo de videojuegos 25
2.2.2. Proceso de desarrollo de videojuegos 27
2.3. Motor de juegos 30
2.3.1. Descripción de algunos motores de videojuegos 31
2.4. Estado del arte 32
2.4.1. Mercado de los videojuegos en el mundo 32
2.4.2. Mercado de los videojuegos en España 34
2.4.3. Industria de los videojuegos en España 34
3. Booster 37
3.1. Introducción 37
3.2. XNA Framework y MonoGame 37
3.2.1. XNA Framework 37
3.2.2. MonoGame 38
3.2.3. La clase Game y el bucle del juego 38
3.2.4. Características 40
3.3. Análisis 42
3.3.1. Niveles 43
3.3.2. Menús 44
3.3.3. Estados de transición 44
3.4. Diseño e implementación 44
3.4.1. Gestor de estados 45
3.4.2. Gestión de entrada 46
3.4.3. Almacén de recursos 48

3.4.4. Cámara 49
3.4.5. Transiciones 50
3.4.6. Menús 52
3.4.7. Entidades 56
3.4.8. Nivel 62
3.4.9. Animaciones 66
3.4.10. Creación de entidades 67
3.4.11. Colisiones 71
3.5. Planificación temporal 77
3.6. Herramientas 80
4. Resultados 81
4.1. Transiciones 82
4.2. Menús 83
4.3. Nivel 84
5. Conclusiones 87
5.1. Conclusiones 87
5.2. Trabajo futuro 88
6. Bibliografía 91
6.1. Referencias 91

Tabla de figuras

Figura 1. Gamepad de la videoconsola NES [4] 14
Figura 2. Gamepad de la videoconsola Xbox 360 [5] 14
Figura 3. Tabla comparativa de jugadores de una región con la población 33
Figura 4. Gráfica comparativa de modelos de distribución utilizados entre 2012 y 2016 33
Figura 5. Gráfica de tiempo invertido en los videojuegos por plataforma 34
Figura 6. Distribución geográfica de empresas de videojuegos en España [47] 35
Figura 7. Distribución de empresas según el número de empleados 35
Figura 8. Distribución de la facturación por regiones 36
Figura 9. Diagrama del bucle del juego en XNA Framework 39
Figura 10. Sistema de coordenadas de XNA Framework y MonoGame [52] 41
Figura 11. Diagrama de casos de uso inicial 43
Figura 12. Diagrama de casos de uso final 43
Figura 13. Diagrama de clases del gestor de estados 45
Figura 14. Diagrama de secuencias de la actualización del videojuego 46
Figura 15. Diagrama de secuencias del mostrado por pantalla del videojuego 46
Figura 16. Estructura del patrón State 46
Figura 17. Diagrama de clases del gestor de entrada 47
Figura 18. Estructura del archivo XML de configuración 47
Figura 19. Estructura del patrón Singleton 48
Figura 20. Diagrama de clases de la clase Recursos y las clases relacionadas 48
Figura 21. Estructura de un archivo XML con información sobre una hoja de sprites 49
Figura 22. Diagrama de clases de la cámara 49
Figura 23. Diagrama de clases de estados de transición 50
Figura 24. Diagrama de estados de GameIntro 50
Figura 25. Diagrama de estados de LoadingState 51
Figura 26. Diagrama de estados de GameOver 51
Figura 27. Diagrama de estados de LevelCompleted 51
Figura 28. Diagrama de clases del estado Menu 52
Figura 29. Diagrama de secuencias del método Initialize de Menu 53
Figura 30. Estructura del patrón Template Method 53
Figura 31. Diagrama de clases de StaticMenu 54
Figura 32. Diagrama de estados de MainMenu 54
Figura 33. Estructura del archivo XML que guarda información de los niveles 55
Figura 34. Diagrama de estados de StoryMenu 55
Figura 35. Diagrama de estados de PauseMenu 55
Figura 36. Diagrama de clases de ScrollableMenu 56
Figura 37. Diagrama de estados de ChallengesMenu 56
Figura 38. Diagrama de clases de las plantillas de entidades 57
Figura 39. Diagrama de clases de Player 58
Figura 40. Diagrama de clases de SimpleTile 59
Figura 41. Diagrama de la clase ScoreObject 59
Figura 42. Diagrama de clases de DamageObject 60
Figura 43. Diagrama de clases de Spike 60
Figura 44. Diagrama de clases de Door y Key 61
Figura 45. Diagrama de clases de ExitEntity 61
Figura 46. Diagrama de clases del nivel 62
Figura 47. Diagrama de secuencias de actualizar nivel 63
Figura 48. Diagrama de secuencias de actualizar mapa 64

Figura 49. Diagrama de secuencias de mostrar el nivel 65
Figura 50. Diagrama de clases de animaciones 66
Figura 51. Estructura del patrón Builder 67
Figura 52. Diagrama de clases de construcción de Player 68
Figura 53. Diagrama de clases de construcción de SimpleTile 68
Figura 54. Diagrama de clases de construcción de ScoreObject 69
Figura 55. Diagrama de clases de construcción de DamageObject 69
Figura 56. Estructura del patrón Factory Method 70
Figura 57. Diagrama de clases de Factory Method para la creación de entidades 70
Figura 58. Situación problemática para la gestión de colisiones con Farseer Physics Engine 72
Figura 59. Gestión de colisiones en movimiento diagonal sobre superficie plana. 73
Figura 60. Gestión de colisiones en movimiento diagonal con huecos 74
Figura 61. Caso problemático en el algoritmo de colisiones de comprobación por losas 74
Figura 62. Estructura del patrón Strategy 75
Figura 63. Diagrama de clases de la gestión de colisiones 75
Figura 64. Gestión de colisiones en movimientos diagonales con la versión mejorada del algoritmo 77
Figura 65. Cronología del desarrollo 80
Figura 66. Diagrama de estados completo 81
Figura 67. Capturas de los estados de transición 82
Figura 68. Capturas de los menús 83
Figura 69. Captura de un nivel 84
Figura 70. Booster: un videojuego de plataformas en 2D. 87

1. Introducción

El desarrollo de un software, al igual que el desarrollo de cualquier tipo de producto, suele tener un plazo, y, aunque los desarrolladores prefieran desarrollar todo el software ellos, no siempre se dispone de todo el tiempo necesario para desarrollar todas las partes que componen el software, por lo que se utilizan librerías. Estas son un conjunto de implementaciones de comportamiento, que ya han sido creadas. Los desarrolladores también pueden crear sus propias librerías, aunque ya existan y puedan realizar el mismo comportamiento de las que quieren crear, pero esto no es siempre posible.

Un caso en el que es muy común usar una gran librería como base en el desarrollo del software es en los videojuegos. En el desarrollo de un videojuego a esta gran librería se le llama normalmente *motor*, y un videojuego, al igual que cualquier otro software, se puede hacer partiendo desde cero, sin hacer uso de un motor ya existente. Todo esto implicaría implementar un motor de renderizado, un motor de físicas o de detección de colisiones, sonidos, animaciones, inteligencia artificial, comunicación en red, etc, que deberían ser reutilizables, para no tener que repetir el mismo proceso cada vez que se quiera desarrollar un videojuego. El desarrollo de este motor implicaría un gran consumo de tiempo en análisis, diseño, implementación, pruebas y mantenimiento, e incluso podría darse el caso en el que se tardará más en desarrollar el motor que en hacer el videojuego, por lo que se suelen usar motores ya existentes.

Los motores se pueden clasificar según su licencia, según el número de dimensiones que permiten, según el lenguaje en el que está desarrollado, según el lenguaje de scripting que usa, según las plataformas en las que se puede ejecutar, etc. Por lo que a la hora de escoger un motor para desarrollar un videojuego hay que tener en cuenta muchos factores, como el presupuesto del que se dispone, cómo será el videojuego que se quiere desarrollar, para qué plataformas se va a desarrollar, con qué lenguajes se sienten más cómodos los programadores, de cuánto tiempo se dispone, etc.

No todos los motores se usan de la misma forma, existen motores con los que se pueden desarrollar videojuegos usando “arrastrar y soltar”, sin necesidad de aprender un lenguaje de programación como C++ o Java, aunque también permiten programar comportamientos usando scripts, Unity y Game Maker son ejemplos de este tipo de motor. También existen otros motores, que pueden ser simplemente considerados frameworks en lugar de motores, estos motores se usan al igual que una librería (heredando de clases predefinidas, sobrescribiendo métodos, simplemente llamando a unos métodos determinados en un orden, etc), ejemplos de estos motores son Libgdx y Cocos2d.

Debido a las limitaciones de tiempo que tiene el trabajo de fin de grado, el autor ha descartado crear su propio motor para desarrollar un videojuego, ya que sólo la creación del motor podría considerarse un trabajo de fin de grado en sí. También se han descartado los motores con licencias de pago. A la hora de escoger un motor, el autor ha decidido buscar motores gratuitos, en los que se pueda programar con C# o Java, que permitan desarrollar juegos en 2D, y que estén muy bien documentados. Teniendo en cuenta estas consideraciones, el autor ha decidido usar XNA Framework.

Pero XNA Framework tiene un problema, que, aunque es muy adecuado para el aprendizaje, Microsoft ha decidido dejar de dar soporte a XNA, por lo que después de XNA 4.0, que es la última versión, no habrá más actualizaciones, por lo que los videojuegos desarrollados con XNA Framework seguirán funcionando en Microsoft Windows, Xbox 360, Zune y Windows Phone 7, pero en Windows Phone 8 no funcionan, y puede que tampoco en las nuevas versiones de Microsoft Windows.

Como solución a este problema existe MonoGame, una implementación de código abierto de XNA Framework, que sirve para desarrollar para las mismas plataformas que XNA Framework, y además, en iOS, Android, Mac OS X, Linux y Windows 8 Metro, aunque en algunas de estas plataformas hay que pagar licencias. MonoGame está en continuo crecimiento y a diferencia de XNA Framework, sigue recibiendo soporte de sus desarrolladores, y permite desarrollar juegos usando el mismo código que se usaría para XNA Framework, lo que hace que toda la documentación de XNA Framework también sirva para MonoGame.

Los videojuegos son software que integran muchas disciplinas: gráficos, sonido, físicas, inteligencia artificial, etc. En este caso, al usar MonoGame, la parte de gráficos y sonido la implementa MonoGame, pero sí es necesario implementar los algoritmos que gestionarían las colisiones, la inteligencia artificial de algunas entidades, hacer uso de diversas estructuras de datos, aplicar patrones de diseño, leer y escribir en archivos, etc, por lo que se aplicarían conocimientos adquiridos en varias asignaturas del Grado en Ingeniería Informática.

Este videojuego, **Booster**, se va a desarrollar usando el framework MonoGame en Microsoft Visual Studio 2013. Para el desarrollo se va a utilizar una metodología orientada a objetos con patrones de diseño y un enfoque iterativo e incremental.

1.1. Motivación

Como motivación personal, al autor desde pequeño le han gustado los videojuegos, y una de las razones por las que decidió escoger el Grado en Ingeniería Informática fue para que el conocimiento que adquiriera le acercara más a su meta de no sólo disfrutar jugando videojuegos, también de ser capaz de hacerlos y disfrutar haciéndolos.

Cuando el autor comenzó su último curso del Grado en Ingeniería Informática, aún no tenía claro qué hacer para su Trabajo de Fin de Grado, ya que, aunque quería hacer un videojuego, tenía la idea de que en un año iba a ser imposible desarrollar un videojuego, excepto videojuegos simples como *Solitario*, *Snake*, *Buscaminas*, etc. Lo que finalmente le hizo decidirse por desarrollar un videojuego fue el documental *Indie Game: The Movie*. Este documental muestra los problemas que han tenido los desarrolladores de videojuegos independientes Edmund McMillen y Tommy Refenes durante el desarrollo de *Super Meat Boy*, y Phil Fish durante el desarrollo de *Fez*, además, Jonathan Blow reflexiona sobre el éxito de su videojuego *Braid*. Los 3 videojuegos son de plataformas. El autor opina que es un documental muy interesante, y recomienda verlo a cualquier persona interesada en el

desarrollo de videojuegos, aunque la gente a la que sólo le guste los videojuegos o sólo sean desarrolladores de software, posiblemente también lo encontrarán interesante. Lo que hizo al autor decidirse a desarrollar un videojuego fueron los tiempos de desarrollo. El caso excepcional es *Fez*, que se anunció en Julio de 2007, y no se completó hasta Abril de 2012, ya que Phil Fish comenta en el documental que había rehecho el videojuego varias veces desde cero. Jonathan Blow desarrolló *Braid* entre Abril de 2005 y Agosto de 2008. En 3 años desarrolló un videojuego de 34 niveles y con unas mecánicas muy complicadas. *Super Meat Boy* se desarrolló desde Enero de 2009 hasta Octubre de 2010. En poco menos de 2 años, Edmund McMillen y Tommy Refenes desarrollaron un videojuego de más de 300 niveles. Teniendo en cuenta estos datos, en especial los de *Super Meat Boy*, el autor estimó que sería capaz de desarrollar un videojuego en un año, con mecánicas no muy complejas, y con pocos niveles.

Sin tener ninguna experiencia previa en el desarrollo de videojuegos, el autor ha decidido que este Trabajo de Fin de Grado será un reto personal en el que creará su primer videojuego, y que con la experiencia adquirida esté más cerca de trabajar desarrollando videojuegos, o tal vez simplemente desarrollar videojuegos por hobby.

1.2. Justificación profesional

El sector del videojuego es la industria tecnológica con mayor crecimiento. Debido a la venta online, a que los videojuegos cada vez alcanzan más ámbitos además del ocio, al impulso de la banda ancha en todo el mundo y a los nuevos modelos de negocio (free to play, publicidad, etc), los videojuegos cada vez abarcan un público mayor, por lo que generan más ingresos. Según un estudio [47], en 2012 se facturaron alrededor de 65 mil millones de dólares, y se estima que para 2016 los ingresos superen los 85 mil millones de dólares.

En los últimos años, en España se han creado una gran cantidad de empresas, en 2012 el 68% de las empresas tenían menos de 5 años. La creación de tantas nuevas empresas se debe a la proliferación de nuevos modelos de negocio a través de internet y a la aparición de nuevos dispositivos móviles, que han ampliado el mercado.

1.3. Objetivos

El principal objetivo de este trabajo de fin de grado, es desarrollar un videojuego, aunque, debido a que se tiene que realizar en un tiempo limitado, no tendría toda la funcionalidad de un videojuego comercial, pero sí la parte más importante.

Para desarrollar el videojuego, habría que cumplir los siguientes sub-objetivos:

- Crear un sistema de gestión de estados para evitar dependencias entre estados.
- Crear menús para el juego.
- Crear un sistema de gestión de entrada por teclado y por gamepad que permita configurar las teclas y botones.
- Crear clases abstractas como plantillas para crear entidades del juego.
- Crear un jugador como una entidad animada.
- Crear elementos del juego (bloques, plataformas, etc).
- Crear el estado nivel en el que un jugador pueda interactuar con los elementos del juego.
- Crear una cámara que siga al jugador por la pantalla.

- Implementar físicas para el movimiento del jugador y para las colisiones con los elementos del juego.
- Permitir pausar el juego.
- Añadir sonidos al juego.
- Guardar el progreso del jugador.
- Cargar la configuración de teclado y gamepad.

El cumplimiento de este objetivo, conlleva también a cumplir otros objetivos, que son: mejorar y ampliar los conocimientos sobre C#, y ganar experiencia desarrollando videojuegos.

1.4. Estructura del documento

En este capítulo se ha hecho una introducción sobre el proyecto, la motivación del autor para realizar este proyecto, la justificación profesional del proyecto y los objetivos que se pretenden cumplir con su realización.

En el capítulo "2. Videojuegos" se explican conceptos sobre los videojuegos y el desarrollo de videojuegos. Además, incluye el estado del arte de la industria de los videojuegos.

En el capítulo "3. Booster" se explica XNA Framework y MonoGame, que es el framework que se ha utilizado para desarrollar el videojuego. Después, se explican los requisitos del proyecto, su diseño y su desarrollo. También se explican las herramientas utilizadas y la planificación temporal del desarrollo del proyecto.

En el capítulo "4. Resultados y discusión" se mostrará el resultado final, con capturas de las pantallas del videojuego.

En el capítulo "5. Conclusiones" se explicarán las conclusiones a las que se ha llegado con el desarrollo de este Trabajo de Fin de Grado, y el posible trabajo futuro.

Por último, el capítulo "6. Bibliografía" contiene todas las referencias bibliográficas de las que se ha sacado información para la realización de este Trabajo de Fin de Grado.

2. Videojuegos

En este capítulo se explicarán algunos conceptos sobre los videojuegos, como qué es un videojuego, qué es un controlador de videojuegos, qué son las plataformas, qué géneros hay, y el impacto de los videojuegos en la sociedad. También se hablará sobre el desarrollo de videojuegos, en concreto en los roles dentro de un equipo de desarrollo, y en el proceso de desarrollo de videojuego. Por último, se hablará del estado del arte en el mundo, y también centrándose más en España.

2.1. Videojuego

Un videojuego es un software orientado al entretenimiento, que, a diferencia de otras formas de entretenimiento, como pueden ser las películas, un videojuego es interactivo. En un videojuego, uno o más jugadores interaccionan a través de un *controlador* con la *plataforma* que ejecuta dicho videojuego para generar una respuesta. Esta respuesta se ve reflejada en una pantalla con el movimiento y las acciones de los personajes, o mediante la reproducción de sonidos, o incluso a través de periféricos hápticos que producen vibración [1, 2].

2.1.1. Controlador de videojuegos

Un controlador de videojuegos es un periférico que permite proporcionar una entrada a un videojuego [3].

Existen muchos tipos de controladores, y lo único que tienen en común todos ellos, es que son la interfaz de usuario que permite que el jugador interactúe con el juego. Algunos de los tipos de controladores son:

2.1.1.1. Gamepad

Un gamepad se sostiene con las dos manos, y se utilizan los pulgares y dedos para proporcionar la entrada. Un gamepad tiene botones de acción, que normalmente se controlan con el pulgar derecho, y controles de dirección que se controlan con el pulgar izquierdo. El control de dirección puede ser un *d-pad*, que es una cruz que permite producir una entrada digital (0 ó 1) en cuatro direcciones, un stick analógico, que es una palanca omnidireccional que permite producir una entrada analógica, o el gamepad también puede tener las dos opciones de control de dirección. Un gamepad también puede tener botones en el centro (para funcionalidades como: start, select, modo, etc), un segundo stick analógico (normalmente usado para controlar la cámara dentro del juego), botones de acción en la parte trasera, y en

las últimas generaciones de videoconsolas, algunos gamepads también tienen *triggers* (gatillos en español) en la parte trasera, y, a diferencia del resto de botones de acción, permiten producir entradas analógicas en lugar de digitales [3, 4].

Figura 1. Gamepad de la videoconsola NES [4]

La *Figura 1* muestra un gamepad de la videoconsola NES (Nintendo Entertainment System), en el que se ven claramente el control de dirección (en este caso es un d-pad) para manejar con el pulgar izquierdo, los botones de acción para manejar con el pulgar derecho, y dos botones centrales para las funcionalidades de start y select.

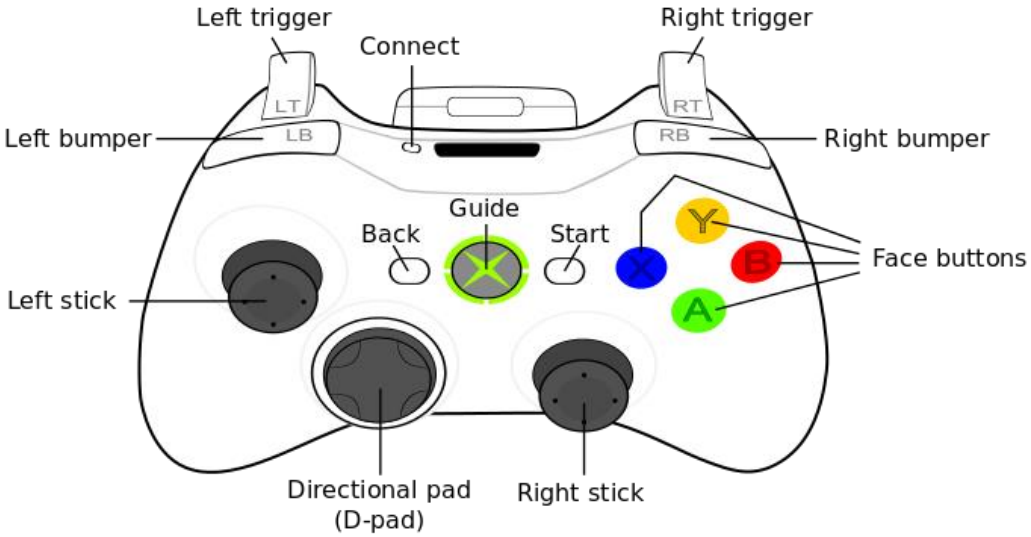


Figura 2. Gamepad de la videoconsola Xbox 360 [5]

La *Figura 2* muestra un gamepad de la videoconsola Xbox 360, en el que, a diferencia del gamepad de la videoconsola NES, tiene los dos tipos de controles de dirección, un d-pad y un stick analógico. También tiene más botones de acción, y además, tiene un segundo stick analógico, botones de acción en la parte trasera y dos triggers.

El gamepad es el principal controlador de la mayoría de las videoconsolas actuales.

2.1.1.2. Teclado y ratón

Un teclado y un ratón son actualmente el principal controlador para videojuegos de ordenador. Algunas videoconsolas también pueden controlarse con teclado y ratón. Normalmente el teclado se usa para mover el personaje, y el ratón para mover la cámara o

para apuntar en videojuegos de disparos. También existen videojuegos en los que sólo se usa el teclado, o en los que sólo se usa el ratón [3].

2.1.1.3. Pantalla táctil

Las pantallas táctiles son otro tipo de controlador, que permiten al jugador interactuar con el videojuego tocando la pantalla, sin necesidad de pulsar botones físicos. Es el controlador principal de los dispositivos móviles, y algunas videoconsolas tienen pantallas táctiles incorporadas en el gamepad, como el gamepad de la videoconsola Wii U [3].

2.1.1.4. Sensor de movimiento

Los sensores de movimiento son otra forma de interactuar con un videojuego. No todos los sensores usan la misma tecnología, por ejemplo, Wii Remote utiliza un acelerómetro y un sensor óptico [3]; PlayStation Move utiliza un acelerómetro, un sensor de velocidad, un magnetómetro, un sensor de temperatura interno, y se ayuda de visión artificial a través de PlayStation Eye para capturar el movimiento del jugador [6]; Kinect utiliza una cámara RGB y un sensor de profundidad para capturar el movimiento del jugador usando visión artificial [7].

2.1.1.5. Otros

Los controladores nombrados anteriormente son los controladores más utilizados actualmente, pero hay muchos más tipos de controladores, tanto controladores antiguos y que ya no se usan o apenas se usan, como pueden ser el joystick, el paddle y el trackball, y controladores específicos de algunos juegos o tipos de juegos, como pueden ser un volante y unos pedales, una pistola de luz, una guitarra, un teclado (instrumento musical), unos tambores, una plataforma de baile, un micrófono, o incluso un urinal interactivo [3].

2.1.2. Plataformas

Una plataforma es una combinación de componentes electrónicos o hardware que, junto con un software, permite que un videojuego se ejecute [8]. Las plataformas más populares son el PC, las videoconsolas, los dispositivos portátiles y las máquinas arcade.

Una videoconsola es un sistema electrónico de entretenimiento que ejecuta videojuegos contenidos en cartuchos, discos ópticos, discos magnéticos, tarjetas de memoria o cualquier dispositivo de almacenamiento [9].

De estas plataformas, sólo las videoconsolas y las máquinas arcade han sido diseñadas exclusivamente para ejecutar videojuegos [2], aunque en las últimas generaciones de videoconsolas, también se han añadido a las videoconsolas características de multimedia, internet, tiendas virtuales y servicios en línea [9].

Las máquinas arcade han perdido popularidad debido a que el PC y las videoconsolas son más rentables y tienen un hardware superior, aunque en Japón siguen teniendo éxito [10].

Las videoconsolas han sido clasificadas por generaciones de videoconsolas. Esta clasificación es determinada por la fecha de lanzamiento y la tecnología existente en el momento. Algunas generaciones están señaladas por los bits del ancho de bus del procesador, pero una videoconsola de generación superior no tiene que tener necesariamente más bits de ancho de bus, ya que la potencia del procesador no sólo depende del ancho de bus, también depende de su estructura y velocidad. En las videoconsolas de las últimas generaciones, ya no sólo hay que tener en cuenta la potencia de la CPU, sino también la de la GPU. Actualmente hay 8 generaciones, y las principales videoconsolas de la octava generación son: Wii U, PlayStation

4 y Xbox One como videoconsolas de sobremesa, y PlayStation Vita, Nintendo 3DS, Nintendo 3DS XL y Nintendo 2DS como videoconsolas portátiles [9].

2.1.3. Géneros

Los videojuegos se clasifican por géneros. Estos géneros, a diferencia de los géneros de libros y películas, no dependen de la ambientación (fantasía, ficción, etc) o del estilo (drama, comedia, etc), sino que dependen del *gameplay* [11]. El gameplay o jugabilidad es cómo se juega, incluyendo las reglas del juego, la trama, los objetivos y cómo conseguirlos, y también la experiencia del jugador [12].

Un juego puede pertenecer a varios géneros. Además, un juego no pertenece simplemente a un género, sino que suele pertenecer a un subgénero de un determinado género. Esto ha hecho que haya distintas clasificaciones de géneros, en las que un subgénero en una clasificación puede considerarse un género en otra y viceversa. Una posible clasificación de géneros es:

2.1.3.1. Acción

El género de acción incluye videojuegos donde la mayoría de retos son pruebas de habilidad. Estos videojuegos suelen requerir una gran velocidad de reacción y coordinación manos-ojos. Normalmente el jugador controla a un personaje, que debe moverse por un nivel, coger objetos, esquivando obstáculos, y luchando con enemigos. A menudo, al final de un nivel o grupo de niveles, el jugador debe vencer a un jefe enemigo, más difícil que otros enemigos. El jugador pierde el juego cuando el personaje se queda sin vida, que se reduce con los ataques de los enemigos y al colisionar con algunos obstáculos. Para ganar el juego, el jugador debe completar unos niveles determinados. También existen videojuegos interminables, con un número infinito de niveles, y la meta del jugador es conseguir la máxima puntuación posible [13].

El género de acción tiene varios subgéneros:

2.1.3.1.1. Beat ‘em up y Hack and slash

Este tipo de juegos de acción consiste en que el personaje debe luchar contra un gran número de enemigos para completar el nivel. Al final de cada nivel suele haber una lucha contra un jefe enemigo. La diferencia entre un *beat ‘em up* y un *hack and slash* es en el primero se lucha sin armas, y en el segundo se lucha con armas cuerpo a cuerpo. *Streets of Rage* es un ejemplo de *beat ‘em up*, mientras que *God of War* es un *hack and slash*, ya que el personaje utiliza armas [14].

2.1.3.1.2. Disparos

En los videojuegos de disparos o *shooters*, el personaje utiliza armas, normalmente una pistola, u otro tipo de armas de largo alcance. El principal objetivo de estos videojuegos es disparar a los enemigos y completar misiones sin que el personaje muera [15].

Hay varios tipos de videojuegos de disparos:

- En un ***shoot ‘em up***, el personaje se puede mover arriba, abajo, izquierda y derecha por la pantalla, y debe disparar lo más rápido posible a la gran cantidad de enemigos que va apareciendo por la pantalla. *Space Invaders* es un *shoot ‘em up* [15].
- En un videojuego de **galería de tiro**, el jugador debe mover un cursor que aparece en la pantalla usando un gamepad para señalar a donde apuntan los disparos. También se pueden jugar con una pistola de luz. Un videojuego de este subgénero es *Shootout* [15].

- Los videojuegos de **disparos con pistola de luz** son videojuegos de galería de tiro, en los que se utiliza una pistola de luz, u otros dispositivos apuntadores. Todos los videojuegos de la serie *Time Crisis* pertenecen a esta categoría [15].
- La principal característica de los videojuegos de **disparos en primera persona** es que simula que el jugador ve el contenido del juego desde el punto de vista del personaje. Este tipo de videojuegos también se conoce como FPS (*first-person shooter*). *Doom*, *Counter-Strike* y *Call of Duty* son juegos de disparos en primera persona [15].
- Los videojuegos de **disparos en tercera persona** se caracterizan por una vista de cámara de tercera persona, que muestra completamente al personaje y sus alrededores. Este tipo de videojuegos también se conoce como TPS (*third-person shooter*). *Gears of War* y *Vanquish* son juegos de disparos en tercera persona [15].
- Un videojuego de **disparos táctico** normalmente simula escaramuzas realistas de escuadras u hombre a hombre. Los videojuegos de la serie *Tom Clancy's Ghost Recon* son shooters tácticos [15].

2.1.3.1.3. Lucha

En los videojuegos de lucha, los personajes luchan entre ellos. Normalmente los personajes pueden realizar ataques especiales, que se ejecutan pulsando botones y moviendo el joystick para introducir rápidamente una secuencia determinada de comandos. Estos videojuegos muestran a los personajes de lado, incluso después de progresar a gráficos en tres dimensiones *Street Fighter* y *Tekken* son juegos de lucha [16].

2.1.3.1.4. Plataformas

En un videojuego de plataformas, el jugador controla a un personaje, que debe avanzar por el nivel saltando entre plataformas o evitando obstáculos. El jugador controla los saltos para evitar que el personaje se caiga de las plataformas o no salte cuando deba saltar. El elemento más común de este tipo de videojuego es la existencia de un botón para saltar. Existen videojuegos en los que el personaje salta de forma automática en algunas situaciones, pero eso no implica que sea un videojuego de plataformas. *Super Mario Bros* y *Crash Bandicoot* son juegos de plataformas en 2D y 3D respectivamente [17].

Dentro de los videojuegos de plataformas hay varios tipos, y los más importantes son:

- Un ***endless runner*** es un videojuego de plataformas en el que el personaje está constantemente avanzando por un mundo generado por procedimientos y teóricamente infinito. El objetivo es llegar lo más lejos posible, saltando para evitar chocarse con obstáculos. Debido a que se pueden realizar pocas acciones, estos videojuegos han tenido mucho éxito en las plataformas móviles, ya que en muchos casos, sólo es necesario tocar la pantalla para saltar. *Temple Run 2* es un *endless runner*, y se convirtió en el videojuego para móviles que más rápido se ha extendido cuando salió en Enero de 2013 [17].
- Un videojuego de **plataformas con puzles** utiliza la estructura de un videojuego de plataformas, donde el reto está en resolver puzles para poder completar el nivel. En el videojuego *The Lost Vikings* hay que usar tres personajes con diferentes habilidades para superar los obstáculos que nos impiden llegar a la salida y completar el nivel [17].
- Un videojuego ***run and gun*** es una mezcla entre videojuegos de plataformas y *shoot 'em up*. Se centran menos en saltos precisos entre o a plataformas para dar más importancia a disparar en todas direcciones. Normalmente, en estos videojuegos no se permite retroceder en el nivel, sólo se puede avanzar hacia adelante. No todos los videojuegos de plataformas en los que disparar es importante pertenecen a este tipo de

videojuegos. En videojuegos como *Megaman* y *Metroid*, disparar es una parte del videojuego, pero también dan mucha importancia a los saltos y a la exploración, a diferencia de *Metal Slug*, que sí es un videojuego *run and gun* [17].

2.1.3.1.5. Otros

Existen otros tipos de videojuegos de acción, como los videojuegos de laberintos estilo *Pac-Man*. Y los videojuegos de ritmos también podrían considerarse como videojuegos de acción, aunque también se pueden clasificar como videojuegos de música, que es otro género distinto. Incluso puede haber videojuegos de acción sin un subgénero claro, como *Frogger* [13].

2.1.3.2. Aventura

Un videojuego de aventura es un videojuego en el que el jugador asume el rol del protagonista en una historia interactiva guiada por la exploración y la resolución de puzles. Estos puzles se suelen resolver haciendo uso de objetos que se han encontrado en el entorno. Casi todos los videojuegos de aventura están diseñados para un solo jugador, ya que la importancia de la historia y los personajes hacen difícil diseñar estos videojuegos para varios jugadores [18].

El género de aventura tiene varios subgéneros:

2.1.3.2.1. Aventura textual

Una aventura textual o ficción interactiva es un videojuego en el que la historia del juego se transmite como texto. Este texto se va mostrando en función de las instrucciones textuales que escriba el jugador, y el videojuego convierte este texto en instrucciones que el videojuego pueda entender. *Colossal Cave Adventure* es una aventura textual, y además, es el que dio nombre al género de aventura [18].

2.1.3.2.2. Aventura gráfica

Las aventuras gráficas son juegos de aventura que utilizan gráficos para transmitir el entorno al jugador. Las aventuras gráficas tienen variedad de formas de entrada, desde conversores de texto hasta interfaces de pantallas táctiles. Las aventuras *point-and-click* son un tipo común de aventuras gráficas, en las que el jugador utiliza un dispositivo apuntador, normalmente el ratón, para interactuar con el entorno y resolver los puzles. Los videojuegos de la serie *Monkey Island* son aventuras gráficas [18].

2.1.3.2.3. Aventura de puzles

Las aventuras de puzles ponen gran énfasis en los puzles lógicos, en lugar de los más tradicionales puzles de objetos. *Myst* y *Professor Layton* son aventuras de puzles [18].

2.1.3.2.4. Novela visual

Una novela visual es un videojuego de fricción interactiva, normalmente con gráficos estáticos, y usando un estilo artístico tipo anime en la mayoría de los casos. Las novelas visuales se distinguen de los otros tipos de videojuegos por su mínimo gameplay. Normalmente, la mayoría de la interacción del jugador está limitada a hacer click para que avance el texto, cambien los gráficos que se están mostrando y el sonido. La mayoría de novelas visuales tienen muchas historias o rutas y muchos finales posibles, y en determinados puntos del videojuego aparecerán decisiones de elección múltiple, cuya elección determinará la dirección por la que avanzará la historia. Este gameplay se compara con los libros de *Choose your own Adventure*. En Japón, este tipo de videojuegos constituyeron casi el 70% de los videojuegos para PC lanzados en 2006 [19].

Los principales géneros de las novelas visuales son: de contenido adulto o *eroge*, que normalmente tienen alguna escena sexual al completar alguna ruta del juego, como *Night Life*; de ciencia ficción como *Steins;Gate*; *nakige* o *utsuge*, que su principal objetivo es que el

jugador empatice con los personajes y llore con escenas emotivas para dejar un gran impacto en el jugador una vez que ha terminado el videojuego, como *Clannad*; y de horror como *Umineko no Naku no Koro Ni* [19].

2.1.3.3. Acción-aventura

Un videojuego de acción-aventura es un videojuego que mezcla elementos de ambos géneros, especialmente los puzles. Estos videojuegos suelen requerir muchas de las habilidades físicas que requieren los videojuegos de acción, pero también ofrecen una historia, muchos personajes, un inventario, diálogos y otras características de los videojuegos de aventura [20].

Dentro de este género hay varios subgéneros. Dentro del género, hay distinción entre videojuegos en primera persona como *Metroid Prime*, y videojuegos en tercera persona como *The Legend of Zelda*. Se separan en otro subgénero los videojuegos de acción-aventura que sean *sandbox* o de mundo abierto como *Grand Theft Auto*. Los videojuegos que mezclan características de los videojuegos de plataformas y de los videojuegos de aventuras, pertenecen al subgénero de videojuegos de plataformas-aventura, que también se ha nombrado como *Metroidvania* debido a los videojuegos *Metroid* y *Castlevania* [20].

Para que un videojuego pertenezca a uno de los subgéneros anteriores, el único requisito que debe cumplir es que tenga una mezcla de determinadas características de los videojuegos de acción y de los de aventura, pero existen dos subgéneros en los que se añaden ciertas características y restricciones al videojuego.

2.1.3.3.1. Sigilo

En los videojuegos de sigilo o *stealth*, a diferencia de la mayoría de videojuegos de acción, el personaje debe evitar ser detectado por los enemigos en lugar de enfrentarse a ellos. Normalmente el personaje puede ocultarse en zonas oscuras o detrás de objetos. El juego también puede permitir que el personaje se disfrace para evitar ser detectado. Otra cosa a tener en cuenta a la hora de ocultarse es el ruido, ya que caminar sobre determinadas superficies hace más ruido, lo que implica que es más probable que el personaje sea detectado. Aunque estos videojuegos se centran en evitar a los enemigos, hay veces en las que es necesario dejar inconsciente a un enemigo o incluso matarlo para poder avanzar sin ser detectado, pero evitando que ese enemigo detecte al personaje y llame a más enemigos. Este género se hizo popular en 1998 tras el éxito de *Metal Gear Solid*, *Tenchu: Stealth Assassins* y *Thief: The Dark Project* [21].

2.1.3.3.2. Survival horror

Los videojuegos *survival horror* se centran en la supervivencia del personaje y en intentar asustar al jugador. En estos videojuegos, el personaje es más débil que en los videojuegos de acción, debido a munición, vida o velocidad limitadas, u otras limitaciones. Como el personaje es débil, tiene que solucionar puzles y evitar a los enemigos en lugar de recurrir a la violencia. Los niveles del juegos suelen ser oscuros y claustrofóbicos, y los enemigos suelen aparecer de forma inesperada. *Resident Evil* y *Silent Hill* son videojuegos *survival horror* [22].

2.1.3.4. Estrategia

Los videojuegos de estrategia son aquellos videojuegos en los que los principales factores para conseguir la victoria son un pensamiento y planificación habilidosos. Específicamente, el jugador debe planificar una serie de acciones contra uno o más oponentes, normalmente con el objetivo de reducir las fuerzas del enemigo. La mayoría de videojuegos de estrategia tienen elementos de guerra. También pueden retar la habilidad del jugador de explorar o gestionar una economía [23].

El género de estrategia tiene varios subgéneros:

2.1.3.4.1. 4X

4X es un subgénero de juegos de estrategia en el que el jugador controla un imperio y "explora, expande, explota y extermina". Estos videojuegos tienen un gameplay muy complejo. Pone énfasis en el desarrollo económico y tecnológico, al igual que en una gran variedad de formas no militares para conseguir la supremacía. Se puede tardar mucho en completar el juego debido a que la micro-gestión que se necesita para mantener el imperio aumenta conforme el imperio crece. *Sid Meier's Civilization* popularizó el nivel de detalle que se convirtió en el elemento básico del género [24].

2.1.3.4.2. Artillería

Los videojuegos de artillería son videojuegos, normalmente por turnos, en los que tanques se atacan unos a otros. Estos videojuegos están entre los primeros videojuegos desarrollados para ordenador. El tema de estos videojuegos es una extensión de los usos originales de los ordenadores, que se usaban para calcular la trayectoria de cohetes y hacer otros cálculos relacionados con lo militar. El videojuego *Scorched Earth*, lanzado en 1991, se considera el arquetipo moderno del género, y en él se basan videojuegos como *Worms* y *Hogs of War* [25].

2.1.3.4.3. Estrategia en tiempo real

En los videojuegos de estrategia en tiempo real o RTS (*real-time strategy*), los jugadores posicionan y manejan las unidades y estructuras que controlan para asegurar áreas del mapa o destruir las posesiones del oponente. Normalmente es posible crear unidades y estructuras adicionales gastando recursos. Estos recursos se consiguen controlando ciertas zonas del mapa o teniendo ciertos tipos de unidades y estructuras para conseguirlos. En concreto, en un videojuego típico de este género hay que recolectar recursos, construir bases, desarrollar tecnologías y controlar unidades, y en tiempo real, mientras el oponente también está manejando sus unidades y estructuras. Ejemplos de videojuegos de este género son *Age of Empires*, *Warcraft* y *Starcraft* [26].

Tower defense es un subgénero de los videojuegos de estrategia en tiempo real. El objetivo de este tipo de videojuegos es intentar evitar que los enemigos crucen un mapa. Para ello, el jugador puede construir trampas para ralentizar a los enemigos y torres que les disparan cuando pasan cerca. Las torres y los enemigos suelen tener habilidades. Las torres cuestan recursos construirlas, y también se pueden mejorar usando recursos. Estos recursos se consiguen cuando se derrota a un enemigo. Algunos ejemplos son *GemCraft* y *Plants vs. Zombies* [27].

2.1.3.4.4. Estrategia por turnos

La única diferencia entre los videojuegos de estrategia en tiempo real y los videojuegos de estrategia por turnos o TBS (*turn-based strategy*) es que se juega por turnos. Como cada jugador sólo puede jugar durante su turno, esto permite que se pueda dedicar más tiempo a planificar las acciones que se realizarán. *Heroes of Might and Magic* es una serie de videojuegos de estrategia por turnos [28].

2.1.3.4.5. Tácticas en tiempo real

Los videojuegos de tácticas en tiempo real o RTT (*real-time tactics*) simulan las consideraciones o circunstancias de los operativos de guerra y las tácticas militares. A diferencia de los videojuegos de estrategia, este tipo de videojuegos carece de la gestión de recursos, y la construcción de bases y unidades. Los videojuegos de tácticas se centran en os aspectos tácticos y operativos de la guerra, tales como las formaciones de las unidades o la

explotación del terreno para conseguir una ventaja táctica. Los combates en los videojuegos de la serie *Total War* son un ejemplo de tácticas en tiempo real [29].

2.1.3.4.6. Tácticas por turnos

Al igual que en los videojuegos de estrategia en tiempo real y por turnos, la diferencia entre los videojuegos de tácticas en tiempo real y los videojuegos de tácticas por turnos o TBT (*turn-based tactics*) son los turnos. En este sentido, aunque los dos subgéneros tienen sus raíces en los juegos de guerra de miniaturas, como *Warhammer Fantasy*, los videojuegos de tácticas por turnos conservan la mecánica de los turnos [30]. Los videojuegos de la serie *Fire Emblem* incentivan el uso de las mejores tácticas posibles para completar las misiones, ya que, si un personaje es derrotado en una misión, ese personaje no estará disponible para el resto de misiones del videojuego.

2.1.3.4.7. Wargame

Los videojuegos de guerra o *wargames* son videojuegos de estrategia en los que se enfatizan las guerras estratégicas o tácticas en un mapa, así como representar la historia lo más parecida a los hechos reales. Una serie de videojuegos de este género es *Nobunaga's Ambition* [23].

2.1.3.5. Rol

En un videojuego de rol o RPG (*role-playing game*), el jugador controla a un personaje (o varios miembros de un grupo) en un mundo ficticio. Los videojuegos de rol tienen sus orígenes en los juegos de rol de lápiz y papel, por lo que usan casi la misma terminología, escenarios y mecánicas. Para completar el videojuego, hay que realizar una serie de misiones o llegar a la conclusión de la historia. El jugador explora el mundo, mientras resuelve puzles y realiza combates tácticos, normalmente por turnos. Una característica clave de estos videojuegos es que los personajes mejoran su poder y habilidades, normalmente ganando puntos de experiencia y subiendo niveles [31].

Los videojuegos de rol se han clasificado en dos estilos: WRPG (*western role-playing game*) y JRPG (*japanese role-playing game*). Por lo general, los WRPGs suelen tener unos gráficos oscuros y realistas, personajes mayores, y se centran más en la libertad y las mecánicas del juego. Por otra parte, los JRPGs suelen tener unos gráficos estilo anime, personajes jóvenes, y le dan una gran importancia a una historia lineal [31].

A los videojuegos de rol, a veces se le incluyen elementos de otros géneros, y dan lugar a:

2.1.3.5.1. RPG de acción

Un RPG de acción conserva todos los elementos característicos de un RPG, con la excepción del combate. El gameplay de los combates es el característico de los videojuegos de acción. La serie de videojuegos *Mass Effect* presenta el gameplay de un videojuego de disparos, pero todos los demás elementos de los videojuegos son elementos de un videojuego de rol [31].

2.1.3.5.2. RPG táctico

Un RPG táctico, no sólo cambia el gameplay del combate como un RPG de acción, también suele eliminar la exploración. El gameplay de los combates es como el de un juego de estrategia tradicional, como el ajedrez. En muchos de los RPGs tácticos, los combates son muy largos y difíciles. Aunque este tipo de videojuegos era muy popular en Japón, no eran muy populares en otros lugares, fue el lanzamiento de videojuegos como *Disgaea* y *Final Fantasy Tactics* para la videoconsola *PlayStation 1* lo que aumentó mucho su popularidad [31].

2.1.3.5.3. MMORPG

Un MMORPG (*massively multiplayer online role-playing game*) es un videojuego de rol, normalmente un RPG de acción, en el que múltiples jugadores están jugando a la vez en un mismo mundo a través de internet. El MMORPG más jugado en Marzo de 2014 es *World of Warcraft* [31].

2.1.3.5.4. Roguelike

Un videojuego *roguelike* se caracteriza por una generación de niveles procedural, un gameplay por turnos, gráficos basados en *tiles* y la muerte suele ser permanente (se pierde todo el progreso). El término *roguelike* proviene del juego *Rogue* de 1980, que definió las características del género [32].

2.1.3.6. Simulación

Los videojuegos de simulación intentan recrear situaciones de la vida real. Dependiendo de lo que intente recrear, los videojuegos de simulación pertenecerán a un género u a otro [33].

2.1.3.6.1. Construcción y gestión

En un videojuego de construcción y gestión, el jugador construye, expande o gestiona comunidades o proyectos ficticios con recursos limitados. Este tipo de videojuegos no tienen ninguna condición de victoria, el jugador puede seguir ampliando y gestionando la ciudad, imperio o negocio hasta que, por tomar decisiones equivocadas, se quede en bancarrota. Hay videojuegos de construcción de ciudades como *Sim City*, o de gestión de negocios como *Game Dev Tycoon* [34].

2.1.3.6.2. Deportes

Un videojuego de deportes simula la práctica de deportes. Este tipo de juegos puede centrarse en jugar al deporte, como la serie de videojuegos *FIFA*, o puede centrarse en la estrategia y la organización, como la serie de videojuegos *Football Manager* [35].

2.1.3.6.3. Vehículos

Los videojuegos de vehículos intentan proporcionar al jugador una interpretación realista sobre operar varios tipos de vehículos (automóviles, aviones, barcos, etc). En algunos videojuegos, las físicas y retos pueden ser tan realistas que es necesario gestionar el combustible. Los videojuegos de carreras, como *Gran Turismo*, aunque se suelen incluir en los videojuegos de deportes, según *Andrew Rollings* y *Ernest Adams* "desde un punto de vista de diseño, pertenecen a los simuladores de vehículos" [36].

2.1.3.6.4. Vida

En un videojuego de simulación de vida, el jugador debe mantener y desarrollar una población de organismos. Puede tratar sobre individuos y sus relaciones, o puede simular un ecosistema. *Nintendogs* y *The Sims* son videojuegos de simulación de vida [37].

2.1.3.6.5. Otros

Hay simuladores de muchos más temas: de comercio, como *Patrician III*; de fotografía, como *Afrika*; de medicina, como *Surgeon Simulator 2013*; e incluso existe *Goat Simulator*, un simulador en el que el jugador controla una cabra [33].

2.1.3.7. Otros

Existen muchos más tipos de videojuegos: como los MMOs, que son videojuegos que se juegan por internet que soportan un gran número de jugadores; videojuegos musicales, en los que el jugador tiene que proporcionar una entrada (usando teclado, ratón, plataforma de baile, periféricos similares a instrumentos musicales, etc) al ritmo de la música; juegos de fiesta, que

tienen una gran variedad de minijuegos para que jueguen a la vez varios jugadores; juegos de mesa y de cartas, que son versiones de los juegos de mesa y cartas tradicionales como el ajedrez o el reversi; videojuegos educativos, que intentan enseñar a través del videojuego; etc [38].

2.1.4. Impacto en la sociedad

2.1.4.1. Demografía

A lo largo del tiempo, se ha tenido la idea de que los videojuegos eran exclusivos para gente joven, pero algunos estudios han demostrado que esa idea es errónea. Mientras que el mercado para los adolescentes y hombres adultos jóvenes, son las otras demografías las que tienen un crecimiento más significante. La *Entertaiment Software Association* (ESA) proporcionó el siguiente resumen para 2011 basado en un estudio de casi 1200 hogares americanos llevado a cabo por *Ipsos MediaCT* [8]:

- El jugador medio tiene 30 años y lleva jugando videojuegos 12 años. El 82% de los jugadores tienen más de 18 años.
- El 42% de todos los jugadores son mujeres, y las mujeres es una de las demografías que más rápido están creciendo en la industria del videojuego.
- Las mujeres adultas representan un mayor porcentaje de la población de jugadores (37%) que los niños menores de 18 años (13%).
- El 29% de los jugadores tienen más de 50 años, y en 1999 eran un 9%.
- El 65% de los jugadores juega videojuegos con otros jugadores en persona (no online).
- El 55% de los jugadores juega videojuegos en sus teléfonos o en dispositivos portátiles.

2.1.4.2. Multijugador

Tradicionalmente, los videojuegos han sido una experiencia social. Los videojuegos multijugador son aquellos que se pueden jugar competitivamente, o cooperativamente usando varios dispositivos de entrada, o con *hotseating* (en videojuegos por turnos, varios jugadores pueden jugar en el mismo dispositivo jugando al videojuego por turnos [39]) [8].

2.1.4.3. Teorías de efectos positivos de los videojuegos

Se ha demostrado que los jugadores de videojuegos tienen mejor coordinación ojo-mano y mejores habilidades visores-motoras, como la resistencia a distracciones, sensibilidad a la información en la visión periférica, y habilidad para contar objetos presentados brevemente, que las personas que no juega a videojuegos [40].

Los investigadores han descubierto que estas habilidades se pueden adquirir entrenando con videojuegos de acción, que tienen retos que necesitan cambiar el centro de atención entre diferentes lugares, pero no con videojuegos que requieren concentración en un solo objeto [8].

En el libro *Everything Bad is Good for You* de Steven Johnson, el autor argumenta que un videojuego demanda más del jugador que los juegos tradicionales como el *Monopoly*. Para experimentar un videojuego, el jugador primero debe determinar los objetivos, al igual que cómo completarlos. Después debe aprender los controles del videojuego, así como el funcionamiento de la interfaz humano-máquina. Además de esas habilidades, que al cabo de un tiempo se vuelven fundamentales, los videojuegos se basan en que el jugador navega (y después domina) un sistema muy complejo con muchas variables. Esto requiere una gran capacidad analítica, así como flexibilidad y adaptabilidad. También argumenta que el proceso de aprender los límites, objetivos, y controles de un videojuego a menudo es un proceso muy

demandante, que utiliza muchas áreas de la función cognitiva. De hecho, la mayoría de los videojuegos requieren gran paciencia y concentración del jugador, y, al contrario de la percepción popular de que los videojuegos dan gratificación instantánea, los videojuegos retrasan la gratificación incluso más tiempo que otras formas de entretenimiento como las películas o incluso los libros [8].

Cheryl Olson sugiere que los videojuegos proporcionan alivio del estrés. Alrededor del 25% de niñas y del 49% de niños utilizan videojuegos violentos para liberar su ira. También sugiere que los videojuegos pueden tener beneficios sociales para los niños, como por ejemplo, proporcionar un tema de conversación y algo con lo que los niños pueden conectar y les ayuda a hacer amigos. Los videojuegos también pueden ayudar a aumentar la autoestima de los niños cuando son capaces de hacer algo bien en un videojuego [40].

Otros estudios han examinado los beneficios de los videojuegos multijugador en el ambiente familiar; el uso de videojuegos en las aulas; los videojuegos online; y los efectos de jugar a videojuegos en la destreza, los conocimientos de informática, la memoria semántica y las habilidades de resolución de problemas [40].

2.1.4.4. Teorías de efectos negativos de los videojuegos

Algunos científicos proponen que algunas condiciones, como el trastorno de personalidad antisocial, pueden determinar aquellos quienes tienen más riesgo de cometer actos violentos después de jugar a videojuegos. Además, las personas predispuestas a tener un comportamiento violento, pueden tener más riesgo de ser afectados adversamente al jugar a videojuegos violentos que otras personas [40].

Los videojuegos violentos pueden tener el efecto de reforzar los estereotipos sexistas. En 1998, un estudio realizado por Tracy Dietz en la Universidad de Florida Central, descubrió que de 33 videojuegos probados, el 41% no tenía personajes femeninos, en el 28% las mujeres se trataban como objetos sexuales, el 21% representaba violencia contra las mujeres, y el 30% no representaba a la población femenina. Por lo tanto, la caracterización de las mujeres tendía a ser estereotípica: muy sexualizada (bellezas con grandes pechos y caderas), dependiente (víctima o "damisela en apuros"), oponentes (malvadas o como obstáculos para el objetivo del videojuego) y triviales (con roles insignificantes). También es cierto que en este estudio no se incluyó una amplia gama de videojuegos, y que incluía videojuegos antiguos que no representan los estándares actuales de la industria. Los descubrimientos de Dietz son apoyados por un estudio realizado en 2003 por Children Now. Este estudio descubrió que los estereotipos de género impregnan la mayoría de los videojuegos: los personajes masculinos (52%) eran más propensos a luchar que los personajes femeninos (32%); casi el 20% de los personajes femeninos estaban hiper-sexualizados, mientras que el 35% de los personajes masculinos eran extremadamente musculosos [40].

También se pueden dar problemas de adicción. Hay casos en los que los jugadores juegan compulsivamente, aislándose de su familia y amigos u otras formas de contacto social, y se centran casi totalmente en logros de los videojuegos en lugar de en otros acontecimientos de la vida más importantes [40].

Otro problema que puede ocurrir es el acoso en línea o los comportamientos de abuso. La naturaleza anónima de internet puede ser un factor que estimule el comportamiento antisocial. Sin una mínima amenaza o castigo, algunos pueden encontrar más fácil tener comportamientos negativos en los videojuegos online [40].

2.2. Desarrollo de videojuegos

Los videojuegos son software, y, al igual que el desarrollo de software, el desarrollo de videojuegos tiene un proceso de desarrollo, aunque es un poco diferente al proceso de desarrollo de software, ya que los videojuegos también hay que tener en cuenta el arte, el audio y el gameplay.

2.2.1. Roles en un equipo de desarrollo de videojuegos

La estructura de un equipo de desarrollo, normalmente está compuesta por cinco disciplinas básicas: ingenieros, artistas, diseñadores de videojuegos, productores, y personal de gestión y apoyo (marketing, abogados, apoyo técnico, administrativos, etc). Además, en cada disciplina pueden haber varios roles [41].

2.2.1.1. Ingenieros

Los ingenieros diseñan e implementan el software que hace que el videojuego y las herramientas funcionen. Los ingenieros se suelen clasificar en 2 grupos básicos:

- Los **desarrolladores de videojuegos** desarrollan el *motor* del videojuego y el propio videojuego. Algunos ingenieros centran su carrera en un solo sistema del motor, como el sistema de rendering, la inteligencia artificial, el audio, o las colisiones y físicas. Algunos ingenieros se centran en la programación del gameplay y en el scripting, mientras que otros prefieren trabajar a nivel del sistema y no se involucran demasiado en el funcionamiento del videojuego. Algunos ingenieros son generalistas, y pueden ayudar en cualquier problema que surja durante el desarrollo.
- Los **desarrolladores de herramientas** desarrollan las herramientas que permiten al resto del equipo trabajar eficazmente, como por ejemplo, editores de escenarios.

A veces se pide a los ingenieros senior que asuman un rol de líder técnico. El ingeniero líder sigue diseñando y escribiendo código, pero también ayudan a gestionar la agenda del equipo, toman decisiones con respecto a la dirección técnica del proyecto, y a veces, también gestionan a las personas directamente desde una perspectiva de los recursos humanos [41].

2.2.1.2. Artistas

Los artistas producen todo el contenido visual y de audio del videojuego, y la calidad de su trabajo puede, literalmente, hacer o romper un videojuego. Los artistas se clasifican en:

- Los **artistas de concepto** crean bocetos y dibujos que proporcionan al equipo una visión de cómo se verá el videojuego terminado.
- Los **modeladores 3D** producen la geometría tridimensional, o modelos 3D, para todo en el mundo virtual del videojuego. Esta disciplina se suele dividir en 2 sub-disciplinas: modeladores del primer plano y modeladores del segundo plano. Los primeros crean los objetos, personajes, vehículos, armas, y otros objetos del mundo del videojuego, mientras que los otros crean la geometría del fondo estático del mundo, como el terreno, los edificios, puentes, etc).
- Los **artistas de texturas** crean las imágenes bidimensionales conocidas como texturas, que se aplican a las superficies de los modelos 3D para proporcionar detalle y realismo.
- Los **artistas de iluminación** colocan todas las fuentes de luz, estáticas y dinámicas, en el mundo del videojuego, y trabajan con el color, la intensidad, y la dirección de la luz para maximizar el impacto emocional de cada escena.

- Los **animadores** dan a los personajes y a los objetos del videojuego movimiento. Los animadores sirven como actores en la producción de videojuego, así como lo hacen en la producción de una película CG (*computer graphics*). Sin embargo, un animador de videojuegos debe tener un conjunto de habilidades único para producir animaciones que sean perfectamente compatibles con las bases tecnológicas del motor del videojuego.
- Los **actores de captura de movimientos** proporcionan datos sobre un conjunto de movimientos, que los animadores procesan para integrarlos en el videojuego.
- Los **diseñadores de sonido** trabajan junto con los ingenieros para producir y mezclar los efectos de sonido y la música del videojuego.
- Los **actores de voz** proporcionan las voces de los personajes de los videojuegos.
- Muchos videojuegos tiene uno o más **compositores**, que componen una banda sonora para el videojuego.

Al igual que con los ingenieros, a menudo se nombran líderes del equipo a los artistas senior [41].

2.2.1.3. Diseñadores de videojuegos

El rol de los diseñadores de videojuegos es diseñar la parte interactiva de la experiencia del jugador, es decir, el gameplay. No todos los diseñadores trabajan en los mismos niveles de detalle. Algunos diseñadores de videojuegos trabajan en un nivel superior, determinando la historia, la secuencia del conjunto de capítulos o niveles, y los objetivos de alto nivel. Otros diseñadores trabajan en los niveles individuales u otras áreas geográficas del mundo virtual, colocando el fondo estático, determinando de dónde y cuándo aparecerán enemigos, colocando suministros como armas y botiquines, diseñando puzles, etc. Otros diseñadores trabajan a un nivel muy técnico, trabajando con los ingenieros del gameplay y/o escribiendo código (a menudo en lenguajes de scripting de alto nivel). Algunos diseñadores de videojuegos eran ingenieros, que decidieron que preferían jugar un rol más activo en determinar cómo se iba a jugar al videojuego.

Algunos equipos tienen uno o más escritores. El trabajo de un escritor en un videojuego puede ser, desde colaborar con los diseñadores de videojuegos para crear la historia del videojuego, hasta escribir líneas de un diálogo [41].

2.2.1.4. Productores

El rol del productor cambia de un estudio a otro. En algunas compañías de videojuegos, el trabajo del productor es gestionar la agenda y servir como director de recursos humanos. En otras compañías, los productores sirven como diseñadores de videojuegos senior. En otros estudios, el productor es un enlace entre el equipo de desarrollo y la unidad de negocio de la compañía (finanzas, abogados, marketing, etc). Algunos estudios pequeños ni siquiera tienen un productor [41].

2.2.1.5. Personal de apoyo y gestión

El equipo de gente que desarrolla el videojuego, normalmente está respaldado por un equipo fundamental de personal de apoyo. Este personal incluye al equipo de dirección ejecutiva del estudio, al departamento de marketing, a personal administrativo, y al departamento de información y tecnología, que compran, instalan, y configuran el hardware y el software para el equipo, y proporcionan apoyo técnico [41].

2.2.1.6. Editores y estudios

Aunque no suelen pertenecer al equipo de desarrollo, un colectivo muy importante en el mundo del desarrollo de videojuegos son las compañías editoras o *publishers*.

El marketing, la manufacturación, y la distribución de un videojuego normalmente la realiza un editor, y no el propio estudio. Muchos estudios de videojuegos no están afiliados con un editor, si no que venden cada videojuego que producen al editor que les ofrezca un mejor trato. Otros estudios trabajan exclusivamente con un solo editor, ya sea por un contrato a largo tiempo, o como un estudio subsidiario de la compañía editora. Los estudios internos, o *first-party*, son estudios que pertenecen a los fabricantes de videoconsolas. Estos estudios desarrollan videojuegos exclusivos para el hardware de entretenimiento fabricado por la compañía padre [41].

2.2.2. Proceso de desarrollo de videojuegos

El proceso de desarrollo se puede dividir en varias etapas:

2.2.2.1. Pre-producción

La etapa de pre-producción o fase de diseño es la fase de planificación del proyecto centrada en la idea y el desarrollo del concepto y la producción de documentos del diseño inicial. El objetivo del desarrollo del concepto es producir una documentación clara y fácil de entender, que describa todas las tareas, planificaciones y estimaciones del equipo de desarrollo. El plan de producción es el conjunto de elementos que se producen en esta fase [42].

La documentación del concepto se puede separar en 3 fases o documentos:

- El **concepto general** es una descripción de unas pocas líneas del videojuego.
- La **propuesta de videojuego** es un documento resumen corto con la intención de mostrar los puntos fuertes del videojuego y detallar porque será rentable desarrollarlo.
- El **plan del videojuego** es un documento más detallado que la propuesta de videojuego. Incluye toda la información que se produce sobre el videojuego. Incluye el concepto general, el género del videojuego, la descripción del gameplay, las características, la historia, el público al que va dirigido, las plataformas hardware, la planificación estimada, el análisis del marketing, los requisitos del equipo, y el análisis de riesgos.

La fase final de la pre-producción también se conoce como análisis técnico, donde se producen documentos del videojuego más detallados:

- Antes de empezar la producción, el equipo de desarrollo produce una primera versión del **documento de diseño del videojuego**, incorporando todo o casi todo el material de la documentación del concepto. El documento de diseño describe el concepto del videojuego y los elementos más importantes del gameplay en detalle. También puede incluir bocetos iniciales de varios aspectos del videojuego.
- Crear **prototipos** de ideas de gameplay y características es una actividad importante que permite a los programadores y diseñadores de videojuegos experimentar con diferentes algoritmos y escenarios usables para un videojuego. Los prototipos pueden crearse antes de completarse el documento de diseño del videojuego, y ayudan a determinar las características que especifica el diseño. También pueden crearse durante el desarrollo para probar nuevas ideas.

2.2.2.2. Producción

La producción es la etapa principal del desarrollo, donde se producen los activos y el código fuente del videojuego.

En esta etapa, los programadores escriben el código fuente, los artistas crean los activos del videojuego (*sprites* o modelos 3D), los ingenieros de sonido crean los efectos de sonido y los compositores componen la música del videojuego, los diseñadores de niveles crean los niveles, los escritores crean los diálogos para las cinemáticas y los NPCs (*non-player carácter*), y los diseñadores de videojuegos siguen desarrollando el diseño del videojuego durante toda la fase de producción [42].

2.2.2.2.1. Diseño

El diseño de un videojuego es un proceso esencial y cooperativo de diseñar el contenido y las reglas del videojuego, que requiere competencias artísticas y técnicas, así como habilidad en la escritura.

Durante el desarrollo, el diseñador de videojuegos implementa y modifica el diseño del videojuego para que refleje la visión actual del videojuego. A menudo se añaden o quitan niveles o características. El arte puede evolucionar, y el trasfondo puede cambiar. Se puede dirigir a una nueva plataforma, así como a una nueva demografía. Todos estos cambios deben ser documentados y difundidos al resto del equipo [42].

2.2.2.2.2. Programación

La programación del videojuego la realizan uno o más programadores. Los programadores desarrollan prototipos para probar ideas, muchas de las cuales puede que nunca se incluyan en el videojuego finalizado. Los programadores incorporan nuevas características que exijan el diseño del videojuego, y arreglan los fallos que se producen durante el proceso de desarrollo. Incluso si se utiliza un motor existente, es necesario programar bastante para personalizar el videojuego [42].

2.2.2.2.3. Creación de niveles

Desde un punto de vista temporal, los primeros niveles de un videojuego son los que más se tarda en desarrollar. Mientras que los diseñadores de niveles y los artistas utilizan herramientas para crear niveles, piden características y cambios a las herramientas para que les permitan desarrollar más rápido y con mayor calidad. Las nuevas características pueden hacer que los niveles antiguos queden obsoletos, por lo que los primero niveles que se han desarrollado pueden desarrollarse y descartarse repetidamente. Los últimos niveles se pueden desarrollar más rápidamente, ya que el conjunto de características es más completo, y la visión del videojuego es más clara y estable [42].

2.2.2.2.4. Producción del arte

La producción del arte es el proceso de crear el arte 2D y 3D del videojuego. Un artista crea el arte del videojuego, como el arte conceptual, los sprites y los modelos de los personajes. Las ilustraciones del videojuego, tales como demos o capturas de pantalla, tienen un gran impacto en los clientes, porque las ilustraciones pueden juzgarse en los análisis, mientras que el gameplay no [43].

2.2.2.2.5. Producción del audio

La producción del audio se puede separar en 3 categorías: efectos de sonido, música, y doblaje.

La producción de efectos de sonido se hace, ya sea ajustando una muestra hasta conseguir el efecto deseado, o replicando el sonido con objetos reales.

La música puede ser sintetizada o tocada con instrumentos.

La música se puede presentar en un videojuego de varias formas:

- Música de ambiente, especialmente en momentos lentos del videojuego, donde la música intenta reforzar la escena del videojuego.
- La música también puede desencadenarse por eventos del videojuego.
- La música de acción, como en secuencias de persecución, batalla y caza, es de ritmo rápido.
- La música de los menús, al igual que la de los créditos, crea un impacto auditivo aunque no haya acción.

Los doblajes crean interactividad entre el gameplay y los personajes. Los doblajes dan personalidad a los personajes del videojuego [42].

2.2.2.2.6. Pruebas

Los testers (personal encargado de realizar las pruebas) empiezan a trabajar una vez que algo es jugable. El videojuego es jugable cuando un nivel, o un subconjunto del videojuego, puede usarse en una medida razonable. Al principio, se tarda relativamente poco en probar el videojuego, y los testers pueden estar trabajando en más de un videojuego. Conforme el videojuego se va acercando al final de su desarrollo, un solo videojuego normalmente hace uso de muchos testers a tiempo completo. Los testers prueban las nuevas características y hacen pruebas de regresión en las ya existentes. Las pruebas son vitales para los videojuegos complejos actuales, ya que simples cambios pueden llevar a consecuencias catastróficas.

Al final del desarrollo, las características y niveles se están terminando rápidamente, y hay más material que necesita ser probado que en cualquier otro momento del desarrollo. Los testers tienen que hacer pruebas de regresión para asegurarse de que las características que se terminaron hace meses siguen funcionando. Las pruebas de regresión son una de las tareas más vitales que se requieren para el desarrollo de software eficaz. Cuando se añaden nuevas características, pequeños cambios en el código pueden producir cambios inesperados en distintas partes del videojuego [42].

2.2.2.3. Hitos

Los proyectos de desarrollo de videojuegos comercial pueden requerir cumplir hitos establecidos por los editores. Los hitos marcan eventos importantes durante el desarrollo del videojuego, y se utilizan para seguir el progreso del videojuego [42].

No hay estándares en la industria para definir los hitos, ya que pueden variar dependiendo del editor, el año, o el proyecto. Algunos hitos comunes en ciclos de desarrollo de dos años son:

- El **primero jugable** es la versión del videojuego que contiene un gameplay y activos representativos. Es la primera versión con los principales elementos funcionales de gameplay. A menudo se basa en el prototipo creado en la pre-producción.
- La **versión alpha** es la etapa en la que la funcionalidad clave del gameplay está implementada, y los activos están parcialmente terminados. Un videojuego en alpha es jugable y tiene todas las características importantes. Estas características se revisan con pruebas y retroalimentación. Se pueden añadir nuevas características pequeñas, pero las características que no están implementadas se pueden descartar. Los programadores suelen centrarse en terminar el código existente, en lugar de añadir nuevas características.
- La **congelación del código** es la etapa en la que no se añade más código al videojuego, sino que se centra sólo en corregir fallos.

- La **versión beta** es una versión del videojuego completa, con todas las características y activos, donde sólo se solucionan errores. Esta versión no contiene fallos que eviten que el videojuego se pueda distribuir. En esta versión ya no se cambian las características, los activos, o el código.
- La **liberación del código** es la etapa donde todos los fallos están arreglados, y el videojuego está listo para distribuirse o enviarse al fabricante de la consola para que lo revise. Esta versión se prueba con el plan de pruebas de aseguramiento de calidad.
- La **versión a manufacturar** es la versión final del videojuego que se utiliza para la producción del videojuego.
- Los **momentos decisivos** son el tiempo extra y sin pagar que piden muchas compañías para cumplir los plazos e hitos, y afecta negativamente a los desarrolladores. Un equipo que no puede cumplir un plazo tiene el riesgo de que el proyecto sea cancelado, o se despidan empleados.

2.2.2.4. Post-producción

Una vez que el videojuego se ha distribuido, empieza la fase de mantenimiento del videojuego.

Los videojuegos desarrollados para videoconsolas apenas tenían un periodo de mantenimiento en el pasado. El videojuego distribuido tendría siempre todos los fallos y características que tuviera cuando fue publicado. Esto era la norma para la videoconsolas, ya que todas las consolas tenían un hardware casi idéntico, y la incompatibilidad, que es causante de muchos fallos, no era un problema.

En los últimos tiempos, ha crecido la popularidad de los videojuegos online para videoconsolas, y se han desarrollado videoconsolas con internet y servicios online. Ahora los desarrolladores pueden mantener el software a través de parches descargables.

El desarrollo para PC es diferente. Los desarrolladores intentan tener en cuenta la mayoría de configuraciones de software y hardware. Sin embargo, el número de posibles combinaciones de hardware y software lleva inevitablemente al descubrimiento de circunstancias que bloquean el videojuego, y que los programadores y los testers no tuvieron en cuenta.

Los programadores esperan un tiempo para el mayor número de reportes de fallos posible. Cuando creen que han conseguido suficiente retroalimentación, empiezan a trabajar en un parche. El parche puede tardar semanas o meses en desarrollarse, pero la intención es que arregle el mayor número de fallos del videojuego que no descubrieron en la última versión publicada, o en raras ocasiones, arreglar fallos no intencionados causados por parches anteriores. A veces, un parche puede incluir características o contenido extra, o incluso alterar el gameplay [42].

2.3. Motor de juegos

En el apartado “2.2. Desarrollo de videojuegos”, se ha mencionado varias veces motor o motor de videojuegos, pero ¿a qué nos referimos cuando hablamos de un motor?

El término “motor de videojuegos” o *game engine* surge a mediados de los años 90, refiriéndose a videojuegos de disparos en primera persona, como el popular *Doom* de id Software. *Doom* fue diseñado con una separación bien definida entre los componentes software (como el sistema de renderizado de gráficos tridimensionales, el sistema de detección de colisiones, o el sistema de audio) y los activos de arte, los mundos del videojuego, y las reglas del videojuego. El valor de esta separación se vuelve evidente cuando

los desarrolladores empiezan a licenciar videojuegos y a transformarlos en nuevos productos creando nuevo arte, diseños de mundos, armas, personajes, vehículos, y reglas del videojuego, con pequeños cambios al software del motor. Un motor de videojuegos es un software extensible y que se puede utilizar como base para muchos videojuegos diferentes, sin realizar modificaciones grandes [41].

2.3.1. Descripción de algunos motores de videojuegos

Existen una gran cantidad de motores de videojuegos, pero como en este trabajo de fin de grado trata de desarrollar un videojuego, y no de desarrollar un motor de videojuegos, sólo se va a hablar de unos pocos motores de videojuegos que podrían haberse usado para desarrollar el videojuego. Los motores de los que se va a hablar son gratuitos o tienen una versión gratuita. La descripción de XNA Framework y MonoGame se realizará en el capítulo "3. Booster", debido a que se va a realizar una descripción más detallada que las de estos motores.

2.3.1.1. Unity

Unity es una herramienta para desarrollar contenidos interactivos y distribuirlos en cualquier sitio. Unity permite crear videojuegos, recorridos arquitectónicos, simulaciones de entrenamiento online, o arte interactivo. Con respecto al desarrollo de videojuegos, permite desarrollar videojuegos tanto en 2D como en 3D.

Unity Editor es una interfaz totalmente integrada y extensible con la que se crean los videojuegos. Con Unity Editor se crean las escenas con terrenos, luces, sonidos, personajes, físicas, etc; se añade interacción mediante scripting, permite probar el videojuego y editarlo a la vez, y distribuir el videojuego a las plataformas elegidas.

En Unity, se escriben scripts de comportamiento simples en JavaScript, C# o Boo. Estos scripts se ejecutan en la plataforma .NET de código abierto, Mono.

Con Unity se pueden desarrollar videojuegos para:

- Móviles
  - iOS
  - Android
  - Windows Phone 8
  - BlackBerry 10
- Escritorio
  - Mac OS X
  - Windows
  - Linux
- Navegador Web
- Videoconsolas
  - Xbox 360
  - Xbox One
  - PlayStation 3
  - PlayStation Vita
  - PlayStation 4
  - Wii U

Existen 2 versiones de Unity: la versión gratuita y la versión Pro, que es de pago. La versión gratuita tiene menos características que la versión Pro, pero esto no impide que se puedan desarrollar y distribuir videojuegos [44].

Algunos de los videojuegos desarrollados con Unity son: *Archangelk*, *Broforce*, *Forced*, *Guns of Icarus Online*, *MechWarrior Tactics*, *Oddworld: New 'n' Tasty*, *Teslagrad*, *Trolls vs Vikings*, etc.

2.3.1.2. LibGDX

LibGDX es un framework de desarrollo de videojuegos en Java que proporciona una API unificada que funciona en todas las plataformas que soporta.

Con LibGDX se puede desarrollar para:

- Windows
- Linux
- Mac OS X
- Android (2.2+)
- BlackBerry
- iOS
- Java Applet
- JavaScript/WebGL

LibGDX permite escribir el código sólo una vez y distribuirlo a varias plataformas sin necesidad de modificar el código.

LibGDX permite programar a tan bajo nivel como se quiera, dando acceso directo al sistema de archivos, a los dispositivos de entrada, a los dispositivos de audio y a OpenGL mediante una interfaz unificada de OpenGL ES 2.0 y 3.0. Encima de estas facilidades de bajo nivel, está construido un conjunto de APIs que ayudan con tareas comunes del desarrollo de videojuegos como el renderizado de sprites y texto, crear interfaces de usuario, reproducir efectos de sonido y música, cálculos de álgebra lineal y trigonometría, conversión de JSON y XML, etc [45, 46].

LibGDX es gratuito y de código abierto, y con él se han desarrollado videojuegos como *Ingress* y *Oh My Goat*.

2.4. Estado del arte

Para explicar el estado del arte, el autor se ha documentado con el Libro Blanco del Desarrollo Español de los Videojuegos [47], que ofrece datos basados en un análisis exhaustivo de la industria del videojuego en España. Estos datos se recogieron entre Julio y Octubre de 2013.

2.4.1. Mercado de los videojuegos en el mundo

Se ha estimado que el sector de los videojuegos es la industria tecnológica con mayor crecimiento. Esto se debe a la venta online, a que los videojuegos cada vez alcanzan más ámbitos además del ocio, al impulso de la banda ancha en todo el mundo y a los nuevos modelos de negocio (free to play, publicidad, etc). Nos encontramos con un mercado global, ya que el aumento de conectividad a nivel mundial permite localizar y lanzar videojuegos en cualquier lugar del planeta.

Según la estimación, el mercado crecerá entre el 6,7% y el 10,5% cada año entre los años 2012 y 2016. Se estimó que los ingresos en el año 2013 serían 70.400 millones de dólares,

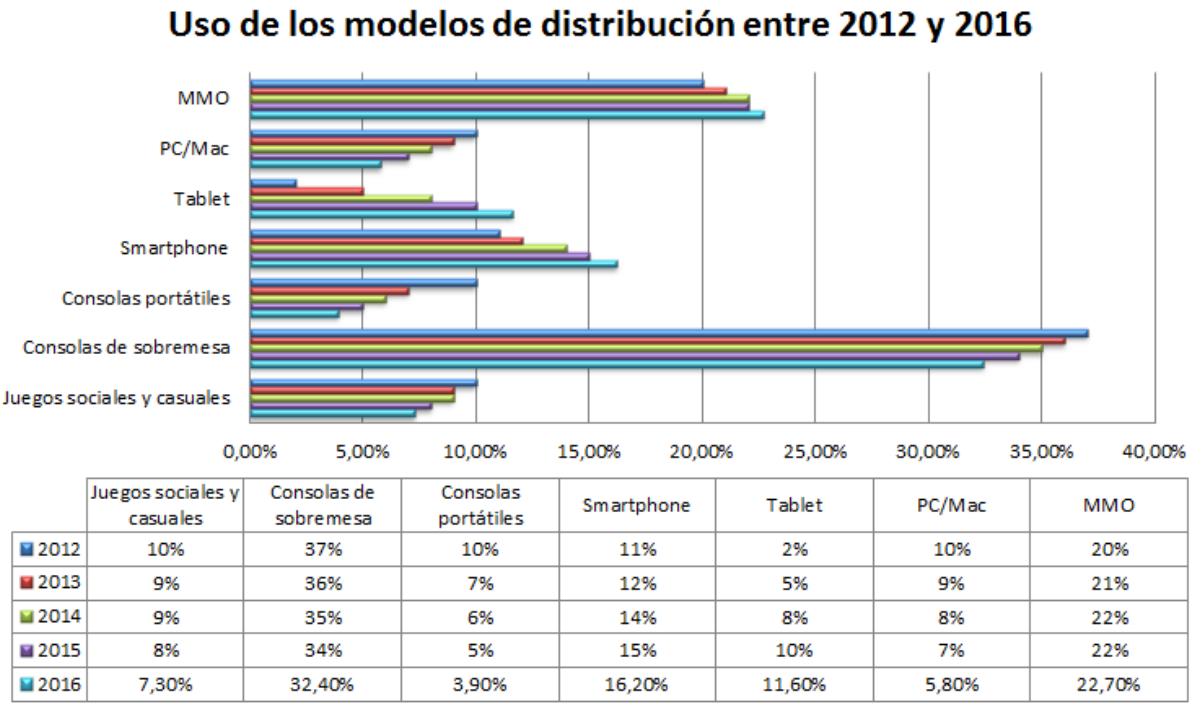
con un crecimiento del 6,2% con respecto al año 2012. Para el 2016 se estimaron unos ingresos de 86.100 millones de dólares.

Según los datos recogidos para realizar esta estimación, a finales de 2013 había 1.231 millones de personas que jugaban a videojuego, que equivale a más de la mitad de la población mundial conectada a internet.

La distribución de jugadores por las distintas regiones está representada con los datos de la *Figura 3*.

Región	Millones de jugadores	Porcentaje de población de la región	Porcentaje de internautas de la región
Norteamérica	192	55%	70%
Europa Occidental	180	45%	58%
China	180	14%	33%
Oriente Medio	145	16%	59%
Asia-Pacífico	140	14%	55%
Europa del Este	120	29%	59%
América Latina	115	20%	46%
Corea-Japón-Oceanía	85	43%	52%
India	75	6%	54%

Figura 3. Tabla comparativa de jugadores de una región con la población



	Juegos sociales y casuales	Consolas de sobremesa	Consolas portátiles	Smartphone	Tablet	PC/Mac	MMO
2012	10%	37%	10%	11%	2%	10%	20%
2013	9%	36%	7%	12%	5%	9%	21%
2014	9%	35%	6%	14%	8%	8%	22%
2015	8%	34%	5%	15%	10%	7%	22%
2016	7,30%	32,40%	3,90%	16,20%	11,60%	5,80%	22,70%

Figura 4. Gráfica comparativa de modelos de distribución utilizados entre 2012 y 2016

En esta estimación también se ha hecho una comparación el cambio de la cuota de mercado alcanzada por cada modelo de distribución entre los años 2012 y 2016, representada en la

Figura 4, donde se puede observar que los smartphones y tablets ganan mercado, mientras que otros modelos de distribución, sobre todo las consolas portátiles, pierden mercado.

2.4.2. Mercado de los videojuegos en España

Según los datos recogidos para realizar estos análisis, en España había 17 millones de jugadores en 2012. El número medio de plataformas utilizadas por jugador era 4,2. La plataforma más utilizada son las videoconsolas, siendo utilizada por un 76% de los jugadores.

Figura 5. Gráfica de tiempo invertido en los videojuegos por plataforma

En 2012, los españoles pasaron, en total, 27 millones de horas diarias jugando a videojuegos, un 30% más que el año anterior. La distribución de este tiempo por plataforma está representada en la *Figura 5*.

Los videojuegos suponen una gran cantidad de ingresos en el país, en 2012 los españoles gastaron 1.900 millones de euros en videojuego, un 17% más que en el año anterior.

Cabe destacar que el mercado de los videojuegos está creciendo en España a un ritmo mayor que en los países de la Unión Europea. El número de horas dedicadas a jugar a videojuegos en España aumentó en un 30% entre 2011 y 2012, mientras que en la Unión Europea sólo aumentó en un 18%. El porcentaje de jugadores aumentó un 13% en España, y un 8% en la Unión Europea. Pero el aumento más significativo está en el número de personas que pagaban por jugar, y no jugaban sólo a juegos gratuitos. Este porcentaje aumentó un 44% en España, y sólo un 17% en la Unión Europea.

2.4.3. Industria de los videojuegos en España

En la industria del videojuego de España existían 330 empresas cuando se realizó este estudio, con una distribución geográfica como la mostrada en la *Figura 6*, siendo la Comunidad de Madrid y Cataluña las comunidades con más empresas, teniendo más de la mitad de empresas del país.

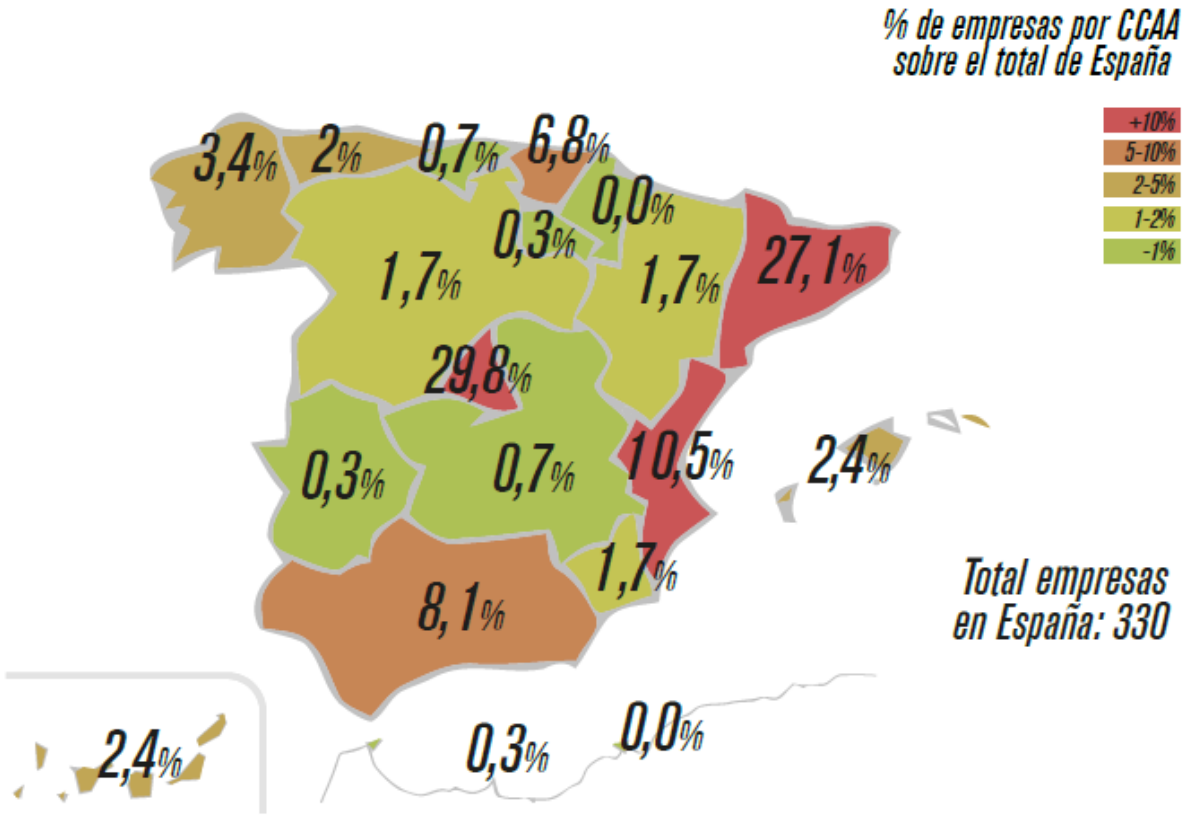


Figura 6. Distribución geográfica de empresas de videojuegos en España [47]

En su mayoría, las empresas de videojuego son pequeñas empresas, el 86,6% tiene menos de 25 empleados. La *Figura 7* representa la distribución de empresas según el número de empleados.

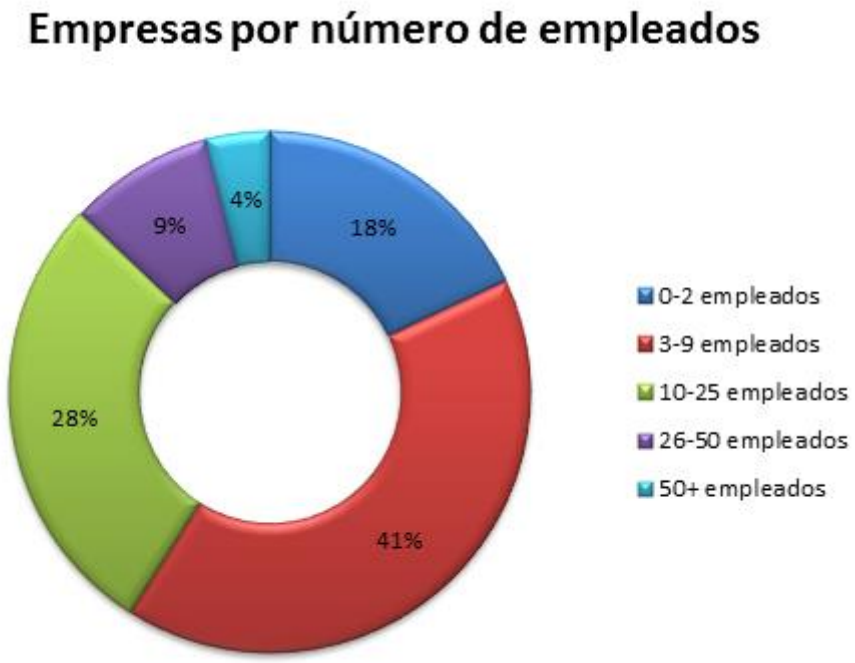


Figura 7. Distribución de empresas según el número de empleados

También cabe destacar que la mayoría de estas empresas, un 68% de ellas, tiene menos de 5 años de vida. La creación de tantas nuevas empresas se debió a la proliferación de nuevos modelos de negocio a través de internet y a la aparición de nuevos dispositivos móviles.

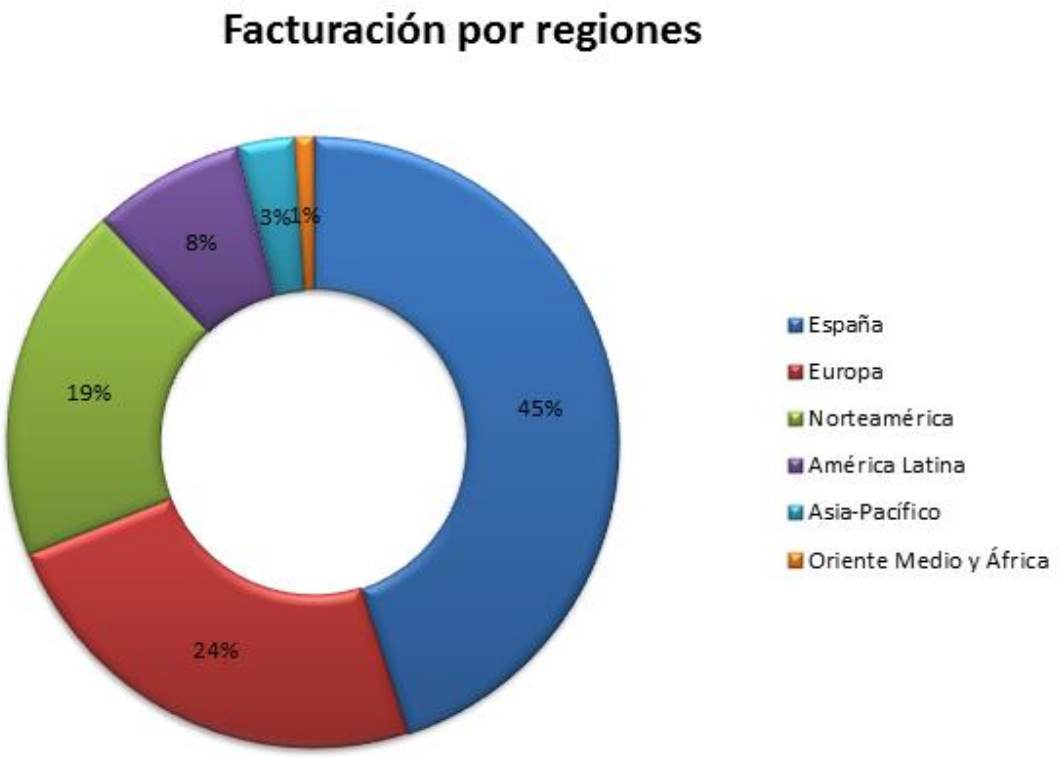


Figura 8. Distribución de la facturación por regiones

De todas las empresas, un 79% se dedica al desarrollo y creación de sus propios videojuegos, y un 55% de las empresas desarrolla videojuegos para terceras empresas. Otras actividades que realizan las empresas son actividades de publishers, creación de herramientas de desarrollo, formación, marketing, etc.

En 2013, la industria española del videojuego facturó 313,7 millones de euros. El 78% de esta facturación procede de la distribución online, mientras que sólo el 22% tiene procede de videojuegos en soporte físico. Dentro de la facturación online, predomina el modelo de pago por descarga con un 36%, seguido del *free to play* con un 21%, y los modelos de subscripción sólo representan el 2%.

Como se muestra en la *Figura 8*, la mayor parte de la facturación proviene del extranjero, con un 55% del volumen del negocio.

3. Booster

3.1. Introducción

En este capítulo se explicará la arquitectura utilizada, en más detalle que las descripciones de motores del apartado "2.3. Motor de juegos". A continuación se explicará el desarrollo del proyecto (análisis, diseño e implementación, planificación temporal y herramientas utilizadas).

3.2. XNA Framework y MonoGame

Como ya se ha explicado en la introducción, de entre varias alternativas para facilitar el desarrollo de un videojuego y no tener que aprender a usar librerías de bajo nivel como OpenGL y DirectX, se ha escogido MonoGame como framework para el desarrollo de videojuegos.

3.2.1. XNA Framework

XNA Framework está basado en la implementación nativa de .NET Compact Framework 2.0 para desarrollar para Xbox 360 y .NET Framework 2.0 para Windows. Incluye un conjunto extenso de librerías, específicas para el desarrollo de videojuegos, que promueven la reutilización de código entre las plataformas objetivo. Este framework se ejecuta sobre una versión de Common Language Runtime que está optimizada para los videojuegos para proporcionar un entorno de ejecución gestionado. Con XNA se pueden desarrollar videojuegos para Windows XP, Windows Vista, Windows 7, Windows Phone y Xbox 360. Como los videojuegos desarrollados con XNA se escriben para Common Language Runtime, se pueden ejecutar en cualquier plataforma que soporte XNA con ninguna o un mínimo de modificación.

XNA Framework encapsula detalles tecnológicos de bajo nivel envueltos en la codificación de un videojuego, asegurándose de que el framework se encargue de diferenciar entre plataformas cuando los videojuegos se portan entre plataformas compatibles, permitiendo a los desarrolladores centrarse más en el contenido y la experiencia de juego [48].

Aunque XNA Framework no ofrece todas las facilidades que pueden ofrecer Unity u otros motores de pago, con él se han desarrollado varios videojuegos de gran calidad, como *Bastion*, *Dust: An Elysian Tail*, *Fez*, *Magicka* y *Skulls of the Shogun*.

3.2.2. MonoGame

MonoGame es una implementación de código abierto de XNA 4.0 Framework. Se puede utilizar con todos los compiladores Mono y con Visual Studio [49].

El objetivo de MonoGame es permitir a los desarrolladores de videojuegos para Windows, Xbox 360 y Windows Phone 7 con XNA portar sus videojuegos a otras plataformas con un esfuerzo mínimo, y proporcionar un gran framework gestionado para los desarrolladores en las plataformas objetivo. MonoGame permite desarrollar para las siguientes plataformas:

- iOS
- Android
- Windows
- Mac OS X
- Linux
- Windows Store Apps
- Windows Phone 8
- PlayStation Mobile
- OUYA

Después de que Microsoft anunciara que iba a dejar de dar soporte a XNA, y que no habrá más versiones después de XNA 4.0, MonoGame dijo que continuaría dando soporte a los desarrolladores que usaban XNA. Con MonoGame, los desarrolladores que usan XNA pueden continuar usando las mismas herramientas que tenían, y además, pueden publicar videojuegos para Windows 8 (XNA no daba soporte) [50].

Con MonoGame se han portado videojuegos desarrollados con XNA Framework, como *Bastion*, *Fez* y *Skulls of the Shogun*, a otras plataformas.

3.2.3. La clase Game y el bucle del juego

Cuando creamos un proyecto de XNA Framework o de MonoGame, se generan 2 clases Program.cs y Game1.cs. También se genera un proyecto de contenido en XNA, o una carpeta llamada Content en MonoGame, que contendrán los activos de arte, sonido, etc.

La clase Program.cs sólo tiene una función, crear un objeto Game1 y ejecutarlo. En esta clase se encuentra una de las pocas diferencias entre XNA y MonoGame. La diferencia es que en la instrucción del precompilador, XNA comprueba con Windows o Xbox 360, y MonoGame con Linux en lugar de Xbox 360, al menos en el caso de un proyecto para Windows o Linux. Además, en MonoGame, antes del método Main se utiliza el atributo STAThread, que indica que la aplicación va a utilizar un único hilo.

La clase Game1 hereda de Game, y tiene por defecto 5 métodos para sobrescribir con el comportamiento específico de nuestro videojuego. Además, tiene 2 variables, la primera de ellas es GraphicsDeviceManager, que tiene una propiedad GraphicsDevice que permite acceder a la GPU desde el videojuego. Todas las operaciones de pintar por pantalla se

ejecutan a través de este objeto. La otra variable es SpriteBatch, que es el objeto clave para mostrar sprites por pantalla, y se crea haciendo uso de GraphicsDevice [51].

- **Initialize** se utiliza para inicializar variables y otros objetos asociados con el objeto Game1.
- **LoadContent** se llama justo después de Initialize, y cada vez que sea necesario volver a cargar el contenido gráfico. En este método es donde se cargarán todo el contenido que requiera el videojuego, como imágenes, modelos, sonidos, etc. Después de ejecutarse este método, el videojuego entrará en el bucle del juego.
- **UnloadContent** se llama cuando el videojuego sale del bucle del juego. En este método se descarga el contenido cargado en LoadContent. Normalmente, XNA gestionará toda la recolección de basura, pero si se ha modificado memoria en algún objeto que requiere una gestión especial, este método será el encargado de hacerlo.
- **Update** define la lógica del videojuego. En este método se comprueban los cambios ocurridos y se actúa en función de ellos.
- **Draw** se encarga de mostrar por pantalla los elementos del videojuego haciendo uso de GraphicsDevice y SpriteBatch.

Una de las diferencias principales entre el desarrollo de videojuegos y el desarrollo de aplicaciones típicas es el concepto de *polling* en lugar de registrar eventos. Muchas aplicaciones que no son videojuegos están desarrolladas para ser dirigidas por eventos. Esto quiere decir que la aplicación está inactiva hasta que el usuario no interactúe con ella. Cuando el usuario pulse un botón, se lanzará un evento, entonces la aplicación se activará y realizará una acción asociada al evento del pulsar el botón.

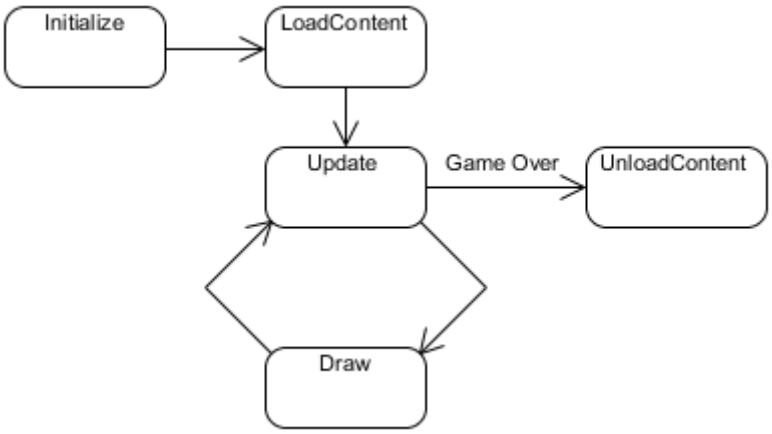


Figura 9. Diagrama del bucle del juego en XNA Framework

El desarrollo de videojuegos está dirigido por polling o búsqueda de eventos, en lugar de esperar a que ocurra un evento. En lugar de que el sistema le diga al videojuego de que se ha pulsado un botón, es el videojuego el que le pregunta al sistema si el botón se ha pulsado. De esta forma el videojuego siempre está realizando acciones, sin tener en cuenta la entrada del usuario. En un videojuego, el personaje se mueve cuando se pulsan determinados botones, pero si no se usara polling, mientras el usuario no proporcione una entrada, los enemigos no se moverán, ni se actualizarán las animaciones, es como si el videojuego se congelara. Para que el videojuego esté actualizándose y mostrándose por pantalla continuamente sin importar la entrada del usuario, se utiliza un bucle infinito, que en cada vuelta comprueba el estado del videojuego, lo actualiza y los muestra por pantalla. Este bucle se llama bucle del juego, y termina cuando se cumple una condición de fin de juego, representado en la *Figura 9* [51].

3.2.4. Características

Aunque la implementación puede variar, XNA Framework y MonoGame funcionan exactamente igual, tienen exactamente las mismas clases y los mismos métodos, ya que MonoGame pretende portar los videojuegos desarrollados con XNA a plataformas que no soportan XNA, si es posible, sin tener que hacer ningún cambio en el código del videojuego desarrollado con XNA. Debido a esto, explicar XNA Framework y algunas diferencias con respecto a MonoGame, explicaría los 2 frameworks.

Debido a que este Trabajo de Fin de Grado es sobre desarrollar un videojuego, y hace uso de XNA Framework y MonoGame, se ha considerado que en lugar de explicar detalladamente todo lo que ofrece XNA Framework y MonoGame, se van a explicar las características que se han utilizado para desarrollar un videojuego en 2 dimensiones.

3.2.4.1. Sistema de entrada

Para consultar la entrada de usuario de teclado y gamepad, XNA Framework proporciona las estructuras KeyboardState y GamePadState.

Para obtener el estado actual del teclado o del gamepad se utilizan las clases Keyboard y GamePad, con el método estático común para ambas GetState. Este método guarda el estado actual, que es el que se guardará en KeyboardState o GamePadState.

En el caso del teclado, se pueden consultar las teclas pulsadas con GetPressedKeys, pero para consultar una tecla específica se utiliza IsKeyDown. Y para consultar si una tecla no está pulsada, se utiliza IsKeyUp.

En el caso del gamepad, los botones están diferenciados. La propiedad DPad devuelve los botones de control de dirección que están pulsados. La propiedad Buttons devuelve los botones de acción que están pulsados. La propiedad Triggers devuelve el valor analógico de los gatillos traseros del gamepad. La propiedad ThumbSticks devuelve los valores analógicos de los sticks analógicos. Y al igual que con el teclado, se puede comprobar el estado de un botón específico con IsButtonDown, para comprobar si está pulsado, y con IsButtonUp, para comprobar si no está pulsado. Además, se puede comprobar si el mando está conectado con la propiedad IsConnected [51].

3.2.4.2. Mostrar elementos por pantalla

Hay 2 tipos de elementos que se pueden mostrar por pantalla: imágenes y texto. En ambos casos es necesario hacer uso de SpriteBatch. SpriteBatch tiene los métodos Begin, con varias sobrecargas para distintas configuraciones, y End, y para mostrar elementos por pantalla, hay que realizar las llamadas a los métodos correspondientes entre estos 2 métodos.

Para mostrar elementos por pantalla se utiliza el sistema de coordenadas de la *Figura 10*, correspondiendo la coordenada (0, 0) al píxel de la esquina superior izquierda de la ventana.

Para mostrar imágenes se utiliza el método Draw, y para mostrar texto se utiliza el método DrawString, y ambos métodos tienen varias sobrecargas.

Los métodos de mostrar imágenes tienen 2 variantes: definiendo la posición o el área de destino. Y requieren, al menos, una imagen y un color que actúa como tinte, y si es blanco no modifica la imagen. Al método también se le puede especificar un área, de forma que sólo se muestre esa área de la imagen, y no la imagen entera. Además, se puede especificar rotación, origen de rotación, escala, profundidad y un efecto de volteo de la imagen.

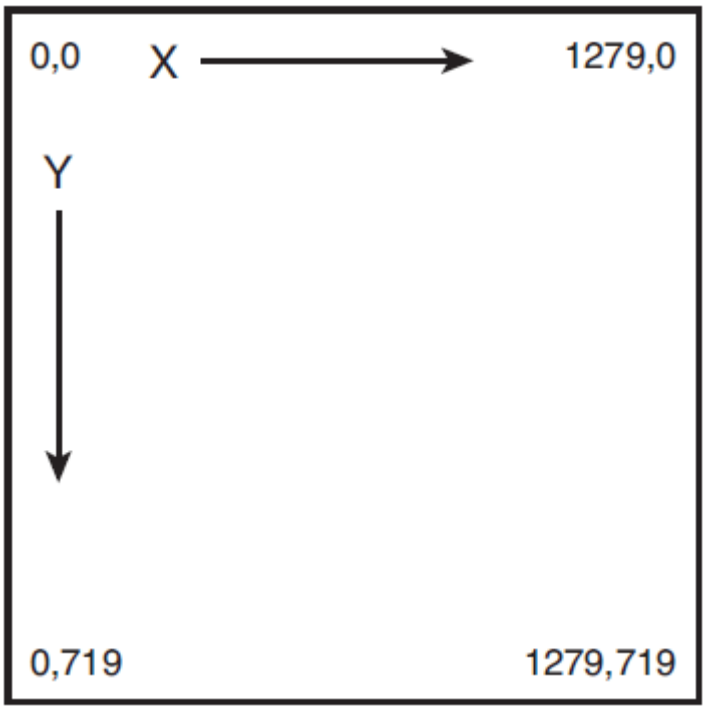


Figura 10. Sistema de coordenadas de XNA Framework y MonoGame [52]

En el caso de texto, se pueden especificar casi los mismos parámetros, excepto que en lugar de una imagen usamos texto y una fuente, y en lugar de especificar áreas, se utiliza sólo la posición.

Mientras que los demás parámetros se entienden fácilmente sólo con el nombre, el parámetro que indica la profundidad puede que no se entienda, ya que se está hablando de imágenes en un sistema de coordenadas de 2 dimensiones. Con la profundidad se puede definir qué elementos se muestran delante de otros, simulando que existe un eje Z aunque realmente tengamos un sistema de coordenadas de 2 dimensiones. De esta forma podemos controlar que, por ejemplo, el personaje se muestre delante de un elemento del fondo.

Para que los elementos se muestren en el orden indicado, es necesario utilizar una sobrecarga del método Begin de SpriteBatch. Uno de los parámetros que se añaden es SpriteSortMode. Este parámetro define el orden en el que se muestran los elementos. Los posibles valores de la enumeración son:

- Deferred: los sprites no se pintan hasta que se llama al método End, entonces se mandan todas las llamadas a la GPU en el orden en el que se hicieron.
- Immediate: las llamadas a Draw se mandan a la GPU conforme se realizan. Es el modo más rápido.
- Texture: funciona como Deferred, pero los sprites se ordenan por imágenes (una imagen puede contener muchos sprites).
- BackToFront: funciona como Deferred, pero los sprites están ordenados de adelante hacia atrás basado en el parámetro de la profundidad.
- FrontToBack: funciona como Deferred, pero los sprites están ordenados de atrás hacia adelante basado en el parámetro de la profundidad.

3.2.4.3. Reproducir sonidos

XNA Framework y MonoGame facilitan la tarea de añadir sonidos y música al videojuego. Para los sonidos, el framework proporciona la clase SoundEffect. La clase SoundEffect sólo

permite reproducir el sonido, pudiendo ajustar el volumen, el ajuste de paso y la panorámica. Este tipo de objetos no permite repetición. Para tener más control sobre el sonido existe la clase SoundEffectInstance. Transformando un SoundEffect en un SoundEffectInstance se puede poner el sonido en repetición, pararlo, pausarlo, resumirlo y consultar el estado actual del sonido (reproduciéndose, pausado o parado).

XNA Framework tiene una herramienta que MonoGame no ha implementado, XACT. Microsoft Cross-Platform Audio Creations Tool (XACT) es una poderosa herramienta de audio-autoría gráfica que permite gestionar una gran cantidad de archivos de audio y controlar su reproducción en el videojuego. Usando la interfaz gráfica de XACT, se puede cargar archivos de audio existente, agruparlos, y añadirles efectos. La API proporcionada por XNA Game Studio puede desencadenar su reproducción. La razón por la que MonoGame no soporta XACT es debido a que Microsoft ha dejado de darle soporte, y en Windows 8 no funciona [52].

3.2.4.4. Content Pipeline

La versión de MonoGame utilizada para desarrollar el videojuego es la 3.0.1, que no tiene el Content Pipeline de XNA Framework totalmente funcional, por lo que no se ha hecho uso de él.

Normalmente, cuando se habla de compilar, se piensa sólo en el código, pero el contenido también debería compilarse. Cuando un archivo se añade al proyecto de contenido de XNA, se compilará la próxima vez que se compile el proyecto. La compilación es un proceso de 2 etapas: importación y procesamiento. Después de que el archivo de contenido ha sido procesado, se crea un archivo con la extensión .xnb (XNA Binary), que es el que leerá XNA cuando se cargue el contenido en memoria.

Un videojuego no necesita procesar los archivos de contenido para poder jugar a él, y aunque XNA obliga a utilizar los archivos .xnb, en MonoGame no son totalmente necesarios. Como el Content Pipeline de XNA Framework no es totalmente funcional en MonoGame 3.0.1, para crear los archivos .xnb es necesario compilarlos, ya sea con XNA o con una herramienta alternativa, pero MonoGame acepta el contenido con extensiones .jpg, .png, .wav, etc. Aun así, es recomendable utilizar los archivos .xnb, ya que ofrecen ventajas. Sin el Content Pipeline, la importación y procesado del contenido se realiza en tiempo de ejecución. Utilizando el ejemplo de la bibliografía [52], si son necesarios 5ms en leer un archivo de disco y 50ms en importarlo y procesarlo, con 50 archivos serían necesarios 2750ms para cargar los datos. Si se supone que se cargan cada vez que se carga un nivel, el jugador tendría que esperar casi 3 segundos cada vez que carga un nivel. Sin embargo, con el Content Pipeline, la importación y procesamiento se realizan cuando se compila el archivo, por lo que tardaría 2500ms una sola vez, y que no afectaría al jugador, por lo que el jugador sólo tendría que esperar 250ms cada vez que se carga un nivel.

3.3. Análisis

Inicialmente, la idea del videojuego a desarrollar era muy ambiciosa, pero debido a tener un tiempo limitado, la idea se acotó en algunas características básicas, a las que se podrían añadir nuevas características, siempre que se dispusiera de tiempo suficiente después de terminar la versión con las características básicas. El diagrama de casos de uso de la *Figura 11* representa los casos de uso que corresponden a los objetivos mencionados en el anteproyecto y en la introducción.

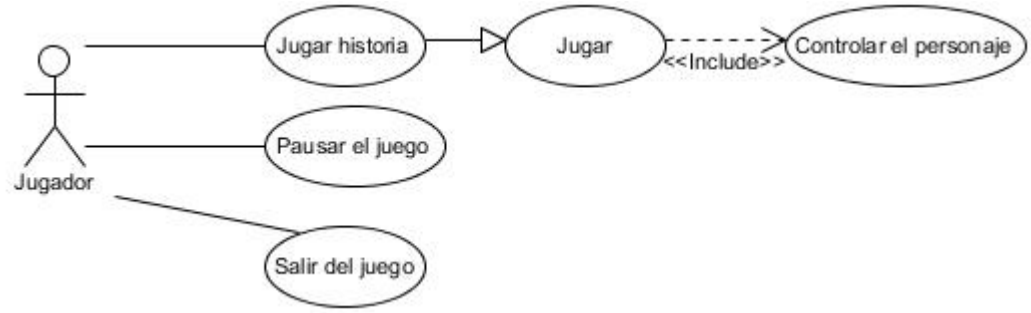


Figura 11. Diagrama de casos de uso inicial

Una vez que el software cumplía los objetivos propuestos, y por tanto, los casos de uso de la *Figura 11*, se amplió para que el videojuego resultante fuera más completo. En la *Figura 12* se muestra el diagrama de casos de uso de la versión final del software.

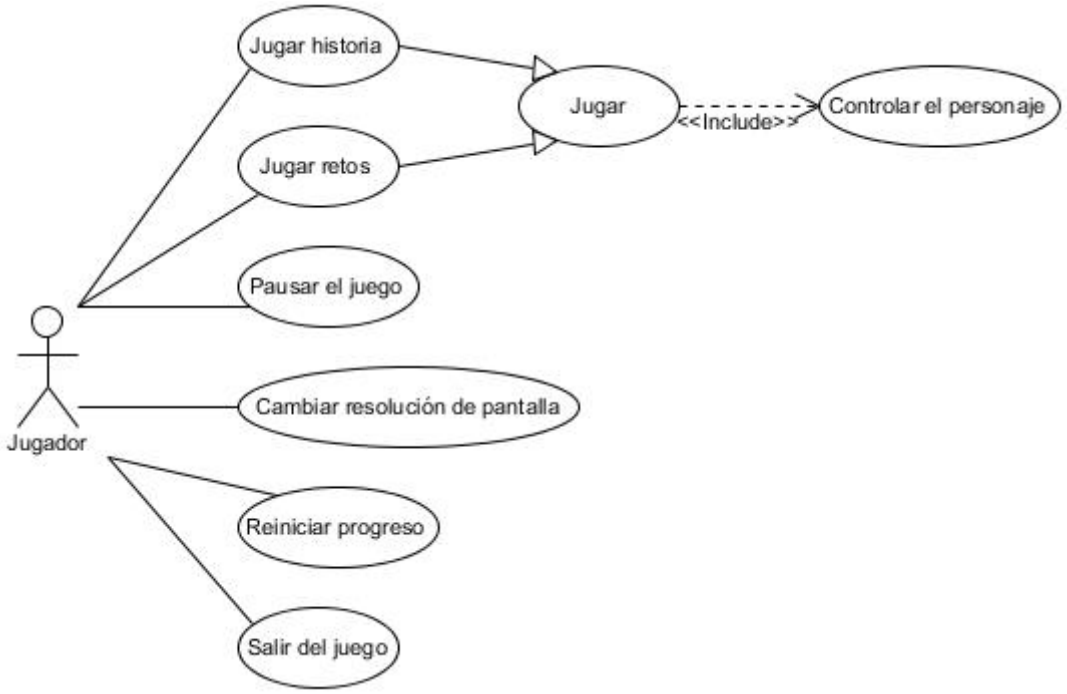


Figura 12. Diagrama de casos de uso final

Sin embargo, estos diagramas apenas pueden explicar en qué consiste el videojuego, por lo que se va a explicar con más detalle a continuación.

Para explicar las características finales del videojuego, se va a dividir en 3 partes: los niveles, los menús, y los estados de transición.

3.3.1. Niveles

En la versión inicial, se decidió que dentro de un nivel habría bloques, plataformas, objetos que dieran puntos, objetos que hicieran daño al personaje y una salida. El jugador podría controlar el personaje con teclado o con un gamepad. Y por último, el jugador podría pausar el videojuego. Las mecánicas iniciales del videojuego son:

- El personaje puede moverse hacia los lados.
- El personaje puede saltar.
- Si el personaje colisiona con un objeto que da puntos, la puntuación conseguida aumenta.

- Si el personaje colisiona con un objeto que hace daño, la vida del personaje disminuye.
- Si la vida del personaje llega a 0, el jugador pierde y termina el nivel.
- Si el jugador consigue llegar a la salida, el jugador completa el nivel.

Una vez que todas estas mecánicas funcionan, se han añadido nuevas mecánicas, que permitirán diseñar niveles que supongan un mayor reto al jugador. Las nuevas mecánicas son:

- El personaje puede utilizar la habilidad especial "Boost" que le permite moverse más rápido, e ignorando la gravedad, durante un tiempo.
- Para no poder abusar de la habilidad especial "Boost", esta tiene un tiempo de espera entre usos.
- El personaje puede coger llaves colisionando con ellas.
- Si el personaje colisiona con una puerta mientras posee alguna llave, el jugador pierde una llave y la puerta se abre.

3.3.2. Menús

El videojuego dispondrá de varios menús. La versión inicial estaba muy enfocada a crear niveles jugables, por lo que había pocos menús, y con pocas opciones. Los menús de la versión inicial son:

- Un menú principal, desde el que se podría acceder al menú de los niveles, y con una opción para cerrar el videojuego.
- Un menú para los niveles, que tendría los niveles jugables y una opción para volver al menú principal. Aparecerían todos los niveles, pero inicialmente sólo estaría disponible el primero, y al completarse un nivel, estaría disponible el siguiente nivel.
- Un menú de pausa, que apareciera al pausar el videojuego durante el nivel, con las opciones de continuar y volver al menú principal.

En la versión final se ha añadido:

- Un menú para retos, y el menú para los niveles de la versión inicial ahora es el modo historia, para diferenciarlos. A diferencia del modo historia, en el menú de retos, todos los niveles están disponibles desde el principio, y no hace falta desbloquearlos.
- Al menú principal se ha añadido la opción de acceder al menú para retos. También se han añadido una opción para poner el videojuego en pantalla completa, y otra para reiniciar el progreso del jugador.

3.3.3. Estados de transición

Los estados de transición no se tuvieron en cuenta en la versión inicial, pero se han añadido para dar un aspecto más profesional al producto final. Estos estados son: la pantalla de inicio, una pantalla de carga que se muestra antes de abrirse un nivel, una pantalla de game over, y una pantalla que muestra la máxima puntuación conseguida en un nivel tras completarlo.

3.4. Diseño e implementación

Para el diseño se ha utilizado una metodología orientada a objetos con patrones de diseño. Para facilitar la comprensión del diseño, la explicación se va a dividir en varias partes en lugar de explicar todo el diseño con un solo diagrama. Para explicar el diseño de cada parte, se va a

hacer uso de diagramas de clases, y diagramas de secuencias y de estados cuando sean necesarios. Si en alguna parte se han utilizado patrones de diseño, también se van a explicar.

3.4.1. Gestor de estados

Al crear un proyecto de XNA Framework o de MonoGame, se crean automáticamente 2 clases: la clase Program, que tiene el método Main que ejecuta el videojuego, y la clase Game1, que se ha renombrado como Booster, y que hereda de la clase Game, y proporciona una plantilla de los métodos que hay que sobrescribir con la lógica del videojuego: Initialize, LoadContent, Update, Draw y UnloadContent. Un videojuego no tiene un solo estado o escena, sino que tiene varios: niveles, menús, pantallas de carga, etc. Para evitar poner la lógica de todos los estados en la clase Booster, se va a hacer uso de un gestor de estados, que va a ser el encargado de cambiar el estado actual, actualizarlo y mostrarlo por pantalla.

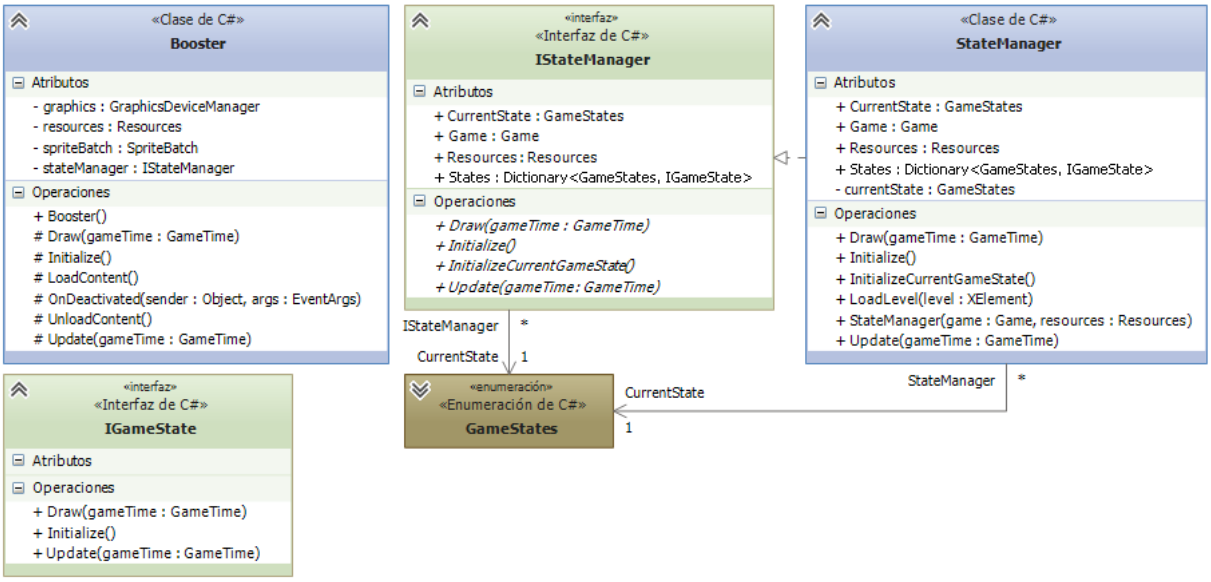


Figura 13. Diagrama de clases del gestor de estados

En la *Figura 13* se puede ver el diagrama de clases del gestor de estados. La clase Booster tiene un objeto del tipo IStateManager, al que va a delegar la responsabilidad de inicializar, actualizar y mostrar por pantalla los estados del videojuego.

Las clases que implementen la interfaz IStateManager van a tener una propiedad de tipo Game, necesaria para acceder a atributos relacionados con la pantalla del videojuego; una propiedad de tipo Resources, que almacena todos los recursos necesarios para el videojuego; y las más importantes, una propiedad States, que es una estructura de datos que almacena todos los estados del videojuego, y una propiedad CurrentState para poder conocer el estado actual. También tendrán que implementar 4 métodos: el método Initialize, para poder reiniciar los valores del objeto sin tener que eliminarlo y volver a crear uno nuevo; los métodos Update y Draw, para actualizar y mostrar el estado actual; y el método InitializeCurrentGameState, para inicializar el estado actual. La estructura de datos escogida para almacenar los estados es un Dictionary, una tabla hash de C#, para asociar a un tipo de estado, definido en la enumeración GameStates, un estado IGameState. De esta forma, el acceso a un estado concreto es más rápido que si se usara la estructura de datos List. StateManager es el gestor de estados principal del videojuego, e implementa la interfaz IStateManager. En esta clase, en el método Initialize creamos todos los estados del videojuego y los añadimos a States. Y por último, cuando CurrentState cambia, se inicializa el nuevo estado llamando a la función InitializeCurrentState.

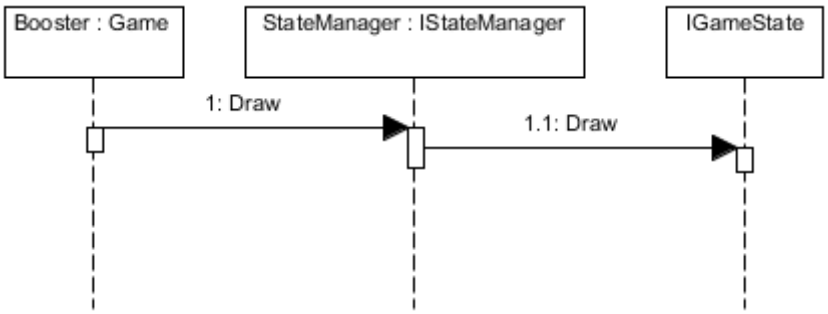


Figura 14. Diagrama de secuencias de la actualización del videojuego

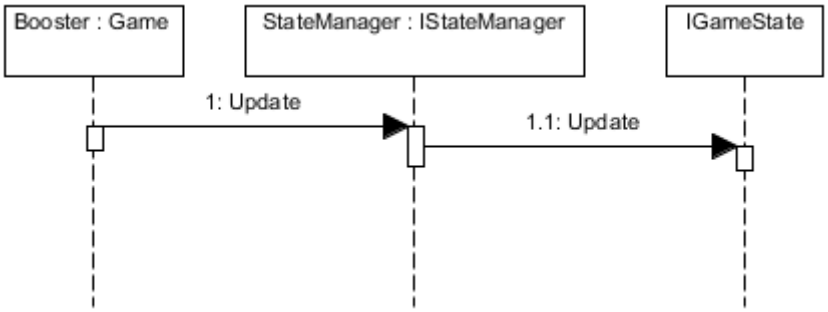


Figura 15. Diagrama de secuencias del mostrado por pantalla del videojuego

La *Figura 14* y la *Figura 15* representan la interacción de estas clases en la actualización y el mostrado por pantalla del videojuego. Los siguientes pasos de la secuencia dependen de la clase que implemente IGameState.

En el gestor de estados se han usado el patrón de diseño State. El patrón State permite a un objeto alterar su comportamiento cuando cambia su estado interno [53, 54]. La correspondencia del diagrama de la *Figura* 16 con el diseño del videojuego es: Context corresponde a StateManager; IState corresponde a IGameState; y la principal diferencia entre los dos diseños es que Context tiene un objeto IState, mientras que StateManager tiene una colección de IGameState, y la clave para acceder al IGameState. El estado del videojuego es CurrentState de StateManager, y se actualiza y muestra por pantalla el estado, que implementa IGameState, asociado a CurrentState. Si cambia CurrentState, el comportamiento cambia, ya que ahora CurrentState está asociado a un estado distinto.

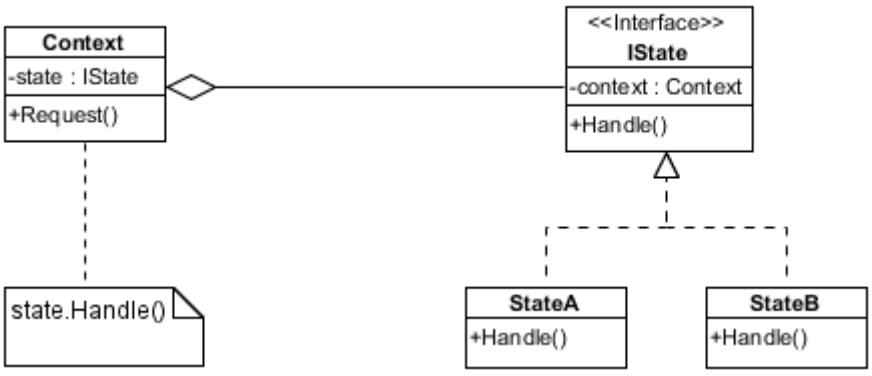


Figura 16. Estructura del patrón State

3.4.2. Gestión de entrada

En XNA Framework y en MonoGame, para saber el estado del teclado, se hace uso de la clase KeyboardState, y para saber el estado del gamepad, se hace uso de la clase GamePadState.

Esto implica que, para que se produzca una acción si se pulsa una determinada tecla del teclado, o un determinado botón del gamepad, hay que comprobar 2 condiciones, Y si se añade otro método de entrada, habría que comprobar otra condición. Además, a veces es necesario comprobar que la tecla o botón no estuvieran pulsados previamente, ya que, por ejemplo en los menús, si no se hace esta comprobación, la aplicación detectaría que la tecla o botón están pulsados y pulsaría automáticamente en la primera opción de los menús siguientes. Esta comprobación del estado previo duplicaría el número de condiciones a comprobar, y en este caso, sería necesario comprobar 4 condiciones.

Pensando detenidamente, implementar una estructura condicional con 4 condiciones no es un problema tan grave. Si se supone que la tecla Q representa la acción de aceptar, y la tecla P representa la opción de cancelar, sería una configuración bastante extraña, y para cambiarla, habría que modificar todos los estados en los que estas teclas estuvieran envueltas en alguna condición. Para evitar estos problemas se ha creado un sistema de entrada.

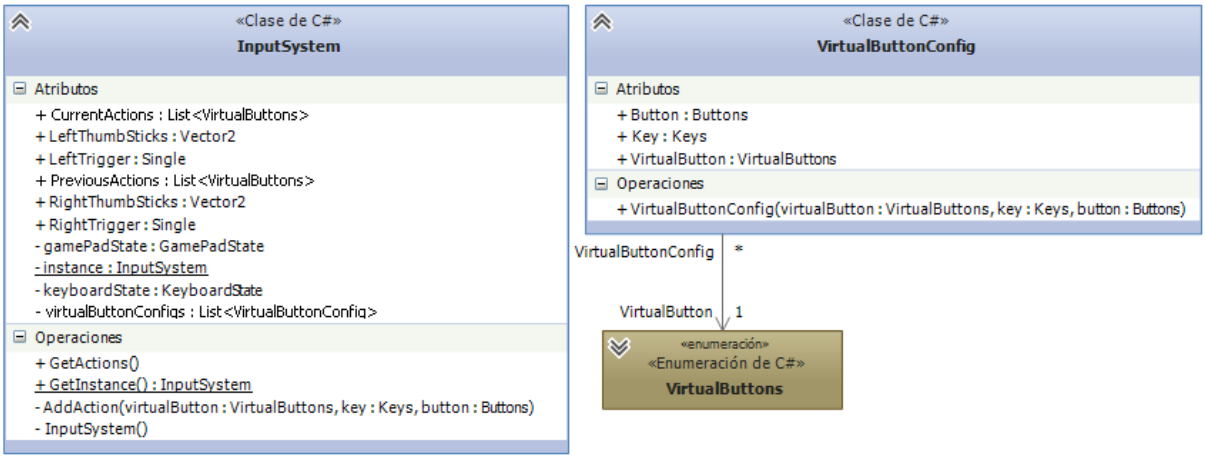


Figura 17. Diagrama de clases del gestor de entrada

Para solucionar el problema de utilizar 1 ó 2 condiciones para cada dispositivo de entrada, se han creado botones virtuales. Como se puede ver en la *Figura 17*, un objeto de la clase VirtualButtonConfig representa un botón virtual, y asocia un identificador, que es un elemento de la enumeración VirtualButtons, con una tecla y un botón del gamepad. InputSystem carga en una lista todos los botones virtuales definidos en un archivo XML con la estructura mostrada en la *Figura 18*. De esta forma el usuario puede configurar las teclas y botones que prefiere utilizar para controlar el videojuego.

```
<Config>
  <Input virtual_button ="Start" key="Enter" button="Start"/>
</Config>
```

Figura 18. Estructura del archivo XML de configuración

Cuando se llama al método GetActions, se carga en PreviousActions el estado de los botones virtuales que estaban pulsados anteriormente, en CurrentActions el estado de los actuales, y en el resto de los atributos públicos, valores que proporcionan algunos botones del gamepad. De esta forma, si se ha asociado a una acción el botón virtual Start, sólo hay que comprobar una condición, en lugar de comprobar la tecla Enter y el botón Start.

Hay algo que diferencia a la clase InputSystem de todas las demás clases que forman el videojuego, es la única clase Singleton. El patrón Singleton, representado en la *Figura 19*, sirve para asegurar que una clase tendrá una instancia única [53, 54].

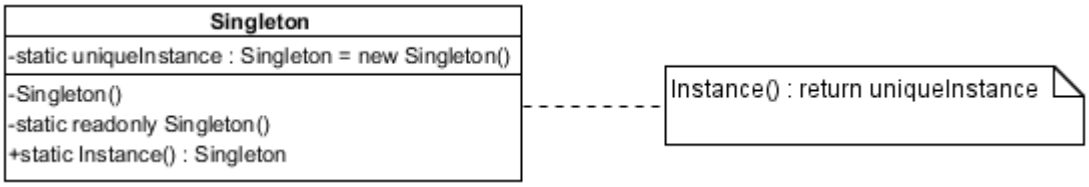


Figura 19. Estructura del patrón Singleton

Existen otras clases en el proyecto que podría ser necesario que sólo permitieran una instancia única, y se pasan por parámetro a varias clases y métodos para evitar volver a instanciarlas, pero si se instancian de nuevo por error, ocupan más memoria, pero no producirían ningún comportamiento erróneo. En cambio, si se vuelve a instanciar InputSystem, PreviousActions estaría vacío, y se produciría el mismo fallo que si no se comprobara, pudiendo entrarse automáticamente en la primera opción del menú al que se acaba de entrar. Por eso el acceso está restringido al método estático GetInstance, y el constructor es privado.

3.4.3. Almacén de recursos

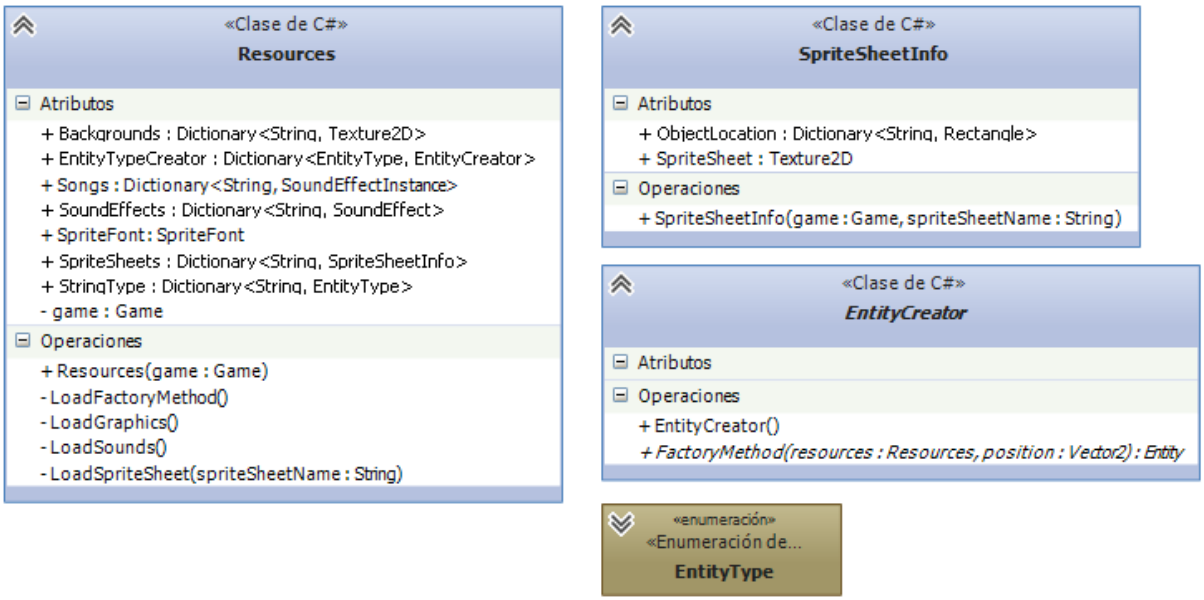


Figura 20. Diagrama de clases de la clase Recursos y las clases relacionadas

En el videojuego se han utilizado imágenes, efectos de sonido y canciones. Para evitar cargar algunos de estos recursos varias veces, se van a cargar una sola vez, y se van a almacenar en un objeto. De esta forma, cada vez que sea necesario utilizar algún recurso, simplemente se utiliza el recurso almacenado en la clase Resources.

Como se puede ver en el diagrama de la *Figura 20*, la clase Recursos almacena las hojas de sprites, los fondos, los efectos de sonido, las canciones, y la fuente que se utilizan en el videojuego. También se almacenan 2 mapas: uno que mapea cadenas de texto a valores de la enumeración EntityType, por razones de eficiencia con respecto a la conversión de cadenas de texto a valores de una enumeración; y otro que mapea a cada tipo de entidad del videojuego una factoría. Las factorías para crear las entidades se explicarán detalladamente en el apartado "3.4.10. Creación de entidades".

Para guardar las hojas de sprites se utiliza la clase SpriteSheetInfo. Al crear un objeto, se le pasa el nombre de un XML, y se lee un XML con la estructura mostrada en la *Figura 21*:

```
<TextureAtlas imagePath="tiles_spritesheet">
	<SubTexture name="box.png" x="0" y="864" width="70" height="70"/>
</TextureAtlas>
```

Figura 21. Estructura de un archivo XML con información sobre una hoja de sprites

A partir de la información del XML, se carga la imagen de la hoja de sprites, y posteriormente se carga un mapa con un identificador y la posición de un objeto dentro de la hoja de sprites. Así, en lugar de tener una imagen para cada elemento que se quiera mostrar por pantalla, cada hoja de sprites tiene muchos elementos, y buscando el elemento por su identificador, podemos mostrar por pantalla sólo el trozo de imagen que corresponde a ese elemento.

3.4.4. Cámara

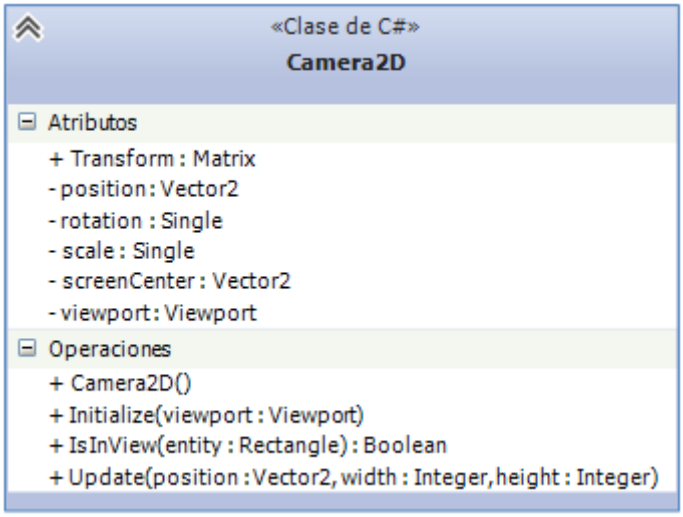


Figura 22. Diagrama de clases de la cámara

La pantalla del videojuego inicialmente es de 800 x 600 píxeles, y limita el contenido que se ve en la pantalla. Con este espacio tan limitado habría que crear niveles con elementos muy pequeños o con muy pocos elementos. Cuando queremos mostrar elementos por pantalla en XNA Framework y MonoGame, se puede pasar por parámetro una matriz al método Begin del SpriteBatch. De esta forma se consigue poder cambiar a que se muestre por pantalla los elementos contenidos en un rectángulo de 800 x 600 píxeles, pero con desplazamiento. La matriz que se usa para determinar qué se muestra por pantalla, es el resultado de varias multiplicaciones con matrices [55]. Como se muestra en la *Figura 22*, al método Update le pasamos una posición. Si por ejemplo, le pasamos la posición del personaje, la cámara intentará posicionarse de forma que el jugador esté en el centro de la pantalla. Con el método IsInView determinamos si un elemento se mostrará en pantalla, para posteriormente no realizar operaciones de pintado de elementos que no se mostrarán.

3.4.5. Transiciones

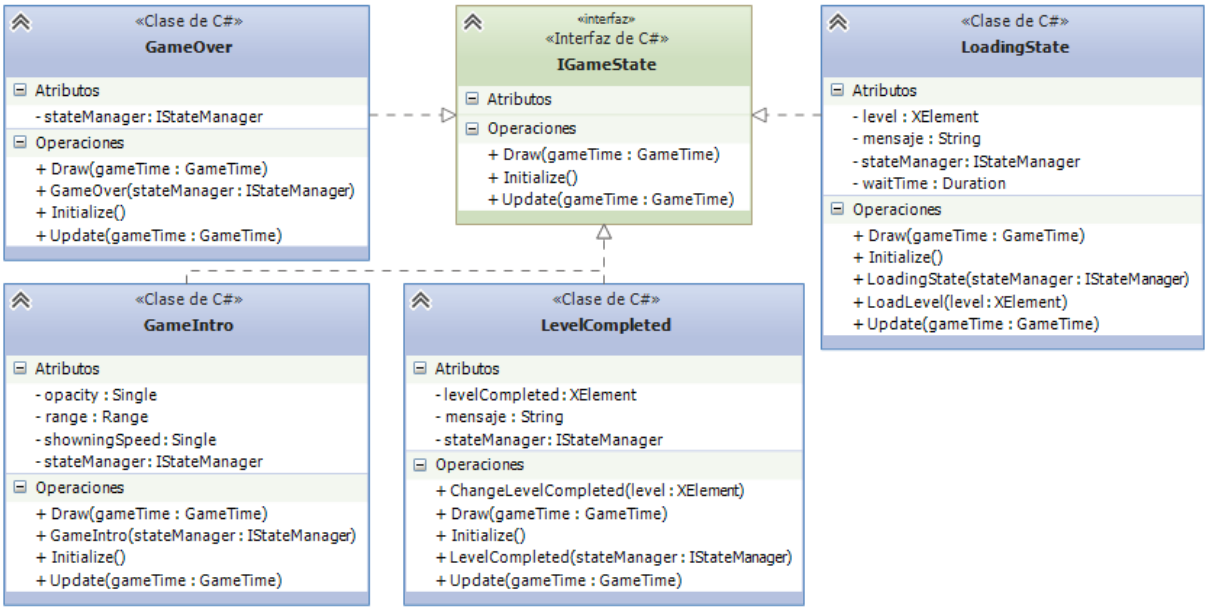


Figura 23. Diagrama de clases de estados de transición

Los estados de transición son la parte menos importante del videojuego. Objetivamente, lo único que debe tener obligatoriamente un videojuego es un estado que nos permita jugar, que en este caso sería un nivel. Si se considera que el videojuego debe ofrecer diferentes opciones, y que no sea simplemente jugar un nivel o una secuencia de niveles predefinida o autogenerada, y sin dejar que el jugador pueda elegir entre varias opciones, entonces es necesario que el videojuego tenga menús. Pero una pantalla de transición no permite hacer nada nuevo al jugador, tal vez se utiliza para mostrarle información, o como la típica primera pantalla de un videojuego, donde se muestra el título del videojuego y un texto en el que suele poner "Pulse Start", y de esta pantalla se pasa al primer menú del videojuego. De todas formas, es algo que siempre han tenido los videojuegos, y queda más claro que en una pantalla ponga "Game Over" en lugar de simplemente salir del nivel cuando aún no se ha completado.

Como se puede ver en el diagrama de la *Figura 23*, hay 4 estados de transición, que implementan IGameState. Aunque los estados son simples, todos tienen algo que los diferencia del resto.

- El estado **GameIntro** es el primer estado que se ve al ejecutar en el videojuego. En este estado simplemente sale el texto "Press Start" parpadeando, con el título del videojuego de fondo, y al pulsar Start o Enter, se pasaría al estado del menú principal. Al poner el videojuego en pantalla completa en el menú principal se cambia a este estado. La *Figura 24* cómo se llega a este estado, y cómo se cambia de estado a partir de este estado.

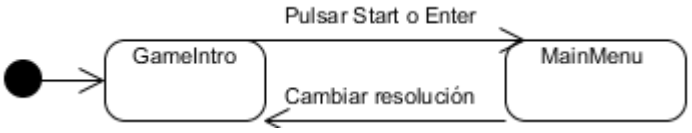


Figura 24. Diagrama de estados de GameIntro

- El estado **LoadingState** se muestra justo antes de empezar un nivel, mostrando el nombre del nivel. Este estado pasa al nivel la información que necesita para cargar el

su contenido, obtenida del elemento de XML, y transcurrido un tiempo, pasa automáticamente al estado del nivel. La información del nivel pasa a este estado a través del menú, o en algunos casos, a través del estado LevelCompleted. La *Figura 25* cómo se llega a este estado, y cómo se cambia de estado a partir de este estado.

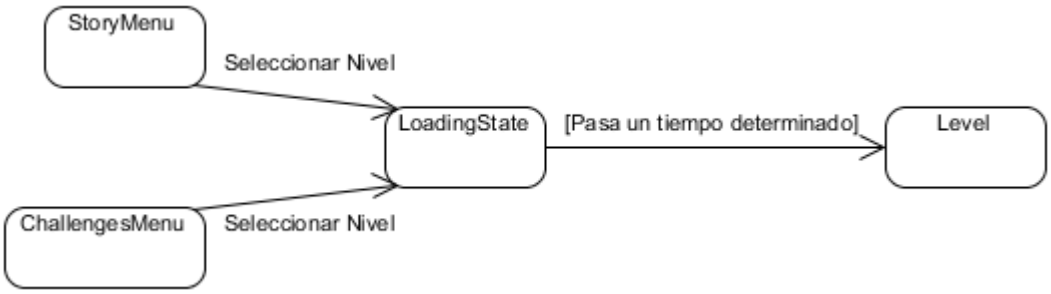


Figura 25. Diagrama de estados de LoadingState

- El estado **GameOver** simplemente muestra el fondo con el texto "Game Over". El videojuego entra en este estado cuando el personaje muere en un nivel. Al pulsar la tecla de aceptar se vuelve al menú principal. La *Figura 26* cómo se llega a este estado, y cómo se cambia de estado a partir de este estado.

Figura 26. Diagrama de estados de GameOver

- El último estado de transición del videojuego es el estado **LevelCompleted**. Al contrario que el estado GameOver, que se mostraba al no completar el nivel, este estado se muestra al completarlo. Este estado, al igual que el estado LoadingState, tiene una referencia a los datos del nivel guardados en un XML, pero los utiliza de una forma diferente. De lo explicado hasta ahora, cuando se consultaba un XML, se hacía con LINQ to XML, que permite navegar por los archivos XML. Pero en este estado, también se hace uso de LINQ, que permite consultar los datos obtenidos del XML como si de una tabla de una base de datos se tratase. De esta forma podemos mostrar la puntuación más alta obtenida en el nivel. Cuando se completa un nivel, en este estado se modifica el archivo XML para desbloquear el siguiente, en el caso de que no estuviera desbloqueado. La *Figura 27* cómo se llega a este estado, y cómo se cambia de estado a partir de este estado.

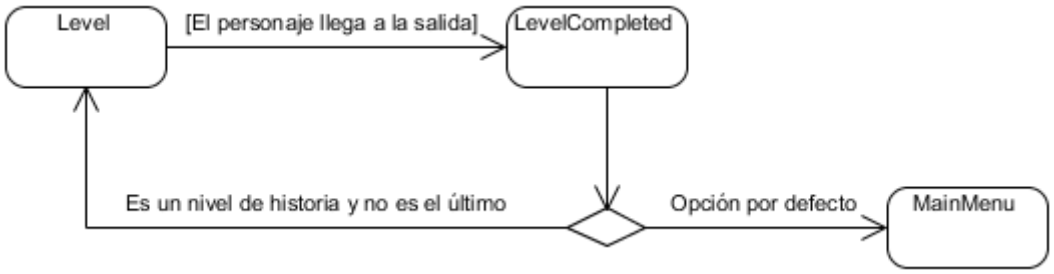


Figura 27. Diagrama de estados de LevelCompleted

3.4.6. Menús

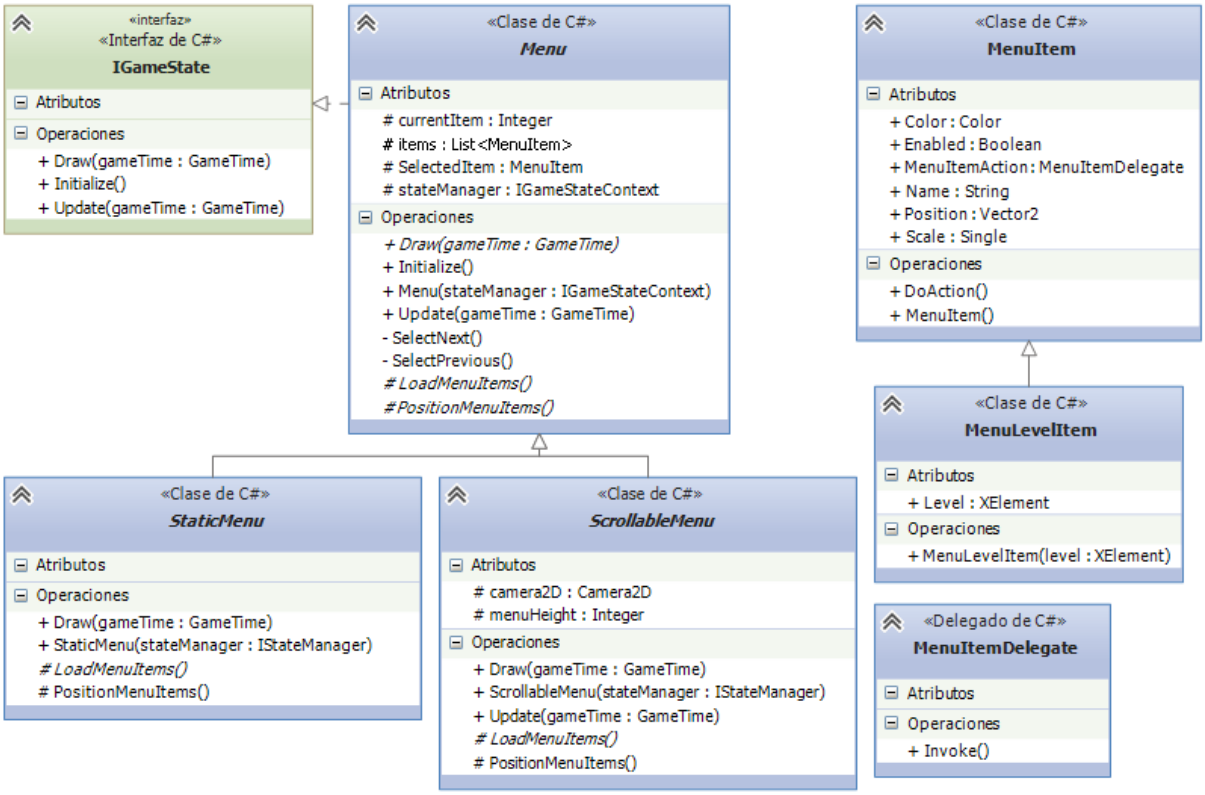


Figura 28. Diagrama de clases del estado Menu

Como se ha mencionado en el apartado "3.4.5. Transiciones", para dar al jugador la opción a elegir, el videojuego debe tener menús con opciones. Los menús del videojuego se pueden comparar con una lista de botones de una aplicación Windows Forms o WPF (*Windows Presentation Foundation*). Sin embargo, en este videojuego no se puede usar el ratón, así que se ha creado un menú en el que se puede navegar con teclado o gamepad.

El diagrama de la *Figura 28* representa el diseño de un menú. Como se ha podido deducir en el apartado "3.4.5. Transiciones", un menú es un estado del videojuego, e implementa IGameState. Un menú está compuesto de una lista de elementos, que son objetos de la clase MenuItem. Inicialmente estará seleccionado el primero, pero si se pulsa arriba o abajo, el método Update detectará la tecla o botón que se está pulsando y llamará al método SelectNext o SelectPrevious, de esta forma se podrá navegar por el menú. Si pulsamos el botón de aceptar, se ejecutará la acción asociada al elemento seleccionado.

Aunque cada elemento podría ser una simple cadena de texto, para poder personalizar cada elemento se ha creado la clase MenuItem. Con esta clase se puede cambiar la posición, el tamaño y el color con los que se va a mostrar cada elemento. De esta forma se puede hacer que el elemento seleccionado se vea más grande y con un color distinto a los otros elementos. Cabe destacar que la clase MenuItem es la única que hace uso de eventos, en este caso de los delegates de C#. Se podría crear una clase que heredara de MenuItem sólo para sobrescribir la acción de un método, pero casi todas las acciones asociadas a los elementos de los menús cambian el estado actual del videojuego, y por tanto, requieren acceso a StateManager. Usando delegates se evitar crear una clase para cada tipo de acción, y también se consigue que haya poco acoplamiento. De esta forma, cada menú declara la acción que se realizará con cada elemento. En los elementos que permiten acceder a un nivel (MenuLevelItem que hereda de MenuItem), también se almacena el nivel al que permiten acceder.

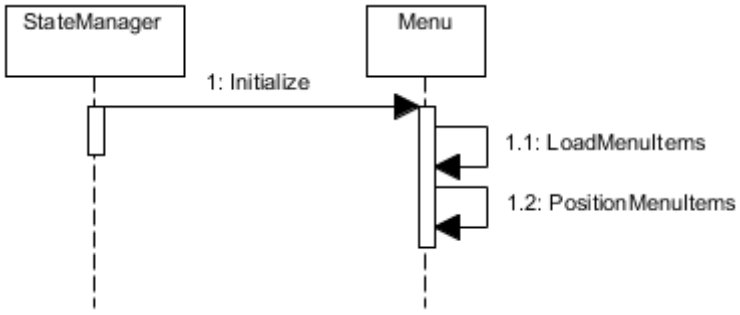


Figura 29. Diagrama de secuencias del método Initialize de Menu

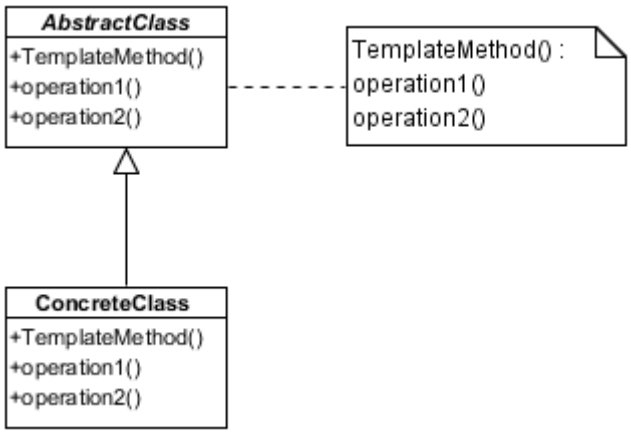


Figura 30. Estructura del patrón Template Method

Como se puede ver en la *Figura 28*, la clase Menu es una clase abstracta, que requiere que las clases que hereden sobrescriban los métodos LoadMenuItems y PositionMenuItems. Estos métodos se ejecutan cuando se ejecuta el método Initialize como muestra la *Figura 29*. Comparando la *Figura 28* y la *Figura 29* con la *Figura 30*, se ve claramente que se ha aplicado el patrón Template Method, se ha delegado a las subclases algunas etapas de un método [53, 54].

La clase Menu sirve como plantilla para crear todos los menús del videojuego, pero para crear varios tipos de menús reutilizando el código, se han creado 2 plantillas que heredan de Menú. Estas plantillas deben implementar los métodos Draw y PositionMenuItems, pero el método LoadMenuItems lo implementará cada menú, especificando los elementos y las acciones asociadas a ellos.

3.4.6.1. Menús estáticos

La primera plantilla que se ha creado es StaticMenu. Un menú que herede de StaticMenu, mostrará todos sus elementos siempre en la misma posición. Sobrescribiendo los métodos PositionMenuItems y Draw, un StaticMenu posiciona y muestra sus elementos siempre en la misma posición. Esta plantilla sirve para menús con pocos elementos, en el caso de tener un menú con muchos elementos, se debería usar un menú con desplazamiento, explicado en el siguiente apartado, “3.4.6.2. Menús con desplazamiento”.

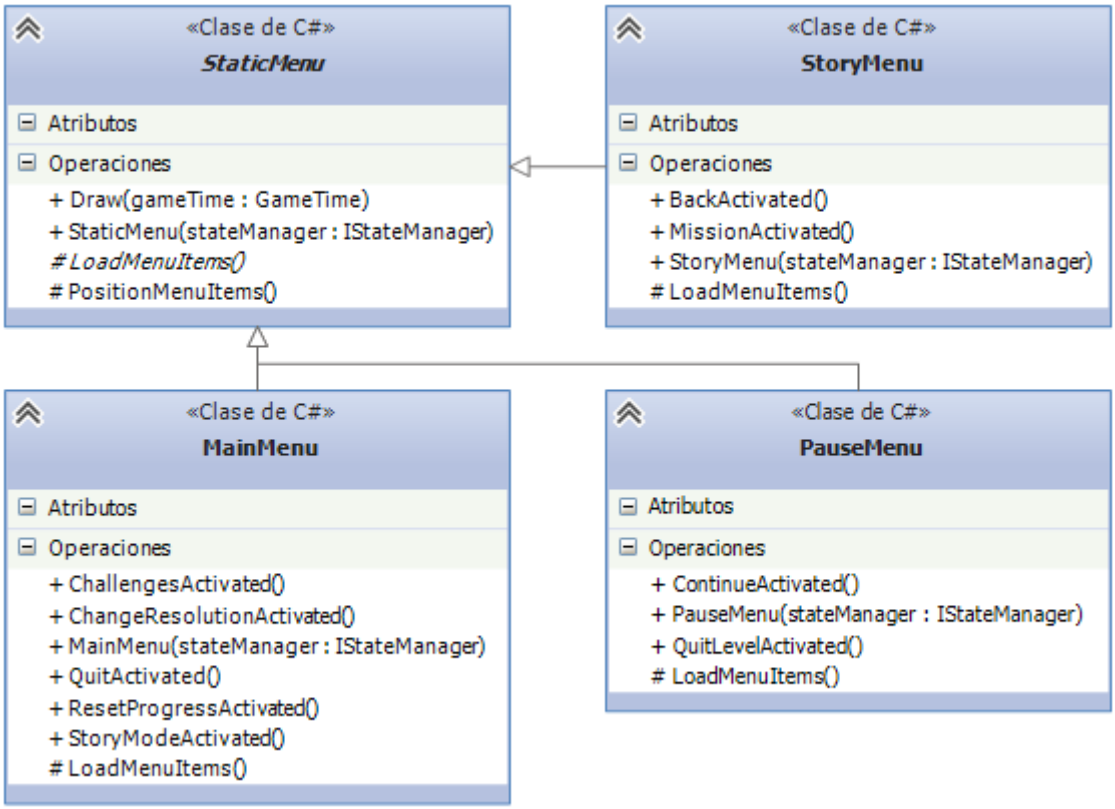


Figura 31. Diagrama de clases de StaticMenu

En la *Figura 31* se muestra la clase StaticMenu y todos los menús que se han hecho utilizando esta plantilla.

- **MainMenu** es el primer menú que nos encontramos en el videojuego. Como se muestra en la *Figura 32*, a este menú se accede desde GameIntro, desde LevelCompleted según la condición representada en la *Figura 27*, desde GameOver y desde PauseMenu; y desde él se puede acceder a los 2 menús de niveles, desde los cuales se puede volver a MainMenu. Al poner el videojuego en pantalla completa, este vuelve al estado GameIntro. Aunque no cambian el estado del videojuego, en MainMenu se puede reiniciar el progreso del jugador o cerrar el videojuego. Para reiniciar el progreso, es necesario modificar el archivo XML de la *Figura 33*, eliminando las puntuaciones y desactivando los niveles de historia excepto el primero.

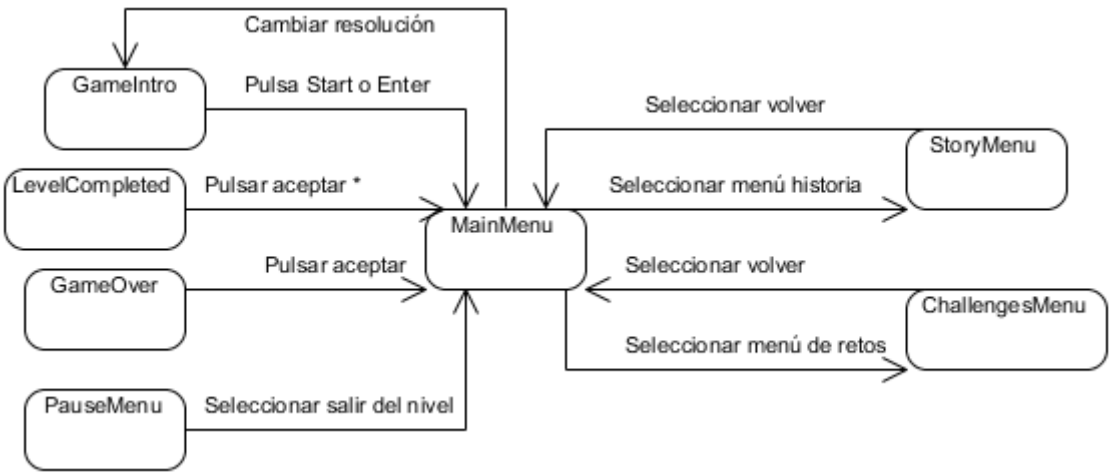


Figura 32. Diagrama de estados de MainMenu

- **StoryMenu** es un menú con varios niveles. Estos niveles se cargan de un archivo XML con la estructura mostrada en la *Figura 33*. En este menú se cargan los niveles de StoryLevels. Inicialmente sólo está disponible el primer nivel, pero los otros también aparecen en el menú, aunque no se pueden seleccionar. Cuando un nivel se completa, el siguiente nivel estará disponible. Aunque el videojuego realmente no tiene una historia, se han considerado estos niveles como niveles de la historia porque siguen una progresión. Como muestra la *Figura 34*, a este menú se accede desde MainMenu, y permite volver a él, y al seleccionar un nivel, el estado del videojuego cambia a LoadingState, que posteriormente redirigirá al nivel.

```
<Levels>
  <StoryLevels>
    <Level name="Mission 1" file="Level 1.txt" enabled="true"/>
    <Level name="Mission 2" file="Level 2.txt" enabled="false"/>
  </StoryLevels>
  <Challenges>
    <Level name="Challenge 1" file="Level 1.txt" enabled="true">
      <Score score="15" date="16-08-2014" />
    </Level>
  </Challenges>
</Levels>
```

Figura 33. Estructura del archivo XML que guarda información de los niveles

Figura 34. Diagrama de estados de StoryMenu

- **PauseMenu** es un menú que aparece cuando se pausa un nivel. Este menú sólo tiene 2 opciones: una para regresar al nivel, y otra para salir del nivel volviendo al menú principal, como se muestra en la *Figura 35*. Este menú también aparece mientras se está jugando al nivel y se deselecciona el videojuego.

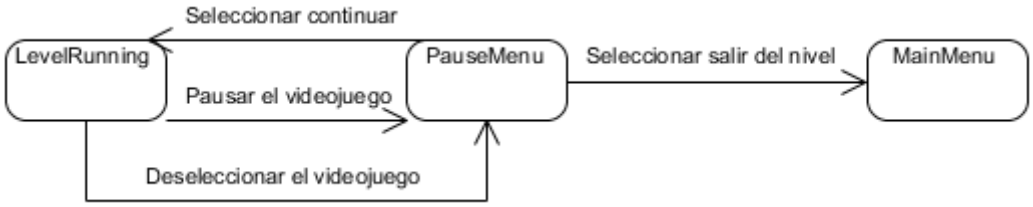


Figura 35. Diagrama de estados de PauseMenu

3.4.6.2. Menús con desplazamiento

Los menús estáticos tienen un problema, si se introducen muchos elementos en el menú, los primeros y los últimos se posicionarían fuera de la pantalla. Como solución, se ha creado una segunda plantilla para crear menús, ScrollableMenu.

Como se puede ver en la *Figura 36*, la clase ScrollableMenu tiene una cámara. Esta cámara permite que, aunque algunos elementos aparezcan fuera de la pantalla, cuando nos desplazamos por el menú, da la sensación de que los elementos también se desplazan, para que el elemento seleccionado siempre esté dentro de la pantalla. Para demostrar el funcionamiento de este tipo de menús, se ha decidido crear un menú de retos, con suficientes

niveles para que no puedan aparecer todos en pantalla a la vez. La *Figura 37* muestra las posibilidades de cambio de estado de ChallengesMenu.

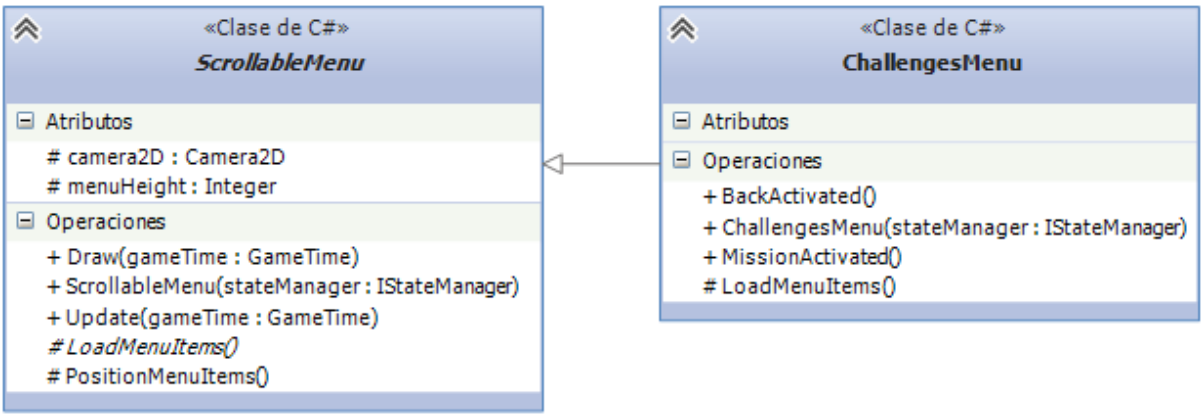


Figura 36. Diagrama de clases de ScrollableMenu

Figura 37. Diagrama de estados de ChallengesMenu

3.4.7. Entidades

Antes de explicar el nivel, que es el único estado que falta por explicar, se van a explicar las entidades que hay dentro de un nivel, para así facilitar la comprensión del nivel.

Al igual que con los menús, con las entidades también se han usado clases abstractas como plantillas. Como se puede ver en la *Figura 38*, la plantilla principal es Entity, que es la clase padre de todas las entidades. Para crear una entidad es necesario especificar una posición. La otra propiedad de Entity sirve para comprobar si la entidad está activa o no. Una entidad puede pertenecer a varias colecciones, si es necesario eliminar una entidad al iterar sobre una colección, habría que recorrer las demás colecciones para eliminarla de todas las colecciones en las que aparece. De esta forma se puede poner la entidad como no activa, y eliminarla de las otras colecciones al iterarlas para realizar alguna operación, y así evitar recorrer todas las colecciones sólo para eliminar un elemento, lo que empeoraría el rendimiento.

Las entidades se han clasificado en 2 tipos: entidades estáticas y entidades animadas. Las entidades estáticas son las que se muestran como una simple imagen. Las entidades animadas tienen una o varias animaciones. Aunque XNA Framework y MonoGame proporcionan las interfaces IUpdateable e IDrawable, debido a que requieren usar propiedades y métodos que no se usan en este videojuego, se han creado las interfaces IUpdateableObject e IDrawableObject. Tanto AnimatedEntity como StaticEntity implementan IDrawableObject, por lo que se podría pensar que debería implementarla Entity, ya que todas las entidades tienen como clase padre a AnimatedEntity o a StaticEntity. La razón por la que Entity no implementa IDrawableObject es porque, en un futuro, se podrían crear entidades que no se muestren en pantalla. Un ejemplo sería si queremos poner que se active un evento cuando el personaje pase por un punto, y en este caso no sería necesario que se muestre ninguna imagen.

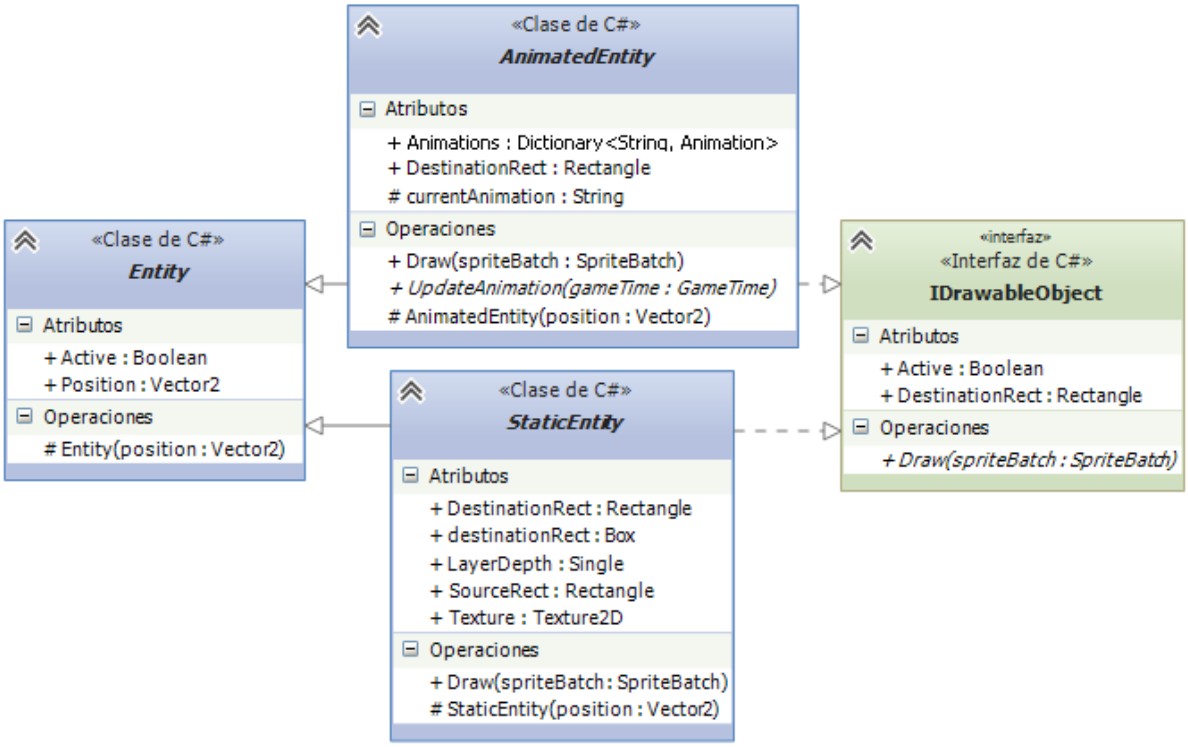


Figura 38. Diagrama de clases de las plantillas de entidades

Para crear las distintas entidades, se ha heredado de estas y se les ha añadido funcionalidad usando interfaces.

3.4.7.1. El personaje

La entidad más importante en el videojuego es el personaje. La clase Player representa al personaje que el jugador controla. El personaje es la única entidad animada en la versión del producto correspondiente a la entrega de este Trabajo de Fin de Grado.

Como se puede ver en la *Figura 39*, la clase Player implementa varias interfaces. Algunas de las interfaces mostradas en el diagrama sólo las implementa Player, por lo que las propiedades y métodos implementados podrían simplemente ser propiedades y métodos de Player, sin necesidad de implementar esas interfaces. La principal razón para la existencia y la implementación de estas interfaces es poder acceder sólo a ciertas propiedades y métodos de Player a través de las interfaces, y ocultar el resto de propiedades y métodos de Player. La interfaz IScore se utiliza para modificar la puntuación del personaje. La interfaz IDamageable se utiliza para reducir la vida del personaje cuando recibe daño. Por último, la interfaz IKeyOwner permite que el personaje consiga llaves y las gaste para abrir puertas.

El personaje actualiza su estado y se muestra por pantalla, haciendo uso de las interfaces IUpdateableObject e IDrawableObject, que implementa AnimatedEntity. El personaje puede tener varios estados (en el aire, herido, etc), que se almacenan en un HashSet de EntityStates.

La interfaz ICollisionable ofrece las propiedades y métodos necesarios para comprobar si la entidad colisiona con otras entidades colisionables. También permite realizar acciones cuando la entidad colisiona con otras entidades. Esta interfaz proporciona la propiedad CollisionType, que permite definir el tipo de objeto colisionable, y en el algoritmo para las colisiones se pueden definir distintas reacciones en función del tipo de objeto colisionable.

La interfaz IMoveable ofrece las propiedades y métodos necesarios para que una entidad pueda moverse. El algoritmo para la gestión de colisiones hace uso principalmente de la

interfaz ICollisionable, y el movimiento del personaje, que es lo que genera que se produzcan colisiones, se controla con IMovable.

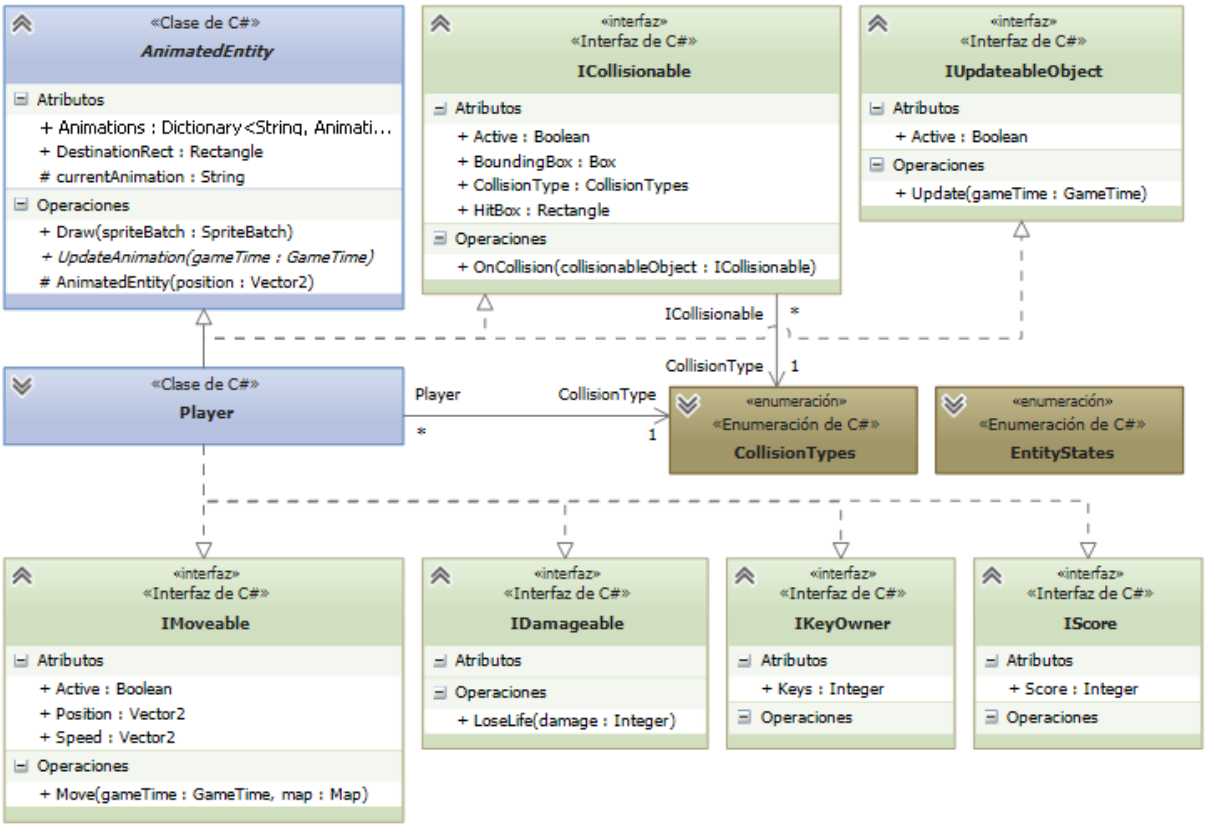


Figura 39. Diagrama de clases de Player

Dependiendo de la entrada recibida, el personaje puede moverse, saltar, o utilizar su habilidad especial "Boost". Para evitar que el personaje pueda caminar por el aire, se ha añadido gravedad. Además, se utiliza aceleración para el movimiento, para que parezca más real, y el personaje frena debido a la fricción. Para que le salto sea fluido, se aplica una gran aceleración, y la gravedad se encarga de que la trayectoria del salto sea parabólica, disminuyendo progresivamente la velocidad en el eje vertical. Para evitar que el personaje pueda saltar en el aire, sólo salta cuando no tiene el estado "en el aire". La habilidad especial "Boost" de aceleración sin gravedad es algo más compleja. Como no interesa que se abuse de la habilidad, se le ha puesto una duración desde que se activa, y un tiempo de recarga para evitar activarla repetidamente. Cada vez que se salta o se activa la habilidad especial "Boost" se reproduce un sonido dependiendo de la acción.

Cuando el personaje recibe daño entra en el estado "herido" y se reproduce un sonido. Mientras el jugador está en este estado, no puede recibir más daño, de esta forma se evita que el jugador muera instantáneamente al recibir daño, ya que recibiría daño en cada actualización, cada pocos milisegundos.

3.4.7.2. Losas simples

El mapa está formado principalmente por *tiles* o losas. Todas las losas de este videojuego son entidades estáticas, y en esta versión de videojuego, sólo existen el personaje y las losas. Una losa es una entidad colisionable como se muestra en la *Figura 40*. Se ha considerado como SimpleTile a las losas más simples, que son aquellas con las que se puede colisionar, pero no se produce ninguna acción al colisionar con ellas, es decir, el método OnCollision está vacío.

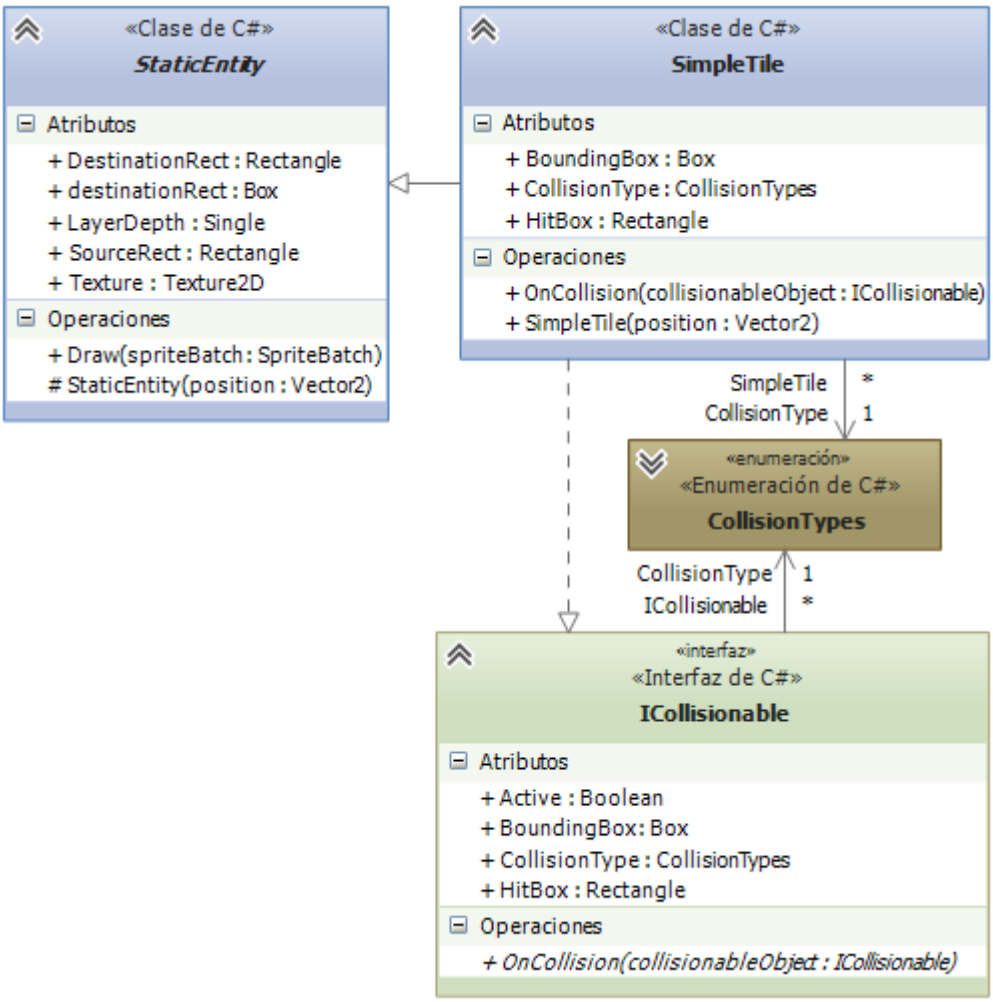


Figura 40. Diagrama de clases de SimpleTile

3.4.7.3. Objetos de puntuación

Como muestra la *Figura* 41, los objetos de puntuación son losas simples, pero sobrescriben el método OnCollision. Cuando otra entidad colisiona con un objeto de puntuación, el método OnCollision comprueba si la entidad colisionable implementa la interfaz IScore. Si es así, se incrementa la puntuación del objeto colisionable con el valor de Score del objeto de puntuación. También se reproduce un sonido en la colisión.

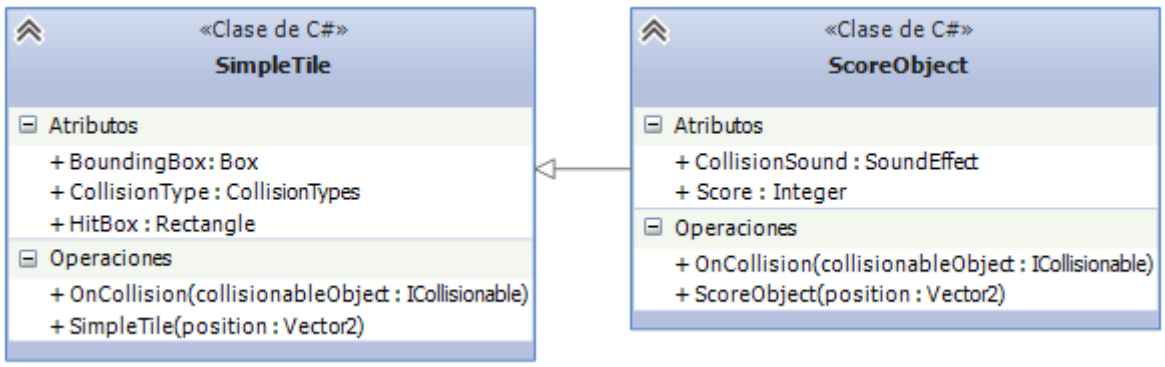


Figura 41. Diagrama de la clase ScoreObject

3.4.7.4. Objetos que hacen daño

Los objetos que hacen daño son los que dan la dificultad al videojuego. Estos objetos también heredan de las losas simples, y como se puede apreciar en la *Figura 42*, también sobrescriben el método OnCollision. En este método se comprueba si el objeto colisionable implementa IDamageable, y si es así, hace uso del método LoseLife para reducir la vida del objeto con el valor de Damage.

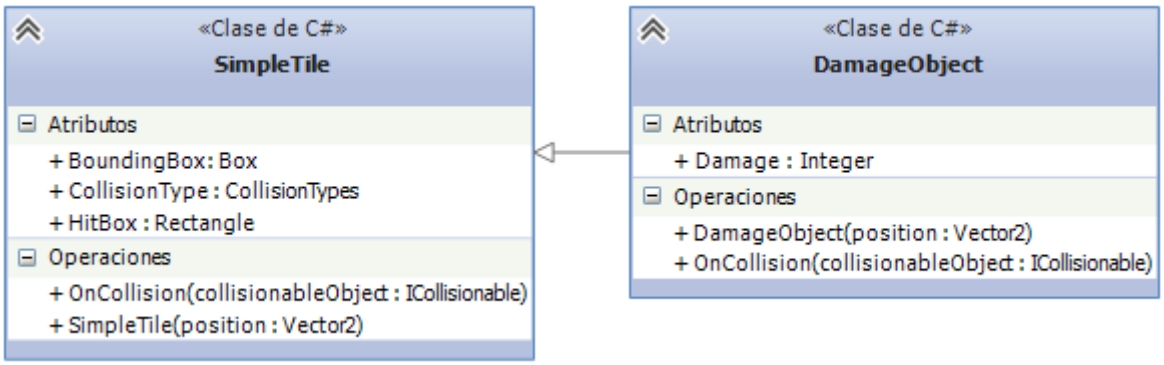


Figura 42. Diagrama de clases de DamageObject

3.4.7.5. Pinchos

Los pinchos son objetos que hacen daño, pero con una particularidad, los pinchos se mueven en vertical dentro de un rango. Como se puede ver en la *Figura 43*, Spike implementa IMoveable. Haciendo uso de la clase Range, definimos que los pinchos sólo puedan desplazarse hasta la posición equivalente a la losa inferior. Para evitar que empiece a subir, o bajar, instantáneamente justo después de llegar a un límite del rango, hacemos uso de la clase Duration, para que no vuelva a moverse hasta que no haya pasado un tiempo.

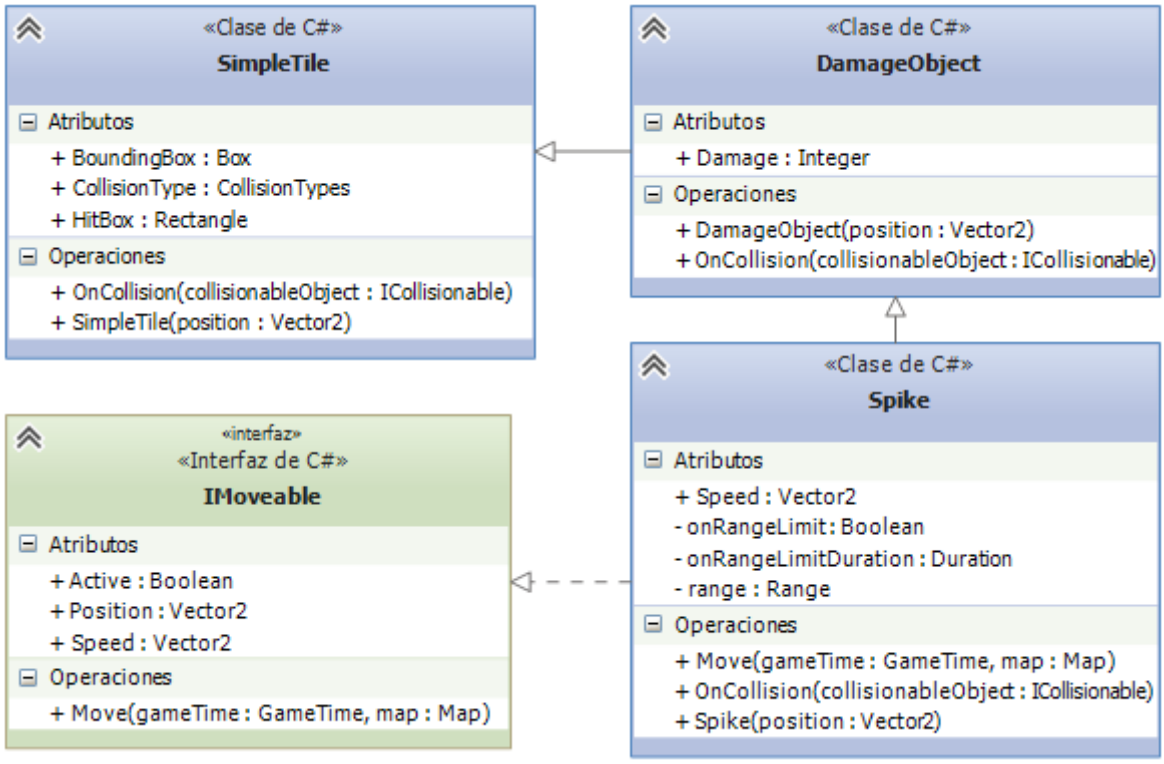


Figura 43. Diagrama de clases de Spike

3.4.7.6. Llaves y puertas

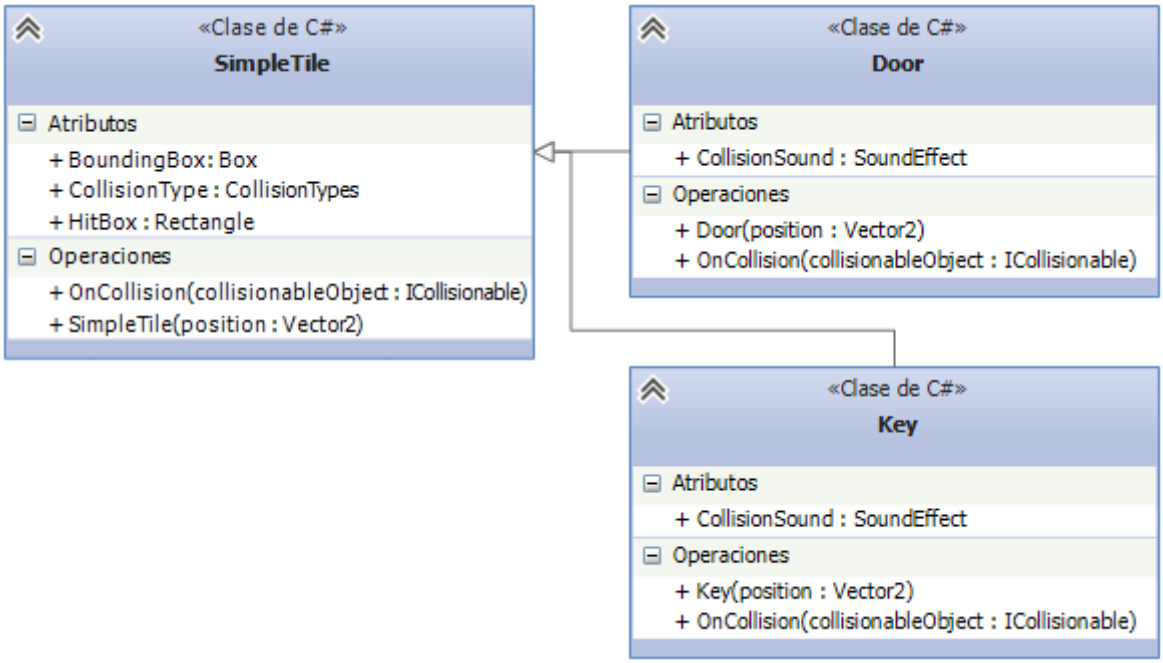


Figura 44. Diagrama de clases de Door y Key

En la versión inicial del videojuego no estaba previsto añadir llaves y puertas, pero al final se han añadido debido a que con el diseño de entidades y creación de entidades, apenas ha supuesto tiempo y esfuerzo. Como muestra la *Figura 44*, y al igual que las entidades anteriores, Door y Key heredan de SimpleTile, y simplemente sobrescriben el método OnCollision. En este caso, la interfaz que se comprueba que implemente el objeto colisionable es IKeyOwner. A través de esta interfaz, el objeto colisionable puede conseguir una llave al colisionar con un objeto de tipo Key, o gastar una llave y abrir una puerta al colisionar con un objeto de tipo Door, mientras el objeto colisionable tenga al menos una llave. Cada entidad reproduce un sonido cuando un objeto que implemente IKeyOwner colisione con ella.

3.4.7.7. La salida

La última entidad es la salida. La salida es una de las entidades más importantes en el videojuego. Un nivel necesita un personaje, un mapa formado por losas, y una salida para poder completar el nivel. La salida también es un tipo de losa simple, como muestra la *Figura 45*, y sobrescribe el método OnCollision. Si el jugador colisiona con la salida, el jugador completa el nivel.

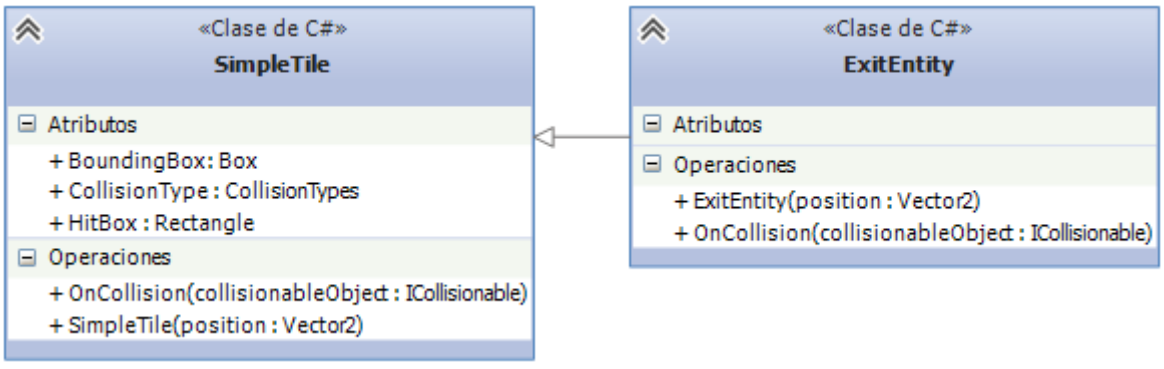


Figura 45. Diagrama de clases de ExitEntity

3.4.8. Nivel

El nivel es el único estado del videojuego que es a su vez un gestor de estados, ya que puede tener varios estados, como se muestra en la *Figura 46*. Una diferencia entre el gestor de estados principal y el nivel, es que cuando se cambiaba de estado no siempre se inicializa el nuevo estado. Un nivel puede tener 2 estados distintos: el estado del menú de pausa, y el estado en el que se está jugando al nivel. El menú de pausa se ha explicado en el apartado "3.4.6.1. Menús estáticos", pero no se ha mencionado que cuando en el menú pausa seleccionamos la opción de continuar, no se inicializa el nivel. Para evitar sobrecargar el estado LevelRunning, se ha separado el mapa del nivel de la cámara, la gestión de entrada y el HUD (*Heads-Up Display*), que es la información que se muestra en pantalla durante la partida, como las vidas, la puntuación, las llaves obtenidas y la barra de información de la habilidad especial "Boost". Si todos los elementos del mapa se almacenaran en una única colección, cada vez que se necesitara mover, actualizar o mostrar por pantalla elementos, habría que recorrer todos los elementos, incluso si no se pudiera mover, actualizar, o mostrar el elemento. Por esta razón se ha hecho uso de varias colecciones, que guardan referencias a los elementos que implementan las interfaces correspondientes. Debido a ser la parte más compleja de explicar y desarrollar, las colisiones se van a explicar lo último, y en este apartado no se va a profundizar en cómo funcionan las colisiones.

Sin duda alguna, el estado más complejo del videojuego es el nivel, por lo que requiere mayor explicación que los otros estados. Para explicar el nivel, se va a dividir la explicación en los 3 métodos que se han ido repitiendo desde la clase Booster hasta el nivel de profundidad actual: Initialize, Update, Draw.

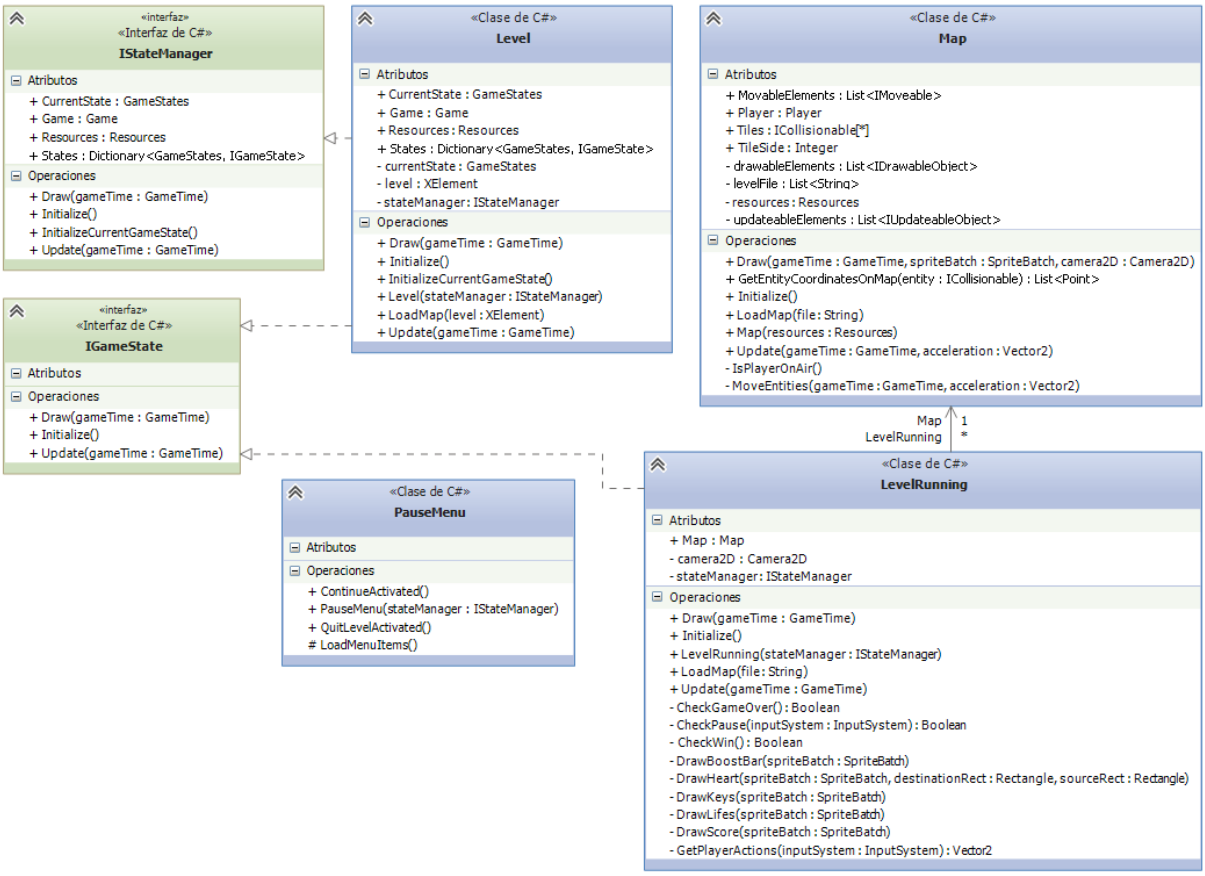


Figura 46. Diagrama de clases del nivel

3.4.8.1. Inicialización del nivel

Para inicializar el nivel necesitamos el archivo que contiene la distribución de elementos en el mapa. Este archivo se ha pasado a Level a través de LoadingState como un objeto XElement, pero en la *Figura 46* se puede ver que el método LoadMap ya no acepta un XElement por parámetro, sino una cadena de texto, que debe ser un archivo. Para poder pasar un archivo, hay que acceder al atributo del objeto XElement que contiene el nombre del archivo de ese nivel. Una vez que el archivo llega al mapa, este lo descompone en una lista de líneas, y después descompondrá cada una en una lista de cadenas de texto.

Para crear el mapa, se tiene en cuenta la cantidad de líneas del archivo y la cantidad de cadenas de texto de la primera línea. Si alguna línea no tiene tantas cadenas de texto como la primera línea, los huecos se rellenarán con bloques. Además, las primeras y últimas filas y columnas siempre van a ser bloques, independientemente de los datos guardados en el archivo. De esta forma, será imposible que el personaje pueda salir de la pantalla. En función de la cadena de texto que se lea, se determinará la entidad que va en esa casilla del mapa. La forma de crear esta entidad se explicará en el apartado "3.4.10. Creación de entidades". Cuando la entidad está creada, consultando las interfaces que implementa, se añade una referencia a la entidad en las colecciones correspondientes.

3.4.8.2. Actualización del nivel

Cuando se actualiza el nivel, se realizan los pasos mostrados en el diagrama de secuencias de la *Figura 47*:

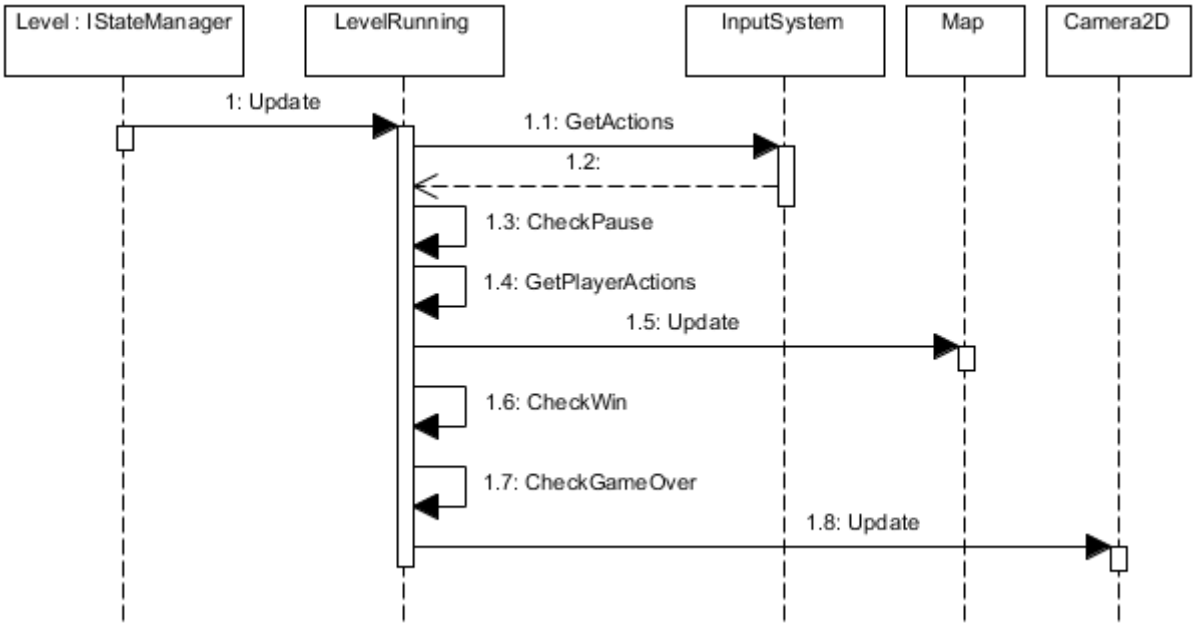


Figura 47. Diagrama de secuencias de actualizar nivel

1. Se obtienen las teclas o botones pulsados.
2. Se comprueba si la tecla o botón de pausar está pulsada, en cuyo caso el estado cambiaría a PauseMenu y se pararía la actualización.
3. Se obtienen las acciones del jugador en función de las teclas y botones pulsados, y del estado del jugador.
4. Se actualiza el mapa.
5. Se comprueba si el jugador ha ganado, es decir, ha llegado a la salida. Si ha ganado, el estado cambia a LevelCompleted y se para la actualización.

6. Se comprueba si el jugador ha perdido, es decir, si el personaje no está activo debido a que su vida ha llegado a 0. Si ha perdido, el estado cambia a GameOver y se para la actualización.
7. Por último, se actualiza la cámara con la posición del personaje para que cuando se muestre el nivel por pantalla, el personaje esté centrado.

Cuando se actualiza el mapa se realizan las operaciones indicadas en el diagrama de secuencias de la *Figura 48*:

1. Se itera sobre la colección de entidades que implementan IUpdateableOject.
    a. En cada iteración se comprueba si la entidad está activa, si no es así, se elimina de la colección. Debido a que se pueden eliminar elementos de la colección, se itera de atrás hacia adelante.
    b. Si la entidad está activa, se actualiza la entidad. Actualizar no implica mover, ya que para mover las entidades es necesario acceder al mapa para comprobar las colisiones.
2. Por último, se mueven las entidades. Para mover las entidades, se itera sobre la colección de entidades que implementan IMoveable.
    a. En cada iteración se comprueba si la entidad está activa, si no es así, se elimina de la colección MovableElements. Esta colección también se recorre de atrás hacia adelante.
    b. Si la entidad está activa, se mueve la entidad, comprobando las colisiones con el mapa.

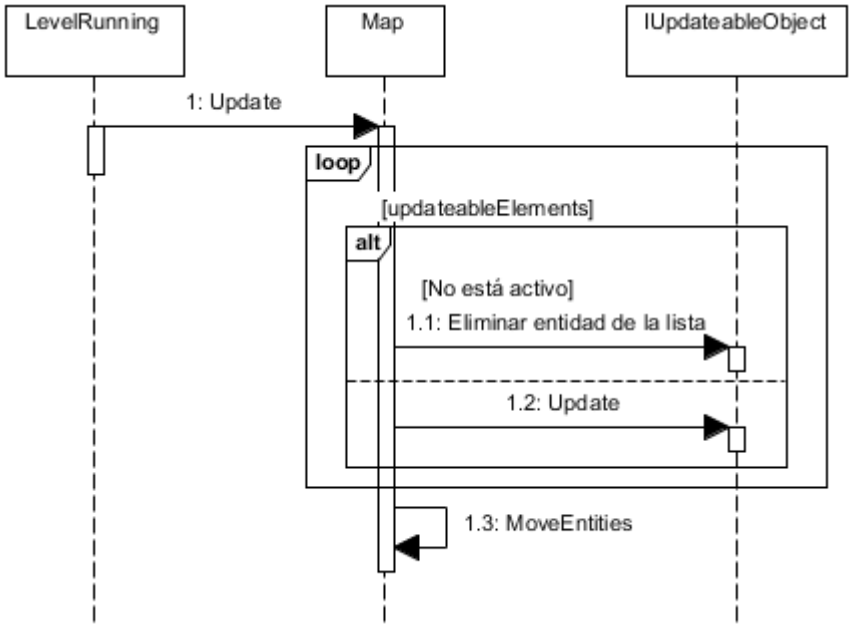


Figura 48. Diagrama de secuencias de actualizar mapa

3.4.8.3. Mostrado por pantalla del nivel

Antes de empezar a llamar a los métodos que mostrarán por pantalla el fondo, el mapa, y el HUD, tenemos que acceder al objeto SpriteBatch. Si nos fijamos en el diagrama de la *Figura 46*, el método Draw de Level y LevelRunning no recibe un SpriteBatch por parámetro, y tampoco tienen un objeto de este tipo como atributo de clase. A lo que sí tienen acceso es a la propiedad Game de IStateManager. En el método LoadContent de Booster se ha creado el objeto SpriteBatch, y se ha agregado como un servicio al videojuego. XNA Framework y MonoGame tienen la propiedad Game.Services, que es una implementación del patrón

Service Locator, y nos permite acceder a los servicios registrados en Game siempre que se tenga acceso a Game.

El proceso de mostrar por pantalla el contenido del nivel está dividido en partes, representadas en el diagrama de secuencias de la *Figura 49*:

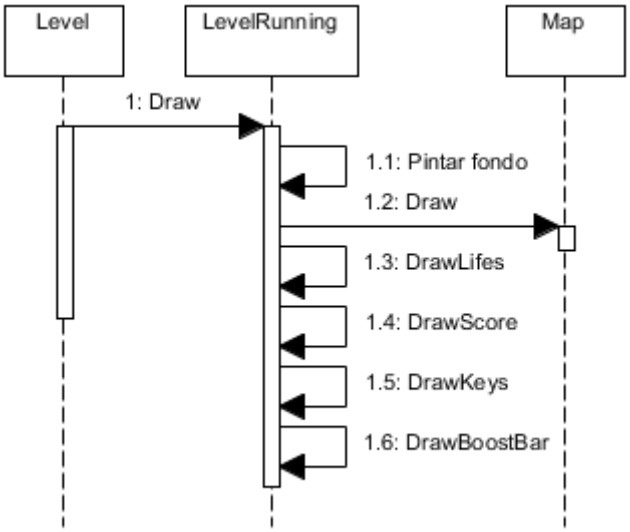


Figura 49. Diagrama de secuencias de mostrar el nivel

1. Como es lógico, lo primero que se pinta queda por debajo de lo demás, como en una pila, así que primero se pinta la imagen de fondo. Se usa el método Begin sin parámetros del SpriteBatch, y al pintar el fondo se utiliza el método End.
2. A continuación, hay que volver a usar el método Begin, pero en este caso pasando la matriz de la cámara, para pintar los elementos del mapa. Para poder tener todos los elementos que se van a pintar en una única colección, y también poder decidir qué elementos se pintan por delante de otros, se ha utilizado el método Draw con el parámetro layerDepth, que permite definir la profundidad con la que se pintará un elemento relativa a los elementos pintados entre las llamadas Begin y End. Para pintar las entidades, se itera sobre la colección de entidades que implementan IDrawableObject:
    a. En cada iteración se comprueba si la entidad está activa, si no es así, se elimina de la colección drawableElements. Esta colección también se recorre de atrás hacia adelante.
    b. Si la entidad está activa, se comprueba que la entidad se vea en la cámara, y en el caso de que la entidad, o parte de ella se vea con la cámara, se pinta la entidad. De esta forma evitamos pintar las entidades que no van a mostrarse por pantalla.
3. Como ya no se va a volver a usar la cámara, volvemos a usar el método Begin del SpriteBatch sin parámetros, y no se llamará al método End hasta que no terminemos de pintar todos los elementos que componen el HUD.
    a. Primero se pinta los corazones que representan las vidas del jugador.
    b. A continuación se pinta la puntuación. Para escoger la imagen que representa cada dígito se hace uso del módulo.
    c. Después se pinta el número de llaves, también haciendo uso del módulo.
    d. Por último se pinta la barra de la habilidad especial “Boost”. La barra se pintará de verde mientras esté llena o vaciándose, y mientras se esté recargando se pintará de rojo. Para determinar la longitud de la barra, se calcula el porcentaje de la duración de la habilidad que falta hasta que la

habilidad entre en el tiempo de recarga, o el porcentaje de la duración de la recarga que ha transcurrido, en el caso de que la habilidad se esté recargando.

3.4.9. Animaciones

La clase AnimatedTexture de XNA Framework y MonoGame permite crear animaciones, y se podría utilizar para crear entidades animadas. La razón por la que se ha implementado la clase Animation es porque AnimatedTexture no permite demasiada personalización, como por ejemplo, no permite escoger el tiempo de duración de cada frame. Mientras que AnimatedTexture requiere que la hoja de sprites de la que se sacan las imágenes de los frames de la animación sólo tenga las imágenes de la animación, situadas en una sola fila y todas del mismo tamaño, la clase Animation que se ha implementado, permite más flexibilidad. Como muestra la *Figura 50*, una animación tiene una lista de frames, y en cada frame se puede decidir qué trozo de una imagen representa, y cuál será la duración de ese frame en la animación. En cada animación se puede decidir el área que ocupará en pantalla, independientemente del tamaño de la imagen; la profundidad con la que se pintará, como se ha explicado en el mostrado por pantalla del nivel, poniendo como ejemplo las entidades estáticas; si la animación se repite; la posición en la que se mostrará; y la escala con la que se mostrará.

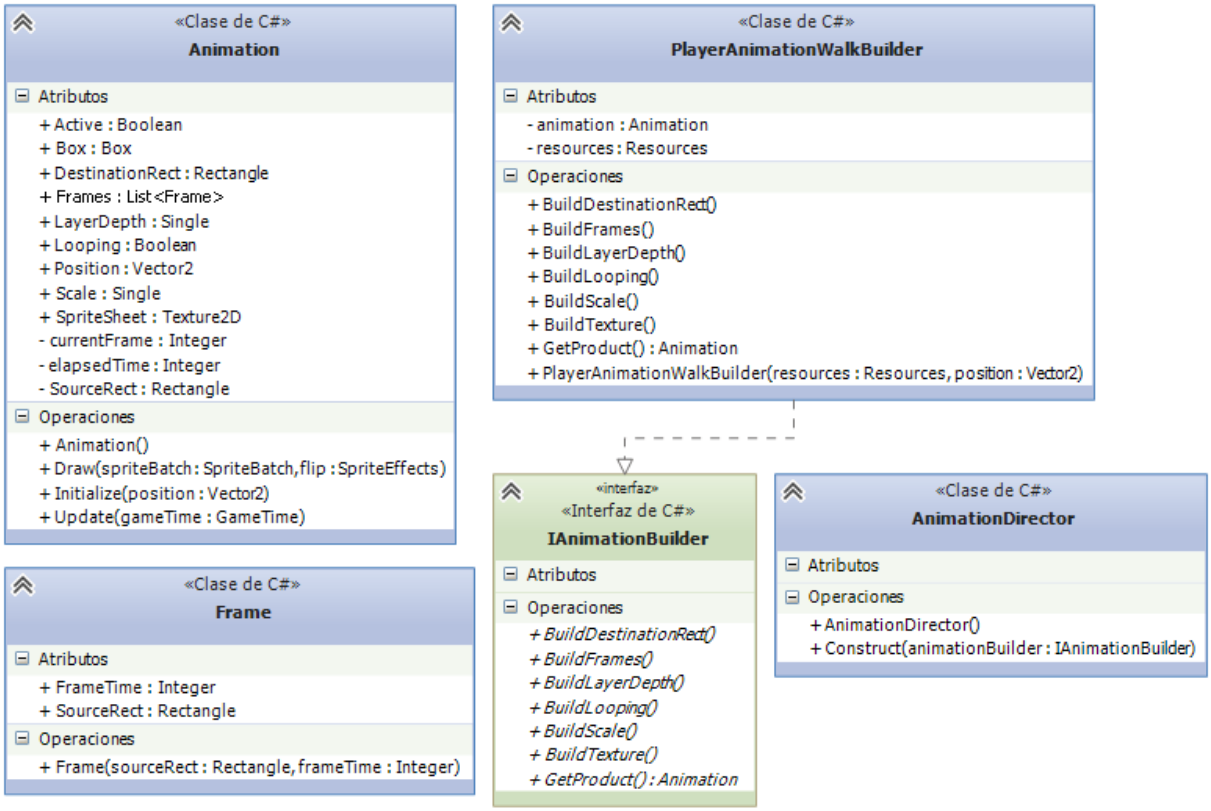


Figura 50. Diagrama de clases de animaciones

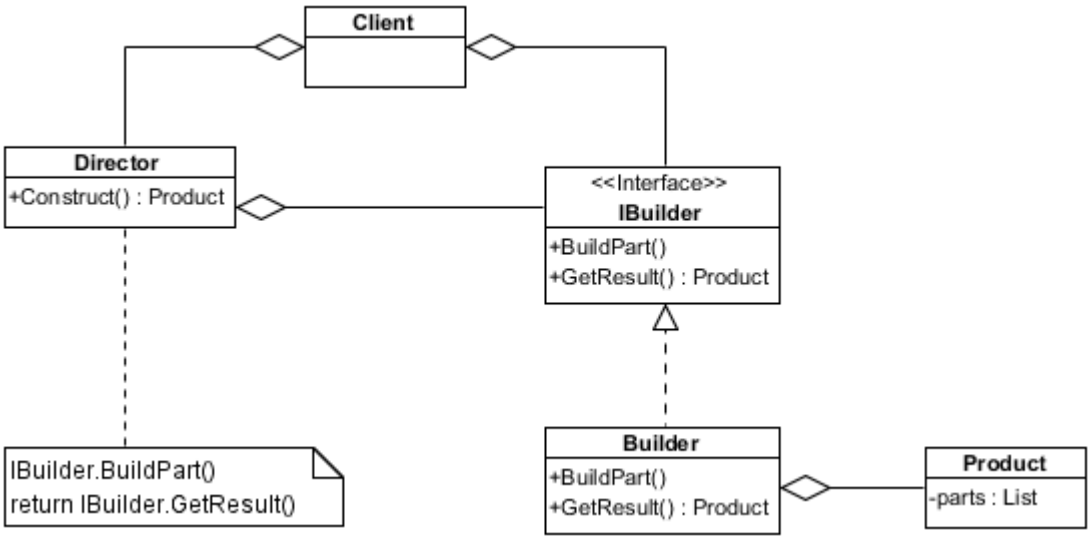


Figura 51. Estructura del patrón Builder

Para crear animaciones se ha hecho uso del patrón Builder. Con el patrón Builder separamos la especificación de un objeto complejo de su construcción. Con el mismo proceso de construcción se pueden crear diferentes representaciones de un objeto [53, 54]. Para que este diagrama sea igual a la estructura del patrón Builder de la *Figura 51*, sólo falta la clase que actuaría de cliente, que es una de las clases encargadas de crear la entidad del personaje, explicada en el apartado "3.4.10. Creación de entidades". Tenemos la clase Animation, con varias propiedades o partes, que corresponde a la clase Product. Para crear un objeto de tipo Animation, tenemos la clase IAnimationBuilder con los métodos Build para construir todas sus partes, y un método para devolver el objeto creado, al igual que la interfaz IBuilder. En la *Figura 50* se muestra un builder que implementa la interfaz, este builder permite construir la animación del personaje andando, y existe un builder para cada animación del personaje. Para construir el objeto haciendo uso de la interfaz IBuilder, hace falta un Director, que, dentro de su método Construct, ejecuta todos los métodos Build de IAnimationBuilder, y por último ejecuta el método GetProduct para obtener el objeto creado y devolverlo a la clase que utiliza el Director, la clase Client.

3.4.10. Creación de entidades

Inicialmente, para crear entidades se utilizaba una clase con un método estático. A este método se le pasaba por parámetro un identificador, para indicar la entidad que se quería crear, y, utilizando un switch, se creaba una entidad u otra. Si nos fijamos en los diagramas de las entidades, cada entidad tiene muchas propiedades, incluyendo las propiedades heredadas. Inicialmente, muchas de estas propiedades se pasaban por parámetro en el constructor, por lo que antes de utilizar el constructor, había que crear y declarar muchos objetos. Esta forma de crear las entidades requería un mínimo de 5 líneas de código para crear una entidad, y en el caso del personaje, con la creación de todas sus animaciones, todavía sin utilizar el patrón Builder para crear las animaciones, superaba las 30 líneas de código. Podemos suponer que cada vez que se quería permitir crear una nueva entidad, este método iba a aumentar considerablemente. El primer paso para reducir el tamaño del método y aumentar su comprensión y mantenibilidad fue usar el patrón Builder para crear las entidades, de esta forma se podía crear cada entidad con sólo 3 líneas de código. Después se utilizó el patrón Factory Method para evitar que el método de crear entidades se tuviera que modificar cada vez que se añadiera la creación de un objeto nuevo.

3.4.10.1. Builder

Aunque en la estructura del patrón Builder sugiere que todas las propiedades del producto se crean o inicializan con los métodos Build, con las entidades no ha sido así, ya que la posición se define al crear la entidad, y por tanto, se pasa por parámetro en el constructor del builder y de la entidad.

- La *Figura 52* representa el diseño de la construcción de **Player**. Se ha utilizado el patrón Builder, al igual que con las animaciones. La clase PlayerBuilder corresponde a la clase Client para la estructura del patrón en la construcción de animaciones.

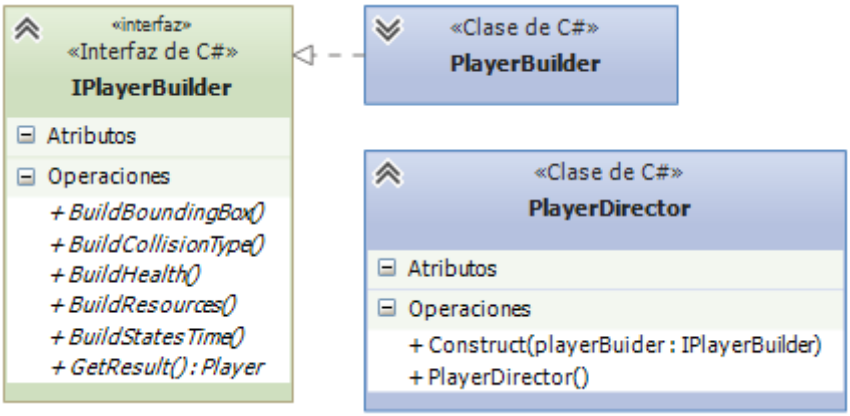


Figura 52. Diagrama de clases de construcción de Player

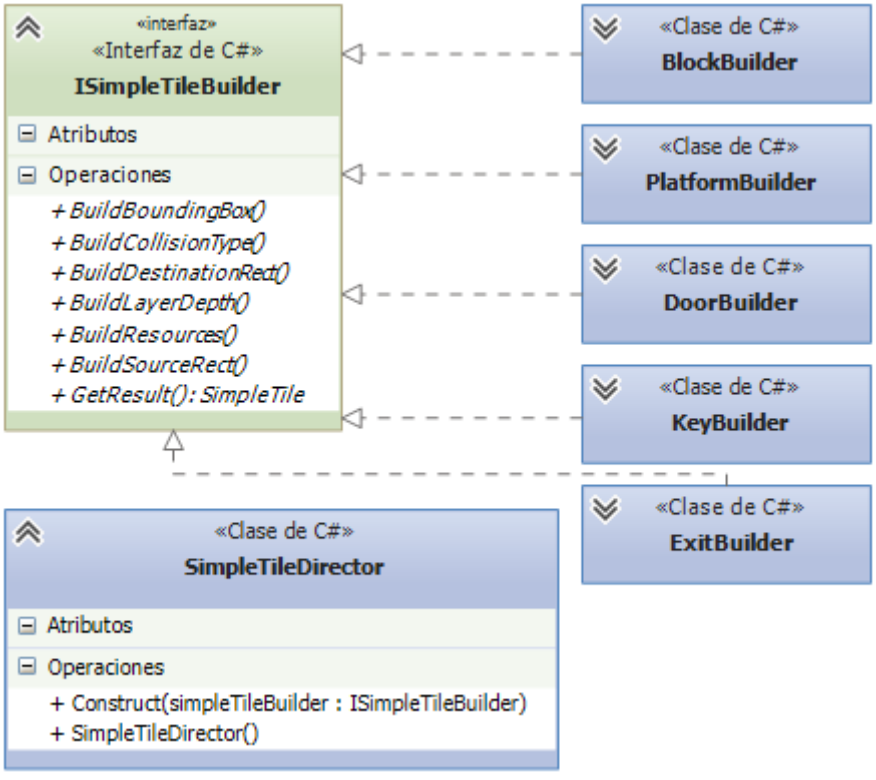


Figura 53. Diagrama de clases de construcción de SimpleTile

- Para crear objetos del tipo **SimpleTile**, se ha utilizado la interfaz ISimpleTileBuilder. Utilizando el mismo Director y la misma interfaz IBuilder, se pueden crear tanto objetos del tipo SimpleTile (BlockBuilder, PlatformBuilder, etc) como objetos de tipos que heredan de SimpleTile, siempre y cuando no tengan nuevos atributos que

puedan configurarse (DoorBuilder, KeyBuilder, ExitBuilder, etc), como está representado en la *Figura 53*. Con el builder de SimpleTile se ve más claro que con el de Player que se el patrón Builder se utiliza para crear representaciones distintas del mismo objeto. A parte de utilizar imágenes distintas para cada elemento, también se puede configurar el tipo de objeto colisionable. Por ejemplo, Un bloque y una puerta no se pueden atravesar, mientras que una plataforma se puede atravesar desde abajo, y una llave no se considera un obstáculo desde ninguna dirección.

- El builder de **ScoreObject** se utiliza principalmente para cambiar la imagen de la entidad y el valor de la puntuación que se obtiene al colisionar con la entidad. En la *Figura 54* se puede ver el método BuildScore, que es la razón por la que no se ha podido reutilizar la interfaz ISimpleBuilder para este tipo de entidades.

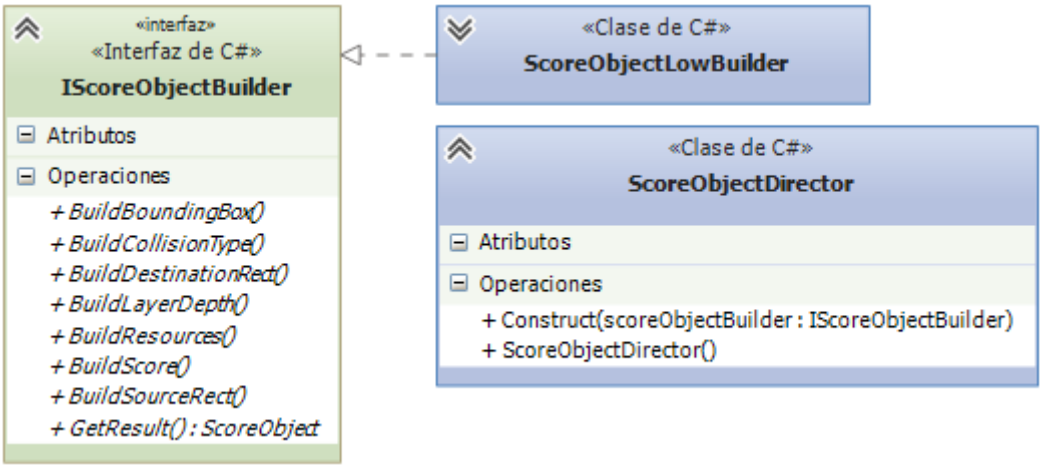


Figura 54. Diagrama de clases de construcción de ScoreObject

- El builder de **DamageObject** se reutiliza para los pinchos, como se puede ver en la *Figura 55*, ya que las propiedades que añaden nuevas a las de DamageObject son siempre las mismas, y no hace falta utilizar un método Build para inicializarlas.

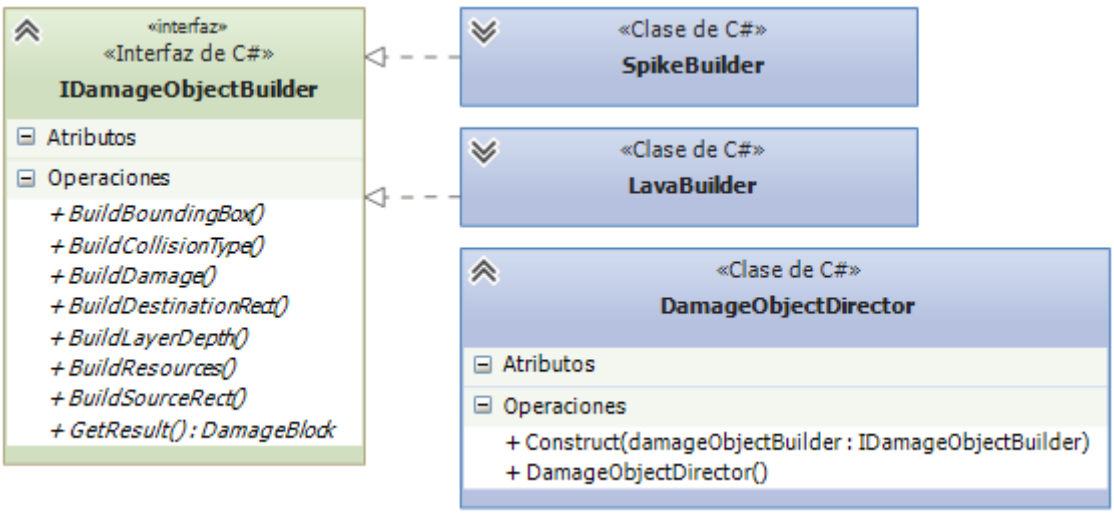


Figura 55. Diagrama de clases de construcción de DamageObject

3.4.10.2. Factory Method

Con el uso del patrón Builder se consiguió reducir un método de 5-30 líneas por representación de una entidad a 3 líneas por representación. Pero aun así, el método iba a

seguir aumentando en tamaño, lo que iría contra el principio Open/closed de la Programación Orientada a Objetos. Para evitar esto, se ha utilizado el patrón Factory Method.

El patrón Factory Method permite una forma de crear objetos, dejando a las subclases decidir qué clase instanciar [53, 54]. Como se puede ver en la *Figura 56*, creando una subclase por cada tipo de producto se respeta el principio Open/closed, de esta forma el software está abierto a extensiones, y cerrado a modificaciones.

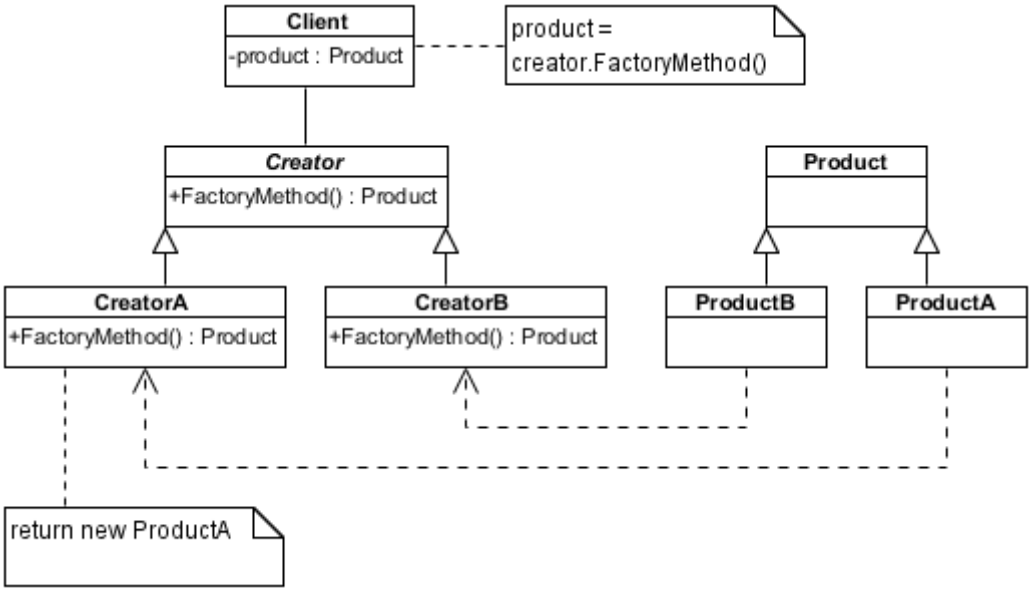


Figura 56. Estructura del patrón Factory Method

La *Figura 57* muestra el patrón Factory Method adaptado a la creación de entidades. En este caso los productos son las entidades, pero en lugar de crear las entidades directamente, se ha combinado el patrón Builder con el Factory Method. De esta forma, con el patrón Builder se definen las representaciones, y con el Factory Method se crean las entidades a través de los directores del patrón Builder. Cuando se ha explicado el patrón Builder aplicado a la creación de entidades, no se ha explicado qué correspondía a la clase Client de la estructura del patrón. La clase Client corresponde a la subclase de EntityCreator que crea la representación del objeto correspondiente.

Figura 57. Diagrama de clases de Factory Method para la creación de entidades

Debido a que las entidades se crean en tiempo de ejecución cuando se lee el contenido del nivel del archivo, es necesario asignar a la cadena de texto que representa la entidad en el archivo una subclase de EntityCreator. Esta asignación se realiza haciendo uso de mapas, de

forma que con la cadena de texto como clave, se obtiene la subclase de EntityCreator necesaria para crear la entidad correspondiente. Se podría pensar que sólo con el uso de mapas se podría suprimir el switch original, pero no es posible introducir la creación de un objeto en un mapa, para ello es necesario introducir otro objeto que tenga la función de crear el objeto deseado, por lo que se ha utilizado el patrón Factory Method.

3.4.11. Colisiones

La gestión de colisiones es la parte del videojuego a la que se le ha dedicado más tiempo. A lo largo del desarrollo del videojuego, ha habido 4 formas distintas y funcionales de gestionar las colisiones. Primero se empezó a implementar un algoritmo sencillo para la gestión de colisiones, y al probarlo por primera vez, el algoritmo falló. Entonces, se buscó como alternativa una librería de físicas para que gestionara las colisiones.

3.4.11.1. Farseer Physics Engine

Farseer Physics Engine es un sistema de detección de colisiones con respuestas físicas realistas. Se puede crear desde un simple videojuego por hobby hasta una simulación robótica compleja. Está basado en Box2D, por lo que tiene muchas características en común, pero también añade características que no tiene Box2D [56].

Para gestionar las colisiones hace faltan cuerpos físicos, que están compuestos por varias partes, pero usando las factorías proporcionadas por la librería, sólo será necesario crear una de las partes, la factoría se encarga de crearlas todas y juntarlas. Los cuerpos físicos se añaden al mundo físico, y cuando este se actualiza, los cuerpos físicos se mueven y Farseer se encarga de gestionar las colisiones.

Pero utilizar un motor de físicas no es tan fácil como parece. El mundo físico mide las distancias en metros, pero para mostrar los elementos por pantalla hay que convertirlos a píxeles. Además, para mover los elementos hay que aplicarles una fuerza, por lo que es más difícil controlar la velocidad, por lo que si añadimos plataformas móviles al videojuego, sería muy difícil de controlar que el personaje se moviera a la vez que la plataforma, pero que se pueda caer de ella, y moverse independientemente de la plataforma mientras no esté encima de ella. Por último, y sin duda la principal razón por la que se han buscado alternativas a un motor de físicas, es que varios bloques juntos no son considerados una superficie plana en el motor de físicas. Para explicar el problema que supone esto, nos vamos a apoyar en la *Figura 58*. Debido a la gravedad, el personaje (P), se mueve en diagonal hacia abajo, y como el motor de físicas no interpreta los bloques 1, 2 y 3 como una superficie plana, a veces el motor interpreta que el personaje está chocando con el bloque 3, como si hubiera una pared invisible que le impide continuar. Para solucionar esto hay que trazar el contorno de los bloques, y añadir al mundo físico el contorno en lugar de los bloques. Pero si eliminamos uno de los bloques, habría que destruir el contorno del mundo físico, y volver a crear un nuevo contorno para substituirlo, lo que puede ser bastante costoso. No es lo mismo realizar una operación costosa en la inicialización, que es sólo una vez y aún no se ha empezado a jugar al nivel, que varias veces durante la ejecución del nivel, que puede ralentizar el videojuego. Por esta razón, se decidió reintentar implementar el algoritmo de gestión de colisiones, teniendo en cuenta que en un videojuego de plataformas en 2 dimensiones basado en losas, las colisiones son relativamente fáciles.

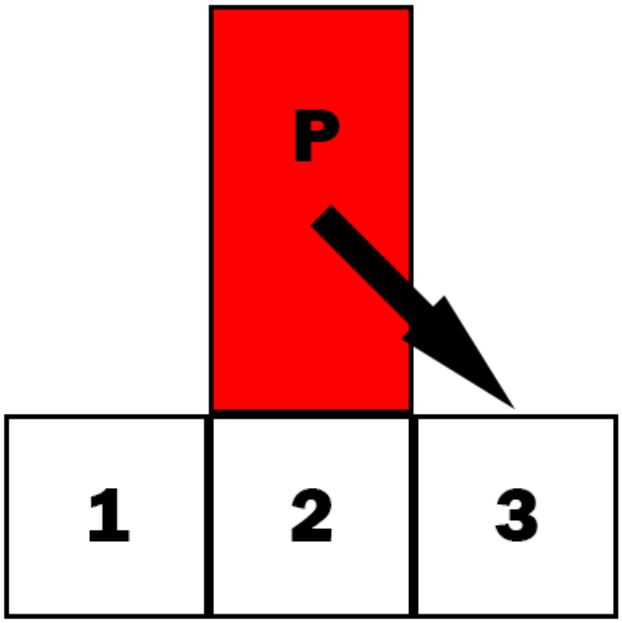


Figura 58. Situación problemática para la gestión de colisiones con Farseer Physics Engine

3.4.11.2. Comprobación píxel por píxel

El siguiente algoritmo funcional recursivo que se implementó comprobaba píxel a píxel. El funcionamiento de este algoritmo era el siguiente:

1. Se obtiene la distancia recorrida por el personaje desde la última actualización.
2. Se descompone el movimiento en muchos movimientos pequeños, que no avanzan más de un píxel en ninguna dirección.
3. Comprueba que el área del personaje después de realizar un movimiento pequeño no interseccione con el área del resto de elementos del mapa.
    a. Si interseccciona, no mueve más el personaje, a no ser que aun deba moverse en los 2 ejes para recorrer la distancia calculada previamente. Entonces llama recursivamente al algoritmo con el movimiento que le queda por realizar en horizontal, y después en vertical. De esta forma, en uno de los 2 ejes, no se moverá, pero se seguirá moviendo en el eje que no tiene obstáculos.
    b. Si no intersecciona, realiza ese movimiento pequeño y pasa a comprobar el siguiente.

Aunque es un algoritmo sencillo de implementar, tiene una gran desventaja, que si el personaje se va a mover 27 veces, habría que recorrer 27 veces todas las losas con las que el personaje pueda colisionar al moverse. Para evitar esta ineficiencia, este algoritmo se ha cambiado por otro que, aunque es más complicado de implementar, es mucho más eficiente.

3.4.11.3. Comprobación por losas

La premisa detrás de esta implementación es que si el personaje se va a mover del punto A al punto B, se recorre la matriz que representa el mapa en busca de obstáculos entre A y B. Para evitar recorrer más losas de las necesarias, se acotan las casillas de la matriz que se recorren en función de la posición actual del personaje y la posición que tendría si no encontrara ningún obstáculo en el movimiento. Se comprueba con los elementos de la matriz si el personaje se vería bloqueado por alguno de ellos. Si es así, el personaje completa su movimiento quedándose justo al lado del elemento que lo bloquea. Este movimiento se realiza moviendo primero al personaje en el eje horizontal, y después en el vertical.

Debido a que sólo interesa encontrar el primer elemento que bloquea al jugador, es necesario recorrer la matriz en un orden distinto dependiendo de la dirección del movimiento, por lo que son necesarios al menos 4 métodos para las distintas direcciones del movimiento. Aunque el movimiento puede tomar 8 direcciones distintas (4 si sólo se mueve en un eje y 4 por las diagonales), añadiendo una condición más a los movimientos horizontales es posible controlar también los movimientos diagonales.

En el caso del movimiento hacia arriba, es tan simple como parar si se encuentra un obstáculo. En el movimiento hacia abajo hay que tener en cuenta que puede haber objetos colisionables que se pueden atravesar hacia arriba, pero no hacia abajo, por lo que habrá que comprobar que el jugador esté por encima de ellos para considerarlos un obstáculo. Los movimientos laterales funcionan como el movimiento hacia arriba, pero para tener en cuenta los movimientos laterales se ha añadido una condición extra, que comprueba si el elemento anterior en el eje horizontal es un elemento bloqueante.

1. La *Figura 59* representa una situación en la que el elemento anterior es un elemento bloqueante. Como se puede ver en la parte izquierda de la figura, el personaje (P), se movería, pero el personaje después de moverse (M) estaría colisionando con los bloques. Primero se comprueba si el personaje, situado pegado al bloque 2, colisiona con él. Para que colisione durante su movimiento, se comprueba que, en una posición intermedia entre las posiciones actual y futura del personaje, si el personaje avanza un píxel más, colisionara con el bloque si este está delante de él. En este caso, P no colisiona con el bloque 2, pero sí colisionaría con el bloque 3, y como hay un bloque justo antes, el personaje no se para, ya que se mueve por una superficie plana. Una vez determinada la posición en el eje horizontal, se intenta mover el personaje hacia abajo, pero como colisiona con los bloques 2 y 3, mantiene su posición en el eje vertical, por lo que la nueva posición del personaje sería la representada en la parte derecha de la figura.

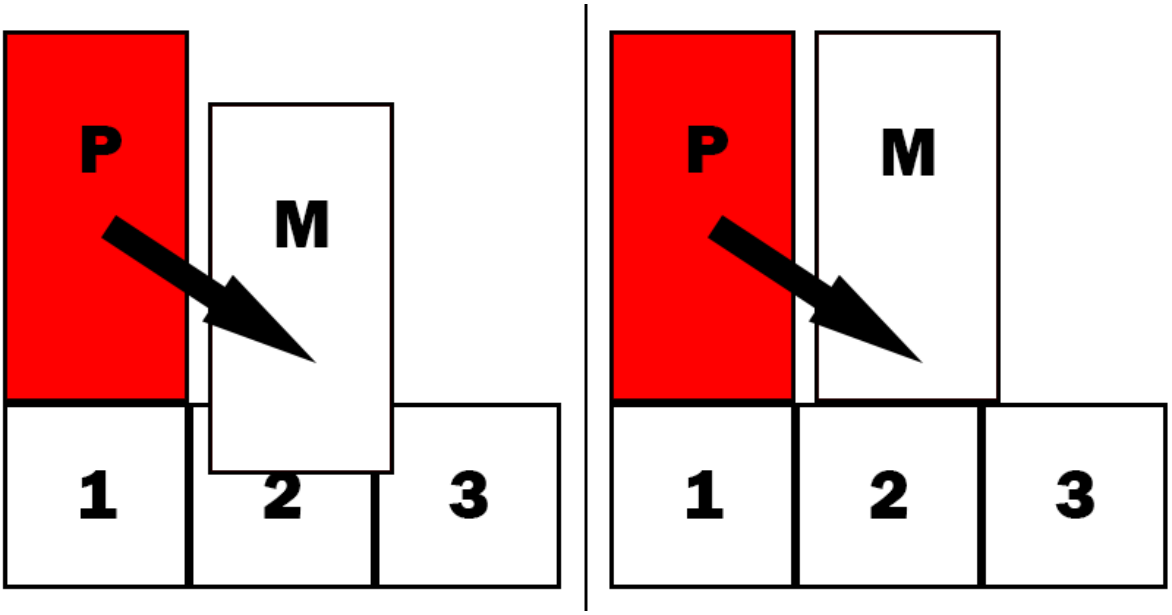


Figura 59. Gestión de colisiones en movimiento diagonal sobre superficie plana.

2. La *Figura 60* representa el otro caso, no hay un elemento anterior, o si lo hay, no bloquea el movimiento. Como se puede ver en la parte izquierda de la figura, el personaje colisionaría con el bloque 3, pero el resultado esperado no es el mismo que en el caso anterior, ya que aquí no hay una superficie plana. En este caso, como hay un hueco antes del bloque 3, la posición en el eje horizontal sería la posición del hueco,

ya que debido a la gravedad, el personaje caería por él. Y al aplicar el movimiento en el eje vertical, la posición final sería la representada en la parte derecha de la figura.

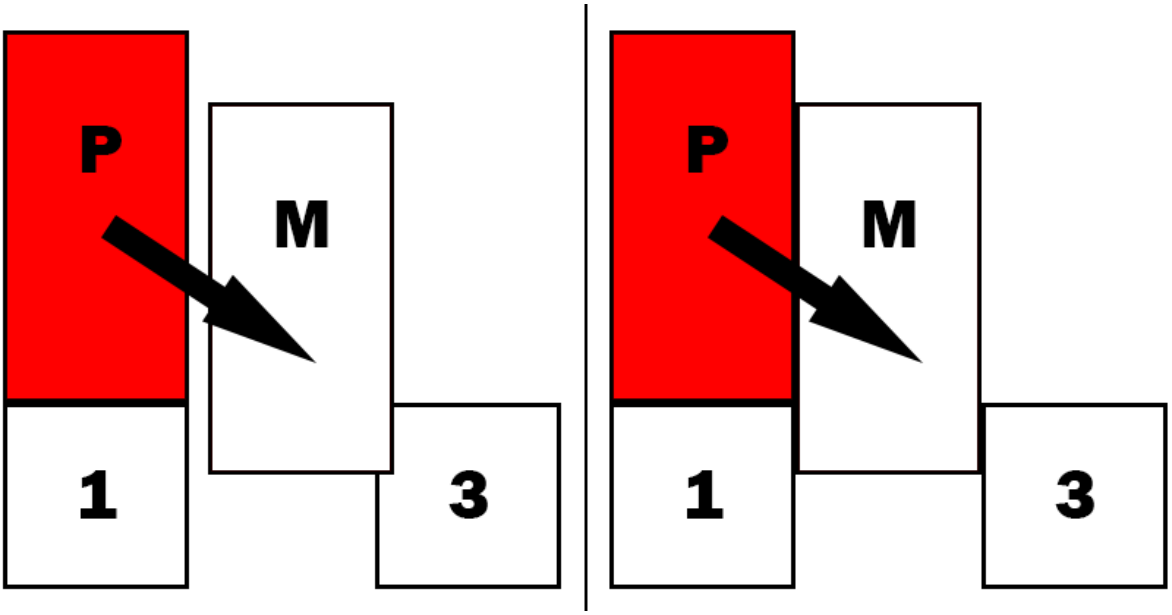


Figura 60. Gestión de colisiones en movimiento diagonal con huecos

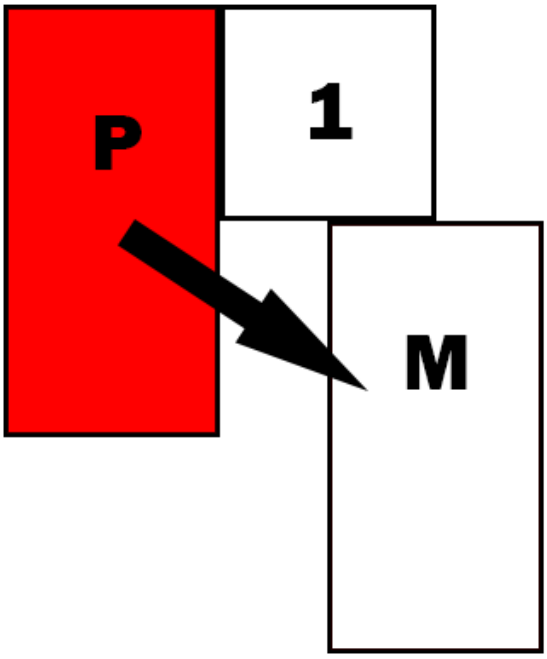


Figura 61. Caso problemático en el algoritmo de colisiones de comprobación por losas

Aunque esta implementación es mucho más eficiente que la comprobación por píxel, sigue teniendo problemas. Por ejemplo, si el personaje estuviera encima del hueco en la *Figura 60*, el algoritmo determinaría que el personaje avanzaría en lugar de caerse por el hueco. La solución a este problema es sencilla, pero hay otro problema no tan sencillo de resolver. Supongamos el caso de la *Figura 61*, con este algoritmo se detectaría colisión al mover hacia la derecha, por lo que el personaje no avanzaría en el eje horizontal, y después bajaría, pero realmente sí sería posible que llegara a la posición de M, bajando y después avanzando hacia la derecha, que en este caso no avanzarías hasta la siguiente actualización del estado del

videojuego. No es fácil poder determinar si el personaje debería moverse primero en el eje horizontal y después en el vertical, o viceversa, ya que cuando le decimos que se mueva, no se tiene ninguna información sobre cómo colisionará con el mapa. Para aplicar esto último, el algoritmo se ha mejorado, y complicado.

3.4.11.4. Comprobación por losas mejorada

El primer cambio con respecto al algoritmo anterior es que el algoritmo anterior tenía 4 métodos (derecha, izquierda, arriba y abajo), y se utilizaban 1 ó 2 de ellos para mover el personaje. La versión mejorada ha separado el movimiento en 8 estrategias, correspondientes a las 8 direcciones que puede tomar el movimiento, en este algoritmo sí se incluyen las diagonales. El patrón Strategy elimina un algoritmo de la clase que lo utiliza y lo pone en una clase diferente. Puede haber varios algoritmos, o estrategias, aplicables a un problema, que si se dejan en la clase que lo utiliza, podríamos encontrarnos con código confuso con muchas comprobaciones [53, 54]. Como muestra la *Figura 62*, la clase Client tiene un objeto que implementa la interfaz IStrategy, y cuando llama al método Algorithm, se ejecuta el algoritmo de la estrategia seleccionada.

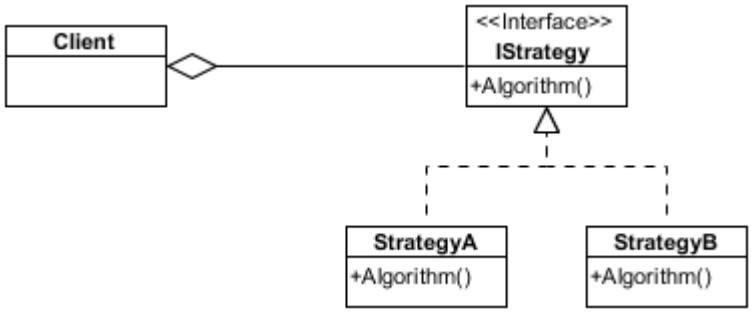


Figura 62. Estructura del patrón Strategy

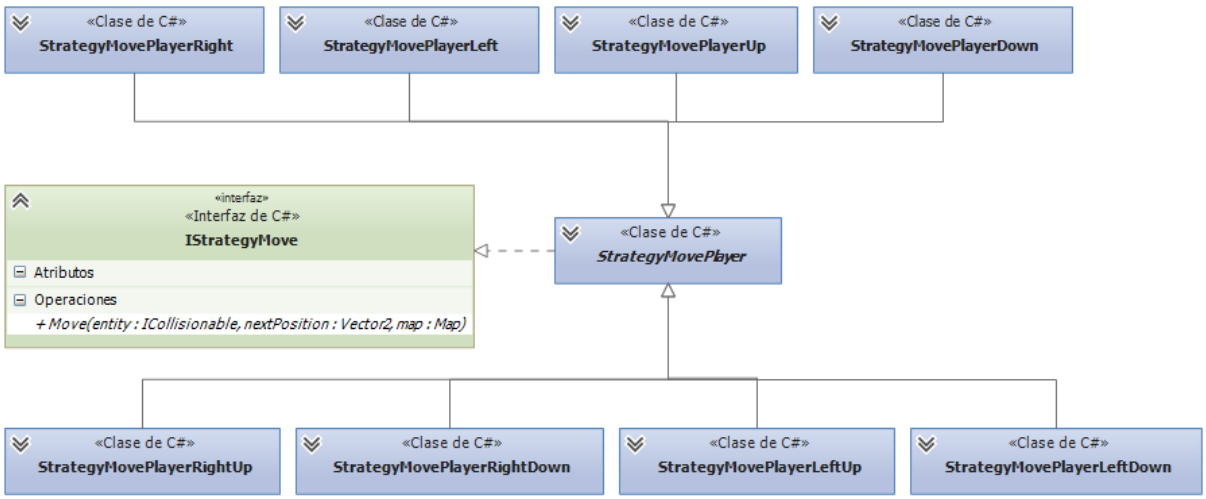


Figura 63. Diagrama de clases de la gestión de colisiones

En este proyecto, el cliente es el personaje, que dependiendo de la aceleración, escoge una estrategia u otra. Como se puede ver en la *Figura 63*, tenemos una interfaz con el método Move, y muchas clases que implementan la interfaz. En este diagrama se muestra que sólo una clase implemente la interfaz, y las estrategias hereden de esta clase, y, aunque esto podría hacernos dudar de que se esté usando el patrón Strategy, realmente se está usando. En este caso, sólo existen estrategias para mover al jugador, pero también podría haberlas para el movimiento de los enemigos. Además, no sería totalmente necesaria la clase StrategyMovePlayer, simplemente haciendo que sus subclases implementen IStrateyMove,

pero la clase StrategyMovePlayer implementa métodos que facilitan la gestión de colisiones a las estrategias, que evitan tener código duplicado en las estrategias.

Las estrategias para mover el personaje arriba y abajo funcionan exactamente igual que en la versión anterior del algoritmo, y los movimientos laterales funcionan como el movimiento hacia arriba, esta vez sin la condición extra, ya que si el personaje tiene aceleración en más de un eje, se utilizan los movimientos diagonales.

En los movimientos diagonales se recorre el mapa hasta que se encuentra una losa con la que el personaje colisionaría.

1. Si no colisiona con ninguna, el personaje se mueve a la posición calculada.
2. Si colisiona por un lateral, se mueve hasta la posición de colisión, y después se mueve en el eje vertical.
3. Si colisiona por arriba o por abajo, se mueve hasta la posición de colisión, y después se mueve en el eje horizontal.

Para moverse en el eje vertical u horizontal no utiliza los métodos de las estrategias que mueven en un solo eje, ya que hay que tener en cuenta que el jugador sigue teniendo aceleración en el otro eje. Esto puede causar que si está cayendo al suelo, primero tenga un movimiento diagonal, al colisionar con el suelo tenga un movimiento en el eje horizontal, pero si hubiera un hueco en la trayectoria, el personaje volvería a tener un movimiento diagonal. En este algoritmo ya está solucionado el problema de qué pasaría si el personaje está justo encima de un hueco, por lo que si se adopta un movimiento diagonal sobre un hueco, el personaje cae por el hueco. Aunque hay un método para el movimiento diagonal que puede llamar a otros métodos para el movimiento en un solo eje, y estos a su vez pueden volver a llamar al método de movimiento diagonal, el personaje para de moverse cuando llega a su posición de destino o se para debido a las colisiones, por lo que no se dan casos de infinitas llamadas recursivas. Hay que puntualizar que, aunque el personaje puede seguir bajando si colisiona con el suelo, no puede seguir subiendo si colisiona con el techo, debido a que perdería la aceleración que lo hacía subir, quedando sólo la gravedad. En conclusión, en un movimiento en un solo eje, el personaje para cuando colisiona con un obstáculo o cuando encuentra un hueco en su lateral o abajo, volviendo a realizar un movimiento diagonal.

Para explicar mejor el algoritmo se va a hacer uso del ejemplo de la *Figura 64*, mostrando un movimiento en 4 pasos. Suponemos que el movimiento a realizar es el indicado en el primer paso.

1. Primero iniciamos un movimiento diagonal, el personaje colisiona con el bloque 1, por lo que se bloquea en un lateral sin que el personaje avance. Como se ha bloqueado en un lateral, el personaje se mueve hacia abajo, hasta llegar al segundo paso.
2. En el segundo paso, el personaje ha bajado hasta encontrar un hueco en su lateral, por lo que se para y comienza un movimiento diagonal, dando lugar al estado representado en el tercer paso.
3. Para llegar al tercer paso, el jugador ha realizado un movimiento en diagonal, y ha parado al colisionar con el bloque 2, también en un lateral. Entonces el jugador inicia otro movimiento hacia abajo, llegando al estado del cuarto paso, y esta vez para porque ha alcanzado la posición en el eje vertical calculada, y no puede avanzar más en el eje horizontal debido a un obstáculo.

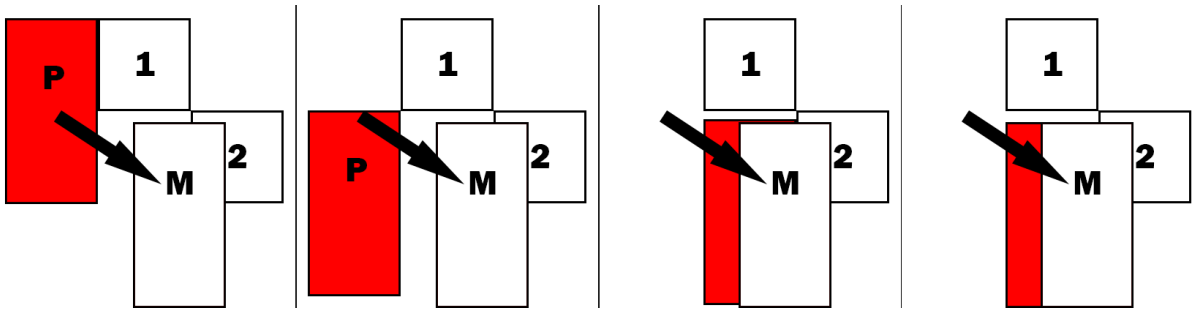


Figura 64. Gestión de colisiones en movimientos diagonales con la versión mejorada del algoritmo

Falta mencionar que, mientras se van haciendo las comprobaciones de colisión con los elementos del mapa, se van añadiendo los elementos con los que colisiona a una colección, donde también se añaden los elementos móviles con los que colisiona, que no están en la matriz que representa el mapa. Al determinar un movimiento, se ejecuta el método OnCollision de estos elementos, para que el jugador pierda vida, consiga puntos, o simplemente se escuche un sonido.

3.5. Planificación temporal

Aunque se empezó a trabajar con XNA Framework en Octubre de 2013, el proyecto no se empezó hasta Diciembre de 2013. En Octubre, el autor se dedicó a aprender el funcionamiento de XNA Framework, buscar tutoriales de videojuegos sencillos hechos con XNA Framework y realizar estos tutoriales con el objetivo de aprender, y buscar información sobre cómo hacer transiciones entre estados, ya que los tutoriales sólo se centraban en la parte jugable, sin ningún tipo de transición.

Debido a las prácticas y exámenes de la universidad, y a las prácticas en empresa, ha habido épocas en las que no se ha podido hacer nada del proyecto por falta de tiempo.

Cuando el autor decidió que iba a desarrollar un videojuego como Trabajo de Fin de Grado en Septiembre de 2013, tenía la cabeza llena de ideas para crear un videojuego de plataformas, y conforme avanzaba el tiempo, su idea del videojuego iba cambiando, haciendo que cada vez fuera más grande. Durante los primeros días de Noviembre de 2013, se realizó un segundo análisis del videojuego, en el que se descartaron varias ideas, algunas debido a que requerirían imágenes hechas expresamente para este videojuego, por lo que no se podrían encontrar paquetes de imágenes que suplieran estas necesidades, y el autor carece de las capacidades para crearlas con una calidad decente.

A continuación, y hasta el 24 de Noviembre, se hizo un diseño preliminar del videojuego. El objetivo de este diseño no era hacer un diseño completo e inquebrantable, sino determinar las relaciones entre las clases, y determinar qué clases tenían elementos comunes, para evitar repetir código creando clases que actuaran como una plantilla para sus subclases.

A partir del 21 de Diciembre se empezó el primer ciclo de desarrollo. Este proyecto se ha desarrollado con un enfoque iterativo e incremental, y las actividades realizadas hasta este momento sirvieron para determinar la división de tareas. No se estimó el tiempo que conllevaría hacer cada tarea, ya que habrá que desarrollar algoritmos complejos, y no se puede estimar cuanto tiempo se tardará en desarrollar un algoritmo, ya que no se puede saber cuánto tiempo habrá que dedicar a solucionar errores hasta que el algoritmo funcione perfectamente. En algunas tareas apenas fue necesario dedicar tiempo al análisis y el diseño, ya que el análisis y diseño iniciales de algunas tareas estaban casi completos. Se fueron

realizando las tareas relacionadas con los objetivos propuestos primero, pero algunas de estas tareas se dejaron para casi al final, ya que, después de leer algunos libros, se determinó que iban a ser rápidas de desarrollar, y se dio prioridad a otras tareas que mejorarían el aspecto y funcionamiento del videojuego o la calidad del código.

La *Figura 65* muestra la cronología del desarrollo, con la tarea desarrollada en cada iteración.

Iter.	Duración	Tarea	Observaciones
1	21/12/2013 – 21/12/2013	Gestor de estados	Debido a no existir todavía ningún estado, no se pudo probar.
2	21/12/2013 – 22/12/2013	Menú	Sólo se desarrolló la clase Menu, como plantilla, y el menú principal.
3	22/12/2013 – 23/12/2013	Entidades básicas	Se crearon las plantillas de las entidades, la clase Animation, el jugador y SimpleTile para poder crear bloques.
4	23/12/2013 – 26/12/2013	Nivel	Se creó el estado nivel, con un jugador que se podía mover por toda la pantalla y bloques.
5	26/12/2013 – 31/12/2013	Colisiones – Farseer Physics Engine	Se añadió física al jugador y se utilizó un motor de físicas para gestionar las colisiones.
6	03/01/2014 – 07/01/2014	Colisiones – Comprobación por píxel	Se desarrolló el segundo algoritmo de colisiones. Aunque la tarea no estaba prevista, se creó para solucionar el problema que ocasionaba el algoritmo anterior.
7	15/01/2014 – 16/01/2014	Gestor de entrada	Se creó el sistema de entrada para reducir las comprobaciones a la mitad.
8	17/01/2014 – 17/01/2014	Cargar los niveles de archivos	Se permitió poder diseñar los niveles en archivos, y así poder cargar niveles distintos sin cambiar el código.
9	17/01/2014 – 17/01/2014	Más menús	El menú inicial tenía la opción de jugar y la de salir. Se añadieron 2 menús para poder elegir entre varios niveles, uno para la historia y otro para los retos.
10	18/01/2014 – 18/01/2014	Cámara	Se creó una cámara para que los niveles pudieran ser más grandes que la pantalla.
11	12/02/2014 – 23/02/2014	Colisiones – Comprobación por losas	Se volvió a cambiar el algoritmo de colisiones.

12	24/02/2014 – 27/02/2014	Añadir nuevas entidades	Se añadieron la salida, objetos de puntuación y objetos que quitan vida. Dependiendo de los parámetros del constructor, SimpleTile crea bloques o plataformas.
13	28/02/2014 – 28/02/2014	Pausar el videojuego	Se permitió que el nivel pudiera tener estados, que serían el estado de jugando y el de pausa.
14	02/03/2014 – 04/03/2014	Almacén de recursos	Se crea un almacén de recursos, para cargar todos los recursos una sola vez, y que las demás clases puedan usarlos sin tener que volver a cargarlos.
15	05/03/2014 – 11/04/2014	Colisiones – Comprobación por losas mejorada	El algoritmo final para la gestión de colisiones ha sido la parte del desarrollo a la que más tiempo se ha dedicado, principalmente por la infinidad de errores que surgieron durante su desarrollo.
16	12/04/2014 – 12/04/2014	Añadir pinchos	Se crearon los pinchos, un objeto que hace daño y además se mueve en el eje vertical.
17	02/06/2014 – 07/06/2014	Creación de entidades con patrones	Se aplican los patrones Builder y Factory Method para crear las entidades de los niveles.
18	08/06/2014 – 09/06/2014	Colisiones con objetos móviles	Permite al algoritmo de colisiones tener en cuenta también los objetos móviles.
19	07/08/2014 – 07/08/2014	Cámara en menú	Agrega una cámara al menú de retos.
20	09/08/2014 – 09/08/2014	Guardar el progreso	Guarda el progreso del jugador en un archivo XML.
21	09/08/2014 – 09/08/2014	Transiciones	Crea estados de transición.
22	09/08/2014 – 09/08/2014	Cargar configuración de entrada	Permite cargar la configuración de entrada de un archivo XML.
23	09/08/2014 – 10/08/2014	Sonidos	Agrega sonidos al videojuego.
24	10/08/2014 – 10/08/2014	HUD	Muestra el HUD del videojuego con imágenes, en lugar de con simple texto.
25	10/08/2014 – 10/08/2014	Añadir llaves y puertas	Agrega llaves y puertas como nuevas entidades.
26	11/08/2014 – 11/08/2014	Pantalla completa	Permite poner el videojuego en pantalla completa.

27	11/08/2014 – 11/08/2014	Reiniciar progreso	Permite reiniciar el progreso del jugador.
28	12/08/2014 – 12/08/2014	Habilidad especial “Boost”	Añade una habilidad especial al personaje, que le permite moverse más rápido y sin gravedad.

Figura 65. Cronología del desarrollo

Durante el desarrollo de una tarea han podido descubrirse y solucionarse errores correspondientes a otras tareas, que no se han añadido a la cronología. Tampoco se han añadido cambios menores como cambiar las imágenes del videojuego, cambiar colores o ajustar valores.

En Enero de 2014 se comprobó que MonoGame funcionaba exactamente igual que XNA Framework, pero hasta Junio de 2014 se siguió trabajando con XNA Framework.

3.6. Herramientas

Tanto para desarrollar el videojuego, como para redactor la memoria, se ha hecho uso se varias herramientas:

- El **lenguaje de programación** que se ha utilizado ha sido C#, un lenguaje de programación orientado a objetos desarrollado por Microsoft.
- Se han utilizado 2 **entornos de desarrollo integrado (IDE)**: Microsoft Visual Studio 2010 y 2013. Como inicialmente se empezó a desarrollar el videojuego con XNA Framework, y este no es compatible con Microsoft Visual Studio 2012 ni 2013, se utilizó Microsoft Visual Studio 2010. Una vez que se pasó el proyecto a MonoGame, se utilizó Microsoft Visual Studio 2013.
- Se han utilizado 2 **frameworks**: XNA Framework 4.0 y MonoGame 3.0.1. Inicialmente se empezó a desarrollar con XNA Framework, y el proyecto estaba bastante avanzado, se comprobó que con el mismo código funcionaba igual en XNA Framework y en MonoGame, por lo que se continuó el desarrollo con MonoGame.
- Como **sistema de control de versiones** se ha utilizado Git. Mientras se desarrollaba con XNA Framework, el repositorio remoto de Git estaba alojado en una Raspberry Pi, y al crear el proyecto de MonoGame (Junio de 2014), se creó un nuevo repositorio, esta vez en GitHub, disponible en: https://github.com/avilin/Booster.
- Para la parte de **ofimática** se han utilizado varias aplicaciones de Microsoft Office: Word para redactar la memoria, Excel para generar gráficas, y PowerPoint para las diapositivas de la defensa del Trabajo de Fin de Grado.
- La memoria incluye **diagramas** como documentación, que se han creado con Microsoft Visual Studio 2013 y con Visual Paradigm 10.0.
- Para **crear los escenarios** se ha utilizado Tiled, un editor de mapas gratuito.
- Para **compilar el contenido** se utilizó inicialmente un proyecto de XNA, hasta que se descubrió XNAFormatter, una pequeña aplicación con la única funcionalidad de convertir el contenido a archivos .xnb.

4. Resultados

En el apartado "3.4. Diseño e implementación" se ha explicado el videojuego por partes, pero no se ha dado una visión del videojuego completo. El diagrama de estados de la *Figura 66* une todos los diagramas de estados del apartado "3.4. Diseño e implementación", para mostrar todas las transiciones entre estados del videojuego.

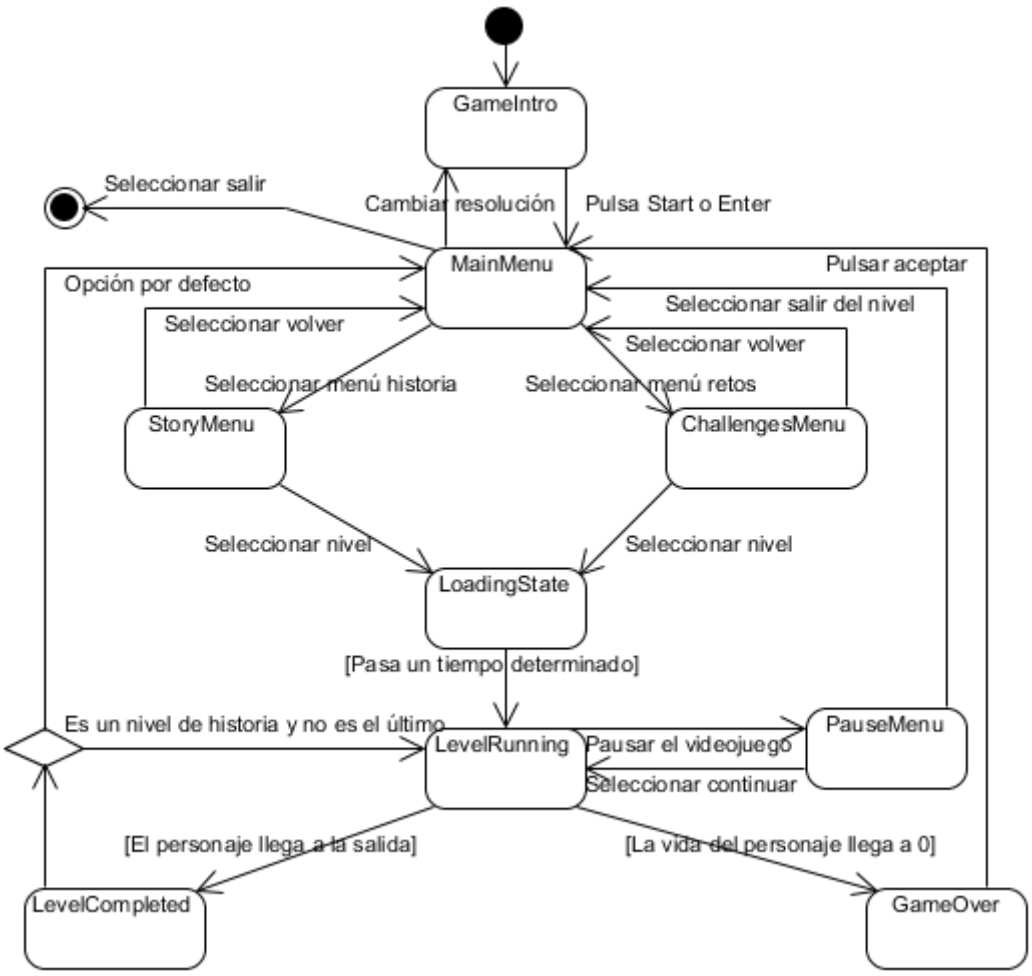


Figura 66. Diagrama de estados completo

Debido a que el autor sólo tiene competencias de ingeniero informático, y no tiene unas notables cualidades de artista, hasta ahora sólo se ha explicado el diseño e implementación del software. Aunque el autor ha creado los fondos de los estados, el resto del arte del videojuego

no lo ha creado él. Los sprites para las entidades del nivel provienen de un paquete de sprites, con licencia Creative Commons, creados por Kenney [57]. Los efectos de sonido se han generado con la aplicación web as3sfxr [58]. La música de fondo se ha descargado de Sound Image [59], que tiene una gran variedad de música y efectos de sonido con licencia Creative Commons.

Se ha creado un proyecto de MonoGame para Windows con OpenGL, y para que el videojuego funcione, es necesario tener instalado en el ordenador la librería OpenAL. Para evitar que el usuario tenga que buscar la librería e instalarla, se ha creado un instalador, que instala esta librería durante el proceso de instalación del videojuego. Para crear un instalador, se ha utilizado un proyecto Setup Wizard, que se ha tenido que crear con Visual Studio 2010, ya que es la última versión de Visual Studio que permite crear estos proyectos.

A continuación se muestran capturas de las distintas pantallas o estados del videojuego.

4.1. Transiciones

Como ya se ha comentado en el capítulo "3. Booster", los estados de transición son los estados más simples, simplemente tienen una imagen con texto. La *Figura 67* muestra las capturas de los estados de transición. La captura de la esquina superior izquierda corresponde al estado GameIntro. La captura de la esquina superior derecha corresponde al estado LoadingState. La captura de la esquina inferior izquierda corresponde al estado GameOver. La captura de la esquina inferior derecha corresponde al estado LevelCompleted.

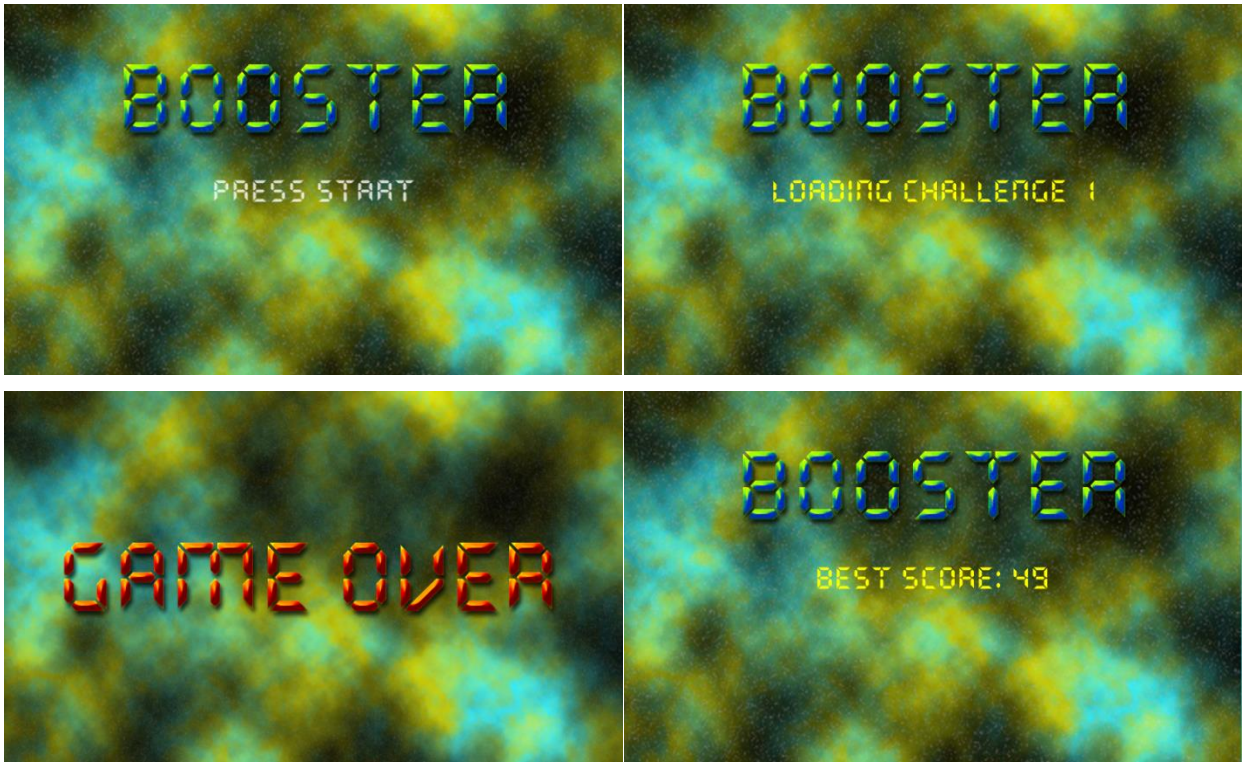


Figura 67. Capturas de los estados de transición

4.2. Menús

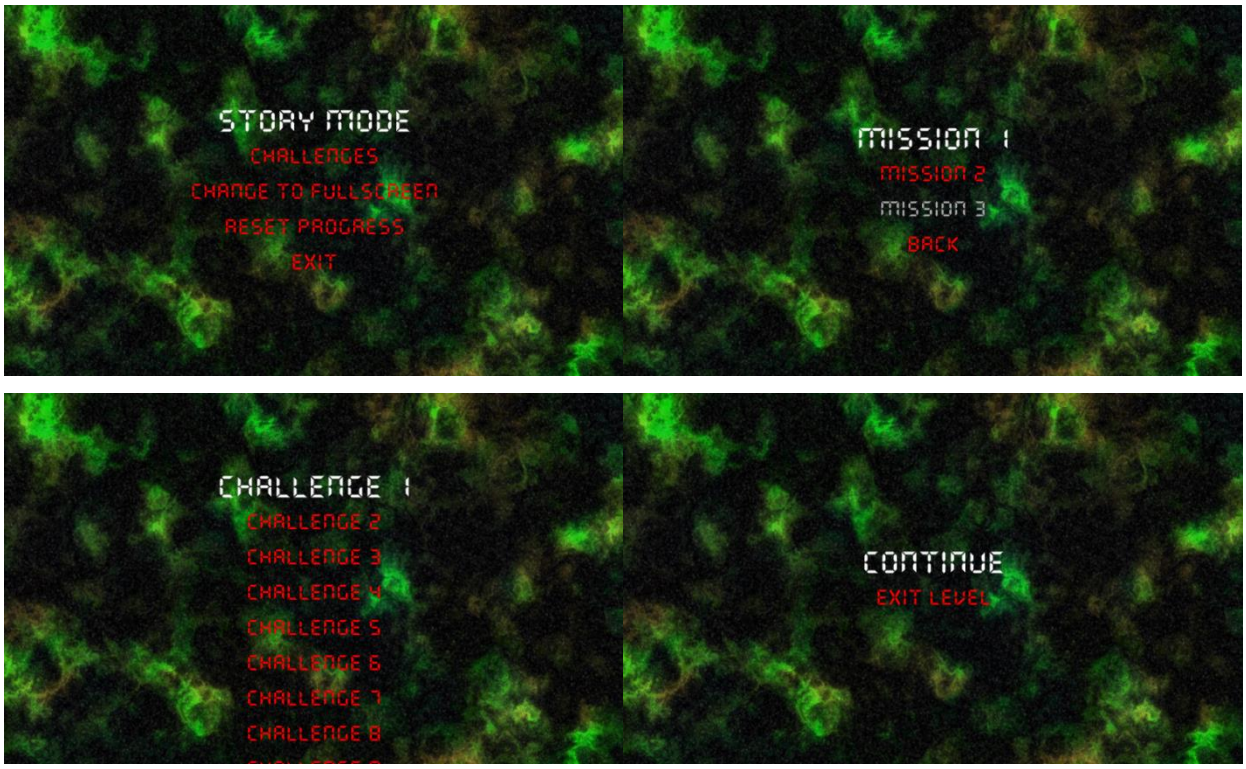


Figura 68. Capturas de los menús

Como se ha explicado en el capítulo "3. Booster", el videojuego tiene 4 menús, mostrados en la *Figura 68*. El menú de la esquina inferior izquierda es el menú de los retos, y es el único menú con desplazamiento, los demás son menús estáticos. Se puede apreciar que uno de los elementos del menús sale cortado, y conforme se baja en las opciones del menú, la cámara se moverá y el elemento cortado, y los posteriores, irán apareciendo en pantalla. En todos los menús se puede apreciar que el elemento seleccionado está en un color distinto y más grande que los otros elementos. Además, en el menú de la historia (esquina superior derecha de la figura), aparece un elemento en gris. Esto significa que ese elemento está bloqueado y no se puede seleccionar. Ya se había comentado que no todos los niveles de la historia están disponibles inicialmente, en este caso, este elemento aparecerá en gris hasta que se complete el nivel "Mission 2". El menú de la esquina superior izquierda es el menú principal, y el de la esquina inferior derecha es el menú de pausa.

4.3. Nivel

Figura 69. Captura de un nivel

En el capítulo "3. Booster", se ha hablado del nivel y sus entidades, pero lo único que se sabía de las entidades era su comportamiento, y la primera vez que entráramos a un nivel, posiblemente no sabríamos a qué identidad corresponden las imágenes que vemos en el nivel.

Como se puede ver en la *Figura* 69, en la esquina superior derecha de la pantalla está el HUD. El HUD está compuesto por:

- **Corazones**: representan las vidas del personaje. El jugador tiene 4 vidas, representadas por 2 corazones. Para representar las vidas impares, se muestra medio corazón.
- **Barra de la habilidad especial "Boost"**: está situada al lado de los corazones. Esta habilidad se ha nombrado "Boost", de ahí el nombre del videojuego, y será esencial aprender bien su funcionamiento para completar los niveles. La barra está en verde hasta que se descarga la habilidad, entonces será roja hasta que vuelva a estar cargada otra vez.
- **Monedas**: representan la puntuación obtenida durante el nivel.
- **Llaves**: representa las llaves que ha cogido el personaje y aún no ha utilizado.

A continuación se muestra a qué entidad corresponden las imágenes que se ven en el nivel:

El personaje (la entidad Player) es un alienígena, y tiene varias animaciones (quieto, andando, en el aire, herido y debilitado). Se mueve con las teclas de dirección o el control de dirección de un gamepad. Salta con la barra espaciadora o con el botón "A" del mando. Se utiliza la habilidad especial "Boost" con el "1" o con el botón "B" del mando.

Los bloques son una representación de la entidad SimpleTile. Pueden representarse con imágenes distintas, para personalizar más el aspecto del nivel. Los bloques no se pueden atravesar.

Las plataformas son otra representación de la entidad SimpleTile. La diferencia con respecto a los bloques, además de la imagen que los representa, es que se pueden atravesar en todas direcciones excepto por arriba.

La entidad DamageObject tiene 3 representaciones posibles, y las 2 bloquean el paso al personaje. Las cajas con un símbolo de advertencia sólo quitan 1 vida (medio corazón) al colisionar con ellas.

La siguiente representación de DamageObject es un bloque verde con cara de cabreado. Aunque parezca estar vivo y no ser un simple bloque, no se ha considerado un enemigo porque es como un DamageObject, y un enemigo podría moverse. Este bloque quita 2 vidas (1 corazón).

La última representación de DamageObject es la lava. La lava no bloquea, pero es la entidad más peligrosa del nivel, ya que quita 4 vidas (2 corazones), por lo que, tenga las vidas que tenga el personaje, al colisionar con la lava, morirá.

Hay un tipo más de DamageObject, que pertenece a la subclase Spike. Los pinchos es el único DamageObject que se mueve, y que se puede atravesar.

Hay 3 representaciones de la entidad ScoreObject, los objetos que dan puntos. La imagen que representa a estos objetos es una moneda, que puede ser de cobre (1 punto), de plata (2 puntos), o de oro (3 puntos).

Las cerraduras en el nivel representan la entidad Door, es decir, una puerta. Las puertas, al igual que los bloques, no se pueden atravesar y bloquean el paso. La diferencia es que, si el personaje tiene una llave, al colisionar con la puerta, esta desaparece, y el personaje pierde la llave.

La llave representa la entidad Key. El personaje coge llaves, y las utiliza para abrir las puertas. Cuando el personaje abre una puerta, pierde una llave. Si no se tiene al menos una llave, el personaje no podrá abrir ninguna puerta.

Por último, el cartel de salida representa la entidad ExitEntity. El personaje debe llegar hasta la salida para completar el nivel.

Factores como tener que conseguir llaves para abrir puertas, o tener que utilizar la habilidad especial “Boost” para resolver situaciones que de otra forma impedirían al jugador completar el nivel, hacen que dentro de los videojuegos de plataformas, Booster sea un videojuego de plataformas con puzles.

5. Conclusiones

5.1. Conclusiones

En este Trabajo de Fin de Grado se ha cumplido una meta personal, desarrollar un videojuego de plataformas en 2D, **Booster**, representado en la *Figura 70*, haciendo uso de los conocimientos adquiridos en el Grado de Ingeniería Informática. Booster se ha desarrollado con C# y el framework MonoGame. Se ha utilizado una metodología orientada a objetos con patrones de diseño, aplicando un enfoque iterativo e incremental. En este videojuego, el autor ha implementado su propio algoritmo de gestión de colisiones.

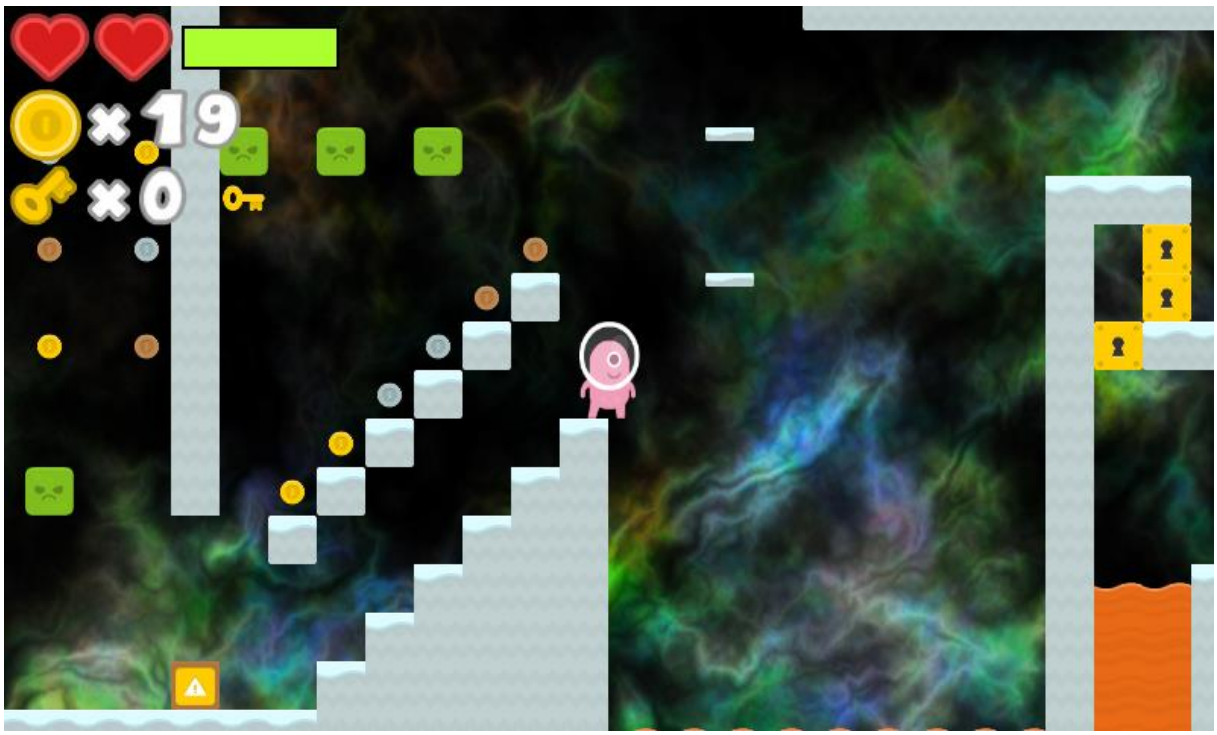


Figura 70. Booster: un videojuego de plataformas en 2D.

El desarrollo de este proyecto ha supuesto varias aportaciones al autor:

- Conocimientos sobre el desarrollo de videojuegos en general.

- Una visión general de la arquitectura y funcionamiento de un motor de juegos.
- Conocimientos sobre el funcionamiento de XNA Framework y MonoGame, y cómo utilizar estos frameworks para desarrollar videojuegos.
- Conocimientos sobre el uso del sistema de control de versiones Git.
- Aplicación de conocimientos adquiridos durante el Grado en Ingeniería Informática, haciendo especial hincapié en el uso de patrones de diseño para el desarrollo del proyecto.
- Aprender las ventajas e inconvenientes de utilizar un motor de físicas en un videojuego de plataformas en 2D.
- Aprender a estructurar la documentación de un proyecto y a redactar de una manera más formal.

5.2. Trabajo futuro

Como se ha comentado en el apartado "3.3. Análisis", inicialmente la idea del videojuego a desarrollar era muy ambiciosa, que se acotó en unas características u objetivos básicos, que se han cumplido. También se han incluido nuevas características al videojuego, y todavía se podría ampliar más. Antes de comentar posibles ampliaciones, se comentarán posibles mejoras al videojuego actual.

- Cabe destacar que el autor no posee grandes dotes artísticas, por lo que el contenido, proveniente de diversas fuentes, no combina demasiado bien. Los fondos tiene un estilo distinto a los sprites de las entidades, y tal vez la música de fondo escogida no haya sido la mejor elección. Así que habría que aprender a crear un buen contenido, o conseguir que algún experto creara el contenido para el videojuego.
- Guardar las puntuaciones y la información de los niveles en una base de datos, para dar la posibilidad de que las puntuaciones de todos los jugadores estén en una base de datos común, y puedan comparar sus puntuaciones con las de otros jugadores.
- Actualmente sólo se ve la máxima puntuación al completar el nivel, pero se podría cambiar a que apareciera una lista con las puntuaciones ordenadas de mayor a menor, indicando también la fecha de cada puntuación.
- Se podrían buscar más alternativas para la gestión de colisiones, ya que el algoritmo actual es bastante complicado y no muy extensible.
- Actualmente las únicas opciones de configuración son poner el videojuego en pantalla completa y reiniciar el progreso del jugador. Estas opciones se podrían introducir dentro de un menú de opciones, y añadir opciones para cambiar el volumen de la música y para cambiar la configuración de teclado y gamepad desde el videojuego (actualmente hay que modificar el archivo XML a mano).
- Cuando se empezó a investigar MonoGame (Enero de 2014), la última versión estable era la versión 3.0.1, y es la versión que se utilizó para desarrollar el videojuego. Ahora (Agosto de 2014), la última versión estable es la 3.2, por lo que se podría estudiar las mejoras que ofrece esta versión y substituirla por la actual, o actualizando la versión del proyecto actual si es posible, o creando un nuevo proyecto con la nueva versión y copiando el contenido del proyecto.
- Por último, y relacionado con el punto anterior, comprobar si MonoGame 3.2 soluciona un error de la librería OpenTK, que no permite cambiar el tamaño de la ventana. Debido a este error, el videojuego se puede poner en pantalla completa, pero no permite desactivarla.

Posiblemente se podría ampliar infinitamente, pero sólo se van a exponer unas pocas ampliaciones:

- Una de las ampliaciones más sencilla es simplemente crear nuevos niveles.
- Añadir enemigos con su propia estrategia de movimiento.
- Añadir lanzamisiles, un elemento más que puede quitarle vida al personaje con misiles.
- Añadir armas u otras habilidades especiales al personaje, para que pueda enfrentarse a sus enemigos.
- Poner varias condiciones de victoria a los niveles, en especial a los retos, como poner un tiempo límite, completar el nivel con la vida entera, etc.
- Permitir realizar un doble salto al personaje.
- Añadir power-ups para el personaje (saltar más alto, inmune durante un tiempo, etc).
- Añadir soporte para un segundo jugador, para que se pueda competir por qué jugador completa antes el nivel.
- Añadir soporte para control por pantalla táctil, para poder portar el videojuego a Windows Phone 8.1, Android y/o iOS.
- Añadir un editor de niveles propio.

Booster es un videojuego de plataformas con puzles, y aprovechando algunas ampliaciones se podría cambiar el género del videojuego:

- Añadiendo enemigos, armas, la posibilidad de disparar en todas direcciones y quitando las monedas y las puertas, nos encontraríamos con un videojuego *run and gun*.
- Con niveles más grandes, en los que se permita explorar, y no simplemente tener que llegar al final del nivel, nos encontraríamos con un videojuego de acción-aventura.

6. Bibliografía

6.1. Referencias

[1] *Definición de videojuego*. Disponible en: http://definicion.de/videojuego/. Última consulta: 17 de Junio de 2014.

[2] *Videojuego*. Disponible en: http://es.wikipedia.org/wiki/Videojuego. Última fecha de consulta: 17 de Junio de 2014.

[3] *Controlador de videojuegos*. Disponible en: http://en.wikipedia.org/wiki/Game_controller. Última fecha de consulta: 18 de Junio de 2014.

[4] *Gamepad*. Disponible en: http://en.wikipedia.org/wiki/Gamepad. Última fecha de consulta: 18 de Junio de 2014.

[5] *Gamepad de la videoconsola Xbox 360*. Disponible en: http://en.wikipedia.org/wiki/Xbox_360_Controller. Última fecha de consulta: 18 de Junio de 2014.

[6] *PlayStation Move*. Disponible en: http://en.wikipedia.org/wiki/PlayStation_Move. Última fecha de consulta: 18 de Junio de 2014.

[7] *Kinect*. Disponible en: http://en.wikipedia.org/wiki/Kinect. Última fecha de consulta: 18 de Junio de 2014.

[8] *Videojuego*. Disponible en: http://en.wikipedia.org/wiki/Video_game. Última fecha de consulta: 26 de Julio de 2014.

[9] *Videoconsola*. Disponible en: http://es.wikipedia.org/wiki/Videoconsola. Última fecha de consulta: 27 de Junio de 2014.

[10] *Arcade*. Disponible en: http://es.wikipedia.org/wiki/Arcade. Última fecha de consulta: 27 de Junio de 2014.

[11] *Géneros de videojuegos*. Disponible en: http://en.wikipedia.org/wiki/Video_game_genres. Última fecha de consulta: 28 de Junio de 2014.

[12] *Gameplay*. Disponible en: http://www.techopedia.com/definition/1911/gameplay. Última fecha de consulta: 28 de Junio de 2014.

[13] *Juegos de acción*. Disponible en: http://en.wikipedia.org/wiki/Action_game. Última fecha de consulta: 18 de Julio de 2014.

[14] *Beat 'em up*. Disponible en: http://en.wikipedia.org/wiki/Beat_%27em_up. Última fecha de consulta: 12 de Julio de 2014.

[15] *Juegos de disparos*. Disponible en: http://en.wikipedia.org/wiki/Shooter_game. Última fecha de consulta: 17 de Julio de 2014.

[16] *Juegos de lucha*. Disponible en: http://en.wikipedia.org/wiki/Fighting_game. Última fecha de consulta: 12 de Julio de 2014.
[17] *Juegos de plataformas*. Disponible en: http://en.wikipedia.org/wiki/Platform_game. Última fecha de consulta: 13 de Julio de 2014.
[18] *Juegos de aventura*. Disponible en: http://en.wikipedia.org/wiki/Adventure_game. Última fecha de consulta: 22 de Julio de 2014.
[19] *Novelas visuales*. Disponible en: http://en.wikipedia.org/wiki/Visual_novel. Última fecha de consulta: 22 de Julio de 2014.
[20] *Juegos de acción-aventura*. Disponible en: http://en.wikipedia.org/wiki/Action-adventure_game. Última fecha de consulta: 22 de Julio de 2014.
[21] *Juegos de sigilo*. Disponible en: http://en.wikipedia.org/wiki/Stealth_game. Última fecha de consulta: 22 de Julio de 2014.
[22] *Survival horror*. Disponible en: http://en.wikipedia.org/wiki/Survival_horror_game. Última fecha de consulta: 22 de Julio de 2014.
[23] *Juegos de estrategia*. Disponible en: http://en.wikipedia.org/wiki/Strategy_video_game. Última fecha de consulta: 23 de Julio de 2014.
[24] *Juegos 4X*. Disponible en: http://en.wikipedia.org/wiki/4X_game. Última fecha de consulta: 23 de Julio de 2014.
[25] *Juegos de artillería*. Disponible en: http://en.wikipedia.org/wiki/Artillery_game. Última fecha de consulta: 23 de Julio de 2014.
[26] *Juegos de estrategia en tiempo real*. Disponible en: http://en.wikipedia.org/wiki/Real-time_strategy. Última fecha de consulta: 23 de Julio de 2014.
[27] *Tower defense*. Disponible en: http://en.wikipedia.org/wiki/Tower_defense. Última fecha de consulta: 23 de Julio de 2014.
[28] *Juegos de estrategia por turnos*. Disponible en: http://es.wikipedia.org/wiki/Videojuego_de_estrategia_por_turnos. Última fecha de consulta: 23 de Julio de 2014.
[29] *Juegos de tácticas en tiempo real*. Disponible en: http://en.wikipedia.org/wiki/Real-time_tactics. Última fecha de consulta: 23 de Julio de 2014.
[30] *Juegos de tácticas por turnos*. Disponible en: http://en.wikipedia.org/wiki/Turn-based_tactics. Última fecha de consulta: 23 de Julio de 2014.
[31] *Juegos de rol*. Disponible en: http://en.wikipedia.org/wiki/Role-playing_video_game. Última fecha de consulta: 25 de Julio de 2014.
[32] *Roguelike*. Disponible en: http://en.wikipedia.org/wiki/Roguelike. Última fecha de consulta: 25 de Julio de 2014.
[33] *Videojuegos de simulación*. Disponible en: http://en.wikipedia.org/wiki/Simulation_video_game. Última fecha de consulta: 25 de Julio de 2014.
[34] *Videojuegos de construcción y gestión*. Disponible en: http://en.wikipedia.org/wiki/Construction_and_management_simulation. Última fecha de consulta: 25 de Julio de 2014.
[35] *Videojuegos de deportes*. Disponible en: http://en.wikipedia.org/wiki/Sports_game. Última fecha de consulta: 25 de Julio de 2014.
[36] *Videojuegos de vehículos*. Disponible en: http://en.wikipedia.org/wiki/Vehicle_simulation_game. Última fecha de consulta: 25 de Julio de 2014.
[37] *Videojuegos de simulación de vida*. Disponible en: http://en.wikipedia.org/wiki/Life_simulation_game. Última fecha de consulta: 25 de Julio de 2014.

[38] *Géneros de videojuegos*. Disponible en: http://en.wikipedia.org/wiki/Video_game_genres. Última fecha de consulta: 26 de Julio de 2014.
[39] *Hotseat*. Disponible en: http://en.wikipedia.org/wiki/Hotseat_(multiplayer_mode). Última fecha de consulta: 26 de Julio de 2014.
[40] *Efectos de los videojuegos*. Disponible en: http://en.wikipedia.org/wiki/Video_game_controversies. Última fecha de consulta: 26 de Julio de 2014.
[41] Jason Gregory. *Game Engine Architecture*. A. K. Peters / CRC Press, 2009.
[42] *Desarrollo de videojuegos*. Disponible en: http://en.wikipedia.org/wiki/Video_game_development. Última fecha de consulta: 28 de Julio de 2014.
[43] *Arte del videojuego*. Disponible en: http://en.wikipedia.org/wiki/Game_art_design. Última fecha de consulta: 28 de Julio de 2014.
[44] *Web de Unity*. Disponible en: http://unity3d.com/. Última fecha de consulta: 5 de Agosto de 2014.
[45] *Web de LibGDX*. Disponible en: http://libgdx.badlogicgames.com/. Última fecha de consulta: 6 de Agosto de 2014.
[46] *GitHub de LibGDX*. Disponible en: https://github.com/libgdx/libgdx/wiki. Última fecha de consulta: 6 de Agosto de 2014.
[47] DEV, Asociación Española de Empresas Productoras y Desarrolladoras de Videojuegos y Software de Entretenimiento. *Libro Blanco del Desarrollo Español de los Videojuegos*. 2014.
[48] *XNA Framework*. Disponible en: http://en.wikipedia.org/wiki/Microsoft_XNA. Última fecha de consulta: 10 de Agosto de 2014.
[49] *Github de MonoGame*. Disponible en: https://github.com/mono/MonoGame/wiki. Última fecha de consulta: 10 de Agosto de 2014.
[50] *Microsoft deja de dar soporte a XNA*. Disponible en: http://www.gamasutra.com/view/news/185894/Its_official_XNA_is_dead.php. Última fecha de consulta: 10 de Agosto de 2014.
[51] Aaron Reed. *Learning XNA 4.0*. O'Reilly, 2010.
[52] Dean Johnson, Tom Miller. *XNA Game Studio 4.0 Programming: Developing for Windows*. Addison-Wesley Professional, 2010.
[53] Laurent Debrauwer. *Patrones de diseño para C#*. Ediciones ENI.
[54] Judith Bishop. *C# 3.0 Design Patterns*. O'Reilly, 2007.
[55] *Cámara para videojuego en 2D*. Disponible en: http://adambruenderman.wordpress.com/2011/04/05/create-a-2d-camera-in-xna-gs-4-0/. Última fecha de consulta: 26 de Agosto de 2014.
[56] *Farseer Physics Engine*. Disponible en: http://farseerphysics.codeplex.com/documentation. Última fecha de consulta: 29 de Agosto de 2014.
[57] *Fuente de las imágenes*. Disponible en: www.kenney.nl. Última fecha de consulta: 29 de Agosto de 2014.
[58] *Generador de sonidos*. Disponible en: http://www.superflashbros.net/as3sfxr/. Última fecha de consulta 29 de Agosto de 2014.
[59] *Fuente de la música*. Disponible en: http://soundimage.org/. Última fecha de consulta: 29 de Agosto de 2014.