



**UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA**

GRADO EN INGENIERÍA INFORMÁTICA

**TECNOLOGÍA ESPECÍFICA DE
COMPUTACIÓN**

TRABAJO FIN DE GRADO

**Virus Breaker
Desarrollo de un videojuego con Unity3D**

Pedro Romero González

Febrero, 2018

VIRUS BREAKER
DESARROLLO DE UN VIDEOJUEGO CON UNITY3D



**UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA**

Tecnologías y Sistemas de Información

**TECNOLOGÍA ESPECÍFICA DE
COMPUTACIÓN**

TRABAJO FIN DE GRADO

**Virus Breaker
Desarrollo de un videojuego con Unity3D**

Autor: Pedro Romero González

Director: Dr. David Vallejo Fernández

Febrero, 2018

Pedro Romero González

Ciudad Real – España

E-mail: pedro9romero4gonzalez@gmail.com

Teléfono: 689 426 432

Web site: <https://gamejolt.com/@nexus64>

© 2018 Pedro Romero González

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Se permite la copia, distribución y/o modificación de este documento bajo los términos de la Licencia de Documentación Libre GNU, versión 1.3 o cualquier versión posterior publicada por la *Free Software Foundation*; sin secciones invariantes. Una copia de esta licencia esta incluida en el apéndice titulado «GNU Free Documentation License».

Muchos de los nombres usados por las compañías para diferenciar sus productos y servicios son reclamados como marcas registradas. Allí donde estos nombres aparezcan en este documento, y cuando el autor haya sido informado de esas marcas registradas, los nombres estarán escritos en mayúsculas o como nombres propios.

TRIBUNAL:

Presidente:

Vocal:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE

VOCAL

SECRETARIO

Fdo.:

Fdo.:

Fdo.:

Índice general

Índice general	v
Índice de figuras	ix
1. Introducción	1
1.1. Contexto	1
1.2. Motivación	1
1.3. Descripción del Proyecto	2
2. Estado del Arte	5
2.1. El mercado de los videojuegos	5
2.1.1. Industria Mundial del Videojuego	5
2.1.2. La industria de los videojuegos en España	6
2.1.3. Retos y tendencias actuales	8
2.2. El proceso de desarrollo de videojuegos	15
2.2.1. Etapas del desarrollo	15
2.2.2. Estructura típica de un equipo de desarrollo	19
2.3. Motores de juegos	25
2.3.1. Descripción	25
2.3.2. Ejemplos de Motores	27
2.3.3. Unity 3D	31
2.4. Inteligencia Artificial en Videojuegos	36
2.4.1. Historia	36
2.4.2. IA Aplicada a Videojuegos: Contexto Actual	37
2.4.3. Métodos de la IA	39
2.4.4. Futuros campos de aplicación	44
3. Arquitectura	47
3.1. Descripción del Juego	47
3.2. Visión General de la Arquitectura	49

0. ÍNDICE GENERAL

3.3. Menú	50
3.4. Control de niveles	53
3.4.1. Carga de Niveles	54
3.4.2. Inicio del juego	60
3.4.3. Condiciones de fin del juego	60
3.5. Control del jugador	62
3.5.1. Componentes	63
3.5.2. Comportamiento del personaje	64
3.6. Físicas e interacción	67
3.6.1. Pelota	67
3.6.2. Elementos Interactivos	71
3.7. Jefe Final	77
3.7.1. Componentes	78
3.7.2. Movimiento	79
3.7.3. Inteligencia Artificial	80
3.8. Patrones de Diseño	82
3.8.1. State	83
3.8.2. Component	83
3.8.3. Prototype	84
3.8.4. Singleton	85
4. Resultados	87
4.1. Proceso de Desarrollo	87
4.1.1. Prototipo	87
4.1.2. Mecánicas de Juego Complementarias	89
4.1.3. Cambios Estéticos	91
4.1.4. Carga de Niveles	93
4.1.5. Jefe Final	95
4.1.6. Ajustes, Refactorización y Correcciones	96
4.2. Encuesta de Opinión	98
4.2.1. Descripción	98
4.2.2. Resultados	99
5. Conclusiones y Trabajos Futuros	105
5.1. Conclusiones	105
5.1.1. Grado de Cumplimiento de los Objetivos	105

5.1.2. Grado de Cumplimiento de Competencias	106
5.1.3. Valoración Personal	107
5.2. Líneas de Trabajo Futuro	108
Bibliografía	111
Videojuegos	112

Índice de figuras

2.1. Crecimiento mundial de la industria del videojuego (por plataformas)	6
2.2. Distribución por países del mercado del videojuego.	7
2.3. Mercado del videojuego en Europa.	7
2.4. Distribución de las empresas en España por número de empleados.	8
2.5. Distribución de las empresas en España por número de empleados.	8
2.6. Fotografía del torneo Intel® Extreme Masters en Katowice, Polonia (Fuente: intelectremasters.com)	9
2.7. Crecimiento del mercado del eSport (Fuente: [DEV17])	10
2.8. <i>League of Legends</i> (Riot Games, 2009), uno de los eSports más populares (Fuente: mobygames.com).	11
2.9. Previsiones de crecimiento del mercado de Realidad Virtual y Aumentada (tabla extraída de [DEV17]) (miles de millones de dólares).	12
2.10. De izquierda a derecha: HTC Vive, Oculus Rift, Samsung VR y PlayStation VR (imágenes tomadas de las páginas oficiales de los productos).	13
2.11. Impresora 3d “replicator” de la compañía Makerbot.	15
2.12. Etapas del desarrollo de un videojuego (Imagen tomada del [Dav15]).	16
2.13. Distribución de roles en un equipo de desarrollo.	20
2.14. <i>Heretic</i> (Raven Software, 1994), es un juego desarrollado con el motor de Doom (Fuente: doom.wikia.com).	25
2.15. Unreal (Epic Games, 1998) (Fuente: mobygames.com).	28
2.16. Captura del entorno de Unreal Engine (Fuente: docs.unrealengine.com).	29
2.17. Captura del entorno de Game Maker Studio (Fuente: pcgamesn.com). .	30
2.18. <i>Undertale</i> (Toby Fox, 2015) (Fuente: Página de Steam del juego).	30
2.19. Captura del entorno de desarrollo de Unity (Fuente: answers.unity.com).	32
2.20. <i>Pac-Man 256</i> (Hipster Whale, 2015), disponible para PC, Android, IOS, PS4 y XBOX One (Fuente: página de Steam del juego).	35
2.21. El Gran Maestro de ajedrez Garri Kaspárov (izquierda) enfrentándose al ordenador Deep Blue (derecha).	37
2.22. En <i>The Sims</i> (Maxis, 2000), los personajes pueden tomar decisiones basándose en sus gustos y necesidades.	38

0. ÍNDICE DE FIGURAS

2.23. <i>Minecraft</i> (Mojang, 2011) es un ejemplo claro de generación procedimental de terrenos.	39
2.24. FSM de alto nivel de una IA jugadora de <i>Pac-Man</i> (Namco, 1980) (figura extraída de [YT18]).	40
2.25. Ejemplo de comportamiento tóxico en <i>League of Legends</i> (Riot Games, 2009).	45
3.1. Captura de pantalla de Virus Breaker.	48
3.2. Pantallas de victoria y derrota.	48
3.3. Diagrama de clases.	50
3.4. Diagrama de escenas.	51
3.5. Diagrama del grupo funcional Menu	52
3.6. Componentes de la cámara.	52
3.7. Diagrama de componentes de la sala.	54
3.8. Diagrama del grupo funcional Level Control	55
3.9. Listado de ladrillos prefabricados y sus colores asociados.	56
3.10. Diagrama del proceso de generación de niveles.	58
3.11. Homenaje a <i>Space Invaders</i> (Taito, 1978) dentro de <i>Arkanoid: Doh It Again</i> (Taito Corporation, 1997).	59
3.12. Texto tutorial.	61
3.13. Modelo del personaje principal.	62
3.14. Diagrama de componentes del personaje principal.	63
3.15. Personaje principal activando la paleta.	64
3.16. Diagrama del grupo funcional Player Input	65
3.17. Diagrama de estados del personaje principal.	66
3.18. Diagrama del grupo funcional Collision Interaction	67
3.19. Pelota moviéndose.	68
3.20. Componentes del objeto Ball.	69
3.21. ParticleCircle (izquierda) y ExplosionParticle (derecha).	70
3.22. Direcciones que tomaría la pelota dependiendo del punto de la paleta que golpee (Aproximada).	72
3.23. Modelo de la puerta.	73
3.24. Diagrama de componentes de la puerta.	74
3.25. Barra de vida de la vida.	74
3.26. Modelos de los ladrillos.	75
3.27. Componentes del objeto Brick.	75
3.28. Jerarquía de clases de Brick.	76

3.29. Modelo del jefe.	77
3.30. Diagrama de componentes del jefe.	78
3.31. Diagrama del grupo funcional Boss	79
3.32. Diagrama de estados del jefe.	82
3.33. Diagrama de clases del patrón State.	83
3.34. Diagrama de clases del patrón Component.	84
3.35. Diagrama de clases del patrón Prototype.	85
3.36. Diagrama de clases del patrón Singleton.	86
 4.1. Captura del prototipo (Recreación).	88
4.2. Guardar los niveles como objetos prefabricados fue una de las alternativas probadas.	90
4.3. Comparativa entre la textura antigua (izquierda) y moderna (derecha) del suelo de la sala.	91
4.4. Estructura de MainCharacter antes (izquierda) y después (derecha) de la modificación.	92
4.5. Pantalla de título antigua.	93
4.6. Captura de Tiled durante el diseño del nivel seis del juego.	94
4.7. Extracción de la funcionalidad de la sala de LevelGenerator.	95
4.8. Jerarquía de la clase Brick antes (izquierda) y después (derecha) de la refactorización.	97
4.9. Comparativa de la sala antes (izquierda) y después (derecha) de los ajustes gráficos.	97
4.10. Diagramas del género (izquierda) y edad (derecha).	100
4.11. Diagramas de la frecuencia de juego.	101
4.12. Diagrama de la opinión.	101
4.13. Diagramas de la dificultad.	102
4.14. Diagramas de niveles superados.	102

Capítulo 1

Introducción

1.1 Contexto

La industria del videojuego es el **principal mercado del ocio y entretenimiento del mundo**, el cual engloba docenas de disciplinas de trabajo y da trabajo a miles de personas alrededor del mundo. Con unas ventas de 116.000 millones de dólares[DEV17] en el año 2017, este mercado ha superado creces a sus competidores más cercanos: sus ganancias son dos veces y media superiores a las del mercado del cine y más de seis veces las del mercado de la música.

El éxito de esta industria se debe principalmente a su gran **capacidad para evolucionar** rápidamente para adaptarse a los gustos y tendencias de los consumidores. Esto ha permitido que los videojuegos hayan podido integrarse con las principales tendencias en tecnología, como lo fue en su momento la aparición de Internet o de los dispositivos móviles o como actualmente está ocurriendo con la realidad virtual o los servicios de distribución en línea.

La expansión del mercado del videojuego conlleva también una **expansión del perfil de sus jugadores**. La aparición de nuevos modelos de juego, como los juegos casuales, ha permitido llevar el mercado a un público que no probaría otros juegos más tradicionales. Adicionalmente, hay que incluir a los usuarios que utilizan los videojuegos para **crear su propio contenido**, como videos o modificaciones de juegos, o los que disfrutan como espectadores viendo jugar a jugadores profesionales.

En resumen, la industria del videojuego no solo es uno de los **mayores mercados actualmente**, también es un sector económico puntero en constante evolución y la forma de vida de millones de jugadores en todo el mundo.

1.2 Motivación

Los videojuegos, desde un punto formal, son **aplicaciones gráficas en tiempo real e interactivas**[Dav15]. Para que el jugador pueda reaccionar de forma adecuada a los eventos del juego, la aplicación debe de ser capaz de renderizar el entorno del juego con una **frecuencia** lo suficientemente alta. Si no, el jugador solo percibirá una sucesión de imágenes estáticas. El componente gráfico del videojuego está acompañado de muchos

1. INTRODUCCIÓN

otros sistemas que sirven para recrear el entorno del juego con la máxima precisión posible: simulaciones físicas, sistemas de sonido, inteligencias artificiales, etcétera.

Diseñar un software en el que se integren tantos sistemas de áreas diversas como los que incluye un videojuego **no es una tarea trivial**. Es necesario diseñar y planificar los distintos sistemas de forma que no haya conflictos o acoplamientos entre ellos. Además, el programa debe estar optimizado de para obtener una frecuencia de refresco de imagen aceptable. La complejidad de esta tarea es tan elevada que muchos estudios utilizan **motores de juego**, frameworks de desarrollo que suministran gran cantidad de funcionalidad necesaria para el desarrollo de juegos.

Al desarrollo puramente software del videojuego hay que añadir la elaboración de los **recursos audiovisuales** que serán utilizados en él. La elaboración de modelos 3D, texturas, músicas, efectos de sonido, animaciones, cinematográficas y demás recursos son procesos que pueden llegar a ser tan complejos como el propio desarrollo software, y requieren de sus propias **herramientas y tecnologías específicas** que deben ser integradas en el proyecto.

Ante la enorme dificultad para una sola persona para contar con el conocimiento y la experiencia necesarios en todas las áreas involucradas, en el desarrollo de videojuegos suele realizarse mediante **equipos de desarrollo**. Dada la naturaleza interdisciplinar de estos equipos, la coordinación es de vital importancia para evitar malentendidos entre profesionales de distintas áreas que puedan desembocar en retrasos en el proyecto.

Los videojuegos cuentan además con un requisito adicional que la diferencia de otros proyectos software: deben resultar **divertidos para el jugador**. La diversión del jugador es un factor aparte de la usabilidad o la experiencia de usuario que requiere de un buen **diseño de juego** y de realizar **pruebas con jugadores** para poder obtener un resultado óptimo.

1.3 Descripción del Proyecto

El objetivo de este proyecto es el diseño y desarrollo de un **videojuego completo** utilizando el motor de videojuegos Unity3D. El videojuego en cuestión será un juego del género Breakout que se desarrollará en un entorno tridimensional. El producto resultante del proyecto será distribuido a través de Internet mediante una o varias de las plataformas de distribución de juegos.

El juego elegido para ser desarrollado en el proyecto es una **adaptación a las tres dimensiones del género Breakout**, un género que nació con el juego clásico del mismo nombre *Breakout* (Atari Inc., 1976). Se trata de un género tradicionalmente 2D, por lo que en esta versión será necesario introducir cambios a las bases de su

diseño e implementar una serie de sistemas que mejoren la experiencia del jugador, compensando las dificultades intrínsecas a un entorno 3D (zona de juego más grande, dificultad para apreciar perspectiva...).

El desarrollo del proyecto se realizará mediante el uso de **Unity3D**, un **motor de juegos** multiplataformas enfocado al desarrollo de videojuegos en 2D y 3D. Se trata de un motor de calidad profesional, dotado de renderizado en tiempo real, simulación de físicas (implementado mediante la librería **PhysX** de Nvidia), tecnología de animación. La funcionalidad central motor está implementada en el lenguaje de programación C++, mientras que la lógica de juego se implementará mediante el lenguaje C#. El entorno de desarrollo integrado **Microsoft VisualStudio** se utilizará para la escritura de código, el entorno recomendado para Unity; mientras que la integración de recursos se realizará en el **editor de Unity**.

Los componentes gráficos del juego se elaborarán mediante el uso de herramientas externas. Los modelos 3D de personajes y otros elementos se modelarán mediante la herramienta de código libre **Blender**, que permite una exportación de modelos sencilla a los formatos admitidos en Unity. Para la elaboración de las texturas de los modelos, los elementos de la interfaz gráfica de usuario y otras imágenes del proyecto se utilizará el editor gráfico **Adobe Photoshop**, por su buena relación potencia/simplicidad y por su mayor base de usuarios frente a la competencia, lo que permite obtener soporte con mayor facilidad.

Durante el desarrollo del proyecto, se llevará a cabo un **control de versiones** mediante el software **Git**, utilizando la plataforma **GitHub** para el almacenamiento de versiones. Para simplificar el uso de Git, se utilizará la herramienta **SourceTree**, una aplicación que funciona como interfaz gráfica para Git. Una vez finalizado el proyecto, la distribución de la aplicación resultante se realizará mediante las plataformas en línea de distribución de juegos **Gamejolt**¹ y **Itch.io**².

¹<https://gamejolt.com/>

²<https://itch.io/>

Capítulo 2

Estado del Arte

En este capítulo se introducirá el contexto actual del desarrollo de videojuegos. La discusión se centrará en los siguientes temas: exponer el **estado socioeconómico** en el que se encuentra el mercado del videojuego, tanto mundial como nacional; describir el proceso y metodologías utilizadas durante el desarrollo de juego; describir los **motores de juegos**, unas herramientas fundamentales a la hora de desarrollar juegos actualmente y hablar del uso de la **inteligencia artificial** en ámbito de los videojuegos.

2.1 El mercado de los videojuegos

2.1.1 Industria Mundial del Videojuego

La industria del videojuego es actualmente **el principal motor del entretenimiento global**, con un público de más de 2.200 millones de jugadores a nivel mundial y unos beneficios 116.000 millones de dólares. Se trata de una industria en expansión, con una tasa de crecimiento anual del 8,8 %. Como se puede observar en la figura 2.1, se estima que para el año 2020 la cotización anual alcance los **143.500 millones de dólares** [DEV17].

Actualmente, la plataforma de distribución que ocupa un segmento mayor del mercado son los **dispositivos móviles**. Debido al constante incremento de potencia de los teléfonos inteligentes, así como a su ubicuidad en la sociedad actual, el mercado de videojuegos para estas plataformas ha experimentado un incremento constante en los últimos años, superando al mercado para PC y al mercado para videoconsolas de sobremesa. A fecha de 2017, el mercado de los juegos para teléfonos inteligentes ha alcanzado los **39.440 millones de dólares**, lo que representa el 34 % del mercado. Por otro lado, el mercado de las consolas portátiles y el de los juegos web casuales son los que presentan un declive más pronunciado, debido seguramente a que ocupan un nicho de mercado similar al de los juegos móviles [DEV17].

El mercado del videojuego se encuentra liderado por la región de **Asia Pacífico**, la cual incluye a países como China, Japón y Corea del Sur entre otros. Esta región por si sola supone prácticamente la mitad de los ingresos globales, con un total de 57.800 millones de dólares generados en el año 2017, de los cuales 32.5 millones fueron

2. ESTADO DEL ARTE

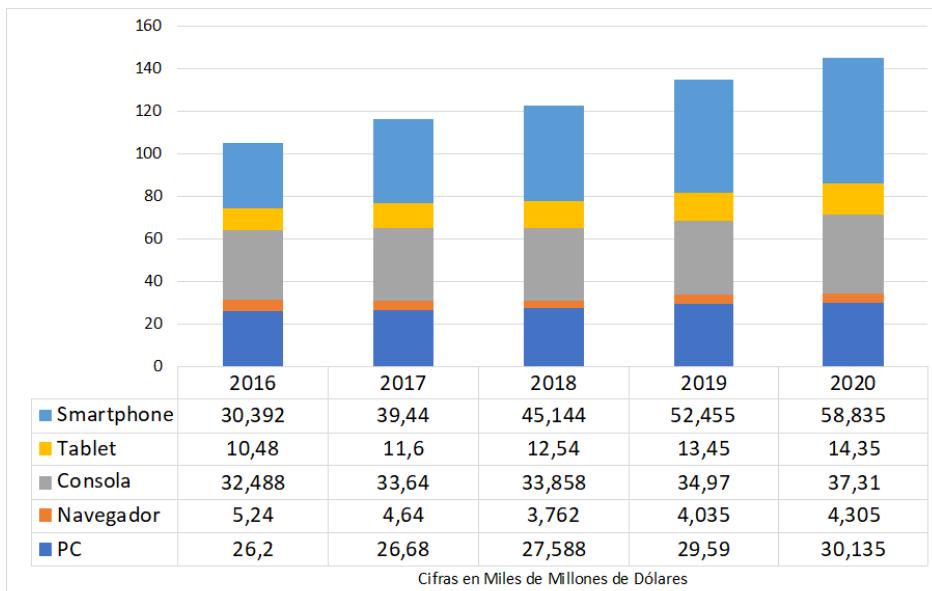


Figura 2.1: Crecimiento mundial de la industria del videojuego (por plataformas).

producidos solamente por China. Detrás del gigante asiático se encuentra **Estados Unidos**, que representa casi la totalidad de los ingresos de la región norteamericana con un total de 25.4 millones de dólares; y Japón, cuya producción asciende a los 14 millones. Por detrás de las principales potencias se encuentran varios países europeos: Alemania, Francia, España e Italia; sin embargo, como puede verse en la figura 2.2, la diferencia en tamaño con respecto a los tres primeros mercados de la lista es abismal [DEV17].

2.1.2 La industria de los videojuegos en España

La industria del videojuego española es la **cuarta mayor de Europa** (como aparece en la figura 2.3) y la novena mayor a nivel mundial. A fecha de 2017, el mercado español del videojuego facturó un total de 1.900 millones de dólares, con un crecimiento del 20 % con respecto al año anterior. Más de la mitad de estos ingresos provienen de la **venta de videojuegos españoles al extranjero**, gracias a la casi total falta de fronteras para la distribución internacional de productos. Este enfoque en el mercado internacional se ve reflejado en factores como la mayor frecuencia del inglés en las producciones españolas que el propio español (99 % contra 95 %) [DEV17].

En total, el sector cuenta con **480 empresas en activo**, sin contar con las más de 100 iniciativas y proyectos empresariales, que se encuentran a la espera de consolidarse como empresas en el corto o medio plazo. En la figura 2.4 se puede apreciar que la mayor parte de estas empresas tienen una plantilla de menos de 5 empleados, formando el 47 % de la industria. Esto se debe en parte a la adecuación de las pequeñas empresas a la creación de juegos de pequeña escala para dispositivos móviles (el principal mercado), pero también se debe a una escasez de puestos de trabajo en las empresas de tamaño

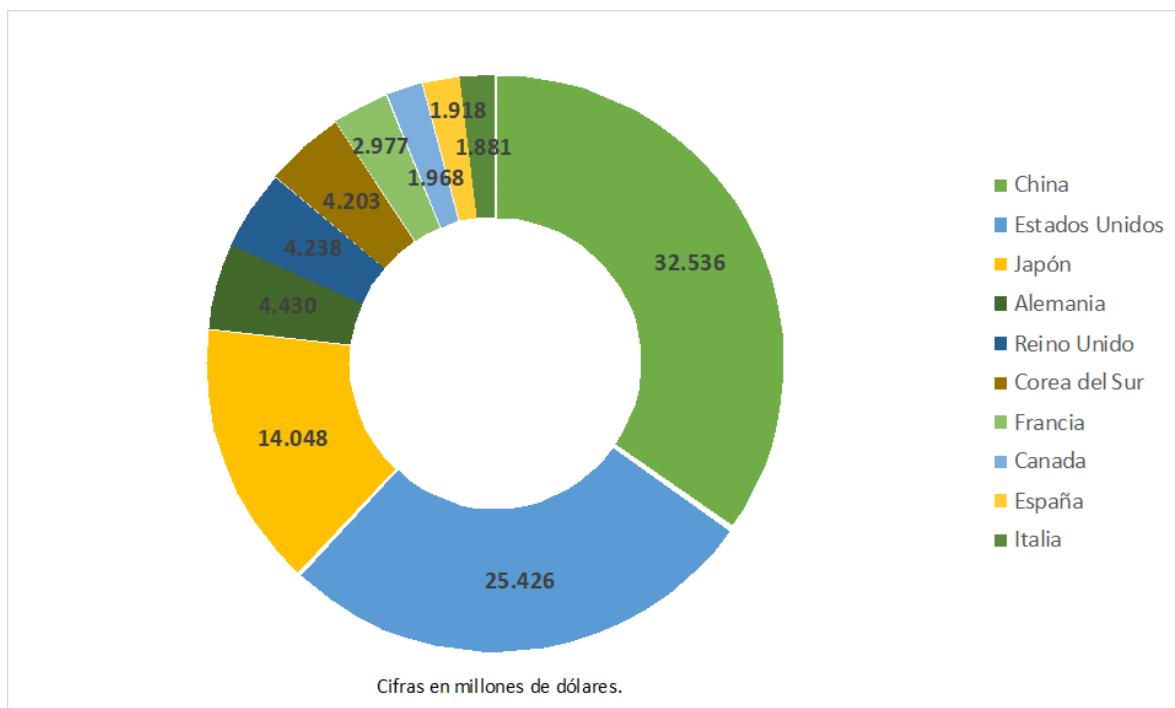


Figura 2.2: Distribución por países del mercado del videojuego.

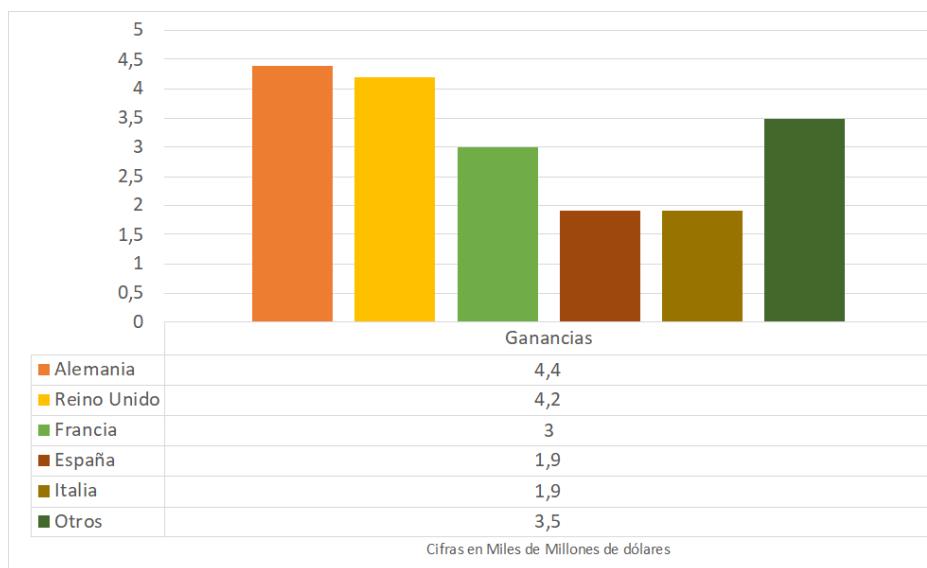


Figura 2.3: Mercado del videojuego en Europa.

mediano y grande y a la saturación del mercado que dificulta el crecimiento de las empresas [DEV17].

La actividad empresarial del país se encuentra centrada en dos comunidades autónomas: **Cataluña** y **la comunidad de Madrid**. De estos dos centros principales destaca Cataluña, donde se concentra el 52% de la facturación del país. Detrás de las dos comunidades principales se encuentran la Comunidad Valenciana, el País Vasco y Andalucía, las cuales suman entre las tres un 28% de las empresas. El resto de las

2. ESTADO DEL ARTE

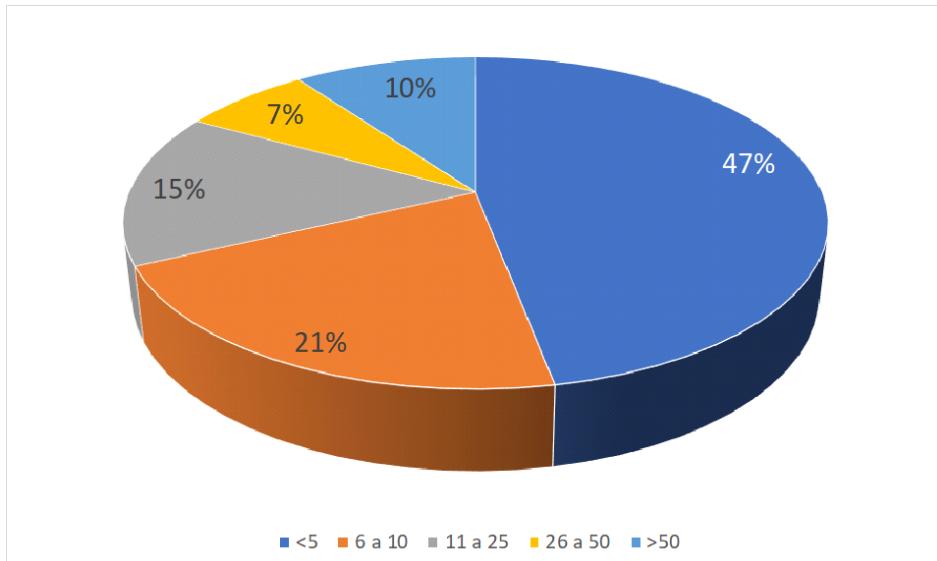


Figura 2.4: Distribución de las empresas en España por número de empleados.

comunidades se quedan muy por detrás de estas cinco primeras como puede verse en la figura 2.5 [DEV17].

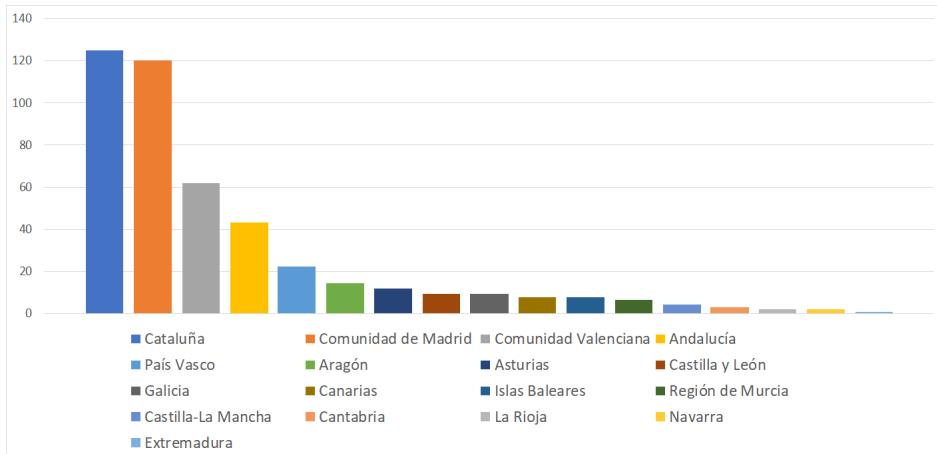


Figura 2.5: Distribución de las empresas en España por número de empleados.

2.1.3 Retos y tendencias actuales

El videojuego ha sido y siendo una industria muy cambiante, que siempre ha intentado integrar las tecnologías más punteras, desde innovadores algoritmos de renderizado gráfico hasta exóticos dispositivos de interacción persona-ordenador. Las principales tendencias que van a influir fuertemente en el mercado en los años venideros son las siguientes:

eSports

Los eSports, también llamados “deportes electrónicos”, es el nombre por el cual se conocen las competiciones de videojuegos multijugador. En los eSports, los jugadores



Figura 2.6: Fotografía del torneo Intel® Extreme Masters en Katowice, Polonia (Fuente: intelextrememasters.com)

profesionales compiten entre ellos en juegos de diversos géneros: disparos en primera persona, lucha, estrategia en tiempo real, MOBAs (Multiplayer Online Battle Arena, ver figura 2.8) entre otras. La popularidad de este fenómeno ha llegado al punto en el que los grandes torneos como el **Intel Extreme Masters** (figura 2.6) se celebran en grandes estadios, están retransmitidos en streaming por Internet e incluso están dotados con premios de grandes sumas de dinero y que en ocasiones superan el millón de euros [DEV17].

Actualmente, el impacto económico del mercado del eSport sigue siendo relativamente bajo, con unos ingresos de solo 660 millones de dólares en el año 2017. Esto se debe en parte a que **el gasto anual del “aficionado” medio es mucho menor que el de los aficionados a los deportes tradicionales** (3,64 dólares frente a 54) lo cual frena las inversiones de muchas compañías patrocinadoras. Sin embargo, las perspectivas de crecimiento son del 35,9 % anual, lo que provocará que para el año 2020 se alcance la cifra de los 1.504 millones de dólares, como se puede apreciar en la figura 2.7.

Otro factor que frena el crecimiento de los eSport es el enorme coste de su producción. En la mayoría de los casos, los eSports **surgen de manera orgánica** alrededor de juegos con una importante comunidad online de jugadores. Para reproducir estas condiciones, las empresas deben realizar importantes inversiones en infraestructura, personal dedicado a la comunidad, servidores escalables para una gran masa de jugadores o premios para los torneos.

Sin embargo, esta inversión **no asegura que un producto tenga éxito** como eSport. Para que un videojuego pueda convertirse en un eSport necesita contar con características básicas: tener un fuerte factor de competición, partidas cortas de no más de 1 hora, sin progresión in-game (la progresión debe basarse en las habilidades del

2. ESTADO DEL ARTE

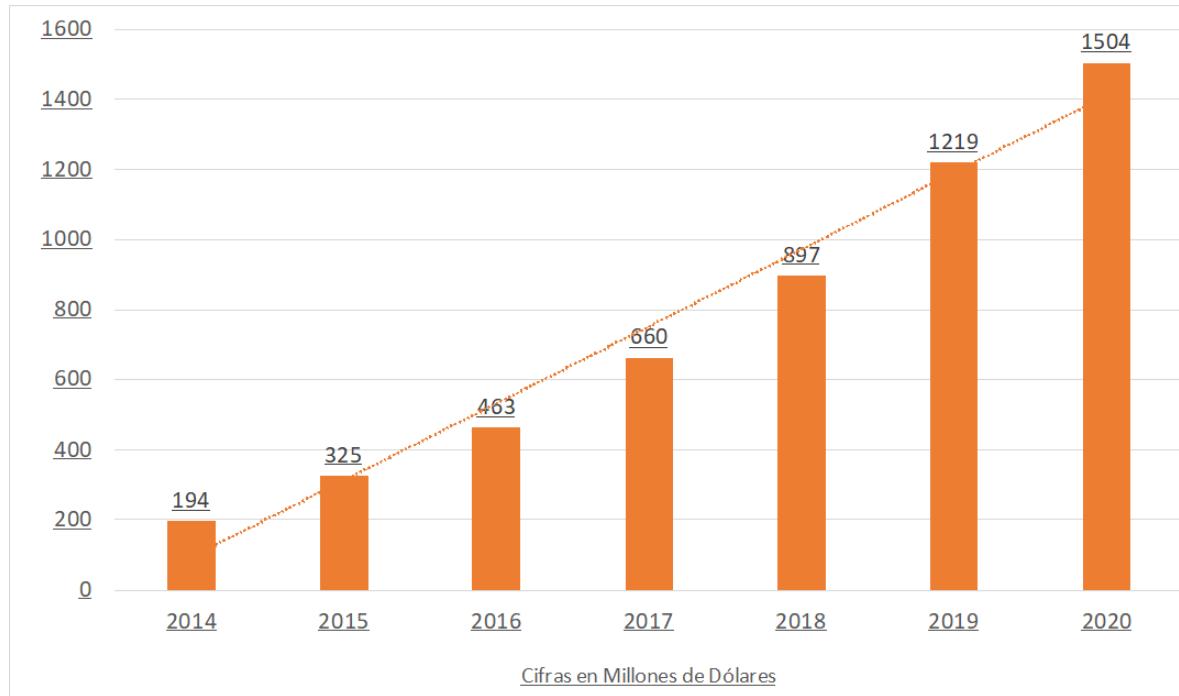


Figura 2.7: Crecimiento del mercado del eSports (Fuente: [DEV17]).

jugador) atractivo sistema de espectador y tener un enfoque al 100 % internacional.

Pese a su gran dificultad, conseguir posicionar un producto como eSports, aporta una serie de beneficios y posibilidades:

- Crear una **base de fans**, una comunidad, algo que aporta un núcleo de consumidores fieles al producto y que le da una nueva dimensión social, muy atractiva para muchos de los consumidores de videojuegos.
- **Prolongar la vida del producto**; al ser competitivo, el jugador fija sus metas ante los otros jugadores, esto incentiva al usuario y le proporciona una motivación para seguir consumiendo.
- **Proporcionar mayor visibilidad**, ya que a pesar de que los productos asentados son extremadamente sólidos, su número es muy reducido, por lo cual hay una demanda latente de usuarios que buscan nuevos eSports.
- **Aumentar la fidelidad de los usuarios** al tratarse de un mercado donde los usuarios tienen un índice de fidelidad mucho más alto que en otros.
- Los jugadores, al estar involucrado con un producto competitivo, ven streaming, leen noticias, siguen torneos, participan en foros, lo que **disminuye el riesgo de abandono del producto**.

Para una empresa pequeña, la producción de un eSports es, en principio, inabordable. Esto se debe principalmente la elevada inversión mencionada anteriormente. Sin



Figura 2.8: *League of Legends* (Riot Games, 2009), uno de los eSports más populares (Fuente: mobygames.com).

embargo, en **asociación con grandes compañías** que puedan invertir en la infraestructura y publicidad necesaria, los pequeños estudios de desarrollo pueden tener una ventaja gracias a su flexibilidad para adaptarse a los cambios intrínsecos de un mercado nuevo.

Realidad Virtual y Realidad Aumentada

La **Realidad Virtual** (normalmente abreviada como VR por las siglas inglesas de Virtual Reality) es la tecnología generada por sistemas informáticos que proporcionan un entorno audiovisual en 3D el que el usuario puede experimentar una inmersión total. Para ello, se hacen usos de cascos especiales equipados con pantallas y sensores de movimiento, los cuales se complementan con mandos equipados también con sensores para permitir una interacción más natural con el entorno virtual.

Por otro lado, la **Realidad Aumentada** o AR es una tecnología que superpone una capa de gráficos generados por ordenador sobre el entorno que rodea al jugador, con la que este puede interaccionar en tiempo real. A diferencia de la VR, no se requiere obligatoriamente de un hardware especial para poder implementar AR; basta únicamente de un dispositivo equipado con una pantalla y una cámara de vídeo, como podría ser un Smartphone.

El sector de la realidad virtual facturó en el año 2016 un total de **2.700 millones de dólares**, 1.100 millones menos de lo previsto según las estimaciones iniciales, mientras que la realidad aumentada generó un total de 1.200 millones de dólares, gracias en gran parte al éxito de *Pokémon Go* (Niantic, 2016). Pese a este lento inicio, las previsiones de crecimiento siguen siendo positivas, alcanzando los 108.000 millones de dólares en

2. ESTADO DEL ARTE

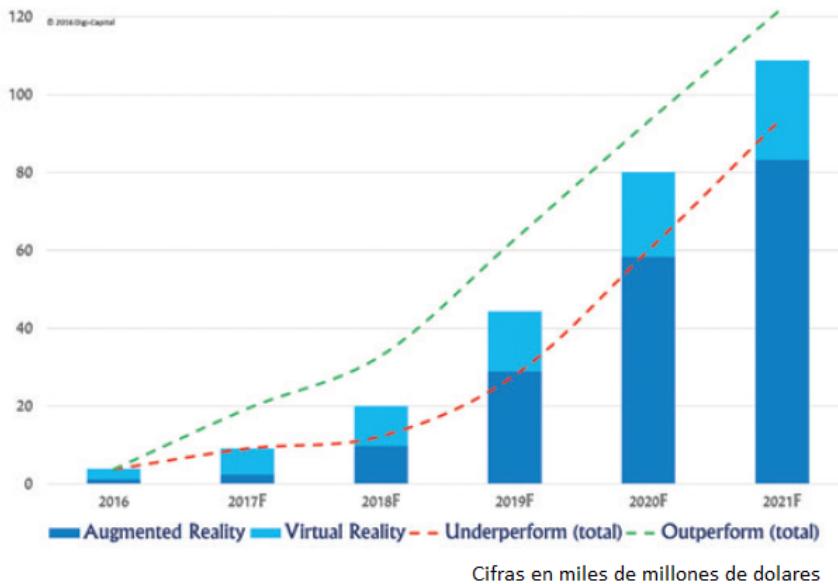


Figura 2.9: Previsiones de crecimiento del mercado de Realidad Virtual y Aumentada (tabla extraída de [DEV17]) (miles de millones de dólares).

el año 2021, como se puede ver en la figura 2.9 [DEV17].

Actualmente, existen varias propuestas de diversas compañías en lo que a equipo de VR se refiere. Estas son algunas de las propuestas más importantes:

- **HTC Vive¹**: es la propuesta de HTC y Valve, orientada a jugadores “hardcore” de PC. Disponible desde abril de 2016, el dispositivo requiere de un PC de gama alta (Valve recomienda un PC con una gráfica GeForce GTX 970). El kit de hardware incluye el casco equipado con dos pantallas de 1080x1200 puntos y 90Hz de frecuencia de actualización, dos sensores espaciales y dos mandos para registrar los movimientos de ambas manos, lo que crea le permite crear un entorno 100 % virtual en el que sumergir al jugador.
- **OCULUS Rift²**: Es la propuesta más veterana de la lista. Empezó como un exitoso proyecto de Kickstarter en 2012 que más tarde fue adquirida por la empresa Facebook dos años más tarde. Al igual que HTC Vive, Oculus está formado por un casco equipado con pantallas de alta resolución, mandos con sensores de movimiento y dos sensores de posición. El equipo necesita estar conectado a un PC de alta gama para poder funcionar correctamente.
- **Samsung Gear VR³**: La propuesta de Samsung es mucho más sencilla y eco-

¹<https://www.vive.com/>

²<https://www.oculus.com/rift/>

³<http://www.samsung.com/es/wearables/gear-vr-sm-r325nzbaphe/>

nómica, orientado más a la reproducción de vídeo en 360º (concepto similar a la realidad virtual, pero con interactividad limitada). El casco incluye una única pantalla y sus mandos carece de detección de movimiento. Estas limitaciones conllevan, por otro lado, un precio mucho más accesible que el de las otras alternativas (99€ contra los más de 500€ de las propuestas más completas)

- **SONY PlayStation VR⁴**: la propuesta de Sony fue lanzada en el año 2016. Al igual que otras alternativas, el sistema se basa en un casco equipado con dos pantallas y sensores de movimiento, pero su principal punto de venta es su compatibilidad con la consola PlayStation 4 de la misma marca. Esto permite aprovechar la potencia y los mandos de control de ésta de la consola.

En la figura 2.10 se puede ver una fotografía de cada uno de los dispositivos descritos.



Figura 2.10: De izquierda a derecha: HTC Vive, Oculus Rift, Samsung VR y PlayStation VR (imágenes tomadas de las páginas oficiales de los productos).

Web 4.0

El término Industria 4.0 fue acuñado por el **Ministerio de Educación y Desarrollo alemán** en su plan estratégico de 10 puntos del año 2016 para mejorar la educación, investigación e industria del país para adaptarlas a las tecnologías de Internet [Lyd]. La estrategia trata cinco áreas principales:

- Fuerte cooperación entre la investigación científica y las empresas.
- Aumentar la innovación en el sector privado.
- Diseminar las tecnologías punteras.
- Internacionalizar la investigación y desarrollo.
- Fondos para individuos con talento.

De entre las distintas tecnologías que podrían categorizarse como parte de la industria 4.0, las que tienen una aplicación más directa en el desarrollo de videojuegos son las siguientes [DEV17]:

La computación en la nube, o **Cloud Computing**, es la tecnología que permite el acceso a servicios informáticos de forma rápida y sencilla a través de Internet. Aunque

⁴<https://www.playstation.com/explore/playstation-vr/>

2. ESTADO DEL ARTE

aún no se ha podido implementar correctamente el **Cloud Gaming** (donde el juego es íntegramente ejecutado en la nube, reduciendo la exigencia de potencia del sistema del jugador), si se utilizan sistemas en la nube en distintas áreas de los videojuegos. Especialmente notable es su uso para el control y almacenamiento de información en juegos multijugador en línea.

El Internet de las cosas es como se conoce a la tecnología que permite dotar de conexión a Internet a todo tipo de pequeños dispositivos como relojes, sistemas de domótica, drones, sensores de todo tipo, robots, etc. Esto permite implementar videojuegos en todo tipo de sistemas, desde consolas portátiles cada vez más pequeñas y económicas, pasando por juguetes interactivos y llegando a la posibilidad de gamificar con facilidad procesos industriales.

Big Data es el proceso de clasificar grandes volúmenes de datos para poder obtener relaciones interesantes y no evidentes entre ellos. El principal uso de las técnicas de BigData en la industria del videojuego es el análisis de la información de los jugadores. Analizando datos de los jugadores tales como el género, la edad, la localización geográfica, los intereses, los gastos realizados, etc. es posible obtener estrategias de negocios eficientes.

Los sistemas Ciberfísicos son un nuevo tipo de sistemas con unos componentes hardware y software estrechamente interconectados, cada uno operando en su propio ámbito, operando e interaccionando de forma distinta dependiendo del contexto [Sid14]. Entre sus aplicaciones se encuentran las redes eléctricas inteligentes, los sistemas de conducción automática de aviones y automóviles o la monitorización médica. Para la correcta manipulación de estos sistemas se requieren de unas interfaces de usuario con un fuerte “lado humano” que permita un uso sencillo e intuitivo. Aquí se podrían utilizar los principios de diseño de juego que permitirían desarrollar un mejor puente entre el lado máquina y la parte de usuario.

La Impresión 3D, también conocida como la producción aditiva es una tecnología que permite producir objetos de forma más sencilla que con las técnicas anteriores gracias a máquinas como la mostrada en la figura 2.11. En combinación con las técnicas de escaneado 3D, los estudios de videojuego pueden generar de forma rápida y eficiente modelos 3D de todo tipo (personajes, mapas, objetos...).

Las nuevas tecnologías de la industria 4.0 serán de gran ayuda para el desarrollo de videojuego. Pero es posible que la industria 4.0 también ofrezca valor a la industria 4.0 en su conjunto. Dado que la creación de videojuegos es una actividad industrial que está vinculada a diferentes áreas de conocimiento que trabajan juntas para conseguir ofrecer un producto, las técnicas y **paradigmas utilizados tienen mucho en común con la nueva forma de trabajar de la industria 4.0**, por lo que es posible que puedan



Figura 2.11: Impresora 3d “replicator” de la compañía Makerbot.

extrapolarse a otras industrias, permitiendo una mejor adaptación a los cambios.

2.2 El proceso de desarrollo de videojuegos

El desarrollo de un videojuego, como el de cualquier otro producto software, debe de ser planificado correctamente y ejecutado siguiendo una metodología adecuada. Sin embargo, el diseño y desarrollo de un videojuego requiere de la participación de campos ajenos a la informática como el diseño de juegos, el diseño gráfico o la composición musical. Una parte importante de la producción consistirá en organizar a un equipo multidisciplinario para poder terminar el proyecto dentro del tiempo y presupuesto acordados [Dav15].

2.2.1 Etapas del desarrollo

El proceso de desarrollo de videojuegos difiere del de otros tipos de software, debido a la necesidad de integrar en un mismo producto elementos de diferentes disciplinas. En ese sentido, el desarrollo de un videojuego **es similar a la producción de una película de cine**, para las cuales también es necesario coordinar el trabajo de profesionales de áreas muy diversas. Partiendo en esta similitud, el desarrollo de videojuego suele organizarse en las tres etapas en las que se divide la producción de una película: **pre-producción, producción y postproducción**.

Para organizar el trabajo en cada una de estas etapas, se suele utilizar el **modelo en cascada de Royce**, el cual se basa en realizar las tareas de forma lineal [Dav15]. En la figura 2.12 y en los siguientes apartados se describe el desarrollo basándose en este modelo.

2. ESTADO DEL ARTE

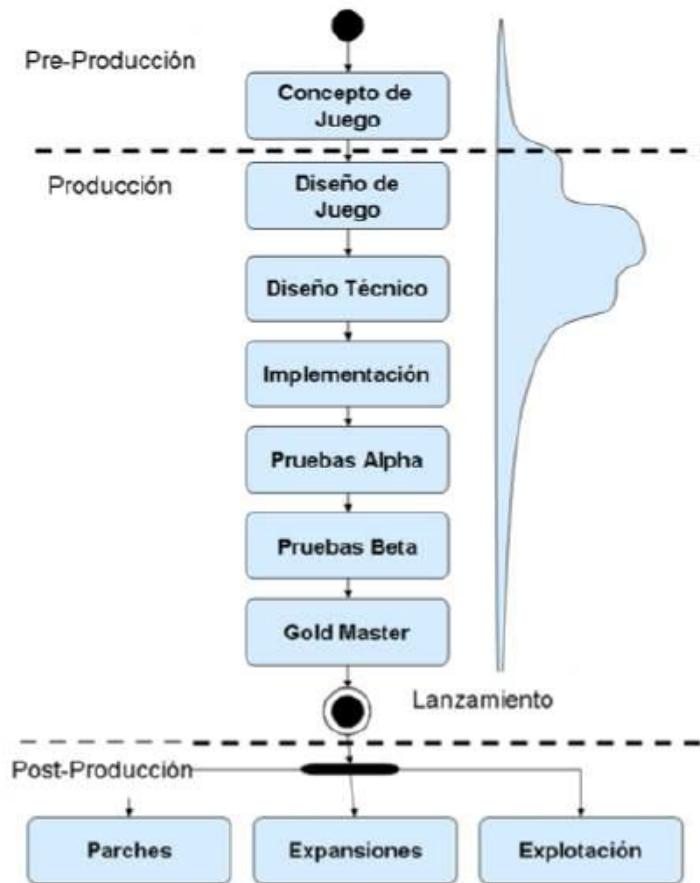


Figura 2.12: Etapas del desarrollo de un videojuego (Imagen tomada del [Dav15]).

Desarrollo del Concepto

El desarrollo de todo videojuego comienza con una idea. Durante la fase de desarrollo del concepto se tomará dicha idea para obtener un **diseño preliminar** listo para preproducción. El objetivo principal de esta etapa es decidir sobre qué tratará el juego y ponerlo por escrito para que cualquier miembro del equipo lo pueda entender con claridad, decidiendo las principales mecánicas, creando el arte conceptual y escribiendo el argumento [Bat04].

Al final de la etapa se habrán elaborado tres documentos: El **high concept**, el **pitch document** y el **concept document**. El **high concept** consiste en una o dos frases que describen a grandes rasgos cómo será el juego, en especial que lo hace distinto de la competencia. El **pitch document**, o “propuesta de juego”, es un pequeño documento de en torno a dos páginas, orientado a ser leído en las reuniones donde el juego sea propuesto a inversores. El documento resume las características del juego y explica por qué será exitoso y rentable.

Finalmente, **concept document** es un extenso documento que explica en detalle las características del juego. Se trata de una versión extendida de *pitch document* que

trata temas como:

- El high concept
- El género del juego
- Características principales y jugabilidad.
- Ambientación e historia.
- Estimación del presupuesto y de la planificación.
- Equipo de desarrollo.
- Análisis de riesgos.

Pre-Producción

La preproducción es la fase de preparación: en la que el equipo diseña y planifica los elementos que serán desarrollados en la etapa de producción. Al final de esta etapa, se debe haber completado el **diseño de juego**, creado la **biblia de arte**, elaborado el **documento de diseño técnico**, establecido el **plan de producción**, y creado un **prototipo del juego final** [Bat04].

El diseño del juego quedará establecido en un **documento de diseño de juego** (o GDD, por sus siglas en inglés). Se trata de un documento “vivo”, que está en constante modificación para adaptarse a los ajustes concretos de diseño que se realizarán durante el desarrollo.

La biblia de arte es una colección de arte conceptual que servirá para definir el estilo artístico del juego desde un primer momento. La biblia incluirá también una librería de imágenes de referencia que puedan ser de ayuda a los artistas que desarrollen los elementos gráficos finales.

El documento de diseño técnico contiene una descripción en detalle la parte técnica del proyecto definiendo las tareas que los desarrolladores deberán afrontar, estimando el coste que dichas tareas tendrán tanto en tiempo como en número de personas y especificando las herramientas y técnicas que utilizará el equipo.

El Plan de Producción recopila la información acerca de cómo se va a desarrollar el proyecto. Este incluye las tareas a realizar junto a los tiempos, costes y dependencias de estas, divididos en varios documentos menores para poder ser organizado mejor:

- **Plan de mano de obra:** Listado del personal, sus horarios y su salario.
- **Plan de recursos** Estimación del coste los recursos externos al proyecto (música, arte, herramientas...)
- **Documento de seguimiento:** Documento donde se realiza un control de los tiempos y plazos del proyecto.

2. ESTADO DEL ARTE

- **Presupuesto:** Contiene el coste mensual del proyecto y el cálculo del presupuesto general
- **Ganancias y pérdidas:** Estimación de las ganancias y pérdidas del proyecto. Debe ir actualizándose según se avanza en el desarrollo.
- **Definición de hitos:** Lista de las distintas “metas” del proyecto, que son puntos del desarrollo donde se habrá terminado una cantidad de trabajo importante.

Una vez diseñado y planificado el proyecto, el equipo empezara a trabajar en la creación de un **prototipo**. Un prototipo es una pieza de software funcional que contiene una pequeña fracción del software final. El desarrollo prototipo servirá para varias funciones: poner a prueba el diseño de juego, concretando de forma más precisa la jugabilidad; realizar un simulacro de desarrollo para determinar las dinámicas del equipo y producir una muestra del juego final a inversores y publicadores [Bat04].

Producción

La producción es la etapa principal del desarrollo del juego. Durante esta etapa se elaborarán e implementarán los elementos descritos y diseñados durante la preproducción: los programadores implementarán los sistemas del juego, los artistas elaborarán los gráficos definitivos, los diseñadores crearán misiones y niveles, etcétera. Dependiendo del juego en cuestión, una producción normal suele durar entre seis meses y dos años, aunque el desarrollo de juegos pequeños para, por ejemplo, dispositivos móviles puede realizarse en menos tiempo aún [Bet03].

La etapa de producción es de **naturaleza iterativa**: El juego va construyéndose en varias etapas en las que se implementan pequeñas porciones de este. Entre etapa y etapa, el juego pasa por un proceso de pruebas en la que se verifica su usabilidad y robustez frente a fallos. Los resultados de las etapas se utilizan como base para recalcular los tiempos y presupuestos de las etapas posteriores. Dividir el desarrollo de esta forma hace que sea más sencillo afrontar problemas y contratiempos que el equipo podría encontrar en las etapas tardías.

Final de Producción y Lanzamiento

Durante las últimas etapas de la producción, el paradigma de desarrollo cambia, pasando el objetivo de implementar contenido nuevo a pulir y ajustar el contenido preexistente. Esta parte de la producción puede dividirse en dos etapas: Alpha y Beta.

La fase **Alpha** o Code-Complete es el punto del desarrollo donde el juego se encuentra en un estado jugable, a falta solo de ciertos vacíos como gráficos provisionales o minijuegos o subsistemas incompletos. El objetivo de esta etapa es el de encontrar y corregir todos los fallos posibles y también probar y ajustar la jugabilidad [Bet03].

En la fase **Beta** o Content-Complete la mayor parte del contenido del juego deberá estar terminado y debe haber pocos o ningún fallo importante en el juego. En esta etapa el juego es puesto en manos de equipos de testing externos a la empresa para realizar análisis exhaustivos en busca de fallos que se le pueden haber escapado al equipo de testing interno. Es también en esta etapa donde la campaña de publicidad del juego deberá ser más fuerte [Bet03].

Una vez superadas las dos etapas de pruebas, la **versión final** del juego es enviada a la distribuidora para que comience la producción de las copias físicas, o para que el juego aparezca en las plataformas de distribución.

Postproducción

Una vez lanzado el juego, el equipo entra en la etapa de **postproducción**. En esta etapa el equipo trabajará en corregir fallos y problemas que los jugadores encontraron tras el lanzamiento y en la elaboración de contenido adicional descargable.

La duración y trabajo de la posproducción depende mucho del juego en cuestión. Hasta hace relativamente poco, los juegos lanzados en videoconsolas carecían completamente de esta etapa debido a la dificultad para modificar los juegos que ya se encontraran en el mercado. Por otro lado, hoy en día la mayor parte de los videojuegos reciben parches y actualizaciones sin importar su plataforma de distribución [Bet03].

Un caso especial sería el de los juegos con un fuerte componente online, como los **MMORPG**, los **MOBA** o los **shooter en línea**. Los desarrolladores de este tipo de juegos, para mantener contenta a su base de jugadores y evitar el estancamiento, lanzan de forma periódica actualizaciones que ofrecen nuevo contenido, mejoras y ajustes. Algunos de estos juegos pueden recibir este tipo de actualizaciones durante años, como *World of Warcraft* (Blizzard Entertainment, 2004) o *Team Fortress 2* (Valve, 2007), ambos en activo desde hace más de 10 años.

2.2.2 Estructura típica de un equipo de desarrollo

El desarrollo de un videojuego puede llevarse a cabo por equipos de desarrollo muy distinto dependiendo de la **extensión del proyecto**, desde una sola persona creando un pequeño juego independiente, el cual realiza todo el trabajo por sí mismo; hasta los equipos de cientos de personas que desarrollan los juegos AAA, organizados en múltiples departamentos, cada uno con su estructura jerárquica.

A pesar de esto, existen una serie de roles que están presentes en todos los desarrollos. En la figura 2.13 se puede ver la estructura de roles. Hay que tener en cuenta que una estructura tan completa solo aparece en los **grandes equipos**. En proyectos pequeños es normal una persona ejerza varios roles, o incluso que una única persona lleve el desarrollo

2. ESTADO DEL ARTE

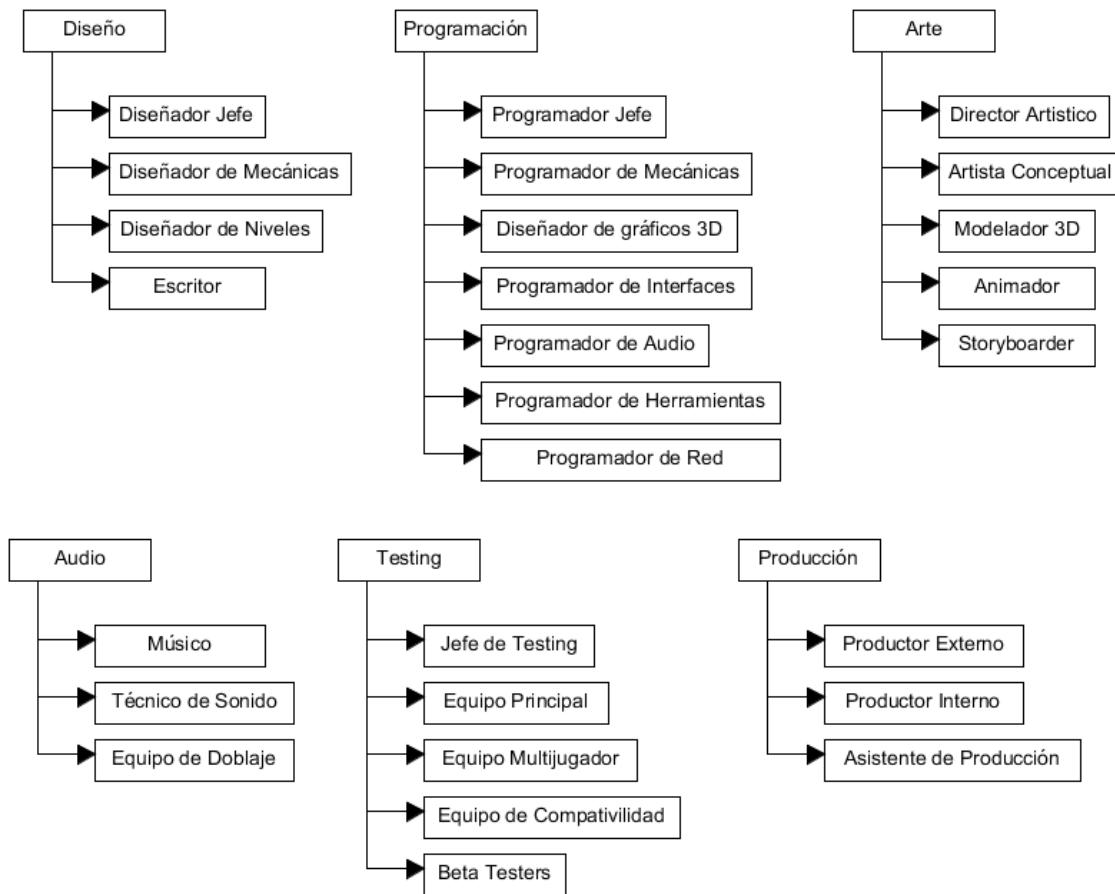


Figura 2.13: Distribución de roles en un equipo de desarrollo.

completo.

Diseño

La primera y más importante parte del desarrollo de un juego es el **Diseño** de este. El trabajo del **diseñador** es de describir el juego con un alto nivel de detalle, definiendo de con precisión todas las mecánicas, personajes, mapas, misiones del juego. Deberá también, hasta cierto punto, coordinar y dirigir el trabajo del resto de miembros del equipo para que puedan implementar correctamente el juego.

En equipos grandes, el rol de diseñador puede dividirse en las siguientes categorías [Bet03]:

1. **Diseñador jefe:** Es el encargado de dirigir al resto de diseñadores, decidiendo que contenido entra o no en el juego. Suele ser la persona que tuvo la “idea” original del juego.
2. **Diseñador de mecánicas:** El diseñador de mecánicas es el encargado de diseñar los distintos sistemas de juego, sirviendo como puente entre el diseñador jefe y los

programadores. Debido a esto, el diseñador de mecánicas suele tener un trasfondo de programador.

3. **Diseñador de niveles:** También llamados diseñadores de misiones son los encargados de crear las distintas etapas que componen el juego, ya sean niveles, misiones, desafíos o puzzles.
4. **Escritor:** La tarea del escritor es la de crear la historia del juego, así como la de escribir los distintos textos de este, como diálogos o descripciones. Se trata de una tarea muy distinta de la de un escritor de novelas o de guiones de película, ya que debe conciliar la narrativa con las exigencias de otros componentes del juego como el diseño, el arte o las limitaciones técnicas.

Programación

El rol del **programador** es el de implementar el juego en forma de código ejecutable. Esto supone el diseño e implementación de todo tipo de componentes imprescindibles: motor de renderizado, librerías para trabajar con sistemas de audio o conectarse por Internet, herramientas para integrar fácilmente el contenido artístico entre otras.

Cuando el equipo de desarrollo es grande, el rol de programador tiende a dividirse para cubrir tareas más específicas [Bet03]:

1. **Programador jefe:** El programador jefe es frecuentemente el programador con más experiencia del equipo y él es el encargado de resolver las tareas más complicadas e importantes del proyecto. Cuando el equipo es muy grande, suelen realizar también tareas de coordinación.
2. **Programador de mecánicas:** Es el encargado de convertir el diseño de juego en código ejecutable. Entre sus tareas se encuentra definir las físicas del mundo del juego, definir las funciones de los distintos objetos y modelar el comportamiento de los personajes.
3. **Programador de gráficos 3D:** Es el responsable de implementar los sistemas para la creación y renderizado de gráficos 3D. El programador de gráficos 3D necesita contar con conocimientos avanzados en cálculo, matemática vectorial y matricial, trigonometría y álgebra.
4. **Programador de interfaces:** Es el encargado de implementar los sistemas de interacción entre el jugador y el juego, normalmente interfaces de control, menús y HUDs (Head-Up Displays).
5. **Programador de audio:** Es la persona responsable de la implementación de los distintos sistemas que se utilizarán para reproducir música y sonido en el juego
6. **Programador de herramientas:** El programador de herramientas tiene la responsabilidad de crear las distintas herramientas que el resto del equipo pueda ne-

2. ESTADO DEL ARTE

cesitar para realizar, o acelerar, su trabajo. Un tipo especializado de programador de herramientas es el programador del editor de niveles, debido a la importancia de esta herramienta para el desarrollo del juego y por la posibilidad de que dicho editor sea lanzado al público como parte del juego.

7. **Programador de red:** Es el encargado de escribir el código que permite a los juegos ser ejecutados entre varios equipos, ya sea código máquina de bajo nivel o la integración de una librería de alto nivel.

Arte

Se denomina **artista** a la persona o grupo encargado de generar los componentes gráficos del juego: modelos 3D de personajes y objetos, texturas, diseño de menús e interfaces, bocetos, animaciones y demás.

Existen varias categorías de artistas distintas dependiendo de en qué rama se especialicen y de cuál sea su rol en la estructura del proyecto [Bet03]:

- **Director artístico:** Asignado al artista con mayor experiencia en la industria, el papel del director artístico es el de organizar y coordinar al resto de artistas para que realicen correctamente su trabajo y el de revisar las piezas producidas para asegurarse de que son consistentes con el estilo artístico establecido.
- **Artista conceptual:** El artista conceptual es el encargado de producir bocetos provisionales que servirán como base para construir los gráficos definitivos del juego.
- **Artista 2D:** Los artistas 2D son expertos en las técnicas tradicionales del dibujo y pintura. Su rol es más notable en los juegos 2D, donde deben producir la mayoría de los componentes gráficos como fondos, *tiles* y *sprites*; pero también tienen un papel notable en los juegos 3D, donde suelen ser los encargados de diseñar interfaces gráficas, crear las texturas de los modelos 3D e incluso realizar trabajos ajenos al propio juego como la creación de imágenes promocionales.
- **Modelador 3D:** Es el encargado de producir los modelos 3D de los distintos componentes del juego tales como personajes, objetos, mapas... Es relativamente común que los modeladores 3D tengan ciertos conocimientos de programación debido a que eran necesarios para trabajar con los primeros programas de modelado 3d.
- **Animador:** Es el encargado de animar los diferentes elementos del juego, desde el simple movimiento de un molino de viento hasta las complicadas expresiones de una cara. Existen dos alternativas para realizar la animación: la técnica de keyframing, que consiste en realizar poses estáticas de los personajes que el programa utiliza para generar la animación; y la captura de movimiento, en la que

los movimientos de un actor son capturados y transferidos al juego mediante un equipo especializado.

- **Storyboarder:** Es el artista encargado de diseñar escenas del juego. Para ello, el Storyboarder crea unas secuencias de arte conceptual que describen los tiempos, diálogos y eventos de las escenas, lo que permite valorarla y validarla antes de iniciar el costoso proceso de producción.

Audio

El trabajo de **audio** en un videojuego viene en tres categorías: música, efectos de sonido y doblaje. Existen especialistas que se dedican exclusivamente a una sola de estas categorías, aunque no es raro encontrarse en pequeños estudios a una persona encargarse tanto de la música como del sonido. Reflejando los tipos de audio, los tres tipos de profesionales son [Bet03]:

1. **Músico:** Es el artista encargado de escribir las composiciones musicales que se escucharán a lo largo del juego. Es muy común que el músico también se haga cargo de interpretar sus composiciones mediante programas de síntesis de música, aunque las grandes producciones pueden permitirse contratar interpretaciones en vivo.
2. **Técnico de sonido:** Los técnicos de sonido son profesionales que se dedican a fabricar o adaptar sonidos para el proyecto en el que trabajen.
3. **Equipo de doblaje:** El trabajo de doblar un videojuego requiere del trabajo de varios profesionales. En primer lugar, está el actor de voz, un actor especializado que interpreta con su voz a uno o varios personajes del juego. El trabajo de los actores está supervisado por un director de doblaje, que además suele encargarse de adaptar el guion y de dirigir a los técnicos de sonido que van a grabar y manipular las voces.

Testing

El aseguramiento de la calidad (o QA por sus siglas en inglés) es un requisito clave para el desarrollo de un videojuego, a la vez de un proceso lento y costoso que debe comenzarse lo más pronto posible para evitar un sobrecoste [Bet03]. El trabajo del tester es el de revisar las distintas versiones del juego en busca de fallos para que los desarrolladores puedan arreglarlos.

Normalmente, los testers de un juego se agrupan en equipos, dirigidos por un **jefe de testing**. Cada equipo de testers se encarga de revisar una faceta distinta del juego: el **equipo principal** se encarga de probar la jugabilidad y los modos de juego individuales, el **equipo multijugador** se ocupa de revisar la jugabilidad en línea, así como los componentes técnicos de las conexiones, el **equipo de compatibilidad** prueba el

2. ESTADO DEL ARTE

juego en diversas plataformas y ordenadores con distintos componentes y el **equipo de localización** comprueba que se halla realizado una correcta traducción a distintos idiomas.

Para evitar la pérdida de punto de vista critico que el equipo principal y el equipo de multijugador pueden sufrir tras haber trabajado con el juego desde el principio del desarrollo, es normal **cambiar los equipos en las últimas etapas** del desarrollo por equipos nuevos que no estén involucrados en el juego.

Junto al trabajo de los equipos profesionales de testing se suelen realizar **campañas de beta testing**. El beta testing consiste en liberar una versión incompleta del juego para que jugadores aficionados los prueben. Las resultados y opiniones de los jugadores son recogidos para utilizarse en el desarrollo de la versión completa. Para realizar una exitosa campaña de beta testing es necesario contar con uno o más organizadores que puedan gestionar la retroalimentación de los usuarios.

Producción

La función principal del **productor** es servir de puente entre el equipo de desarrollo y el resto de la empresa. El productor debe tener un conocimiento profundo del juego y de los demás miembros del equipo, de forma que pueda explicarlo de forma correcta en las muchas reuniones que se tendrán con otros departamentos, como por ejemplo el de marketing [Bat04].

El productor se encarga de realizar la gestión del proyecto, coordinando al equipo, realizando la programación de las etapas del proyecto y gestionando los posibles riesgos.

Existen tres tipos de productores dependiendo de su especialidad. Estos son:

1. **Productor externo:** Este tipo de productor trabaja para la compañía editora y se encarga de supervisar al equipo de desarrollo para asegurarse de que se cumplen los acuerdos establecidos por ambas partes.
2. **Productor interno:** Esta clase de productor trabaja en la compañía desarrolladora y se encarga tanto de realizar una gestión interna del proyecto como de actuar de representante de del equipo.
3. **Asistente de producción:** Los asistentes de producción se encargan de realizar las tareas a las que el productor jefe del proyecto no puede dedicarse personalmente. Normalmente se trata administrar detalles concretos como administrar recursos, realizar el papeleo o gestionar los servidores y la página web.

2.3 Motores de juegos

2.3.1 Descripción

Un **motor de juego** es un framework software que facilita el desarrollo de videojuegos proveyendo al programador de la funcionalidad general necesaria para cualquier juego [War].

El término “motor de juegos” se remonta a mediados de los años noventa, cuando apareció el género de los juegos de **disparos en primera persona**. *Doom* (id Software, 1993), uno de los primeros juegos de este género, había sido diseñado de forma que existía una **separación bien definida** entre los componentes software principales (como el motor de renderizado 3D o el sistema de detección de colisiones) y los assets gráficos, los mundos y las reglas del juego. Gracias esta separación, juegos como *Heretic* (Raven Software, 1994) (ver figura 2.14) pudieron ser desarrollados cambiando solamente el arte, niveles o armas de títulos anteriores, manteniendo intacto el motor [Gre09].



Figura 2.14: *Heretic* (Raven Software, 1994), es un juego desarrollado con el motor de *Doom* (Fuente: doom.wikia.com).

Este concepto inició las comunidades de “modding”, que son grupos de aficionados que desarrollaban juegos nuevos modificando juegos antiguos, y para finales de los noventa, juegos como *Quake III Arena* (id Software, 1999) y *Unreal* (Epic MegaGames, 1998) habían sido diseñados pensando en la reusabilidad de su código. Actualmente, la mayor parte de las compañías desarrolladoras de juegos adquieren la licencia de

2. ESTADO DEL ARTE

motores desarrollados por terceros, mucho más económico que desarrollar desde cero todos los componentes.

La línea que separa el motor del juego suele ser difusa y **depende en gran medida del juego concreto**. El principal factor que se utiliza para distinguir un motor de juego de un juego que haya sido desarrollado de forma “íntegra” es la presencia de una arquitectura orientada a datos. Se trata de un paradigma de programación que se basa en diseñar software con la intención de procesar datos, en lugar de ejecutar secuencias instrucciones fijas.

Idealmente, debería ser capaz de “reproducir” cualquier juego a partir de los datos de su contenido, de la misma forma que un programa reproductor de música lee y reproduce canciones. Sin embargo, en la realidad los motores de juegos suelen estar **optimizados para un determinado género juego o una plataforma específica** debido a que de esa forma es posible obtener software más eficiente y con mayores prestaciones, aplicando técnicas y patrones de diseño específicos de esos géneros/plataformas.

Componentes de un Motor

Los motores de juego son softwares muy complejos, por lo que suelen estar construidos a partir de módulos independientes, lo que facilita enormemente su mantenimiento. Normalmente, un motor de juegos se compone de los siguientes módulos [Gre09]:

- **El núcleo:** Se trata de una aplicación compleja que recoge gran cantidad de herramientas y utilidades necesarias para el desarrollo del juego. El núcleo suele incluir una librería de funciones matemáticas, herramientas para la gestión eficiente de memoria, estructuras de datos y clases personalizadas, etc.
- **Motor de renderizado:** Posiblemente el componente más grande y complejo del juego, el motor de renderizado es el software encargado de generar los gráficos del juego. Estos motores suelen estar construidos siguiendo una estructura de capas: primero el **render de bajo nivel**, que se encarga de dibujar primitivas a la mayor velocidad posible; el **grafo de escena** determina qué sección de la escena es visible, lo que permite reducir el número de llamadas al render de bajo nivel; la capa de **efectos visuales**, que contiene el sistema de partículas, los efectos de pantalla completa, o las luces dinámicas; y finalmente la capa de **frontend**, la cual sirve para el renderizado de imágenes 2D que se superpondrán a la escena tridimensional del juego, como los menús, el HUD (Head Up Display) o videos pre-renderizados. Aparte del motor gráfico, los motores de juego actuales suelen incluir también un **sistema de animación**, el cual permita dotar de movimientos naturales a los personajes y elementos del juego.
- **Gestor de recursos:** Se trata de una interfaz que permite un acceso unificado

a los distintos assets que forman el juego (modelos, texturas, sonidos, scripts...).

- **Motor de Físicas:** El motor de físicas permite realizar la detección de colisiones entre entidades del juego, así como la simulación de comportamientos físicos realistas para dichas entidades. Hoy en día, las compañías no suelen programar sus propios motores de físicas, en su lugar adquieren motores desarrollados por terceros, como *Havok*⁵ o *PhysX*⁶.
- **Entrada y salida del jugador:** Este sistema se encarga de gestionar la información de entrada del jugador, suministrada mediante el mando de juego o el teclado y ratón. El sistema toma la información en bruto de entrada y permite al programador acceder a ella de forma más útil, limpiando los datos de entrada, creando eventos de activación de teclas y proveyendo de sistemas para mapear funciones a distintas teclas o botones y para detectar secuencias de pulsaciones. Este sistema también provee funciones para la salida de datos relacionado con los mandos de control, como activar y desactivar la vibración o emitir sonidos.
- **Sistema de audio:** Es el componente encargado de la reproducción de la música y efectos de sonido del juego. Aunque su complejidad depende en gran medida de las necesidades del motor concreto, la mayoría incluyen sistemas para producir efectos como sonido 3D o música dinámica.
- **Networking:** Son los sistemas encargado de realizar la conexión del juego con Internet para, en la mayoría de los casos, realizar partidas en línea con otros jugadores. El soporte de sistemas para el juego multijugador tiene un gran impacto en la mayoría de los componentes del motor, por lo que estos suelen ser desarrollados pensando desde el principio en el modo multijugador, e implementando el modo de un jugador como un caso específico del multijugador.
- **Fundamentos de la jugabilidad:** Esta capa implementa una serie de sistemas que permiten implementar la Jugabilidad. Suele incluir un lenguaje de Scripting, un sistema de eventos, inteligencia artificial, cámaras...
- **Herramientas de depuración:** Estas herramientas facilitan la tarea de depurar y optimizar el juego. Incluyen herramientas para dibujar en pantalla, consolas de comandos, sistemas para grabar y reproducir sesiones de juego...

2.3.2 Ejemplos de Motores

Unreal Engine

El **Unreal Engine**⁷ es un popular motor de juego desarrollado por la compañía Epic Games. Originalmente desarrollado como motor propietario para el juego *Unreal*

⁵<https://www.havok.com/physics/>

⁶<https://www.geforce.com/hardware/technology/physx#source=gss>

⁷<https://www.unrealengine.com/>

2. ESTADO DEL ARTE

(Epic MegaGames, 1998) (figura 2.15), Epic Games pronto empezó a cerrar tratos con otras compañías que querían utilizar el motor en sus proyectos. Actualmente el motor se encuentra en su **versión 4** y es uno de los más populares del sector, habiendo ganado incluso el premio Guinness al “motor de juegos más exitoso”⁸ con un total 408 juegos (a fecha de julio de 2014) desarrollados con Unreal.



Figura 2.15: Unreal (Epic Games, 1998) (Fuente: mobygames.com).

Unreal Engine es un motor orientado al **desarrollo de juegos AAA**, proyectos de gran envergadura llevados por equipos grandes. Uno de sus puntos fuertes es su potente motor de rendering el cual da soporte a **gráficos fotorrealistas** y permite el uso de efectos de post-procesados complejos entre otras características. El scripting en Unreal se realiza mediante el sistema **Blueprint** de scripting visual, el cual permite programar conectando de forma gráfica bloques de código. El motor permite también escribir código directamente en C++, lo que aumenta su flexibilidad. El paquete de herramientas del motor también incluye programas como un editor de escenas (figura 2.16) generadores de terreno, editores de materiales, herramientas para animación... Sin embargo, se trata de un motor **complicado y difícil de aprender a utilizar**, además de que la potencia que requiere lo hace poco adecuado para el desarrollo para plataformas móviles.

La licencia de uso de **Unreal Engine** es gratuita, sin embargo, en proyectos comerciales Epic Games cobra un 5 % de las ganancias a partir de los 3.000\$ por trimestre.

⁸<http://www.guinnessworldrecords.com/world-records/most-successful-game-engine>



Figura 2.16: Captura del entorno de Unreal Engine (Fuente: docs.unrealengine.com).

En casos especiales, es posible negociar otros tipos de licencias con Epic Games.

Game Maker Studio

Game Maker Studio⁹ es un motor de juegos desarrollado por Yoyo Games. El programa fue lanzado originalmente en 1994 bajo el nombre de **Amino** como una herramienta para la creación de animaciones. Desde entonces, el programa ha ido evolucionado hasta convertirse en un motor de juegos de calidad profesional.

Game Maker Studio está diseñado para ser muy sencillo de usar: su principal uso es como una herramienta para gente sin conocimientos de programación, para el desarrollo rápido de juegos pequeños o para la creación de prototipos. La interfaz de usuario de su entorno de desarrollo (ver figura 2.17) permite la creación de juegos sin escribir ni una sola línea de código, gracias a su sistema “**Drag and Drop**”, con el que se programa conectando diversos bloques de código. Para el desarrollo de juegos más complejos, el motor ofrece un lenguaje de programación propio llamado **Game Maker Language**.

El motor permite exportar con facilidad a distintas plataformas como PC, dispositivos móviles o HTML5. El entorno integrado de Game Maker incluye herramientas complementarias como un **editor gráfico** y un **editor de mapas** para centralizar el desarrollo. Sin embargo, la sencillez del motor también se refleja en su potencia: Game Maker Studio carece de soporte para gráficos tridimensionales complejos, y su arqui-

⁹<https://www.yoyogames.com/gamemaker>

2. ESTADO DEL ARTE

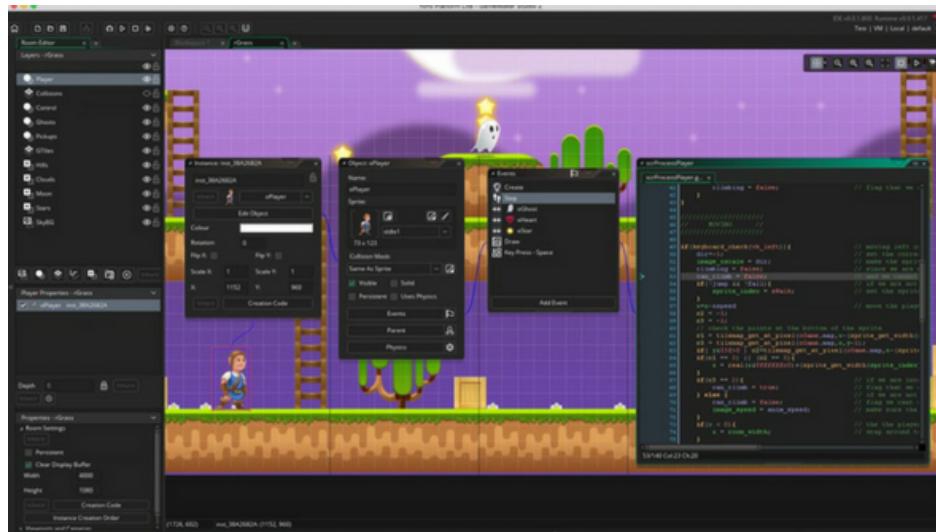


Figura 2.17: Captura del entorno de Game Maker Studio (Fuente: pcgamesn.com).

tectura dificulta el desarrollo de proyectos de gran envergadura. Estas limitaciones no han impedido la creación de juegos exitosos o revolucionarios con este motor, como podría ser *Undertale* (Toby Fox, 2015), mejor juego de PC del año 2015¹⁰.

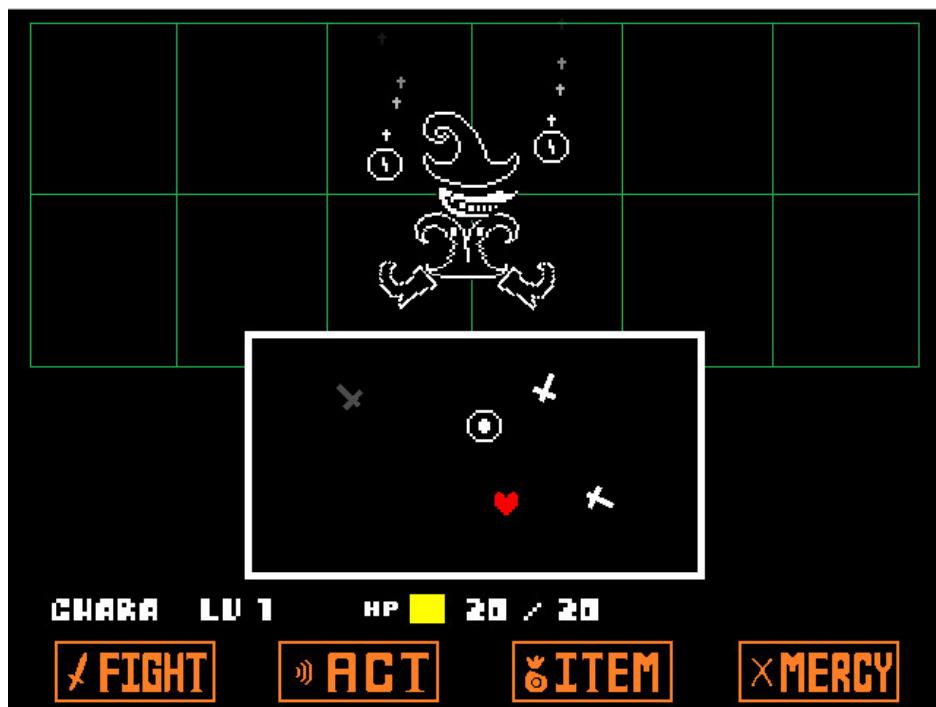


Figura 2.18: *Undertale* (Toby Fox, 2015) (Fuente: Página de Steam del juego).

La licencia de la versión actual de Game Maker Studio (Game Maker Studio 2) se encuentra a la venta por distintos precios dependiendo de la plataforma de distribución para la que se quiera trabajar: desde la versión básica por 39\$ anuales hasta la versión profesional permanente con posibilidad de exportación a IOS, Android y consolas por

¹⁰http://www.ign.com/wikis/best-of-2015/PC_Game_of_the_Year

399\$. Existen también una versión de prueba gratuita que cuenta con unas prestaciones reducidas y un plan de pago para su uso en centros educativos.

2.3.3 Unity 3D

Unity¹¹, conocido popularmente como **Unity3D**, es un motor de juego multiplataformas diseñado para el desarrollo de videojuegos y otras aplicaciones interactivas tanto en tres dimensiones como en dos dimensiones. Se trata de uno de los motores líderes en el mercado, con más de 170.000 juegos desarrollados con él motor¹², desde pequeños juegos para Android hasta grandes juegos para PC y consolas de sobremesa.

El desarrollo de este motor empezó en el año 2005 de manos de los desarrolladores **David Helgason, Joachim Ante y Nicholas Francis**. Inicialmente el motor era compatible solo con el sistema operativo Mac OS X, pero ya entonces estaba vigente la filosofía principal del motor: **una herramienta fácil de usar**, con un sistema de carga de recursos simple y una interfaz completamente gráfica [Haa14]. Con el tiempo, sucesivas versiones del motor fueron desarrolladas para mejorar tanto el rendimiento técnico como su portabilidad a distintos sistemas operativos. Hoy en día (con la versión 2017.5.3), el entorno integrado de desarrollo de Unity es compatible con Windows, Linux y Mac, con posibilidad de exportar juegos a gran variedad de plataformas: desde dispositivos móviles a videoconsolas.

Características Técnicas

Unity cuenta con un **motor de renderizado propietario** de calidad AAA. Se trata de un motor de renderizado basado en físicas que permite la creación de entornos tridimensionales fotorrealistas con iluminación en tiempo real. Adicionalmente, Unity ofrece **soporte para el uso de gráficos 2D** con el fin de facilitar el desarrollo de juegos de pequeña escala o para dispositivos móviles. El renderizado en Unity se controla mediante shaders escritos en lenguaje Shaderlab, lo que permite al programador escribir sus propios shaders para obtener diversos efectos visuales.

Para la simulación de las físicas del mundo del juego, Unity ofrece dos motores de físicas, uno para entornos tridimensionales y otro para entornos en 2D. El motor de físicas 3D de Unity es **PhysX**, desarrollado por NVIDIA, un motor de físicas multiplataformas con soporte para simulación de cuerpos sólidos, tejidos partículas y fluidos. Por otro lado, para la simulación de físicas en entornos bidimensionales Unity utiliza 2D **Box2D**, un motor de código libre que, si bien cuenta con una funcionalidad más limitada que PhysX, es muy eficiente, ideal para dispositivos móviles y consolas portátiles.

¹¹<https://unity3d.com/unity>

¹²https://unity3d.com/sites/default/files/pr_downloads/bythenumbersunitytechnologies.pdf

2. ESTADO DEL ARTE

El motor está incluido en un **entorno de integrado de desarrollo**. La principal característica de este entorno es el llamado **Editor de Unity** (figura 2.19), una aplicación gráfica en que se encuentra unificada la mayor parte de la funcionalidad necesaria para el desarrollo. El editor incluye una interfaz para la creación y edición de las escenas del juego y los objetos que estas contiene, realiza la gestión de recurso, permite ajustar la configuración... El editor incluye también la función “play” que permite iniciar el juego en cualquier momento e incluso realizar cambios en este mientras se encuentra en ejecución.

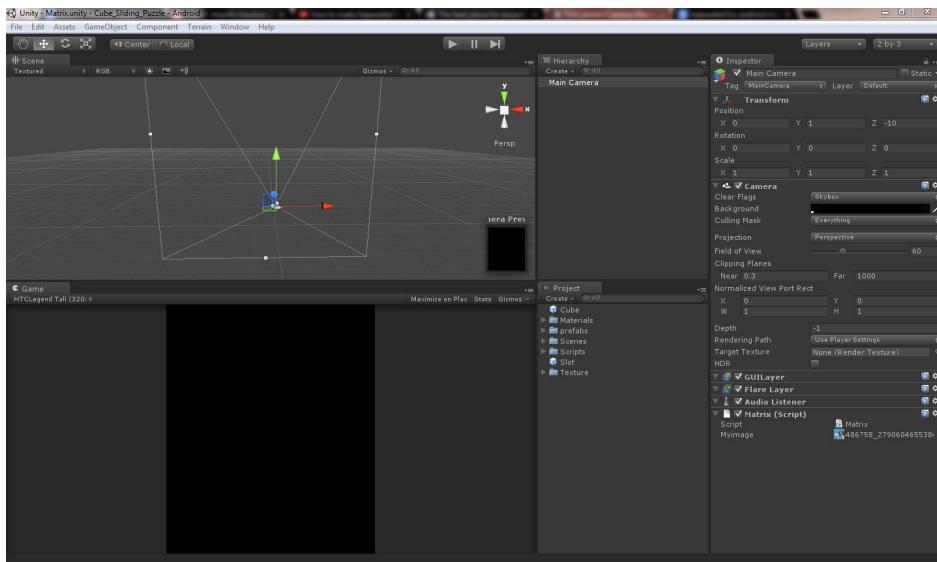


Figura 2.19: Captura del entorno de desarrollo de Unity (Fuente: answers.unity.com).

Pero la principal característica del editor de Unity es que es **completamente personalizable**. Mediante **scripts de editor**, el programador puede añadir nuevas ventanas, botones e inspectores al editor para así suplirlo de las funcionalidades adicionales que un proyecto dado pueda necesitar. Las librerías para la creación de scripts de editor están integradas con las que el motor utiliza en la implementación del juego, por lo que el programador tiene acceso nativo clases y tipos incluidos de Unity, lo que facilita la creación de herramientas para el desarrollo. Los scripts de editor se guardan como un asset más del proyecto, por lo que el editor puede configurarse individualmente para cada proyecto concreto.

La carga y gestión de **assets** en Unity es extremadamente sencillas: los recursos se guardan en forma de **archivos convencionales** dentro de una jerarquía de carpetas dentro del proyecto. Una vez que estos archivos se encuentren dentro de la jerarquía, Unity importara automáticamente cualquier cambio realizado sobre ellos con herramientas externas, lo que agiliza enormemente el desarrollo. Además, Unity cuenta con una tienda en line de recursos para juegos, completamente en el propio editor. Esto permite a los desarrolladores adquirir e integrar fácilmente todo tipo de recursos, desde

texturas a sistemas software completos.

Programación

La programación en Unity se realiza mediante **Scripts**, pequeños programas de código interpretado que se utilizan normalmente para controlar el comportamiento de objetos. Los Scripts de Unity se programan en **C#**, un lenguaje de programación desarrollado por Microsoft para su plataforma .NET. Se trata de un lenguaje de programación orientado a objetos, con una sintaxis derivada de C, pero con un proceso de compilación mucho más rápido y flexible. Además de C#, los scripts de Unity pueden desarrollarse utilizando **UnityScript**, un lenguaje de programación basado en JavaScript, pero que se encuentra en proceso de ser deprecado en favor de C#.

Le desarrollo en el entorno Unity es diferente a como se realizaría en un entorno de desarrollo integrado convencional. Esto se debe principalmente a dos factores: en primer el lugar por el énfasis que se hace en el uso del Editor, el cual es necesario para la inicialización y gestión de diversos componentes del juego. En segundo lugar, Unity hace uso de estructuras y clases específicas con las que se debe trabajar, lo que obliga al programador a cambiar su forma de programar.

La clase principal de Unity se llama **GameObjects**¹³. Los objetos de esta clase se instancian en el editor de Unity o mediante código con el método estático **Instantiate**¹⁴, sin embargo, la funcionalidad de los GameObjects viene dada por los objetos de la clase **Component**¹⁵. Esta clase es padre de multitud de componentes con diferente funcionalidad. Algunos de los componentes más importantes de Unity son los siguientes:

- **Transform**¹⁶: Este componente almacena la posición, rotación y escala del objeto en la escena. Cada transform tiene un parente, lo que permite aplicar las transformaciones de forma jerárquica. Como su función es tan importante, todos los GameObjets tienen uno asociado.
- **Collider**¹⁷: Los colliders son una familia de componentes que permiten realizar la detección de colisiones. Existen muchos tipos de colliders dependiendo de su forma (BoxCollider, SphereCollider...). La clase Collider y sus hijos interaccionan con el motor de físicas 3D de Unity, para las colisiones entre objetos en 2D se utiliza la familia de clases **Collider2D**.
- **Rigidbody**¹⁸: Esta clase se utiliza para la simulación física de cuerpos rígidos

¹³<https://docs.unity3d.com/ScriptReference/GameObject.html>

¹⁴<https://docs.unity3d.com/ScriptReference/Object.Instantiate.html>

¹⁵<https://docs.unity3d.com/Manual/Components.html>

¹⁶<https://docs.unity3d.com/ScriptReference/Transform.html>

¹⁷<https://docs.unity3d.com/ScriptReference/Collider.html>

¹⁸<https://docs.unity3d.com/ScriptReference/Rigidbody.html>

2. ESTADO DEL ARTE

en 3D. Rigidbody contiene métodos para aplicar cambios de posición y rotación a objetos basándose en su velocidad, aceleración, fricción... Para objetos en 2D, existe una clase equivalente llamada **Rigidbody2D**.

- **Renderer**¹⁹: Este componente permite renderizar modelos y texturas en la escena.
- **Audio Source**²⁰: Los audio source son componentes que permiten a los objetos emitir clips sonido. El componente permite aplicar transformaciones en tiempo real a los sonidos que emite, como modificar su volumen o la altura de su tono.
- **Animator**²¹: Este componente permite añadir animaciones a los objetos. Se trata de una máquina de estados finitos que reproduce clips de animación de forma condicional. Estos clips pueden producir variaciones en las propiedades de otros componentes del objeto en función del tiempo.
- **Particle System**²²: Es un componente que emite **partículas**, pequeñas imágenes 2D que permiten crear efectos visuales como fuego o explosiones.

El programador puede crear sus propios componentes (llamados **Scripts** extendiendo la clase **MonoBehaviour**²³. Esta clase, hija de la clase Component, provee de una serie de métodos especiales llamados **Eventos** los cuales son llamados automáticamente por Unity en momentos clave de la ejecución. Algunos de estos eventos son **Start**, que es llamado al principio de la ejecución; **Update**, que se ejecuta una vez por fotograma o **OnCollision** al que se llama cuando el objeto colisiona con otro.

Los GameObject se crean dentro de unas estructuras llamadas **Escenas**²⁴. Cada escena contiene un entorno o menú del juego en forma de archivo, así es posible organizar mejor las distintas secciones del juego completo. Normalmente solo una escena se encuentra activa al mismo tiempo, pudiéndose cambiar por otra tanto en el editor como en tiempo de ejecución, pero es posible cargar múltiples escenas al mismo tiempo.

Licencia y Plataformas

Unity es un programa de **código cerrado**, para poder utilizarlo es necesario adquirir la licencia de manos de Unity Technologies. Existen distintos tipos de planes de pago a la hora de adquirir la licencia, estos son:

1. **Licencia Personal**: Esta licencia está pensada para usuarios amateur o estudiantes. La licencia incluye acceso a toda la funcionalidad del motor junto con

¹⁹<https://docs.unity3d.com/ScriptReference/Renderer.html>

²⁰<https://docs.unity3d.com/ScriptReference/ AudioSource.html>

²¹<https://docs.unity3d.com/ScriptReference/ Animator.html>

²²<https://docs.unity3d.com/ScriptReference/ ParticleSystem.html>

²³<https://docs.unity3d.com/ScriptReference/ MonoBehaviour.html>

²⁴<https://docs.unity3d.com/Manual/CreatingScenes.html>

ciertos plugins como sistemas de publicidad y pago en la aplicación. Sin embargo, esta licencia no puede ser adquirida por ninguna entidad que más de 100.000\$ anuales, viéndose obligada a adquirir alguna de las otras licencias. Esta licencia es totalmente gratuita.

2. **Licencia Plus:** Esta licencia está pensada para estudios pequeños. Junto con toda la funcionalidad de la licencia personal, esta licencia incluye herramientas para gestionar métricas del juego y realizar estudios de rendimiento exhaustivos. Esta licencia también tiene una cifra de beneficios máxima, obligando a sus usuarios a adquirir la licencia pro si se superan los 200.000\$ de beneficio anuales. El precio de esta licencia es de 35\$ anuales.
3. **Licencia Pro:** Es la licencia para grandes estudios. Esta versión de la licencia cuenta con un mejor soporte para juegos multijugador, un mejor sistema para realizar análisis de métricas y acceso anticipado a las nuevas características, junto con toda la funcionalidad de las licencias anteriores. El precio de esta licencia es de 125\$ al mes.

Todas las licencias incluyen el editor de Unity con soporte para **Windows, Linux y Mac OS**. Independientemente de la plataforma de desarrollo elegida, Unity permite la exportación a más **20 plataformas distintas**: Dispositivos móviles (tanto Android como IOS), ordenadores de sobremesa, videoconsolas de última generación y web. Unity incluye herramientas para facilitar la conversión entre distintas plataformas. Juegos desarrollados con Unity como *Pac-Man 256* (Hipster Whale, 2015) (figura 2.20) están disponibles en múltiples plataformas gracias a esta característica.

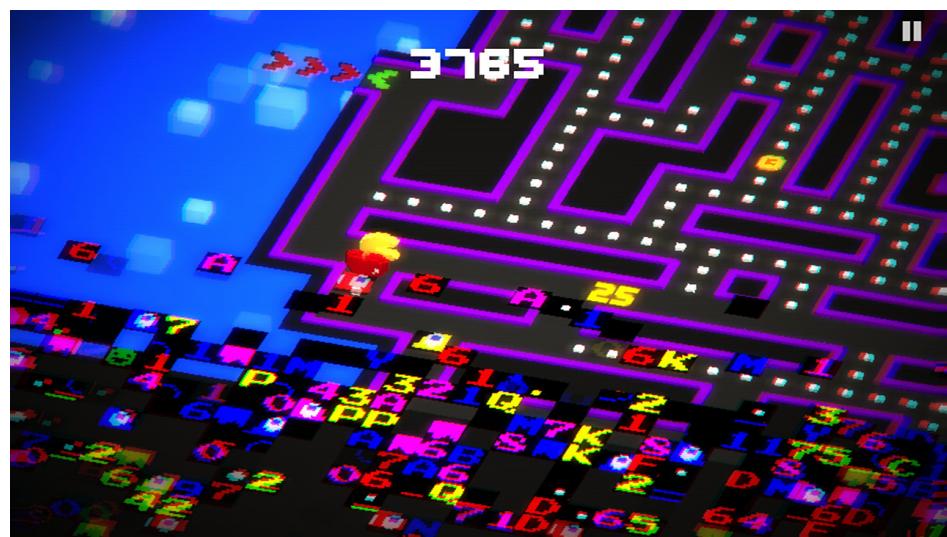


Figura 2.20: *Pac-Man 256* (Hipster Whale, 2015), disponible para PC, Android, IOS, PS4 y XBOX One (Fuente: página de Steam del juego).

2.4 Inteligencia Artificial en Videojuegos

2.4.1 Historia

Desde los principios de la inteligencia artificial, los expertos han dedicado una importante cantidad de tiempo y esfuerzo para construir sistemas inteligentes con el propósito de que pudiesen jugar a juegos con un nivel igual o superior al humano. Este enfoque en el estudio de los juegos se debe a que **ofrecen un campo de estudio ideal para la inteligencia artificial**: los juegos son problemas complejos que ofrecen desafíos para múltiples campos de la inteligencia artificial y además son tan populares que se dispone de cantidades inmensas de información sobre ellos [YT18].

Los primeros programas capaces de jugar contra humanos surgieron en los años cincuenta. Uno de los ejemplos más antiguos es el algoritmo ajedrecista de **Alan Turing** de 1948 [Tur53], el cual era “ejecutado” en papel por un humano que seguía manualmente los pasos del algoritmo. El primer Software capaz de jugar a un juego fue la inteligencia artificial para el juego *OXO* (Alexander S. Douglas, 1952). En 1959, **Arthur L. Samuel** [Sam59] programó una inteligencia artificial para jugar a las Damas la cual contaba con un sistema de aprendizaje y jugaba a nivel *amateur* [RN08].

Sin embargo, estos programas eran muy simples y no planteaban reto alguno contra jugadores experimentados cuando se trataba de juegos con una cierta complejidad, como el ajedrez. Hubo que esperar a los años noventa para que empezaran a surgir programas capaces de derrotar a grandes maestros de distintos juegos: en el año 1994 el programa **Chinook Checkers** derrotó al campeón mundial de la Damas **Marion Tinsley**²⁵ y 3 años más tarde el super-ordenador **Deep Blue** venció a gran maestro del ajedrez **Gary Kasparov**²⁶ (figura 2.21). Hoy en día, la inteligencia artificial ha demostrado ser capaz de superar a los jugadores humanos en casi cualquier juego, con ejemplos tales como la inteligencia artificial **Watson** ganando el concurso de televisión **Jeopardy** en 2011²⁷ o el programa **AlphaGo** derrotando a **Ke Jie**, el jugador número uno de Go²⁸ (un juego con una complejidad varios ordenes de magnitud superior al ajedrez).

Hoy en día, con el auge de los videojuegos, ha comenzado el estudio para la resolución de juegos continuos en tiempo real (en contraste a los juegos de mesa tradicionales, que son discretos). Estos juegos presentan un desafío mayor, pero ya se han realizado avances como el algoritmo desarrollado por **Google DeepMind** en 2014 para la resolución de varios juegos de la consola clásica Atari 2600²⁹.

²⁵<https://webdocs.cs.ualberta.ca/~chinook/project/>

²⁶<http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/>

²⁷https://www.youtube.com/watch?v=WFR3lOm_xhE

²⁸<https://www.theverge.com/2017/5/25/15689462/alphago-ke-jie-game-2-result-google-deepmind-china>

²⁹<https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/>



Figura 2.21: El Gran Maestro de ajedrez Gari Kaspárov (izquierda) enfrentándose al ordenador Deep Blue (derecha).

2.4.2 IA Aplicada a Videojuegos: Contexto Actual

El uso de la inteligencia artificial en la industria del videojuego difiere en varios puntos a su aplicación habitual en el campo académico de los juegos. La principal diferencia es el tipo de problema que se intenta resolver utilizando inteligencia artificial en cada una de las ramas: en la inteligencia artificial aplicada a jugar a juegos se busca obtener un sistema capaz de jugar de manera óptima para derrotar a cualquier adversario humano, sin embargo, en la inteligencia artificial orientada a videojuegos lo que se busca es crear sistemas con los que **mejorar la experiencia de juego del jugador**. Esta diferencia de objetivos hace que la inteligencia artificial no se aplique únicamente a la creación de oponentes virtuales, sino también al modelado del comportamiento de **personajes no jugadores** y a la **generación procedural de contenido** [YT18].

En el ámbito de los videojuegos, a la Inteligencia Artificial que interacciona con el juego como un jugador más suele llamarse **bot**. Estos bots predominan en juegos competitivos donde es necesario un oponente con un **grado de inteligencia elevado** para suponer un reto al jugador, como los juegos de estrategia, los juegos de lucha o los juegos de disparos en primera persona. Debido al elevado coste y complejidad de desarrollar una Inteligencia Artificial potente para estos juegos, por no hablar de la potencia requerida para ejecutarla, en la mayor parte de los videojuegos la inteligencia artificial “**hace trampas**”, juega teniendo acceso una serie de ventajas que los jugadores humanos no tiene. Estos sistemas podrían, por ejemplo, acceder a información oculta del jugador (como la posición o recursos) a la hora de planificar estrategias o dispondrían de más y mejores recursos que sus adversarios [YT18].

2. ESTADO DEL ARTE

En la mayoría de los juegos, la inteligencia artificial no se dedica al modelado de bots como los descritos anteriormente, sino que lo más común es que se utilice para controlar el comportamiento de **personajes no jugadores**, o NPCs (siglas inglesas de Non-Player Character). Estos pueden tener comportamientos muy variados dependiendo de su papel en el juego: pueden actuar como adversarios, servir de ayuda para el jugador, formar parte de un puzzle, contar una historia o simplemente formar parte del trasfondo de la acción. Dependiendo de la función asignada, la inteligencia artificial de un NPC puede variar desde una simple torreta que dispara a intervalos regulares hasta complejos sistemas de toma de decisiones como en *The Sims* (Maxis, 2000) (figura 2.22)



Figura 2.22: En *The Sims* (Maxis, 2000), los personajes pueden tomar decisiones basándose en sus gustos y necesidades.

Los NPCs en videojuegos se diseñan con dos objetivos en mente. En primer lugar, se busca crear una **ilusión de inteligencia**, de forma que los jugadores crean que el NPC es un ser inteligente con un comportamiento creíble, para que le resulte más fácil sentirse **inmerso** en la acción del juego. En segundo lugar, él debe buscarse hacer que el propio jugador se sienta inteligente al interaccionar con los NPCs. Esto se logra, especialmente cuando se trata de adversarios, haciendo que su comportamiento sea hasta cierto punto **predecible**, de forma que el jugador pueda desarrollar estrategias para enfrentarse/interaccionar con ellos [RN08].

La otra aplicación principal en los videojuegos es la **Generación Procedimental de Contenido**. La generación procedural de contenido es el nombre que reciben los métodos que permiten generar el contenido de un juego de forma automática o con

solo un mínimo de intervención humana. Actualmente, la mayor parte de los juegos que hacen uso de la generación procedimental la utilizan para la **creación automática de mapas** (como en el caso de *Minecraft* (Mojang, 2011) (ver figura 2.23)) u objetos (como por ejemplo en *Diablo II* (Blizzard Entertainment, 2000)). La generación procedural de contenido puede utilizarse tanto como una **herramienta durante el desarrollo**, que serviría para generar contenido que luego sería refinado por los desarrolladores; o podría formar parte del juego final, generando nuevo contenido al gusto del jugador de forma automática [YT18].



Figura 2.23: *Minecraft* (Mojang, 2011) es un ejemplo claro de generación procedural de terrenos.

2.4.3 Métodos de la IA

Existen varios tipos de métodos y algoritmos los cuales pueden ser utilizados para construir inteligencias Artificiales. La elección entre los distintos tipos de métodos debe realizarse dependiendo del tipo de problema que se intenta resolver y de los recursos de los que se dispone, tanto las prestaciones del dispositivo en el que van a ser implementados como margen de tiempo máximo de la ejecución del algoritmo [YT18].

Los Métodos de la inteligencia artificial pueden ser agrupados en las siguientes categorías, debido a sus características y aplicaciones similares:

Métodos Ad Hoc

Esta clase de métodos de IA es una de la primera y, en el sector del videojuego, la más común. Su nombre proviene de la locución latina que, traducida, significa “para esto”, y hace referencia a que se tratan de **soluciones precisas para problemas concretos**, las cuales no pueden generalizarse ni aplicarse a otros problemas distintos [YT18].

2. ESTADO DEL ARTE

Pese al notable problema que provoca la falta de reusabilidad de estos métodos, son los más utilizados en el desarrollo de videojuegos. Esto se debe a que, por lo general, son muy fáciles de diseñar, visualizar, implementar y depurar; además requerir de muy pocos recursos cuando se aplican a problemas pequeños.

El primero de estos métodos son las **máquinas de estados finitos** o FSM por sus siglas en inglés, el método dominante en la industria para el desarrollo de artificiales hasta mediados de los 2000 [YT18]. Este método se basa en dividir el comportamiento de la inteligencia artificial en varios **estados** independientes, cada uno con su propia lógica. Estos estados se encuentran conectados mediante **transiciones** las cuales cambian de un estado a otro si se cumplen determinadas condiciones.

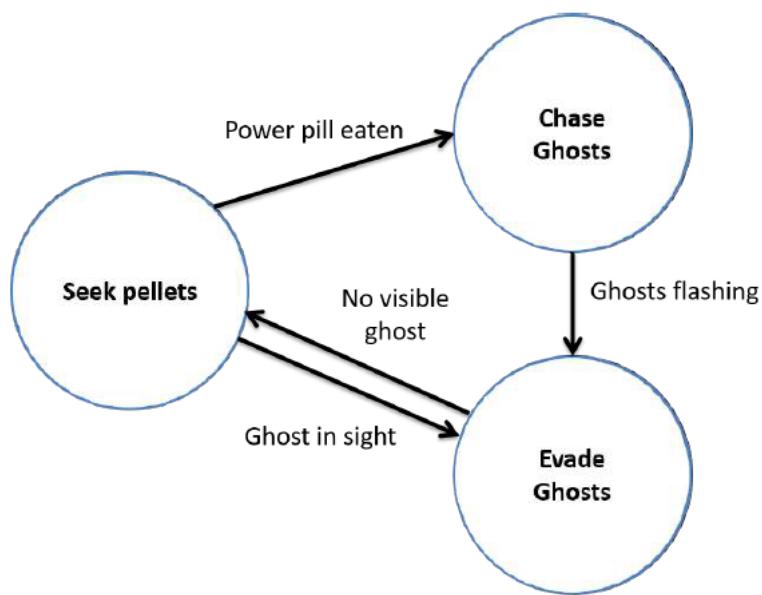


Figura 2.24: FSM de alto nivel de una IA jugadora de *Pac-Man* (Namco, 1980) (figura extraída de [YT18]).

La implementación de las máquinas de estados es **tremendamente simple, son fáciles de depurar, intuitivas y muy flexibles**, pudiéndose aplicar incluso a problemas no relacionados con la inteligencia artificial como la gestión de menús del juego [Dav15]. Sin embargo, su complejidad escala muy rápido con problemas grandes y además tienden a ser muy previsibles al carecer de capacidad de adaptación [YT18].

Una evolución directa de las máquinas de estados son los **árboles de comportamiento**. Este tipo de Inteligencias Artificiales ejecutan una secuencia de **comportamientos**, los cuales se encuentran ordenados en una estructura en árbol. El programa recorre el árbol en profundidad, ejecutando los comportamientos por los que pasa. Si la ejecución del comportamiento es *exitosa*, se prosigue con la ejecución, pero si es un *fallo*

entonces se reinicia la ejecución [YT18]. Aparte de los comportamientos, los arboles de comportamiento incluyen otros tres tipos de nodos:

- **Secuencia:** Este nodo ejecuta en secuencia el comportamiento de sus nodos hijos. Su ejecución será exitosa si lo fue la de todos sus nodos hijos.
- **Selector:** Este nodo selecciona de entre sus nodos hijos uno para ejecutar, basándose en algún tipo de filtro (los más comunes son los probabilistas o los basados en prioridad). Si falla la ejecución del hijo seleccionado, se puede elegir otro hijo o devolver un fallo.
- **Decorador:** Este tipo de nodo añade modificaciones a la ejecución de su nodo hijo. Las más habituales son las que repiten la ejecución mientras se cumpla cierta condición (tiempo transcurrido, número de ejecuciones...).

Los arboles de comportamiento **son mucho más flexibles** que las máquinas de estados, ya que es mucho más sencillo añadir y quitar comportamientos. Además, elementos como los nodos selectores probabilistas permiten diseñar mucho más predecibles. Este tipo de inteligencia artificial ya ha sido implementada en varios títulos comerciales como *BioShock* (2K Games, 2007) o *Halo 2* (Bungie Studios, 2004)³⁰.

El método Ad Hoc más reciente se llama **inteligencia artificial basada en Utilidad**. La idea principal de este método es que cada posible acción o estado de la inteligencia artificial tiene asignado un **valor de utilidad**, que representa como de útil es dicha acción o estado dada la situación actual del sistema. La utilidad se calcula a partir de la combinación de uno o varios **factores de decisión**, variables que representan el estado del mundo del juego.[Rab14].

Gracias a la facilidad para la creación de nuevos factores de decisión, y la facilidad para asociarlos con la utilidad, este método es mucho más **modular y extensible** que los métodos Ad Hoc anteriormente mencionados. Se trata de una técnica relativamente nueva en la industria, por lo que su uso aún no está muy extendido.

Algoritmos de búsqueda

Una de las bases de la inteligencia artificial son los **algoritmos de búsqueda**, dado que la mayor parte de los problemas de la inteligencia artificial podrían plantearse usando este tipo de algoritmos [YT18]. Los algoritmos de búsqueda se basan en, dado un **problema** y un **estado inicial**, encontrar una secuencia de acciones que permitan alcanzar la **solución** del problema partiendo del estado inicial.

Para encontrar la solución del problema, los algoritmos de búsqueda hacen uso de **árboles de búsqueda**, un tipo de grafo dirigido en el cual los **nodos** o hojas representan estados del problema, mientras que las **aristas** o ramas representan las acciones

³⁰https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php

2. ESTADO DEL ARTE

que provocan la transición entre dos estados. El agente inteligente recorre el árbol partiendo del **nodo raíz** (el estado inicial) buscando la secuencia de ramas que lleven a la solución del problema [RN08].

Partiendo de esta base, existen numerosos algoritmos de búsqueda distintos, los cuales utilizan diferentes estrategias a la hora de escoger de entre los distintos nodos del árbol cual debe ser analizado. Dependiendo del tipo de estrategia de selección, los algoritmos de búsqueda pueden agruparse en las siguientes categorías:

- **Búsqueda no informada:** Se trata de los algoritmos más básicos, en los que se realiza la búsqueda sin ningún tipo de información adicional sobre el objetivo. Sus variantes más simples son el algoritmo **primero en anchura** (explora todas las acciones de un estado antes de pasar al siguiente) y el **primero en profundidad** (explora una secuencia de ramas todo lo que puede antes de volver e intentar otra secuencia distinta)
- **Búsqueda primero el mejor:** En estos algoritmos se cuenta con información adicional sobre el nodo objetivo, la cual se utiliza para determinar que nodos deben explorarse primero. Un tipo de algoritmos Primero el Mejor son los algoritmos **A***, en los cuales los nodos se seleccionan basándose en su **coste** (suma de la distancia entre el nodo y el nodo raíz y la distancia estimada al objetivo).
- **Búsqueda MiniMax:** Este tipo de algoritmos se utilizan para resolver problemas que involucran a dos adversarios enfrentados. En estos algoritmos se va alternando entre jugadores **min** y **max**, los cuales intentan llegar a sus respectivos estados de victoria opuestos. El espacio de búsqueda de estos algoritmos suele ser muy grande, por lo que suelen usar funciones de evaluación para evitar recorrer el árbol de búsqueda completo.
- **Árbol de búsqueda de Monte Carlo:** Se trata de una familia de algoritmos diseñados para resolver problemas **No deterministas** y/o de **Información Imperfecta**. Para evaluar la calidad de un estado dado, el algoritmo utiliza simulaciones aleatorias de partidas a partir de ese estado. La siguiente acción será con la que empezó el mayor número de partidas victoriosas.

Cuando se aplican a juegos, los arboles de búsqueda suelen aplicarse para resolver dos tipos de problemas: para la búsqueda de caminos y para resolver juegos basados en turnos discretos.

Algoritmos de Optimización

Los **algoritmos de optimización**, a diferencia de la búsqueda en árbol, se centran en obtener una solución correcta, ignorando los pasos que llevaron a esta. Para ello, el algoritmo empieza tomando una solución sub-optima, la cual va modificando en

múltiples iteraciones utilizando una **función de Aptitud** como guía hasta obtener una solución con una Aptitud máxima[YT18].

Existen varios tipos de algoritmos de optimización, dependiendo de los métodos que elijan para la selección de la solución inicial, para la evaluación de aptitud o para la modificación:

- **Búsqueda local:** Este tipo de algoritmos consisten en, dada una solución, revisar todas las soluciones que difieran de dichas soluciones por una distancia mínima. Si alguna de las soluciones mejora a la solución original, se remplaza la solución original por la nueva solución y repite el algoritmo; si no, la solución original es elegida como la solución óptima.
- **Algoritmos evolutivos:** Los algoritmos evolutivos son un tipo de algoritmos de optimización basados en la evolución por **selección natural Darwiniana** que se observa en la naturaleza. Su funcionamiento se basa en un conjunto amplio de soluciones candidatas. EN cada iteración del algoritmo, las distintas soluciones se “cruzan” para obtener soluciones mixtas, las cuales son evaluadas. Finalmente, se forma un nuevo conjunto de soluciones a partir de las soluciones más exitosas. El algoritmo se repite hasta obtener la solución óptima.

Aprendizaje Automático

El aprendizaje automático es una rama de la computación que permite dotar a los ordenadores de la capacidad de **aprender** a realizar una tarea utilizando datos en lugar de programarlo específicamente para esa tarea [Sam59]. Los algoritmos de aprendizaje automático suelen aplicarse a problemas donde es muy difícil, o incluso imposible, programar un algoritmo implícito que pueda resolverlo, como por ejemplo el filtrado de correos o la visión por ordenador.

Una forma de clasificar los algoritmos de aprendizaje automático es basándose en el tipo de **Retroalimentación** que reciben. De esta forma, surgen las siguientes categorías:

- **Aprendizaje supervisado:** Se trata de algoritmos que extraer los atributos y características comunes de los integrantes de grupos etiquetados. Estos algoritmos funcionan recibiendo conjuntos de muestra formado por datos etiquetados en distintos grupos y categorías. Analizando los conjuntos de muestra, el algoritmo debe ser capaz de asignar nuevos datos a la categoría correcta. Existen muchos tipos de algoritmos de Aprendizaje supervisado, como por ejemplo las **Redes Neuronales**, que funcionan imitando la estructura de las neuronas del cerebro.
- **Aprendizaje por refuerzo:** Los algoritmos de aprendizaje por refuerzo se inspiran en el **conductismo psicológico**, basándose en la forma en la que los

2. ESTADO DEL ARTE

animales aprenden a tomar decisiones basándose en los estímulos positivos y negativos de su entorno. Estos algoritmos suelen implementarse mediante un agente inteligente que interacciona con un entorno mediante acciones. Con cada acción, el agente recibe un estímulo positivo o negativo, que almacena con la intención de descubrir la secuencia de acciones que maximice el estímulo positivo a largo plazo mediante técnicas similares a los algoritmos de optimización

- **Aprendizaje no supervisado:** El aprendizaje no supervisado son un conjunto de algoritmos que sirven para encontrar asociaciones y patrones en un conjunto de datos sin ningún tipo de información de refuerzo adicional. Existen varias aproximaciones a este tipo de algoritmos, como el **clustering**, que consiste en agrupar elementos de un conjunto basándose en su similitud entre ellos y su diferencia con el resto de los grupos.

La principal limitación del aprendizaje automático es la necesidad de convertir los datos de entrada a **representación de datos** interna que permita al algoritmo encontrar y clasificar los patrones correctamente. Ante la dificultad de elaborar las representaciones de ciertos tipos de información (como imágenes o el lenguaje natural) surgieron los algoritmos de **Deep Learning**, métodos de aprendizaje automático especializados en el aprendizaje de modelos de representación.

Los algoritmos de deep learning se caracterizan por utilizar **múltiples capas de unidades de procesamiento**. Cada capa recibe unos datos de entrada que procesa para generar convertirlos en un sistema de representación de más alto nivel, los cuales serán usados como entrada en la siguiente etapa. La estructura de capas no está diseñada por un ingeniero humano, sino que son generadas utilizando algoritmos de aprendizaje automático de propósito general, el cual puede ser tanto supervisados como no supervisados cite [Yan15].

2.4.4 Futuros campos de aplicación

La inteligencia artificial orientada a juegos evoluciona de manera paralela a los propios videojuegos, que viven ahora más que nunca una época dorada debido al aumento constante tanto en popularidad y prestigio, lo que supone una mejora tanto en la potencia de las máquinas que los ejecutan como en las técnicas que se utilizan en su desarrollo. Esta situación cambiante ha causado la aparición de nuevos problemas que se esperan poder solucionar con el uso de técnicas de inteligencia artificial [YT18].

Una las posibles aplicaciones de la inteligencia artificial es la de realizar **testing de juegos**. El programa jugaría a los juegos en busca de fallos tanto informáticos como de diseño (como problemas de balanceo o maneras de hacer trampas en el juego) de forma automatizada, ahorrando una gran cantidad de trabajo a los desarrolladores. Aunque ya existen herramientas que ofrecen una forma rudimentaria de testing automático, aun

es necesario mejorar aspectos como la categorización de los errores encontrados.

La **minería de datos de juego** es otro uso prometedor de la inteligencia artificial. Se trata de una técnica alternativa al testing habitual de juegos en la que se recolecta y analiza la **información de comportamiento de la base de jugadores** de un juego dado con el fin de mejorar dicho juego [Yan12]. Esta técnica se ha popularizado gracias a la proliferación de juegos con un fuerte componente online que facilita la recogida de datos. Sin embargo, los volúmenes de datos con los que se trabaja son tan masivos que los algoritmos de minería de datos actuales no son capaces de analizarlos completamente [Yan12].

La **dirección de juegos** consiste en una inteligencia artificial que modifica eventos del juego en tiempo real basándose en las reacciones de los jugadores con acciones tales como modificar la dificultad, reproducir música y sonido o modificar el entorno con tal de mejorar la experiencia del jugador. Actualmente, los juegos que mejor ha implementado un sistema con esta características es la saga Left 4 Death de Valve³¹. Se trata de una rama con mucho potencial que aún no ha sido explorado completamente.

Finalmente, una tarea de gran importancia para los juegos online, a pesar de no formar parte de los juegos en sí, es la **motorización de los chats**. El gran volumen de mensajes que se envían a través de los chats de los juegos online más populares hace que sea imposible la moderación manual, lo que crea un entorno de juego toxico para los jugadores (ejemplo en la figura 2.25). Compañías como Riot Games están empezando a utilizar algoritmos de aprendizaje automático para entrenar sistemas para detectar y eliminar los mensajes inapropiados de los chats³².



Figura 2.25: Ejemplo de comportamiento tóxico en *League of Legends* (Riot Games, 2009).

³¹<http://www.l4d.com/blog/>

³²<https://www.nature.com/news/can-a-video-game-company-tame-toxic-behaviour-1.19647>

Capítulo 3

Arquitectura

Este apartado describe la arquitectura software de **Virus Breaker**. La descripción comenzará con una recapitulación de los eventos con los que un jugador se encontrará cuando inicie una sesión de juego. A esta descripción le seguirá a un análisis técnico de la estructura interna del juego, comenzando por las **escenas** en las que este se divide y terminando con un análisis de cada uno de los **objetos** que intervienen en el desarrollo de la acción.

3.1 Descripción del Juego

En Virus Breaker (figura 3.1), el jugador controla a un personaje encerrado en una sala cubica. El objetivo del juego es el de **salir de la sala abriendo una enorme puerta** que ocupa la totalidad de la pared norte de la sala. Para lograrlo, el jugador debe **golpear una pelota** usando una paleta para golpear con ella la puerta. Tras varios golpes, la puerta se abrirá. Sin embargo, la puerta estará cubierta por un **muro de ladrillos** que el jugador deberá romper primero para poder alcanzar la puerta. Por otro lado, si la pelota golpea tres veces consecutivas el suelo, el jugador perderá la partida.

El juego contendrá varias **salas**. Las salas tendrán unas dimensiones constantes, pero distintas configuraciones de bloques cubriendo la puerta, y distintos valores de “puntos de vida” para la puerta. Cuando el jugador abra la puerta de una sala, el personaje saldrá de ella y se moverá a otra sala distinta. Las salas están **ordenadas con dificultad creciente**, de modo que el jugador se encontrará primero con las salas más sencillas antes de tener que enfrentarse a las más complicadas.

La última sala del juego contendrá un **jefe final** al que el jugador deberá derrotar para completar el juego. Este jefe es un enemigo móvil que, adherido a la puerta de la sala, intentará bloquear la pelota para evitar que esta golpee la puerta. El jefe se comportará de **forma natural** para que el jugador tenga la impresión de que se está enfrentando a un oponente inteligente.

Adicionalmente a la jugabilidad principal, el juego contará una serie de **menús** que servirán de puente entre las distintas sesiones de juego. Estos menús son: la **pantalla**

3. ARQUITECTURA

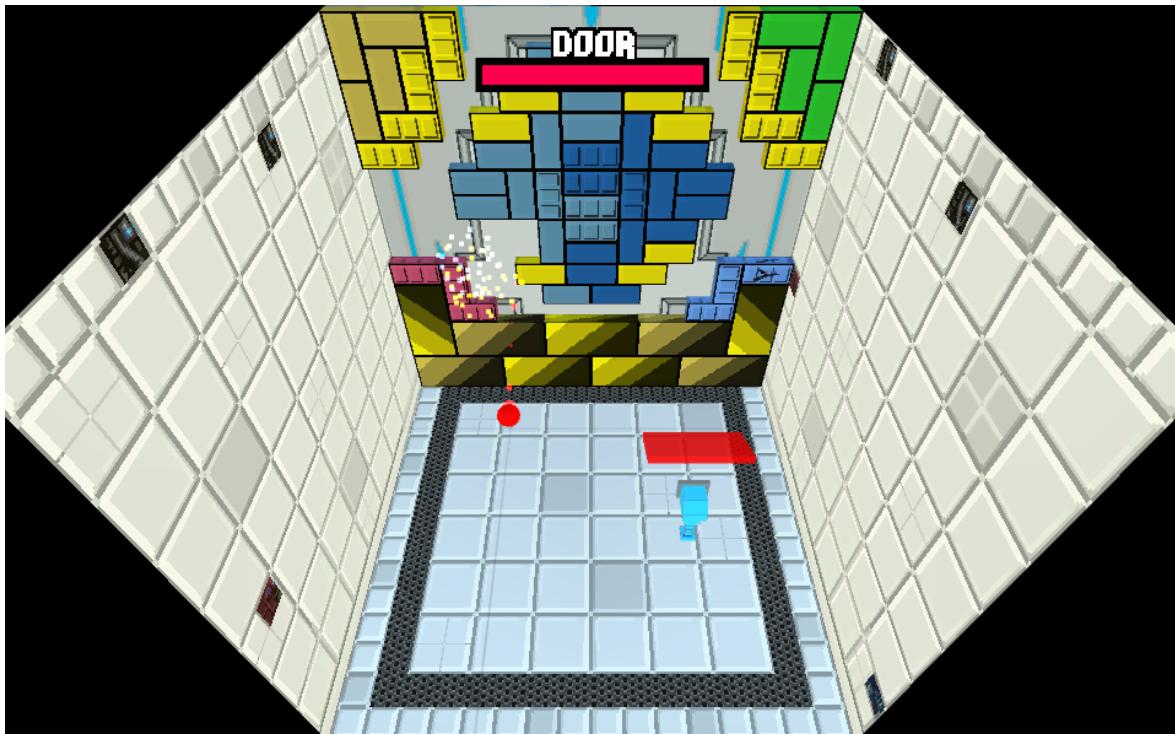


Figura 3.1: Captura de pantalla de Virus Breaker.



Figura 3.2: Pantallas de victoria y derrota.

de título, que se muestra al principio del juego, la **pantalla de fin del juego** que aparece cuando el jugador es derrotado y la **pantalla de victoria**. En la figura 3.2 pueden verse las pantallas de victoria y fin de juego.

El **progreso del jugador** en el juego se guarda entre partidas, de forma que, si el juego se cierra, el jugador pueda empezar por el mismo nivel en el que se encontraba la siguiente vez que inicie el juego. Cuando se derrota al jefe final, la partida guardada se elimina para permitir que el jugador pueda volver a jugar desde el principio si lo desea.

Como programa informático, Virus Breaker constará de un **archivo ejecutable**,

acompañado de una carpeta que almacena ficheros complementarios. Para iniciar el juego, el jugador deberá simplemente iniciar el archivo ejecutable, sin necesidad de ningún tipo de instalación previa.

3.2 Visión General de la Arquitectura

La arquitectura de este proyecto está basada en la **arquitectura de componentes** de Unity, en la que el comportamiento y propiedades de los objetos está determinado por los **componentes** que estos tengan adheridos. Sin contar con los componentes incluidos de serie en Unity3D, para este proyecto se implementaron **17 componentes**.

Para implementar los componentes se utilizó el lenguaje de programación **C#** debido a que es el lenguaje más utilizado en el desarrollo con Unity3D y por tanto es para el que existen un mayor número de recursos como librerías o tutoriales. Los componentes desarrollados son **clases** de C# que heredan de la clase **MonoBehaviour**, clase abstracta que implementa la funcionalidad común a todos los componentes. En adición a los componentes, también se implementaron dos **clases de soporte** adicionales para facilitar el funcionamiento de ciertos componentes.

Las clases elaboradas para el proyecto pueden agruparse en varios **grupos funcionales** según su papel dentro del proyecto. En la figura 3.5 pueden verse la organización de las clases en sus respectivos grupos.

Los grupos funcionales son los siguientes:

- **Menu:** Se trata de clases involucradas la gestión de los menús del juego. La función de estas clases es la de leer la entrada del jugador durante las escenas de menús e iniciar la transición a las escenas pertinentes.
- **Level Control:** En este grupo se encuentran las clases implicadas en la correcta ejecución del nivel. El grupo incluye el generador de niveles, que carga la configuración del nivel desde archivo, y los componentes encargados de gestionar que los eventos de la partida se ejecuten en el orden correcto.
- **Player Control:** Este grupo incluye una única clase: el comportamiento del jugador. La clase lee la entrada del jugador y ejecuta las acciones correspondientes del personaje principal.
- **Collision and Interaction:** En este grupo se encuentra el comportamiento de la pelota y el de los objetos que reaccionan a su colisión.
- **Boss Control:** Este grupo contiene las clases que determinan el comportamiento del jefe final del juego.

3. ARQUITECTURA

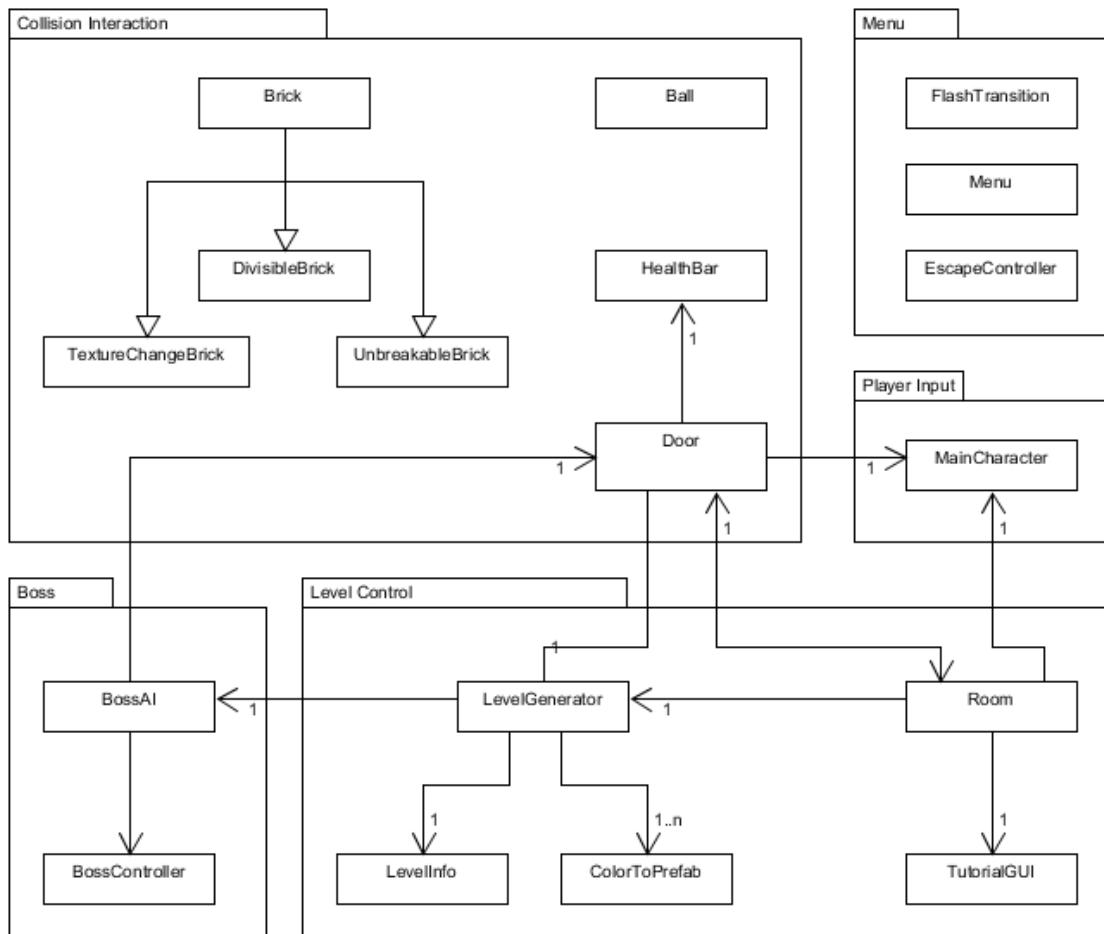


Figura 3.3: Diagrama de clases.

3.3 Menú

Virus Breaker cuenta con cuatro **escenas**: La **escena de juego**, que es donde ocurre toda la acción y tres **escenas de menú**. Las escenas de menú sirven como intermedio entre sucesivas sesiones de juego y permiten que el jugador se pueda prepararse antes de empezar con el juego. Esta división se eligió por encima de utilizar una única escena debido a que permite llevar un **control implícito del estado del juego**. Si todo el juego se desarrollase en la misma escena, sería necesario implementar sistemas adicionales para controlar que los comportamientos de los objetos solo se ejecuten en los momentos adecuados.

El comportamiento del juego en estas tres escenas es prácticamente es muy básico: El juego muestra un texto animado de gran tamaño, acompañado de un segundo texto (también animado) con instrucciones para el jugador. Si el jugador pulsa la tecla **espacio**, la escena cambiará a la escena de juego. Los tres menús son los siguientes:

- **La escena de título.** Esta escena aparece nada más iniciar el juego. Esta escena

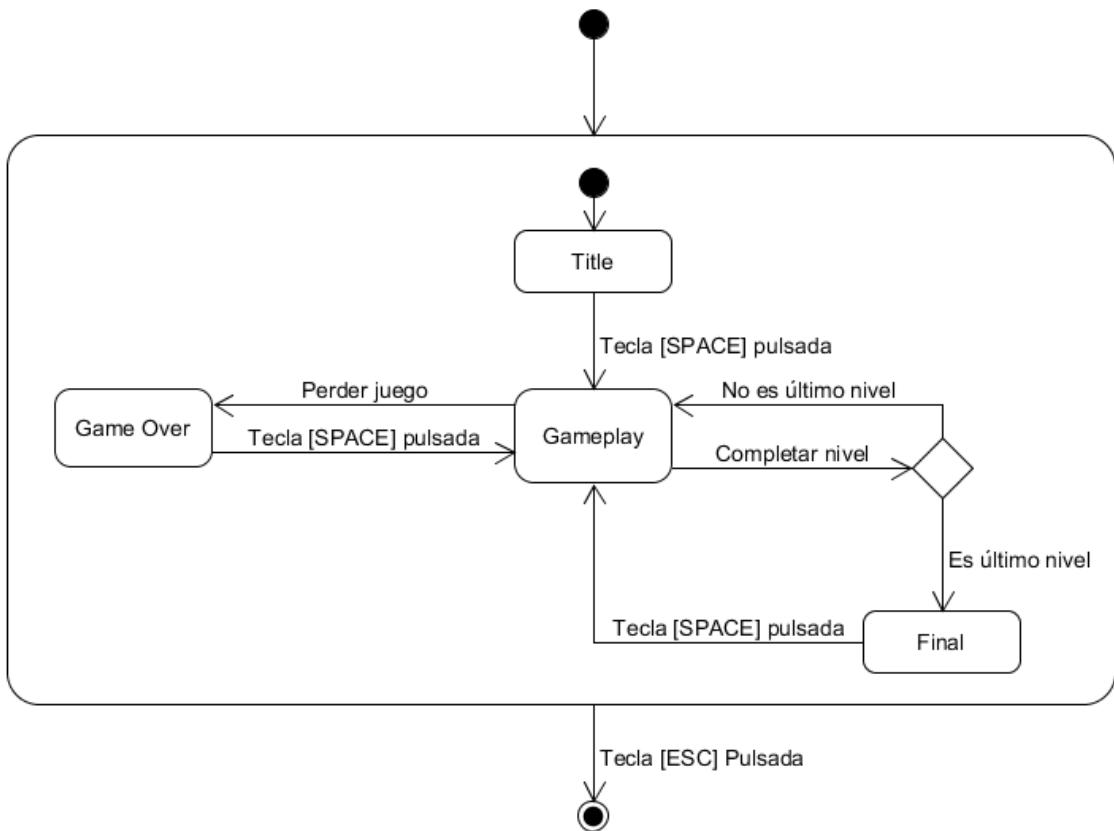


Figura 3.4: Diagrama de escenas.

muestra el título del juego, el nombre del desarrollador.

- **La escena de fin del juego.** Esta escena aparece cuando el jugador pierde durante la partida.
- **La escena de fin del juego.** Esta escena aparece cuando el jugador supera todos los niveles del juego.

En la figura 3.4 se pueden ver las transiciones entre las distintas escenas a modo de diagrama de estados.

Las clases que gestionan el comportamiento de los menús y otros comportamientos ajenos a la jugabilidad se encuentran el grupo funcional Menú. Se trata de un grupo pequeño, aislado del resto de grupos funcionales ya que su funcionalidad es exclusiva de los menús. Las clases de este grupo son **Menu**, **EscapeController** y **FlashTransition**. En la figura 3.5 se puede ver un diagrama de las clases de este grupo.

La clase primera clase, **Menu**, se encarga de controlar el comportamiento de los menús. Esta clase lee la entrada del jugador durante el método **Update** a la espera de que este pulse la tecla espacio. Cuando detecta la pulsación, la clase inicia el cambio de escena. Esta clase es un componente asociado al texto que muestra las instrucciones.

3. ARQUITECTURA

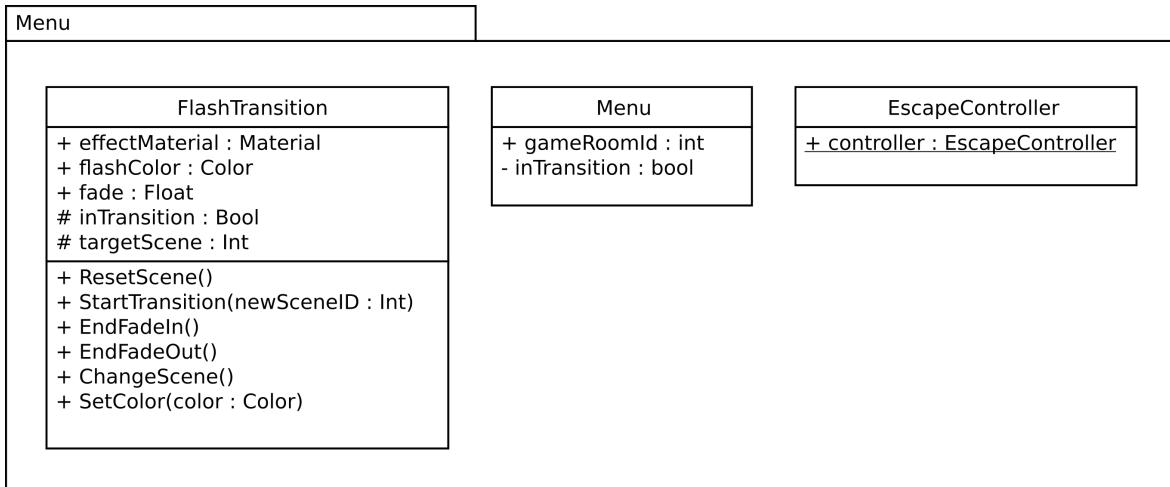


Figura 3.5: Diagrama del grupo funcional **Menu**.

El cambio de escena se lleva a cabo mediante la clase **FlashTransition**. Esta clase es un componente asociado a la cámara de la escena (el objeto encargado de renderizar la escena) que se encarga de producir una transición suave entre dos escenas. La transición se activa el método público **StartTransition**, a través del cual el componente recibe la escena de destino. El componente aplica un fundido en negro a la escena para a continuación cargar la escena de destino mediante el **SceneManager** el gestor de escenas de Unity. En la escena destino, una cámara idéntica produce el efecto a la inversa.

El fundido en negro se implementa aplicando una textura semitransparente sobre la imagen capturada por la cámara, cuyo grado de transparencia cambia con el tiempo gracias a un componente **Animator**. En la figura 3.6 puede verse la configuración de componentes de la cámara.

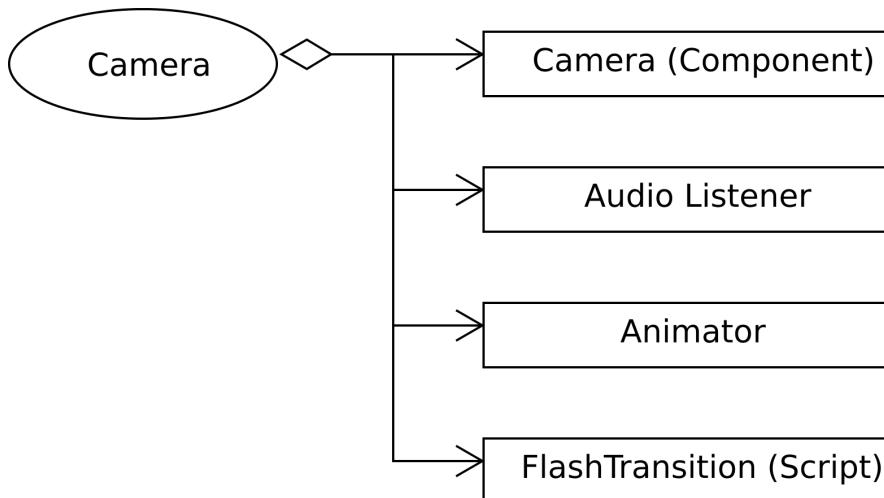


Figura 3.6: Componentes de la cámara.

Para la animación de los textos también se utilizó un componente Animator que variaba con el tiempo el ángulo y tamaño de estos. Los textos pequeños fueron creados mediante el sistema de interfaz gráfica de usuario de Unity, pero los textos grandes fueron modelados en **Blender** y luego importados en Unity.

Adicionalmente al controlador de menús se encuentra la clase **EscapeController**. Esta clase permite al jugador cerrar el juego en cualquier momento pulsando la tecla **escape**. La clase es un componente asociado a un objeto vacío en la **escena de título** el cual persiste entre escenas gracias al método de Unity **DontDestroyOnLoad**. Para evitar que varias instancias de este controlador estén presentes, lo que podría provocar errores, se utiliza el patrón de diseño **Singleton**.

3.4 Control de niveles

La acción del juego ocurre principalmente en la **escena de juego**. Esta escena contiene la **sala**, el escenario donde tiene lugar el juego, así como los diversos objetos que forman el juego como el personaje principal o la pelota. Aunque la mayor parte de la acción viene dada por la interacción de los objetos contenidos en la sala, la propia sala también tiene su propio comportamiento, indispensable para la ejecución correcta del juego.

Esta sala es un conjunto de GameObjects agrupados de forma jerárquica (figura 3.7), de forma que los diferentes objetos que la conforman tienen una funcionalidad distinta. Los GameObjects son:

- **Room:** Es el objeto principal o “padre” que contiene a los demás. Contiene tres componentes: un **Audio Source**, el script **LevelGenerator** y el script **Room**.
- **Walls:** Se trata de cuatro rectángulos que forman las tres paredes de la sala y el techo (que es funcionalmente idéntico a las paredes). Estos GameObjs tienen dos componentes: Un **MeshCollider** y un **Renderer**.
- **Floor:** El suelo de la sala es similar a las paredes (un GameObject con un **Mesh-Collider** y un **Renderer**), pero está marcado de forma diferente para activar un comportamiento distinto en la pelota.
- **Door:** La puerta de la sala ocupa la pared norte de la sala.

La función principal de la sala es la de gestionar la ejecución del resto de objetos, de forma que su inicialización y destrucción se ejecuten de forma correcta y sin solapamiento. Gracias a la sala, la ejecución de la escena del juego se divide en cuatro etapas bien diferenciadas:

- **Carga del nivel:** La configuración del nivel actual es cargada desde archivo y aplicada a la sala.

3. ARQUITECTURA

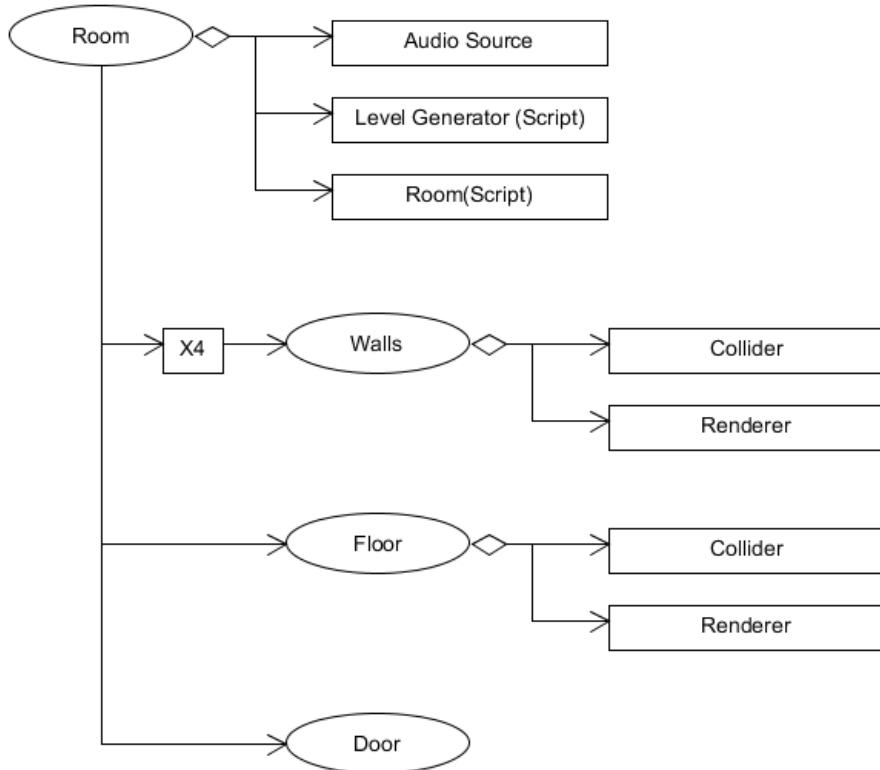


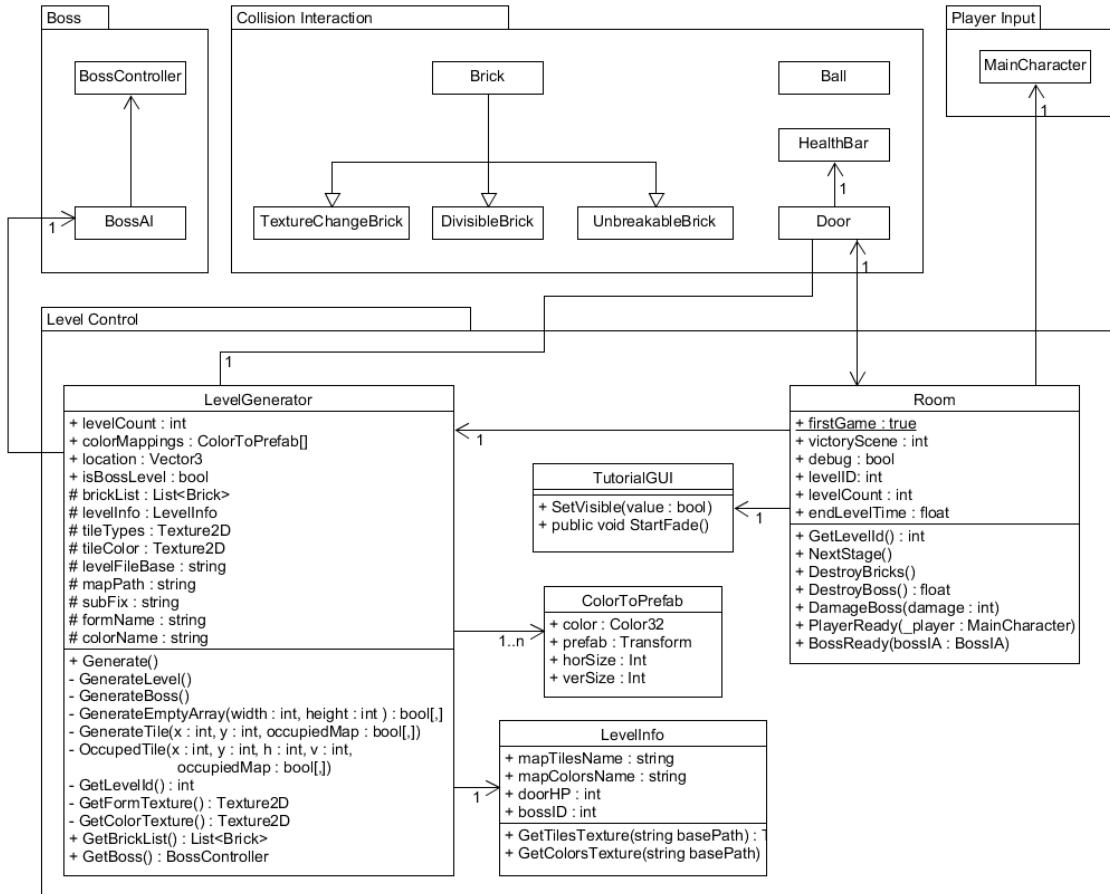
Figura 3.7: Diagrama de componentes de la sala.

- **Introducción:** Los objetos de la sala realizan su inicialización.
- **Juego:** El jugador toma el control del personaje principal.
- **Fin del juego:** Los objetos realizan su finalización. Esta etapa puede tener dos variantes: la **victoria**, en la que el jugador supera el nivel y pasa al nivel siguiente y la **Derrota**, en la que el jugador pierde y es enviado a la escena de fin del juego.

El comportamiento de la sala está definido mediante las clases contenidas en el Grupo funcional **Level Control**. Las clases de este grupo son **LevelGenerator** y **Room**, dos componentes que se asocian al objeto principal de la sala; **ColorToPrefab** y **LevelInfo**, dos clases de soporte para el generador de niveles y **TutorialGUI**, la clase encargada del comportamiento del tutorial del juego. Un diagrama de este grupo puede verse en la figura 3.8.

3.4.1 Carga de Niveles

La primera etapa de la ejecución de la escena de juego es la carga del nivel. En esta etapa, la escena, a través de su componente **LevelGenerator** accede a una serie de archivos que contienen la configuración del nivel. El componente se encarga de leer dicha información y usarla para generar los ladrillos del nivel en su configuración correcta, asignar los puntos de vida a la puerta del juego e iniciar el jefe final en caso

Figura 3.8: Diagrama del grupo funcional **Level Control**.

de que se tratase de un nivel de jefe.

Implementación

La información de un nivel del juego se encuentra almacenada en forma de tres archivos: un fichero **JSON** y dos imágenes **.PNG** de 16X16 píxeles. Las imágenes PNG codifican propiedades de los ladrillos en la sala en sus píxeles, los cuales representan las posibles posiciones en las que un ladrillo puede estar colocado sobre la puerta de la sala. La primera de las imágenes determina a través del color de sus píxeles el tipo de ladrillo que debe colocarse en cada posición de la puerta, mientras que la segunda imagen determina el color del ladrillo colocado en dicha posición. El archivo JSON, por otro lado, contiene la información adicional del nivel, puntos de vida y si se trata de un jefe, y la dirección a las dos imágenes.

La lógica para la carga de niveles se encuentra en el componente **LevelGenerator** de la sala. Este componente tiene como atributos la información necesaria para la carga de nivel como índice del nivel actual, formato del nombre de los archivos, la ruta a la carpeta de donde se guardan los archivos o coordenadas del punto de aparición.

3. ARQUITECTURA

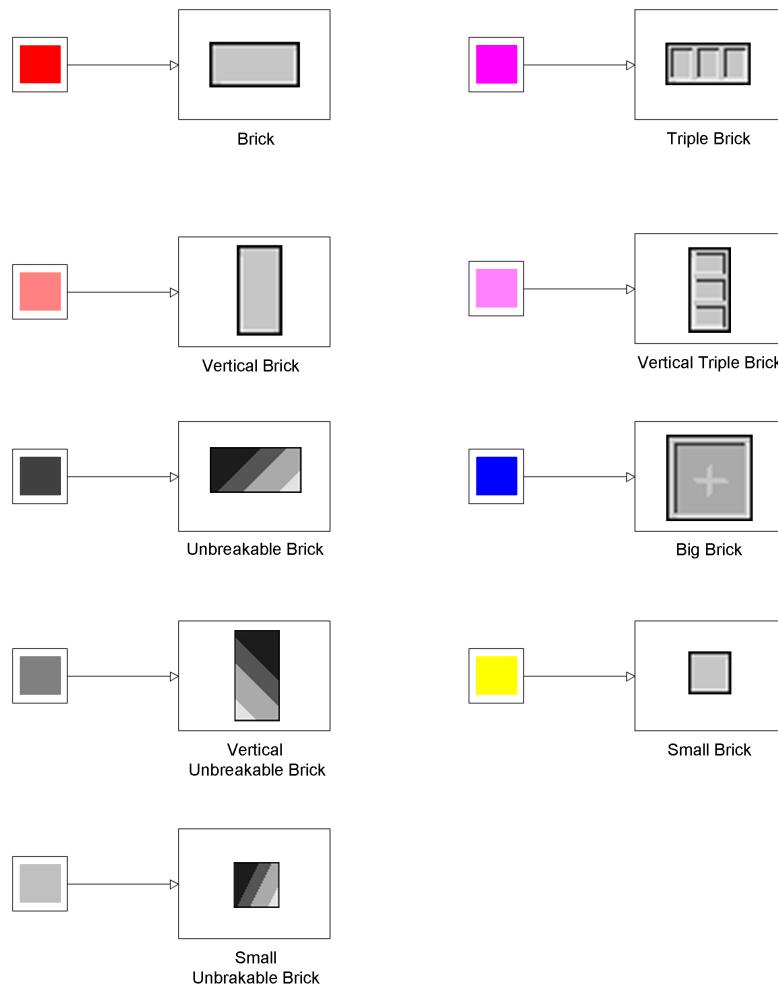


Figura 3.9: Listado de ladrillos prefabricados y sus colores asociados.

El componente contiene una lista de objetos de la clase **ColorToPrefab** que sirve para relacionar cada posible color con un tipo de ladrillos. La clase ColorToPrefab es muy sencilla, estando formada por cuatro atributos: el color, el objeto prefabricado y dos enteros que determinan la longitud del prefabricado. En la figura 3.9 se pueden ver la lista de todos los ladrillos prefabricados junto a sus colores correspondientes.

Para iniciar la carga del nivel se utiliza el método público **Generate** de la clase. El método sigue el siguiente procedimiento para realizar la carga del nivel:

1. El componente **lee archivo JSON** determinado por el índice del nivel actual utilizando la clase **Resources**¹ de Unity, que permite acceder a assets guardados en una carpeta especial del proyecto.
2. Usando la clase **JsonUtility**² de Unity, la información del archivo es convertida

¹<https://docs.unity3d.com/ScriptReference/Resources.html>

²<https://docs.unity3d.com/ScriptReference/JsonUtility.html>

en un objeto de la clase **LevelInfo** lo que facilita su manipulación.

3. Se **determina si se trata de un nivel de jefe** o no. En caso afirmativo, se detiene el proceso y se empieza a preparar la batalla con el jefe. En caso contrario, se continua con el proceso.
4. Se cargan **las imágenes del nivel** en forma de dos texturas de la clase Texture2D. A efectos prácticos, cada textura es una matriz bidimensional de objetos de clase Color.
5. Se instancia una **matriz de ocupación**, una matriz bidimensional booleana con las mismas dimensiones que las texturas, la cual sirve para marcar que porciones de la puerta están cubiertas por ladrillos.
6. Se recorre la primera textura **obteniendo los colores de sus píxeles**. Si la posición correspondiente en la matriz de ocupación está inactiva se realizan las siguientes acciones:
 - a) El color seleccionado se compara con todos los elementos de la lista de objetos **ColorToPrefab**. Si alguno de los ColorToPrefab contiene el color seleccionado, se instancia un ladrillo del tipo determinado. Si no hay coincidencia, se pasa al siguiente pixel.
 - b) Se accede al color del pixel de la posición actual de la segunda textura. Este color se asigna al ladrillo.
 - c) Se marca la posición del ladrillo en la matriz de ocupación. Dado que los ladrillos pueden ocupar más de una “casilla”, se marcan también las casillas colindantes basándose en los tamaños almacenados en el objeto **ColorToPrefab**.
7. Se asigna **el número de golpes** que requiere la puerta.

En la figura 3.10 pueden verse de forma gráfica los pasos de ejecución del método

Justificación

Cuando empezó el desarrollo del sistema de carga de niveles se elaboró una lista de requisitos que el editor de niveles elegido debía cumplir para resultar útil en el desarrollo. Los requisitos eran los siguientes:

- **Precisión** a la hora de colocar y alinear los bloques. El editor debe ser capaz de posicionar los ladrillos en una cuadricula de forma automática (para así mantener la estética de juego deseada).
- Posibilidad de elegir individualmente el **color** para cada uno de los bloques del nivel, independientemente de su comportamiento o posición. Esto permite crear dibujos con los bloques, lo que hace los niveles más memorables (ejemplo en la figura 3.11).

3. ARQUITECTURA

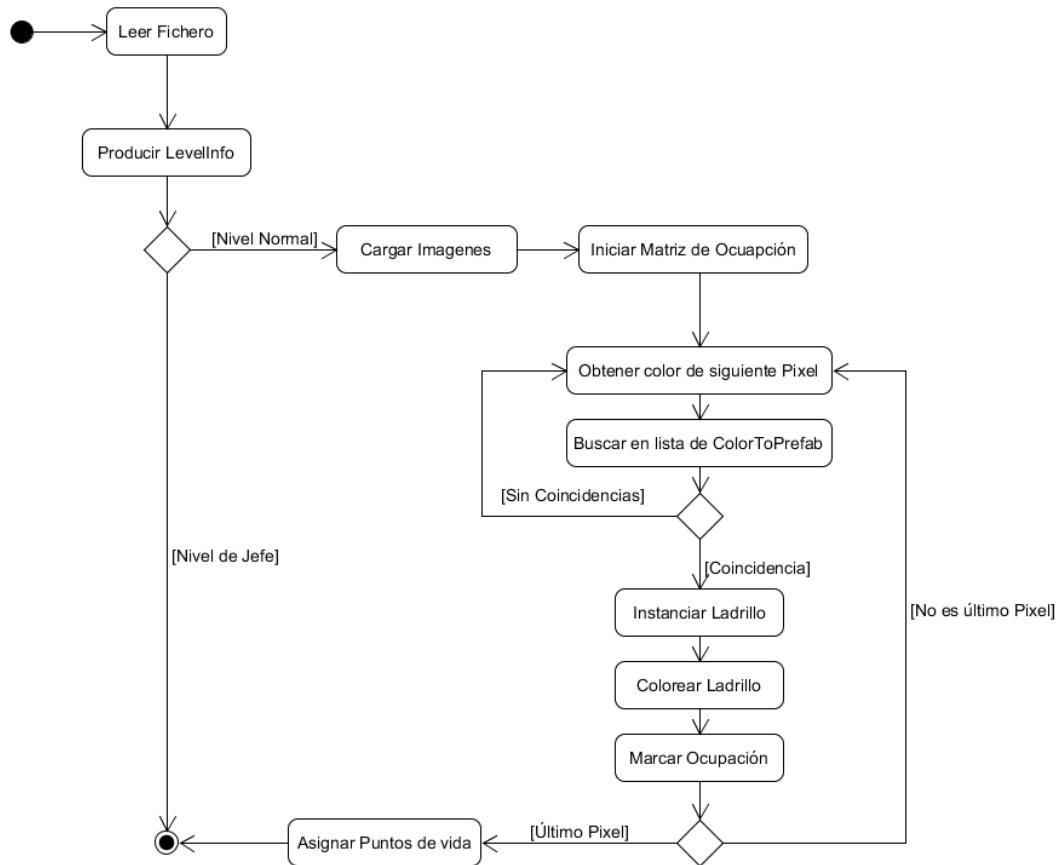


Figura 3.10: Diagrama del proceso de generación de niveles.

- **Simplicidad** en el proceso de creación. La creación de niveles debe de ser rápida, para facilitar la iteración el diseño.

El componente LevelGenerator **cumple de sobra los requisitos exigidos**. El sistema está completamente discretizado, por lo que los ladrillos siempre estarán colocados en sus posiciones correctas; la selección de colores es intrínseca al sistema y la producción de niveles es muy rápida, puesto que nuevas imágenes pueden ser generadas con facilidad en cualquier programa de diseño gráfico. Adicionalmente, la implementación del sistema es capaz de ignorar pequeños errores en el mapa, como asignación de colores a casillas vacías o la presencia de colores que no se corresponden con ningún tipo de ladrillos, lo que reduce aún más los tiempos de producción.

El fichero JSON se añadió a la estructura de niveles como sistema para “localizar” las dos imágenes que forman el nivel y también para almacenar información adicional que no pudiese almacenarse de forma intuitiva como otra imagen. El formato de este archivo se eligió para aprovechar la clase **JsonUtility** de Unity, que permite construir objetos C# a partir de su representación en JSON de forma instantánea.

Sin embargo, el sistema por imágenes tiene una serie de limitaciones que hay que

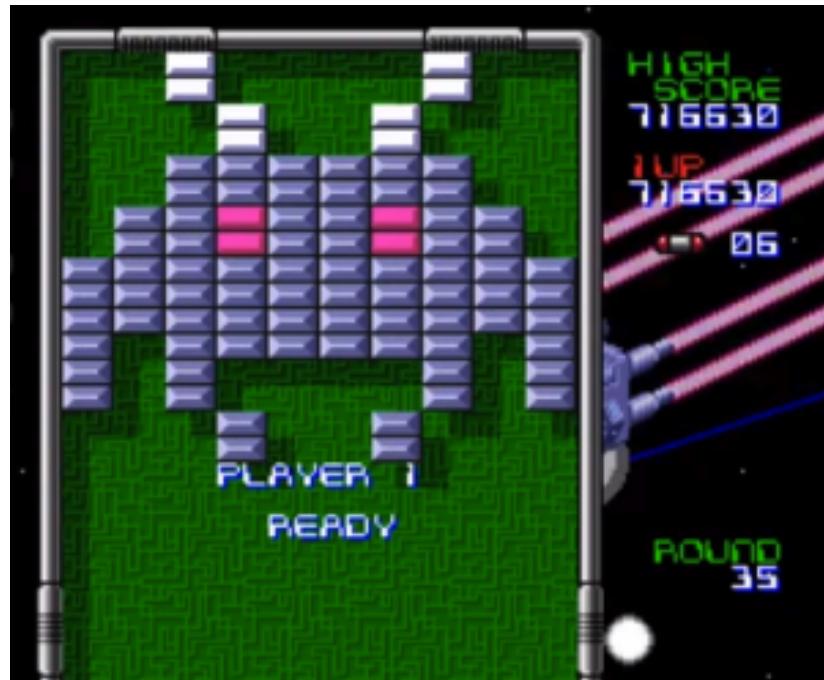


Figura 3.11: Homenaje a *Space Invaders* (Taito, 1978) dentro de *Arkanoid: Doh It Again* (Taito Corporation, 1997).

tener en cuenta durante el diseño de niveles:

- En primer lugar, **solo una pequeña cantidad de colores pueden ser asignados a tipos de ladrillos**. Esto se debe a las discrepancias entre los formatos de colores (los componentes RGB de los colores se guardan como enteros entre 0 y 255 en las texturas, pero como números de punto flotante entre 0 y 1 en la clase Color), la cual provoca perdida de información durante la conversión y nos fuerza a utilizar valores de color que sean fracciones exactas de 256.
- En segundo lugar, el sistema no contempla la posibilidad de **rotar los ladrillos**. Debido a que esta funcionalidad era necesaria para la construcción de algunos niveles, fue necesario crear ladrillos prefabricados previamente rotados.
- En segundo lugar, se trata de un sistema poco escalable, permitiendo un tamaño único de campo de juego y poca capacidad para “personalizar” los tipos de bloque más allá de su color; por lo que sería necesario realizar cambios importantes en caso de que se necesitaran construir niveles más complejos

La solución ideal sería implementar un **editor de niveles** específico para este juego, el ofrezca la funcionalidad deseada y además pueda modificarse o extenderse en caso de que los requisitos cambien. Sin embargo, teniendo en cuenta el alcance del proyecto, el coste la producción de una herramienta así no era rentable.

3.4.2 Inicio del juego

Cuando la escena de juego es cargada, el componente **Room** de la sala se encarga de coordinar la inicialización del juego. Este tiene asociadas referencias a la mayor parte de los objetos de la escena, lo que le permite llamar a los métodos correspondientes con facilidad. La secuencia de eventos del inicio del juego es la siguiente:

1. **Generación del nivel:** El componente Room invoca el método **Generate** del componente **LevelGenerator** iniciando la carga del nivel. Se trata de un proceso lento, por lo que puede tardar unos segundos. Sin embargo, la ralentización en la ejecución es camuflado con el efecto de cambio de escena.
2. **Entrada del jugador:** El **Personaje Principal** entra en la sala a través de la pared sur. El personaje principal inicia esta acción automáticamente al cargarse la sala y una vez alcanza el centro de la sala, se detiene y realiza invoca al método **PlayerReady** de Room para notificar que su entrada ha finalizado.
3. **Entrada del jefe:** Si se trata de un nivel de jefe, el jefe comienza su entrada en la sala. Al igual que el jugador, el jefe invoca al método **BossReady** de Room una vez su animación ha terminado.
4. **Activación de la pelota:** Una vez que el jugador (y el jefe) han finalizado su entrada, Room invoca el método **Activate** de la pelota, provocando que se vuelva visible y comience con su movimiento.
5. **Control del jugador:** Inmediatamente después de activar la pelota, el componente Room invoca el método **Activate** del personaje principal, el cual empieza a reaccionar a las órdenes del jugador comienza el juego.

Durante el inicio del nivel, se le muestra al jugador un **mensaje en pantalla con las instrucciones** del juego (figura 3.12). Este mensaje está implementado mediante el sistema de **interfaz de usuario**³ de Unity, el cual permite añadir con facilidad textos, imágenes y otros elementos bidimensionales a la escena de forma que siempre sean visibles por la cámara. En componente, **TutorialGUI** se encarga de hacer desaparecer el tutorial después de un tiempo. Además, el componente se asegura de que el tutorial solo aparece durante el **primer nivel al que se juegue en la sesión de juego**, de forma que no resulte molesto al jugador.

3.4.3 Condiciones de fin del juego

Una vez finalizada la introducción, se da paso a la etapa de **Juego**. Esta etapa continua hasta que se alcanza un **estado de victoria o de derrota**, en los cuales el componente Room de la sala ejecuta una serie de eventos de fin del juego que culminan en un cambio de escenas.

³<https://docs.unity3d.com/Manual/UISystem.html>

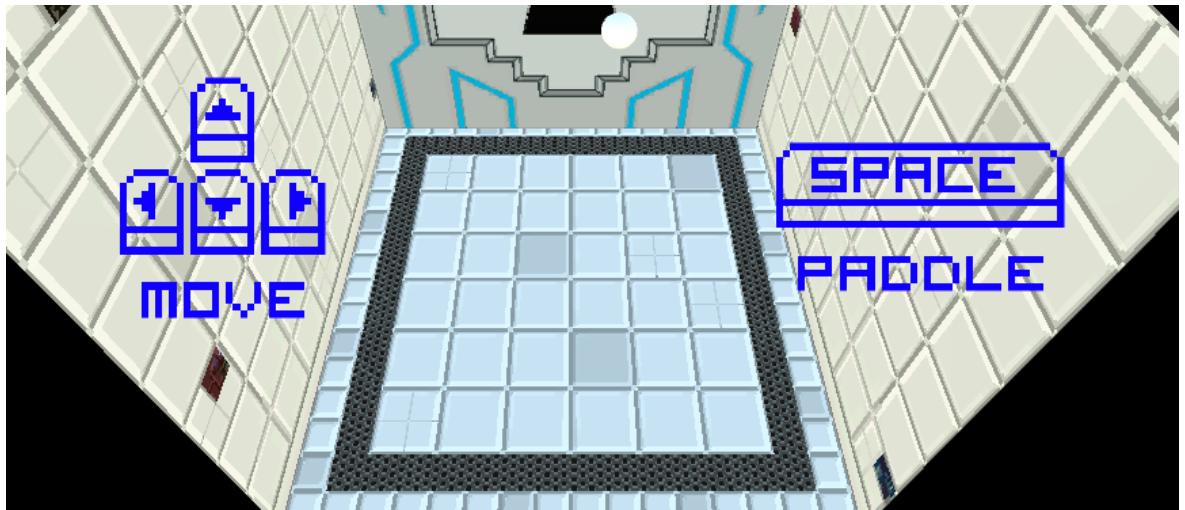


Figura 3.12: Texto tutorial.

El **estado de victoria** tiene lugar cuando la puerta de la sala se queda sin puntos de vida. Cuando esta condición se cumple, empieza la siguiente secuencia de acciones:

1. **Ocultar pelota:** Cuando la pelota golpea la puerta y esta no le quedan más puntos de vida, la pelota detiene su movimiento y desactiva tanto su collider como su renderer, provocando que se vuelva invisible e intangible.
2. **Animación de la puerta:** Al recibir un golpe y quedarse con cero puntos de vida, la puerta inicia su animación de apertura.
3. **Romper los ladrillos:** Cuando acaba la animación de la puerta, esta llama al método **DestroyBricks** del componente Room, lo que provoca que los ladrillos aun en pie sean destruidos.
4. **Salida del personaje principal:** Una vez finalizada la animación de la puerta, esta llama al método **WinCinematic** del personaje principal para que este salga de la sala por la puerta.
5. **Cambio de escena:** Cuando el personaje principal se ha alejado lo suficiente de la sala, esta llama al método **NextStage** de la sala, lo que provoca un cambio de escena. Si se trataba del ultimo nivel, se carga la **escena de victoria**, si no, se vuelve a cargar la escena de juego, incrementando el índice de nivel.

El **estado de derrota** se alcanza cuando la pelota golpea tres veces consecutivas el suelo de la sala. Cuando esta condición se cumple, empieza la siguiente secuencia de acciones:

1. **Destrucción de la Pelota:** La pelota comienza su animación de destrucción
2. **Derrota del jugador:** Cuando la pelota acaba su animación, se envía un mensaje al jugador para que comience con su animación de derrota.

3. ARQUITECTURA

3. **Cambio de escena:** Cuando termina la animación de la pelota, esta llama al método **GameOver** de la sala, el cual inicia la transición a la **escena de fin del juego**

3.5 Control del jugador

El **personaje principal** es el avatar del jugador dentro del juego y puede controlarlo directamente pulsando teclas del teclado. El aspecto de este personaje es el de un pequeño robot (figura 3.13).

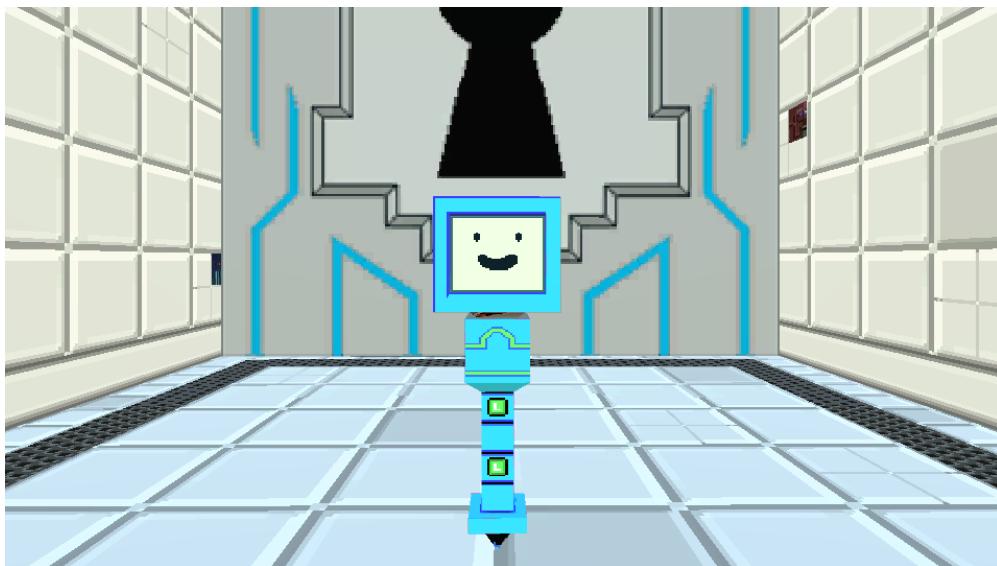


Figura 3.13: Modelo del personaje principal.

El comportamiento del personaje principal es bastante sencillo:

- Cuando el jugador pulsa las flechas de dirección, el personaje **se mueve hacia dicha dirección a velocidad constante**.
- Cuando el jugador pulsa la tecla espacio, el personaje **despliega una “paleta”** frente a él. El jugador debe utilizar esta paleta para redirigir la pelota. La paleta desaparece tras unos segundos, durante los cuales el personaje no puede moverse.
- Si la pelota golpea al personaje principal, este **se queda aturdido** unos segundos durante los cuales no obedecerá las instrucciones del jugador.

En adición a su comportamiento habitual, el personaje tiene tres **animaciones cinematográficas** que se utilizan al principio y final de los niveles del juego. Estas animaciones se reproducen cuando empieza un nivel, cuando el jugador supera un nivel y cuando el jugador pierde. Durante las animaciones, el jugador no puede controlar al personaje

3.5.1 Componentes

El personaje principal está implementado mediante dos GameObjects anidados (ver figura 3.14). El **GameObject padre** contiene la lógica del personaje, determinada por los componentes que este tiene asociados.

- Un **Rigidbody**. Este componente permite dotar al personaje de un comportamiento **basado en físicas**. El componente almacena las propiedades físicas del personaje (velocidad, aceleración, gravedad...) e implementa métodos para poder aplicarle fuerzas y cambios de velocidad.
- Un **Collider**. Este componente se utiliza en la **detección de colisiones** entre el personaje principal y el resto de los objetos de la escena. En concreto, el personaje principal tiene asociado un **BoxCollider**, el cual tiene forma de prisma rectangular.
- Un **Audio Source**, es el componente encargado de reproducir los sonidos del personaje.
- Un **Animator**, el cual reproduce las animaciones del personaje principal. Las animaciones (realizadas en el propio editor de Unity) estar guardadas como “clips” que el Animator carga dependiendo del estado interno del personaje.
- Un **Script** de la clase **MainCharacter**, el cual se ocupa de recibir la información de entrada del jugador y usarla para determinar qué acciones deben realizar los demás componentes.

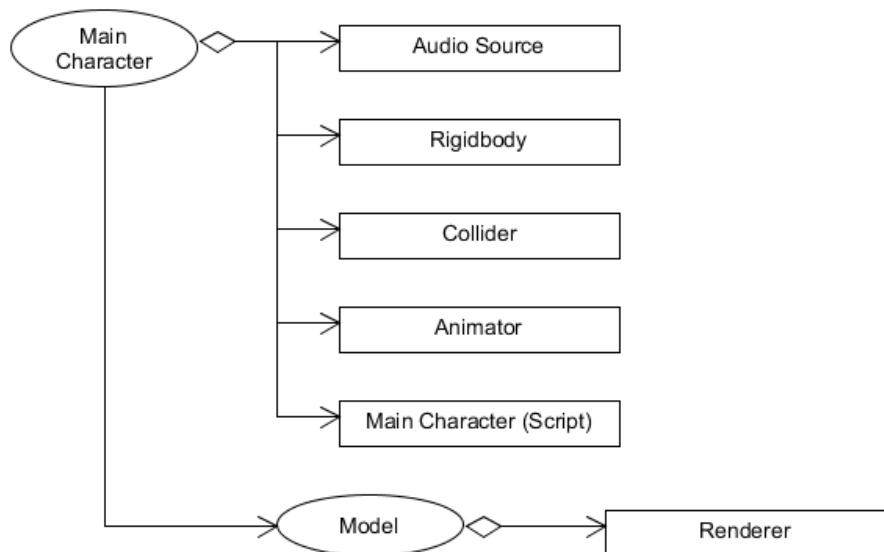


Figura 3.14: Diagrama de componentes del personaje principal.

El **GameObject hijo** se encarga únicamente de contener renderizar el modelo del personaje a través del componente **MeshRenderer**. El motivo de esta división es un

3. ARQUITECTURA

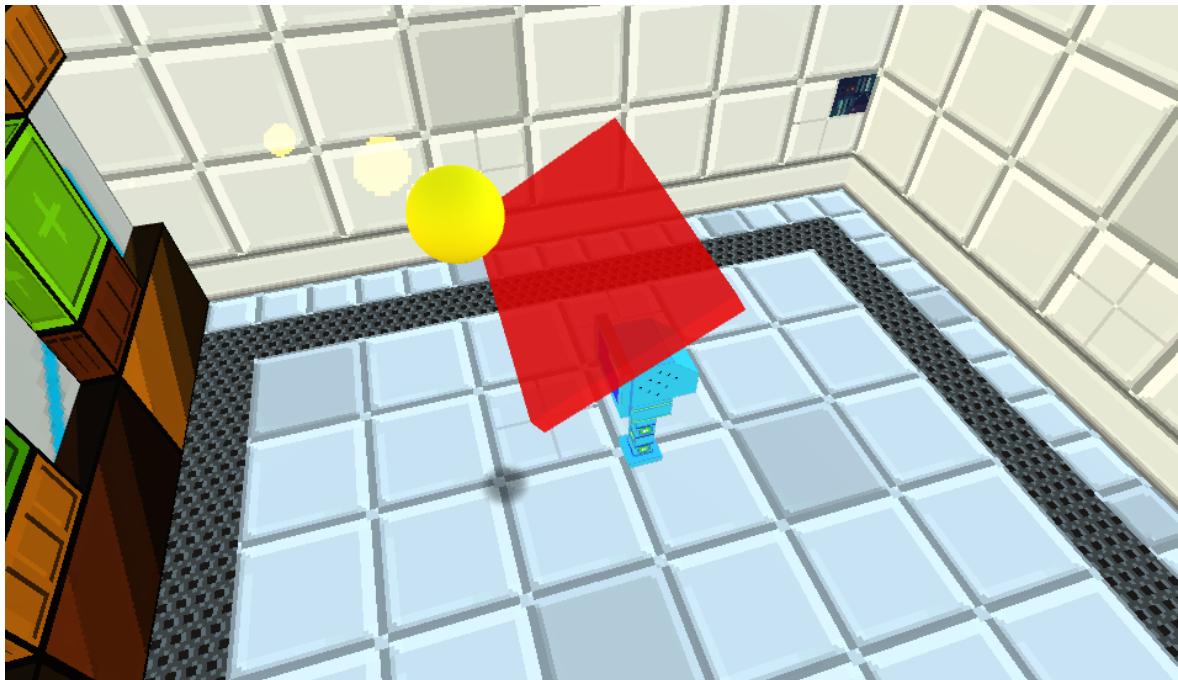


Figura 3.15: Personaje principal activando la paleta.

conflicto entre los componentes **Rigidbody** y **Animator** del GameObject principal debido a que ambos manipulan la rotación y posición del personaje, por lo que fue necesario mover el modelo a un objeto anidado. De esta forma, el rigidbody manipula la **posición y rotación globales** del personaje, mientras que el Animator controla las **propiedades locales** del objeto anidado.

Adicionalmente al objeto principal se encuentra la **Paleta** (figura 3.15). Se trata de un GameObject muy sencillo que cuenta con solo dos componentes: un **Renderer** y un **BoxCollider**. La paleta está guardada como **Prefab** en los assets del juego, de forma que el personaje principal puede instanciarla cuando sea necesario.

3.5.2 Comportamiento del personaje

La clase **MainCharacter** es la encargada de coordinar el comportamiento del resto de componentes del personaje principal y la única clase del grupo funcional **Player Input** (figura 3.16). Se trata de un componente asociado al objeto principal del personaje el cual lee la entrada del jugador, procesa las teclas pulsadas y usa esa información para llamar a los métodos pertinentes del resto de componentes.

El comportamiento de MainCharacter se basa en una **maquina de estados finitos**, dado que las distintas acciones que el personaje puede realizar pueden modelarse muy bien como estados independientes, lo que además permite prevenir y aislar mejor los fallos de programación durante el desarrollo. En la figura 3.17 se encuentran el diagrama de estados, los cuales se describen a continuación:

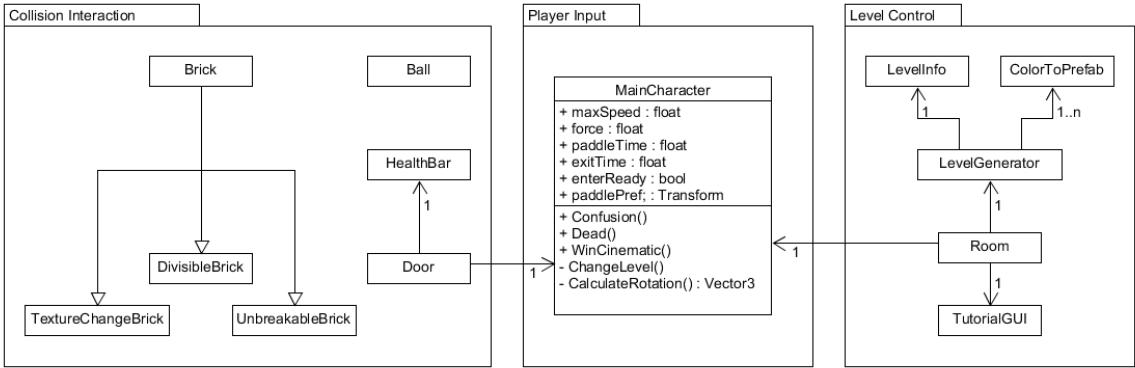


Figura 3.16: Diagrama del grupo funcional **Player Input**.

El estado **Enter** es el estado inicial del personaje. En este estado el personaje avanza desde el exterior de la sala hacia el centro de esta. Una vez alcanzado el centro de la sala, el personaje pasa al estado **Stand**. El desplazamiento se realiza mediante el componente **Rigidbody**, modificando el parámetro de la velocidad a un valor constante. Durante este estado, el **Collider** se encuentra desactivado, para evitar que el jugador choque con las paredes de la sala.

El estado **Stand** es el estado por defecto del personaje, el que se encuentra activo cuando no se detecta ninguna pulsación de teclas ni interacciones con otros objetos. En este estado, el personaje permanece inmóvil ejecutando su animación de espera. El personaje cambiará de estado cuando detecte una pulsación de teclas: al estado **Move** si se han pulsado las flechas de dirección y al estado **Paddle** si se ha pulsado la tecla espacio.

En estado **Move** el personaje se mueve a velocidad constante en la dirección determinada por las flechas de dirección que el jugador esté pulsando. El componente se encarga de generar un vector que apunta a en la dirección resultante de sumar las direcciones de las flechas pulsadas, el cual se introduce como parámetro en el método **AddForce** del componente **Rigidbody**. El método **AddForce** aplica una fuerza determinada por el vector suministrado al objeto provocando su desplazamiento. Se utiliza este método en lugar de modificar la velocidad directamente porque así se obtiene una ligera aceleración inicial, que es mucho más agradable y natural para el jugador. La fuerza que utiliza para el movimiento es muy alta y va acompañada de un factor de fricción también elevado, lo que permitía controlar al personaje con precisión.

El estado **Paddle** el personaje instancia una **paleta** frente a él. Al mismo tiempo que se instancia la paleta comienza una cuenta atrás de unos segundos, tras los cuales la paleta se destruye y el personaje retorna al estado **Stand**. Durante la ejecución de este estado, el jugador no podrá controlar el movimiento del jugador, sin embargo, la

3. ARQUITECTURA

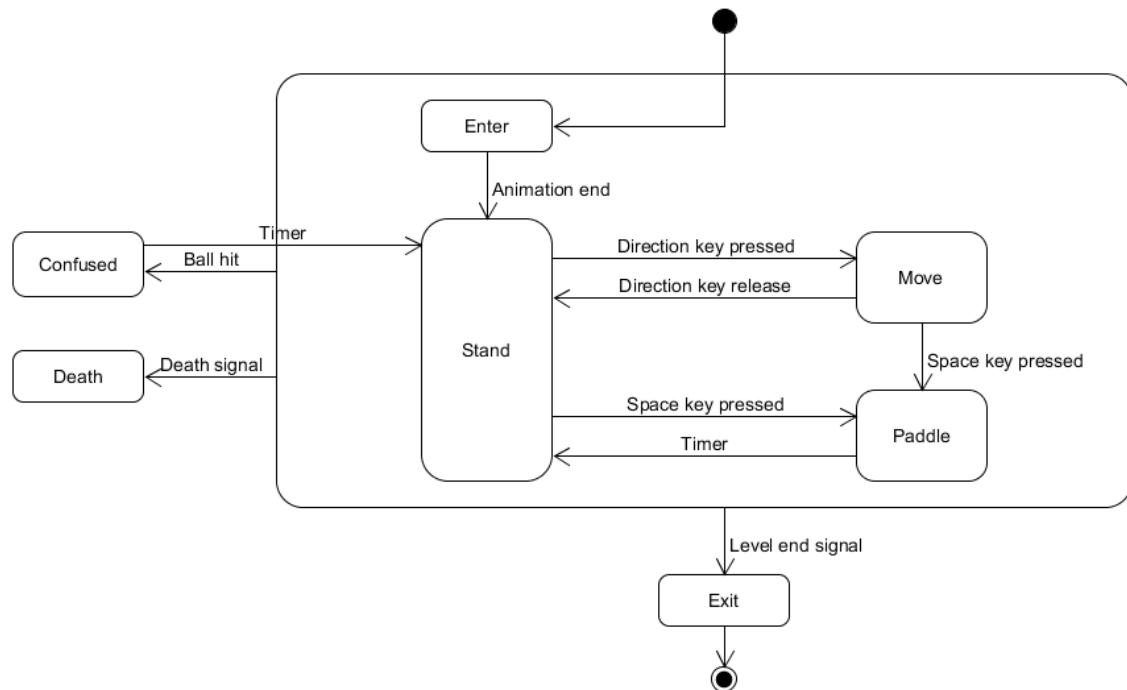


Figura 3.17: Diagrama de estados del personaje principal.

fricción entre el personaje y el suelo de la sala se reduce en este estado, por lo que el personaje conservará parte de la velocidad que tuviese en estados anteriores gracias a la inercia.

El personaje entra en el estado **Confused** cuando es golpeado por la pelota, independientemente de en qué estado se encontrase antes. El personaje permanecerá en este estado durante unos segundos mientras el **animator** ejecuta una animación de mareo. Finalizado el estado, el jugador volverá al estado **Stand**.

El estado **Dead** comienza cuando el personaje recibe una llamada a su método **Dead**, es cual sirve para notificarle de la destrucción de la pelota. Este método esta siempre activo, por lo que el personaje puede entrar en este estado desde cualquier otro. Al entrar en este estado, el **Animator** reproduce la animación de muerte del personaje, en la que este se desploma derrotado y se queda inmóvil. La lectura de entrada queda desactivada, por lo que el jugador pierde el control sobre el personaje.

El estado **Exit** empieza cuando se llama al método **WinCinematic** del personaje. Esta llamada se produce cuando el jugador supera el nivel actual. Durante este estado, el personaje avanza hacia el frente, atravesando la puerta de la sala (ahora abierta) y creando la ilusión de que “avanza al siguiente nivel”. Este movimiento del jugador se implementa de la misma forma que el del estado **Move**, usando el método **AddForce** del componente **RigidBody**.

3.6 Físicas e interacción

La dinámica de este juego surge principalmente de las **interacciones físicas entre los distintos objetos** contenidos en la sala. Obviando las simples colisiones entre el personaje y la sala que sirven para evitar que este no abandone la sala, todas las colisiones se producen entre **la pelota** y otro objeto. Cada objeto de la sala reacciona de forma distinta al impacto con la pelota.

Las clases que gestionan estos comportamientos se encuentran en el grupo funcional **Collision Interaction**. Las clases incluidas en este grupo son **Ball**, el componente que codifica el comportamiento de la bola; **Brick** el componente del comportamiento de los ladrillos de la puerta (y sus subclases **TextureChangeBrick**, **DivisibleBrick** y **UnbreakableBrick**) y los componentes **Door** y **HealthBar** de la puerta. La figura 3.18 muestra un diagrama de este grupo.

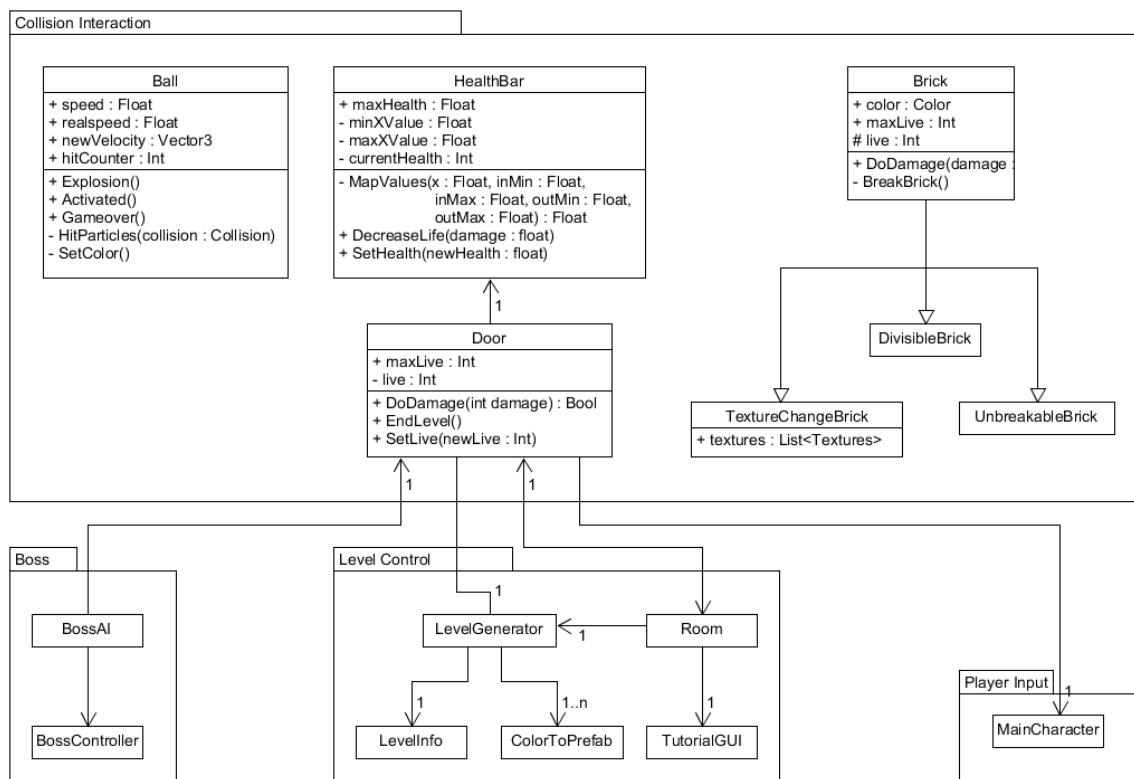


Figura 3.18: Diagrama del grupo funcional **Collision Interaction**.

En los siguientes apartados se describirá en detalle a la pelota, así como los distintos objetos que pueden colisionar con esta.

3.6.1 Pelota

Podríamos considerar **la pelota** como el **arma del personaje principal**, el medio por el que el jugador interacciona con los objetos del juego. La pelota viaja en trayec-

3. ARQUITECTURA

toria rectilínea, rebotando en las paredes y el techo de la sala del juego. Al golpear la puerta, o los bloques que la cubren, estos reciben daño y la pelota rebota, pero si la pelota golpea el suelo de la sala es ella la que recibe daño, no el jugador. Si la pelota golpea el suelo tres veces, esta se destruye y partida se acaba. El jugador debe intentar golpear la pelota con la paleta para redirigirla hacia la puerta, evitando que toque el suelo.

Visualmente, la pelota es una simple esfera de color uniforme, como puede verse en la figura 3.19. Para facilitarle al jugador la tarea de seguir la trayectoria de la pelota, esta va dejando a su paso un rastro luminoso y produce con cada colisión un efecto visual de impacto.

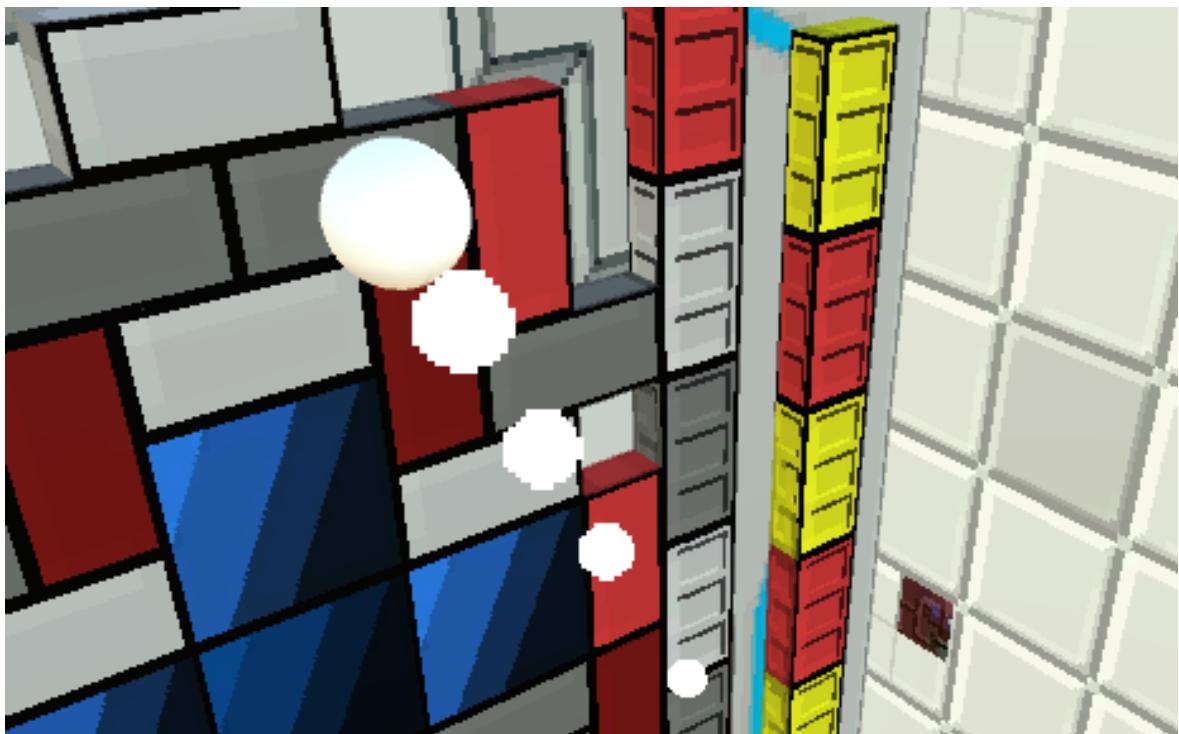


Figura 3.19: Pelota moviéndose.

Componentes

La pelota está implementada mediante dos **GameObject** anidados. El primer GameObject tiene asociados una serie de componentes que modelan su comportamiento (la estructura de componentes puede verse en la figura 3.20. Estos componentes son:

- Un **MeshRenderer** encargado de renderizar el modelo de la bola, el cual es una sencilla esfera con una textura lisa.
- Un **SphereCollider** que se utiliza para la detección de colisiones entre la pelota y el resto de los objetos de la sala.

- Un **Rigidbody** que sirve para dotar a la pelota de propiedades físicas.
- Un **Audio Source**, el cual reproduce los sonidos de impacto entre la pelota y los otros objetos.
- Un **Animator** para ejecutar la animación de destrucción de la bola.
- Un **Particle System**, un componente que permite crear animaciones y efectos especiales utilizando **partículas**, pequeños gráficos bidimensionales. En la pelota, el ParticleSystem se utiliza para crear una **trayectoria** que marca el camino que siguió la pelota.
- Un **Script** de la clase **Ball** que controla el comportamiento de la bola, determinando su estado y determinando qué reacción debe ejecutar cuando colisiona con otros objetos.

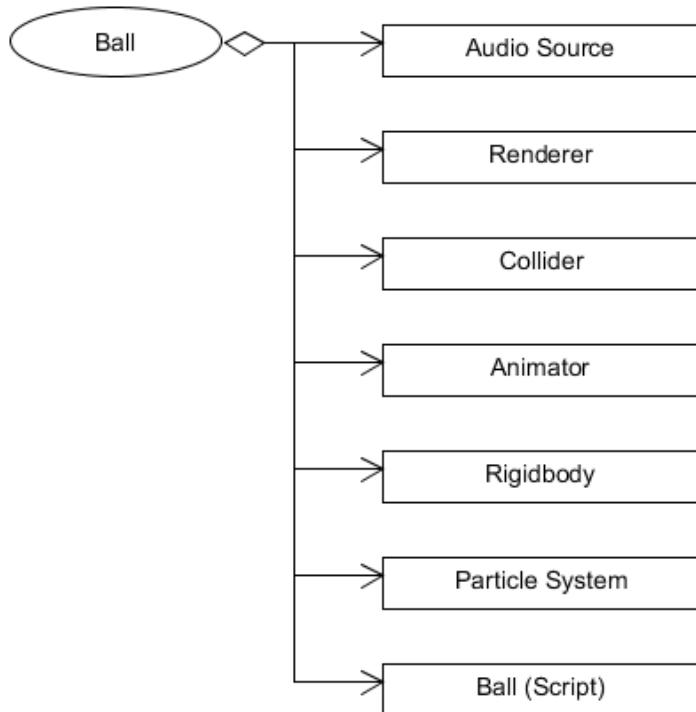


Figura 3.20: Componentes del objeto Ball.

Anidado a este primer GameObject se encuentra el **emisor de sombra**. La función de este GameObject es la de producir una sombra esférica contra el suelo de la sala gracias a su componente **Proyector**. El motivo por el que se emplea este componente en lugar de utilizar el sistema de iluminación de Unity para crear la sombra es que no se busca crear una sombra realista de la pelota, sino una referencia en el suelo de la posición de la pelota que permita al jugador seguirla más fácilmente. Si se quisiese obtener el mismo efecto con el sistema de iluminación sería necesario utilizar una configuración de luces muy específica que desmejoraría el reto de gráficos.

3. ARQUITECTURA

Adicionalmente, la pelota tiene acceso a dos objetos prefabricados que implementan sus efectos visuales: **ParticleCircle** y **ExplosionParticle**. ParticleCircle genera una explosión de partículas en forma de anillo mientras que ExplosionParticle genera una explosión esférica. Ambos objetos tienen un solo componente: un ParticleEmitter ya que es la pelota la encargada de completar el resto de sus funciones. En la figura 3.21 pueden verse los efectos de ambos objetos.

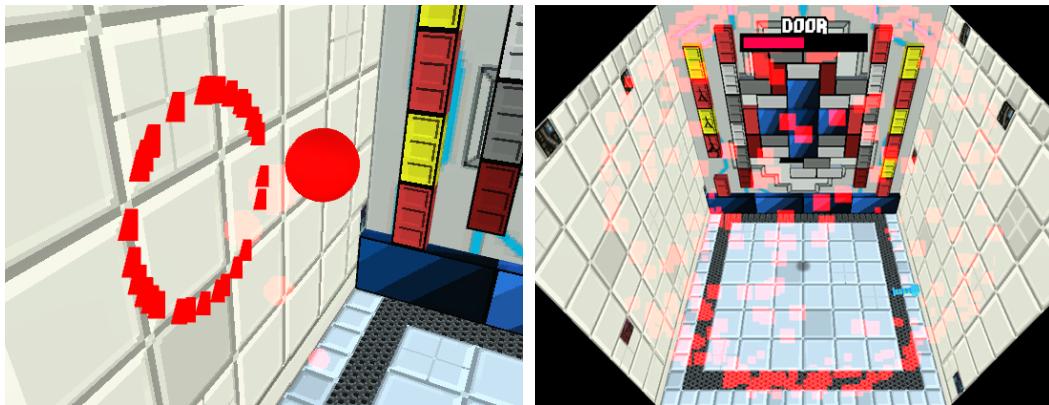


Figura 3.21: ParticleCircle (izquierda) y ExplosionParticle (derecha).

Se eligió utilizar estos dos objetos como emisores de partículas en lugar de asignárselas directamente a la pelota debido a que esta ya contaba con emisor de partículas para dibujar su trayectoria y Unity no permite añadir más de un componente del mismo tipo al mismo objeto. Esta limitación podría haberse resuelto también asociando dos GameObjects adicionales permanentes a la pelota con los dos emisores, pero habrían necesitado de lógica adicional para determinar cuándo empezar y terminar la emisión, por lo que se optó por la solución actual.

Comportamiento

El componente **Ball** es el encargado de dotar a la pelota de su comportamiento. Su funcionamiento se basa en una máquina de estados finitos con un total de tres estados: **Normal**, **Locked** y **Destroy**.

Los estados **Locked** y **Destroy** son estados cinemáticos utilizados durante las secuencias de inicio y final del nivel. Durante el estado **Locked** la pelota tiene desactivados sus componentes **Renderer**, **Rigidbody** y **Collider**, por lo que es inmóvil, invisible e intangible. Este es el estado en el que la pelota se encuentra durante las secuencias de introducción y de victoria. Por otro lado, durante el estado **Locked** el componente **Animator** de la pelota reproduce una animación de destrucción, que culmina con la instanciación de un **ExplosionParticle**. Este es el estado que utiliza la pelota durante la derrota.

Durante el resto de la ejecución, la pelota se encuentra en el estado **Normal**. En este

estado, la pelota se mueve a velocidad constante por la sala rebotando en los diversos objetos contenidos en esta, así como en sus paredes, suelo y techo. El movimiento de la pelota se consigue gracias al componente **Rigidbody**. Al entrar en el estado **Normal**, la pelota recibe un impulso en una dirección aleatoria mediante el método **AddForce** del rigidbody y a partir de ahí se moverá de forma perpetua al haber sido configurada sin fricción.

La pelota cuenta con un **contador de golpes**, el cual aumenta cuando la pelota colisiona con ciertos objetos. Cuando este contador llega a tres, la pelota es destruida y el jugador pierde la partida. El contador de golpes también sirve para determinar el **color** de la pelota: blanca si no ha recibido golpes, amarilla si ha recibido uno y roja si ha recibido dos o más.

Cuando la pelota colisiona con otro objeto físico, esta debe determinar de qué tipo de objeto se trata para poder producir la reacción correspondiente. La detección de colisiones se realiza con el método **OnCollision**, el cual es llamado por Unity cuando se produce una colisión entre la pelota y cualquier otro objeto dotado de un Collider, el cual es suministrado como parámetro del método.

La identificación de los objetos se consigue gracias a sus **Tags**, un atributo presente en todos los GameObjects que permite distinguirlos rápidamente. El tag se utiliza junto a una estructura **switch** para elegir la reacción más apropiada de la pelota de entre una serie de **métodos privados**. Este sistema, aunque funcional para el estado actual del proyecto, es poco escalable a largo plazo y sería necesario desacoplar las distintas reacciones de la pelota del resto de su comportamiento.

3.6.2 Elementos Interactivos

Paredes y Suelo

Las paredes, el suelo y el techo son objetos hijos de la sala. Estos objetos carecen de comportamiento en sí mismo, pero pueden producir cambios en el comportamiento de la pelota cuando esta choca con ellos.

Si la pelota choca contra **las paredes** de la sala, cambiará de dirección rebotando de forma natural. Al no tener fricción, la pelota conservará su velocidad tras el rebote. Este comportamiento no se encuentra codificado dentro del componente **Ball**, sino que se trata de la consecuencia de las propiedades físicas del componente **Rigidbody**. A efectos prácticos, el techo de la sala es otra pared más, con un comportamiento idéntico al de las otras paredes.

Sin embargo, el comportamiento de la pelota cuando golpea el **suelo** de la sala es distinto. Al impactar contra el suelo, el contador de golpes se incrementará en uno. Para paliar el efecto negativo de este tipo de colisión, tras el impacto la pelota en

3. ARQUITECTURA

dirección a la puerta de la sala. De esta forma, aunque el jugador haya se encuentre más cerca de perder la partida, tendrá más posibilidades de reponerse dado que se le ha “regalado” un golpe adicional a la puerta.

Personaje Principal

El objetivo principal del jugador es conseguir que el personaje principal **golpee la pelota con la paleta**. La dirección hacia la que rebote la pelota tras la colisión depende del punto de la paleta en el que se produjo el impacto, en lugar de en el ángulo de colisión. La trayectoria de salida de la pelota será más perpendicular al plano de la paleta cuanto más cerca de su centro se encuentre, y más paralela cuanto más alejado del centro sea. En la figura 3.22 se pueden ver de forma aproximada las trayectorias.

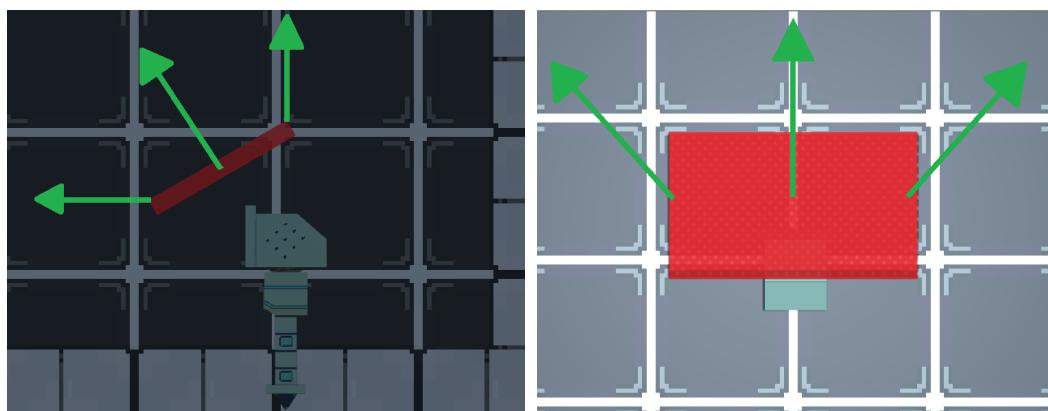


Figura 3.22: Direcciones que tomaría la pelota dependiendo del punto de la paleta que golpee (Aproximada).

Por otro lado, si la pelota golpea al **personaje principal** directamente, este se quedará aturdido unos instantes. Tras el impacto, la pelota rebotará de forma natural.

Puerta

La **puerta** es una estructura que ocupa la pared norte de la sala (figura 3.23). El objetivo del juego es el de abrir la puerta golpeándola con la pelota. La puerta tiene un contador de **puntos de vida** que disminuye con cada impacto de la pelota. Cuando el contador llega a cero, la puerta se abre y el nivel se acaba. Además de con el descenso de sus puntos de vida, la puerta reaccionara a los impactos de la pelota moviendo sus paneles.

El GameObject que se utiliza para implementar la puerta se encuentra **anidado dentro de la sala**. La puerta tiene a su vez dos objetos anidados que hacen de paneles de la puerta. Los componentes de la puerta son:

- Un **BoxCollider** utilizado en la detección de colisiones.

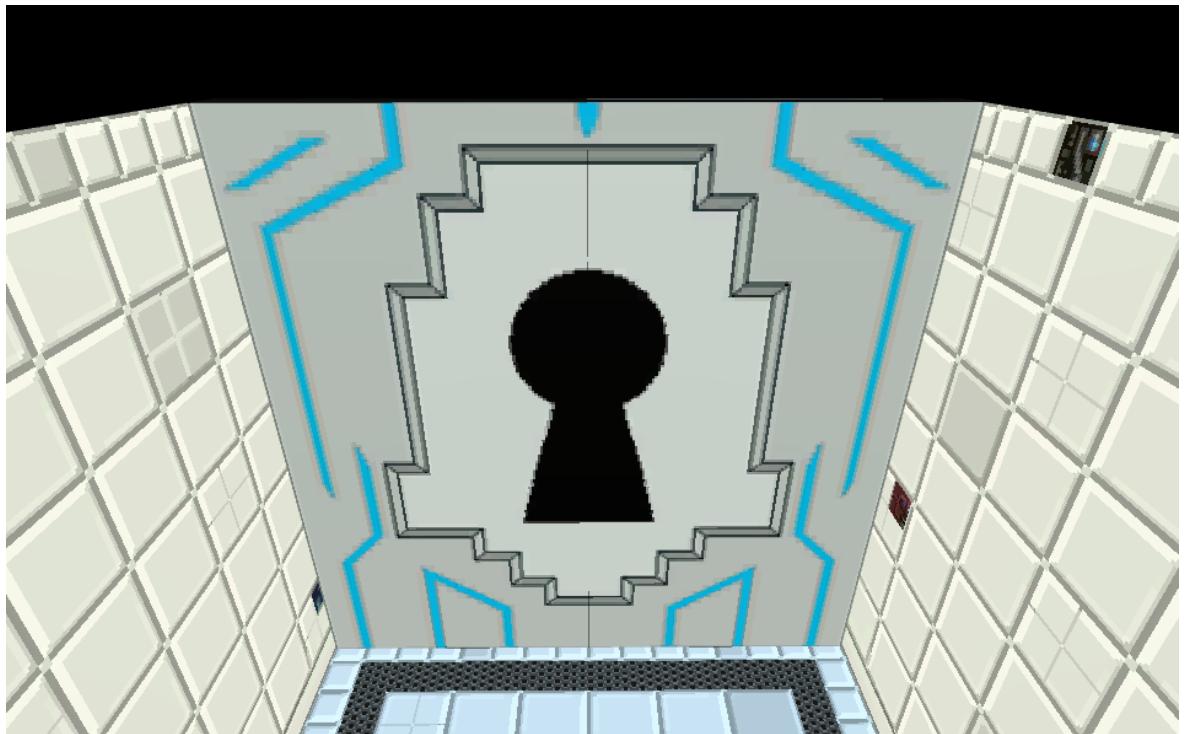


Figura 3.23: Modelo de la puerta.

- Un **Animator** que se encarga de reproducir las animaciones de la puerta
- Un **Script** de la clase **Door** la cual se encarga de gestionar el comportamiento de la puerta.

EN la figura 3.24 se puede ver de forma gráfica esta relación de objetos.

Cuando la pelota impacta con la puerta, esta llama al método **DoDamage** del componente **Door** de la puerta. Este método disminuye los puntos de vida de la puerta, inicia la animación de golpe y, en caso de que la puerta tenga cero puntos de vida, inicia la animación de apertura de la puerta.

Para mostrarle al jugador los puntos de vida que tiene la puerta se utiliza una **barra de vida** (figura 3.25. Se trata de un objeto de **interfaz de usuario** de Unity que muestra gráficamente el porcentaje de puntos de vida que le queda a la puerta. El componente **HealthBar** se encarga de tomar los puntos de vida de la puerta y utilizarlos para calcular el tamaño de la barra.

Tras impactar con la puerta, la pelota rebotara de forma natural. Adicionalmente, el parámetro **UseGravity** del Rigidbody de la pelota cambiará su valor a verdadero, por lo que la pelota empezará a caer en dirección al suelo de la sala. La gravedad de la pelota no se desactivará hasta que no golpee el suelo de la sala o la paleta del jugador.

3. ARQUITECTURA

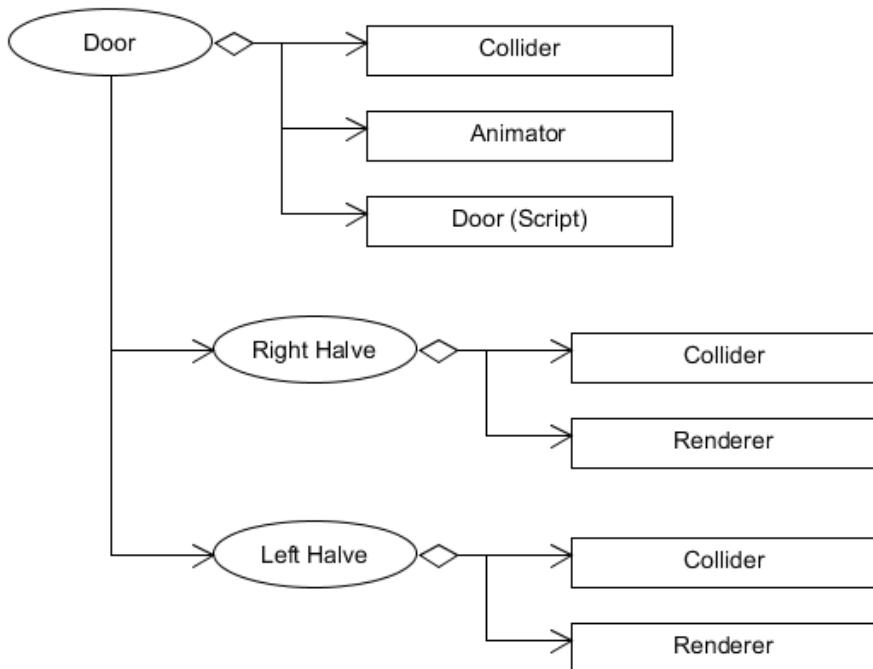


Figura 3.24: Diagrama de componentes de la puerta.

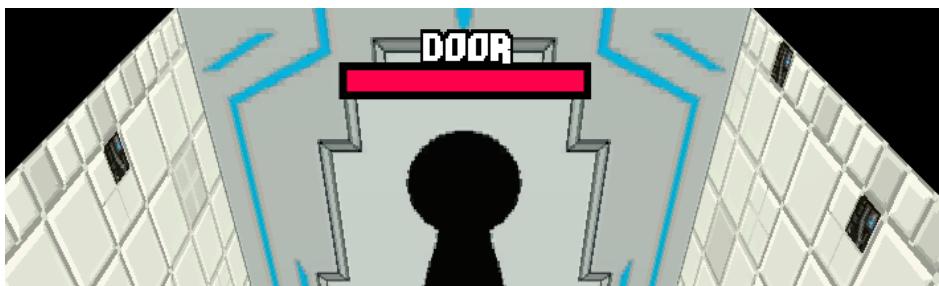


Figura 3.25: Barra de vida de la vida.

Ladrillos

Los **ladrillos** son el principal obstáculo del juego, ya que protegen la puerta de los golpes de la bola. Cada vez que la pelota golpea un ladrillo, este pierde un punto de vida. Cada ladrillo tiene asignado un **color** que tiñe su textura. Cuando sus puntos de vida llegan a cero, el ladrillo se destruye. Los ladrillos se colocan delante de la puerta, en una **cuadricula** de 16 X 16 casillas. Cada nivel del juego tiene una configuración de ladrillos diferente.

Para darle variedad al juego, existen varios tipos de ladrillos (ver figura 3.26):

- **Ladrillo básico:** Es el ladrillo básico. Solo tiene un punto de vida. Este ladrillo mide 1 X 2 casillas.
- **Ladrillo Multi-golpe:** Este ladrillo tiene tres puntos de vida, por lo que es más difícil romperlo. La textura de este ladrillo cambia con cada golpe para mostrar

el daño infligido. Este ladrillo mide 1 X 2 casillas.

- **Ladrillo Divisible:** Es un ladrillo de gran tamaño, mide 2 X 2 casillas. Aunque solo tiene un punto de vida, al romperse genera 4 ladrillos pequeños de 1 X 1 casillas de tamaño.
- **Ladrillo Pequeño:** es un ladrillo idéntico al básico, salvo porque solo mide 1 X 1 casillas. Solo aparecen como resultado de la rotura de un ladrillo divisible.
- **Ladrillo Irrompible:** Como su nombre indica, este ladrillo no puede ser destruido por la pelota. Miden 2 X 3 casillas.

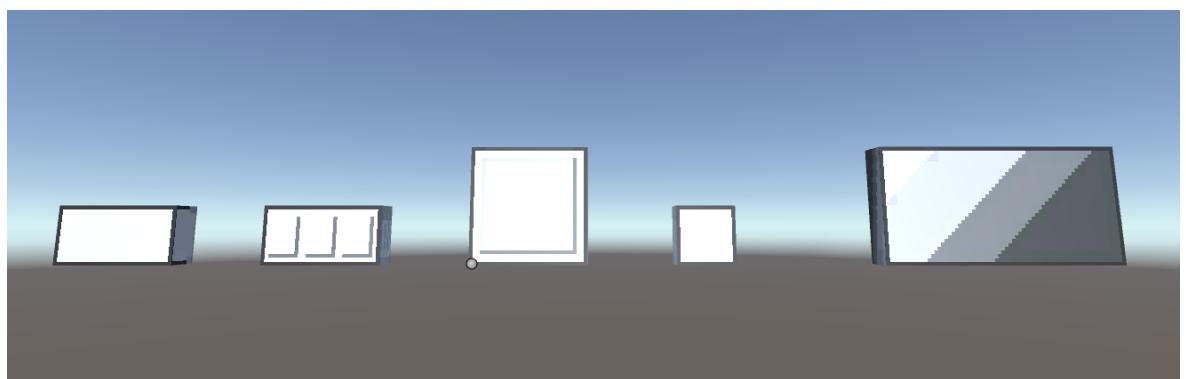


Figura 3.26: Modelos de los ladrillos.

Para implementar los ladrillos se utilizan dos GameObjects anidados, como se muestra en 3.27. El primer GameObject se encarga de dotar al ladrillo de funcionalidad, mientras que el segundo contiene el modelo 3D del mismo. Los componentes que contienen los ladrillos son los siguientes:

- Un **BoxCollider**, para la detección de colisiones.
- Un **Particle System**, para la animación de destrucción.
- Un script de clase **Brick**, que contiene la funcionalidad del ladrillo.

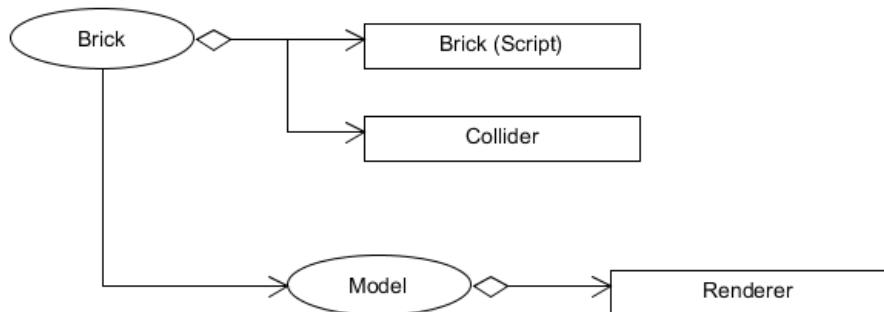


Figura 3.27: Componentes del objeto Brick.

3. ARQUITECTURA

Cuando la pelota impacta contra un ladrillo, llama al método **DoDamage** de su componente **Brick**. Este método recibe como parámetro una cantidad de “puntos de daño” la cual sustrae a los puntos de vida actuales del ladrillo. Además, este método también destruye los ladrillos que han perdido todos sus puntos de vida y emite partículas del color del ladrillo con cada golpe recibido.

Los ladrillos básicos tendrían el comportamiento anteriormente distinto, pero el resto de los tipos de ladrillos requieren comportamientos específicos. De la clase **Brick** heredan tres clases hijas con código adicional para poder implementar todos los tipos de ladrillos, tal y como aparece en la figura 3.28. Estas clases son:

- **TextureChangeBrick**: Este script cambia la textura del ladrillo dependiendo del número de puntos de vida. Se utiliza en los **ladrillos multi golpes** para mostrar el ladrillo en varios estados de destrucción.
- **DivisibleBrick**: Este componente se utiliza en la implementación de los **ladrillos divisibles**. Cuando este tipo de ladrillos se destruye, el componente instancia cuatro **bloques pequeños**.
- **UnbreakableBrick**: Este componente ignora las llamadas de la bola, por lo que el ladrillo no pierde puntos de vida. Se utiliza en los **ladrillos irrompibles**.

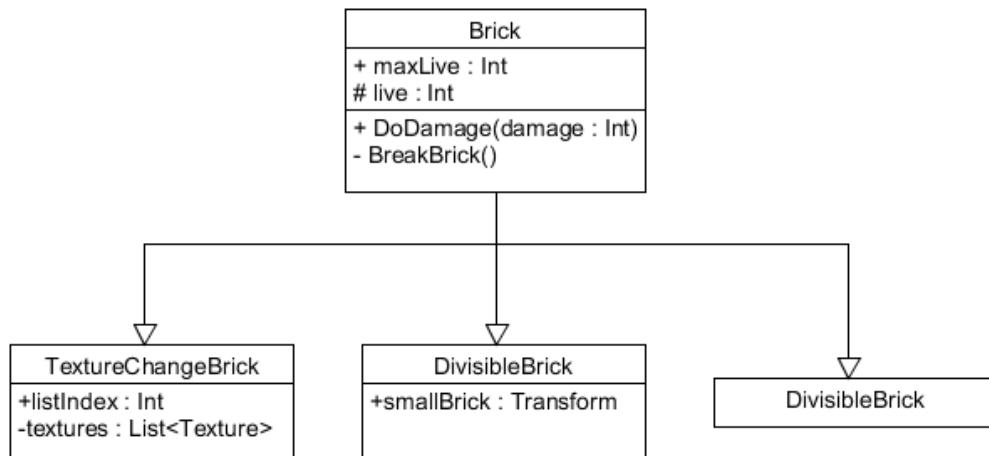


Figura 3.28: Jerarquía de clases de Brick.

La reacción de la pelota al golpear un ladrillo es idéntica a la que tiene cuando golpea la puerta: la pelota rebotará de forma natural, pero se activará su **gravedad**, por lo que la pelota empezará a caer en dirección al suelo de la sala. De la misma manera, la gravedad de la pelota permanecerá activa hasta que golpee el suelo o la paleta.

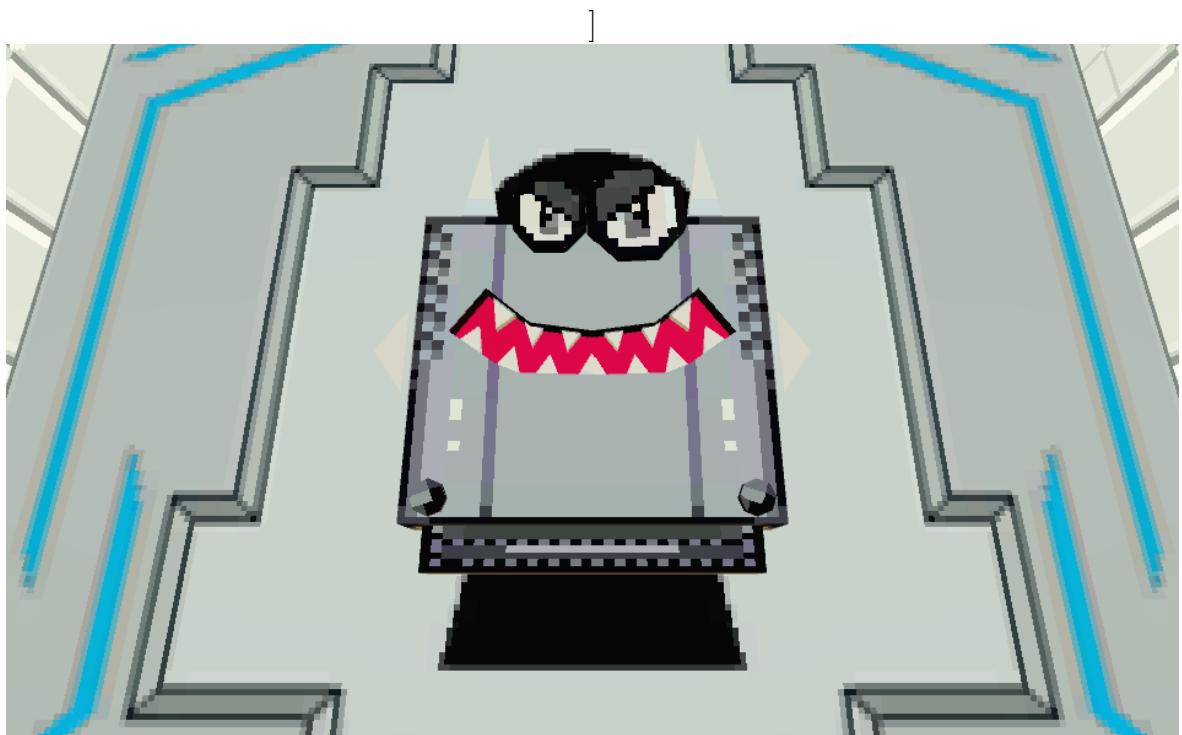


Figura 3.29: Modelo del jefe.

3.7 Jefe Final

Un **jefe final**, o monstruo jefe, es un enemigo poderoso al que los jugadores deben enfrentarse para poder alcanzar algún objetivo dentro del juego [Bjo]. Los jefes sirven principalmente para tres propósitos: dar clausura a una sección del juego, al colocarse al final de una sección, nivel o fase; ofrecen variedad al juego, al cambiar las mecánicas de juego; y finalmente sirven como “test” para el jugador, al tratarse de un desafío mucho mayor que los anteriores.

Para este juego, el jefe final es el único “enemigo” al que se enfrenta el jugador, que aparece en el nivel 11 del juego. Este jefe se comporta como un “**clon**” o “rival” del jugador: su objetivo es impedir que la pelota golpee la puerta, moviéndose y redirigiéndola de forma similar a como lo haría el jugador. La forma de derrotar a este jefe es idéntica de a como se supera cualquier fase: Golpeando la puerta suficientes veces.

El jefe se mueve a **velocidad constante**, siempre pegado a la pared, intentando bloquear la pelota. Sin embargo, si detuviese siempre la pelota, sería un desafío imposible de superar, lo que no sería divertido para el jugador. La idea es que este jefe se juegue como si de una partida de tenis se tratase, con dos adversarios intercambiándose la pelota hasta que uno de los dos falle. Por ello, el jefe tendrá un **sistema de energía** que determinará su eficiencia a la hora de jugar e irá consumiéndose según avance la partida. En la figura 3.29 puede verse el modelo 3D de este enemigo.

3. ARQUITECTURA

3.7.1 Componentes

Para implementar al jefe se utilizan **dos GameObjects anidados**. El GameObject principal contiene los componentes necesarios para el funcionamiento del jefe. Los componentes de este primer GamObject son los siguientes:

- Un **Rigidbody** que sirve para dotar al jefe de **propiedades físicas**, como velocidad o aceleración.
- Un **BoxCollider** el cual permite realizar la detección de colisiones entre el jefe y otros objetos.
- Un **Animator** el cual dota al jefe de animaciones de modo que su comportamiento no resulte demasiado estático.
- Un **Audio Source** que sirve para emitir sonidos y música. El jefe lo utiliza para emitir sonidos en situaciones concretas, como al golpear la pelota o al ser derrotado.
- Un **Script** de clase **BossController**. Este componente determina que movimientos es capaz de realizar el jefe.
- Un **Script** de clase **BossAI**. Este componente se encarga de controlar el comportamiento del jefe. En la figura 3.30 se pude ver un diagrama de este objeto y sus componentes.

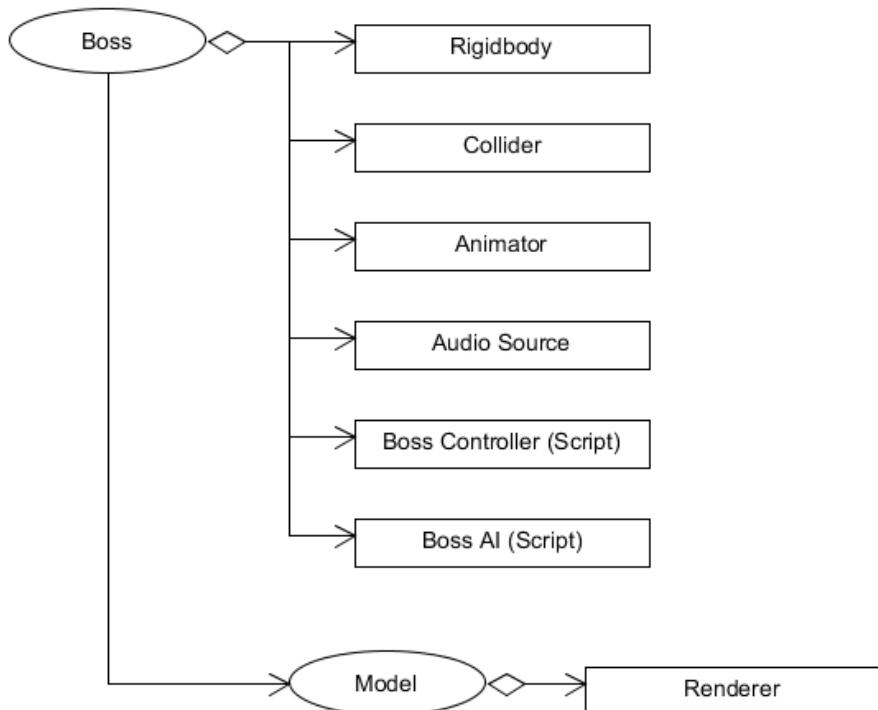


Figura 3.30: Diagrama de componentes del jefe.

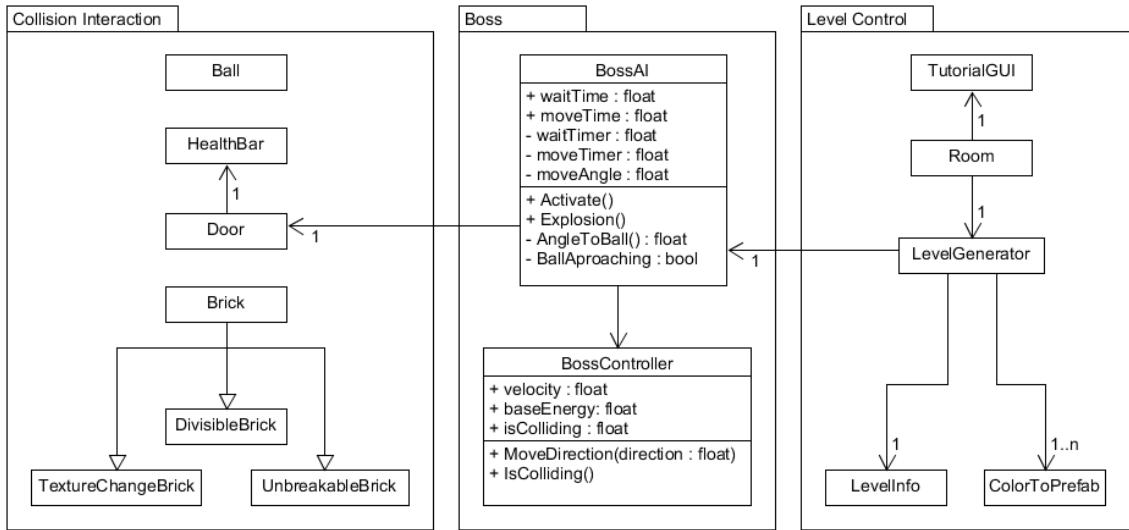


Figura 3.31: Diagrama del grupo funcional **Boss**.

El segundo GameObject contiene el modelo 3D del jefe, el cual es dibujado en pantalla gracias a un componente **MeshRenderer**. Esta separación se realizó para evitar colisión entre los comportamientos de los componentes **Rigidbody** y **Animator**.

El jefe no **forma parte de la escena del juego** al comienzo de la ejecución, sino que está almacenado en los assets del juego en forma de **Prefab**. El prefab es instanciado por el componente **LevelGenerator** de la sala durante la generación del nivel en el caso de que la información del nivel indique que se trata de un nivel del jefe. Al ser instanciado, el jefe es asignado como GameObject hijo de la puerta, de forma que su posición sea siempre relativa a esta.

El comportamiento del jefe se encuentra definido por las clases del grupo funcional **Boss**. Las dos clases de este grupo son **BossController**, que implementa las posibles acciones que el jefe puede realizar y **BossAI**, una inteligencia artificial que determina qué acciones debe ejecutar el jefe en un momento dado. El diagrama 3.31 muestra gráficamente la composición de este grupo.

La división del comportamiento del jefe en dos componentes se realizó para reducir la complejidad que habría supuesto un único competente monolítico. De esta forma, cada componente pudo ser implementado y revisado de forma independiente, lo que redujo los errores y el tiempo de desarrollo.

3.7.2 Movimiento

El componente **BossController** contiene la información y las funciones que el jefe necesita para realizar sus acciones. Las acciones que el jefe puede realizar son **moveverse**, modificar sus **valores de energía**, ser **activado y destruido** y **reproducir**

3. ARQUITECTURA

sonidos.

El movimiento del jefe se produce mediante el método **MoveDirection**. Este método mueve al jefe a velocidad constante en la dirección suministrada como parámetro. La dirección del movimiento viene dada como un ángulo, el cual se utiliza para calcular el vector de dirección paralelo al plano de la puerta. El movimiento se implementa mediante el método **AddForce** del componente **Rigidbody** del jefe, de esta forma el movimiento tiene una aceleración inicial que lo hace más natural y además tiene en cuenta las colisiones con las paredes de la sala de forma automática.

La velocidad del movimiento del jefe está determinada por un **Sistema de energía**. La cantidad de energía se encuentra almacenada en la variable **energy**, la cual empieza inicializada con el mismo valor que el atributo **maxEnergy** de la clase. Para calcular la velocidad, se multiplica un valor de velocidad inicial por la proporción entre energy y una tercera variable llamada **baseEnergy** que contiene el valor inicial de maxEnergy. Para modificar las variables del sistema de energía se deben utilizar los métodos **DecrementEnergy**, que decrementa energy en el porcentaje introducido por parámetro, y **IncrementMaxEnergy**, que incrementa maxEnergy en el porcentaje introducido e iguala el valor de energy al nuevo valor de maxEnergy. Ambos métodos calculan también el nuevo valor de la velocidad .

El jefe incluye dos métodos que sirven para activar y detener su ejecución: **Activate** y **StartDestruction**. Ambos métodos funcionan modificando el valor de unas variables (active y destroy) que indican si el jefe se encuentra activo o si ha sido destruido. Estas variables son tomadas por el componente **animator** para determinar que animaciones reproducir.

Adicionalmente a su comportamiento más físico, el jefe tiene una serie de efectos de sonido que debe reproducir en momentos concretos de su comportamiento. Estos sonidos se reproducen mediante una serie de funciones públicas:

- PlayStartSound, que reproduce un “rugido” cuando el jefe entra en escena.
- PlayBallSound, para el sonido de impacto entre la pelota y el jefe.
- PlayHurtSound, que es el sonido que emite el jefe cuando la pelota golpea la puerta.
- PlayDefeatSound, para el sonido que el emite al ser destruido.

3.7.3 Inteligencia Artificial

El componente **BossAI** implementa el patrón de comportamiento del jefe, determinando cuándo y en qué dirección debe moverse el jefe, que animación debe reproducirse y cuando reproducir los efectos de sonido. La separación de esta inteligencia artificial de la implementación de las acciones del jefe se realizó principalmente para facilitar

su desarrollo: de esta forma se podía determinar en caso de error o de comportamientos no intencionados si se trataba de un problema de la inteligencia artificial o de la implementación del movimiento. Adicionalmente, este sistema en dos piezas permitiría cambiar la inteligencia artificial con facilidad para poder construir jefes con distintos comportamientos.

La función principal de la inteligencia artificial es la de mover al jefe en dirección a la pelota, de forma que pueda detenerla antes de que colisione con la puerta. Este movimiento se realiza a través del método **MoveDirection** el componente **BossController**, el cual requiere del ángulo del movimiento. La inteligencia artificial obtiene el ángulo con el método privado **AngleToBall**, el cual, usando la posición de la bola, el jefe, y el vector normal al plano de la puerta es capaz de calcular el ángulo entre el jefe y la proyección de la pelota en la superficie de la puerta.

El movimiento del jefe no es la única función de su inteligencia artificial. Para dar la impresión de que el jefe se comporta con una mínima inteligencia, su comportamiento incluye fases de reposo intercaladas entre las de movimiento, así como animaciones para su introducción en el nivel, para cuando recibe daño y para su destrucción. La implementación de estos comportamientos se realiza mediante una **máquina de estados finitos** (figura 3.32 dotada de los siguientes estados:

- **Out:** El jefe espera fuera del área de juego, inmóvil, a la espera de que la variable **Active** de **BossController** esté en “verdadero”.
- **Intro:** El jefe entra en la sala, moviéndose hacia su posición inicial en el centro de la sala. Acto seguido, el jefe realiza su animación inicial, moviéndose ligeramente y emitiendo sonidos. Este movimiento se implementa mediante el componente **Animator** en lugar de BossController ya que el jefe debe atravesar el techo sólido de la sala.
- **Wait:** En este estado, el jefe permanece inmóvil a la espera de un estímulo que le haga cambiar al estado **Move**. El estímulo es que la pelota se mueva en dirección a la puerta. El jefe cuenta con un **tiempo de reacción**, por lo que una vez que entra en este estado deberá permanecer en el durante al menos ese tiempo antes de poder cambiar al estado **Move**
- **Move:** El jefe se mueve en dirección a la pelota utilizando los métodos de BossController. El movimiento se detendrá si se consigue golpear la pelota o si la pelota golpea la puerta.
- **Hurt:** En este estado el jefe permanece inmóvil mientras se reproduce una animación de daño. Tras esta breve pausa, el jefe retorna al estado **Wait**.
- **Death:** Cuando la puerta se abre, el jefe realiza una animación de destrucción mediante el componente **Animator**, para después volverse invisible mientras

3. ARQUITECTURA

produce una explosión de partículas (instanciando un objeto de la clase **ParticleExplosion**.

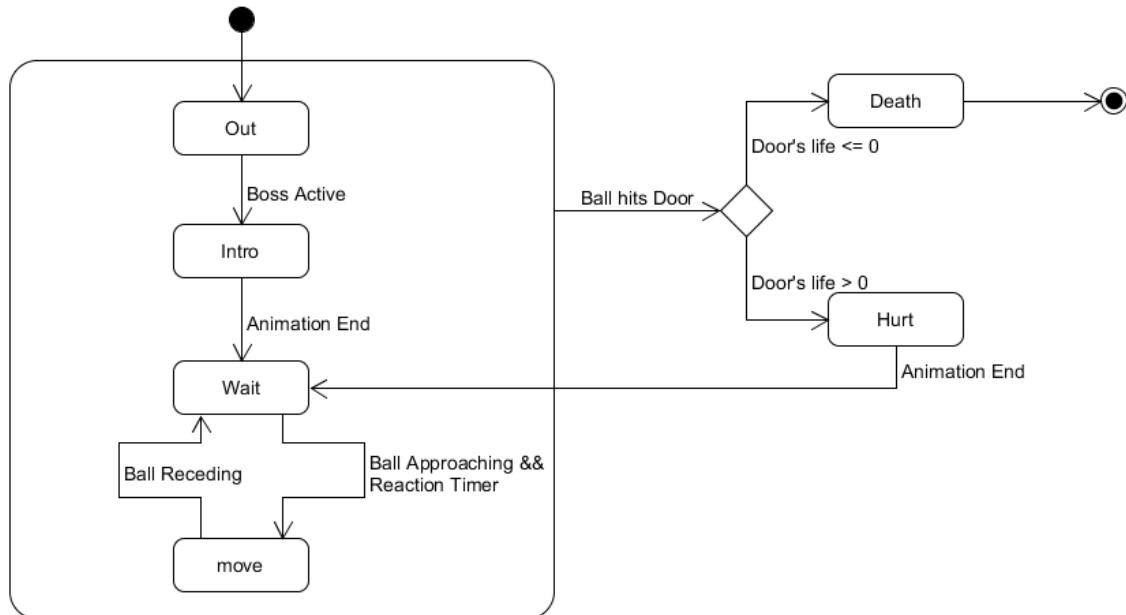


Figura 3.32: Diagrama de estados del jefe.

3.8 Patrones de Diseño

Un **Patrón de diseño** es, en el contexto de la ingeniería de software, una solución reutilizable para un problema frecuente. Los patrones de diseño se presentan como descripciones o plantillas de cómo debe resolverse el problema más que como fragmentos de código que puedan añadirse directamente a un programa, de modo que el programador pueda aplicarlos a distintos contextos dependiendo de sus necesidades.

El concepto de patrón de diseño aplicado al diseño de software surgió en el libro **Design Patterns: Elements of Reusable Object-Oriented Software** [EB95], donde se introduce no solo la idea de los patrones de diseño, sino también algunos de los patrones más utilizados como **Singleton** o **Abstract Factory**. Este libro se inspira en una obra anterior llamada **A Pattern Language** de Christopher Alexander, una obra de estructura similar en la que se describen patrones de diseño en arquitectura [Nys04].

En los siguientes apartados se describirán los patrones de diseño aplicados en el desarrollo de Virus Breaker.

3.8.1 State

El patrón **State** permite que un objeto pueda cambiar su comportamiento dependiendo de su estado, de forma que parece que el objeto ha cambiado [EB95].

Este patrón se utiliza en clases cuyo comportamiento varía mucho en función de su **estado interno**. Para encapsular cada comportamiento correctamente y evitar que parte de ellos se ejecuten en estados incorrectos, se construyen clases que implementan una **interfaz estado**. La clase principal tiene un **estado activo** cuyos métodos llama durante su ejecución. El código común permite a cada estado **cambiarse por otro** cuando se cumplen ciertas condiciones internas. En el diagrama 3.33 se puede ver una estructura de clases que ejemplifica este patrón.

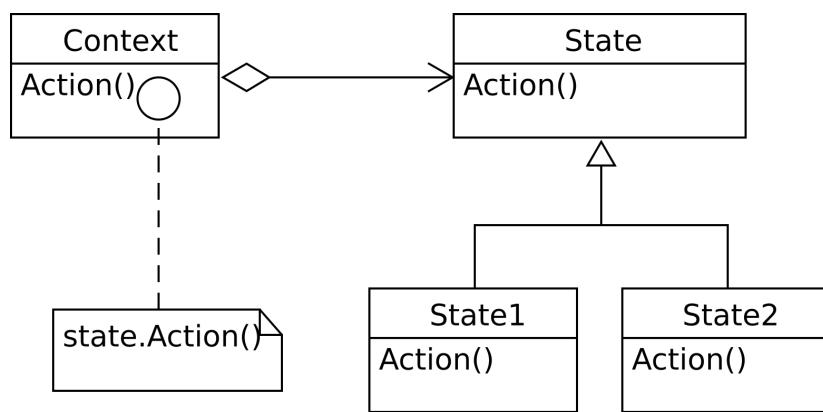


Figura 3.33: Diagrama de clases del patrón State.

En el ámbito del videojuego, State se utiliza en la implementación de máquinas de estados, especialmente las dedicadas a la inteligencia artificial. Debido a la aparición de otras técnicas como los **árboles de comportamiento**, el uso de este patrón no es tan prominente como lo fue en el pasado, pero sigue siendo muy útil en la lectura de entrada, en la navegación de menús o en el procesamiento de textos.

En virus breaker, este patrón se utiliza en la implementación del comportamiento del personaje principal, la pelota y el jefe. Esta implementación se realiza mediante plugin⁴ externo. Además, el componente **Animator** de Unity utiliza una máquina de estados finitos para la gestión de animaciones.

3.8.2 Component

El patrón componente permite a una sola entidad cubrir múltiples dominios sin provocar el acoplamiento entre los distintos dominios cubiertos [Nys04].

La forma de implementar este patrón es crear clases **componente** que encapsulan los métodos y atributos que se utilizan para tratar con cierto dominio del programa

⁴<https://github.com/thefuntastic/Unity3d-Finite-State-Machine>

3. ARQUITECTURA

(como podría ser la entrada del jugador, el sonido o las físicas). Cuando la funcionalidad de una clase afecta a varios dominios del programa, se le asignan los componentes de dichos dominios, en lugar de acceder a los dominios directamente, lo que provocaría un serio acoplamiento del código. Un diagrama con una estructura de clases de ejemplo de este patrón puede verse en la figura 3.34.

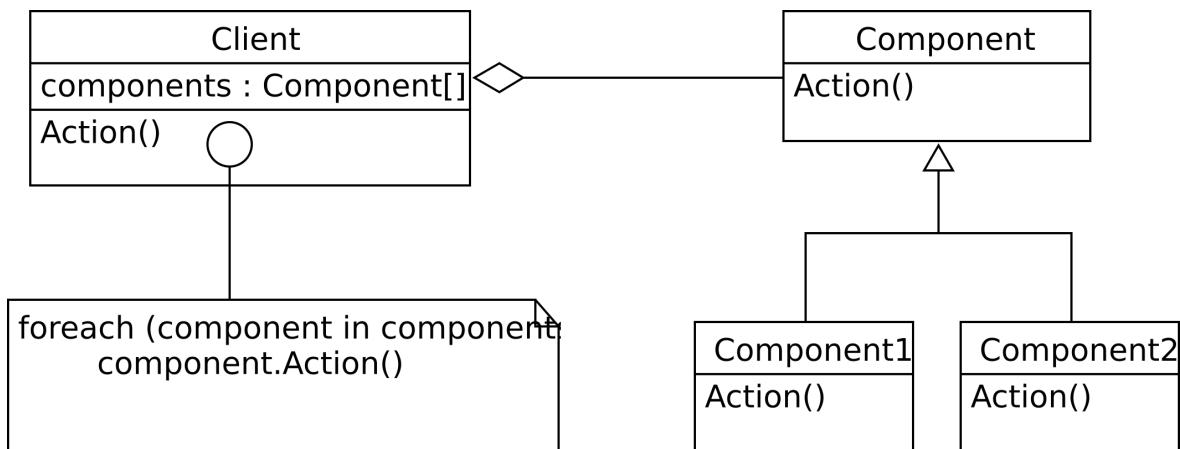


Figura 3.34: Diagrama de clases del patrón Component.

Cuando este patrón se utiliza de forma generalizada en un código, las clases pasan a convertirse en **contenedores de componentes** cuya funcionalidad se deriva completamente de los componentes que tiene asociados. La única función de estas clases suele ser la de coordinar y gestionar el paso de mensajes entre los distintos componentes.

Actualmente, varios motores de juegos utilizan este patrón en la implementación de entidades del juego, dado que una misma entidad necesita casi siempre controlar varios dominios del programa. Algunos de estos motores son Microsoft XNA⁵ y Unity3D.

3.8.3 Prototype

Este patrón permite especificar tipos de objetos utilizando una **instancia prototípico** como referencia. Los nuevos objetos se crean realizando copias del prototípico [EB95].

Los prototípicos se utilizan para ocultar los detalles de la producción de objetos de una clase a sus usuarios, dotando a las instancias de la capacidad de copiar de sí mismas, normalmente a través de un método **clone**. Por ello, este patrón resulta especialmente útil cuando el programa necesita crear diferentes tipos de objetos dependiendo de algún parámetro en tiempo de ejecución o para encapsular procesos de producción de instancias complejas. En el diagrama 3.35 se puede ver una estructura de clases que ejemplifica este patrón.

En Unity3D, el patrón Prototype se implementa en su sistema de **objetos prefabricados**.

⁵<https://msdn.microsoft.com/es-ES/dn308572>

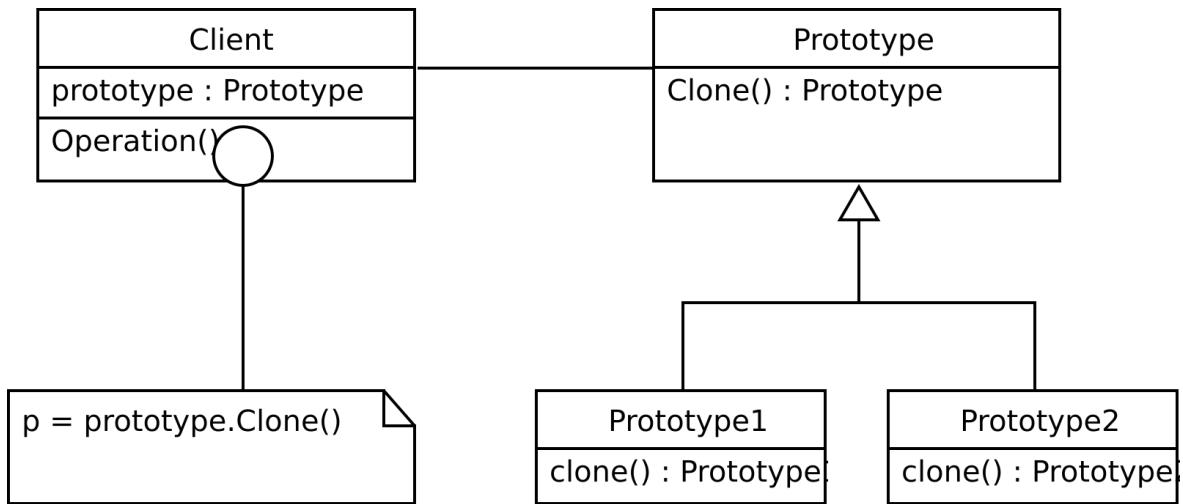


Figura 3.35: Diagrama de clases del patrón Prototype.

bricados. Este sistema permite guardar los GameObjects creados en el editor a modo como un assets del proyecto de tipo **Prefab**. Los prefabs pueden ser instanciados desde el editor o en tiempo de ejecución mediante el método estático **Instantiate**, el cual puede aceptar cualquier GameObject, aunque no se encuentre guardado como prefab. Este sistema simplifica la creación de nuevos GameObjects que, al estar formados por múltiples **Componentes** asociados, sería desmedidamente engorrosa.

En Virus Breaker, se utilizan prefabricados en múltiples elementos del juego: los ladillos, la paleta y el jefe son todos objetos prefabricados que se instancian en tiempo de ejecución cuando son necesario. Adicionalmente, otros objetos como la pelota, el jugador y la sala se encuentran guardados como prefabs a modo de copia de seguridad.

3.8.4 Singleton

Este patrón asegura que una clase tenga una sola instancia, y provee acceso global a ella [EB95].

Los singleton se utilizan en sistemas informáticos donde es importante que ciertas clases tengan una única instancia disponible en todo momento. Es útil para sistemas globales a los que deben acceder múltiples objetos, como un sistema de archivos o un gestor de dispositivos externos. Para asegurar la unicidad y accesibilidad de la instancia, el constructor de la clase es privado y en su lugar se utiliza un método estático que da acceso a una instancia de la clase, o la crea en caso de que aún no esté instanciada. La figura 3.36 muestra el diagrama de la estructura mínima una clase Singleton.

El patrón cuenta con mala fama debido a que es común utilizarlo de forma incorrecta, añadiendo más restricciones de las requeridas e introduciendo **estado global**, que

3. ARQUITECTURA

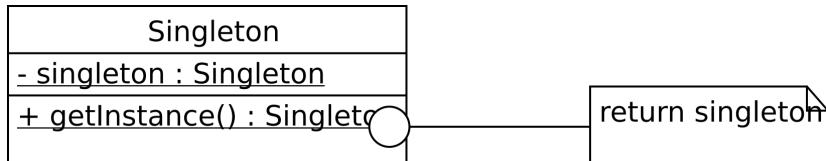


Figura 3.36: Diagrama de clases del patrón Singleton.

dificulta la depuración del código y aumenta la propensión a errores.

En este proyecto, se utiliza este patrón en la clase **EscapeController**, el componente que permite salir de la aplicación en cualquier momento pulsando la tecla escape. Al tratarse de un componente, el patrón no puede aplicarse de forma literal por dos motivos:

1. El **constructor** de un componente no puede ser modificado ya que interferiría con los sistemas de Unity. En su lugar, en el evento **Awake** que es invocado al principio de la ejecución revisa si ya existen componentes de esa clase, y lo destruye en caso afirmativo.
2. Al ser un componente, necesita **estar asignado a un GameObject** para funcionar, los cuales se destruyen por defecto durante los cambios de escena. Para evitarlo, durante la instanciación se llama a la función **DontDestroyOnLoad** que permite que el objeto persista entre escenas.

Capítulo 4

Resultados

4.1 Proceso de Desarrollo

El desarrollo de Virus Breaker se dividió en varias **iteraciones** con el objetivo de reducir los riesgos del proyecto. El objetivo de cada iteración es obtener, a partir del resultado anterior, un **producto funcional** más sofisticado que el de la iteración anterior. Al planificar el desarrollo de esta forma, es más fácil adaptar el diseño a problemas detectados durante el desarrollo y además permite que, en caso de tener que detener el proceso antes de su completitud, se puede obtener un resultado mínimamente viable.

Originalmente, el proyecto se planeó en **iteraciones de un mes**. Al finalizar cada iteración, se utilizaban los datos obtenidos para planificar las iteraciones siguientes. Dividir las iteraciones en unidades de un mes permitía que las de trabajo fuesen lo suficientemente pequeñas como para poder aplicar cambios en la planificación de forma efectiva, al mismo tiempo que permitía desarrollar secciones del juego lo suficientemente grandes como para poder obtener información útil sobre el progreso.

Sin embargo, el desarrollo del proyecto tuvo que ser detenido varias ocasiones debido a las exigencias de los estudios y al comienzo de las prácticas laborales. Por culpa de esto, las distintas iteraciones se encontraban separadas por **periodos de inactividad**. Esto obligó la inclusión **iteraciones de refresco** en las que el proyecto debía ser revisado para volver a familiarizarse con el código y las convenciones de desarrollo.

Al final, las iteraciones en las que se dividió el proyecto fueron muy distintas a las planeadas en un primer momento y además tenían una duración variable. En los siguientes apartados se describen las distintas etapas del desarrollo.

4.1.1 Prototipo

Un **prototipo** es, en el ámbito del desarrollo de videojuegos, es una implementación parcial del producto final que se utiliza para probar ideas y conceptos antes de pasar a la producción completa. El uso de prototipos permite encontrar fallos de diseño difíciles de descubrir sobre el papel sin arriesgarse a encontrarlos durante la producción, cuando su corrección sería tremadamente costosa.

4. RESULTADOS

El prototipo de Virus Breaker consistía en los elementos básicos del juego: la pelota, el jugador, la sala y los ladrillos básicos. Todos estos elementos se encontraban reducidos a su mínima expresión con el objetivo de poder ser implementados en el menor tiempo posible. De la misma forma, el juego carecía tanto de **condición de victoria como de derrota**. Las diferencias entre esta versión y la versión definitiva son, agrandes rasgos, las siguientes:

- La pelota no producía ninguno de los **efectos de partículas** actuales y su comportamiento a la hora de colisionar era mucho más simple. La pelota tampoco tenía puntos de golpe, por lo que no podía ser destruida.
- El **personaje principal** carecía de modelo (era un prisma cuadrangular) y animaciones. Aunque su movimiento era similar al actual, la paleta hacia rebotar a la pelota de forma más simplista. Tampoco podía ser derrotado ni confundido.
- La **sala** carecía de puerta (eran cuatro paredes) y utilizaba una textura por defecto en todas sus caras. Ninguno de sus elementos tenía más comportamiento que ser sólido físicamente.
- Solo había **un tipo de ladrillo**, el básico, que carecía de textura o efectos visuales. Su única función era la de ser destruido si lo golpeaba la pelota. Estos ladrillos se colocaban manualmente en la puerta mediante el Editor de Unity.
- La acción del juego se desarrollaba en **una única escena**.
- No había **interfaz gráfica de usuario**.

Una captura de este prototipo puede verse en la figura 4.1.

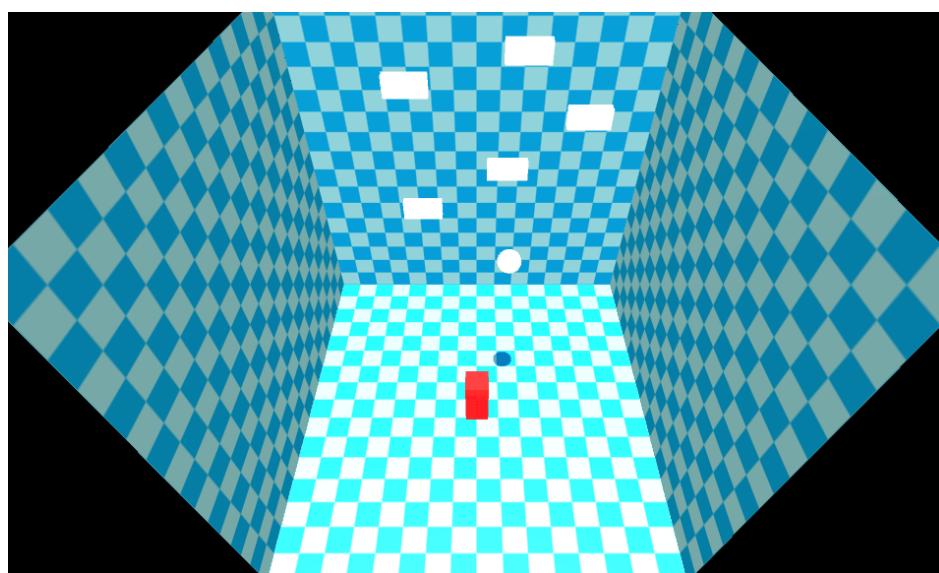


Figura 4.1: Captura del prototipo (Recreación).

Esta complementación del juego fue muy útil para calcular los valores exactos de

velocidad de la pelota y el jugador, importantes para lograr una sensación de juego agradable. Por otro lado, fue posible identificar unos problemas fundamentales con el movimiento de la pelota.

En primer lugar, al rebotar con la “puerta”, era bastante común que acabase rebotando con la pared del sur de la sala y su movimiento se volviese mucho más difícil de controlar para el jugador. En segundo lugar, era muy difícil controlar la pelota con la paleta, primero la extremada dificultad de darle (la pelota tendía a quedar entre el jugador y la paleta) y en segundo porque no había forma de controlar la dirección de salida de forma efectiva. Estos problemas pudieron ser **resueltos de forma efectiva** añadiendo cambios a los comportamientos de la pelota, el jugador y la paleta que no se encontraban en el diseño inicial.

El desarrollo de este prototipo duró un mes, el tiempo esperado del desarrollo esperado. El ritmo del desarrollo fue de aproximadamente un elemento por semana, aunque el personaje principal necesitó de tiempo extra.

4.1.2 Mecánicas de Juego Complementarias

El prototipo resultante de la etapa anterior podía considerarse lo que en diseño de juegos se conoce como **juguete** un sistema interactivo carente de objetivos [Bur15]. Para ser considerado un juego, el jugador deberá tener **objetivos**, los cuales se alcanzarán mediante toma de **decisiones significativas**. Lejos de ser un fallo del proceso de desarrollo, comenzar implementando un juguete permite pulir correctamente su sistema de control para que sea fácil y divertido de usar.

Por consiguiente, el objetivo será el de dotar al proyecto de los **mecánicas complementarias** que añadan objetivos. El objetivo que se eligió para el juego era el de superar todos los niveles del mismo, al mismo tiempo que se evita un **estado de derrota**. La implementación de los objetivos se realizó mediante la adición de los siguientes elementos y mecánicas:

- **La puerta** de la sala, y, por consiguiente, la condición de victoria del nivel.
- **Los puntos de golpe de la pelota**, y la condición de derrota asociada a dichos puntos.
- **Los estados adicionales del personaje principal**, es decir, el estado de confusión y el estado de derrota.
- **Distintos niveles** y transiciones entre ellos.

El desarrollo de esta etapa comenzó con la implementación de los nuevos estados del personaje principal, seguida de la implementación del sistema de puntos de vida de la pelota y sus correspondientes estados adicionales. Para facilitar su desarrollo, se

4. RESULTADOS

utilizó un plugin de **máquina de estados finitos** para Unity¹ que permitía dividir la ejecución de los componentes en diferentes estados.

En este estado, las escenas de victoria y derrota no estaban implementadas, por lo que el juego empezaba directamente en el primer nivel y, en la derrota, el nivel simplemente se reiniciaba. De la misma forma, el sistema de guardado tampoco estaba implementado, por lo que cada vez que se empezaba el juego se volvía al primer nivel.

Una vez terminados los nuevos estados, el desarrollo se centró en la construcción de niveles. Una serie de niveles fueron diseñados en papel, y después fueron implementados mediante el editor de Unity en forma de distintas escenas. Según se iban necesitando para la construcción de niveles, se implementaron el resto de los tipos de ladrillos, así como su función para asignarles colores.

Junto con los niveles, se implementó la puerta del juego con un comportamiento básico. Esta disponía de puntos de vida junto con la función para realizar cambios de escena. Sin embargo, la transición entre distintas escenas era un cambio brusco debido a la falta del componente **FlashTransition** que no fue implementado hasta mucho después.

La implementación de niveles era **un proceso lento y propenso a fallos**: cada vez que se colocaba un ladrillo había que ajustar su posición a la cuadricula de forma manual lo que ralentizaba enormemente el proceso de desarrollo. De la misma forma, para los colores de los ladrillos debían ser seleccionados uno a uno y el resultado no podía verse en el editor de Unity, por lo que había que ejecutar la prueba del juego para comprobar que los ladrillos tenían los colores correctos.

En adición a estos problemas, tener los niveles implementados en distintas salas era tremadamente ineficiente ya que los cambios realizados en objetos comunes de un nivel debían ser propagados manualmente al resto de niveles. El problema se intentó paliar con el uso de **objetos prefabricados** (figura 4.2), pero debido a las limitaciones de los objetos prefabricados hicieron que fuese solo una solución parcial.

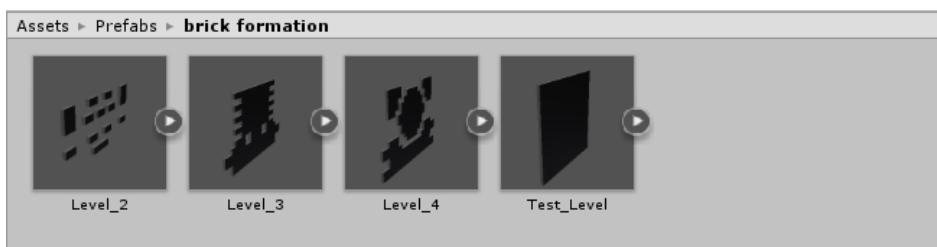


Figura 4.2: Guardar los niveles como objetos prefabricados fue una de las alternativas probadas.

¹<https://github.com/thefuntastic/Unity3d-Finite-State-Machine>

La duración de esta iteración fue de dos meses, más tiempo del esperado. La causa de esta tardanza fue principalmente los problemas en la implementación de niveles. Fue en este punto donde se decidió utilizar un editor de niveles externos, ya que el incremento en el tiempo de desarrollo del sistema de carga de niveles se vería compensado por la aceleración en la implementación de niveles.

4.1.3 Cambios Estéticos

El objetivo de esta iteración era la de pulir estéticamente el juego de forma que pudiese ser publicado como una **alpha abierta** en internet y así obtener feedback de usuarios. Los principales cambios añadidos en esta iteración serían texturas y modelos a los distintos objetos, efectos de sonido a las colisiones, animaciones, corrección de errores y las pantallas de inicio y final del juego.

En primer lugar, se elaboraron **texturas para los distintos objetos**: los ladrillos, la puerta, las paredes y el personaje. Las texturas eran imágenes de baja resolución que imitaban el estilo artístico de los juegos de los años 80 y 90, no muy distintas de texturas definitivas (figura 4.3). Mientras que la sala y los ladrillos conservaron sus simple modelos por defecto, fue necesario elaborar modelos para el personaje principal y la puerta. El personaje principal necesitaba un modelo que lo hiciese más expresivo que un simple prisma, pero la puerta necesito un modelo por cuestiones técnicas: los paneles de la puerta, a pesar de ser prismas rectangulares como por ejemplo los ladrillos, necesitaba que cada una de sus caras tuviese una textura distinta, algo difícil de implementar en los modelos por defecto de Unity. Ambos modelos fueron elaborados usando el programa **Blender3D**²

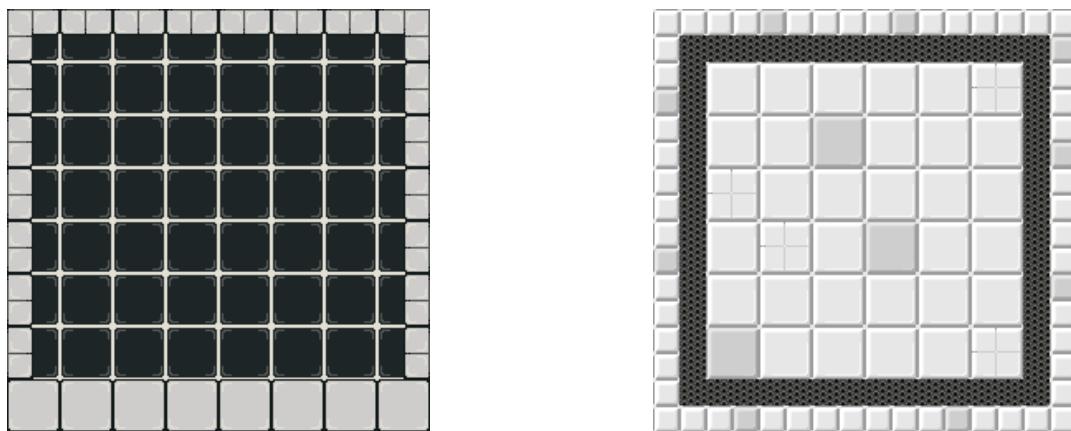


Figura 4.3: Comparativa entre la textura antigua (izquierda) y moderna (derecha) del suelo de la sala.

La pelota no recibió ningún modelo ni textura, ya que su aspecto no era muy distinto del que se buscaba, pero si se le añadieron los efectos de partículas que aún se conservan

²www.blender.org

4. RESULTADOS

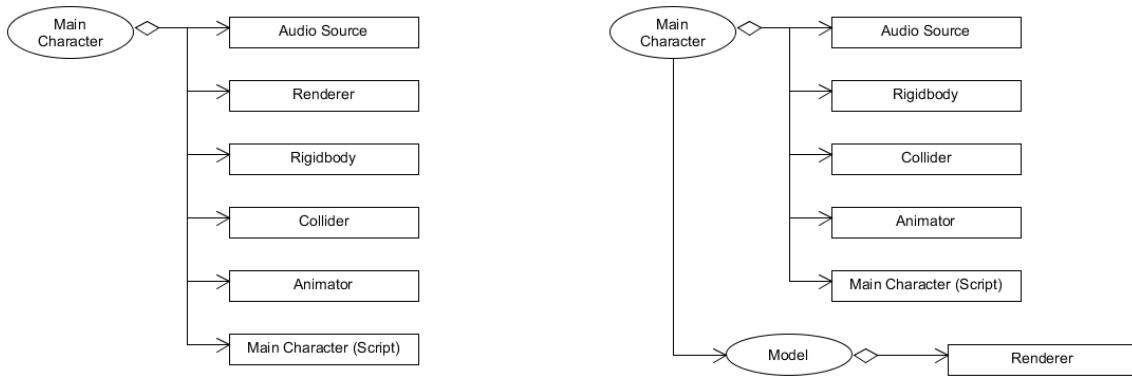


Figura 4.4: Estructura de MainCharacter antes (izquierda) y después (derecha) de la modificación.

en la versión final. La paleta tampoco recibió modelo o textura, pero se ajustaron los valores de su **material** para darle un aspecto semitransparente.

El siguiente paso en esta iteración era la inclusión de **animaciones** a los distintos objetos del juego. Específicamente, el personaje principal, la pelota y la puerta fueron los objetos que recibieron animaciones nuevas para volverlos más expresivos. Sin embargo, en el proceso de adicción de animaciones se descubrió que los componentes **Animator** y **Rigidbody** entraban en conflicto debido a que ambos modificaban valores de posición y rotación de los objetos a los que estaban asignados.

Para solucionar este problema fue necesario replantear **la implementación estos objetos**, separando sus componentes en dos: un objeto principal invisible que contiene el comportamiento del objeto con un objeto asociado que se utiliza para contener los modelos y texturas. En la figura 4.4 se puede ver el cambio realizado en el personaje principal, el cual es similar al que se aplicó a otros objetos. A pesar de parecer un cambio pequeño, la modificación de la estructura de GameObjects obligó a modificar los componentes asociados y volver a generar las animaciones.

Aparte de los fallos causados por las animaciones, también se encontraron **fallos en la pelota y el personaje principal**. Se trataba de fallos en el cálculo de trayectoria de la pelota y en los cambios de estado del personaje principal. Solucionar estos problemas también consumió una cantidad importante de tiempo. En el proceso de revisar y corregir la pelota también se añadieron funciones para emitir efectos de sonido.

El desarrollo de esta iteración concluyó con la implementación de **las pantallas de título y de fin del juego**. En esta versión, estas pantallas estaban implementadas íntegramente mediante el sistema de interfaz de usuario de Unity, aunque su comportamiento era idéntico al actual. La figura 4.5 muestra cómo se veía la pantalla de título en esta versión.

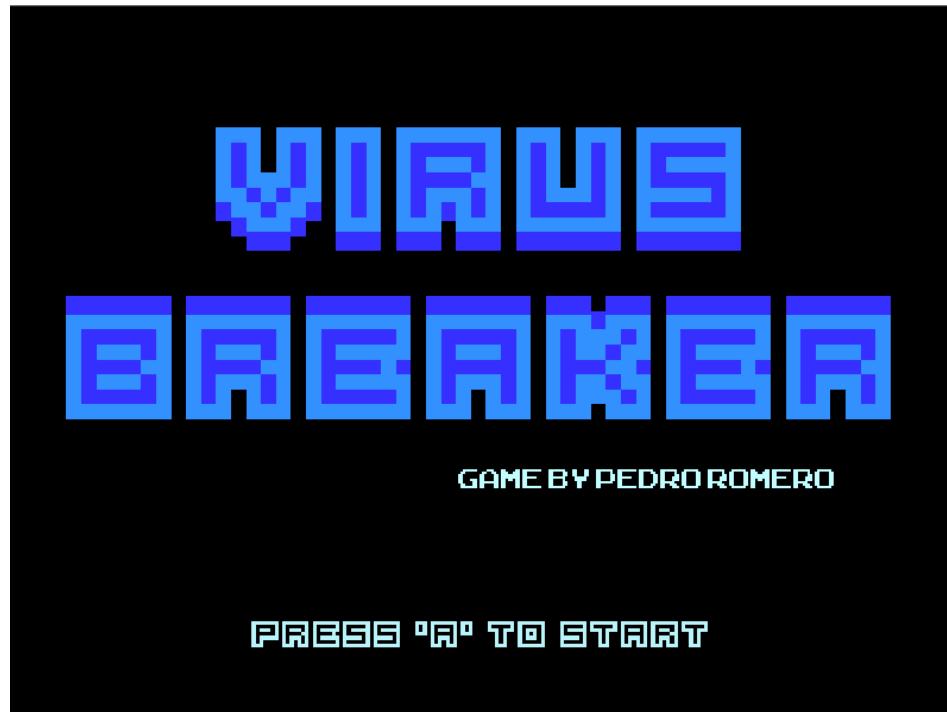


Figura 4.5: Pantalla de título antigua.

El desarrollo de esta etapa cubrió varios meses, debido a los problemas encontrados durante su realización y a interrupciones externas. Una vez finalizada, la versión del juego obtenida en esta etapa fue publicada en la página [itch.io](#)³, un portal dedicado a la distribución de juegos. El juego fue marcado como título en desarrollo y lanzado gratis. Debido a que no se realizó mayor publicidad sobre el juego que su publicación, apenas se recibieron visitas.

4.1.4 Carga de Niveles

El uso de una única escena de juego con un sistema de carga de nivel se eligió por encima del uso de varias escenas con diferentes configuraciones debido principalmente al gran número de elementos comunes que existen entre los distintos niveles (la geometría de la sala, el personaje, la pelota, la puerta...). De haberse optado por utilizar múltiples escenas, la repetición de elementos **supondría un gasto innecesario de recursos** y generaría problemas a la hora de realizar **modificaciones sobre los elementos comunes**, cuyos cambios tendrían que propagarse manualmente en todos los niveles.

En una primera versión, se planteó el uso del programa **Tiled**⁴ (figura 4.6), un editor de mapas de propósito general que permite la edición de mapas basados en baldosas o “tiles” utilizando un sencillo procesador del lenguaje. Los mapas desarrollados ge-

³[itch.io/](#)

⁴[http://www.mapeditor.org/](#)

4. RESULTADOS

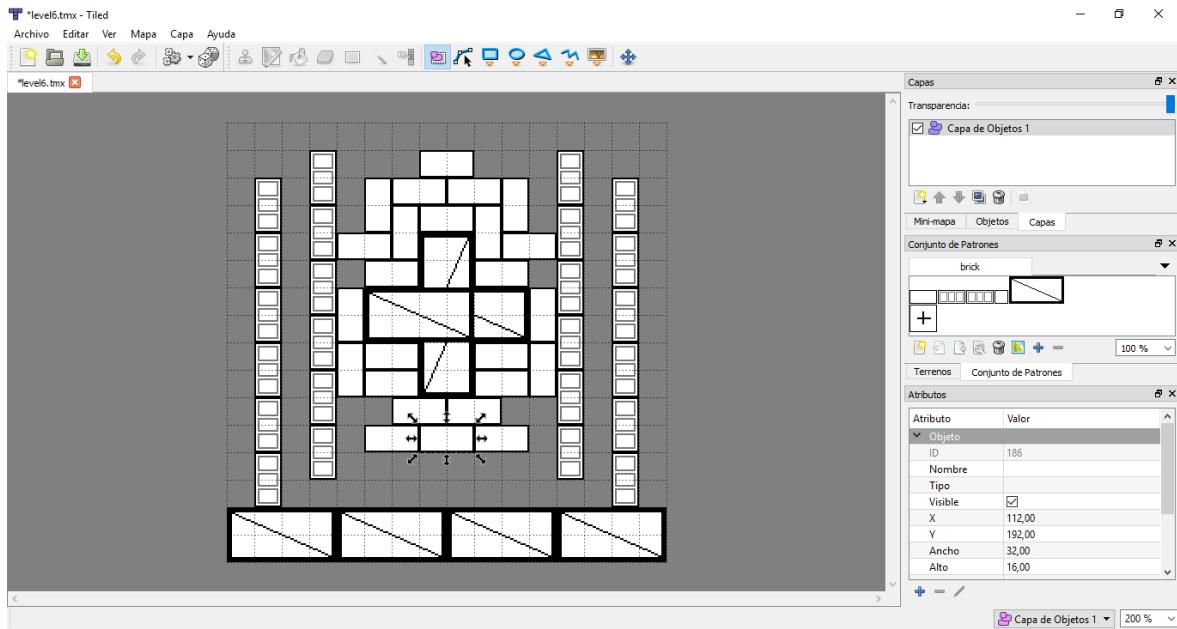


Figura 4.6: Captura de Tiled durante el diseño del nivel seis del juego.

nerados con tiles eran exportados en formato **XML**. Para cargar los mapas en Unity se implementó un sencillo procesador del lenguaje XML que leía los ficheros y usaba su información para instanciar los ladrillos en los lugares correctos y con su color adecuado.

Sin embargo, utilizar Tiled **suponía varios inconvenientes**. En primer lugar, al ser un programa de propósito general, Tiled contaba con mucha **funcionalidad adicional como que resultaba innecesaria** para el proyecto y que por tanto solo ralentizaba la producción de niveles. Al mismo tiempo, este programa **carecía de cierta de ciertas funciones clave** para la elaboración del nivel. Por ejemplo, no era posible previsualizar la selección individual de colores de los ladrillos, la cual tenía que hacerse escribiendo los valores RGB del color en cada ladrillo.

El sistema de carga de niveles actual fue desarrollado para remplazar al sistema basado en Tiled, ya que se el tiempo de desarrollo del nuevo sistema sería menor que el de terminar todos los niveles con el sistema antiguo. Junto con el nuevo sistema, también se aisló la carga de niveles del resto de la funcionalidad de la sala creando dos componentes: **LevelGenerator** para la carga de niveles y **Room** para la coordinación de los elementos de la sala. La figura 4.7 muestra con exactitud como se produjo esta división.

Ambas iteraciones del sistema de carga de niveles se completaron en aproximadamente un mes (cada una). Entre ambas iteraciones hubo un periodo largo de inactividad.

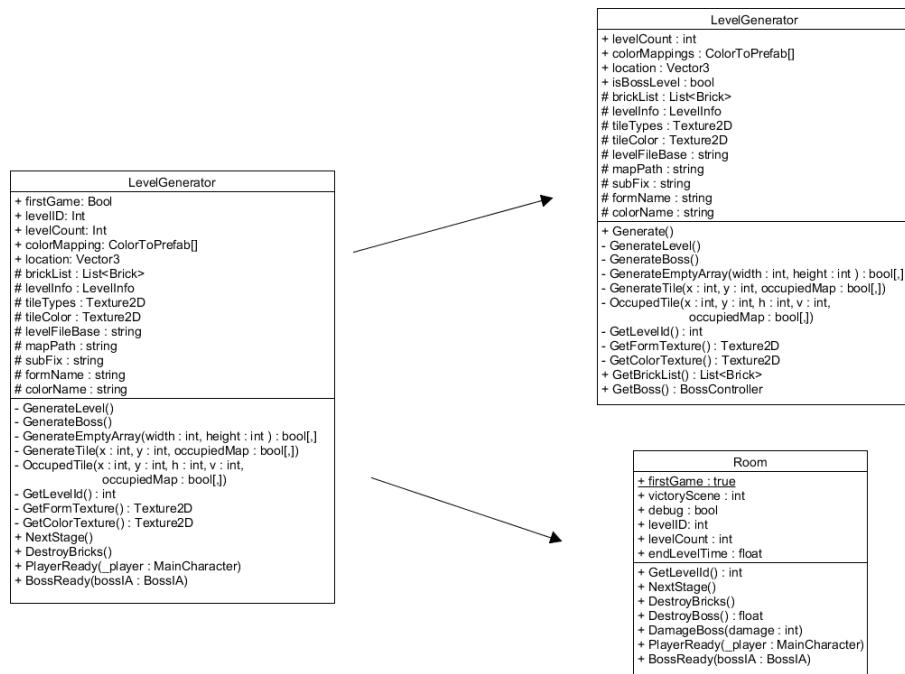


Figura 4.7: Extracción de la funcionalidad de la sala de LevelGenerator.

4.1.5 Jefe Final

Durante el diseño del juego, se plantearon varias **mecánicas adicionales** que serían añadidas una vez las mecánicas básicas fuesen implementadas para ampliar dar más variedad al juego. Debido a las limitaciones de tiempo del proyecto, solo una de las mecánicas adicionales podría ser implementadas. Al final se decidió implementar un jefe final para el juego.

Una vez decidido que la mecánica escogida sería la implementación de un jefe final, se diseñaron varios posibles jefes para el juego. El coste en tiempo de implementar un jefe obligaba a elegir a un solo jefe para el juego, por lo que el resto de los jefes fueron descartados una vez que se eligió al jefe actual. Estos jefes descartados eran:

- Un **Ciempiés gigante**, el cual se movía por las paredes y suelos de la sala de forma pseudoaleatoria entorpeciendo al jugador. Tendría una inteligencia artificial que le permitiría moverse sin nunca colisionar con su propio cuerpo.
- Una **Figura hecha de ladrillos**. Este jefe funcionaba de forma opuesta al actual, moviéndose para evitar que la pelota golpeara sus ladrillos, ya que el jugador ganaba si le golpeaba suficientes veces.
- Un **Mago**, el cual no interaccionaba directamente con el jugador, pero podía dificultar la partida del jugador reparando ladrillos y desviando la pelota.

En la primera etapa de su desarrollo, el comportamiento del jefe se modeló como un único componente **Boss**. Esta implementación estaba plagada de errores, el más

4. RESULTADOS

notable era que el jefe se “despegaba” de la puerta a la hora de buscar la pelota. Al tratarse de un diseño monolítico, era casi imposible aislar el error, ya que podía ser a causa de una implementación incorrecta del movimiento o de un cálculo equivocado de la dirección.

Este problema obligó a descartar la implantación del jefe y remplazarla por la estructura en dos componentes que se utiliza en la versión final. El desarrollo comenzó con la implementación de **BossController**, el componente con que modela el movimiento del jefe. Para realizar pruebas, se utilizaba un componente de pruebas que permitía enviar órdenes de movimiento a BossController mediante pulsaciones del teclado. Una vez asegurado que el movimiento carecía de fallos, se implementó la inteligencia artificial del jefe en el componente **BossIA**.

Una vez finalizado el comportamiento básico del jefe, se decidió darle una **animación de entrada en la sala y otra de derrota**, para volverlo más espectacular. A raíz de esta decisión, se elaboró también una secuencia de introducción, victoria y derrota general para todos los niveles del juego. Este sistema se montó por encima de otros eventos que ocurrían al principio y final del juego, como la carga del nivel o la animación de destrucción de la pelota.

Debido a la dificultad de implementar el jefe final, el problema con la implementación inicial que obligó a volver a implementar todo el sistema y la complicada integración con el resto del juego, esta iteración fue muy larga, ocupando varios meses.

4.1.6 Ajustes, Refactorización y Correcciones

El objetivo de esta etapa era el de corregir el código del juego y terminar los gráficos definitivos para así poder lanzar la versión final.

La corrección del código consistió en una **revisión completa** de todas las clases que lo componían para arreglar todos los fallos de programación que habían pasado desapercibidos en las etapas anteriores. La mayor parte de los fallos detectados fueron errores en las transacciones entre estados y en las llamadas entre distintos objetos. Cambien se ajustó la interfaz gráfica de usuario, que no funcionaba correctamente en todas las resoluciones.

Además de la corrección de errores, la revisión sirvió para **reorganizar y limpiar el código** del juego para hacerlo más comprensible. La limpieza se realizó principalmente para poder realizar labores de mantenimiento de forma más eficiente. Algunos de los cambios fueron:

- **Reorganización de clases:** diversas clases fueron modificadas para que su comportamiento fuese más entendible. El caso más notable fue el de la clase **Brick**, cuyas clases hijas fueron completamente reorganizadas, como puede verse en la

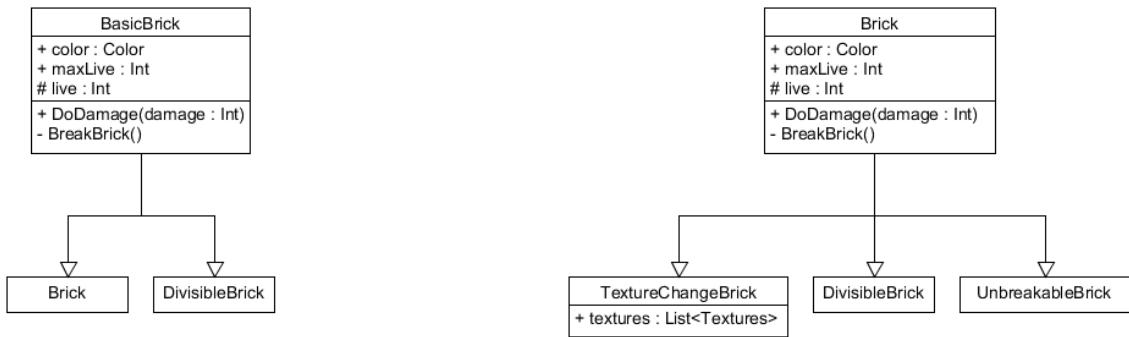


Figura 4.8: Jerarquía de la clase Brick antes (izquierda) y después (derecha) de la refactorización.

figura }.

- **Estandarización de nombres** de métodos y variables: Los nombres de las clases, métodos y variables del código fueron modificados para ser adaptados al estilo de escritura **Camel Case** que es el estándar de Unity.
- **Refactorización del código** para mejorar su eficiencia: durante la revisión se modificaron partes del código que, sin ser incorrectas, disminuían el rendimiento del programa.

Las nuevas texturas fueron elaboradas una vez terminó la limpieza del código. El objetivo del cambio de las texturas era el de **aumentar el contraste entre la sala y los elementos contenidos en ella**. El principal cambio se realizó en las texturas de la sala, que fueron completamente rediseñadas para tener un color más claro. Cambios menores fueron aplicados a las texturas del personaje principal para darles un color más saturado que contrastara mejor con el nuevo fondo claro.

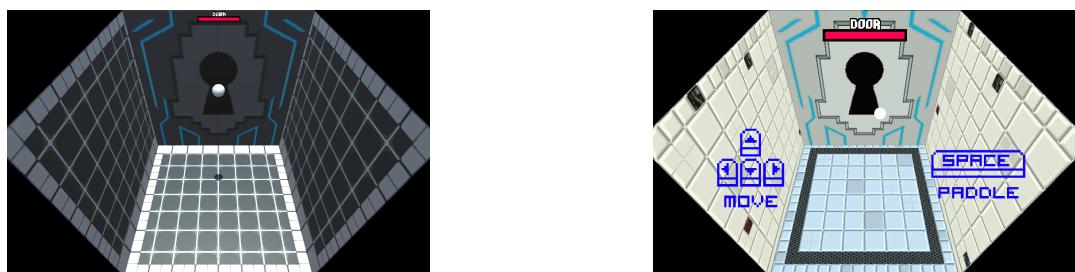


Figura 4.9: Comparativa de la sala antes (izquierda) y después (derecha) de los ajustes gráficos.

Los **valores de iluminación de la escena** de juego también fueron modificados para obtener un mejor contraste entre colores. Los cambios en la iluminación redujeron la visibilidad de la sombra de la pelota, la cual es importante para guiar al jugador, por lo que fue necesario añadir a la pelota un componente **Proyector** para emitir una

4. RESULTADOS

sombra artificial sobre el suelo.

Esta iteración duró aproximadamente un mes y sirvió como conclusión del desarrollo. La versión final producida fue publicada en itch.io y en gamesjolt.com, acompañadas de una encuesta de opinión.

4.2 Encuesta de Opinión

Para poder obtener información sobre la opinión de los jugadores acerca del juego se elaboró una **encuesta de opinión**. Esta encuesta se compone de una serie de preguntas sobre las distintas áreas del juego, así como sobre el usuario que permitirán no solo valorar la calidad del producto de forma objetiva sino también determinar el **público objetivo** del juego.

4.2.1 Descripción

Para la elaboración de la encuesta se utilizó la herramienta **Google Forms**⁵ del paquete de ofimática en line de Google. Esta herramienta permite elaborar formularios y encuestas de forma sencilla, los cuales pueden ser distribuidos por correo electrónico o mediante un enlace URL. El método de distribución mediante enlaces directos fue el utilizado para este formulario.

La **estructura de preguntas** de la encuesta tuvo que ser desarrollada desde cero al no existir ningún tipo de estándar o plantilla. Sin embargo, se utilizaron formularios similares desarrollados para otros juegos como base, analizando sus estructuras y utilizando las preguntas más recurrentes o que mejor se adaptaban al juego. La encuesta puede dividirse en dos secciones: las preguntas sobre el jugador y las preguntas sobre la experiencia del juego.

La primera parte de la encuesta está dedicada a preguntas acerca de la persona encuestada. Estas preguntas servirán para discernir a que **grupo de jugadores** pertenece el encuestado, lo que nos permitirá clasificar las respuestas. Las preguntas de esta parte son:

1. **Genero** del encuestado.
2. **Grupo de edad** del encuestado. Los posibles grupos de edad son:
 - Menor de 13.
 - Entre 13 y 17.
 - Entre 18 y 24.
 - Entre 25 y 34.
 - Entre 35 y 54.

⁵<https://docs.google.com/forms>

- Mayor de 55.
3. **Frecuencia de Juego** del encuestado. los posibles grupos de frecuencias son:
- Todos los días.
 - Un par de veces por semana.
 - Un par de veces al mes.
 - Una vez cada varios meses.

La segunda parte de la encuesta está formada por preguntas referentes a la **opinión sobre el juego** del encuestado. Estas preguntas son:

1. **¿Te pareció divertido el juego?**. La pregunta pide una valoración en una escala de 1 a 10.
2. **¿Te resultaron intuitivos los controles?**. La pregunta pide una valoración en una escala de 1 a 10.
3. **Valora la dificultad del juego**. La pregunta pide una valoración en una escala de 1 (muy fácil) a 10 (muy difícil).
4. **¿Te gustó el estilo gráfico del juego?**. La pregunta pide una valoración en una escala de 1 a 10.
5. **¿Te gustó la música y sonidos del juego?**. La pregunta pide una valoración en una escala de 1 a 10.
6. **¿Cuantos niveles superaste?**. La pregunta se responde con un campo de texto. idealmente, la respuesta será un número.
7. **¿Que te pareció la duración del juego?**. La pregunta pide una valoración en una escala de 1 a 10.
8. **¿Encontraste algún problema o error en el juego?**. La pregunta se responde con un campo de texto. Al encuestado se le pide que describa el error en caso de haber encontrado alguno.
9. **¿Comentario o sugerencia adicional?**. Esta pregunta es un espacio libre para que el encuestado exprese las opiniones que no encajen en otras secciones.

El enlace a este cuestionario fue publicado en la descripción del juego en sus dos páginas de distribución. También se publicó el enlace en las redes sociales Twitter y Reddit. Adicionalmente, el enlace fue distribuido por el grupo de contactos personales del desarrollador. Para cubrir un público mayor, la encuesta se redactó en español y en inglés.

4.2.2 Resultados

Los resultados de la encuesta **no resultaron ser tan útiles como se esperaba**. Sumando todos los medios de distribución, solo se obtuvieron 8 votos, una cantidad

4. RESULTADOS

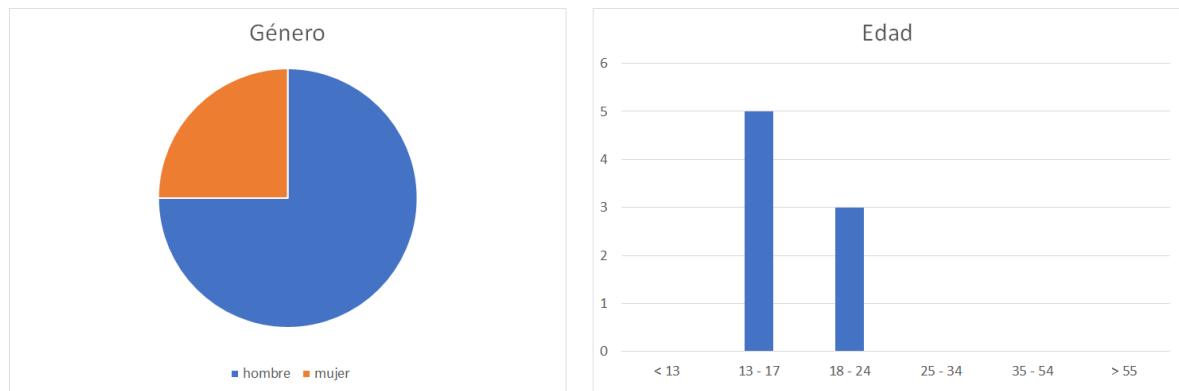


Figura 4.10: Diagramas del género (izquierda) y edad (derecha).

demasiado pequeña como para poder realizar un estudio que resulte.

De todas formas, a modo de práctica, **se realizará un análisis de los datos obtenidos** en la encuesta. El análisis se dividirá en tres secciones: la determinación del público del juego, el análisis de la opinión de los jugadores y la reseña de las quejas y peticiones más recurrentes.

Audiencia

Los datos obtenidos apuntan a que el público del juego es varones (75 % de los encuestados), de edades comprendidas entre los 13 y los 24 años, siendo ligeramente más común la franja de edad de entre 13 y 17 años (62 % de los encuestados). Las tablas de la figura 4.10 contienen estos datos. Estas cifras encajan con el perfil tradicional de jugador de videojuegos.

La gran mayoría de los encuestados (87.5 %) tienen una frecuencia de juego muy alta, con un 37.5 % que juega de forma diaria a algún tipo de videojuego. En la figura 4.11 están representados estos datos.

Opinión

Los **resultados de la encuesta de opinión** se pueden ver en la figura 4.12. En esta tabla se puede apreciar como todos los encuestados valoraron el juego de forma muy positiva, con ningún campo valorado por debajo de los cinco puntos. En promedio, la nota del juego contando todos sus apartados es de 8.7 puntos, recibiendo la nota media más baja los controles (8.25) y la más alta el apartado sonoro (9.25).

La **opinión sobre la dificultad** debe ser valorada independientemente, ya que los mejores resultados serían las notas cercanas a los cinco puntos. En la figura 4.13 se aprecia como la mayor parte de las notas se encuentran cerca de los cinco puntos, aunque un porcentaje significativo supera este valor, dando una nota media de 6.14

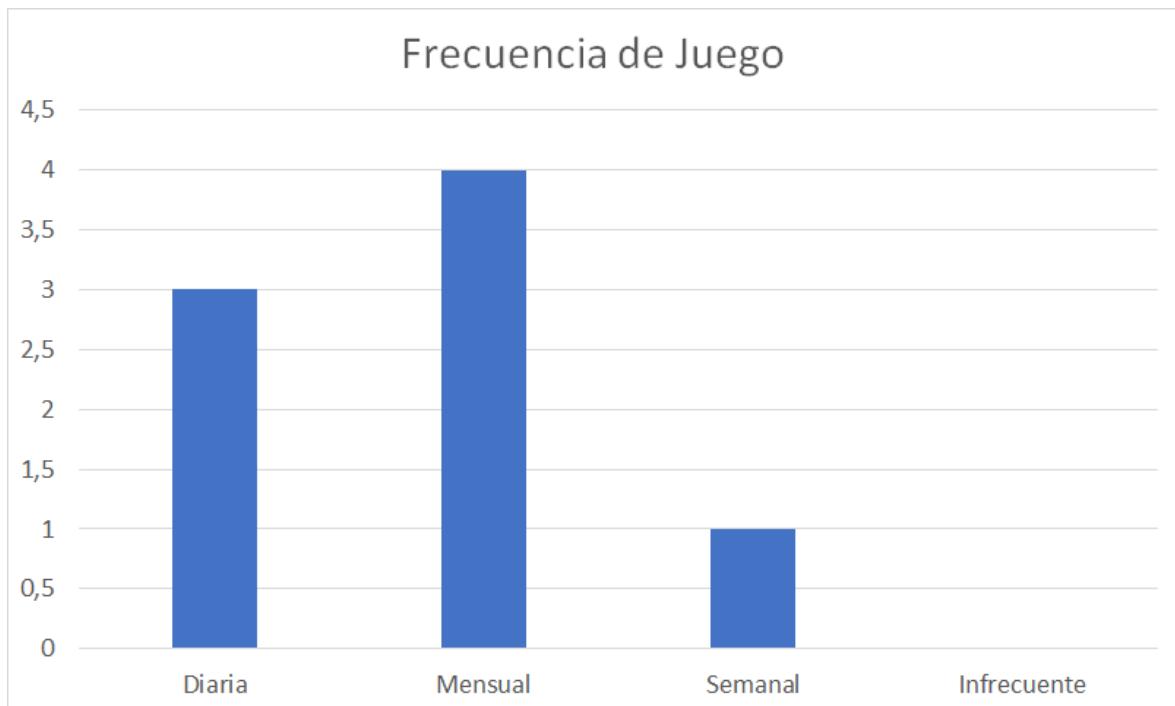


Figura 4.11: Diagramas de la frecuencia de juego.



Figura 4.12: Diagrama de la opinión.

puntos.

La cantidad de niveles superados por los encuestados se muestra en la figura 4.14 indica que 78 % d de los jugadores no consiguieron terminar el juego, siendo el nivel

4. RESULTADOS

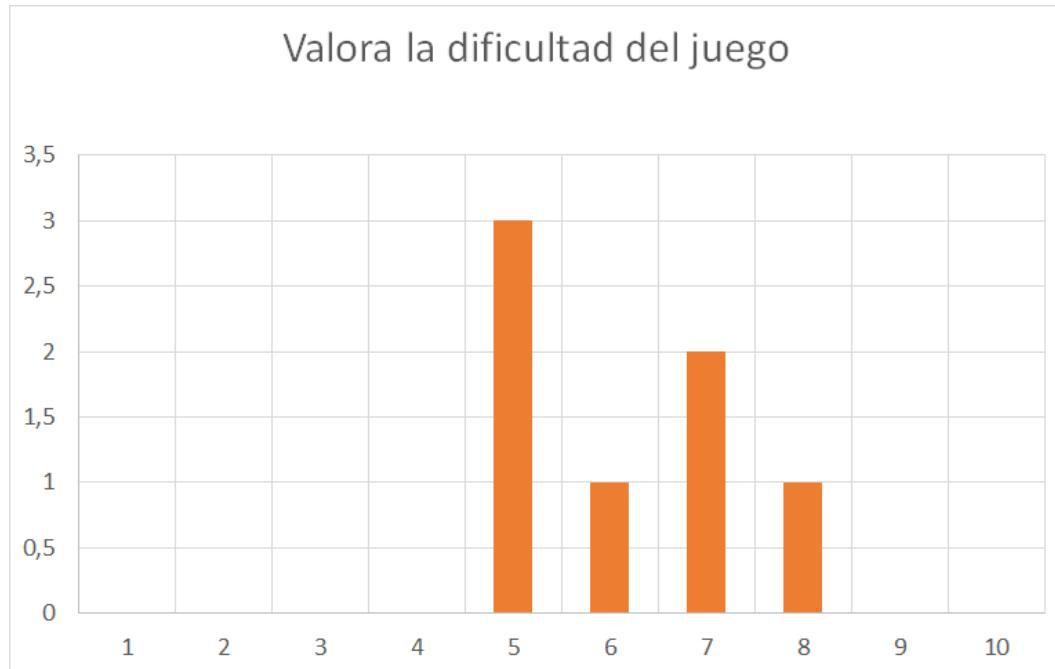


Figura 4.13: Diagramas de la dificultad.



Figura 4.14: Diagramas de niveles superados.

cuatro el principal punto donde los jugadores dejaban el juego (18 % de los encuestados).

En cuanto a las quejas y sugerencias, varios usuarios indicaron que tenían problemas con los controles del juego, específicamente con la **activación de la paleta**. Otro

usuario indicó que en la versión actual del juego no existe ninguna forma de gestionar las **partidas guardadas**.

Conclusiones

Los resultados de la encuesta indican que el juego ha tenido una recepción positiva entre los encuestados, los cuales lo han valorado positivamente en la mayoría de las categorías analizadas.

La encuesta también ha señalado a un problema en el **control de la paleta del personaje principal**, tanto mediante comentarios directos de los jugadores como con una nota menor en el apartado de controles del juego. Un control impreciso puede ser también la causa de el gran número de jugadores que no terminaron el juego y también de la dificultad elevada.

Capítulo 5

Conclusiones y Trabajos Futuros

5.1 Conclusiones

El desarrollo de este proyecto comenzó con la imposición de una serie de **objetivos** que debían ser completados para que el proyecto pudiese ser considerado exitoso. Estos objetivos incluían tanto requisitos formales del software tanto como unas expectativas de como luciría el resultado final. En esta sección, se va a realizar una comparación entre el juego en su versión final y estos objetivos.

5.1.1 Grado de Cumplimiento de los Objetivos

Objetivo Principal

El objetivo de este proyecto es el de **diseñar y desarrollar un videojuego completo con gráficos en tres dimensiones para PC** utilizando el motor de juegos **Unity**. Como todo producto software, el desarrollo de un videojuego requiere de una producción bien planificada que permita terminar el proyecto dentro de los márgenes de tiempo y presupuesto previstos, sin embargo, en la elaboración de un videojuego intervienen factores que no son frecuentes en el desarrollo software. El desarrollo de videojuegos es un proceso multidisciplinar en el que se debe integrar código, arte, sonido y animación en un qué además de funcional debe de resultar **divertido** para sus usuarios.

El resultado del **proyecto es un videojuego completo** en el que se encuentran **implementadas todas las características básicas** incluidas en el diseño. Para su desarrollo fue necesario utilizar el entorno integrado de desarrollo Unity3D, el cual utiliza una metodología de desarrollo no convencional en el desarrollo de software, pero bastante común en el ámbito de los videojuegos. El desarrollo software fue complementado con la elaboración de **modelos 3D y texturas** para los diversos elementos del juego. Al carecer de un artista dedicado en el equipo, la tarea de elaborar estos gráficos recayó también en el programador.

Objetivos Secundarios

De forma paralela al objetivo principal, también se plantearon unos objetivos secundarios en relación con la clase del juego que se pretendía desarrollar.

5. CONCLUSIONES Y TRABAJOS FUTUROS

El primero de estos objetivos era el de **trasladar la jugabilidad del género a un entorno 3D**, solucionando los diversos problemas que suponía el traslado de las mecánicas de un juego tradicionalmente bidimensional a las tres dimensiones. El juego final logra adaptar perfectamente el género Breakout a este nuevo entorno gracias a los ajustes realizados en los elementos principales del juego. Los cambios realizados en las físicas de la pelota y el control de personaje principal sirvieron para corregir los problemas intrínsecos del nuevo medio 3D al mismo tiempo que mantenían la esencia de este tipo de juegos.

Sin embargo, la falta de tiempo provocó que algunos elementos y mecánicas comunes en el género Breakout no pudieron ser implementados en el juego. Los **Power-Ups** o potenciadores, objetos que al ser recolectados mejoran temporalmente al jugador, y un **sistema de puntuación** acompañado de un sistema de ranking son dos constantes en el género que no pudieron ser implementadas.

El segundo objetivo secundario era el de implementar el juego de forma que sea **fácilmente expandible** con más niveles y elementos según fuese necesario. Este objetivo fue cumplido de forma parcial debido a que, aunque la creación de niveles es un proceso sencillo y rápido, y es posible añadir nuevos tipos de ladrillos con relativa facilidad, el sistema es bastante cerrado a ampliaciones en bastantes áreas. Tareas como niveles con distintos tamaños o con elementos en los suelos y paredes requerirían de cambios importantes en los sistemas actuales.

5.1.2 Grado de Cumplimiento de Competencias

Con el desarrollo de este videojuego se quería poner a prueba las **competencias** adquiridas en el grado en ingeniería informática, en especial dos competencias de la rama de la computación:

- La capacidad para desarrollar y evaluar sistemas interactivos y de presentación de información compleja y su aplicación a la resolución de problemas de diseño de **interacción persona computadora**.
- Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los **sistemas inteligentes** y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación.

Los videojuegos son, por definición, sistemas interactivos. Desde las primeras etapas del proyecto, la principal prioridad del proyecto fue que la **interacción entre el jugador y el juego** fuese lo más natural y cómoda posible. Durante el desarrollo, numerosos ajustes y sistemas fueron añadidos sobre la jugabilidad base para mejorar la experiencia de juego. Algunos de estos cambios fueron:

- El tutorial al principio del juego, el cual está diseñado para informar al usuario

sin resultar molesto o intrusivo.

- El control del personaje principal fue ajustado para que respondiera a la perfección a la entrada del jugador.
- La dirección en la que se mueve la pelota es modificada mediante código adicional para que se adapte a las expectativas del jugador, aunque no se correspondan con el movimiento natural.
- La paleta de colores y la iluminación está ajustada para que al jugador le resulte fácil distinguir los distintos elementos del juego entre sí.

El comportamiento del jefe final es la principal muestra de **inteligencia artificial** del proyecto. Este enemigo es utilizado en el nivel final del juego como el último desafío para el jugador, sirviendo como clausura de la experiencia de juego. Lejos de comportarse de forma perfecta para evitar que el jugador gane el juego, el jefe se comporta como si se tratase de un oponente humano, introduciendo defectos intencionales en sus movimientos.

Este comportamiento está implementado mediante una **máquina de estados finitos**. Sin ser una implementación demasiado compleja, la máquina de estados permite encapsular los distintos comportamientos del jefe y asegurar que en todo momento el jefe reacciona de la manera esperada. Gracias al sistema de componentes de Unity, **la inteligencia artificial es independiente de la implementación del movimiento del jefe**, lo que nos permitiría cambiarla con facilidad por otra con un comportamiento distinto en caso de necesidad.

5.1.3 Valoración Personal

Aparte de los objetivos cumplidos y de las competencias demostradas, durante el desarrollo de este proyecto pudieron completarse **varios logros que son dignos de mención**. De la misma forma, el proyecto tuvo varios fallos importantes que deben evitarse en futuros trabajos.

El principal logro del proyecto es el **sistema de carga de niveles**. Aunque de alcance reducido, este sistema cumple con todos los requisitos necesarios para el desarrollo rápido de niveles. Al no estar incluido en el diseño original del juego, la carga de niveles tuvo que ser diseñada a mitad del desarrollo e integrada con el resto de los sistemas. A pesar de las dificultades, el sistema de carga de niveles pudo ser implementado de forma satisfactoria, lo que prueba la capacidad de adaptación del modelo de desarrollo iterativo.

Otro logro importante del proyecto fue el de **implementar completamente el estilo artístico**: todos los elementos del juego cuentan con su modelo y textura propios, junto con pequeñas animaciones simples al personaje principal, a la puerta, al jefe y

5. CONCLUSIONES Y TRABAJOS FUTUROS

a los textos de los menús. Los efectos de partículas añadidos a ciertas animaciones ayudaron enormemente a dar dinamismo a los elementos del juego. Esto pudo lograrse a pesar de **no contar con un artista dedicado para el proyecto**, gracias a la elección de un estilo artístico minimista.

Por otro lado, el principal punto negativo del juego es su **poca extensión**. Aunque el juego cumple con los requisitos mínimos para estar dentro del género **Breakout**, también carece de muchos de los elementos comunes en el género, especialmente en los representantes más modernos. Elementos del juego como potenciadores o enemigos, que habrían servido para dar más variedad a los niveles; sistemas suplementarios para alargar la vida del juego, como una tabla de puntuaciones o un selector de niveles.

Otro punto negativo, esta vez a nivel técnico, fue el sistema escogido para implementar las **máquinas de estados finitos**. El plugin utilizado funciona dividiendo la ejecución de un componente en varios estados, creando **eventos** para cada estado. El problema es que todos los eventos se implementan dentro de la misma clase y, por tanto, del mismo archivo, por lo que hasta el tamaño de las clases crece muy rápidamente con cada estado añadido.

Este problema, junto con la **falta de funcionalidad suplementaria para los estados** (como la posibilidad de ejecutar acciones al finalizar un estado), hacían muy difícil implementar correctamente los estados y provocaron varios fallos que costaron mucho tiempo y esfuerzo en ser resueltos, pero el plugin estaba tan integrado en la base del proyecto que no era posible sustituirlo. Alternativas a esta implementación deberán ser tomadas en cuenta en futuros proyectos.

5.2 Líneas de Trabajo Futuro

Debido a las limitaciones de tiempo y alcance de este proyecto, el juego resultante está lejos de los estándares de calidad de la industria actual. Para alcanzar un **nivel de calidad profesional**, el juego deberá ser ampliado en múltiples áreas para añadir gráficos y sonidos de mejor calidad, aumentar su duración con más niveles y optimizar su código.

Uno de los problemas a la hora de expandir el juego en su versión actual es que el **sistema que controla los eventos del juego esta descentralizado**, dividido entre los distintos objetos del juego. Para poder añadir nuevos elementos más rápidamente, sería conveniente desarrollar un sistema centralizado localizado en un único objeto que se encargue de coordinar el resto de los elementos del juego. De esta forma, futuras adiciones al juego supondrán modificar un solo objeto en lugar de a todos los objetos.

Otro factor limitante en la expansión del juego es la **carga de niveles**. El sistema

actual, si bien cumple con los requisitos esperados, no da soporte a características tan básicas como salas de distintos tamaños o de colocar elementos en puntos que no sean la puerta de la sala. La mejor opción para el desarrollo de la nueva carga de niveles es implementar el sistema dentro del editor de Unity mediante **scripts de editor**, lo que ahorraría la necesidad de integrar una herramienta nueva al proceso de desarrollo.

Finalmente, un punto importante a mejorar del desarrollo es realizar una mejor **campaña de difusión**, que permita llegar a un público mayor. Un componente importante de la futura campaña sería el de elaborar un **blog de desarrollo** en el que se publicaría de forma regular los avances en el proyecto para así atraer a jugadores antes incluso de que el proyecto esté terminado. También será importante publicitar el juego antes de su lanzamiento con un tráiler.

Bibliografía

- [Bat04] Bob Bates. *Game Design*. Boston: Thomson Course Technology, 2004.
- [Bet03] Erik Bethke. *Game Development and Production*. Plano, Texas: Wordware Publishing, 2003.
- [Bjo] Staffan Bjork. *Game Design Patterns*. URL: http://virt10.itu.chalmers.se/index.php/Main_Page.
- [Bur15] Keith Burgun. *Clockwork Game Design*. Routledge, 2015.
- [Dav15] David Villa David Vallejo Carlos González. *Desarrollo de Videojuegos: Un Enfoque Práctico*. Ciudad Real, España, 2015.
- [DEV17] Desarrollo Español del Videojuego DEV, ed. *Libro Blanco del Desarrollo Español del Videojuego*. Madrid, España, 2017.
- [EB95] Ralph Johnson Erich Gamma Richard Helm y John Blissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Gre09] Jason Gregory. *Game Engine Architecture*. Wellesley, Massachusetts: A K Peters, Ltd., 2009.
- [Haa14] John Haas. “A History of the Unity Game Engine”. En: (2014).
- [Lyd] Bill Lydon. *Industry 4.0 - Only One-Tenth of Germany’s High-Tech Strategy*. URL: <https://www.automation.com/automation-news/article/industry-40-only-one-tenth-of-germanys-high-tech-strategy>.
- [Nys04] Robert Nystrom. *Game Programming Patterns*. Geneber Benning, 2004.
- [Rab14] Steve Rabin. *Game AI Pro: Collected Wisdom of Game AI Professionals*. CRC Press, 2014.
- [RN08] Stuart Russell y Peter Norvig. *Inteligencia Artificial: Un Enfoque Moderno*. Pearson, 2008.
- [Sam59] Arthur L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. En: *IBM Journal of Research and Development* (1959), págs. 210-229.

BIBLIOGRAFÍA

- [Sid14] James D. McCalley Siddhartha Kumar Khaitan. “Design Techniques and Applications of Cyber Physical Systems: A Survey”. En: *IEEE Systems Journal* 9.2 (2014), págs. 350-365. DOI: <https://doi.org/10.1109/JYST.2014.2322503>.
- [Tur53] Alan M. Turing. “Digital computers applied to games”. En: *Faster than thought* (1953), pág. 101.
- [War] Jeff Ward. *What is a Game Engine?* URL: https://www.gamecareerguide.com/features/529/what_is_a_game_.php.
- [Yan12] Georgios N. Yannakakis. “Game AI Revisited”. En: *Proceedings of the 9th conference on Computing Frontiers* (2012), págs. 285-292.
- [Yan15] Geoffrey Hinton Yann LeCun Yoshua Bengio. “Deep Learning”. En: *Nature* (2015), págs. 436-444.
- [YT18] Georgios N. Yannakakis y Julian Togelius. *Artificial Intelligence and Games*. <http://gameaibook.org>. Springer, 2018.

Videojuegos

- [Cor97] Taito Corporation. *Arkanoid: Doh It Again.* 15 de ene. de 1997.
- [Dou52] Alexander S. Douglas. *OXO.* 1952.
- [Ent00] Blizzard Entertainment. *Diablo II.* 29 de jul. de 2000.
- [Ent04] Blizzard Entertainment. *World of Warcraft.* 23 de nov. de 2004. URL: <https://worldofwarcraft.com>.
- [Fox15] Toby Fox. *Undertale.* 15 de sep. de 2015.
- [Gam07] 2K Games. *BioShock.* 19 de ago. de 2007.
- [Gam09] Riot Games. *League of Legends.* 27 de oct. de 2009. URL: <https://leagueoflegends.com>.
- [Inc76] Atari Inc. *Breakout.* 13 de mayo de 1976.
- [Max00] Maxis. *The Sims.* 4 de feb. de 2000.
- [Meg98] Epic MegaGames. *Unreal.* 22 de mayo de 1998.
- [Moj11] Mojang. *Minecraft.* 18 de nov. de 2011.
- [Nam80] Namco. *Pac-Man.* 21 de mayo de 1980.
- [Nia16] Niantic. *Pokémon Go.* 6 de jul. de 2016.
- [Sof93] id Software. *Doom.* 10 de dic. de 1993.
- [Sof94] Raven Software. *Heretic.* 23 de dic. de 1994.
- [Sof99] id Software. *Quake III Arena.* 2 de dic. de 1999.
- [Stu04] Bungie Studios. *Halo 2.* 9 de nov. de 2004.
- [Tai78] Taito. *Space Invaders.* 1 de jun. de 1978.
- [Val07] Valve. *Team Fortress 2.* 10 de oct. de 2007. URL: <https://teamfortress.com>.
- [Wha15] Hipster Whale. *Pac-Man 256.* 20 de ago. de 2015.

Este documento fue editado y tipografiado con L^AT_EX empleando
la clase **esi-tfg** (versión 0.20180617) que se puede encontrar en:
https://bitbucket.org/arco_group/esi-tfg

[respeta esta atribución al autor]

