



**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.:

Resumen

En este proyecto se ha realizado el desarrollo de un videojuego completo para PC utilizando el motor de juegos Unity3D. El juego es una reinvención del clásico *Breakout*, adaptándolo a las tres dimensiones mediante cambios en su diseño.

Esta memoria describe la situación socioeconómica del mercado del videojuego tanto mundial como nacional a modo de justificación del desarrollo del proyecto, citando entre otros factores el enorme crecimiento que está teniendo esta industria. A continuación, se realizará un análisis del proceso de desarrollo de un videojuego, y una descripción de la estructura de equipo de desarrollo de videojuegos. También realizaremos un estudio del uso de la inteligencia artificial en el campo del videojuego.

La memoria también describe en detalle la implementación de este proyecto, describiendo la estructura de su código y explicando los patrones de diseño en los que se basa el motor. Acompañando al análisis general, se realizará una revisión de los puntos más destacados del proyecto como las mecánicas de control del personaje principal, el sistema de carga de niveles y la implementación del jefe final del juego.

Finalmente, el documento cerrará con una revisión del resultado final del proyecto, junto con una lista de posibles mejoras para futuras versiones.

Índice general

Resumen	v
Índice general	vii
Índice de figuras	ix
1. Introducción	1
1.1. Motivación	1
1.2. Diseño de juego	2
1.2.1. Concepto	2
1.2.2. Jugabilidad	2
1.2.3. Estilo y Ambientación	4
1.2.4. Plataforma	5
2. Estado del arte	7
2.1. El mercado de los videojuegos	7
2.1.1. Industria mundial del Videojuego	7
2.1.2. La industria de los videojuegos en España	9
2.1.3. Retos y tendencias actuales	10
2.2. El proceso de desarrollo de videojuegos	17
2.2.1. Etapas del desarrollo	18
2.2.2. Estructura típica de un equipo de desarrollo	22
2.3. Motores de juegos	28
2.3.1. Descripción	28
2.3.2. Ejemplos de Motores	31
2.3.3. Unity 3D	34
2.4. Inteligencia Artificial en Videojuegos	37
2.4.1. Historia	37
2.4.2. IA Aplicada a Videojuegos: Contexto Actual	39
2.4.3. Métodos de la IA	41

0. ÍNDICE GENERAL

2.4.4. Futuros campos de aplicación	46
3. Arquitectura	49
3.1. Descripción del Juego	49
3.2. Escenas	51
3.2.1. Menus	52
3.2.2. Escena de Juego	53
3.3. Objetos	57
3.3.1. Personaje Principal	58
3.3.2. Bola	61
3.3.3. Ladrillos	65
3.3.4. Sala y Puerta	67
3.3.5. Jefe	70
4. Conclusiones	75
4.1. Resultados	75
4.2. Mejoras Futuras	76
4.2.1. Escenarios	77
4.2.2. Enemigos	78
4.2.3. Sistemas Complementarios	79
Bibliografía	81
Videojuegos	84

Índice de figuras

1.1.	Breakout (Atari, 1976)	2
1.2.	Relación entre desafío y la ansiedad/aburrimiento.	4
1.3.	El estilo del juego está influenciado por títulos como Megaman Legends(Capcom, 1997)	5
1.4.	Gamejolt	6
1.5.	Itch.io	6
1.6.	Juegos desarrollados con Unity3D	6
2.1.	Crecimiento Mundial de la industria del videojuego (por plataformas)	8
2.2.	Distribución por países del mercado del videojuego.	8
2.3.	Mercado del Videojuego en Europa.	9
2.4.	Distribución de las empresas en España por número de empleados.	10
2.5.	Distribución de las empresas en España por número de empleados.	10
2.6.	Fotografía del torneo Intel® Extreme Masters en Katowice, Polonia	11
2.7.	Crecimiento del mercado del eSport	12
2.8.	<i>League of Legends</i> (Riot Games, 2009), uno de los eSports más populares	13
2.9.	Previsiones de crecimiento del mercado de Realidad Virtual y Aumentada (tabla extraída de [DEV17]) (miles de millones de dólares))	14
2.10.	De izquierda a derecha: HTC Vive, Oculus Rift, Samsung VR y PlayStation VR	15
2.11.	Áreas de la Industria 4.0 (imagen tomada de [DEV17])	16
2.12.	Impresora 3d “replicator” de la compañía Makerbot.	17
2.13.	Etapas del desarrollo de un videojuego (Imagen tomada del [Dav15]).	18
2.14.	Distribución de roles en un equipo de desarrollo.	22
2.15.	Shigeru Miyamoto (creador de <i>Super Mario Bros.</i> (Nintendo, 1985), <i>The Legend of Zelda</i> (Nintendo, 1986) y <i>Donkey Kong</i> (Nintendo, 1981)) es posiblemente el diseñador de videjouuegos más famoso.	23
2.16.	Jonh Romero es el co-fundador de ID Software y programador de juegos como <i>Doom</i> (id Software, 1993) o <i>Quake</i> (id Software, 1996).	25

0. ÍNDICE DE FIGURAS

2.17. Akira Toriyama, el autor de Dragon Ball, ha trabajado como artista en juegos como <i>Dragon Quest</i> (Chunsoft, 1986) o <i>Chrono Trigger</i> (Square, 1995).	26
2.18. Yoko Shinomura es una compositora conocida por su trabajo en videojuegos como <i>Final Fantasy XV</i> (Square Enix, 2016) o <i>Street Fighter II: The World Warrior</i> (Capcom, 1991).	27
2.19. <i>Heretic</i> (Raven Software, 1994), es un juego desarrollado con el motor de Doom.	29
2.20. <i>Unreal</i> (Epic Games, 1998)	32
2.21. Captura del entorno de <i>Unreal Engine</i>	33
2.22. Captura del entorno de <i>Game Maker Studio</i>	33
2.23. <i>Undertale</i> (Toby Fox, 2015)	34
2.24. Captura del entorno de desarrollo de <i>Unity</i>	36
2.25. <i>Pac-Man 256</i> (Hipster Whale, 2015), disponible para PC, Android, IOS, PS4 y XBOX One	38
2.26. El Gran Maestro de ajedrez Garri Kaspárov (izquierda) enfrentándose al ordenador Deep Blue	39
2.27. En <i>The Sims</i> (Maxis, 2000), los personajes pueden tomar decisiones basándose en sus gustos y necesidades.	40
2.28. <i>Minecraft</i> (Mojang, 2011) es un ejemplo claro de generación procedimental de terrenos.	41
2.29. FSM de alto nivel de una IA jugadora de <i>Pac-Man</i> (Namco, 1980) (figura extraída de [YT18])	42
2.30. Ejemplo de comportamiento tóxico en <i>League of Legends</i> (Riot Games, 2009).	47
 3.1. Pantalla de título.	49
3.2. Pantalla de juego.	50
3.3. Pantallas de victoria y derrota.	50
3.4. Editor de escenas de <i>Unity</i>	52
3.5. Diagrama de las escenas del juego	52
3.6. Diagrama de las escenas del juego	54
3.7. Homenaje a Space Invaders(Taito, 1978) dentro de Arkanoid: Doh it Again (Taito, 1997)	55
3.8. Diagrama del proceso de generación de niveles	57
3.9. Modelo del personaje principal	59
3.10. Diagrama de componentes del personaje principal.	60
3.11. Diagrama de estados del personaje principal	61
3.12. Personaje principal activando la paleta	62

3.13. Pelota moviéndose	62
3.14. Componentes del objeto Ball.	63
3.15. Direcciones que tomaría la pelota dependiendo del punto de la paleta que golpee (Aproximada)	64
3.16. Efectos de partículas: impacto en muro (izquierda) y explosión (derecha)	64
3.17. Modelos de los ladrillos	65
3.18. Componentes del objeto Brick.	66
3.19. Jerarquía de clases.	66
3.20. Sala vista desde el editor.	67
3.21. Diagrama de componentes de la sala.	68
3.22. Modelo del Jefe.	71
3.23. Diagrama de componentes del jefe.	71
3.24. Diagrama de estados del jefe.	73
4.1. Relación entre desafío y la ansiedad/aburrimiento.	75
4.2. Relación entre desafío y la ansiedad/aburrimiento.	76
4.3. Relación entre desafío y la ansiedad/aburrimiento.	77

Capítulo 1

Introducción

1.1 Motivación

Independientemente de la rama del desarrollo de los videojuegos en la que se pretenda trabajar, el factor decisivo que usan las empresas para determinar a qué profesional contratar son los juegos que ha desarrollado[BS12].

A través de la lista de juegos en las que una persona ha trabajado, los reclutadores y directores de recursos humanos pueden determinar la valía de un futuro empleado. Especialmente, sirve para valorar la capacidad de la persona para terminar un proyecto de gran envergadura, realizar una gestión de recursos correcta y su creatividad e iniciativa. De forma más específica, en el caso concreto de los programadores, una lista de proyectos completados (que pueden ser tanto juegos como herramientas de desarrollo) permite valorar su forma de desarrollar, mostrando si es capaz de escribir código comprensible y libre de errores.

La visibilidad que te da tener juegos completados no solo afecta a las empresas que buscan empleados. Tener una lista de juegos también sirve como carta de presentación para los jugadores, los cuales la podrán utilizar a la hora de decidir si deben o no adquirir un juego de dicho desarrollador, o invertir en una campaña de *crowdfunding*.

Desarrollar videojuegos también es beneficioso para cualquier aspirante a desarrollador de videojuegos. Durante el desarrollo de cualquier videojuego se plantearán problemas (“¿Cómo se modela el comportamiento de un enemigo?”, “¿Qué editor de niveles es más conveniente?”, “¿Cuánto tiempo me llevará implementar esta característica concreta?”) los cuales muchas veces volverán a aparecer en muchos, sino todos, los futuros proyectos, por lo que con cada juego, el desarrollador ganará experiencia que le facilitará el trabajo en sus siguientes proyectos y le permitirá emprender proyectos cada vez más ambiciosos.

Con estas ventajas en mente se eligió el desarrollo de un videojuego completo como proyecto de Fin de Grado.

1. INTRODUCCIÓN

1.2 Diseño de juego

1.2.1 Concepto

Este juego será de estilo-Breakout, género que se caracteriza por controlar una paleta con la que se redirige una pelota con el objetivo de destruir un muro de ladrillos o bloques. Sin embargo, este juego transcurrirá en un entorno totalmente tridimensional (a diferencia que otros juegos del género, los cuales tiene una área de juego bidimensional).

En esta adaptación a las tres dimensiones, determinados elementos del juego han sido modificados respecto al estándar del género, inspirándose en otros juegos o deportes como el Squash¹



Figura 1.1: Breakout (Atari, 1976)

1.2.2 Jugabilidad

En este juego, 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 control del personaje principal es sencillo, ya que solo cuenta con dos acciones: moverse por el suelo de la sala y activar la paleta. El movimiento del personaje es un movimiento en ocho direcciones (delante, atrás, derecha, izquierda y diagonales), con

¹<http://www.realfederaciondesquash.com/>

una aceleración alta para dar una buena sensación de control. La activación de la paleta permitirá controlar la dirección de la bola, haciendo que esta rebote, impidiendo que caiga en el suelo y, idealmente, redirigiéndola hacia la puerta. La paleta estará activa durante un tiempo limitado, durante el cual el personaje permanecerá inmóvil. Si el personaje es golpeado por la pelota directamente, cuando la paleta está desactivada, el personaje quedará aturdido unos instantes, durante los cuales no podrá realizar ninguna acción.

La pelota se moverá por la sala rebotando de forma natural por sus paredes. Al golpear la puerta, o los bloques que la cubren, la pelota les causará daño. Los bloques se destruirán tras recibir daño suficiente, mientras que la puerta se abrirá, permitiendo al jugador salir de la sala. Si la pelota rebota tres veces en el suelo de la sala sin que el jugador la golpee con la paleta, esta será destruida y la partida se acabará.

La puerta de la sala está cubierta por un muro de ladrillos o bloques. Los bloques se colocan sobre la puerta alineados en una cuadricula de 16 X 16 unidades. Existen varios tipos de bloques, con distintas propiedades, los cuales pueden estar colocados con una rotación de 0° o de 90° y pintados de distintos colores. Los tipos de bloques son los siguientes:

- Bloque Básico: Es el bloque normal, el cual se rompe al recibir un golpe de la pelota. Las dimensiones de este bloque son una unidad de altura y dos de anchura.
- Bloque Triple: Idéntico en dimensiones al bloque básico, pero se romperá tras recibir tres golpes en lugar de uno.
- Bloque Divisible: Se trata de un bloque cuadrado de dos unidades de altura y anchura. Al ser golpeado, este bloque se romperá en cuatro Bloques Pequeños.
- Bloque Pequeño: Es un bloque de pequeño tamaño (una unidad de altura y anchura).
- Bloque Irrompible: Este bloque no puede ser destruido por la pelota, por muchos golpes que reciba. Es también un bloque de gran tamaño, ya que mide el doble que el bloque básico (cuatro unidades de ancho y dos de altura).

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. Este aumento gradual mantendrá al jugador en un estado mental llamado “fluir” en el cual se tiene una concentración extrema en el juego [flow]. De forma similar, los distintos tipos de bloques serán introducidos de forma progresiva

1. INTRODUCCIÓN

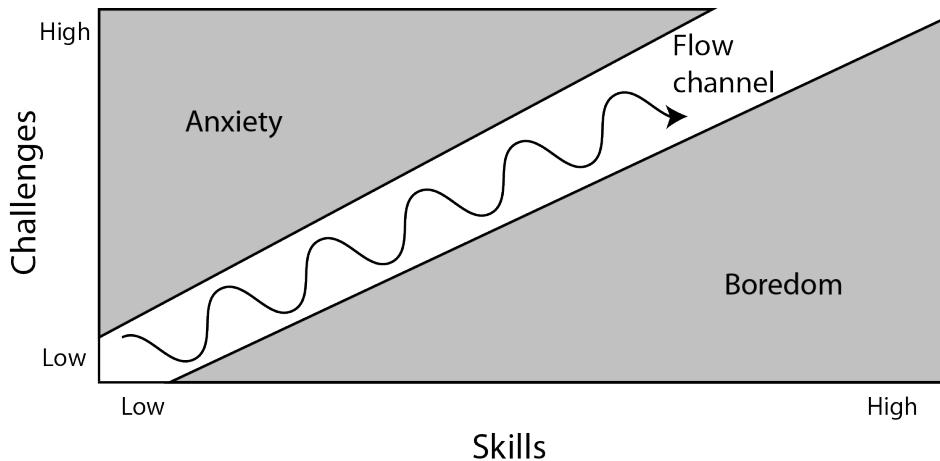


Figura 1.2: Relación entre desafío y la ansiedad/aburrimiento.

1.2.3 Estilo y Ambientación

El equipo de desarrollo de este juego está formado por una única persona, la cual debe realizar la producción completa del juego, que incluye diseño, programación, producción de los assets gráficos y creación de la música y sonidos. Dado que la responsabilidad de la producción artística cae en manos de un desarrollador que debe, además, ocuparse de otras responsabilidades dentro del proyecto en lugar de en un profesional que se dedique a tiempo completo en la producción, los posibles estilos gráficos y sonoros son limitados.

La elección del estilo artístico se limitaba a dos opciones: utilizar assets con licencia de libre uso, obtenidos de páginas como OpenGameArt.com², o producir el arte utilizando un estilo sencillo y minimalista que pueda producirse en poco tiempo. La opción que se eligió fue la producción propia utilizando un estilo minimalista, ya que daría al juego un aspecto más único y cohesivo. Dado que el diseño del juego requiere de gráficos tridimensionales, se optó por un diseño de personajes y escenarios con un bajo número de polígonos, al mismo tiempo que se empleaban texturas con una resolución y paleta de colores muy limitada, creadas usando el estilo artístico “Pixel Art”, el cual imita las limitaciones gráficas de los ordenadores y videoconsolas antiguas. Los modelos resultantes de aplicar este estilo recuerdan a modelos de papel o maquetas construidas con bloques.

Las opciones para la producción musical son mucho más reducidas. Debido a la dificultad de crear una banda sonora desde cero, la única opción es utilizar música con licencia de libre uso. Para que la música esté en consonancia con el estilo artístico, se utilizará música de estilo “Chiptune” el cual, al igual que los gráficos, imita las limitaciones técnicas de videoconsolas y ordenadores antiguos. Esto mismo se aplicará a los efectos de sonido, también con licencia de libre uso e inspirados en los sonidos de

²<https://opengameart.org/>



Figura 1.3: El estilo del juego está influenciado por títulos como Megaman Legends(Capcom, 1997)

videojuegos antiguos.

El estilo artístico resultante es muy “Digital”, con polígonos visibles, píxeles de gran tamaño, colores limitados y música y sonido sintéticos. La consolidación de este estilo sirvió para definir la ambientación narrativa del juego: se decidió que el juego estará ambientado en el interior de un ordenador, con los personajes y entornos representando programas. Aunque el juego no hace uso de esta narrativa, al carecer de diálogos y cinematográficas, esta temática se empleó en el diseño de los gráficos.

1.2.4 Plataforma

Este juego ha sido diseñado para ser jugado en un PC. El control del personaje principal se realiza mediante las flechas de dirección (movimiento) y la tecla Espacio (paleta y navegar por menús).

La exportación de este juego a dispositivos móviles como Android o IOS no sería posible debido a que el sistema de control no es compatible con pantallas táctiles, por lo que sería necesario realizar una adaptación, ya sea implementado un sistema de control del personaje basado en la pantalla táctil o mediante un controlador virtual en pantalla. Ambas aproximaciones aumentarían el coste de desarrollo del juego. Además, el juego no está optimizado para este tipo de dispositivos, de menor potencia que los PC, por lo que podría no funcionar correctamente.

La distribución del juego se realizará online, a través de las páginas de distribución de juegos Game Jolt³ y itch.io⁴. Estos sitios ofrecen a sus usuarios la posibilidad de subir, sin ningún tipo de coste, el juego a la página para que otros usuarios puedan

³<https://gamejolt.com/>

⁴<https://itch.io/>

1. INTRODUCCIÓN

descargarlos (o, en el caso de Game Jolt, jugarlos directamente en el navegador si el juego es compatible). Ambas páginas ofrecen la posibilidad tanto de vender el juego como de distribuirlo de forma gratuita. Dada la naturaleza del proyecto, se ha optado por la distribución gratuita del mismo.

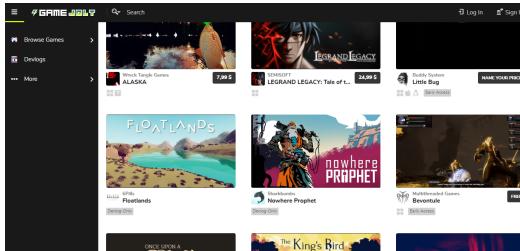


Figura 1.4: Gamejolt

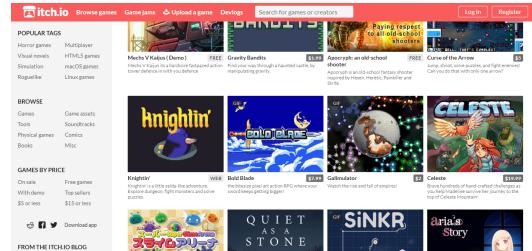


Figura 1.5: Itch.io

Figura 1.6: Juegos desarrollados con Unity3D

Capítulo 2

Estado del arte

En este apartado se va a introducir el contexto actual del desarrollo de videojuegos. La presentación 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

El mercado del videojuego es actualmente **la industria del entretenimiento de mayor tamaño**, superando en tamaño tanto a la industria cinematográfica como a la discográfica. Se trata de una industria creciente, con una tasa del crecimiento del 6,6 % y que cuenta ya con más de 2.200 millones de jugadores en todo el mundo. Como podemos 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

2. ESTADO DEL ARTE

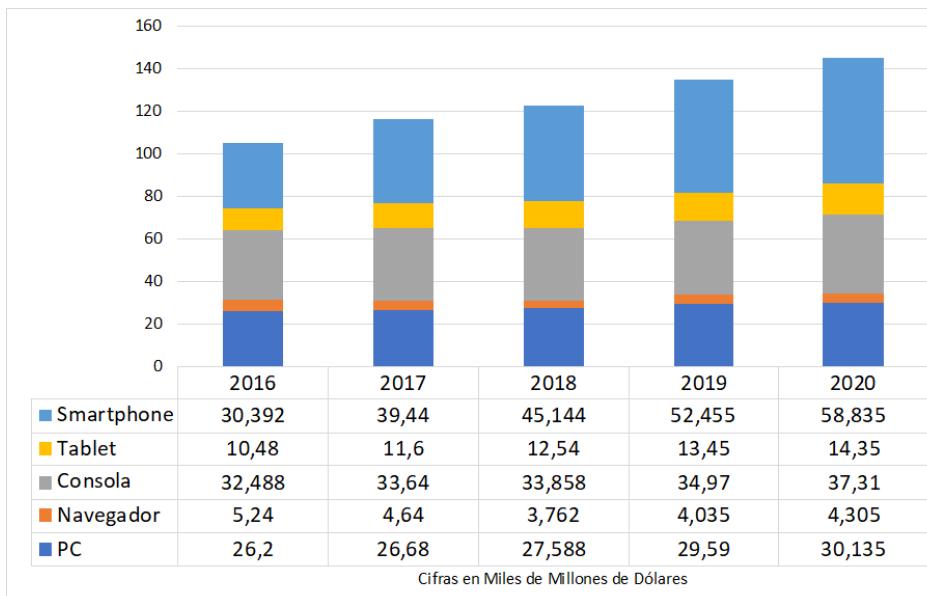


Figura 2.1: Crecimiento Mundial de la industria del videojuego (por plataformas).

57.800 millones de dólares generados en el año 2017, de los cuales 32.5 millones fueron 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 encontramos 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].

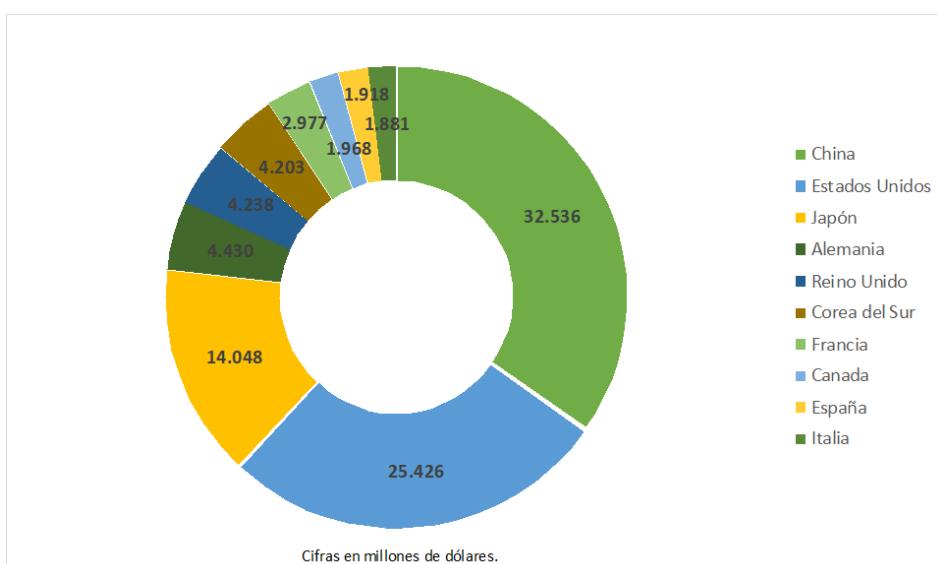


Figura 2.2: Distribución por países del mercado del videojuego.

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].

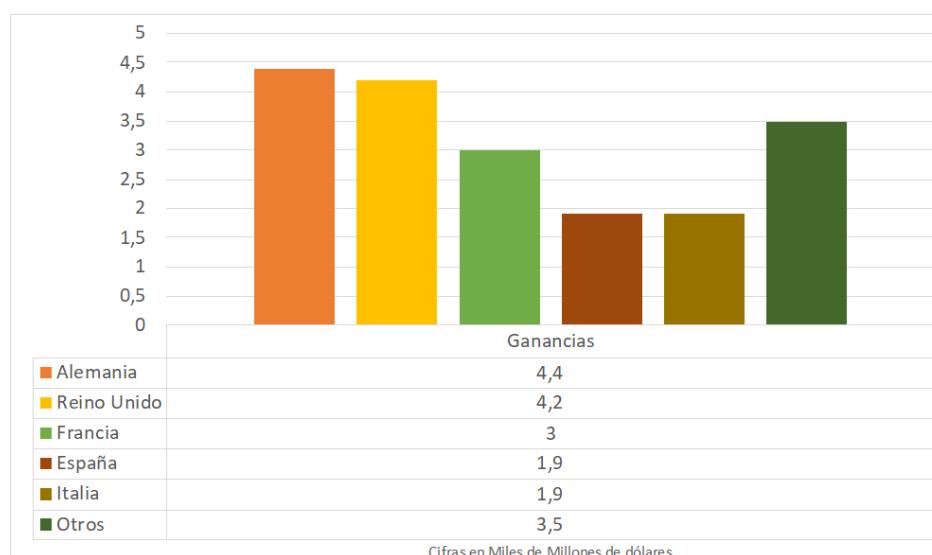


Figura 2.3: Mercado del Videojuego en Europa.

En total, el sector cuenta con **480 empresas en activo**, a las que debemos añadir las 130 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 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 comunidades se quedan muy por detrás de estas cinco primeras como puede verse en

2. ESTADO DEL ARTE

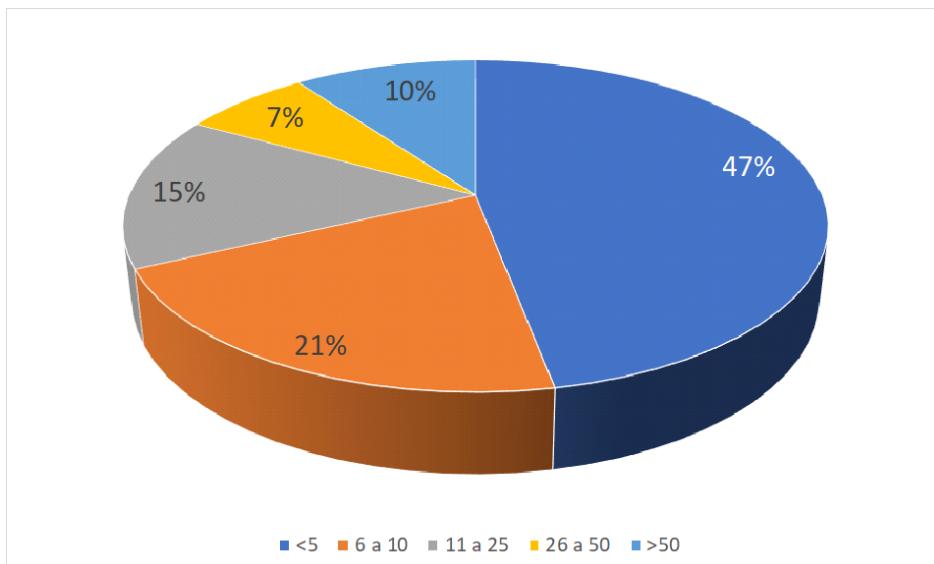


Figura 2.4: Distribución de las empresas en España por número de empleados.

la figura 2.5 [DEV17].

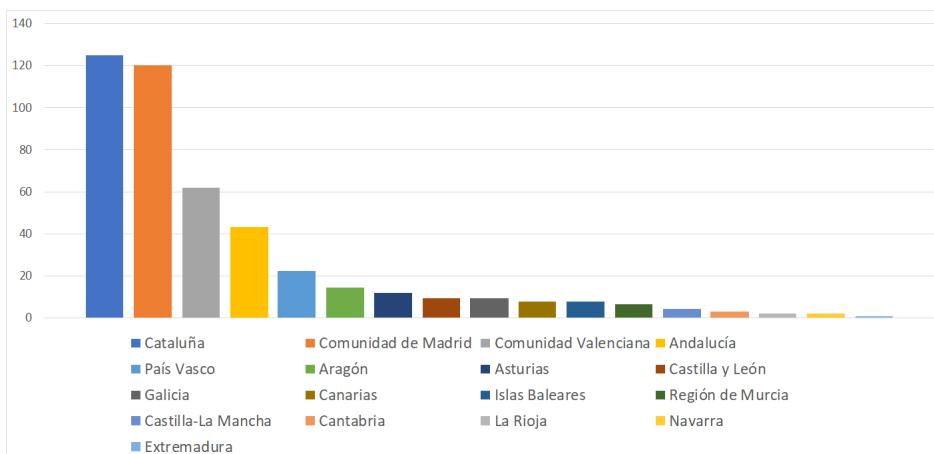


Figura 2.5: Distribución de las empresas en España por número de empleados.

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.

A continuación, listaremos algunas de las tendencias que van a influir fuertemente en el mercado en los años venideros:

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

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].



Figura 2.6: Fotografía del torneo Intel® Extreme Masters en Katowice, Polonia

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 la el enorme coste de su producción. En la mayoría de los casos, los eSports **surgen de manera orgánica** al rededor 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 siempre esto asegura que su producto se convierta en un éxito. 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 jugador) atractivo sistema de espectador y tener un enfoque al 100 % internacional.

2. ESTADO DEL ARTE

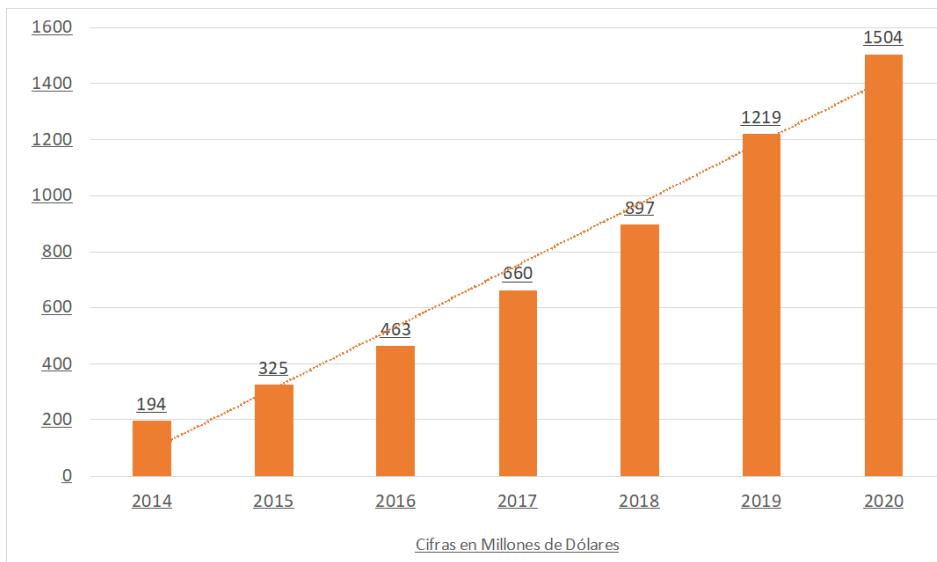


Figura 2.7: Crecimiento del mercado del eSport

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, inabarcable. Esto se debe principalmente la elevada inversión mencionada anteriormente. Sin 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.



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

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 interactuar 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 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”

¹<https://www.vive.com/>

2. ESTADO DEL ARTE

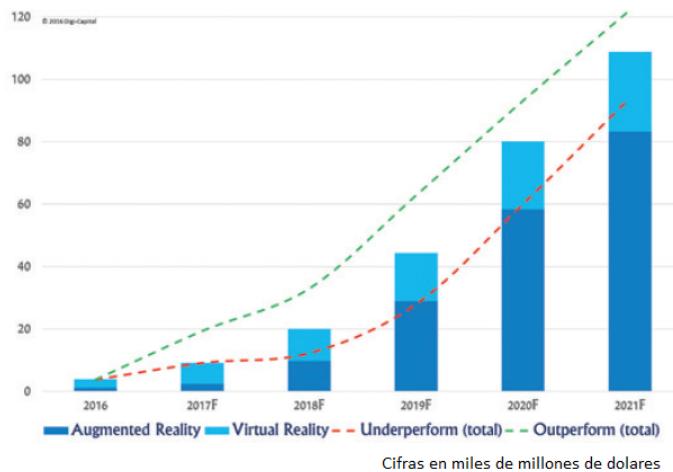


Figura 2.9: Previsiones de crecimiento del mercado de Realidad Virtual y Aumentada (tabla extraída de [DEV17]) (miles de millones de dólares))

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 económica, 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

²<https://www.oculus.com/rift/>

³<http://www.samsung.com/es/wearables/gear-vr-sm-r325nzvaphe/>

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

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

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 (listadas en la figura 2.11), vamos a describir aquellas que tienen mayores aplicaciones en el desarrollo de videojuegos [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 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

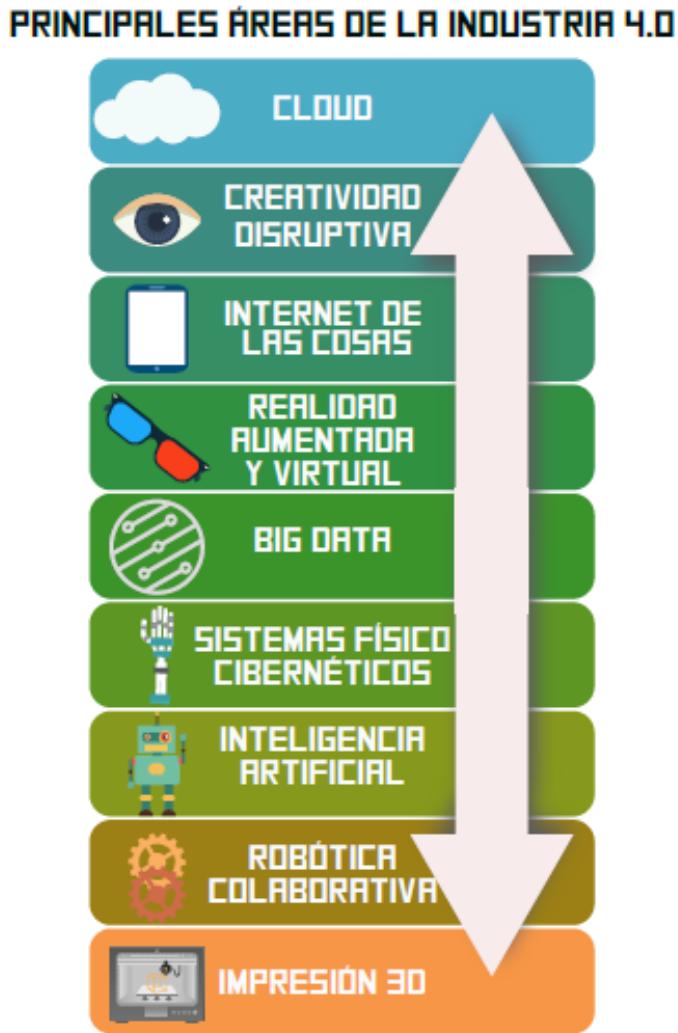


Figura 2.11: Áreas de la Industria 4.0 (imagen tomada de [DEV17])

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 2.12. 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...).



Figura 2.12: Impresora 3d “replicator” de la compañía Makerbot.

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 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

2. ESTADO DEL ARTE

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 Post-Producción**.

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 2.13 y en los siguientes apartados se describe el desarrollo basándose en este modelo.

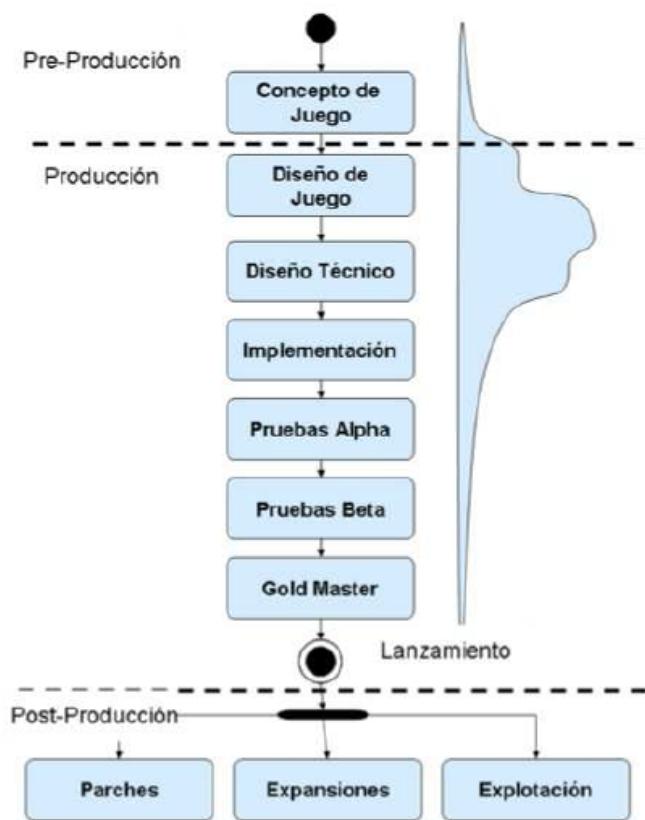


Figura 2.13: Etapas del desarrollo de un videojuego (Imagen tomada del [Dav15]).

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 pre-

producción. 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 pre-producción 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.

2. ESTADO DEL ARTE

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.
- **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 elaboraran 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 mini juegos o sub-sistemas 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.

Post-Producción

Una vez lanzado el juego, el equipo entra en la etapa de **Post-Producción**. 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 post-producción 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.

2. ESTADO DEL ARTE

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 2.14 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 persona ejerza varios roles, o incluso que una única persona lleve el desarrollo completo.

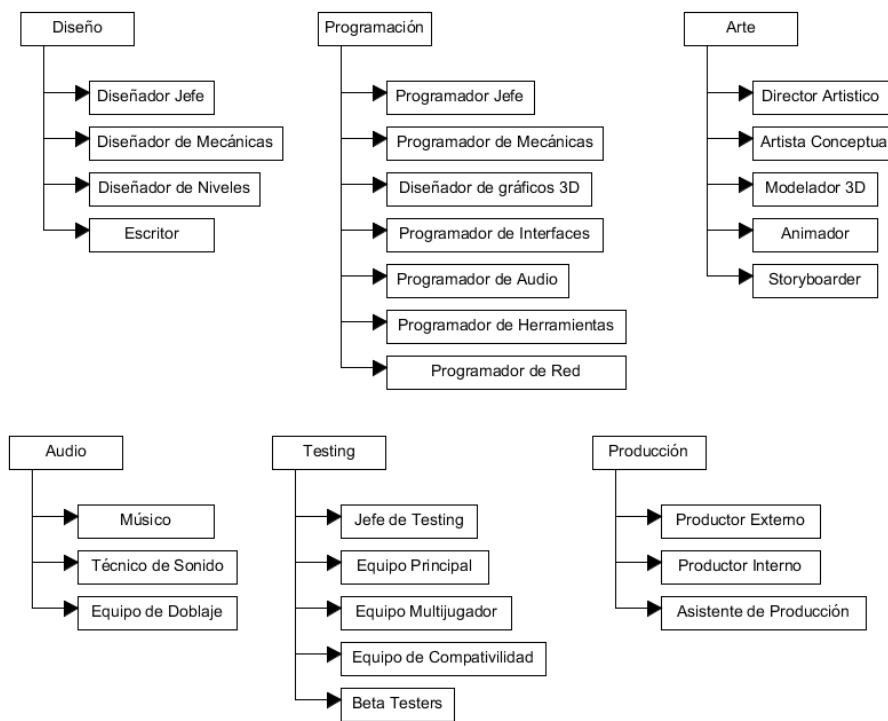


Figura 2.14: Distribución de roles en un equipo de desarrollo.

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.

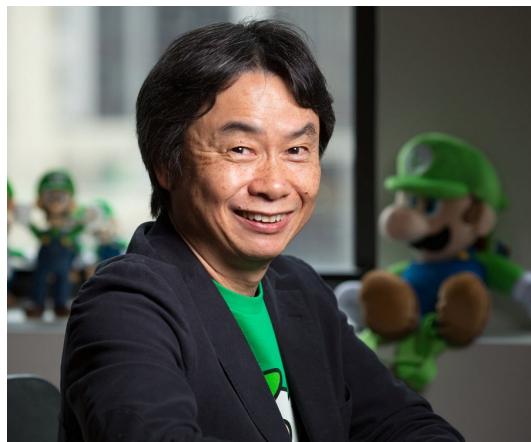


Figura 2.15: Shigeru Miyamoto (creador de *Super Mario Bros.* (Nintendo, 1985), *The Legend of Zelda* (Nintendo, 1986) y *Donkey Kong* (Nintendo, 1981)) es posiblemente el diseñador de videjuegos más famoso.

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

2. ESTADO DEL ARTE

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 necesitar 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 un biblioteca 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]:



Figura 2.16: Jonh Romero es el co-fundador de ID Software y programador de juegos como *Doom* (id Software, 1993) o *Quake* (id Software, 1996).

- **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.

2. ESTADO DEL ARTE

- **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.



Figura 2.17: Akira Toriyama, el autor de Dragon Ball, ha trabajado como artista en juegos como *Dragon Quest* (Chunsoft, 1986) o *Chrono Trigger* (Square, 1995).

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.



Figura 2.18: Yoko Shinomura es una compositora conocida por su trabajo en videojuegos como *Final Fantasy XV* (Square Enix, 2016) o *Street Fighter II: The World Warrior* (Capcom, 1991).

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 juego en diversas plataformas y PCs 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 2.19) pudieron ser desarrollados cambiando solamente el arte, niveles o armas de títulos anteriores, manteniendo intacto el motor [Gre09].

Este concepto inició las comunidades de “Modding”, que son grupos de aficionados que desarrollaban juegos nuevos modificando juegos antiguos, y para finales de los



Figura 2.19: *Heretic* (Raven Software, 1994), es un juego desarrollado con el motor de *Doom*.

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 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 leer y reproducir 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.

⁵<https://www.havok.com/physics/>

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

- **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 encargados 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* (Epic MegaGames, 1998) (figura 2.20), 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.

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 2.21) generadores de terreno, editores de materiales, herramientas para animación... Sin embargo, se trata de un motor **complicado y difícil de aprender a utilizar**,

⁷<https://www.unrealengine.com/>

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



Figura 2.20: Unreal (Epic Games, 1998)

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. 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 2.22) 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**

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

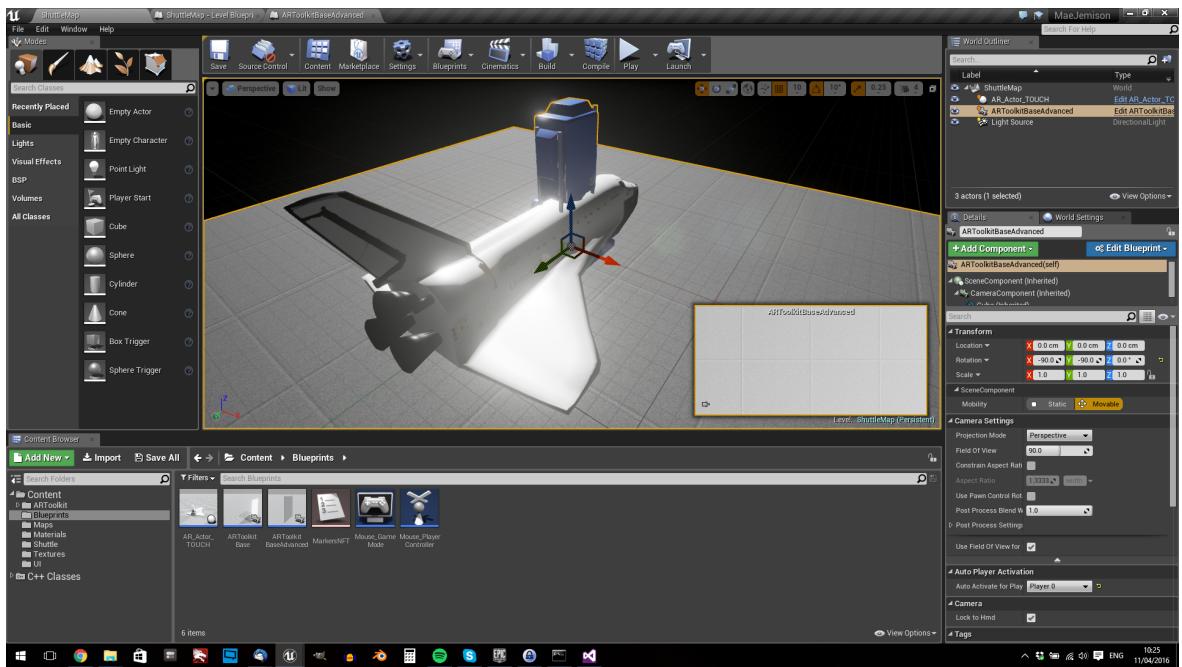


Figura 2.21: Captura del entorno de Unreal Engine

Language.

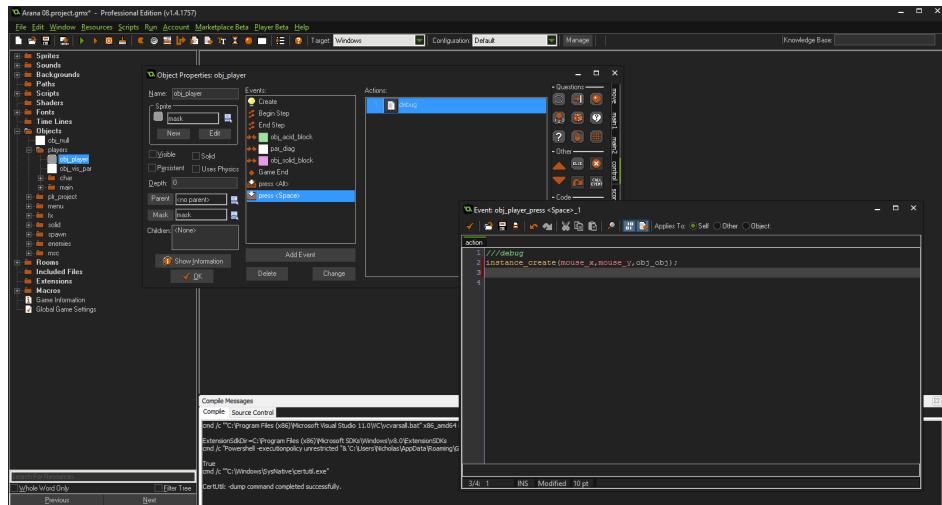


Figura 2.22: Captura del entorno de Game Maker Studio

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 arquitectura dificulta el desarrollo de proyectos de gran envergadura. Estas limitaciones no han impedido la creación de juegos exitosos o revolucionarios con este motor, como

2. ESTADO DEL ARTE

podría ser *Undertale* (Toby Fox, 2015), mejor juego de PC del año 2015¹⁰.

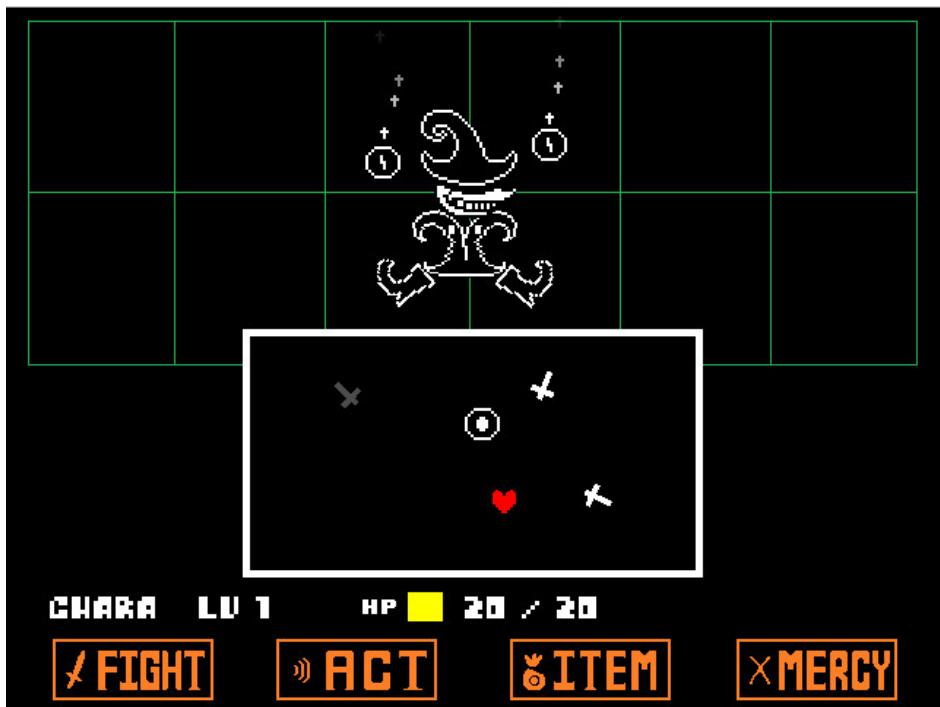


Figura 2.23: *Undertale* (Toby Fox, 2015)

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 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 el 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,

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

¹¹<https://unity3d.com/unity>

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

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.

El motor está incluido en un Entorno de Integrado de desarrollo. La principal característica de este entorno es el llamado **Editor de Unity** (figura 2.24), 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.

La carga y gestión de recursos 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.

2. ESTADO DEL ARTE

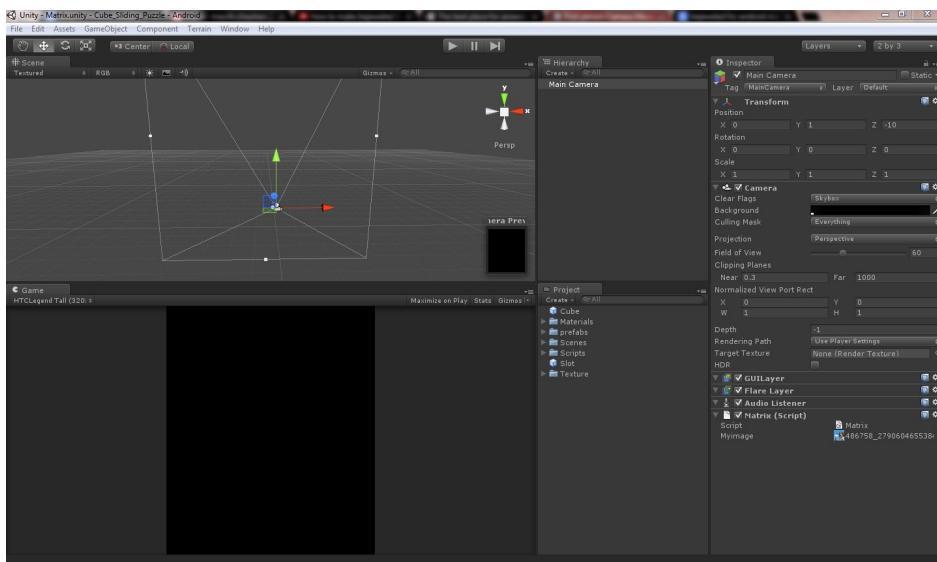


Figura 2.24: Captura del entorno de desarrollo de Unity

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#.

Los scripts pueden ser entonces asignados a **GameObjects**, la única clase de objetos que pueden ser instanciada dentro del editor de Unity. Los Scripts que tiene asignados son lo que determina el comportamiento del GameObject, que por si solo carece de funcionalidad. Se trata de una implementación del patrón de diseño **Component**, el cual permite desacoplar el código de los programas manteniendo la posibilidad de crear entidades que interactúen con diversos ámbitos del separado del programa.

A parte de en el código del juego, Unity permite **escribir código para su propio editor** utilizando C#. La programación del editor funciona de forma muy similar a la del propio Unity, mediante scripts de editor que modelan el comportamiento de las distintas ventanas, botones e inspectores del editor. Los scripts de editor, al implementarse con las mismas librerías que el juego, tienen acceso a todas las clases y tipos incluidos de Unity, lo que facilita la creación de herramientas para el desarrollo.

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 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 2.25) están disponibles en múltiples plataformas gracias a esta característica.

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].



Figura 2.25: *Pac-Man 256* (Hipster Whale, 2015), disponible para PC, Android, IOS, PS4 y XBOX One

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**¹³ y 3 años más tarde el super-ordenador **Deep Blue** venció a gran maestro del ajedrez **Gary Kasparov**¹⁴ (figura 2.26). 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 2011¹⁵ o el programa **AlphaGo** derrotando a **Ke Jie**, el jugador número uno de Go¹⁶ (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

¹³<https://webdocs.cs.ualberta.ca/~chinook/project/>

¹⁴<http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/>

¹⁵https://www.youtube.com/watch?v=WFR3lOm_xhE

¹⁶<https://www.theverge.com/2017/5/25/15689462/alphago-ke-jie-game-2-result-google-deepmind-china>



Figura 2.26: El Gran Maestro de ajedrez Gari Kaspárov (izquierda) enfrentándose al ordenador Deep Blue

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 2600¹⁷.

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 que hemos visto en el apartado anterior. 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 Procedimental 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

¹⁷<https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/>

2. ESTADO DEL ARTE

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].

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 2.27)



Figura 2.27: 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 procedural la utilizan para la **creación automática de mapas** (como en el caso de *Minecraft* (Mojang, 2011) (ver figura 2.28) 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 2.28: *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].

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.

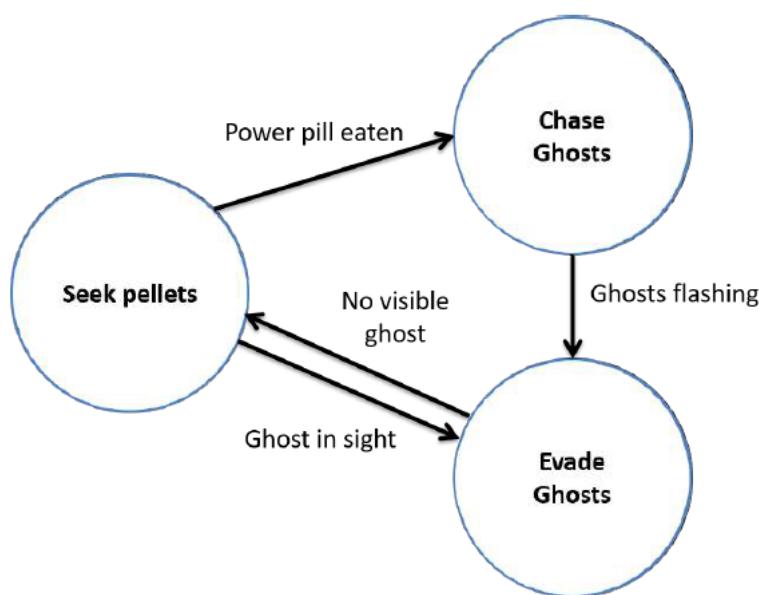


Figura 2.29: FSM de alto nivel de una IA jugadora de *Pac-Man* (Namco, 1980) (figura extraída de [YT18])

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 ade-

más 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 mas 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)¹⁸.

El método Ad Hoc más reciente se llama **Inteligencia artificial basada en Utilidad**. Este método consiste en asignar a los agentes inteligentes **funciones de utilidad** la cual la importancia de distintas características del agente o su entorno, como podrían ser la proximidad de un enemigo o la cantidad de munición restante. Usando estas funciones, el agente elige de entre todas sus posibles acciones cual debe realizar en un momento dado [YT18].

Este método se basa en el modelo matemático de la **lógica difusa**. Se trata de una generalización de la lógica clásica pensada especialmente para el modelado de **situaciones imprecisas o ambiguas**, ya que en lugar de trabajar con valores binarios se emplea el rango de números decimales. El uso de esta lógica está bastante extendido en el ámbito de los electrodomésticos y del control industrial, al permitir un control más “suave” del funcionamiento de máquinas [Dav15].

Comparada con los otros métodos Ad Hoc, la inteligencia artificial basada en utilidad es mucho más **modular y extensible**. La lista de funciones de utilidad puede

¹⁸https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php

2. ESTADO DEL ARTE

ampliarse, añadiendo más consideraciones a medida que son necesarias en el desarrollo y además pueden ser reutilizadas entre distintos agentes o incluso entre distintos proyectos. Se trata de una técnica relativamente nueva en la industria, por lo que su uso aun no está muy extendido.

Búsquedas en Árbol

Una de las bases de la inteligencia artificial son los **algoritmos de búsqueda en árbol** son una de las bases de la inteligencia artificial, dado que la mayor parte de los problemas de la inteligencia artificial podrían plantearse usando este tipo de algoritmos [YT18].

La búsqueda en árbol es un tipo de estrategia **resolvente de problemas**, cuyo comportamiento se basa en producir una lista de acciones que permita solucionar un problema dado. Para encontrar esta secuencia, se hace uso de un **árbol 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 que provocan la transición de un estado a otro. El agente inteligente recorre el árbol partiendo del nodo raíz buscando la secuencia de ramas que resuelvan el problema siguiendo un algoritmo concreto [RN08].

Existen numerosos algoritmos de búsqueda distintos, los cuales 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 más 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.

2. ESTADO DEL ARTE

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 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.

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] A continuación mencionaremos algunos de los futuros campos de aplicación de las técnicas de inteligencia artificial:

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 propiedad es la saga Left 4 Death de Valve¹⁹. 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 si, 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 2.30). Compañías como Riot Games están empezando a utilizar algoritmos de aprendizaje automático para entrenar sistemas para detectar y eliminar los mensajes inapropiados de los chats²⁰.



Figura 2.30: Ejemplo de comportamiento tóxico en *League of Legends* (Riot Games, 2009).

¹⁹<http://www.l4d.com/blog/>

²⁰<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

Como programa informático, Virus Breaker constara 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.

Una vez iniciado, el juego presentará al jugador una sencilla **pantalla de título**. La funcionalidad de esta pantalla será limitada, solo mostrará el título y el nombre del desarrollador del juego; e informará al jugador de que botón debe pulsar para iniciar una partida. En la figura 3.1 se puede ver la apariencia de esta pantalla de título en la versión final del juego.



Figura 3.1: Pantalla de título.

Una vez iniciada la partida, el jugador pasará a controlar al **personaje principal**

3. ARQUITECTURA

e intentará superar los desafíos que le presente el juego. Una versión básica de esta pantalla se puede ver en la figura 3.1. El juego se dividirá en varios **niveles**, cada vez que el jugador supere uno, el juego le presentará otro de dificultad mayor. Esta secuencia de niveles concluirá con el enfrentamiento contra un **Jefe Final**. Al derrotar al jefe final, el juego mostrara una **pantalla de victoria**. Esta pantalla será funcionalmente idéntica a la pantalla de título: mostrara un mensaje y permitirá al jugador volver a jugar al juego si pulsa un botón.

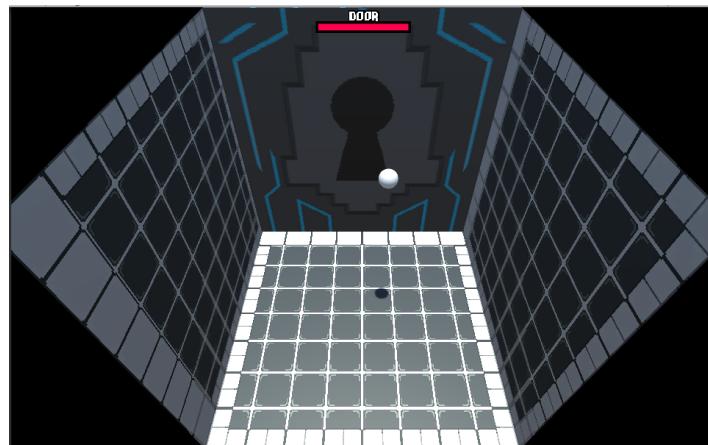


Figura 3.2: Pantalla de juego.

Si el jugador es derrotado durante el juego, se le mostrará una pantalla de **fin del juego**, donde se le ofrecerá la posibilidad de volver a iniciar la partida desde el ultimo nivel superado. A igual que la pantalla de victoria, la funcionalidad de esta pantalla será idéntica a la de la pantalla de título. La similitud entre estas tres pantallas se puede apreciar en la figura 3.1



Figura 3.3: Pantallas de victoria y derrota.

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.

3.2 Escenas

Desde un punto de vista formal, este juego, al igual que el resto de juegos, es una **aplicación gráfica interactiva** con renderizado en tiempo real [Dav15]. Este tipo de aplicaciones están construidas sobre un bucle de ejecución (llamado comúnmente **bucle de juego** en el ámbito del videojuego) el cual esta formado por tres pasos:

1. **Renderizar** una imagen a traves de la pantalla del usuario.
2. **Leer** la entrada que el usuario suministre.
3. **Modificar** el estado del programa, del cual depende la siguiente renderización.

Cuando se trata de un videojuego, este sistema de renderizado es acompañado de sistemas de audio, simuladores de físicas y otros subsistemas que permita dotar al juego de realismo e inmersividad.

En la gran mayoría de los videojuegos se pueden distinguir una serie de “etapas” o “estados”, en cada uno de los cuales se ejecuta una lógica distinta independiente. En nuestro caso, cada una de las “pantallas” descritas en el apartado anterior serían uno de estos estados. La mayor parte de los motores de juego permiten e incluso fomentan que programador realice estas divisiones del juego ofreciendo implementaciones de estos estados. dependiendo del motor, estas divisiones se pueden llamar **salas, mundos o escenas**.

En Unity, los estados del juego reciben el nombre de **Escena**¹. A grandes rasgos, las escenas son un conjunto de entidades de juego situadas en un entorno tridimensional. Estas entidades se encuentran organizadas en un **grafos de escena**, un tipo de estructura de datos que permite representar las relaciones jerárquicas que existen entre los distintos objetos del juego o “nodos” haciendo uso de un grafo dirigido sin hijos [Dav15]. El uso de este tipo de estructura permite gestionar con facilidad escenas tridimensionales complejas, permitiendo que se apliquen las operaciones transformaciones (como la traslación, rotación y escalado) de un nodo padre a todos sus nodos hijos de forma automática y facilita tareas como las búsquedas de entidades concretas.

La suite de desarrollo de Unity incluye un editor de escenas, que permite la creación de estas mediante una interfaz gráfica. Las escenas se almacenan en archivos, de la misma forma que lo hacen las texturas o los modelos 3D, lo que facilita su gestión. Para la manipulación de escenas en tiempo de ejecución se utiliza el módulo **SceneManager**² que permite realizar cambios de escenas, cargar escenas en segundo plano y solapar varias escenas simultáneamente entre otras funciones.

Para Virus Breaker, se optó por una estructura en cuatro Escenas: tres escenas de “menú” (menú de inicio, fin del juego y victoria) y una escena de juego, como puede

¹<https://docs.unity3d.com/Manual/CreatingScenes.html>

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

3. ARQUITECTURA

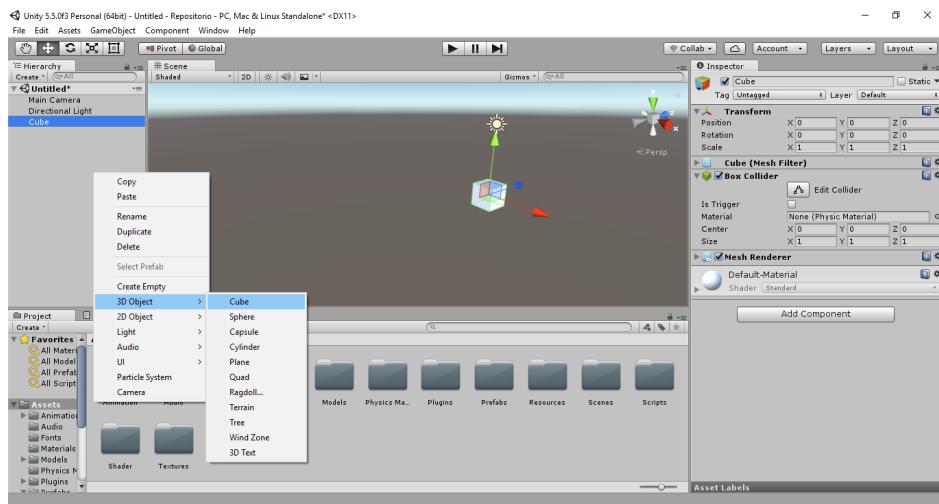


Figura 3.4: Editor de escenas de Unity.

verse en la figura 3.5. A continuación, se realizará una descripción de estos tipos de escenas

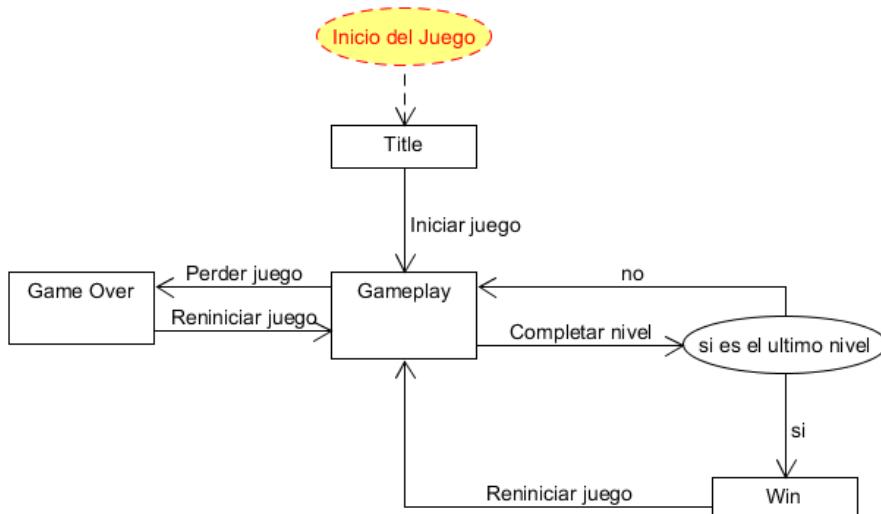


Figura 3.5: Diagrama de las escenas del juego

3.2.1 Menús

En el campo de la informática, un **menú** es un tipo de interfaz de usuario en la que se presenta una lista de elementos (que pueden ser comandos, opciones o atributos) de entre los cuales el usuario puede elegir [05]. Su principal ventaja frente a otras alternativas (como el control por línea de comandos) radica en que el usuario **no necesita aprender de memoria todas las opciones** del menú para utilizarlo. En los videojuegos, la función de los menús suele ser la de contener las acciones necesarias para la ejecución del juego, pero que no pertenecen a la jugabilidad. Estas pueden ser guardar

y cargar partidas, cambiar la configuración técnica del juego o elegir entre distintos modos de juego.

En Virus Breaker contamos con tres escenas que podrían considerarse menús: la **pantalla de título**, la **pantalla de victoria** y la **pantalla de derrota**. El comportamiento del juego en estas tres pantallas es prácticamente idéntico: El juego muestra un texto animado de gran tamaño, acompañado de un segundo texto con instrucciones para el jugador. Si el jugador pulsa el botón que aparece en las instrucciones, la escena cambiará a la escena de juego.

La animación de los textos se realizó mediante **Mecanim**³, el sistema de animaciones integrado de Unity, que permite realizar cambios en parámetros de objetos de juego en función del tiempo. Los textos pequeños fueron creados también en Unity, pero los textos grandes fueron modelados en **Blender**⁴ y luego importados en Unity.

El código que controla los menús es realmente simple, un pequeño script en C# revisa si el jugador ha pulsado la tecla correspondiente y cuando esto pasa llama al gestor de escenas de Unity para cargar la escena de juego.

3.2.2 Escena de Juego

La escena de juego es en la que ocurre **la acción del juego**, donde el jugador controla el personaje principal intentando completar los distintos niveles del juego. Como puede verse en el diagrama 3.5, se trata de la escena central del juego, a donde se dirigen el resto de las escenas.

Se trata por tanto de **la escena más compleja del juego**, la que tiene mayor número de elementos e interacciones y en la que más tiempo pasa el jugador durante su partida. En la figura 3.6 se pueden ver detallados los distintos elementos de alto nivel que tiene esta escena:

1. **Sala:** La sala es el entorno en el que ocurre la partida. Está compuesta de tres paredes, suelo, techo y una puerta. La sala lleva el control del comportamiento general de la partida, encargándose de tareas como realizar transiciones de salas o activar y desactivar otros elementos según sea necesario.
2. **Personaje Principal:** Es el elemento sobre el que el jugador tiene un control directo. Puede moverse y crear una paleta. Su comportamiento se basa en una máquina de estados finitos.
3. **Pelota:** La pelota es un objeto que se mueve por la sala rebotando en sus paredes. Es posible el elemento más importante de la sala, ya que la mayor parte de interacciones entre objetos se realizan a través de la pelota

³<https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html>

⁴<https://www.blender.org/>

3. ARQUITECTURA

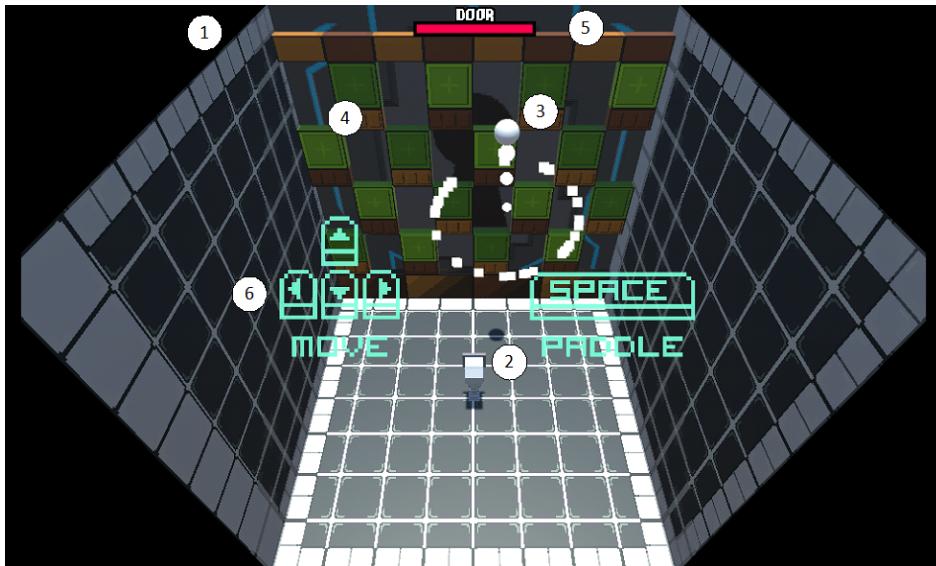


Figura 3.6: Diagrama de las escenas del juego

4. **Ladrillos:** Los ladrillos son unos elementos destructibles los cuales forman un muro que protege la puerta. Hay distintos tipos de ladrillos, cada uno con un comportamiento distinto.
5. **Resistencia de la puerta:** Se trata de una imagen que representa cuantos golpes necesita la puerta para abrirse.
6. **Tutorial en pantalla:** Son dos imágenes que muestran los controles del juego. Solo aparecen la primera vez que se inicia una partida, para no sobrecargar al jugador con información.

Carga de Niveles

Este juego cuenta con varios niveles, cada uno con una configuración de ladrillos distinta. Dado el gran número de elementos comunes que existen entre los distintos niveles (la geometría de la sala, el personaje, la pelota, la puerta...), utilizar una sala distinta por cada uno de los niveles **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.

Para paliar con estos problemas, se optó por utilizar un editor externo para producir los niveles. Los niveles se **almacenarían en forma de archivos** que la escena se encargaría de leer para generar las formaciones de ladrillos correspondientes. La funcionalidad que se necesitaba para el editor de niveles era la siguiente:

- **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 3.7).
- **Simplicidad** en el proceso de creación. La creación de niveles debe de ser rápida, para facilitar la iteración el diseño.



Figura 3.7: Homenaje a Space Invaders(Taito, 1978) dentro de Arkanoid: Doh it Again (Taito, 1997)

En una primera versión, se planteó el uso del programa **Tiled**⁵, 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 generados con tiles eran exportados al formato **XML**, el cual podía ser cargado en Unity. Sin embargo, al ser un programa de propósito general, Tiled contaba con mucha funcionalidad innecesaria que ralentizaba la producción de niveles, al mismo tiempo que carecía de ciertas funciones clave (como la selección individual de colores de bloques).

En la iteración final, se optó por almacenar la información de los niveles en forma de imágenes. Este nuevo sistema lee **pequeñas imágenes** que contienen la información del nivel codificada en los colores de sus pixeles. En este sistema toma dos imágenes para generar el nivel. La primera imagen sirve para determinar el tipo y posición de los bloques, mientras que la segunda imagen determina los colores de cada bloque. Un **archivo JSON adicional** sirve para “enlazar” ambas imágenes y para contener

⁵<http://www.mapeditor.org/>

3. ARQUITECTURA

información adicional, como la cantidad de golpes que debe recibir la puerta para abrirse o de si debe iniciarse o no un combate de jefe.

La lógica para la carga de niveles se encuentra en la clase **LevelGenerator**. Esta clase contiene la información necesaria para la carga de nivel (índice del nivel actual, formato del nombre de los archivos, coordenadas del punto de aparición...). El proceso para generar el nivel es el siguiente:

1. La clase **lee archivo JSON** determinado por el índice del nivel actual. Usando la clase `JsonUtility`⁶ de Unity, la información del archivo se encapsula automáticamente un objeto para facilitar su acceso.
2. Se **determina si se trata de un nivel de jefe** o no. En ese caso se detiene el proceso y se empieza a preparar la batalla con el jefe. En caso contrario, se continua con el proceso.
3. 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 `Color`. Llamaremos **Textura de tipos** a aquella que determina el tipo de los ladrillos y **Textura de colores** a aquella que determina sus colores.
4. Se recorre la textura de tipos **obteniendo los valores de sus píxeles**. Cada pixel se compara con una tabla de equivalencias que relaciona un tipo de ladrillo con un color. En caso de acierto, se instancia un bloque del tipo determinado, al cual se asigna el color del pixel de la textura de colores de la misma posición.
5. Dado que los ladrillos ocupan más de una “casilla”, cada vez que uno es creado se marcan las posiciones que ocupan en una “**matriz de ocupación**”. durante el recorrido de la textura, se ignorarán los píxeles que correspondan a posiciones ocupadas.
6. Finalmente, se asigna **el número de golpes** que requiere la puerta.

De esta forma es posible producir niveles rápidamente, los cuales son fáciles de modificar. 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.

Por otra parte, el este sistema también limitaciones:

- 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

⁶<https://docs.unity3d.com/ScriptReference/JsonUtility.html>

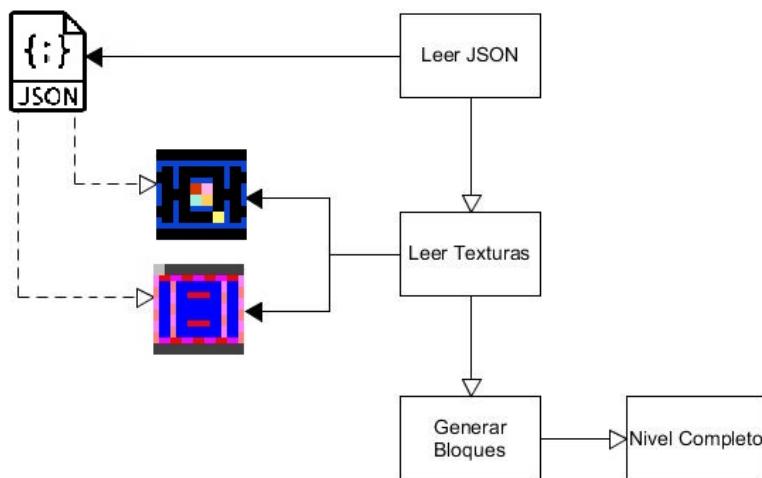


Figura 3.8: Diagrama del proceso de generación de niveles

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, 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

3.3 Objetos

El comportamiento de un videojuego desarrollado en Unity viene dado por los **objetos** que hay en el juego y las interacciones que existen entre estos. Estos objetos deben de poder influir en distintos dominios del código, como el renderizado, el sonido o las físicas, lo cual puede causar un importante problema de acoplamiento entre clases. Para solucionar este problema, Unity hace uso del patrón de diseño **Component**. Este patrón de diseño consiste en aislar los distintos dominios en clases **componentes**, las cuales pueden ser asociadas a un objeto contenedor de componentes [game_programming_patterns].

La clase contenedor de componentes en Unity se llama **GameObjects**⁷. Los GameObjects se instancian mediante el editor de Unity o mediante código con el método estático **Instantiate**⁸. Estos objetos carecen de funcionalidad, a excepción de una serie de métodos para realizar búsquedas entre los GameObjects de una escena. La funcionalidad de los GameObjects viene dada por los objetos de la clase **Component**⁹. Esta clase es padre de multitud de componentes con diferente funcionalidad. Algunos de los

⁷<https://docs.unity3d.com/ScriptReference/GameObject.html>

⁸<https://docs.unity3d.com/ScriptReference/Object.Instantiate.html>

⁹<https://docs.unity3d.com/Manual/Components.html>

3. ARQUITECTURA

componentes más importantes de Unity son los siguientes:

- **Transform**¹⁰: 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 GameObjects tienen uno asociado.
- **Collider**¹¹: 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**¹²: Esta clase se utiliza para la simulación física de cuerpos rígidos 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**¹³: Este componente permite renderizar modelos y texturas en la escena.
- **Audio Source**¹⁴: 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**¹⁵: 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**¹⁶: Es un componente que emite **partículas**, pequeñas imágenes 2D que permiten crear efectos visuales como fuego o explosiones.

A parte de los componentes incluidos en Unity, los desarrolladores pueden construir sus propios componentes y añadirlos a los GameObjects. La clase **MonoBehaviour**¹⁷ se utiliza como base para estos componentes, incluye una serie de métodos llamados **eventos** que son llamados automáticamente cuando se cumplen ciertas condiciones, como por ejemplo al principio de la ejecución (evento Start).

¹⁰<https://docs.unity3d.com/ScriptReference/Transform.html>

¹¹<https://docs.unity3d.com/ScriptReference/Collider.html>

¹²<https://docs.unity3d.com/ScriptReference/Rigidbody.html>

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

¹⁴[https://docs.unity3d.com/ScriptReference/](https://docs.unity3d.com/ScriptReference/<AudioSource.html)

¹⁵<https://docs.unity3d.com/ScriptReference/Animator.html>

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

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

En los siguientes apartados contienen una descripción de los distintos objetos que forman el juego. Estas descripciones describen la función del objeto, jerarquía de GameObjects que lo forman, los Componentes que tienen asociados y el código de los scripts que definen su comportamiento.

3.3.1 Personaje Principal

El **personaje principal** es el avatar del jugador en el juego, al que controla directamente mediante el teclado. Su comportamiento es bastante sencillo: el personaje se mueve cuando el jugador pulsa las flechas de dirección, y crea una “paleta” cuando se pulsa la tecla espacio. Usando la paleta, el personaje es capaz de redirigir la pelota, pero si la pelota impacta contra el directamente se quedará aturdido unos segundos. La paleta solo permanece activa unos instantes, en los cuales el personaje permanece inmóvil, por lo que presenta un desafío para el jugador, que debe saber posicionarse correctamente. El personaje reproduce también una animación al inicio de los niveles y otra al final, en la que entra y sale de la sala del juego.

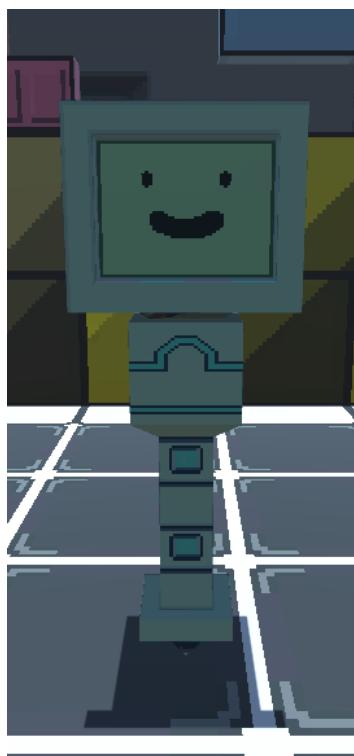


Figura 3.9: Modelo del personaje principal

Para la implementación del jugador se utilizan dos GameObjects anidados. El GameObject padre se encarga de la lógica del personaje mientras que el objeto hijo sirve para contener el modelo del personaje. La lógica del GameObject principal viene dada por una serie de componentes que tiene asociados:

- Un **Rigidbody** y un **Collider**. Estos componentes se encargan, respectivamente,

3. ARQUITECTURA

te, de almacenar las propiedades físicas del personaje (velocidad, aceleración, gravedad...) y de realizar la detección de colisiones con otros objetos.

- Un **Audio Source**, el cual se encarga de reproducir los sonidos del personaje.
- Un **Animator**, el cual reproduce las animaciones del jugador. 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 coordinar las acciones del resto de componentes.

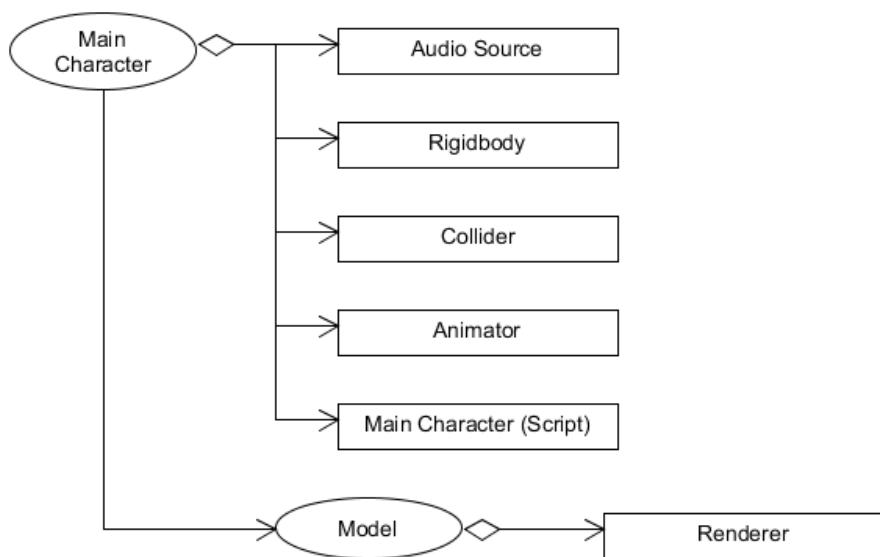


Figura 3.10: Diagrama de componentes del personaje principal.

El Script del personaje principal basa su funcionamiento en una **Maquina de Estados Finitos**. Los estados (Figura 3.11) que componen el comportamiento del personaje principal son los siguientes:

- **Enter**: Es el estado inicial. En este estado el personaje, posicionado fuera de escena, se mueve hacia el interior de la sala hasta colocarse en el centro de esta. Durante este estado, la colisión y la gravedad están desactivadas, lo que le permite entrar en la sala cerrada. Una vez terminada la animación, pasa al estado **Stand**.
- **Stand**: Se trata del estado principal. En él, el personaje espera inmóvil a recibir las órdenes del personaje. Si se pulsan las flechas de dirección, el personaje pasará al estado **Move** y si se pulsa la tecla espacio, se pasará al estado **Paddle**.
- **Move**: En este estado el personaje se mueve a velocidad constante en la dirección determinada por las flechas pulsadas. Cuando ninguna de las flechas de dirección esté siendo pulsadas, se volverá al estado **Stand**. En este estado también se puede pulsar la tecla espacio para moverse entrar en el estado **Paddle**.

- **Paddle:** En este estado el personaje crea frente a él un objeto Paleta que sirve para desviar la pelota. La paleta tiene un temporizador que hace que, al acabarse, la paleta desaparezca y el personaje vuelve al estado **Stand**. Durante este estado el jugador no se puede mover, aunque si se entró desde el estado **Move** todavía se conservará parte de la velocidad debido a la inercia.
- **Confused:** El personaje entra en este estado cuando la pelota le golpea, independientemente de en qué estado se encuentre. En este estado, el jugador girará aturdido durante unos instantes antes de volver al estado **Stand**, como penalización para el jugador por no haber golpeado la pelota correctamente.
- **Dead:** El personaje entra en este estado cuando la pelota es destruida, sin importar el estado anterior. En este estado el personaje se desploma derrotado y se queda inmóvil, a espera de la pantalla de “fin del juego”.
- **Exit:** Cuando se completa un nivel, el personaje entra en este estado. El estado tiene dos partes: primero, el personaje permanece inmóvil a la espera de que la puerta del nivel se abra; y una vez esta esté abierta, el personaje avanza a través de ella hacia el siguiente nivel.

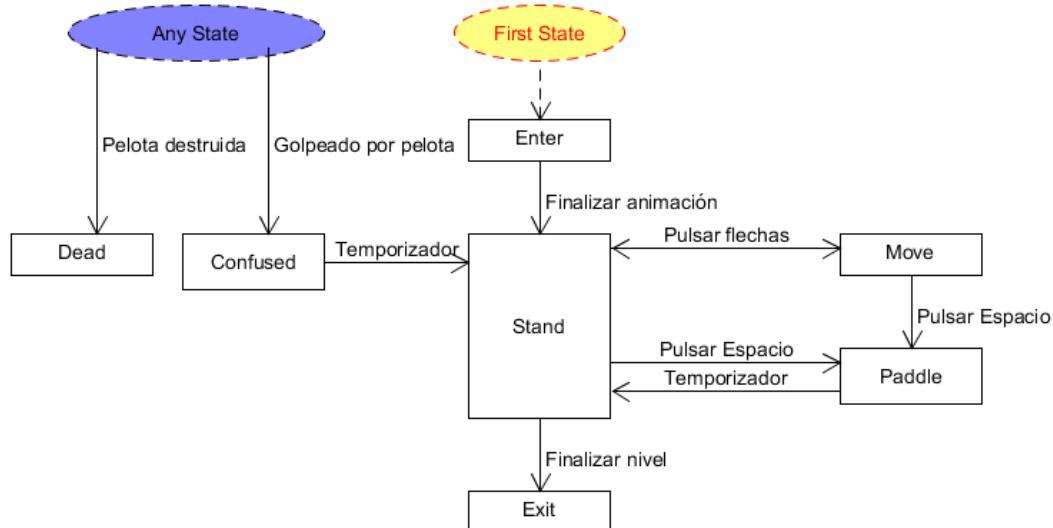


Figura 3.11: Diagrama de estados del personaje principal

Una parte fundamental del comportamiento del personaje es su movimiento, presente en la mayor parte de los estados, tanto los controlados por el jugador (Move y Paddle) como los movimientos automáticos (las animaciones de Enter y Exit). Estos movimientos están implementados mediante el sistema de físicas, aplicándole al jugador una fuerza en la dirección en la que se desea que se mueva. Se utilizan fuerzas en lugar de modificar directamente el valor de velocidad del personaje para darle al movimiento una ligera aceleración inicial, que es mucho más agradable y natural para el jugador. La fuerza

3. ARQUITECTURA

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. Esto se combina con una reducción de la fricción en el estado **Paddle** para que el personaje conservase parte de su velocidad al sacar la paleta, que sirve para aumentar el margen de error del jugador a la hora de intentar golpear la pelota.

La **Paleta** que el jugador crea durante el estado **Paddle** es un **GameObject** con un **Renderer** y un **Collider** asociado. Este objeto esta guardado como un **Prefab** que el personaje instancia al principio del estado. Durante la instanciación, el personaje incluye a la paleta en su jerarquía de objetos, para que siga los movimientos del personaje. Al finalizar el estado, la instancia de la paleta es destruida.

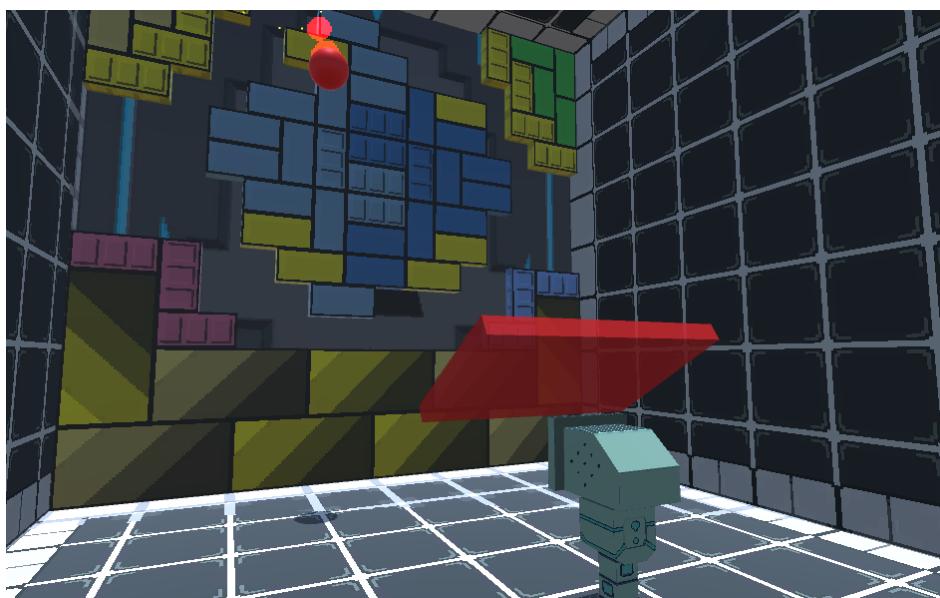


Figura 3.12: Personaje principal activando la paleta

3.3.2 Bola

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 trayectoria 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.

La pelota se compone de un único **GameObject**, el cual tiene una serie de componentes asociados:

- Un **Renderer**, el cual dibuja en pantalla el modelo de la bola.
- Un **SphereCollider** para la detección de colisiones

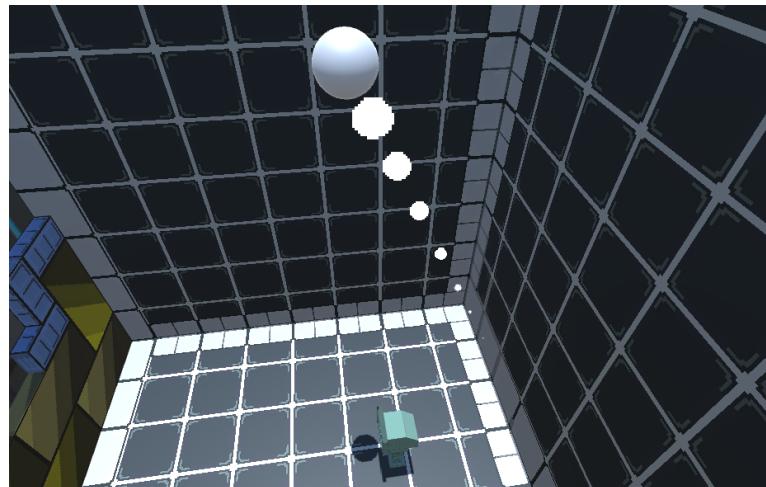


Figura 3.13: Pelota moviéndose

- Un **Rigidbody** para el control de movimiento.
- Un **Audio Source** que emite los sonidos de impacto contra otros objetos.
- Un **Animator** para ejecutar la animación de destrucción de la bola.
- Un **Particle System** que se utiliza para producir varios efectos especiales.
- Un **Script** de la clase **Ball** que controla el comportamiento de la bola, especialmente su reacción cuando colisiona con otros objetos.

El script **Ball** basa su funcionamiento en una máquina de estados finitos al igual que el jugador; sin embargo, es significativamente más simple ya que cuenta solo con tres estados: **Normal** (su comportamiento normal de movimiento), **Destroy** (animación de destrucción de la pelota que se reproduce al final del juego) y **Locked** (estado especial en el que la pelota permanece quieta e invisible, para la cinemática de inicio y final del nivel).

La complejidad de este script reside en la **tabla de interacciones** de la bola, debido a que la bola tiene una reacción distinta dependiendo de con qué objeto del juego con el que colisione. Las posibles reacciones son:

- Al golpear la **Puerta** de la sala o los **Bloques** que la cubren, estos sufirán un punto de daño, y la pelota rebotará. Al iniciarse el rebote, la pelota empezará a caer en dirección al suelo de la sala.
- Al golpear las **Paredes** o el **Techo** de la sala, la pelota rebotará de forma natural.
- Al golpear el **Suelo** de la sala, la pelota recibirá un punto de daño. Además, la pelota rebotará, pero no de forma natural sino de forma que vaya en dirección a la puerta. Esto sirve principalmente para facilitar la tarea del jugador, “regalándole” un golpe a la puerta. La pelota también pierde la fuerza de la gravedad.

3. ARQUITECTURA

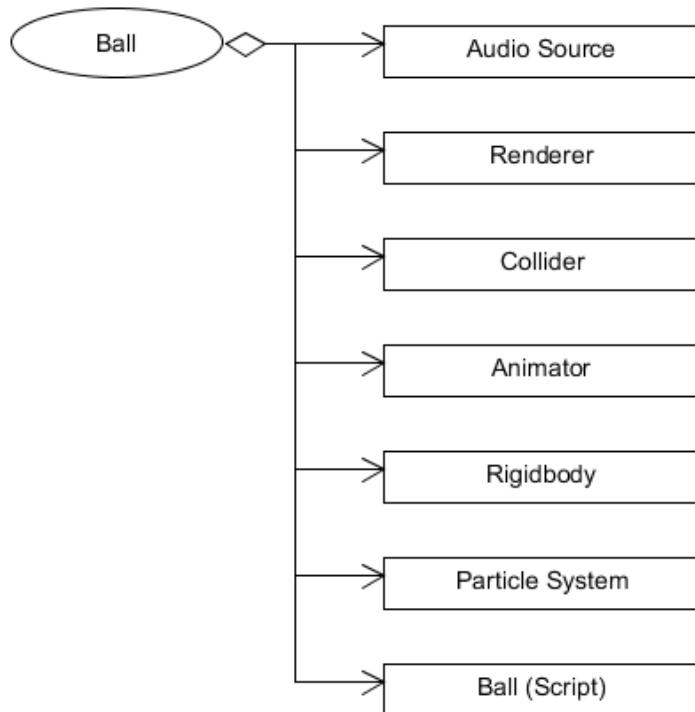


Figura 3.14: Componentes del objeto Ball.

- Al golpear al **jugador**, este se quedará aturdido unos instantes y la pelota rebotará de forma natural.
- Al golpear la paleta del jugador, la pelota rebotará y perderá la fuerza de la gravedad. La dirección en la que rebotará la pelota dependerá del punto de la paleta en el que golpeó la pelota, siendo la dirección normal al plano de la paleta si se golpea justo en el centro.

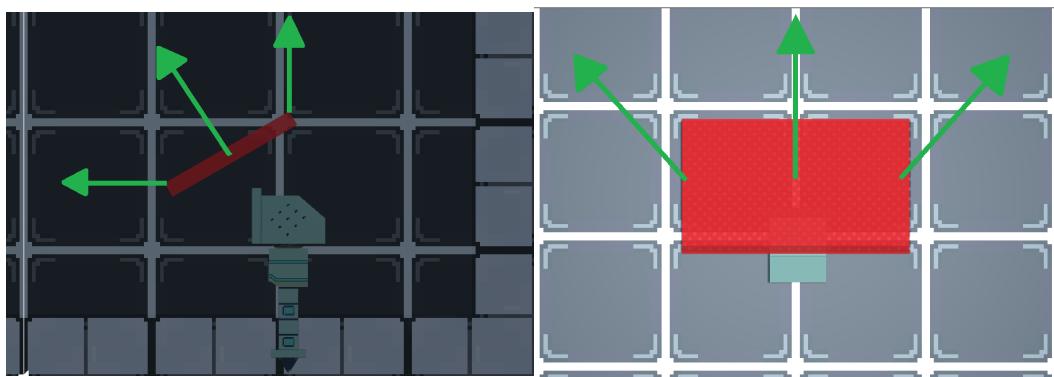


Figura 3.15: Direcciones que tomaría la pelota dependiendo del punto de la paleta que golpee (Aproximada)

El movimiento de la pelota se basa íntegramente en el motor de físicas de Unity. Al entrar en su estado **Normal**, la pelota recibe un impulso en una dirección dada y a

partir de ahí se moverá sin fricción de forma perpetua. Los movimientos “antinaturales” mencionados, el cambio de dirección al golpear el suelo o la paleta se realizan cambiando el vector de la velocidad de la pelota por un vector nuevo generado manualmente por código.

Aunque la pelota no tiene animación en si misma, a su comportamiento se le han añadido unos efectos de partículas, que sirven tanto para embellecer el juego como para crear guías visuales para el jugador. La pelota cuenta con tres efectos animados de partículas: una “**cola**” que marca la trayectoria que siguió la bola; un efecto circular que aparece cuando la bola **colisiona con las paredes**, el techo y el suelo, que sirven para marcar el punto de impacto de la bola y un efecto de **explosión final**. La trayectoria se produce mediante el componente **Particle System** de la pelota, pero para el efecto de impacto y explosión se hace uso de objetos externos, los cuales son instanciados, producen sus partículas y se eliminan automáticamente.

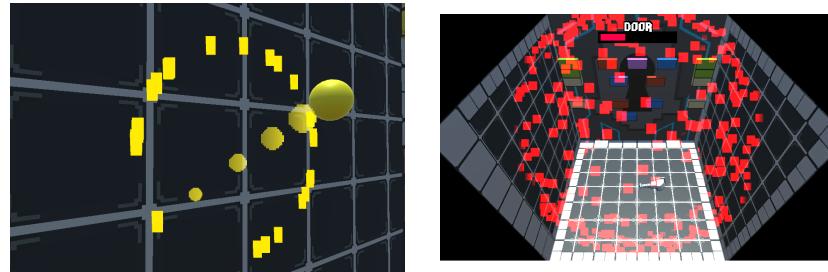


Figura 3.16: Efectos de partículas: impacto en muro (izquierda) y explosión (derecha)

3.3.3 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. 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:

- **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.

3. ARQUITECTURA

- **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.

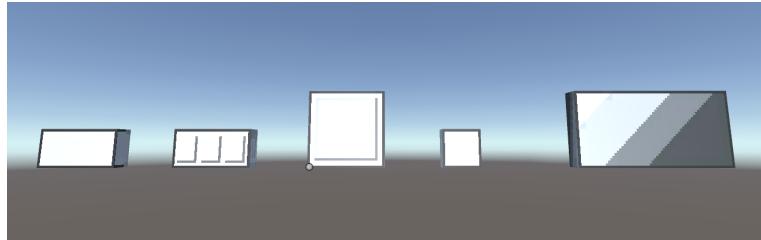


Figura 3.17: Modelos de los ladrillos

Independientemente de su comportamiento, cada ladrillo puede tener asignado un **color**. Esto permite dar variedad a los niveles y además hacerlos más memorables haciendo dibujos con los ladrillos.

Para implementar los ladrillos se utilizan dos GameObjects anidados. 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.

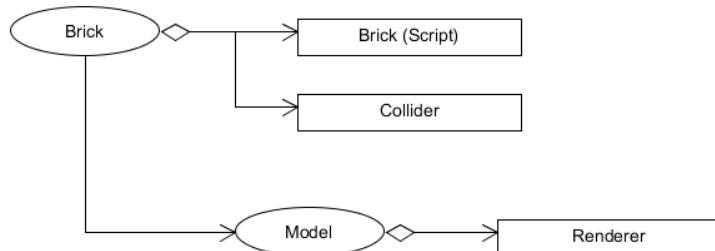


Figura 3.18: Componentes del objeto Brick.

El método principal de **Brick** se llama **DoDamage**. Este método recibe como parámetro una cantidad de “puntos de daño” la cual sustraer a los puntos de vida actuales del ladrillo. 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. Este sería el comportamiento de los ladrillos básicos, para programar los comportamientos específicos de los distintos tipos de ladrillos otros scripts que la funcionalidad de la clase **Brick**.

El cambio de textura de los ladrillos multigolpe está programado en la clase **TextureChangeBrick**, en la cual se añade código adicional al método **DoDamage** para cambiar la textura del ladrillo de entre las presentes en una lista. El comportamiento del bloque divisible está en la clase **DivisibleBrick**, en la que durante la destrucción del ladrillo se instancian los cuatro ladrillos pequeños (que son ladrillos básicos pero con un modelo más pequeño). El script **UnbreakableBrick** es el que usan los ladrillos irrompibles, y en el el método **DoDamage** está vacío.

Todos estos ladrillos se guardan como prefabs entre los assets del juego, para que el generador de niveles los pueda instanciar.

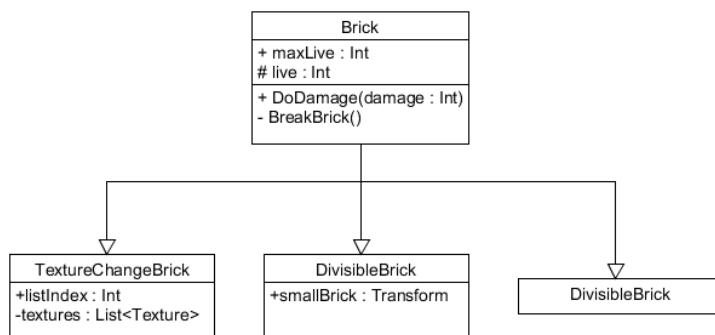


Figura 3.19: Jerarquía de clases.

3.3.4 Sala y Puerta

La **sala** es el entorno donde transcurre la acción del juego. Se trata de una habitación cúbica con una puerta cerrada en una de sus paredes. Frente a la puerta hay un muro formado por ladrillos cuya distribución cambia dependiendo del nivel. La puerta reacciona a los golpes con la pelota, moviéndose y perdiendo “vida” con cada golpe. Cuando la vida de la puerta llega a cero, la puerta se abre y el jugador avanza al siguiente nivel.

La implementación de la sala se basa en una combinación de GameObjects agrupados de forma jerárquica. Esto permite añadir funcionalidad diferente a los distintos elementos que conforman la sala. 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 GameObjects tienen dos componentes: Un **MeshCollider** y un **Renderer**.
- **Floor:** El suelo de la sala es similar a las paredes (un GameObject con un **MeshCollider** y un **Renderer**), pero está marcado de forma diferente para activar un comportamiento distinto en la pelota.

3. ARQUITECTURA

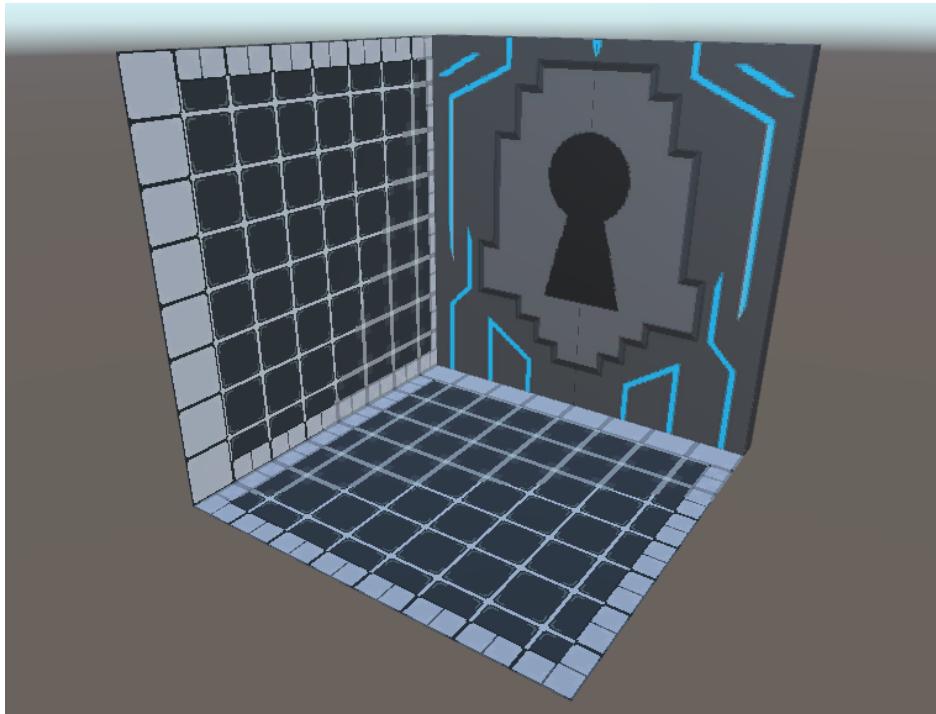


Figura 3.20: Sala vista desde el editor.

- **Door:** La puerta de la sala ocupa la pared norte de la sala. Este GameObject tiene asociados tres componentes: un **BoxCollider**, un **Animator** y un script de clase **Door**. Anidados, la puerta contiene dos objetos **panel** independientes.

Los dos scripts con los que cuenta la sala cumplen dos funciones muy distintas. **LevelGenerator** contiene el código responsable de la generación de niveles tal y como se describió en el apartado 3.2.2. El script **Room**, por otro lado, se encarga de la gestión del estado de la sala.

Durante la ejecución del juego tienen lugar diversos eventos que deben producirse en un orden determinado. Estos eventos son producidos por diversos scripts asociados a GameObjects diferentes. El script **Room** sirve como intermediario entre los diversos objetos, recibiendo los mensajes producidos en los eventos y reenviándolos a los scripts que se encargan de iniciar los eventos siguientes. Dado la simplicidad de la secuencia de eventos, el sistema de mensajes se implementa mediante una serie de funciones públicas en el script **Room**.

La secuencia de eventos del juego puede agruparse en las siguientes etapas:

1. **Preambulo:** Es la secuencia de eventos anterior a que el jugador tome el control del juego. Los eventos de los que consta son los siguientes:
 - a) **Generación del nivel:** Cuando la escena se carga, el script **Room** llama a **LevelGenerator** y le suministra el índice del nivel que debe generarse.
 - b) **Entrada del jugador:** Simultáneamente a la carga del nivel, el **Personaje**

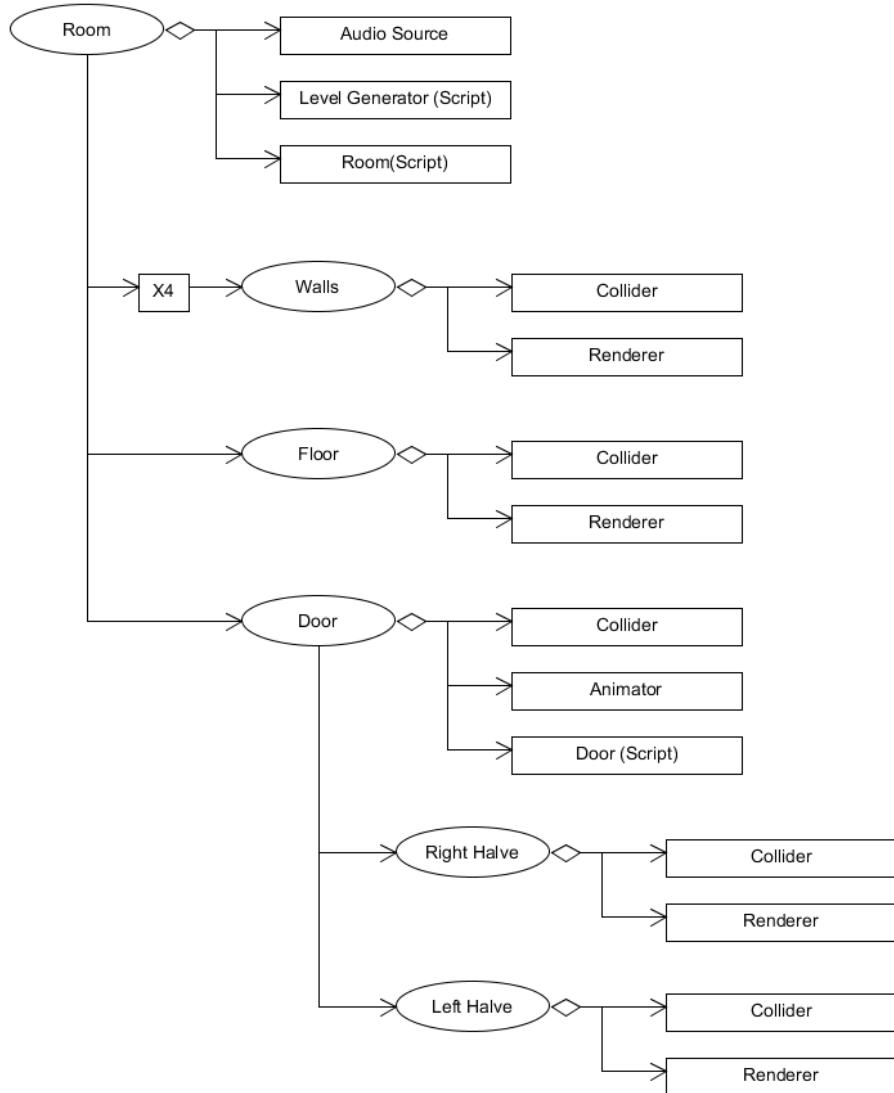


Figura 3.21: Diagrama de componentes de la sala.

Principal avanza desde fuera de la sala a su interior a través de la pared sur. El jugador llama a la sala cuando alcanza el centro de esta.

- c) **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 llama a la sala una vez su animación ha terminado.
 - d) **Activación de la pelota:** Una vez que el jugador (y el jefe) han entrado en la sala, la pelota se activa, haciéndose visible en el centro de la sala.
 - e) **Control del jugador:** Al mismo tiempo que se activa la pelota, el personaje principal empieza a reaccionar a las órdenes del jugador comienza el juego.
2. **Juego:** En esta etapa de la ejecución, el jugador tiene control sobre el personaje principal. La etapa acaba cuando la puerta se queda sin vidas (lo que inicia la etapa de **Victoria**) o cuando la pelota toca el suelo tres veces (lo que inicia la

3. ARQUITECTURA

etapa de **Derrota**).

3. **Victoria:** Cuando el jugador supera un nivel, se sucede la siguiente secuencia de eventos:

- a) **Ocultar pelota:** Se envía un mensaje a la pelota para detener su movimiento y desactivar tanto su collider como su renderer, provocando que la pelota se vuelva invisible e intangible.
- b) **Animación de la puerta:** Al mismo tiempo que se desactiva la pelota, se envía un mensaje a la puerta para que empiece su animación de apertura.
- c) **Romper los ladrillos:** Cuando acaba la animación de la puerta, esta manda un mensaje a todos sus ladrillos para que se destruyan.
- d) **Salida del personaje principal:** Una vez finalizada la animación de la puerta, se envía un mensaje al personaje principal para que salga de la sala por la puerta.
- e) **Cambio de escena:** Cuando el personaje principal se ha alejado lo suficiente de la sala comienza el 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.

4. **Derrota:** La secuencia de acciones que ocurren cuando el jugador pierde el juego son las siguientes:

- a) **Destrucción de la Pelota:** Se envía un mensaje a la pelota, que comienza su animación de destrucción.
- b) **Derrota del jugador:** Cuando la pelota acaba su animación, se envía un mensaje al jugador para que comience con su animación de derrota.
- c) **Cambio de escena:** Un tiempo después del final de la animación del jugador, se inicia el cambio de escena a **la escena de fin del juego**.

El comportamiento de la **Puerta** es independiente de el de la sala. La función de su script, **Door**, es la de detectar la colisión de la pelota con la puerta para así reducir los puntos de vida de esta e iniciar el final del nivel si los puntos de vida llegan a cero.

3.3.5 Jefe

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 suponer una variación con respecto al juego corriente 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 determinara su eficiencia a la hora de jugar e ira consumiéndose según avance la partida.



Figura 3.22: Modelo del Jefe.

La implementación del jefe utiliza **dos GameObjects anidados**. El GameObject principal contiene los componentes necesarios para el funcionamiento del jefe, mientras que el GameObject anidado contiene el modelo de este. Los componentes del GameObject padre son los siguientes:

- **RigidBody**: Este componente almacena las propiedades físicas del jefe, como su velocidad o aceleración.
- **Collider**: Este componente permite realizar la detección de colisiones entre el jefe y otros objetos.
- **Animator**: Permite añadir animaciones al GameObject, las cuales alteran las propiedades de sus componentes basándose en el tiempo. Se utiliza para crear dotar al jefe de animaciones sencillas de modo que no resulte demasiado estático.
- **Audio Source**: 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.
- Dos **Scripts** que controlan el comportamiento del jefe: **BossController** y **BossAI**.

3. ARQUITECTURA

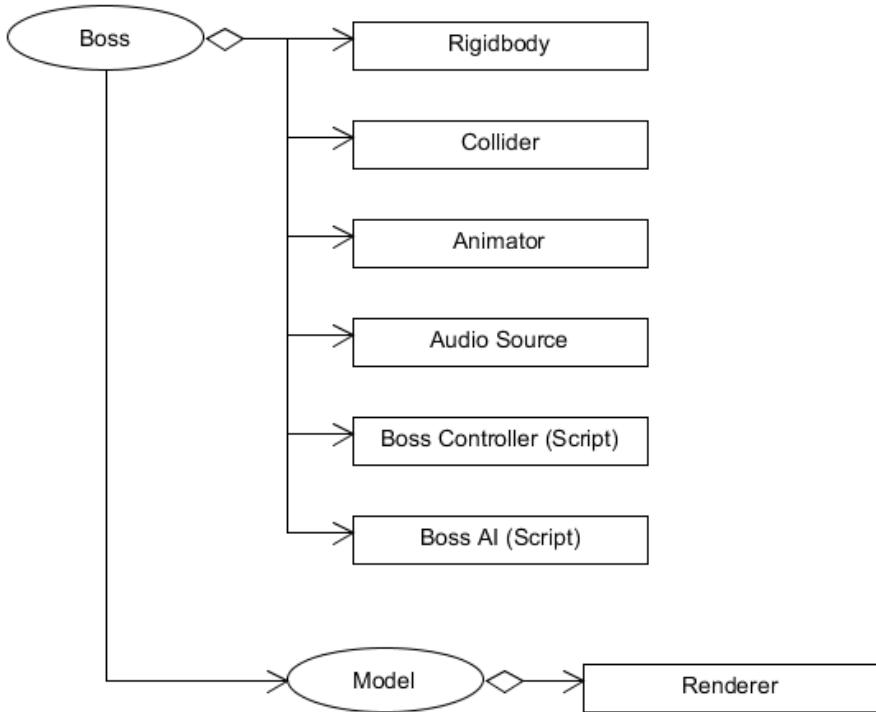


Figura 3.23: Diagrama de componentes del jefe.

El comportamiento del jefe se encuentra dividido en dos scripts para separar las acciones que puede realizar el jefe de la inteligencia artificial que determina cuales de esas acciones debe realizar. Los dos scripts implementan el patrón de diseño **Strategy**[Dav15].

BossController contiene la información y las funciones necesarias para facilitar el movimiento del jefe. El método principal de esta clase, *MoveDirection()*, permite mover al jefe a velocidad constante en el ángulo suministrado como parámetro, siempre de forma paralela al plano de la puerta. El movimiento se implementa a través del componente Rigidbody, aplicando fuerza constante en la dirección indicada, lo que crea un movimiento con una ligera aceleración que resulta más natural que un cambio instantáneo de velocidad.

El script implementa también el sistema de energía. El jefe tiene una variable que determina la cantidad de **energía** que tiene el jefe, la cual se utiliza para calcular la velocidad de su movimiento: cuanta más energía, más rápido es el movimiento. La energía baja cada vez que la pelota golpea al jefe e incrementa cuando golpea la puerta.

El segundo script, **BossAI**, implementa la inteligencia artificial del jefe. El script se encarga de determinar cuándo y en qué dirección debe moverse, que animación debe reproducirse y cuando reproducir los efectos de sonido. La implementación

de este comportamiento se realiza mediante una **Máquina de Estados Finitos**. Los estados de los que se compone son los siguientes:

- **Out:** El jefe espera fuera del área de juego, inmóvil, a la espera de la entrada del jugador en la sala.
- **Intro:** El jefe se mueve a su posición inicial y realiza su animación inicial. Este movimiento se implementa mediante el uso de animaciones en lugar de mediante el BossController dado que debe atravesar las paredes de la sala.
- **Wait:** En este estado, el jefe permanece inmóvil a la espera de un estímulo que le haga cambiar iniciar el movimiento, en concreto, que se detecte que la pelota se aproxima a la puerta.
- **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, en la que el jefe se vuelve invisible y emite una explosión de partículas.

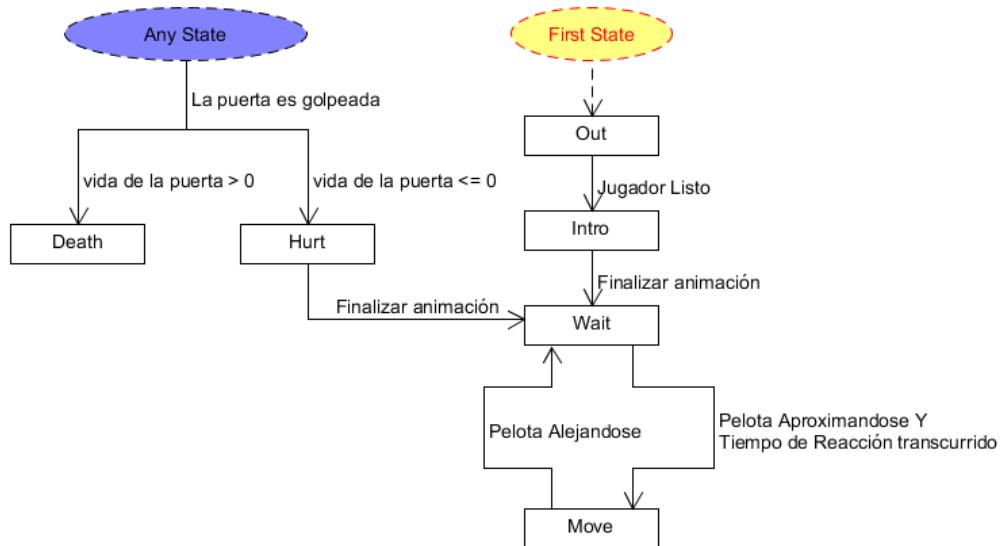


Figura 3.24: Diagrama de estados del jefe.

Capítulo 4

Conclusiones

4.1 Resultados



Figura 4.1: Relación entre desafío y la ansiedad/aburrimiento.

El desarrollo del juego tuvo éxito en implementar las características básicas del juego. Algunas de estas características son:

- El control de personaje principal fue desarrollado completamente, de forma que es preciso y fácil de controlar. Se pudo implementar unas pequeñas cinemáticas para su entrada y salida de la sala.
- El movimiento de la pelota y su interacción con el resto de los elementos también fue implementado con éxito, incluyendo las modificaciones a las trayectorias de rebote naturales que mejoraban la experiencia de juego.
- La funcionalidad deseada para el sistema de carga de niveles pudo ser implementada a la perfección, con posibilidad de fácil ampliación en caso de necesidad.

4. CONCLUSIONES

- Relacionado con los niveles, se pudieron implementar los cinco tipos de bloques descritos en el diseño.
- El jefe final pudo ser implementado.
- Las pantallas de título y fin del juego pudieron ser implementadas y decoradas con textos animados. En adición a estas dos primeras pantallas, se creó una pantalla de victoria.



Figura 4.2: Relación entre desafío y la ansiedad/aburrimiento.

El estilo artístico del juego pudo ser implementado completamente. Todos los elementos del juego cuentan con su modelo y textura propios, gracias a la simpleza del estilo gráfico. También se añadieron pequeñas animaciones simples al personaje principal, a la puerta, al jefe y a los textos de los menús. Las partículas que se utilizan para enfatizar determinadas acciones, como el movimiento de la pelota o la destrucción de bloques, se implementaron utilizando texturas sólidas de baja resolución, lo que conjuga con el resto del estilo.

El juego se encuentra alojado tanto en GameJolt(<https://gamejolt.com/games/virus-breaker/316764>) como en Itch.io(<https://pedro-romero.itch.io/virus-breaker>). En ambas páginas, el juego puede ser descargado gratuitamente para la plataforma Windows.

4.2 Mejoras Futuras

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. En caso de querer retomar este proyecto por un estudio profesional, el juego debería ser modificado para añadir

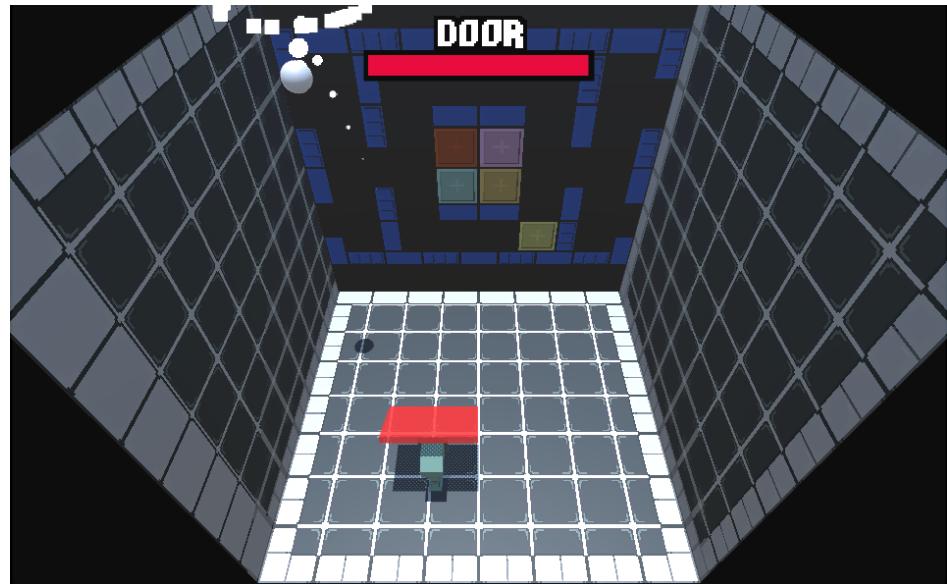


Figura 4.3: Relación entre desafío y la ansiedad/aburrimiento.

gráficos y sonidos de mejor calidad, aumentar su duración con más niveles y limpiar y optimizar su código.

A continuación, hablaremos de algunos de los elementos que podrían ser añadidos en versiones futuras del juego. Muchos de ellos son elementos que se idearon originalmente para formar parte de esta versión del proyecto, pero fueron descartados por falta de tiempo y recursos.

4.2.1 Escenarios

Lo primero que se añadiría al juego serían más niveles de juego. Al aumentar el número de niveles supondrá un incremento de la duración del juego, lo que resultará más atractivo para el jugador. Dado que los juegos de este género pueden superar el centenar de niveles (por ejemplo, LEGO Bricktopia¹ cuenta con 150 niveles) por lo que sería necesario implementar un número mayor de niveles para poder competir con ellos.

La cantidad de niveles que se pueden realizar no es infinita: solo alternando la configuración de los bloques que existe actualmente se llegará pronto a un punto en el que no se podrán crear más niveles sin caer en la repetición. La solución a este problema sería la implementación de más elementos de juego que dotaran de variedad a los niveles.

La creación de más tipos de bloques sería una de las primeras soluciones, ya que podrían integrarse con relativa facilidad al sistema actual de codificación de niveles. Entre los posibles bloques estarían los bloques explosivos (destruyen bloques cercanos al ser destruidos), bloques móviles que pudiesen programarse para crear formaciones

¹<https://www.bigfishgames.com/games/1252/legobricktopia/>

4. CONCLUSIONES

móviles, o bloques con propiedades físicas distintas que alteraran el movimiento de la pelota (aumentando su velocidad o haciendo que rebote en ángulos extraños).

Otro posible elemento por implementar serían los niveles móviles. En estos la cámara se movería por una sala mucho más grande que la sala convencional. En estos niveles el objetivo del jugador sería más luchar contra el movimiento de la cámara para evitar quedar fuera del encuadre que el de abrir la puerta. Los principales problemas para implementar estos niveles serian:

1. Se debería modificar la forma en la que los niveles se guardan para dar soporte a la nueva información necesaria para estos niveles (principalmente, tamaño y velocidad de movimiento).
2. Habría que cambiar la estructura de la sala base para dar soporte al tamaño variable.
3. Convendría implementar un sistema que gestionase la creación de bloques, el cual fuese creando los bloques según aparecen en el encuadre, y los elimine según se salgan de este, para optimizar el juego.

4.2.2 Enemigos

Los enemigos son elementos del juego que dificultan activamente el progreso al jugador, proveyendo al jugador de desafíos que superar[Bjo]. Los enemigos son comunes en este género de juegos (apareciendo incluso en títulos pioneros como Arkanoid (1986, Taito)) y permiten dar variedad e interactividad a los niveles del juego. El juego final incluye un único enemigo, el jefe final, aunque durante el desarrollo se barajó la opción de crear algunos enemigos, pero fue descartada para reducir el coste de su programación.

Los dos enemigos diseñados eran los siguientes:

- **El Trepador:** Se trataba de un enemigo con forma de gusano que reptaba a velocidad constante por la sala, pudiendo incluso trepar por el techo y paredes. Podía aturdir al personaje principal si este lo tocaba, por lo que el jugador debía esquivarlos mientras se movían por la sala.
- **El Replicador:** Este enemigo se colocaría en la puerta e imitaría el aspecto de los bloques. Cada cierto tiempo, este enemigo crearía una copia de si mismo en un espacio libre cercano, por lo que el jugador debería darse prisa o acabarían reparando"la pared. Estos enemigos podían ser destruidos si se les golpeaba con la pelota.

También sería necesario la adicción de jefes finales, para dar variedad a los desafíos finales. Durante el desarrollo se barajaron varios jefes finales distintos, pero fueron descartados en favor del jefe actual dado que solo se tenía tiempo para implementar

uno. Estos diseños podrían recuperarse, se tendría que tener en cuenta el coste de implementar un jefe, cuyo comportamiento es bastante más complejo.

4.2.3 Sistemas Complementarios

Fuera del diseño de niveles, el juego necesita de ciertos sistemas complementarios que sirvan para prolongar su vida útil.

El primero de estos sistemas sería una tabla de puntuaciones. Las tablas de puntuaciones son unas listas que almacenan la puntuación que han alcanzado los jugadores al terminar el juego con el propósito de que puedan ser comparadas. La implementación de estas tablas, junto con la de un sistema de puntuación (que podría estar basado en factores como el tiempo de juego, cantidad de bloques destruidos o número de niveles superados) aumentaría enormemente la vida útil del juego ya que alentaría al jugador a que volviera a jugar con el propósito de superar su propia puntuación o la de otro jugador. Idealmente, la tabla de puntuaciones sería online, compartida entre todos los jugadores, pero eso supondría un coste adicional debido a la infraestructura necesaria.

El segundo sistema que podría implementarse para aumentar la vida útil del juego es un sistema de generación procedimental de niveles. Este sistema podría producir una cantidad de niveles infinita, lo que solucionaría el problema de la corta duración y reduciría el tiempo de desarrollo dedicado a la generación manual de niveles. Además, basándose en este sistema podrían implementarse otros sistemas complementarios, como un sistema de “niveles diarios” en la que todos los jugadores jugarían a un mismo grupo de niveles (sistema presente en juegos como *The binding of Issac: Rebirth*(Nicalis, 2014) o *Leap Day*(Nitrome, 2016).

ANEXOS

Bibliografía

- [05] *Appel Human Interface Guidelines*. Apple Computer, Inc, 2005.
- [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.
- [BS12] Brenda Brathwaite y Ian Schreiber. *Breaking into the Game Industry: Advice for a Successful Career from Those Who Have Done It*. Boston: Course Technology, 2012.
- [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.
- [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>.
- [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.
- [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/JST.2014.2322503>.

BIBLIOGRAFÍA

- [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.
- [YT18] Georgios N. Yannakakis y Julian Togelius. *Artificial Intelligence and Games*. <http://gameaibook.org>. Springer, 2018.

Videojuegos

- [Cap91] Capcom. *Street Fighter II: The World Warrior*. 6 de feb. de 1991.
- [Chu86] Chunsoft. *Dragon Quest*. 27 de mayo de 1986.
- [Dou52] Alexander S. Douglas. *OXO*. 1952.
- [Eni16] Square Enix. *Final Fantasy XV*. 29 de nov. de 2016.
- [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>.
- [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.
- [Nin81] Nintendo. *Donkey Kong*. 9 de jul. de 1981.
- [Nin85] Nintendo. *Super Mario Bros*. 13 de sep. de 1985.
- [Nin86] Nintendo. *The Legend of Zelda*. 21 de feb. de 1986.
- [Sof93] id Software. *Doom*. 10 de dic. de 1993.
- [Sof94] Raven Software. *Heretic*. 23 de dic. de 1994.
- [Sof96] id Software. *Quake*. 22 de jun. de 1996.
- [Sof99] id Software. *Quake III Arena*. 2 de dic. de 1999.
- [Squ95] Square. *Chrono Trigger*. 11 de mar. de 1995.
- [Stu04] Bungie Studios. *Halo 2*. 9 de nov. de 2004.

VIDEOJUEGOS

- [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.

VIDEOJUEGOS

Este documento fue editado y tipografiado con L^AT_EX empleando la clase **esi-tfg** (versión 0.20180501) que se puede encontrar en:
https://bitbucket.org/arco_group/esi-tfg

[respeta esta atribución al autor]

