Análisis, diseño y desarrollo de un videojuego en 2D vista lateral

Universidad de Las Palmas de Gran Canaria

Escuela de Ingeniería Informática

Trabajo de Fin de Grado

Autora: Marina Ronda Marrero

Tutores: Agustín Trujillo Pino y Adrián Rivero Pérez

Las Palmas de Gran Canaria, Julio 2016

Contenido

1. Introducción 5
1.1. Objetivos y Motivación 5
1.2. Sobre la memoria 5
2. Historia y estado actual 7
2.1. Historia 7
2.2. Estado actual 13
3. Justificación de las competencias específicas cubiertas 15
4. Aportaciones 17
5. Elección de Herramientas 19
5.1. Elección del motor de juego 19
5.2. Elección del lenguaje de programación 21
6. Estudio y aprendizaje de Unity 23
6.1. Fundamentos básicos de Unity 23
6.1.1. Fundamentos del editor en 3D y 2D 23
6.1.2. Luces 29
6.1.3. Partículas 30
6.1.4. Scripting 30
6.1.5. Láseres 31
6.1.8. Mallas de navegación 32
6.1.9. Animaciones con Mecanim 32
6.1.10. Persistencia 33
6.1.11. Audio 33
6.1.12. Granadas Teletransportadoras 33
6.1.13. Explosiones y Explosiones cinemáticas 33
6.2. Juegos realizados como ejemplos 33
6.2.1. Ejemplo de juegos en 2D, catch game 33
6.2.2. Ejemplo de juego en 2D, tipo Angry Birds 34
6.2.3. Ejemplo de juego en 2D, runner infinito 34
6.2.4. Ejemplo de juego en 2D, juego Top-Down de naves 35
7. Sobre el diseño de prototipos 37
7.1. Factor de diversión 37
7.2. Estudios y realización de juegos de arcade clásicos de gran éxito 37
7.2.1. Pong 38
7.2.2. Space Invaders 41
7.2.3. Arkanoid 44

8. Ideas y diseño de prototipos 47
8.1. Propuestas y estudio 47
8.2. Primera selección 50
8.3. Desarrollo de prototipos 50
8.4. Elección final 57
8.5. Análisis de la propuesta 57
9. Desarrollo del juego final 61
9.1. Definición de los assets a utilizar 61
9.2. Casos de uso 61
9.3. Desarrollo del juego a partir del prototipo inicial 63
9.3.1. Personajes 64
9.3.2. Escenario 73
9.3.3. Pelota 74
9.3.4. Golpe de la pelota 75
9.3.5. Condiciones de victoria y derrota. 76
9.3.6. Pelota, 3 estados, 3 colores y daño según el color. 76
9.3.7. Barras de vida 77
9.3.8. Pruebas y feedback 78
9.3.9. Audio, menús y nueva vista de derrota o victoria 79
9.3.10. Menús 81
9.4. Desarrollo de la IA enemiga. 84
9.5. Diseño, arte y animación 89
9.5.1. Concepto e ilustraciones 89
9.5.2. Animación 96
9.5. Depuración de errores 97
10. Normativa y Legislación 99
10.1. Leyes que afectan a este proyecto 101
11. Conclusión y mejoras futuras 103
12. Manual de usuario 105
12.1. Contexto y objetivo 105
12.2. Ejecutando el juego 105
12.3. Menú de inicio 107
12.3.1. Opciones 107
12.3.2. Modo de juego 108
12.4. Dificultad 108
12.5. Controles 109

13. Glosario 111
14. Bibliografía 113

1. Introducción

1.1. Objetivos y Motivación

Se espera que el desarrollo de este proyecto favorezca la adquisición de competencias relacionadas con el diseño y elaboración de videojuegos.

El objetivo principal es el desarrollo de un videojuego completo y jugable teniendo en cuenta las limitaciones. Para ello tendremos que familiarizarnos con las distintas etapas de creación de videojuegos desde el diseño (ideas, análisis, prototipos) hasta el desarrollo del juego en sí. Consideramos importante que el resultado sea divertido, puesto que éste es un factor básico en cualquier videojuego que se precie.

Otro de los objetivos de este trabajo es diseñar e implementar una Inteligencia Artificial enemiga. De esta manera el oponente debería comportarse de forma similar a como lo haría un rival humano. Para lograr este objetivo también tendremos que estudiar el campo de las IAs.

Como consumidora de videojuegos el sector de los videojuegos resulta atractivo como posible salida profesional. Consideramos también que el proyecto es bastante completo y se necesitarán trabajar distintos aspectos, por lo que esperamos desarrollar diversas habilidades durante el proceso.

1.2. Sobre la memoria

Esta memoria consta de la documentación del Trabajo de Fin de grado *Análisis, diseño y desarrollo de un videojuego en 2D vista lateral*. La estructura de la misma cuenta con un apartado en el que se tratarán los objetivos y motivación del trabajo desarrollado. A continuación, hablaremos sobre la historia y el estado actual de la industria de los videojuegos. También se hablará sobre la justificación de las competencias adquiridas durante el desarrollo de este trabajo de fin de grado, así como las aportaciones realizadas.

En los siguientes apartados pasamos a hablar de la elección de las herramientas de trabajo y como se llevó a cabo el estudio de la herramienta Unity, con la que la alumna no había trabajado antes.

En el siguiente bloque comenzamos ya con el diseño de videojuegos en sí, las ideas propuestas y el estudio de las mismas, sin olvidarnos del desarrollo de prototipos. Tras la selección de idea definitiva se realizó un último análisis de la misma para poder enfocar correctamente su desarrollo.

Este desarrollo se describe en el siguiente apartado, hablamos también del desarrollo de la Inteligencia artificial, así como el arte, la animación y el diseño gráfico del juego. Finalmente, aunque dentro del mismo bloque hacemos un repaso de los problemas y bugs encontrados durante el desarrollo del juego.

Pasamos entonces a hablar de la normativa y legislación vigentes respecto a la informática y el ámbito de los videojuegos.

Finalmente, queda la conclusión y las mejoras futuras que se plantean respecto al videojuego, así como el manual de usuario.

2. Historia y estado actual

2.1. Historia

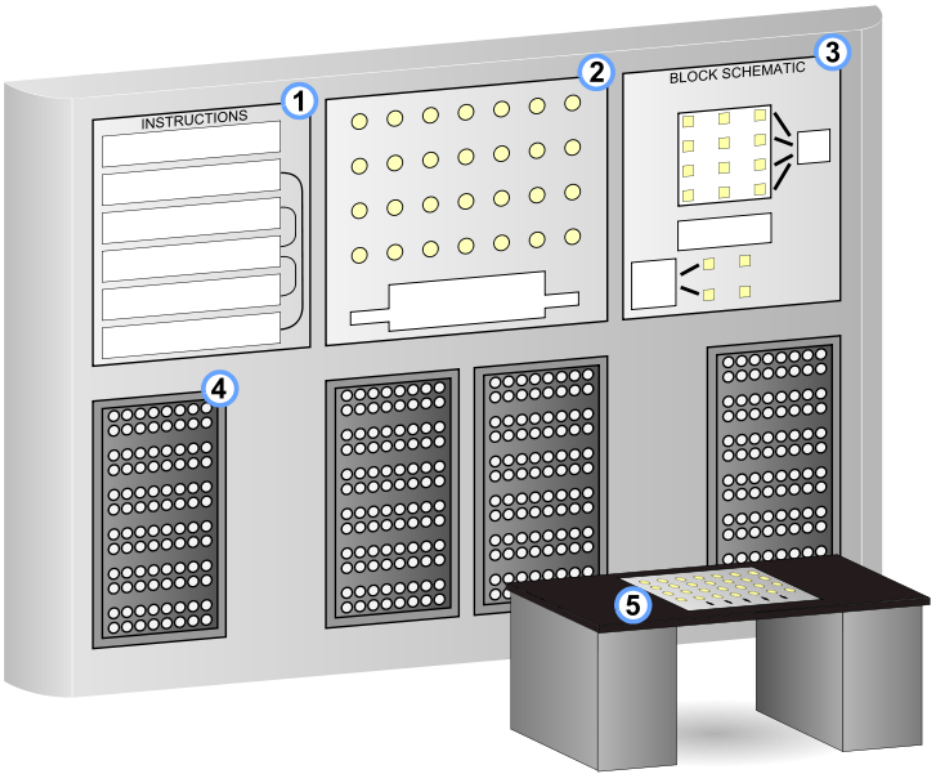
El término “videojuego” ha evolucionado con el paso de las décadas desde una definición puramente técnica a un concepto general para definir un nuevo tipo de entretenimiento interactivo.

Los primeros juegos de ordenador como tal surgieron en los 50 y había tres tipos principales, entrenamiento y programas de instrucción, programas de investigación en campos como la inteligencia artificial y programas de demostración concebidos para impresionar o entretener al público. Muchas veces eran descartados después de las exhibiciones y es imposible estar completamente seguros de quien desarrolló el primer juego de ordenador.

El primer juego conocido para ordenador fue una simulación de ajedrez desarrollada por Alan Turing y David Champernowne, claed Turochamp, que se completó en 1948 pero no llegó a implementarse en un ordenador.

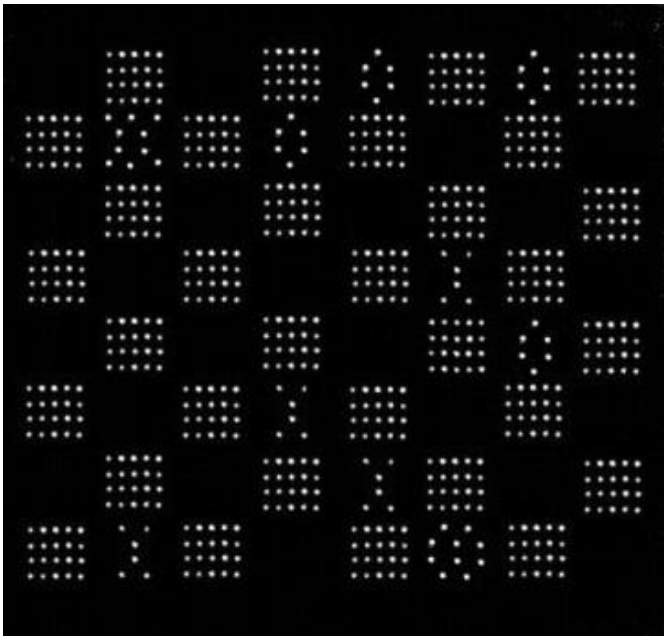
El primer juego electrónico del que se tiene constancia es el *Bertie the Brain*, creado en 1950 que consistía en un juego de Tres en raya, construido por Dr. Josef Kates para la *Canadian National Exhibition*. La máquina que se construyó para este juego era de cuatro metros de alto y los dibujos (cruces y rayas) se mostraban mediante luces de bombillas.

Casi un año después se presentó en 1951 el ordenador *Nimrod* en *el Festival of Britain*, creado por Ferranti. Usando un panel de luces estaba exclusivamente diseñado para jugar al juego Nim. Los movimientos se realizaban por los jugadores pulsando botones, esta acción se traducía correspondientemente en las bombillas.



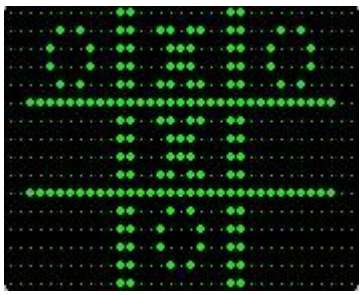
Dibujo del ordenador Nimrod

Por aquel entonces se empezaban a desarrollar juegos no visuales en distintos laboratorios de investigación, por ejemplo, Christopher Strachey desarrolló un simulador de juego de damas que inicialmente fracasó en su primera prueba en 1951, pero que se concluyó con éxito en 1952. Este es el primer juego conocido que fue creado para un ordenador de propósito general, en lugar de en una máquina exclusivamente construida para ello como fue el caso de *Bertie*.



Checkers de Christopher Strachey

El programa de Strachey inspiró a Arthur Samual a desarrollar su propio juego de damas en 1952 para IBM, sucesivas iteraciones lograron el desarrollo de una inteligencia artificial rudimentaria para 1955 y una versión se mostró en televisión en 1956. El OXO de Alexandre Douglas también incorporaba monitor.

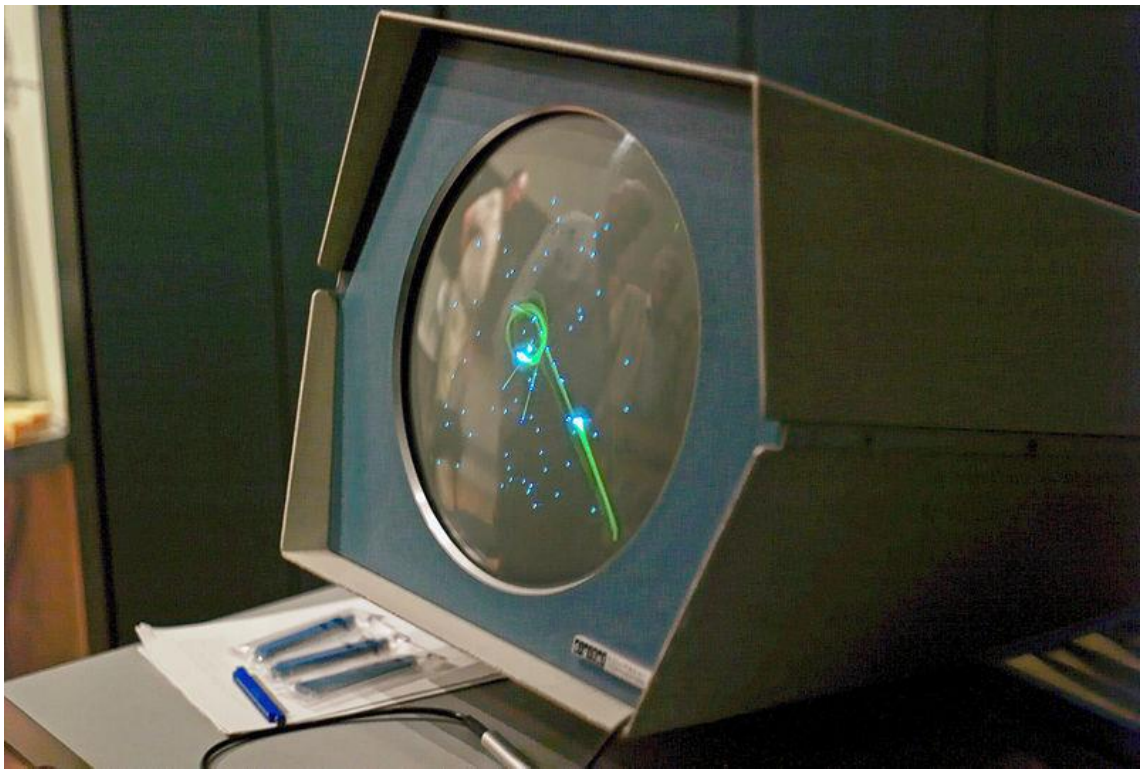


OXO, Alexander Douglas

A pesar de que ambos juegos tenían monitor, el primer juego que incorporaba gráficos que se actualizaban en tiempo real fue un juego de billar programado por Willian Brown y Ted Lewis, diseñado específicamente para una demostración del ordenador MIDSAC en la Universidad de Michigan en 1954.

Quizás el primer juego creado únicamente para entretenimiento en lugar de para demostrar la potencia de una tecnología, entrenar personal o ayudar en una investigación fue el *Tennis for Two,* diseñado por William Higinbotham y construido por Robert Dvorak en 1958. El juego se desarrolló en un ordenador analógico con gráficos en un osciloscopio.

En 1961 se creó el juego *Spacewar!* desarrollado por un conjunto de estudiantes del MIT. El juego, considerado el primer juego de disparos, mostraba un duelo entre dos naves espaciales, cada una controlada por un jugador. El juego se volvió muy popular entre los estudiantes del MIT y se extendió por la Costa Oeste.



Spacewar!

En 1967 Ralph Bear y un asociado crearon el primer juego en utilizar pantalla con un scan de mapa de bits, que se reproducía directamente por medio de la modificación de una señal de vídeo. Fue la primera videoconsola hogareña.

Magnavox Odyssey.

Videojuegos de arcade de primera generación.

En 1972 después de ver una demostración del Magnavox Odyssey previo a su salida, Nolan Bushnel encargó al recién contratado Al Alcorn a realizar su propia versión del tenis de mesa de la Odyssey como práctica. El juego resultó ser tan divertido que Atari lo sacó al mercado como Pong. Pong llegó al mercado en grandes cantidades en Marzo de 1973. El éxito no desbancó a máquinas tradicionales como el pinball pero sirvió de fundación para futuras máquinas de arcade de videojuego.

En 1974 debido a la saturación del mercado de juegos de "pelota y palas", las grandes compañías se pasaron a otros géneros, principalmente juegos de carreras, de combates de 1 vs 1 y de disparos. Entre estos encontramos títulos como *Gran Tank 10*, *Wheels*, *Gun Fight* o *Sea Wolf*.

Primera generación de consolas hogreñas

Magnavox sacó en 1975 dos nuevos sistemas que solo contaban con juegos de pelota y palas, el Odyssey 100 y el Odyssey 200. Atari por su parte, entró en el mercado ese mismo año con el sistema Home Pong, diseñado por Harold Lee.

Después de 1977 el mercado dedicado de las consolas se colapsó. Una nueva ola de sistemas programables golpeó el mercado comenzando con Fairchild Channel F en 1976, que ofrecía la posibilidad de comprar y jugar una amplia variedad de juegos almacenados en cartuchos.

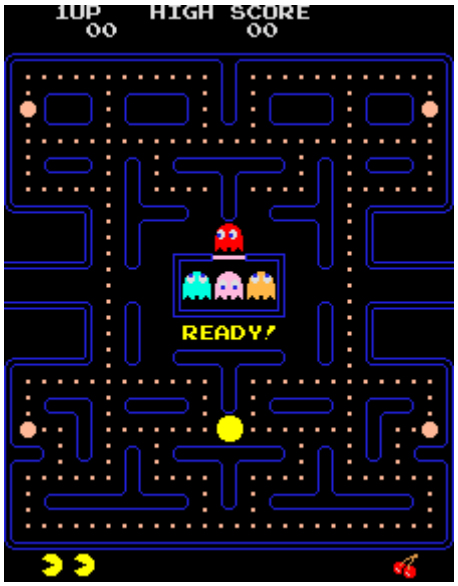
Fairchild Channel F

También en 1976 Mattel sacó *Auto Race*, la primera consola portátil de la historia, pronto sacaría también *Football*, con mucho más éxito.

En 1978 Magnavox introdujo *Odyssey II* para hacer competencia a la *VCS 2600* de Atari. Mientras tanto en Japón, Toshihiro Nishikado adoptó la nueva tecnología microprocesador e influenciado por *Speed Race* creó *Space Invaders*. El juego obtuvo un éxito e grandes dimensiones y se convirtió a todos los formatos importantes de la época, dando lugar a numerosas continuaciones y clones. Este juego situó a la industria japonesa en el mercado mundial de videojuegos y dio paso a la conocida *como Edad dorada de los videojuegos*.

La edad de oro de los videojuegos

En el verano de 1982 la fiebre de los videojuegos aumentó considerablemente. La industria incrementaba sus beneficios en un 5% mensuales desde *Space Invaders*, contaban con un numeroso público y había máquinas recreativas en todas partes.



Los gráficos vectoriales se habían utilizado con cierta asiduidad desde varios años antes pero el primer videojuego en usar la tecnología fue *Space Wars* en 1977, una versión del *Spacewar!* Atari por su parte lanzó en 1979 *Lunar Lander* y *Asteroids*, el que obtuvo un éxito inmediato y es otro de los grandes clásicos.

El color llegó a los videojuegos con Galaxian en 1979, de la mano de Namco, un juego que resultó muy popular en su época y marcaba la evolución del género. Combinando gráficos vectoriales con color tenemos Defender de 1981, el primer matamarcianos horizontal y uno de los títulos más rentables de la historia. Luego encontramos Frogger con avances gráficos superiores, pero ninguno fue rival para Pac-Man de 1980, por Toru Iwatani, considerado uno de los videojuegos más populares de todos los tiempos.

Toru Iwatani pretendía con Toru acercar a las mujeres a los videojuegos, y no solo tuvo un rotundo éxito entre hombres y mujeres, también inauguró el mercado de merchandising de videojuegos.

Entre otros juegos populares de la época se encontraban Donkey Kong y Q*bert

Con el éxito de los ordenadores personales a finales de 1970, una serie de rivales surgió a principio de 1980. Los videojuegos dominaban las librerías de software de los ordenadores personales.

Videojuegos online

Los *Bulletin Board Systems* se usaban ocasionalmente para jugar online, algunos ofrecían acceso a varios juegos desde aventuras de texto a juegos de apuestas como el blackjack. El primer juego de red escrito para ordenador personal y de uso comercial fue el *Snipes*. Este juego junto a *Maze War* y *Spasim* están considerados los precursores de los juegos multijugador como *Doom* en 1993.

En 1979 salió el primer sistema portátil que usaba cartuchos intercambiables, *Microvision*. Nintendo sacó en 1980 su línea *Game & Watch*, también de consolas portátiles.

En 1983 se produjo la caída de la industria de videojuegos, así como se arruinaron varias compañías norteamericanas de consolas y videojuegos. Las causas de la caída fueron: un mercado saturado por juegos de poca calidad realizado por terceras partes, fracaso de importantes títulos de Atari y el surgimiento de los ordenadores como nuevas y mejores plataformas de juego frente a las consolas.

En 1985 Nintendo asaltó el mercado estadounidense, como punta de lanza estaba el título *Super Mario Bros*. En 1987 el mercado de las videoconsolas había regresado. En 1987 salió la NES. En 1985 Sega sacaba su Master System.

George Lucas fundó en 1982 Lucasfilm Games, dedicada al desarrollo de videojuegos. La compañía realizaba diversos juegos, sacaron videoaventuras como The Secret of Monkey Island (1190) que aprovechaban las características de los ordenadores de la época.

En 1190 la consola Sega Mega Drive controlaba el mercado, lanzada en 1988 disponía de títulos como *Sonic the Hedgehog*. En respuesta Nintendo sacó en 1991 la Super Nintendo que reconquistó el mercado.

En 1989 la aparición de la Game Boy supuso una revolución en su campo, posicionándose como líder en las consolas portátiles.

Con la evolución de la tecnología y el mercado aparecerían nuevos géneros, como el de estrategia por turnos, con Simcity en 1989 y luego Civilization en 1991, este último sentaba las bases definitivas del género. A este le han sucedido títulos como Warcraft (1994), Age of Empires (1997) y Commandos (1998).

En la década de los 90 se volvió sobre las teorías de Sutherland sobre realidad virtual. Surgieron así los primeros juegos en 3D como Tailgunner (1979) o Battlezone (1980) que usaban gráficos lineales dando ilusión de profundidad. Los simuladores de vuelo lideraban el área, pero pronto se extendieron a otro tipo de juegos. En 1992 surgió Alone in the Dark, un título que inauguró el género de survival horror.

En 1991 surge también Catacombs 3D, que iniciaba el género de shooters en primera persona. Seguido por Wolfenstein 3D en 1992 con un mejor motor gráfico y mejoras de jugabilidad y que cosechó un gran éxito.

En 1993 Doom sacudió el mercado, no solo destacaba por sus gráficos sino por su inusitada violencia. Además, fue el primer videojuego en permitir modding (modificación de su diseño y niveles por parte de sus jugadores). También popularizó el juego en línea.

Con las distintas empresas de videojuego compitiendo por el mercado de las videoconsolas, Sony que no tenía mucha fuerza en el mercado, fundó una división para desarrollar la Playstation basada en CD-ROM frente a las consolas de cartuchos y que tuvo un éxito rotundo en el mercado.

En 1996 Nintendo sacaba su nueva consola la Nintendo 64 pero Playstation seguía siendo la líder del mercado.

En el siglo XXI la industria de los videojuegos es multimillonaria y de unas dimensiones difícilmente imaginables algunos años antes. La pérdida de peso de las máquinas recreativas, la

popularización de las consolas (Playstation 2, X-Box, Wii, etc.) y el rápido desarrollo de la telefonía móvil que permite jugar a videojuegos en el móvil han marcado los últimos años en el sector.

2.2. Estado actual

El mundo de los videojuegos es un fenómeno global, prácticamente una subcultura en sí misma. Los videojuegos han aumentado exponencialmente en popularidad con el paso del tiempo y tienen una influencia sustancial en la cultura popular. La cultura ha evolucionado recientemente además de la mano de internet y los juegos para móviles.

No nos olvidamos de los juegos multijugador y online que permiten una mayor sociabilización entre usuarios. Jugar a videojuegos puede hacerse como entretenimiento o incluso de forma competitiva, la nueva tendencia “deportes electrónicos” se ha vuelto más y más aceptada en nuestra sociedad. Realizándose incluso competiciones internacionales (como por ejemplo de “League of Legends” o “Pokemon”). Hoy en día los videojuegos pueden encontrarse en redes sociales, televisión, libros, películas o incluso música.

El mundo de los videojuegos, aunque predominantemente masculino es sus inicios, es cada vez más popular entre las mujeres, desde 2010 las mujeres forman casi el 50% de jugadores. No es así en el campo de desarrollo de videojuegos, donde la cantidad de mujeres pasa a ser bastante menor, más aún si nos limitamos a mujeres programadoras.

La media de edad de jugadores es 20 y tantos. Aunque la media aumenta con el paso del tiempo a medida que los jugadores se hacen mayores.

El campo de los videojuegos ha crecido mucho en los últimos años a pesar de ser un campo relativamente reciente. Es un sector competitivo, pero con la popularización de motores y entornos de open source se ha permitido el acceso al sector a pequeños equipos que producen videojuegos conocidos como indies (en algunos casos producidos incluso por una única persona), alejados un poco de la idea tradicional, permitiendo mayor variedad de ideas y originalidad.

Nos encontramos por tanto ante un campo amplio con numerosas posibilidades.

3. Justificación de las competencias específicas cubiertas

Con el desarrollo de este trabajo de fin de grado cumplimentamos los apartados CII01, CII02, CII018 y TFG01 del proyecto docente, los cuales detallamos a continuación.

CII01 - Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente. Esta competencia ha quedado cubierta durante las fases de análisis, diseño y desarrollo del proyecto. Se elaborará sobre estas fases en los apartados Análisis y diseño de propuestas, *Ideas y diseño de prototipos y Desarrollo del juego final*, que encontramos en este documento. Además, se han tenido que tomar decisiones sobre distintos softwares a utilizar, de este tema se hablará en los apartados *Elección del motor de juego* y *Elección del lenguaje de programación*. Se puede afirmar que se han respetado los principios éticos y la legislación y normativa vigente durante el desarrollo de este proyecto.

CII02 - Capacidad para planificar, concebir, desplegar y dirigir proyectos, servicios y sistemas informáticos en todos los ámbitos, liderando su puesta en marcha y su mejora continua y valorando su impacto económico y social. Para la realización de este proyecto se ha seguido una planificación en la que se asignaban tareas semanales que se evaluaban también semanalmente. Los resultados se probaron con usuarios dispuestos y se tuvo en cuenta el feedback para la mejora del prototipo. En los apartados de *Aportaciones* y en el de *Conclusión y mejoras futuras* se recoge el impacto social del producto desarrollado.

CII018 - Conocimiento de la normativa y la regulación de la informática en los ámbitos nacional, europeo e internacional. En el apartado *Normativa y legislación*, exploraremos esta competencia.

TFG01 - Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario, consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería en Informática de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas. Está competencia quedará cubierta tras la defensa del TFG.

4. Aportaciones

La realización de este proyecto ha favorecido la adquisición de nuevas competencias, así como el afianzamiento de otras, adquiridas previamente durante el grado. Por ejemplo, se han a cabo tareas de planificación, análisis, diseño y elaboración de documentación relativa al proyecto.

Adquisición de nuevos conocimientos relacionados con el diseño y desarrollo de videojuegos.

También se han adquirido y reforzado competencias técnicas, lenguaje de programación, el uso de máquinas de estado para desarrollo de la IA, etc.

Aprendizaje y manejo de una herramienta utilizada para desarrollo de videojuegos.

A nivel general en el campo social, los videojuegos tienen potencial para ser una herramienta cultural y educativa, además del componente comercial. Es una de las industrias más potentes en cuanto entretenimiento. Aunque empezó siendo un mercado más sesgado (y hoy en día en muchos casos se sigue diseñando videojuegos principalmente para ese sesgo demográfico), la realidad es que el consumo de videojuegos no está limitado por edades o género. Además, la diversidad de los mismos promovida por la apertura del mercado gracias a la popularización de herramientas de open-source, ha permitido a la industria acceder a un mayor número de consumidores.

La industria de los videojuegos mueve grandes cantidades de dinero y genera muchísimos puestos de trabajo. En 2009 la industria de videojuegos generaba en España más dinero que la música y el cine juntos.

En España, el mercado de videojuegos ha ido evolucionando a medida que evolucionaba el mercado internacional. Ciudades como Madrid y, especialmente, Barcelona, representan los puntos de máximo desarrollo de una industria de ocio y entretenimiento aún emergente en nuestro país, pero con importantes perspectivas de futuro.

Se prevé que la industria de los videojuegos en España alcance los 723,6 millones de euros de facturación y que doble el número de empleados hasta 2017. En concreto, generará trabajo para 5.271 personas, más del doble de las 2.630 con los que cerró 2013 y que supone una tasa de crecimiento anual del 21%.

Los videojuegos tienen distintas aplicaciones a nivel social. En el campo de la salud mental, por ejemplo, encontramos *CIPOActivity* que potencia las capacidades cognitivas y motoras. Este juego utiliza el dispositivo Kinect, que permite al usuario jugar sólo con el movimiento del cuerpo. El juego *SinaSprite*, está dirigido a mujeres de entre 25 a 50 con altos niveles de estrés, ansiedad o depresión. Y por último comentar el juego *Flowy*, que fue diseñado para ayudar a lidiar con los ataques de pánico.

En el campo de la neurociencia, por otra parte, un estudio de la Universidad Central de Michigan ha descubierto que los pacientes con lesión cerebral traumática pueden mejorar su coordinación mediante videojuegos. Así mismo en la Universidad de Sevilla han desarrollado un videojuego para las personas con parálisis cerebral.

También en la medicina, mediante el uso de simuladores o para pacientes de determinadas enfermedades. O incluso para terapia física, gracias a consolas como la Wii o juegos de baile y coordinación.

Por supuesto los juegos también tienen aplicaciones en el campo de la educación. Uno de los ejemplos es Minecraft que se usa para conocimientos abstractos y trabajo en equipo entre otras cosas. O el *Portal*, que se usa para enseñar física o matemáticas. Es tanto el éxito que tuvo este juego en el ámbito educativo que *Valve*, creador del juego, decidió hacer un "Steam para Escuelas". También en el ámbito de la música el juego *Perfect Pitch* permite a los estudiantes formar sus propias orquestas.

Podemos concluir que las posibles aplicaciones de los videojuegos son muy variadas.

5. Elección de Herramientas

Para el desarrollo de este trabajo se han contemplado distintos motores de juegos, aunque finalmente nos hayamos decantado por Unity.

5.1. Elección del motor de juego

Motor de Videojuegos

Un motor de juego es un software diseñado para la creación y desarrollo de videojuegos. Se usan para crear juegos para consolas, móviles y PC. Estos motores suelen proveer de un conjunto de funcionalidades clave, normalmente un motor de rendering para gráficos 2D o 3D, un motor de físicas o detector de colisiones, sonido, scripting, animación, gestión de memoria, entre otros.

El proceso del desarrollo de videojuegos se suele economizar, en gran parte, reutilizando o adaptando el mismo motor gráfico para crear distintos juegos.

Unreal Ungine 4

Unreal Engine 4 es el motor lanzado por Epic Games y el sucesor de UDK. UE4 tiene unas increíbles capacidades gráficas, incluyendo dinámicas de luz avanzadas y un sistema nuevo de partículas que puede manejar hasta un millón de partículas en una escena a la vez.

UE4 tiene una curva de aprendizaje con bastante pendiente. Con esta nueva versión se ha dejado atrás UnrealScript como lenguaje para usar C#.

Un detalle importante a tener en cuenta es que UE4 solo puede desarrollar para PC, Mac, iOS, Android, Xbox One y Play Station 4. Para las consolas no obstante es necesaria una licencia aparte y un SDK para cada una de ellas. Así pues, las plataformas de móvil y PC serían la opción más viable para nuevos desarrolladores.

UE4 es desde hace casi un año completamente gratis sin necesidad de subscripción. Esto lo hace extremadamente accesible a cualquiera que quiera comenzara desarrollar juegos, en caso de beneficios se aplica un 5% de derechos si el juego gana 3000$ por trimestre.

Unity3D 5

Unity es un motor de juego para distintas plataformas desarrollado por *Unity Technologies* y usado para desarrollar juegos para ordenador, consolas, teléfonos móviles y páginas web. Inicialmente se anunció solo para OS X, pero desde entonces se ha extendido para cubrir 21 plataformas. Es el software de desarrollo software (SDK) por defecto para la Wii U.

Unity ofrece una gran variedad de características y una interfaz relativamente fácil de aprender a manejar. Destaca por integrar un gran número de plataformas. Unity soporta assets de los grandes softwares de 3D como Maya, Blender o 3ds Max. Además, se le ha añadido soporte para 2D nativo, soporta sprites y físicas 2D, convirtiendo Unity en un gran motor para el desarrollo de juegos 2D. Los lenguajes disponibles en Unity para sus scripts son Javascript, C++ y Boo.

Aunque Unity soporta integración de casi cualquier aplicación 3D, sufre no obstante en las limitadas capacidades para editar dentro del propio motor. Los assets tendrán que ser creados de forma externa, en contrapartida Unity tiene una gran librería de assets que pueden ser descargados, siendo algunos de pago y otros gratis (el precio lo determina el autor).

Unity cuenta con tres tipos de licencia, la versión Pro, que cuesta 125$ mensuales y viene con varias características adicionales como una capa de servicios "Pro", también está la versión Plus que cuesta 35$ mensuales y pone el límite en ingresos anuales en 200k$, además ofrece una capa de servicios "Plus". Finalmente tenemos la versión "Personal" que es la básica de Unity y es gratuita, el límite de ingresos anuales es de 100k$.

Unity es un motor normalmente asociado con juegos móviles, no obstante, con la salida de Unity 5 y su nuevo sistema de render ha habido un enorme aumento en las capacidades del motor, incluyendo shading basado en físicas, iluminación global en tiempo real entre otros.

CryENGINE 5

CryENGINE es un motor poderoso diseñado para el desarrollo de videojuegos de la compañía Crytek que fue introducido en el primer juego de la saga Far Cry. Está diseñado para ser usado en PC y consolas, incluyendo la PlayStation 4 y la Xbox One. Las capacidades gráficas de este motor superan las de Unity pero igualan a las del Unreal Engine 4. Tienen físicas realistas y un sistema de animación avanzado entre otras características.

Aunque CryEngine es un motor muy potente, también tiene una curva de aprendizaje complicada sobre todo a la hora de usar el motor de forma productiva. Puede ser incluso más difícil si no se tienen conocimientos de otros motores de juego.

CryENGINE 5 está disponible según el modelo *Pay what you want* mediante el cual pagas lo que quieras para obtener el motor (0 euros incluido). Además, está libre de derechos.

Conclusiones

Resumiendo, hemos determinado que Unity es un buen motor para juegos móviles, 2D y 3D. Unreal permite crear juegos con gráficos fotorrealistas o juegos side-scroller simples en 2D. CryENGINE tiene también unas capacidades gráficas muy potentes y el hecho de poder pagar lo que se pueda por el motor completo resulta muy atractivo.

Finalmente, hemos optado por realizar el proyecto en Unity por distintos factores. Principalmente porque consideramos que es el que tiene una menor curva de aprendizaje, lo cual es importante al disponer de un tiempo limitado para desarrollar el proyecto y aprender previamente el uso de la herramienta. Además, es el motor que nos permite más flexibilidad para realizar un videojuego en 2D. También dispone de numerosa documentación online. Finalmente, Unity dispone de gran información y una comunidad importante, además de gran cantidad de assets a disponibilidad de los usuarios. Al poderse obtener los tres motores de forma gratuita el factor económico no ha influenciado la decisión.

5.2. Elección del lenguaje de programación

Unity nos permite trabajar con tres lenguajes, JavaScript, C# y Boo. Se ha optado por el uso de C# debido a varios motivos, uno de ellos es que es el lenguaje de Unity sobre el que más documentación se puede encontrar, la mayoría de tutoriales y referencias se encuentran en este lenguaje. JavaScript es además un lenguaje más limitado en comparación, recomendado para aquellos que comienzan a programar. Por otra parte, respecto a Boo, no se ha encontrado apenas documentación en este lenguaje para Unity, además en un lenguaje más reciente y tiene una sintaxis con la que habría que familiarizarse antes de comenzar a programar en este lenguaje. Lo cual alargaría aún más el tiempo que habría que emplear en el proyecto.

6. Estudio y aprendizaje de Unity

Las primeras semanas se dedicaron al estudio del motor de juego puesto que no se había utilizado previamente. Procuraremos explicar los fundamentos básicos de Unity para poder entender la terminología usada en esta memoria.

La página de Unity cuenta con mucho material de aprendizaje, entre ellos numerosos cursos con vídeos demostrativos en inglés. Los vídeos estudiados se adjuntan como anexo en bibliografía al final de este trabajo.

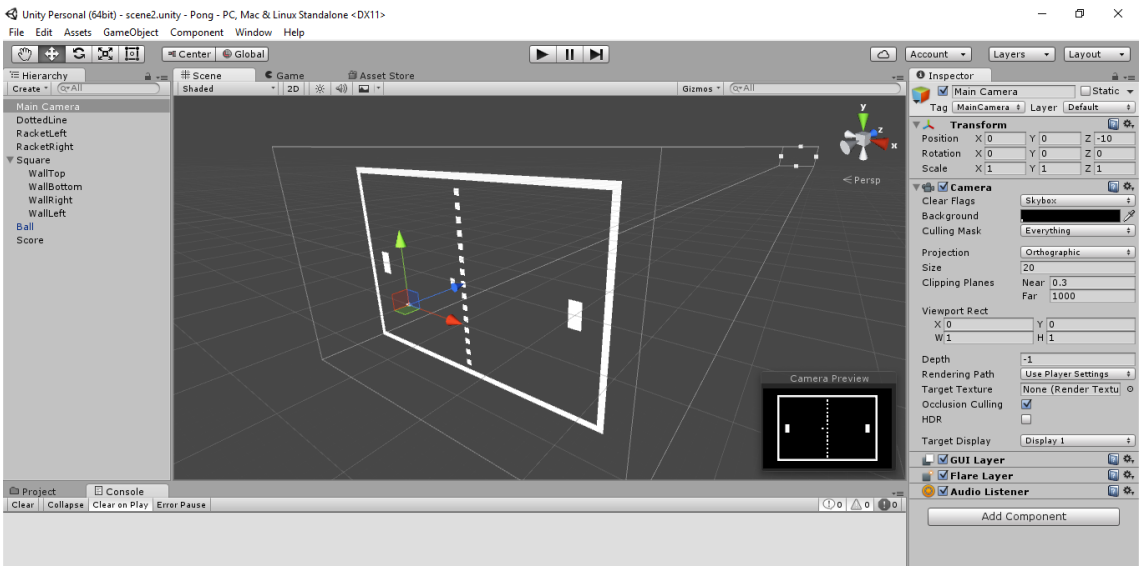
Durante el desarrollo de este TFG se comenzó por estudiar los conceptos básicos respecto a la interfaz y el editor de Unity.

6.1. Fundamentos básicos de Unity

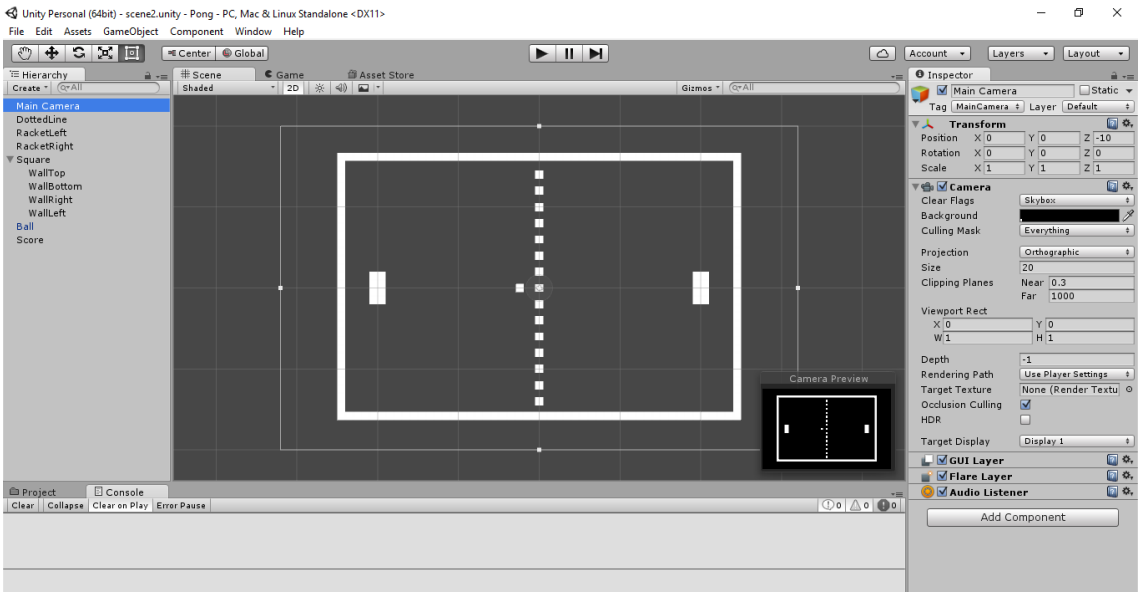
6.1.1. Fundamentos del editor en 3D y 2D

Inicialmente aún no se habían concretado las características del juego a realizar se estudió la herramienta en todos los sentidos. En el primero de los cursos se estudió la interfaz, controles, vistas y layouts entre otras cosas. En el segundo se estudió el editor desde el punto de vista 2D usando el framework de Unity.

Por ejemplo, a continuación, vemos un juego concebido en 2D con cámara ortográfica (de la que hablaremos en el apartado de cámaras) que está en modo 3D. Podemos ver que disponemos de un eje de coordenadas cartesianas en la pantalla de la escena.



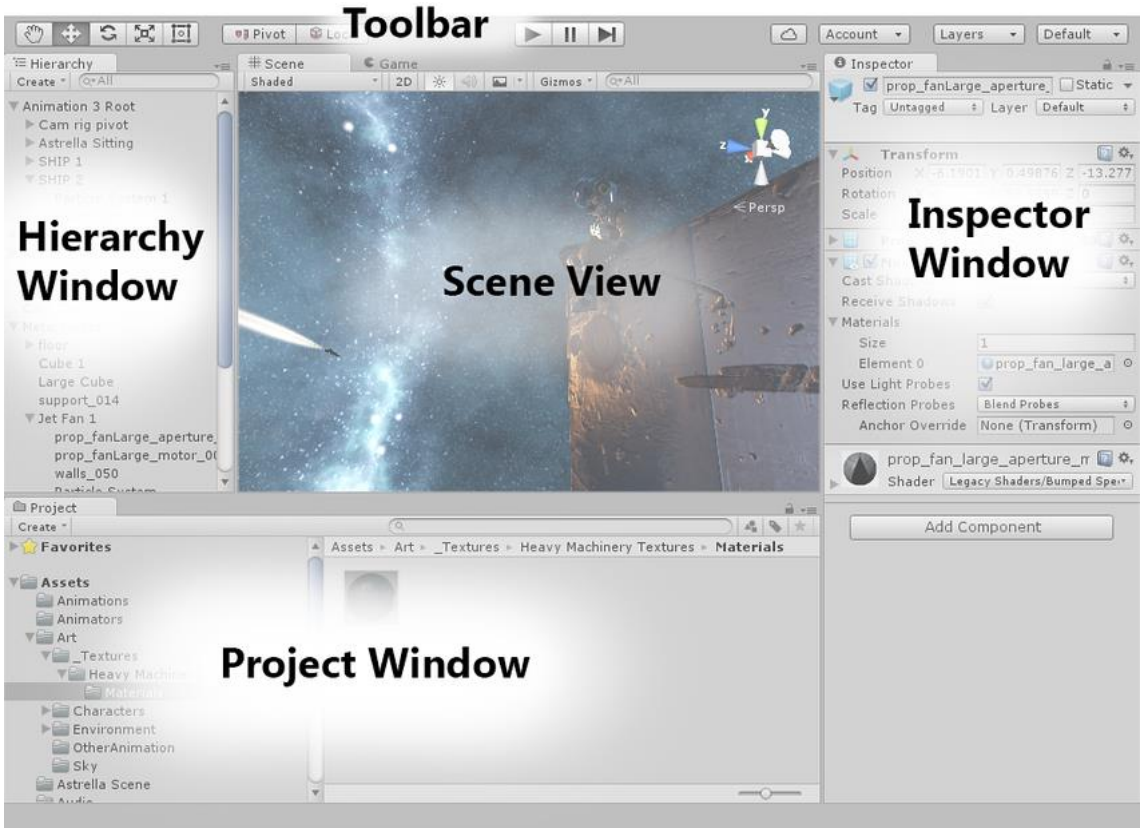
Por otra parte, tenemos el modo 2D del mismo juego, las coordenadas desaparecen y podemos apreciar la diferencia a la hora de visualizar la escena.



Interfaz

La ventana principal del editor está compuesta por ventanas en pestañas, que pueden ser reorganizadas, agrupadas, separadas y ancladas. Esto significa que la apariencia del editor puede ser distinta de un proyecto a otro y también entre distintos desarrolladores dependiendo de la preferencia personal y el tipo de trabajo que se está haciendo.

La distribución por defecto permite acceso práctico a las ventanas más comunes. En la imagen posterior podemos ver la distribución por defecto.

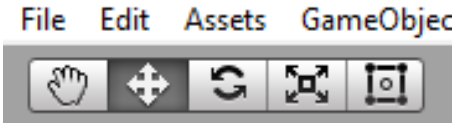


Captura de la interfaz obtenida del manual de Unity.

A continuación, detallamos las partes más relevantes:

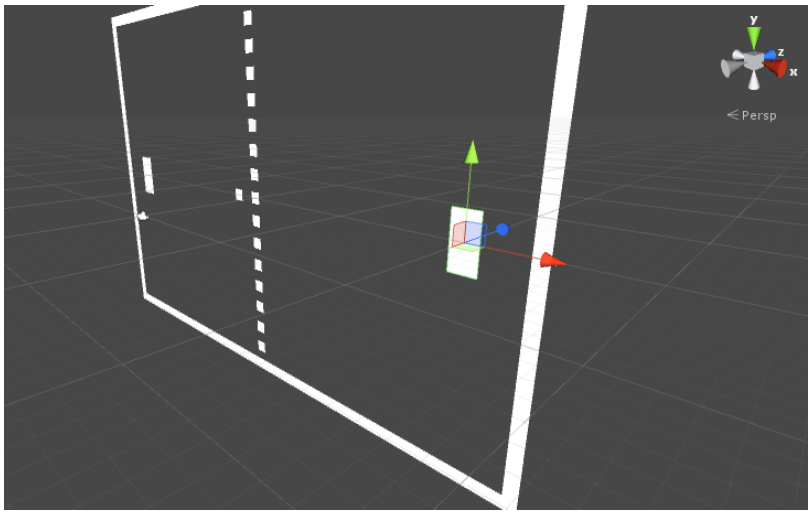
- **La vista de escena**. En esta ventana podemos construir la escena y navegar por ella. Se puede mostrar en perspectiva 2D o 3D.
- **La vista del juego**. En esta imagen relegada a segundo plano, contiene una vista previa del juego, desde esta vista se puede reproducir el juego y jugarlo.
- **La ventana del proyecto**. Contiene la librería de assets del proyecto. Se pueden importar objetos 3D, texturas, imágenes…, y también generar otros en la propia plataforma Unity, como en el caso de los prefabs. Los assets importados permanecerán en el proyecto al que fueron añadidos, aunque se cierre Unity.
- **La ventana de jerarquía.** Es una vista jerárquica que representa todos los elementos de la escena actual.
- **El inspector**. Permite ver y editar un objeto seleccionado actualmente. También contiene la configuración para ciertas herramientas como la herramienta de terrenos si el terreno está seleccionado. Los campos del inspector varían según el tipo de objeto seleccionado.
- **La barra de herramientas.** Provee acceso a las características principales del motor. Por ejemplo, contiene herramientas básicas para manipular la vista de la escena y los objetos en ella (rotar, desplazar). Además, también provee acceso a tu cuenta Unity y un editor del menú entre otras cosas. La barra de herramientas no se puede editar.

Desplazamiento en la escena y mover objetos

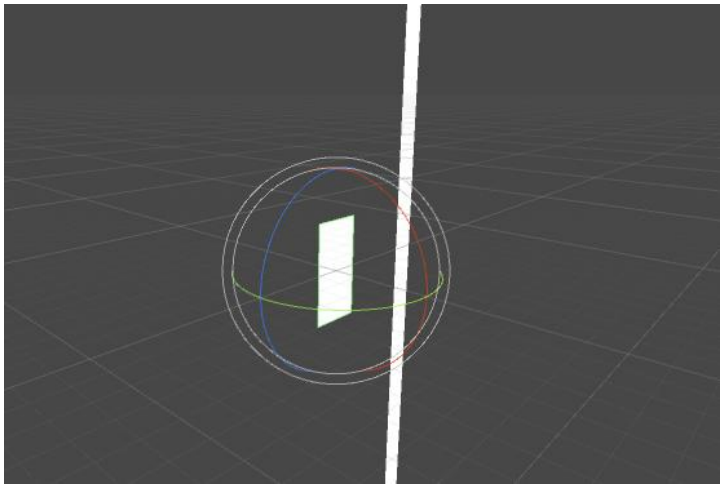


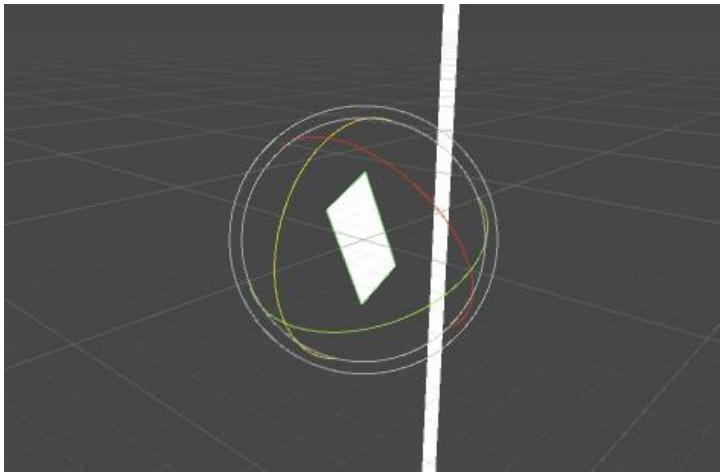
Los iconos que están debajo del menú principal permiten interactuar con los objetos a excepción de la mano que nos desplaza por la escena. Funcionan de forma similar en 3D y 2D.

La cruz pinta los ejes x,y,z en el objeto que hemos seleccionado, de forma que podemos desplazarlo en cualquiera de esos ejes haciendo uso del ratón.

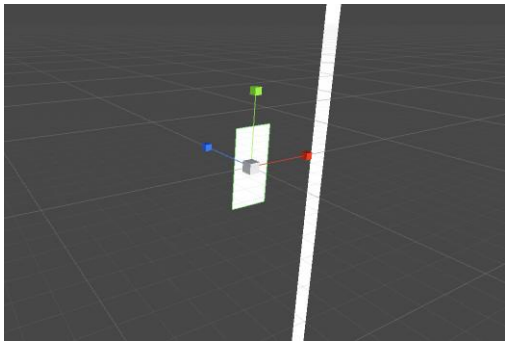


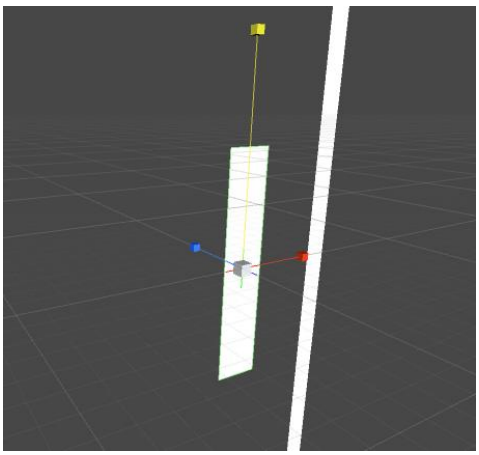
El siguiente botón nos permite rotar un objeto por cualquiera de sus ejes.



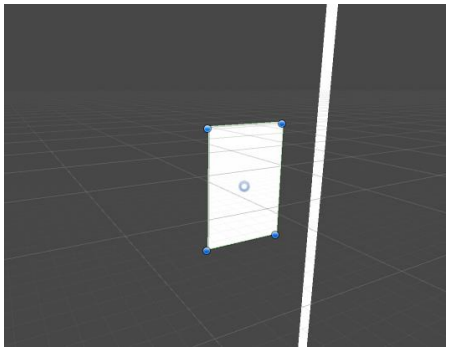


El botón a continuación con forma de caja con flechas nos permite cambiar la proporción de un objeto dependiendo del eje que modifiquemos.





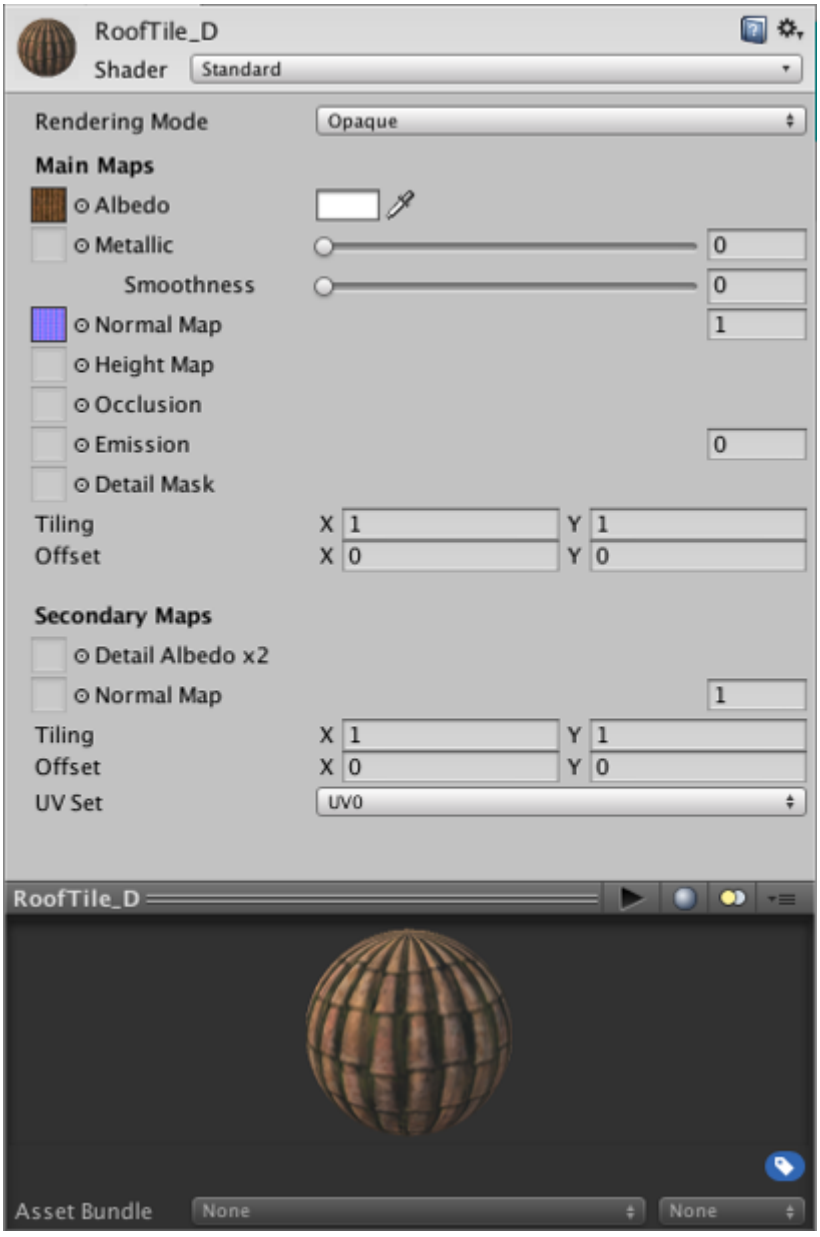
El último botón nos permite cambiar el tamaño de un objeto utilizando sus vértices, además de moverlo con el ratón.



Para desplazarnos por la escena en Unity tenemos varias opciones. La que se usó principalmente durante el desarrollo del trabajo consistía en click derecho más los botones WSDA (como en cualquier juego de primera persona), y el Mayús izquierda para esprintar. Manteniendo el click derecho pulsado también nos permitía cambiar la orientación de como vemos la escena imitando de nuevo a los juegos de primera persona. Consideramos que ésta era la forma más intuitiva de moverse por la escena. No obstante, existen otros métodos de desplazamiento como el uso de las flechas de dirección del teclado y la utilización de botones de Unity.

Modelos y materiales

Estudiamos los modelos, materiales, shaders y texturas en Unity. Los ficheros de modelo contienen información como el propio modelo 3D de un personaje u objeto, o la animación del mismo. El material por otra parte, usado conjuntamente con otros componentes de renderizado, cumplen una función esencial a la hora de definir como se muestra un objeto.



Una ventana inspector de un Material

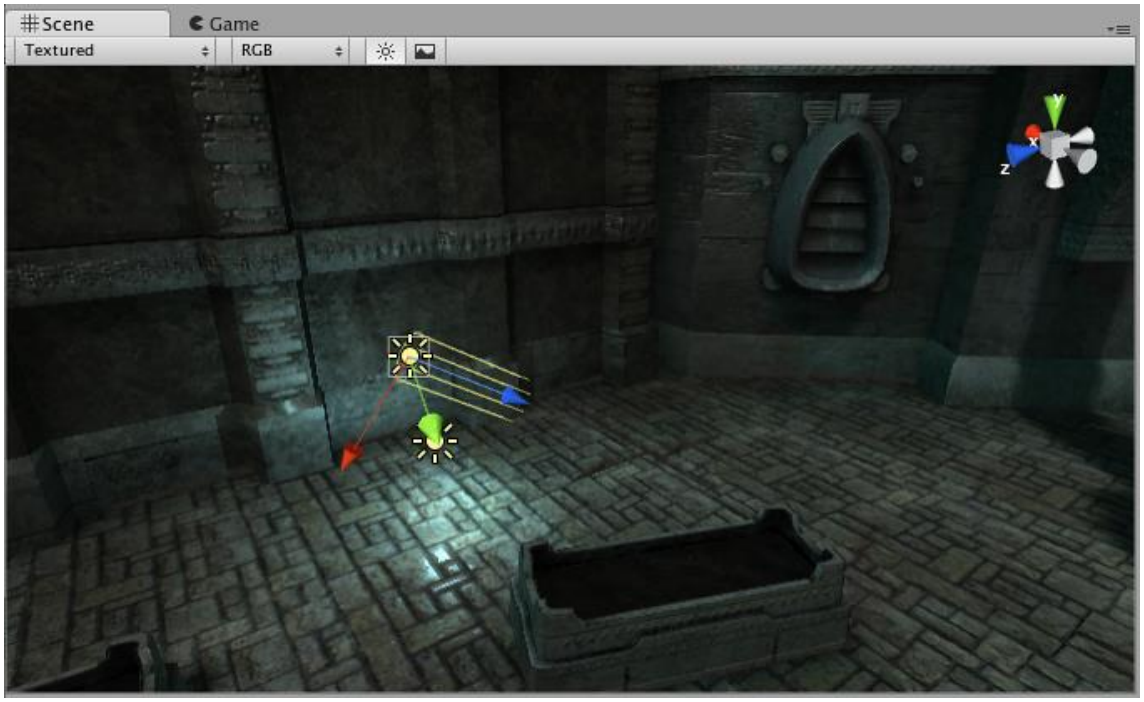
Cámaras

Las cámaras son elementos que capturan y muestran el mundo al jugador. Mediante la personalización y manipulación de las cámaras la presentación del juego puede ser verdaderamente única. La vista de cámara puede ser en perspectiva u ortográfica. En perspectiva la cámara renderizará los objetos con su perspectiva intacta. En modo ortográfica la cámara renderizará los objetos de forma uniforme sin sentido de perspectiva alguno.

Se permite un número ilimitado de cámaras por escena y pueden programarse para renderizar en cualquier orden, en cualquier lugar de la pantalla o solo en ciertas zonas de la pantalla. La vista de la cámara incluso puede ponerse en una textura y se puede aplicar a un objeto.

6.1.2. Luces

La luz es una parte esencial en cada escena. La luz define el color y el tono de un entorno 3D. En la mayoría de escenas probablemente se trabajarán con más de un foco de luz. Hay diferentes tipos de luces, por defecto tenemos la luz ambiental que reproduce la iluminación natural, independiente de donde esté situada. Además, la luz puede "colorearse".



6.1.3. Partículas

Las partículas son pequeñas imágenes simples que se muestran y se mueven en grandes números en un sistema de partículas. Cada partícula representa una pequeña parte de un fluido o una entidad amorfa y el efecto de todas las partículas juntas crea la impresión de una identidad única. Un ejemplo de esto es una nube de humo, cada partícula tendría una pequeña textura de humo. Cuando todas las pequeñas nubes se unieran en un área de la escena, el efecto resultante sería de una nube mayor y con volumen.

6.1.4. Scripting

Con los cursos de scripting, de los que hablamos al principio de este apartado, nos familiarizamos con el lenguaje que usa Unity, así como las herramientas propias del mismo. Por ejemplo, cómo hacer referencia a objetos de la escena.

MonoBehaviour: es la clase base de la que todos los scripts derivan. En C# hay que especificarlo.

Awake: esta función propia de Unity se ejecuta cuando la instancia del script se carga. Awake se suele utilizar para inicializar variables o estados de juego antes de que el juego comience. Esta función se ejecuta después de que se hayan inicializado los distintos objetos así que se puede usar sin temor a que no respondan. Awake siempre se llama antes que la función Start, también propia de Unity.

Start: esta función se llama en el frame en el que se habilita un script, justo antes que cualquiera de los métodos Update se hayan llamado por primera vez. Tal y como sucede en el caso del Awake, Start se llama una única vez por tiempo de vida del script. No obstante, Awake se llama en la fase de inicialización del objeto que contiene el script, independientemente de si el script está habilitado o no. Start puede no ser llamado al mismo tiempo que el Awake si el script no está habilitado.

Update: este método se llama cada frame si el MonoBehaviour está activo. Es la función más utilizada para implementar cualquier tipo de comportamiento de juego.

FixedUpdate: esta función se llama cada determinada frecuencia de frames, si MonoBehaviour está activo. Cuando tratamos con Rigidbody se debe usar FixedUpdate en lugar de Update. Esto se debe a que el FixedUpdate se ejecuta con la misma frecuencia siempre, por lo que se considera más adecuado para manejar acciones que requieran continuidad/constancia como el movimiento.

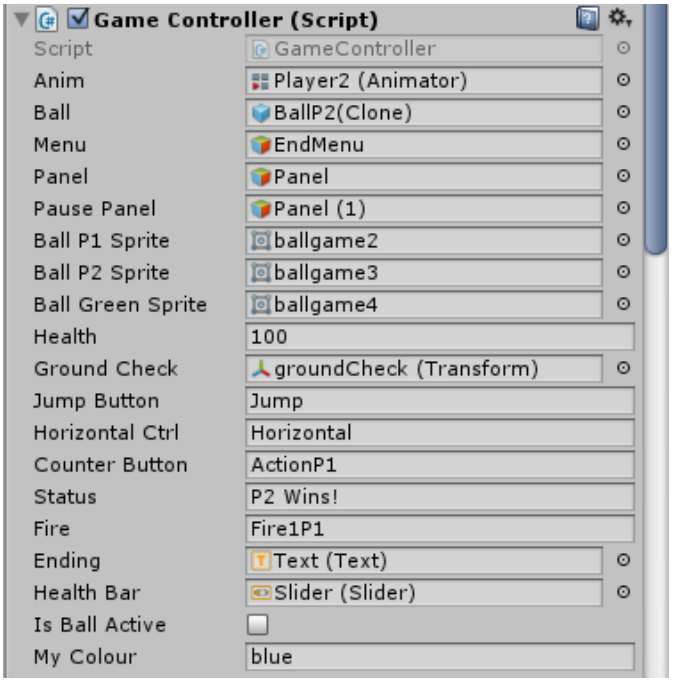
En ocasiones necesitaremos que un objeto no se destruya al pasar de una escena a otra, para esto Unity presenta algunas herramientas como el DontDestroyOnLoad. Este método permite que el objetivo no sea automáticamente destruido al cargar una escena nueva. Si el objeto es un componente o un gameObject tampoco se destruirá su jerarquía interna.

Para acceder a otros componentes en Unity tenemos que hacer uso del GetComponent<componente> (). Para hacer la llamada debemos pasarle el tipo de componente que queremos obtener. Además, un GetComponent solo se puede hacer de un gameObject.

RigidBody: este elemento controla la posición de un objeto mediante simulaciones físicas. Para obtener el RigidBody de un objeto hay que usar el GetComponent. Alternativamente, existe también el RigidBody2D para juegos en 2D.

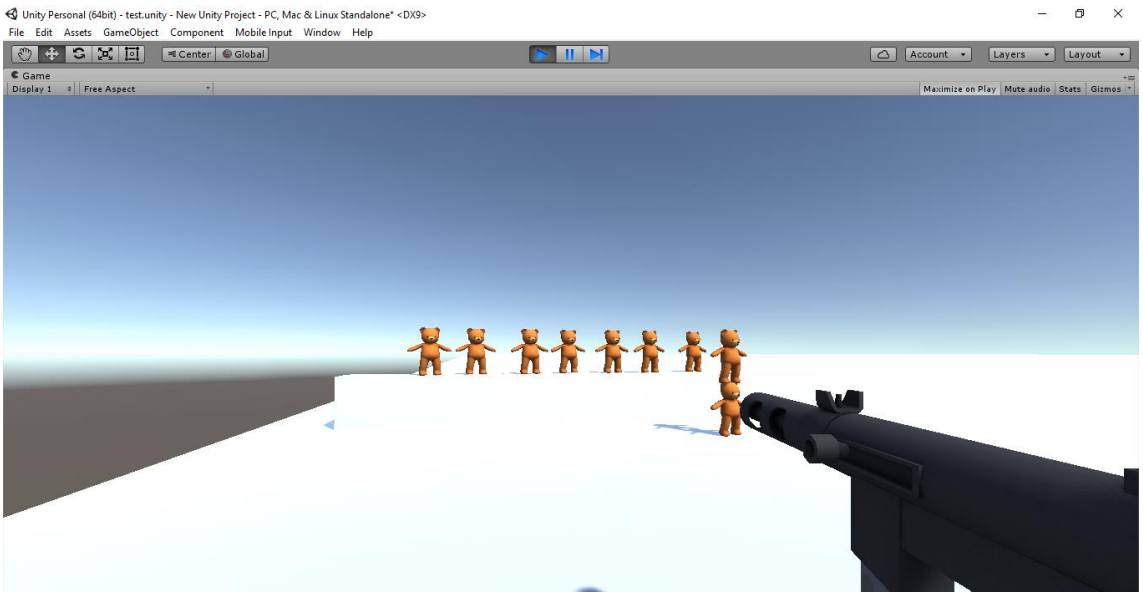
El rigidbody tiene características como la posición (en el caso de 2D, x e y) o la velocidad. Además de métodos como el AddForce que se utilizarán durante este proyecto y permiten aplicar una fuerza al rigidbody de un objeto.

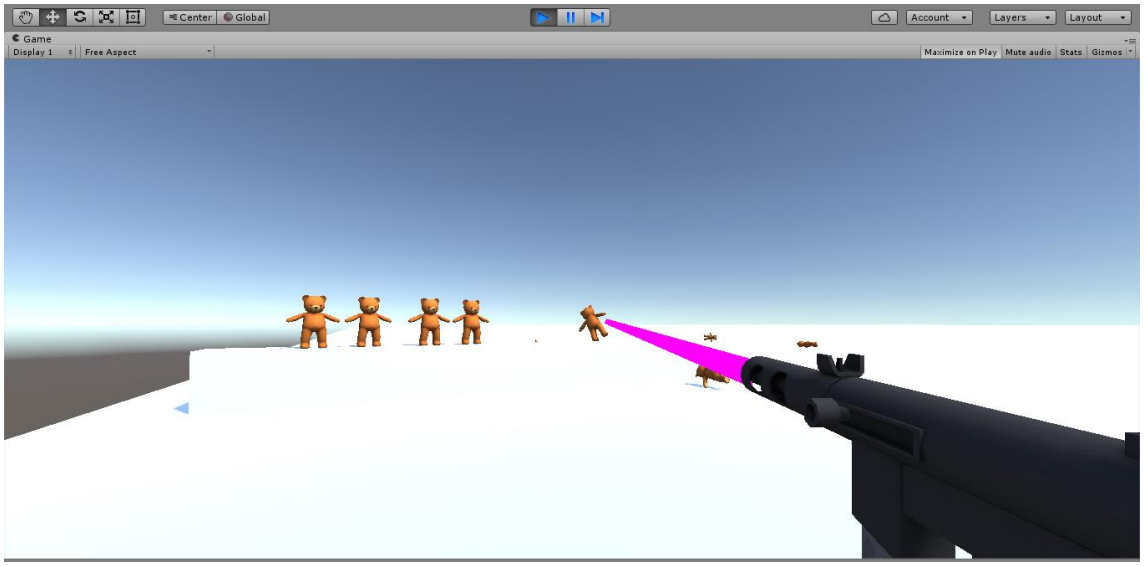
Unity permite asignar valores mediante interfaz a variables públicas. Esto resulta particularmente útil a la hora de asignar gameObjects, panales, canvas, etc. a una variable.



6.1.5. Láseres

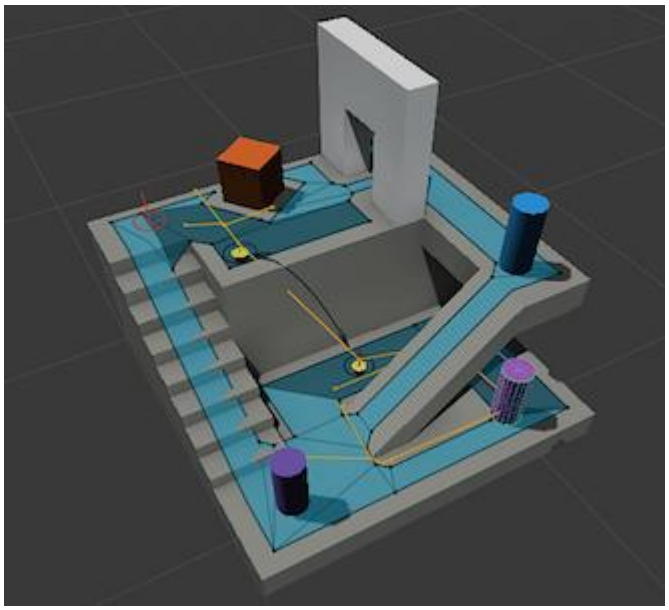
Se trabajó sobre un shooter en primera persona usando raycast y line render El raycast envía un láser invisible desde el origen hasta que encuentre un collider; el line render es similar, pero en este caso se trata de una línea y se usa en 2D.





6.1.6. Mallas de navegación

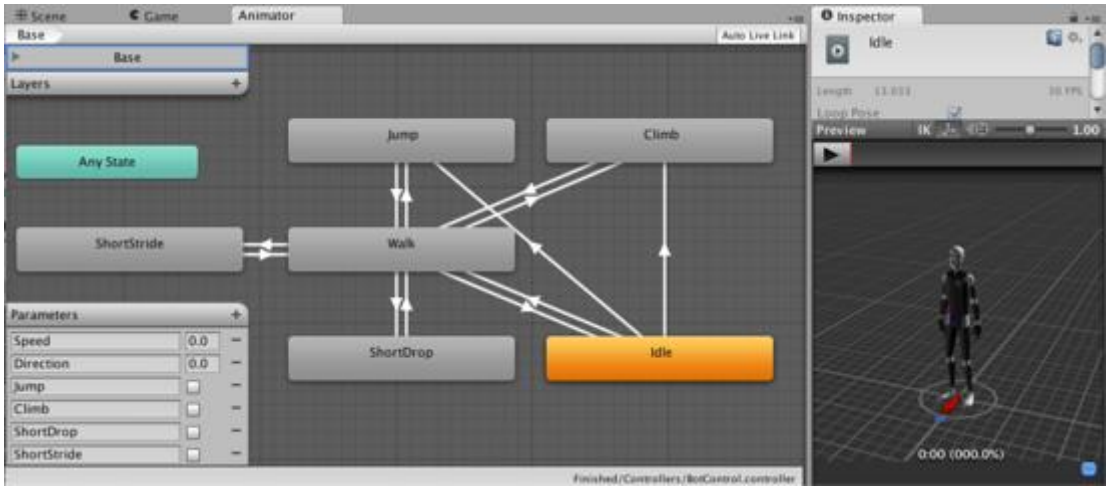
Las mallas de navegación permiten crear “caminos” para los personajes.



Esto permite que los personajes puedan moverse de forma “inteligente” en el mundo del juego.

6.1.7. Animaciones con Mecanim

Unity tiene integrada una herramienta de animación y máquina de estados llamada Mecanim. Mecanim no sólo permite animar personajes, sino que también permite animar elementos de la Interfaz, puertas abriéndose, cambio de luces, etc. Cualquier valor público puede animarse con Mecanim.



6.1.8. Persistencia

Otro de los temas tratados fue como guardar información entre distintas escenas, o entre ejecuciones del juego. Particularmente instrucciones como el DontDestroyOnLoad que han sido de utilidad en el proyecto.

6.1.9. Audio

Se trató también el componente audio en Unity. Fuentes, escuchas, audio 2D y 3D, además de scripting relacionado.

6.1.10. Granadas Teletransportadoras

En este curso se trató una mecánica de juego consistente en lanzar granadas que, al colisionar, teletransportan al jugador al lugar del impacto. Se usaron distintos sistemas para lograr el efecto deseado, comparando puntos débiles y fuertes de cada uno.

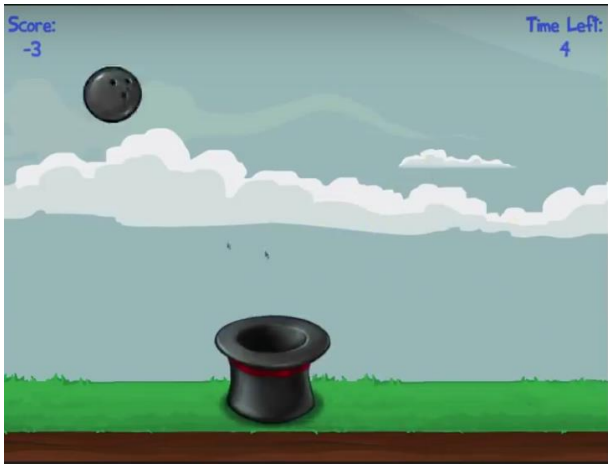
6.1.11. Explosiones y Explosiones cinemáticas

En estos cursos se trató el uso de físicas como la fuerza de impacto y el conocimiento adquirido en partículas, audio para recrear efectos de explosiones con cinemática.

6.2. Juegos realizados como ejemplos

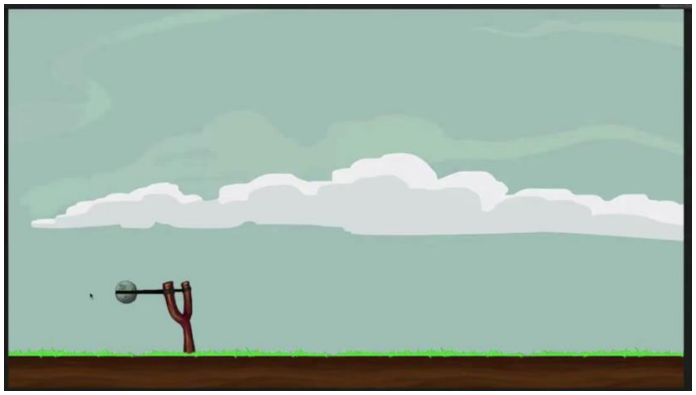
6.2.1. Ejemplo de juegos en 2D, catch game

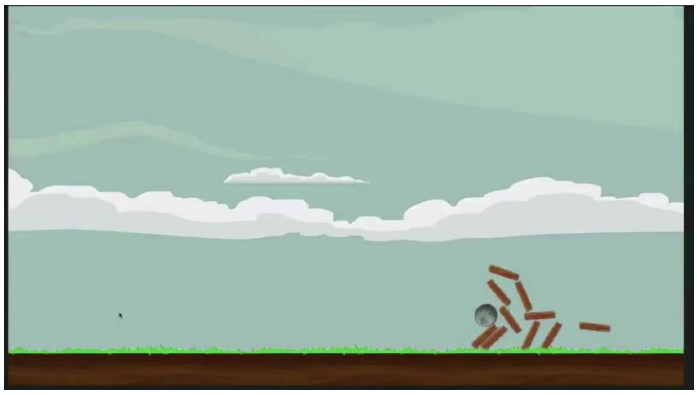
Se realizaron varios juegos siguiendo los vídeo tutoriales, los assets 2D utilizados para estos tutoriales se descargaron gratuitamente de la tienda de Unity. Con estos recursos se realizó primero un juego que generaba elementos aleatorios (bombas o pelotas) que debían ser esquivadas o atrapadas según el caso, por un sombrero que podíamos mover horizontalmente.



6.2.2. Ejemplo de juego en 2D, tipo Angry Birds

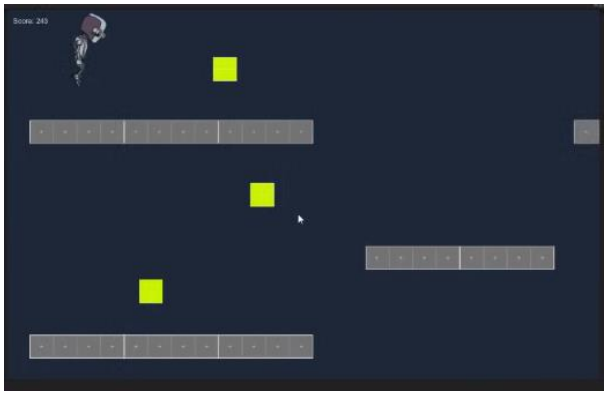
El segundo juego era un juego tipo Angry Birds. En el que aplicamos físicas para simular el lanzamiento de un tirachinas.





6.2.3. Ejemplo de juego en 2D, runner infinito

También se realizó un runner infinito en el que el escenario se generaba automáticamente y los elementos del mismo se iban destruyendo a medida que se avanzaba en la carrera. Se aprendió a realizar una generación automática del terreno para juegos de este tipo.



6.2.4. Ejemplo de juego en 2D, juego Top-Down de naves

También se siguió un vídeo tutorial sobre un juego Top-Down (vista superior) en 2D. En el que se aprendió a animar láseres en 2D.

7. Sobre el diseño de prototipos

Un prototipo de un juego debe abarcar la jugabilidad completa del mismo, incluidas condiciones de victoria y de derrota. Si bien, el prototipo es el juego en su versión más simple y minimalista.

En muchas ocasiones, habrá que cambiar elementos de jugabilidad y mecánica respecto al prototipado inicial, pero la idea es que con un prototipo se pueda mostrar el potencial del juego completo. Es importante que, si se plantea presentar el juego a un inversor, el prototipo resulte también visualmente atractivo.

7.1. Factor de diversión

Aunque pueda parecer un concepto bastante claro y que en ocasiones damos por sentado., lo cierto es que encontrar el factor de diversión a la hora de desarrollar un juego puede ser bastante complicado. Es posible que la idea que tuviéramos resultara no ser tan divertida como pensábamos, una vez plasmada, cosa que es de hecho bastante frecuente. Por esto también recalcamos la importancia del prototipado, pues nos permite ver si una idea podría funcionar o no.

Con este aspecto se tuvo cierta dificultad a la hora de desarrollar prototipos puesto que inicialmente se obvió el factor diversión o se dio por sentado. Para indagar más sobre este tema se realizó un estudio de juegos retro de éxito. Hablaremos sobre esto a continuación.

7.2. Estudios y realización de juegos de arcade clásicos de gran éxito

Se estudiaron tres juegos arcade de gran éxito, *Pong*, *Arkanoid* e *Space Invaders*. Se plantearon dos preguntas principales: ¿Qué los hace divertidos? ¿Qué hay de diferente respecto a las nuevas versiones de estos juegos? Nuestro objetivo era encontrar el factor de diversión.

¿Qué los hace divertidos?

Consideramos que son juegos de mecánica sencilla y enganchan. Inicialmente tienen poca dificultad lo que permite hacerse con los controles del juego. A medida que el tiempo pasa o se sube de nivel, la dificultad va aumentando. Esto conlleva que el jugador no se frustre demasiado pronto y que cuando pierda quiera seguir jugando para superarse a sí mismo. El objetivo no es tanto acabar el juego sino cual es la máxima puntuación que es capaz de obtener el jugador por lo que son juegos potencialmente "interminables".

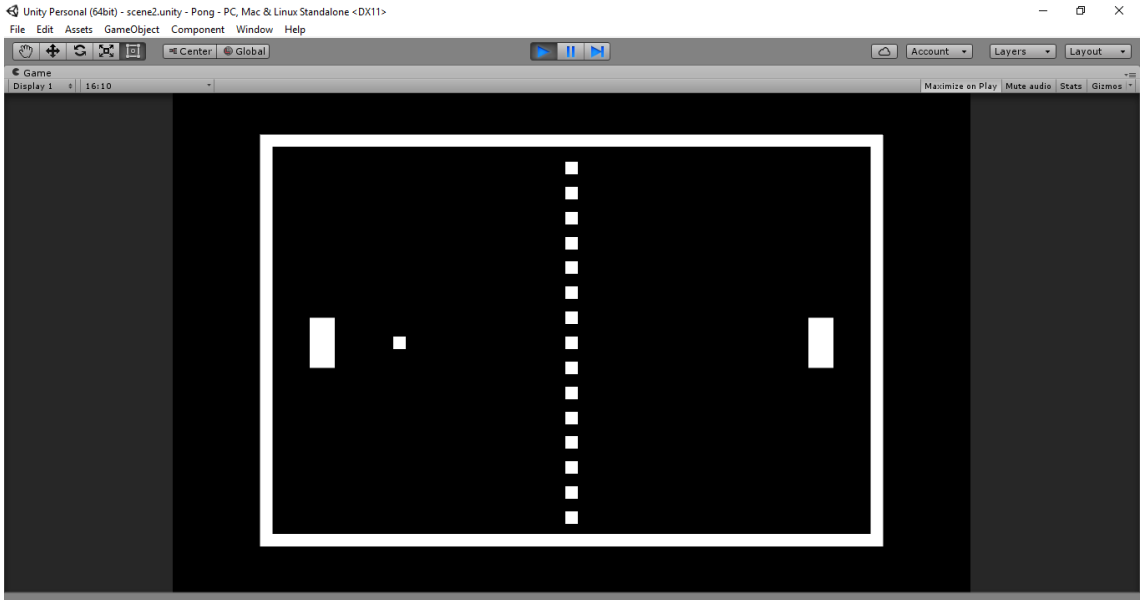
¿Qué hay de diferente respecto a las nuevas versiones de estos juegos?

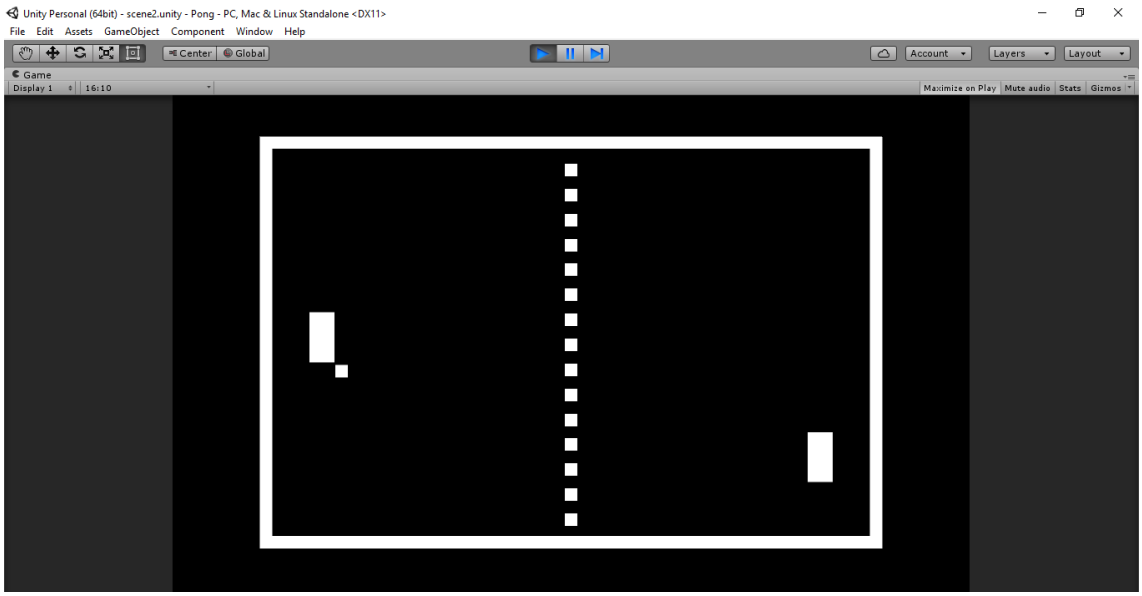
Principalmente conservan la misma mecánica que los juegos originales, pero se mejoran visualmente y se añaden distintas variables (como powerups). En ocasiones se pierde la diversión que aportaba el juego original. Uno de los Pong a los que se jugó, por ejemplo, tenía una IA muy simple y se limitaba a seguir la pelota, pero al ser la raqueta más lenta que la pelota si se conseguía que la bola fuera de un extremo a otro la IA no podía alcanzar la pelota.

Se realizó además una versión de cada uno de estos juegos clásicos. Los cuales trataremos en detalle a continuación.

7.2.1. Pong

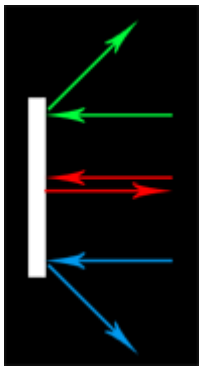
El *Pong* es un juego en dos dimensiones que simula un tenis de mesa. El jugador controla en el juego una paleta moviéndola verticalmente en la parte izquierda de la pantalla, y puede competir tanto contra un oponente controlado por computadora, como con otro jugador humano que controla una segunda paleta en la parte opuesta. Los jugadores pueden usar las paletas para golpear la pelota hacia un lado u otro. El objetivo consiste en que uno de los jugadores consiga más puntos que el oponente al finalizar el juego. Estos puntos se obtienen cuando la pelota alcanza un borde de la pantalla sin que el contrario haya podido devolverla.





Si la pelota se perdía por la derecha o la izquierda el juego se reiniciaba. Cada vez que la pelota se perdía por un lado se suma un punto al jugador contrario al que pierde la pelota. De esta forma, cuando uno de los dos llega a 10, gana y la partida acaba.

Para calcular la dirección de la pelota calculamos la posición de la pelota respecto a la pala tal y como mostramos a continuación, la dirección de salida de pelota dependerá de donde golpee la pala.



```
-

float hitFactor(Vector2 ballPos, Vector2 racketPos,
    float racketHeight) {
    // vector of the direction of the ball / racket height
    return (ballPos.y - racketPos.y) / racketHeight;
}

void OnCollisionEnter2D(Collision2D col) {
    // If the ball collided with a racket, then:
    //   col.gameObject is the racket
    //   col.transform.position is the racket's position
    //   col.collider is the racket's collider

    // Hit the left Racket?
    if (col.gameObject.name == "RacketLeft") {
        // Calculate hit Factor
        float y = hitFactor(transform.position,
            col.transform.position,
            col.collider.bounds.size.y);

        // Calculate direction, make length=1 via .normalized
        Vector2 dir = new Vector2(1, y).normalized;

        // Set Velocity with dir * speed
        GetComponent<Rigidbody2D>().velocity = dir * speed;
    }

    // Hit the right Racket?
    if (col.gameObject.name == "RacketRight") {
        // Calculate hit Factor
        float y = hitFactor(transform.position,
            col.transform.position,
            col.collider.bounds.size.y);

        // Calculate direction, make length=1 via .normalized
```

Para implementar esta situación usamos un OnCollisionEnter2D para averiguar si la pelota ha hecho alguna colisión. Identificamos primero con qué objeto colisiona, si es con la raqueta izquierda o derecha (de esto dependerá la dirección de salida de la pelota). Con el método hitFactor calculamos el vector de posición en el que se devolverá la pelota dividido por la longitud de la pala. Se calcula posteriormente la dirección, la x por defecto será positiva mientras que la y es la dirección calculada previamente. Además, se le asigna una velocidad a la pelota.

El caso de la pala derecha es similar pero la dirección de la pelota en el eje x es contrario al de la pala izquierda.

Sobre el score, si la pelota colisiona con cualquiera de las paredes se manda un mensaje al objeto Score, que guarda la puntuación de los jugadores, de forma que se aumenta la puntuación de uno de los jugadores según quién haya perdido la pelota.

```
    if (col.gameObject.name == "WallLeft") {
        //Instantiate (newBall);
        score.gameObject.SendMessage("RightScore");
        Destroy (this.gameObject);

    }
    if (col.gameObject.name == "WallRight") {
        //Destroy (ball);
        score.gameObject.SendMessage("LeftScore");
        Destroy (this.gameObject);

    }
}
```

La condición de victoria es llegar a los 10 puntos.

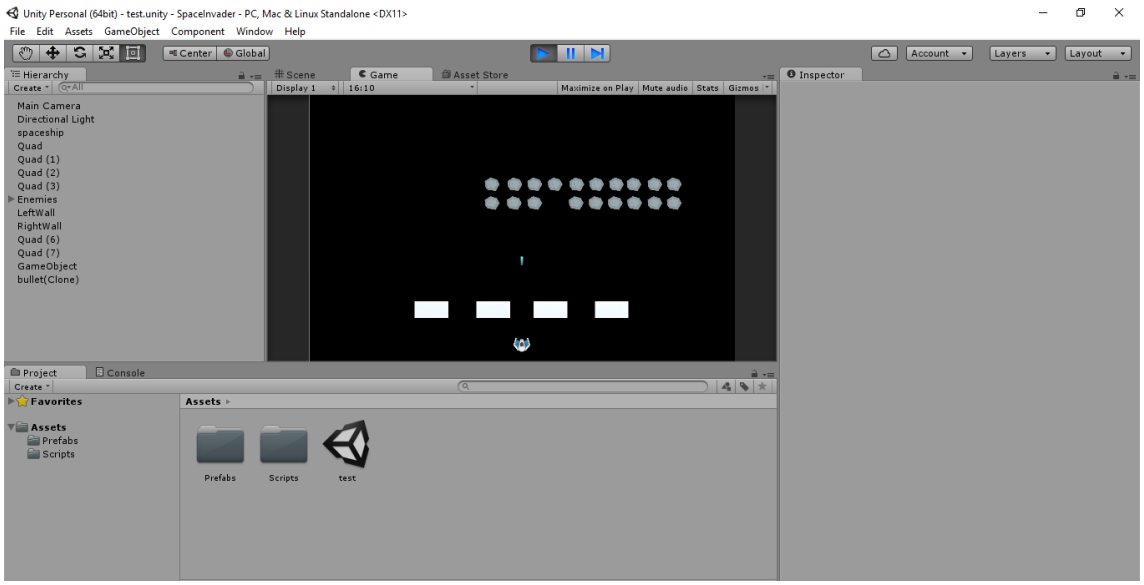
```
public class Score : MonoBehaviour {
    public float scoreLeft;
    public float scoreRight;
    public GameObject newBall;

    // Use this for initialization
    void Start () {
        scoreLeft = 0;
        scoreRight = 0;
    }

    void RightScore () {
        scoreRight++;
        if (scoreRight == 10) {
            Debug.Break ();
            return;
        }

        Instantiate (newBall);
    }

    void LeftScore () {
        scoreLeft++;
        if (scoreLeft == 10) {
            Debug.Break ();
            return;
        }

        Instantiate (newBall);
    }
```

Con Instantiate generamos un estado similar al del comienzo del juego, pues se instancia una pelota en la posición y condiciones iniciales.

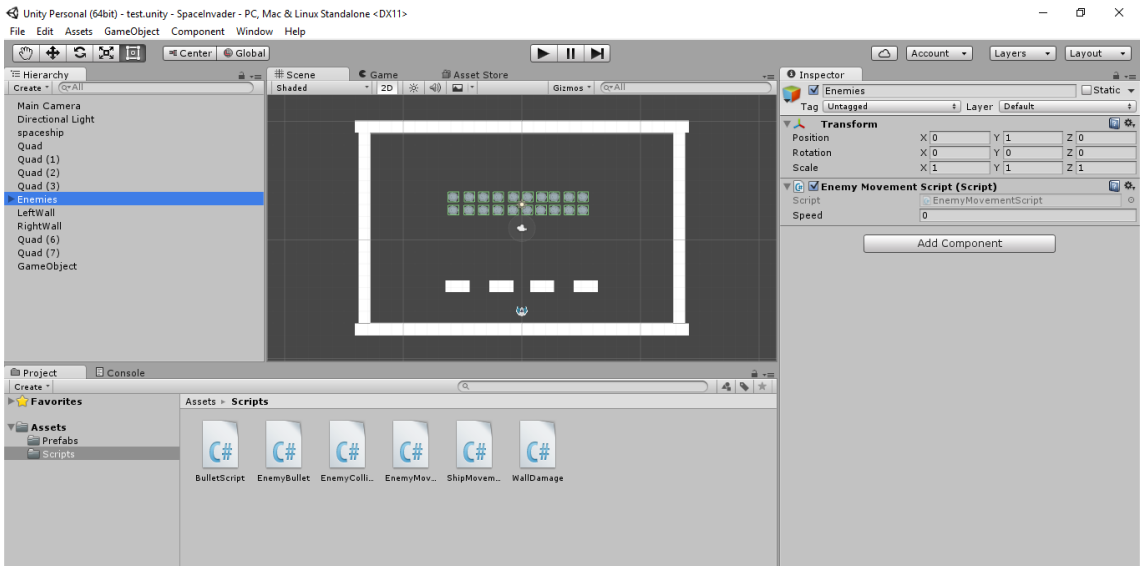
7.2.2. Space Invaders

Es uno de los videojuegos más importantes de la historia. Su objetivo es eliminar oleadas de alienígenas con un cañón láser y obtener la mayor cantidad de puntos posible. Para el diseño del juego, Nishikado se inspiró en *Breakout*, La guerra de los mundos y *Star Wars*.

El jugador controla un cañón que puede moverse a la derecha o izquierda y un botón de disparo. Tiene que ir destruyendo los extraterrestres invasores que van acercándose a la tierra cada vez más rápidamente a medida que el jugador va destruyendo a los enemigos. Este ciclo se puede repetir en forma indefinida. Si los invasores llegan al cañón controlado por el jugador, el juego termina



En la realización de este prototipo se usaron clones de un mismo elemento colocados manualmente. Para el movimiento de los mismos todos los enemigos pertenecen a un mismo objeto que contiene el script de movimiento.



El script de movimiento es muy simple. Al comienzo de la escena se le asigna una velocidad específica al objeto conjunto, y durante cada frame se actualiza. Time.deltaTime es el tiempo en segundos que duró el frame anterior. El transform.Translate por su parte representa el vector de movimiento, siendo el orden de las variables x,y,z. Por lo tanto, estamos estableciendo que se desplace horizontalmente.

El Transform, que poseen todos los objetos de una escena, permite establecer posición, escala y rotación de un objeto.

```
using UnityEngine;
using System.Collections;

public class EnemyMovementScript : MonoBehaviour {
    public float speed;

    // Use this for initialization
    void Start () {
        speed = 1f;
    }

    // Update is called once per frame
    void Update () {
        float translation = Time.deltaTime * speed;
        transform.Translate(translation, 0, 0);
    }

}

```

Un poco más complicado es el controlador de colisiones que tiene cada enemigo individualmente.

```
using UnityEngine;
using System.Collections;

public class EnemyCollision : MonoBehaviour {
    public Vector2 dropAmount = new Vector2 (0, 1);
    public GameObject spaceship;
    public GameObject enemyBullet;


    void OnCollisionEnter2D (Collision2D col) {
        // Mirar si la colision es con uno de los muros
        if (transform.parent.GetComponent<EnemyMovementScript>().speed > 0) {
            if (col.gameObject.name == "RightWall") {
                // Cambiar la dirección hacia el otro lado
                transform.parent.GetComponent<EnemyMovementScript> ().speed *= -1f;
                transform.parent.transform.Translate (-dropAmount);
            }
        }

        if (transform.parent.GetComponent<EnemyMovementScript> ().speed < 0) {
            if (col.gameObject.name == "LeftWall") {
                // Cambiar la dirección hacia el otro lado
                transform.parent.GetComponent<EnemyMovementScript> ().speed *= -1f;
                transform.parent.transform.Translate (-dropAmount);
            }
        }


    }
```

La función OnCollisionEnter2D es propia de Unity, se activa cuando hay colisiones con otros objetos con collider en la escena. El objeto contra el que se produce la colisión es la variable que se le pasa y es detectado automáticamente por Unity.

Así pues, en nuestro caso, cuando los enemigos chocan contra una de las paredes se activa este método. En este caso hemos aprovechado para comprobar cómo se hacían llamadas a componentes del "padre" de un elemento. Aunque con el conocimiento adquirido se habría realizado este script de forma distinta. Nos aseguramos también de que la colisión es con el objeto que esperamos. Y entonces cambiamos el sentido del movimiento del padre (el conjunto de enemigos) y además hacemos que avancen hacia el jugador.

```
31      void OnTriggerEnter2D (Collider2D col) {
32          string name = col.gameObject.name;
33
34          // If it collided with a bullet
35          if (name == "bullet(Clone)") {
36              // Destroy itself (the enemy)
37              Destroy(gameObject);
38
39              // And destroy the bullet
40              Destroy(col.gameObject);
41          }
42
43          // If it collided with the spaceship
44          if (name == "spaceship") {
45              // Destroy itself (the enemy) to keep things simple
46              Destroy(gameObject);
47          }
48      }
49
50  }
51
```

En este script disponemos también de otra función nativa de Unity OnTriggerEnter2D, esto nos permite detectar colisiones con elementos con la casilla "OnTrigger" marcada. Por lo tanto, no todas las colisiones se detectarán.

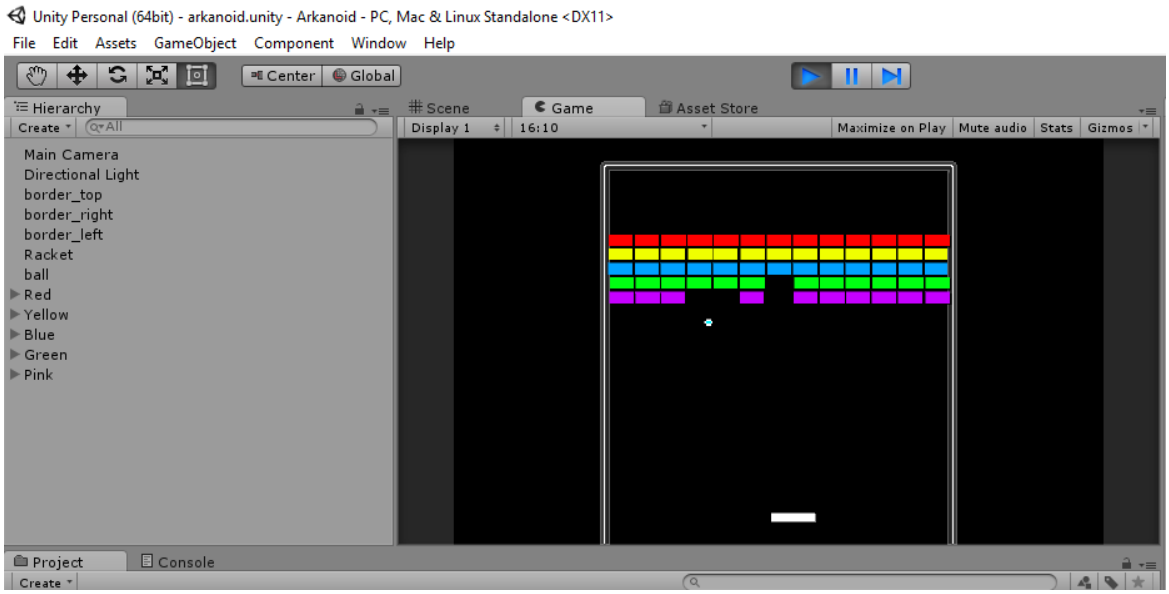
Nos interesa sobre todo detectar las colisiones de las balas del jugador, si estás colisionan con cualquiera de los enemigos estos se autodestruyen y destruyen la nave enemiga.

Las defensas tienen su propio código que lleva la cuenta de la "vida" que tienen o les queda. Puesto que cuando las balas enemigas o aliadas los golpean van reduciendo su "vida" hasta quedar destruidas.

7.2.3. Arkanoid

El jugador controla una pequeña plataforma, que impide que una bola salga de la zona de juego, haciéndola rebotar. En la parte superior hay "ladrillos" o "bloques", que desaparecen al ser tocados por la bola.

Cuando no queda ningún ladrillo, el jugador pasa al siguiente nivel, donde aparece otro patrón de bloques. Cada vez que un jugador pasa de pantalla el nivel de dificultad va en aumento.



Al igual que con Space invaders los bloques se podrían haber distribuido utilizando una matriz, aunque se siguió el mismo sistema que en el Space Invaders. En este caso cada bloque de un mismo color pertenece a un objeto padre.

Cada bloque contiene un script que de forma similar a lo que hemos visto en el Space Invaders, utiliza un OnCollisionEnter2D para detectar si la pelota ha impactado y en caso de que así sea, destruir el bloque.

La pala tiene su propio código en el que se establece el eje de desplazamiento y la velocidad del mismo.

Por último, la pelota contiene el código por el cual rebota de la forma en la que lo hace. Este código fue reutilizado del Pong y funciona de manera similar.

¿Cómo se hacen el Arkanoid, Pong y Space Invaders más difíciles?

En el Arkanoid la dificultad aumenta debido a la existencia de distintos tipos de bloques y la colocación de los mismos. Mientras más cerca están los bloques de la raqueta, por ejemplo, menos tiempo de reacción se tiene. Además, la velocidad de la bola varía y hay bloques que deben golpearse varias veces antes de conseguir destruirlos.

En el Pong la dificultad radica en la habilidad del oponente (otro jugador o IA). En el caso de la IA por ejemplo, una velocidad alta de movimiento hace más difícil ganar. Respecto al Space Invaders, la velocidad aumenta a medida que se destruyen las hordas enemigas. Además, las nuevas hordas son más numerosas y aparecen cada vez más cerca del jugador.

8. Ideas y diseño de prototipos

8.1. Propuestas y estudio

Durante las primeras semanas se propusieron hasta 20 juegos distintos de los que posteriormente cuatro se analizaron y se consideraron como proyecto.

Juego 1

El personaje principal puede clonarse y utiliza sus clones para avanzar, resolver puzles o luchar. Podría ser, bien con un único clon al principio, e ir aumentando el número gradualmente; o pudiendo hacer múltiples clones a la vez, pero mejorando el tiempo que pueden mantenerse activos. En ambos casos habría un gasto de energía por mantener clones activos. Si se pasa de esta barra de energía podría comenzar a consumir vida.

Juego 2

El personaje principal es de plastilina, se usan moldes para que el personaje pueda cambiar de forma y esto le permite pasar por determinados lugares o utilizar habilidades distintas para ir avanzando. Tipo plataformas.

Juego 3

En esta propuesta el contexto es un poblado medieval donde en determinadas noches de invierno, criaturas extrañas se llevan a los habitantes. Nadie sale de noche ni nadie se adentra en el lugar del que proceden las criaturas. El personaje principal decide ir en busca de x, raptado por las criaturas, el personaje solo puede avanzar de día y debe buscar refugio antes de que caiga la noche. No se puede luchar contra los enemigos. Tipo survival, misterio, primera persona.

Juego 4

El personaje principal y el jugador interactúan de alguna forma, mediante diálogos, el personaje es consciente de que hay alguien más ayudándole, aunque no sabe de dónde procede la voz que le habla. En este juego se controlaría al personaje principal en determinadas circunstancias, aunque principalmente se le darían órdenes o sugerencias para avanzar. Plataformas, 2D.

Juego 5

El personaje principal aparece en un mundo ajeno, su último recuerdo es haber muerto. El mundo es alienígena y extraño, donde los personajes proceden de distintas épocas y nadie sabe por qué razón han aparecido ahí. Exploración.

Juego 6

El personaje principal (se propone un hámster en una bola) rueda permanentemente y hay que interactuar con el entorno para retirar obstáculos, evitar caídas mortales, etc. 2D

Juego 7

Un grupo de supervivientes de un accidente comparten balsa salvavidas con víveres limitados. El jugador decide cuándo usar los recursos. Se generan dificultades aleatorias, enfermedad,

tiburones, etc… Se puede intentar lograr que todos o la mayoría de los personajes sobrevivan o el personaje principal puede intentar salvarse solo a sí mismo a costa de los demás.

Juego 8

El personaje principal es una foca que debe desplazarse de isla en isla en un archipiélago evitando a los tiburones. En tierra la foca está a salvo, pero para avanzar debe meterse en el agua, se pueden usar cuevas y pequeños recodos para esquivar a los predadores. Alimentarse/pescar permite recuperar vida.

Juego 9

Se empieza con un pequeño círculo con una fuerza de atracción baja que debe acercarse a círculos más pequeños para atraparlos en su órbita y crecer. Con cuidado de no quedar atrapados en la órbita de un círculo de mayor tamaño.

Juego 10

Se trata de destruir edificios usando explosivos. Para ello se disponen de distintos tipos de explosivos, mayor o menor potencia, distintos efectos, etc. Y deben ser colocados estratégicamente para demoler las construcciones. En ciertos niveles puede bastar con tirar el edificio en cualquier dirección, pero en otros puede haber un edificio a la derecha que no queramos destruir, así que debemos forzar al edificio a caer a la izquierda o sobre sí mismo. En otras ocasiones, poner poca carga de manera regular puede suponer que el edificio pierda "altura", pero no quede derruido.

Juego 11

Plataformas, el personaje principal está permanentemente saltando lo que añade cierta dificultad a la hora de controlarlo.

Juego 12

Se controla a un grupo de personajes con habilidades distintas, se usan las distintas habilidades para avanzar. Por ejemplo, un personaje puede ser elástico y estirarse para formar un puente, hacer de "liana" o permitir a los demás llegar hasta sitios altos.

Juego 13

Personaje principal cambia de forma para avanzar adquiriendo características de animales.

Juego 14

En un entorno cerrado se tienen que llevar a cabo misiones de asesinato, el personaje principal puede decidir cómo llevar a cabo la misión, usando veneno, flechas desde una posición ventajosa, apuñalamiento, etc. Y evitar ser descubierto durante la misión.

Juego 15

Un personaje tiene que escalar por las ramas de un árbol, hay diferentes caminos y el personaje puede saltar de rama en rama. En niveles posteriores se añaden obstáculos, por ejemplo, un incendio y que el fuego persigue al personaje.

Juego 16

El personaje principal es una criatura de nieve, avanza rodando y va perdiendo nieve por el desgaste y al ser herido. Para recuperar vida tiene que recoger nieve.

Juego 17

El personaje principal está en una habitación cerrada, no hay instrucciones (salvo como controlar al personaje), tiene que encontrar la forma de salir.

Juego 18

El personaje principal tiene un grupo de pequeños siervos (duendes, gnomos, goblins) a su disposición. Son una pequeña marea que el personaje principal puede utilizar para luchar o avanzar. Por ejemplo, se les puede ordenar inmovilizar a un enemigo para que el personaje principal pueda atacar con facilidad o hacer que formen un puente para poder sobrepasar un obstáculo.

Juego 19

Juego de peleas en los que los personajes se lanzan objetos desde un lado y otro de la pantalla (por ejemplo, pelotas). El personaje puede bloquear un objeto, devolverlo o esquivar, además de atacar.

Juego 20

Eres una criatura gigante y tienes que destruir una ciudad mientras te ataca un robot gigante. Ganas puntos mientras más destrucción causes en un tiempo determinado antes de que el ejército te detenga. Bonos si causas destrucción tirando al robot o consigues que el robot destruya cosas.

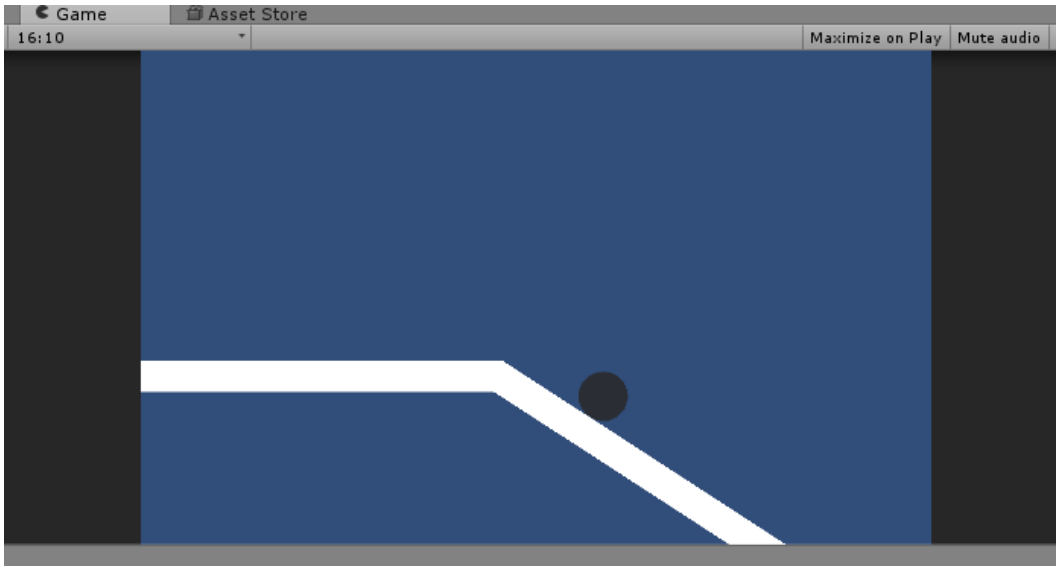
8.2. Primera selección

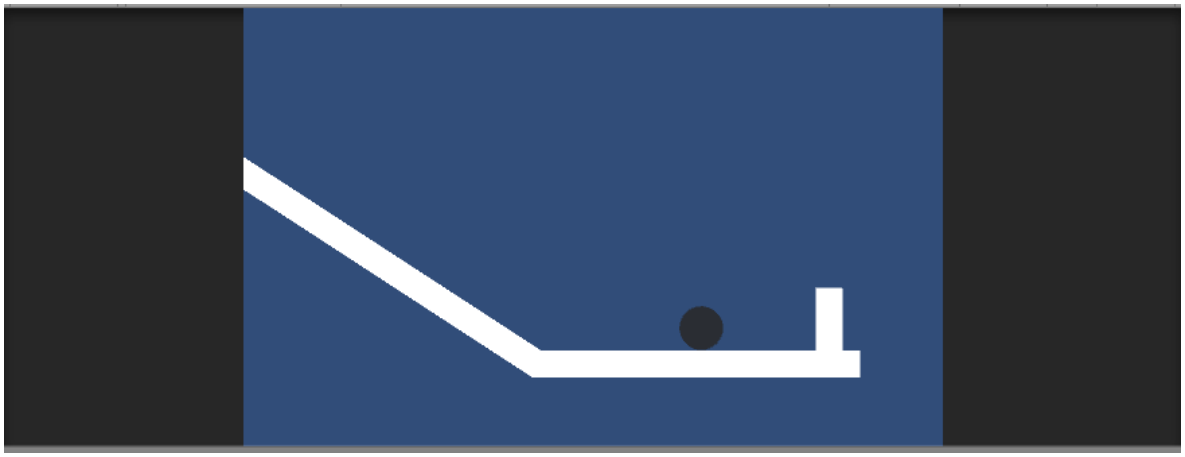
La primera selección se hizo con los juegos 6, 7, 11, 18, 19, 20 aunque finalmente se realizaron los prototipos para cuatro de estos juegos, quedándonos entonces con las ideas 6, 11, 19 y 20.

8.3. Desarrollo de prototipos

Roll Prototype

En este prototipo la intención es que el jugador interactúe con el entorno para permitir que la pelota siga rodando.





En este prototipo desarrollamos un script para que a medida que la cámara se desplazara se fueran destruyendo los elementos del juego que quedaban fuera del mismo. Pues el planteamiento era ir generando un entorno 2D que se fuera destruyendo a medida que se avanzaba.

En este caso utilizábamos los collider de los objetos, si el jugador se chocaba con un Destroyer (si se sale de la pantalla) el juego acaba. Si cualquier otro objeto entra en contacto con un Destroyer se elimina.

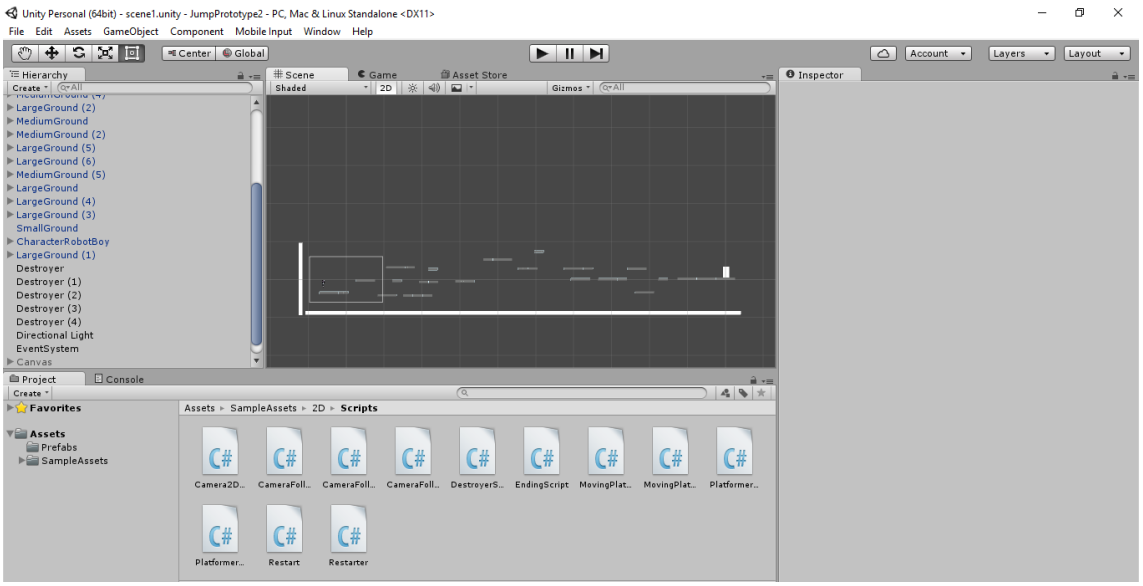
DestroyerScript.cs | Pass.cs | Rolling.cs

No selection

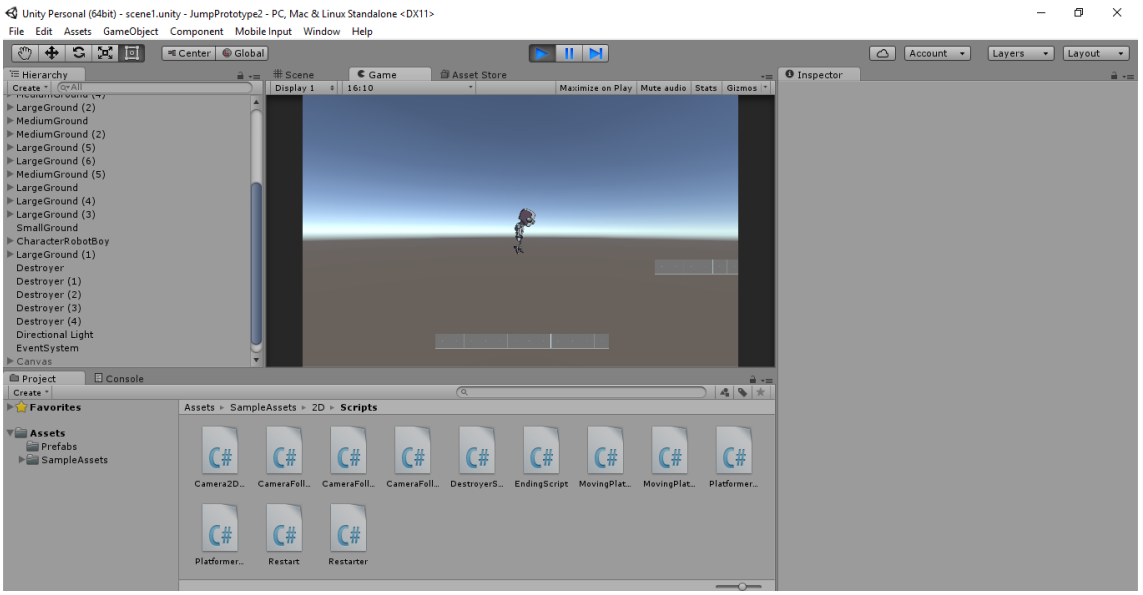
```
1  using UnityEngine;
2  using System.Collections;
3
4  public class DestroyerScript : MonoBehaviour {
5
6      void OnTriggerEnter2D (Collider2D other) {
7          if (other.tag == "Player") {
8              Debug.Break ();
9              return;
10         }
11
12         if (other.gameObject.transform.parent) {
13             Destroy (other.gameObject.transform.parent.gameObject);
14         } else {
15             Destroy (other.gameObject);
16         }
17     }
18 }
19
```

Jump Prototype

El prototipo de este juego en 2D es un escenario finito, aunque se podría ampliar a un escenario generado a medida que se va avanzando. La condición de victoria es llegar al final del recorrido (el rectángulo blanco).

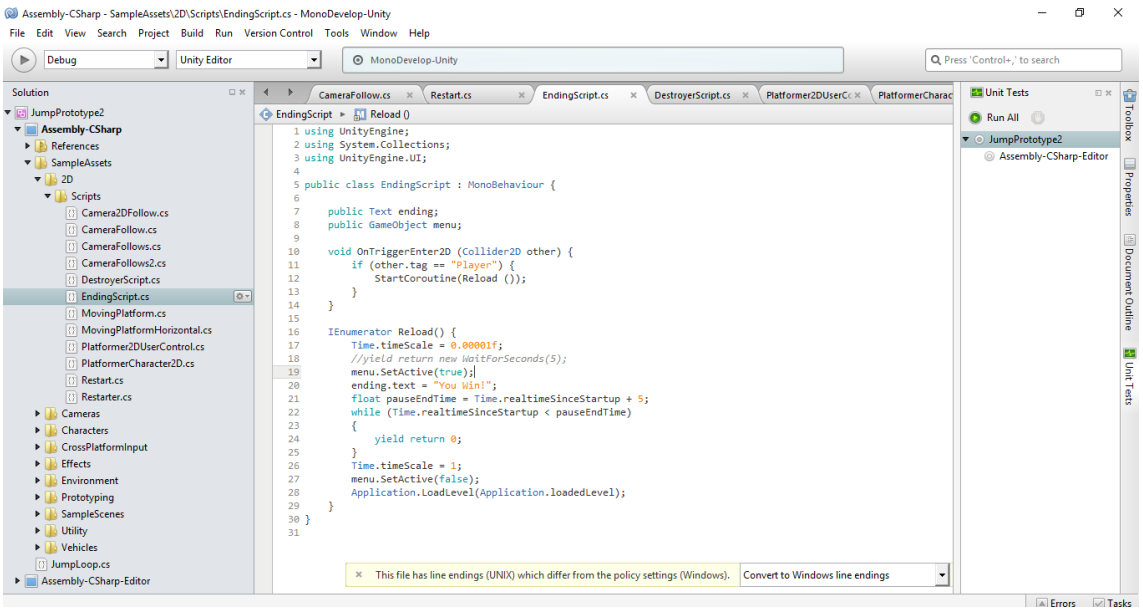


Algunos de las plataformas tienen movimiento horizontal o vertical, para añadir dificultad al juego.



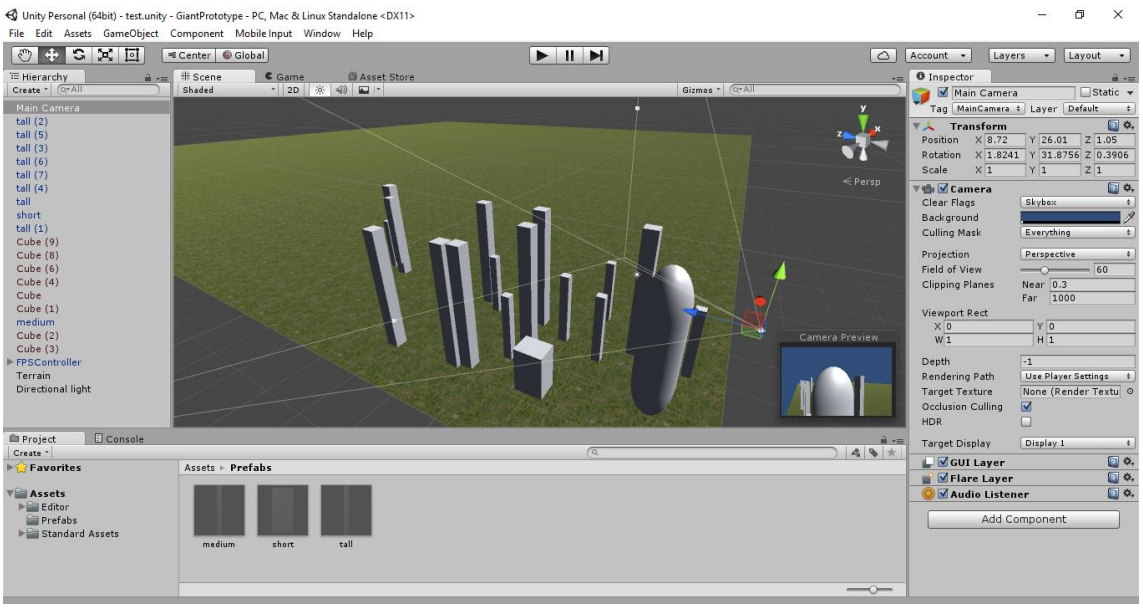
Script de fin de partida:

Cuando el jugador colisiona con el objeto que marca el final del nivel, se lanza una corrutina en la que el tiempo se “detiene”, esto es, se ralentiza tanto que parece estar detenido; para mostrar el mensaje de victoria. Después de 5 segundos se retoma el tiempo normal de juego y se recarga el nivel. Este código se reutilizará parcialmente para el prototipo realizado finalmente.

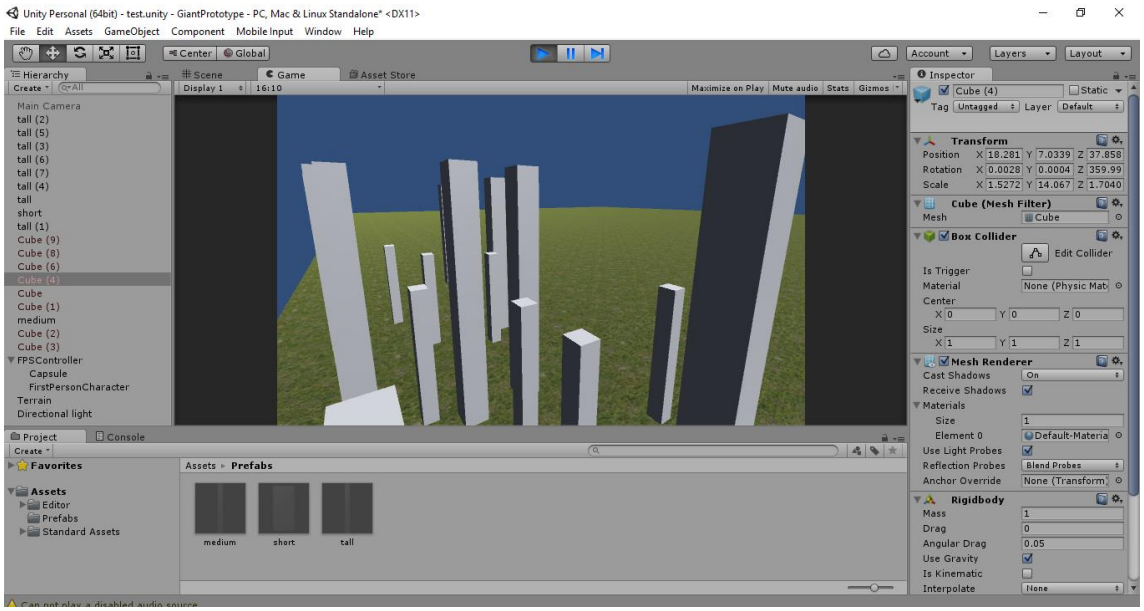


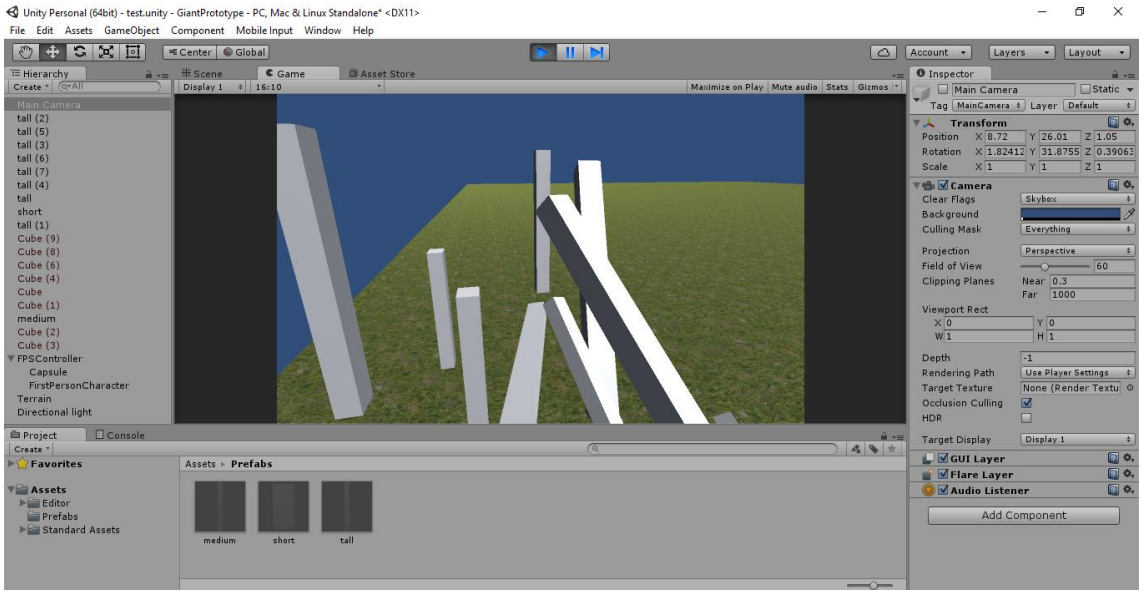
Giant Prototype

La versión inicial de este prototipo era muy básica porque se pretendía representar las mecánicas del juego de forma simple y rápida. La idea es que el personaje principal pueda destruir edificios a medida que avance.



A continuación, tenemos un ejemplo de la vista en primera persona del juego en su inicio y después de haber avanzado destruyendo “edificios”.

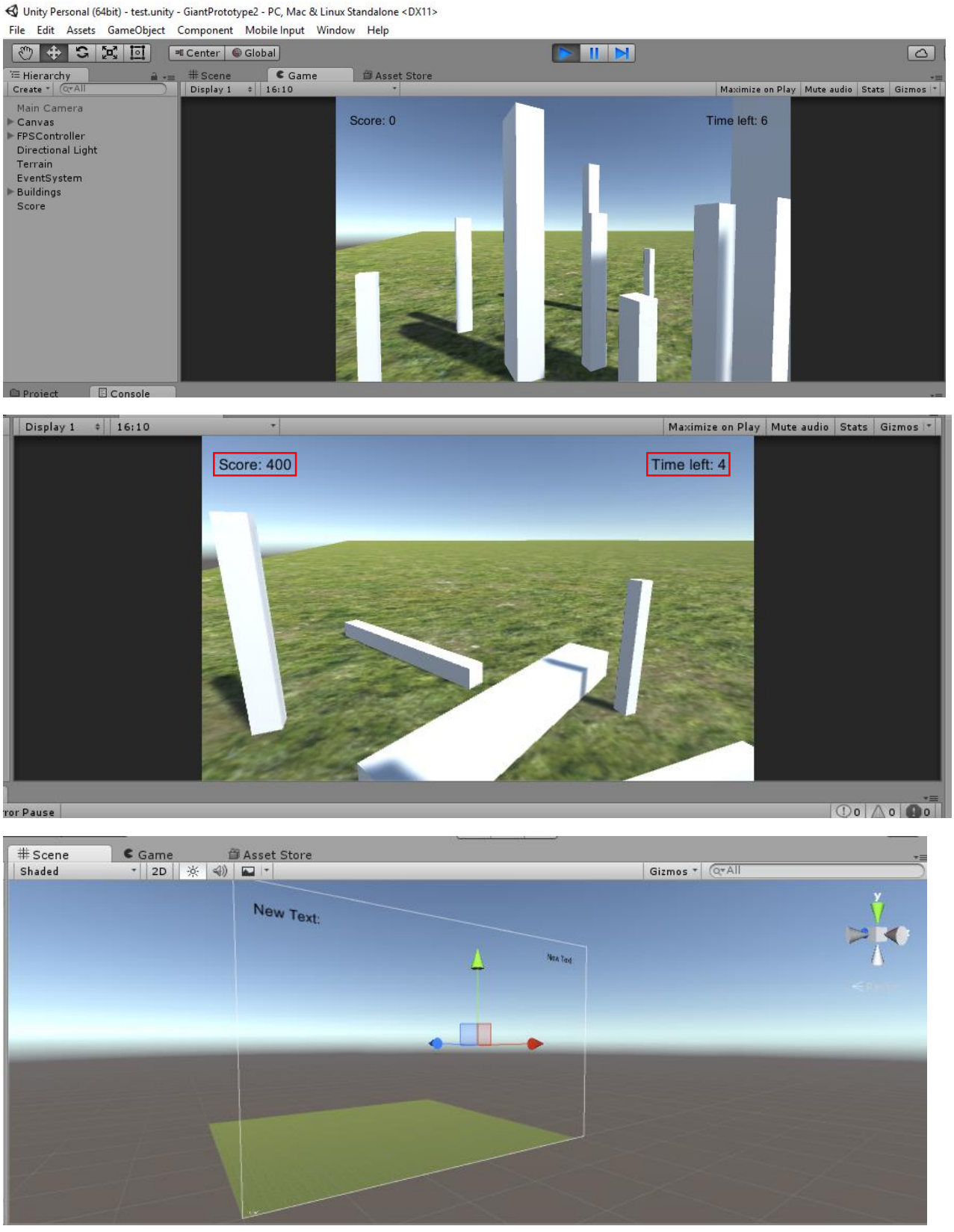




Se desarrolló una segunda versión del prototipo algo más refinada, con un contador de puntuación, así como de tiempo. De forma que el juego fuera autoconclusivo. La finalidad era tirar la mayor cantidad de edificios posibles en el tiempo dado. Para el prototipo el tiempo era corto de forma que se permitía realizar las pruebas con mayor agilidad.

El desarrollo de este juego permitió familiarizarnos con elementos de Unity como la interfaz de un juego y la forma de representar datos en la misma.

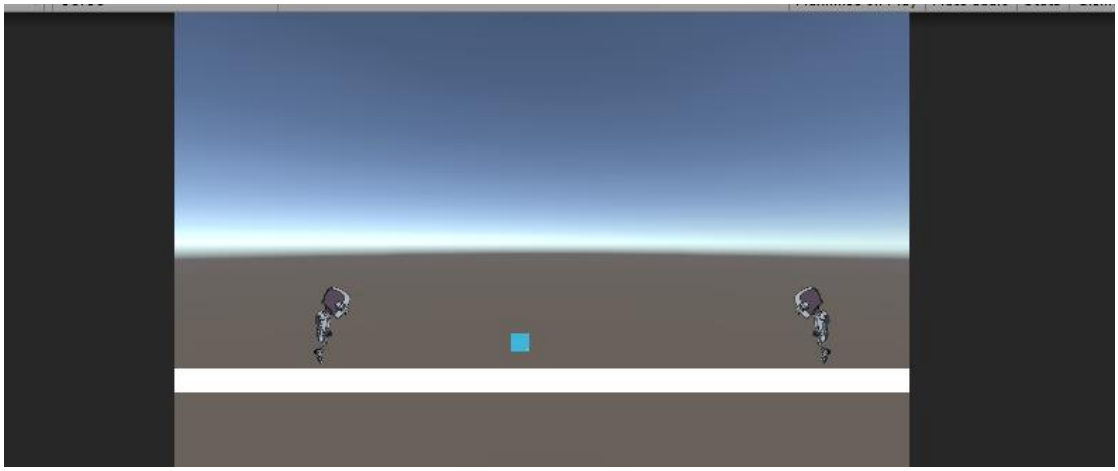
Segunda versión



Canvas en vista 3D en el editor.

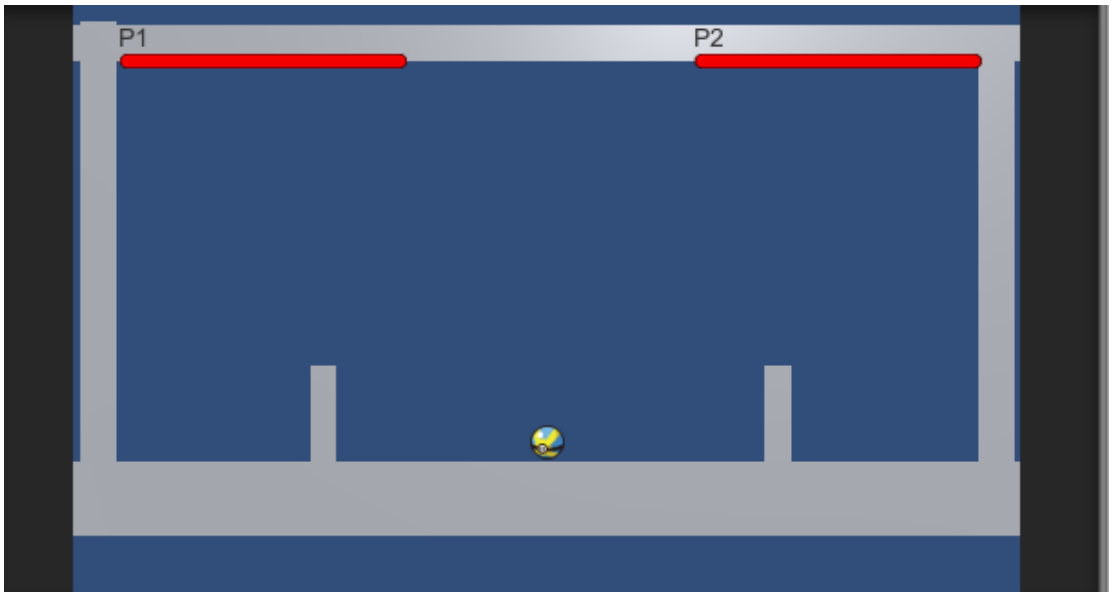
Throw Prototype

Este prototipo pretendía ser un juego de peleas en el que los jugadores lanzaban armas contra su enemigo en lugar de usar los puños o patadas. El concepto inicial que podemos ver a continuación no captura la mecánica actual del juego, pues aún se estaba tratando de plasmar la idea. Al principio el objeto se lanzaba en línea recta.



Pikachu Volleyball y *Lethal League* fueron dos juegos que se estudiaron para plasmar la idea de un juego de peleas con una pelota. De ahí nació la versión actual del juego, en el que se utiliza una pelota con físicas 2D (de forma que rebota) y los jugadores para dañar al contrario. No obstante este prototipo inicial, aún con la nueva mecánica, no abarcaba la jugabilidad definitiva juego.

Hablaremos más de este prototipo en el desarrollo del juego porque fue el juego elegido.



8.4. Elección final

En cuanto a los prototipos desarrollados, Roll tenía potencial de convertirse en un juego para móviles, lo cual habría necesitado un estudio adicional y más tiempo. Por otra parte, descartamos el Jump por ser un juego de plataformas, puesto que es el tipo de juego más sencillo para realizar en 2D en Unity.

Después de jugar a los distintos prototipos y probarlos se optó por desarrollar el Throw Prototype o Killer Ball. Consideramos que se trataba de un juego con potencial, divertido y que nos permitiría obtener un resultado completo, pero con posibilidades de ampliación. El concepto del juego se tenía claro, pero aún había que definir qué se abarcaría exactamente en el proyecto. Por ello se realizó un análisis de la propuesta de juego. Algunas de las ideas expuestas a continuación no se llegaron a plasmar.

8.5. Análisis de la propuesta

A continuación, adjuntamos el estudio y análisis inicial que se hizo de la propuesta. Respondimos a varias preguntas relacionadas con la idea del juego. Se debe tener en cuenta que este análisis comprendía una lluvia de ideas de las cuales muchas no se llegaron a implementar, ya fuera por tiempo o porque se desestimaron posteriormente.

¿Cómo avanza la dificultad a medida que pasa el tiempo de juego?

Aumenta el número de pelotas en juego, a un número x de golpes (no rebotes) la pelota es destruida.

Los jugadores pueden obtener boost especiales que hagan que la pelota haga más daño al golpearla.

IA más agresiva o competente.

¿Qué consigo al golpear la pelota?

La pelota no hace daño al personaje que la golpeó por última vez, en cambio si el enemigo es golpeado con ella sufre daño. Además, se puede intentar dirigir la pelota hacia el enemigo al golpearla.

¿Cómo puedo hacer que el jugador tenga mayor control sobre la dirección de la pelota?

Además de aprovechar la superficie del jugador para dirigir el balón quizás se podría controlar también usando el teclado de dirección al mismo tiempo que se golpea. Aunque sería similar al *Lethal League,* un juego ya existente que se estudió a raíz del desarrollo de este juego.

Podría implementarse una acción de bloqueo (defender), si bloqueas el balón éste pierde fuerza y se para. Pierdes un poco de vida, pero es más fácil controlar hacia dónde quieres dirigirlo.

¿Cómo añadiría más pelotas?

Opciones:

Siempre debe haber una pelota como mínimo en juego.

Las pelotas podrían añadirse de forma aleatoria, pero con un mínimo de tiempo entre balón y balón. También podrían ser ítems que se recogen y luego los jugadores deciden cuando lanzarlos. Otra alternativa es que cada jugador tenga su "munición" inicial y decida cuando le conviene más lanzar un balón extra. Se podrían combinar varias de estas opciones.

Powerups:

Objetos para recuperar vida o para cargar energía.

Una "barrera" que se destruye con cierto número de golpes o con un golpe potente.

Distintos tipos de bolas, pelotas explosivas (explotan al primer rebote y dañan en un rango) o pelotas paralizantes (explotan al primer rebote y electrocutan-paralizan al personaje pero no hacen daño)

Estudiamos precedentes, juegos similares que ya existen:

Lethal League

El personaje se enfrenta a otro/otros golpeando una pelota. Cada vez que se golpea la pelota el movimiento se congela durante un instante antes de que la pelota sea lanzada con mayor velocidad. Si la pelota da al enemigo este pierde vida y se retira momentáneamente de la ronda. Cuando el resto de oponentes han perdido una vida empieza una nueva ronda y la pelota se asigna al jugador que perdió primero durante la última ronda.

Se usan controles de dirección además de "z" y "x", uno para picar el balón y el otro para golpearlo. Además, con los botones de dirección se puede controlar si queremos que la pelota se lance hacia arriba, hacia abajo o recta. Si se mantiene pulsado el botón "z" antes de darle al balón se da un golpe con fuerza, también se puede dar más fuerza al balón saltando y golpeando la pelota hacia abajo. Cada personaje tiene un ataque especial que se puede realizar al llenar la barra inferior y para usarlo se pulsa dos veces "z".

Gana el jugador que sigue "vivo" cuando su contrincante o contrincantes han vaciado su barra de vida.

Pokemon volleyball game

Este juego consiste básicamente en un juego de volleyball en 2D lateral, en el que dos pikachus se enfrentan jugando con una pokeball. El personaje puede moverse y saltar, además de golpear la pelota o rechazarla. Cada vez que la pelota toca la zona del rival el contrario gana un punto.

Como diferenciar nuestro juego:

Uso de varios balones a la vez. El balón queda destruido tras un número x de golpes.

El uso de bloqueo.

Es más difícil controlar con precisión la dirección de la pelota ya que no hay "pausa" en el momento del golpe.

Añadir un ataque especial que provoca un golpe potente y destruye el balón al golpear al enemigo o la pared. No se puede bloquear.

Distintos tipos de fondos, distintos terrenos: por ejemplo, un nivel con foso de agua, si la pelota cae al foso se pierde. Además, si el personaje cae al agua pierde o le resta vida hasta que salga.

9. Desarrollo del juego final

Partiendo del prototipo realizado Throw o Killer Ball pasamos a desarrollar el prototipo del juego final. Se procuró buscar inspiración en los juegos retros estudiados y se aprovecharon los conocimientos adquiridos realizando los prototipos anteriores.

9.1. Definición de los assets a utilizar

A continuación, estableceremos los assets empleados en este proyecto.

Elementos de UI de unity	Para el menú de inicio, el evento "pause" y un posible Stock gráfico: • Canvas • Paneles • Imágenes (fondo de inicio, menús, etc.) • Botones • Sliders
Gráficos	Fondos • Fondo del escenario 1 Sprites • Sprites del Jugador 1 (correr, saltar, golpear y golpear en salto) • Sprites del jugador 2 (correr, saltar, golpear y golpear en salto) • Sprites de la pelota y sus distintos estados
Scripts	• GameModeController • GameController • EnemyAI • EnemyAIMedium • EnemyAIHard • EnemyAgressiveAI • BallScript • Restart • VolumeScript
Animaciones	• Animación de movimiento para los personajes.
Sonidos	• Música ambiente del menú • Música ambiente de la partida • Quejido del jugador al ser golpeado • Sonido de la pelota al golpear un objeto

9.2. Casos de uso

Cuando un usuario ejecute el juego lo primero que aparece tras la pantalla de Unity es el menú de inicio, donde se podrán realizar las siguientes acciones:

- **Iniciar una partida nueva:** Accederemos a un nuevo menú en el que se podrá elegir el tipo de juego, un solo jugador o dos jugadores, también podremos volver al menú inicial. Una vez elegidos el juego comienza.

- **Acceder a las opciones:** Se podrán modificar las opciones de juego, en este prototipo solo se han implementado las opciones de volumen. Aceptamos para volver al menú principal.
- **Salir del juego:** cerrará la aplicación.

Durante el curso de una partida, el jugador tendrá las siguientes opciones:

- **Mover el personaje:** Con los botones de dirección correspondientes, a y d para el personaje 1 y, en caso de que hubiera 2 jugadores, los botones de dirección izquierda y derecha para el personaje 2.
- **Saltar:** En el juego se podrá realizar en cualquier momento un salto con el personaje, con el botón W para el jugador 1 y el botón de dirección superior para el jugador 2.
- **Golpear:** En cualquier momento durante el juego el jugador 1 podrá golpear la pelota con el espacio, mientras que el jugador 2 podrá hacerlo con el alt derecha.
- **Pausar:** Para pausar el juego se ha de pulsar el botón Escape, entonces se pasará a un menú de pausa con la opción a continuar, volver al menú principal o salir del juego.

A continuación, mostramos las posibles acciones del jugador en ambos casos, es decir, si el jugador está en el menú de inicio o durante el desarrollo de la partida.

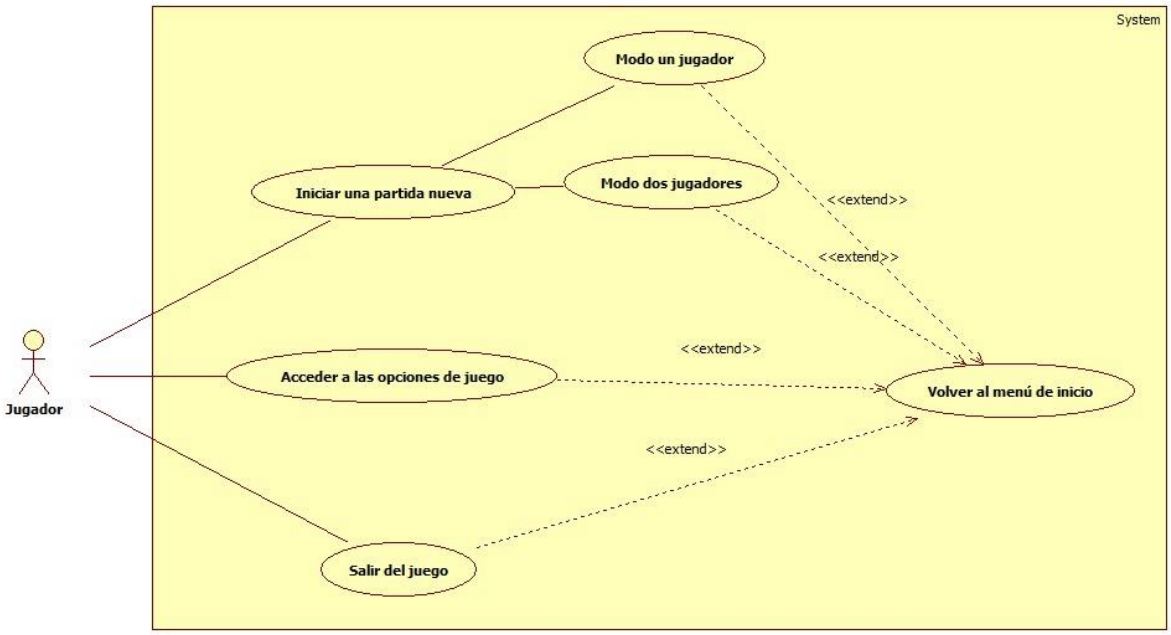


Diagrama de casos de uso para una interfaz de inicio.

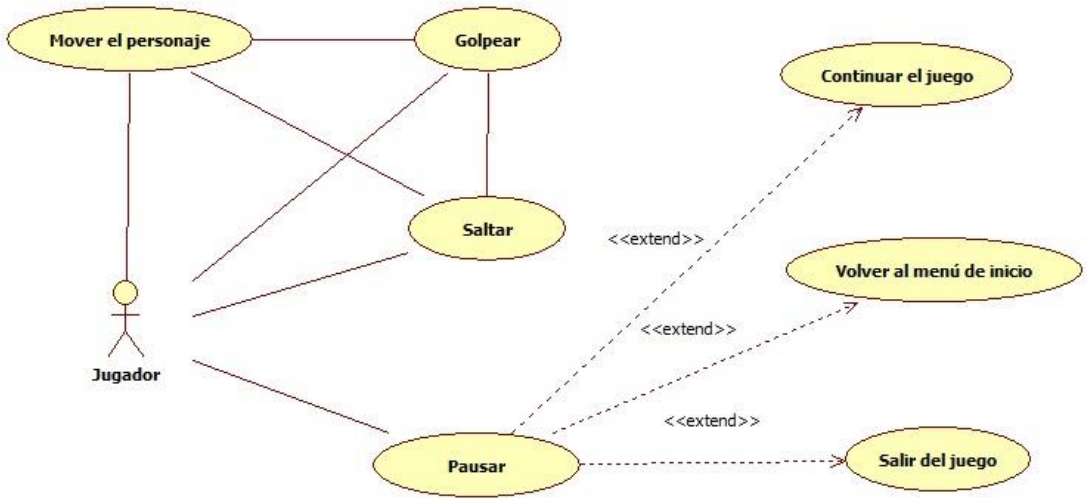
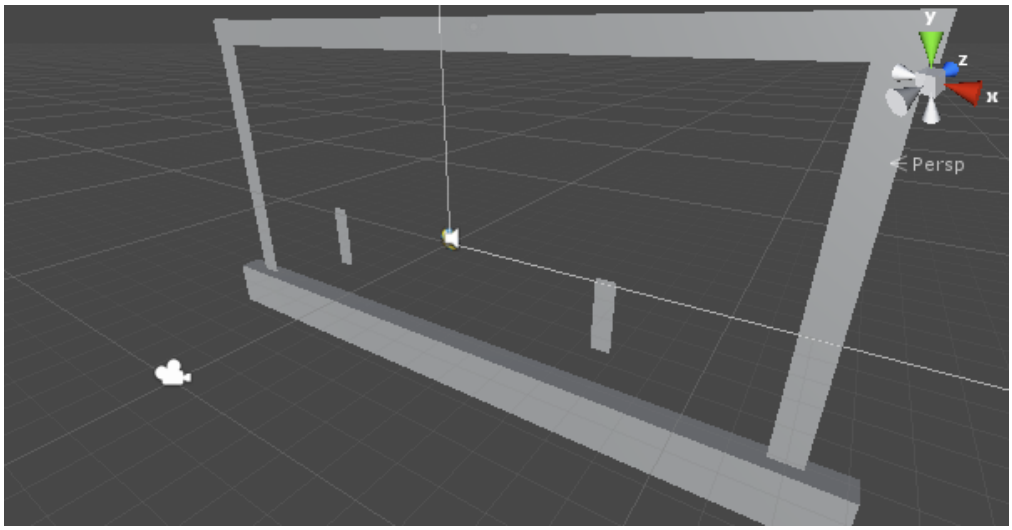


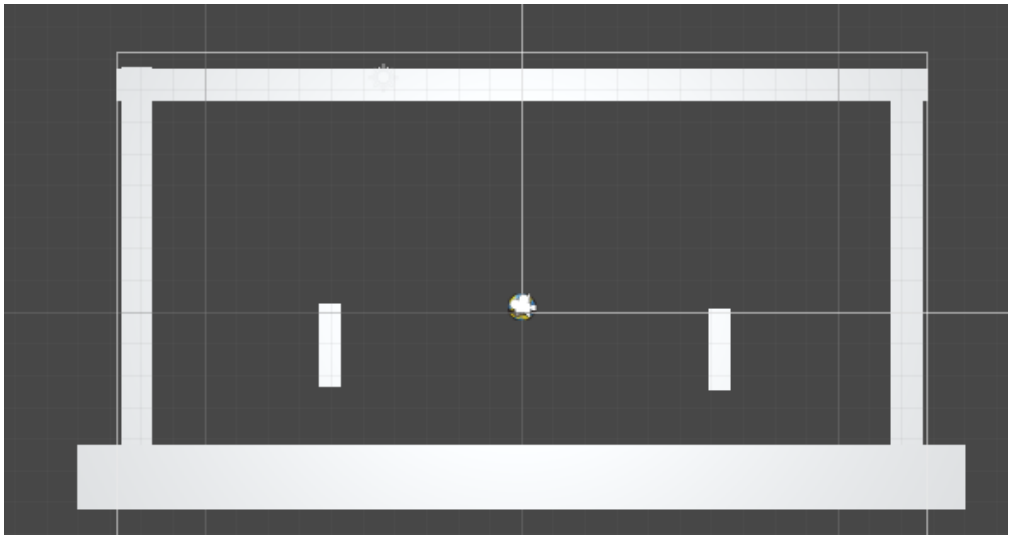
Diagrama de casos de uso durante la partida, sería el mismo para el Jugador 2.

9.3. Desarrollo del juego a partir del prototipo inicial

El prototipo desarrollado inicialmente se quedaba bastante corto respecto a la funcionalidad deseada. Se realizaron numerosas mejoras para su posterior revisión.

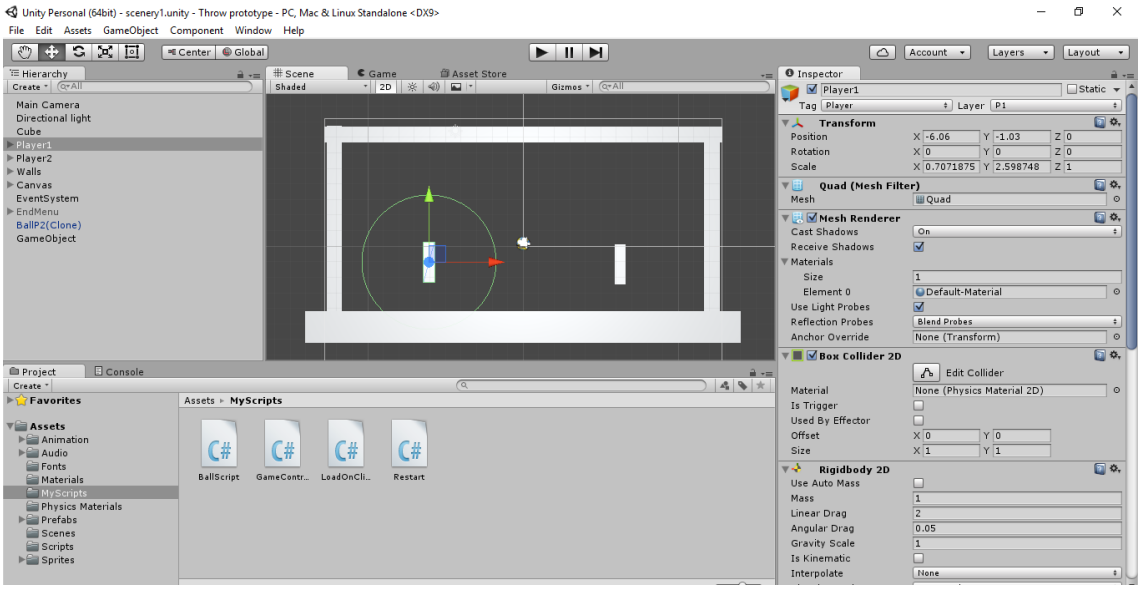
Se realizó un nuevo proyecto desde 0, reciclando parte del código anterior. Entre los cambios más destacados visualmente es que los personajes pasaron a ser un objeto de Unity, el quad. Iremos por partes hablando de los elementos iniciales del prototipo.





9.3.1. Personajes

Cada uno de los personajes es un objeto de tipo quad como se mencionó anteriormente. Se optó por utilizar un quad para los personajes puesto que este objeto es muy parecido a un plano como podemos apreciar en la captura de pantalla, pero con solo una unidad de longitud. Además, sólo tiene 2 triángulos mientras que el plano tiene 200. En este caso el quad nos sirve de placeholder para el futuro script.

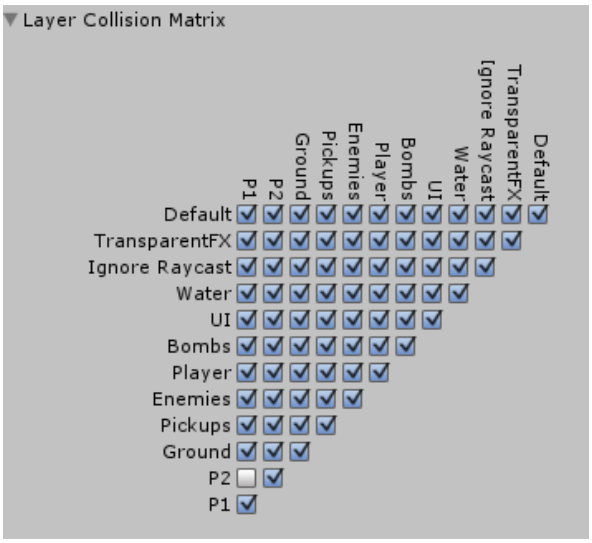


A la derecha podemos ver las características de nuestro Player1. Vemos que tiene un transform donde están las coordenadas de posición, rotación y la escala del objeto. Tenemos también la parte visual del objeto que son el quad y el mesh renderer.

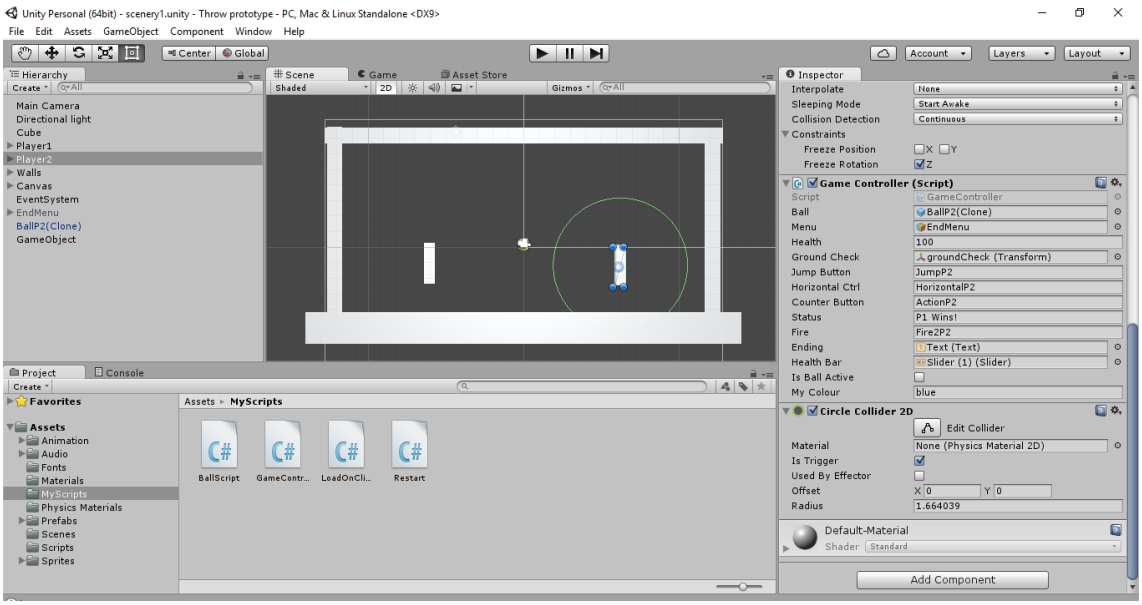
En nuestro caso también tenemos un box collider 2D, este box collider nos permitirá detectar colisiones con el objeto. Se corresponde con la figura y permite entre otras cosas que el objeto pueda “apoyarse” sobre una superficie o no salir de un recinto, como es el caso.

Inicialmente los dos personajes colisionaban entre sí, cosa que no era deseable. Se trató de utilizar el "is Trigger" para solucionarlo, pero de esta forma tampoco se encontraban las colisiones contra el suelo, por ejemplo, y el personaje caía fuera de la pantalla.

Finalmente se optó por utilizar las capas de la escena. De esta forma, al estar cada jugador en una capa distinta (P1 y P2) respectivamente, nunca chocan entre sí. Para lograr esto está la matriz de colisiones. La matriz de colisiones establece que capas colisionan entre sí y cuales se ignoran, así pues, si desmarcamos la interacción entre la capa P1 y P2, los elementos de estas capas no detectan colisiones entre sí. Por lo tanto, los dos personajes no se chocan entre sí.



Además del BoxCollider el objeto también tiene un Rigidbody 2D. Esto permite que el personaje sufra los efectos de la gravedad y adquirir fuerzas o aplicarlas, puesto que pasa a estar bajo control del motor de físicas de Unity. La principal diferencia entre un RigidBody2D y un Rigidbody es que le primero solo puede moverse por los ejes x e y. Un RigidBody2D y un Rigidbody no interactúan entre ellos.



Como podemos ver tenemos otro collider, pero esta vez se trata de un collider circular. Éste está marcado como "isTrigger" porque con él pretendemos detectar cuándo un objeto entra en su

área, pero no colisionar con él. En términos del juego este collider es el área de golpeo de la pelota. El jugador solo podrá golpear la pelota si ésta está en dicha área.

Pasaremos a hablar del script del jugador.

```
void Awake (){
    Time.timeScale = 1;
    rigidb = GetComponent<Rigidbody2D>();

    // If there is a GameModeController then we check the game mode
    // if the mode is single player we enable the AI script
    // if the mode is two players, we disable both AI script
    if (GameObject.Find ("GameModeController")) {
        gameMode = GameObject.Find ("GameModeController");
        if (gameMode.GetComponent<GameModeControllerScript>().singlePlayer == true) {
            enemy.GetComponent<EnemyAgressiveAI> ().enabled = false;
            enemy.GetComponent<EnemyAI> ().enabled = true;
        }

        if (gameMode.GetComponent<GameModeControllerScript>().twoPlayers == true) {
            enemy.GetComponent<EnemyAgressiveAI> ().enabled = false;
            enemy.GetComponent<EnemyAI> ().enabled = false;
        }
    }

}
```

En el estado awake ponemos en marcha el tiempo y asignamos el rigidbody del propio jugador a una variable. A continuación, si existe el GameModeController que establece el modo de juego, comprobamos en qué modo estamos jugando. Si es en un solo jugador activamos el script de la IA enemiga. Si jugamos en modo dos jugadores, desactivamos ambos scripts de IA.

```
// Update is called once per frame
void Update () {

    healthBar.value = health;

    grounded = Physics2D.Linecast(transform.position,
        groundCheck.position, 3 << LayerMask.NameToLayer("Ground"));

    if (Input.GetButtonDown(jumpButton) && grounded)
    {
        jump = true;

    }

    // if health is below or equal to 30, set Mode to aggressive
    // and difficult high
    if (gameMode.GetComponent<GameModeControllerScript> ().hard == true) {
        if (gameMode.GetComponent<GameModeControllerScript> ().singlePlayer == true) {
            if (enemy.GetComponent<GameController> ().health == 10f) {
                enemy.GetComponent<EnemyAgressiveAI> ().enabled = true;
                enemy.GetComponent<EnemyAI> ().enabled = false;
                enemy.GetComponent<EnemyAIEasy> ().enabled = false;
                enemy.GetComponent<EnemyAIMedium> ().enabled = false;
            }
        }
    }


}
```

En el método Update, actualizamos el valor de la barra de vida a la salud actual del personaje. También actualizamos el valor de grounded, esta variable nos permite saber si el jugador está en contacto con el suelo o no.

Comprobamos también si se ha pulsado el botón de saltar mientras se está en el suelo para evitar el doble salto. De ser así, la variable jump pasa a ser verdadera.

Se comprueba la salud del enemigo, si estamos en modo un jugador y la salud del enemigo pasa a ser 10, activamos el modo agresivo pero solo en el nivel difícil.

```
void FixedUpdate(){
    // pause menu
    if (Input.GetKey ("escape")) {
        StartCoroutine ("Pause");
    }
    //if the character is moving
    if (GetComponent<Rigidbody2D>().velocity.x != 0) {
        //Debug.Log ("velocidad no es 0");
        GetComponent<Animator>().SetBool ("IsRunning", true);
    } else {
        GetComponent<Animator>().SetBool ("IsRunning", false);
        //Debug.Log ("velocidad 0");
    }
```

En el FixedUpdate comprobamos primero si se ha pulsado la tecla de escape, para comenzar entonces con la corrutina de pausa.

Además, comprobamos si el personaje se está desplazando, para ello miramos la velocidad de su rigidBody. En caso de que se esté moviendo enviamos un mensaje al animator para que ponga

la variable IsRunning a verdadero. Esto activará la animación de correr. En caso contrario ponemos a falso la variable.

```
// Animation code
    if (this.name == "Player1") {
        //if (GetComponent<Rigidbody2D> ().velocity.x > 0) {
        if (Input.GetKey(KeyCode.D)){
            GetComponent<SpriteRenderer> ().flipX = true;
            P1IsFacingFront = true;
        }
        if (Input.GetKey(KeyCode.A)){
            GetComponent<SpriteRenderer> ().flipX = false;
            P1IsFacingFront = false;
        }
    }
    if (this.name == "Player2") {
        if (Input.GetKey(KeyCode.LeftArrow)){
            GetComponent<SpriteRenderer> ().flipX = false;
            P2IsFacingFront = true;
        }
        if (Input.GetKey(KeyCode.RightArrow)){
            GetComponent<SpriteRenderer> ().flipX = true;
            P2IsFacingFront = false;
        }
    }
```

A continuación, determinamos el jugador del que se trata y la tecla pulsada para voltear el sprite según el personaje mire hacia la izquierda o la derecha.

```
// if the ball is to the right of player1 and the player1 is facing front, ball is i nfront
// else if ball is to the left and player is not facing front, ball is in front.
if (this.name == "Player1") {
    if (ball.GetComponent<Rigidbody2D> ().position.x < transform.position.x && !P1IsFacingFront ||
        ball.GetComponent<Rigidbody2D> ().position.x > transform.position.x && P1IsFacingFront){
        counterable = true;
    } else  {
        counterable = false;
    }
}

if (this.name == "Player2") {
    if (ball.GetComponent<Rigidbody2D> ().position.x > transform.position.x && !P2IsFacingFront ||
        ball.GetComponent<Rigidbody2D> ().position.x < transform.position.x && P2IsFacingFront){
        counterable = true;
    } else  {
        counterable = false;
    }
}
```

Comprobamos si el jugador está mirando en dirección a la pelota, en caso afirmativo ponemos la variable counterable a verdadero, es decir, puede golpearse. En caso negativo ponemos la variable a falso y no se podrá golpear la pelota. No es deseable que el personaje pueda golpear de espaldas, aunque inicialmente se permitía.

```
float h = Input.GetAxis (horizontalCtrl);

if (h * rigidb.velocity.x < maxSpeed) {
    rigidb.AddForce (Vector2.right * h * moveForce);
}

if (Mathf.Abs (rigidb.velocity.x) > maxSpeed){
    rigidb.velocity = new Vector2 (Mathf.Sign (rigidb.velocity.x) * maxSpeed, rigidb.velocity.y);
}

if (Input.GetButtonDown (counterButton) && canHit == true) {

    GetComponent<Animator> ().SetTrigger ("Punch");
    if (inradius && counterable) {

        isBallActive = true;
        Counter ();
    }

    StartCoroutine ("Wait");

}
```

En una variable asignamos el valor del eje horizontal (será diferente según el jugador). Si la velocidad en ese eje, es menor que la velocidad máxima, se le añade una fuerza de movimiento.

Si la velocidad redondeada es mayor que la velocidad máxima permitida, asignamos a la velocidad del rigidbody un nuevo valor por debajo de la velocidad máxima. Esto lo hacemos para ajustar la velocidad en caso de que tenga poca o demasiada.

Si se ha pulsado el botón de contrataque y se puede golpear al enemigo, enviamos un mensaje al animator para que ejecute la animación del puñetazo. Además, si la pelota está en el rango de golpeo y además está frente al jugador, llamamos a la función counter(). Una vezterminado comenzamos la corrutina Wait. Esta corrutina evita que el jugador pueda golpear constantemente.

```
    if (jump) {
        rigidb.AddForce (new Vector2 (0f, jumpForce));
        jump = false;
    }


    if (grounded) {
        GetComponent<Animator> ().SetBool ("Jump", false);
    } else {
        GetComponent<Animator> ().SetBool ("Jump", true);
    }


    // if the ball is slow
    if (ball.GetComponent<Rigidbody2D> ().velocity.magnitude < 10f) {
        //print ("soy verde");
        ballStatus = "green";
        ball.GetComponent<SpriteRenderer> ().sprite = ballGreenSprite;

    }

}
```

Comprobamos si el personaje está saltando, de ser así le asignamos una fuerza y ponemos el jump a falso. Esto significa que después de saltar no se podrá saltar de nuevo.

Por otra parte, si el personaje no estuviera en el suelo enviamos un mensaje al animator para que ejecute la animación de saltar o regrese al estado ocioso en caso de tocar el suelo.

Finalmente terminamos el FixedUpdate comprobando si la velocidad de la pelota es menor que 10. De ser así la pelota pasa a estado neutro.

```
// method to know if the ball is in the area
void OnTriggerEnter2D (Collider2D other) {
    if (other.tag == "Ball") {
        inradius = true;
        //print ("estoy en radio");

    }
}

// method to know if the ball is in the area
void OnTriggerExit2D (Collider2D other) {
    if (other.tag == "Ball") {
        inradius = false;
        //print ("im out");
    }
}

// method to know if the ball hit the player collider
void OnCollisionEnter2D (Collision2D col) {
    if (col.gameObject.tag == "Ball") {
        ballSpeed = ball.GetComponent<Rigidbody2D> ().velocity.magnitude;
        LoseHealth ();
    }
}
```

Con el OnTriggerEnter2D podemos averiguar cuando la pelota está en el área de golpeo de un jugador. El OnTriggerExit2D por otra parte nos da información sobre cuando la pelota deja de estar en el área. Con una variable actualizamos el estado de la pelota cuando se produce uno de estos casos.

Por otra parte, tenemos el OnCollisionEnter2D que nos permite saber cuándo la pelota golpea el collider interno del jugador. En este caso guardamos el valor de velocidad de la pelota y llamamos al método LoseHealth del que hablaremos más adelante.

```
void Counter () {
    if (gameObject.name == "Player1") {
        ballStatus = "blue";
        ball.GetComponent<SpriteRenderer> ().sprite = ballP1Sprite;
        //print ("soy azul");
    }
    if (gameObject.name == "Player2") {
        ballStatus = "red";
        ball.GetComponent<SpriteRenderer> ().sprite = ballP2Sprite;
        //print ("soy rojo");
    }
    ball.GetComponent<AudioSource> ().Play();

    ball.GetComponent<Rigidbody2D> ().velocity =
        Vector3.ClampMagnitude (ball.GetComponent<Rigidbody2D> ().velocity, maxForward);
    ball.GetComponent<Rigidbody2D> ().AddForce (
        (ball.transform.position - transform.position) * 30, ForceMode2D.Impulse);
}
```

En el método Counter comprobamos inicialmente el jugador, si es el jugador 1 la pelota pasa a tener el estado azul, cambiándose también su sprite. En cambio, el jugador 2 pasa a tener el estado rojo.

Se reproduce el sonido del golpe a la pelota.

También se asigna la velocidad y la fuerza a la pelota en la dirección correspondiente, creando el efecto de golpeo deseado.

El método LoseHealth lo describiremos más adelante.

```
IEnumerator Reload() {
    panel.SetActive (true);
    Time.timeScale = 0.00001f;
    ending.text = status;
    float pauseEndTime = Time.realtimeSinceStartup + 5;
    while (Time.realtimeSinceStartup < pauseEndTime)
    {
        yield return 0;
    }
    Time.timeScale = 1;
    menu.SetActive(false);
    Application.LoadLevel(Application.loadedLevel);
}

IEnumerator Pause() {
    pausePanel.SetActive (true);
    Time.timeScale = 0.00001f;
    while (pausePanel.activeSelf == true) {
        yield return 0;
    }
    Time.timeScale = 1;

}
```

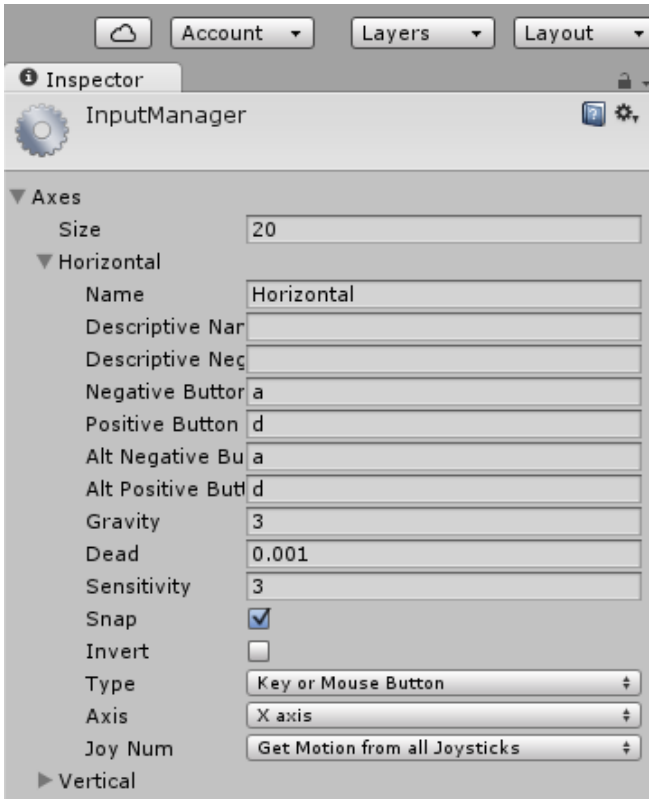
La corrutina Reload es el menú final que se activa cuando uno de los personajes muere. Se activa un panel secundario con opción a volver a jugar, ir al menú de inicio o salir del juego. El tiempo de juego se detiene. Tras 5 segundos se arranca el tiempo se retira el panel y se recarga el nivel.

El caso de Pause es similar, activamos el panel de pausa con su menú correspondiente y se reactiva le tiempo cuando el jugador haya elegido una de las opciones.

```
IEnumerator Invulnerability() {
    hit = true;
    yield return new WaitForSeconds (3f);
    hit = false;
}


IEnumerator Wait()  {
    print ("waiting");
    canHit = false;
    yield return new WaitForSeconds(0.5f);
    canHit = true;

}
```

Invulnerability hace invulnerable al jugador durante 3 segundos después de haber sido golpeado para evitar que se detecten múltiples colisiones. El Wait por otra parte evita que el jugador esté permanentemente golpeando y deja un tiempo de 0.5 segundos entre golpe y golpe.

La interfaz de Unity nos permite asignar teclas o joystick, por ejemplo, a los distintos ejes de movimiento. Así por tanto en el eje que hemos llamado Horizontal, tenemos distribuidos los botones negativos (ir hacia el eje x negativo, o positivo, ir hacia el eje x positivo), en este caso esto se traduciría en desplazamientos horizontales hacia izquierda o derecha.

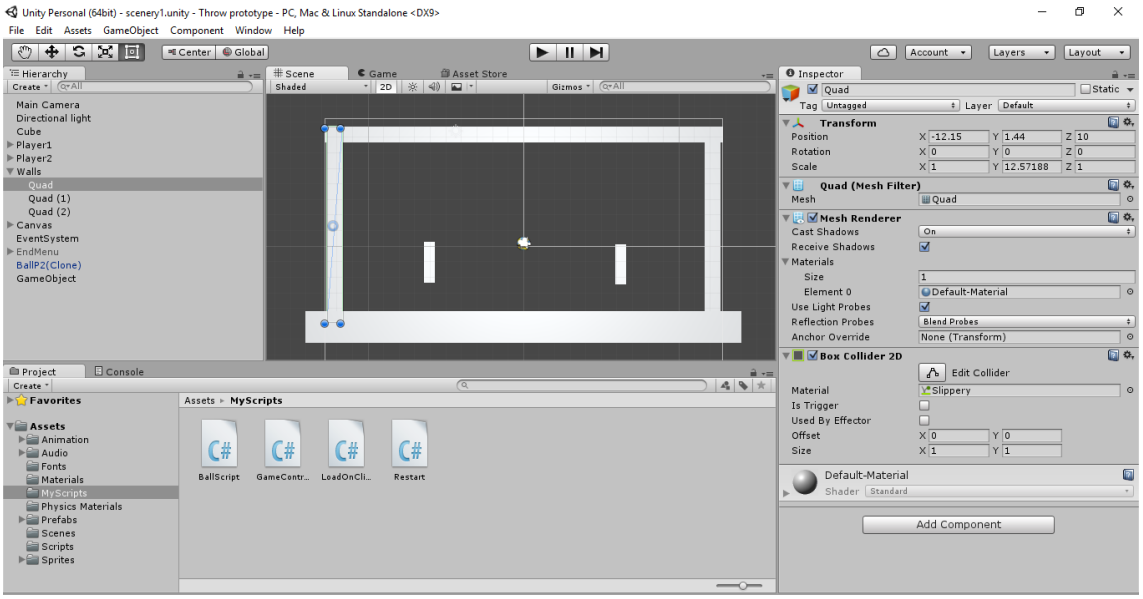


Una vez explicado esto, el Input.GetAxis (“nombre del eje”) nos permite saber por ejemplo si el personaje se está desplazando en alguno de los ejes. De forma que se puede utilizar en lugar de GetButton o GetKey.

9.3.2. Escenario

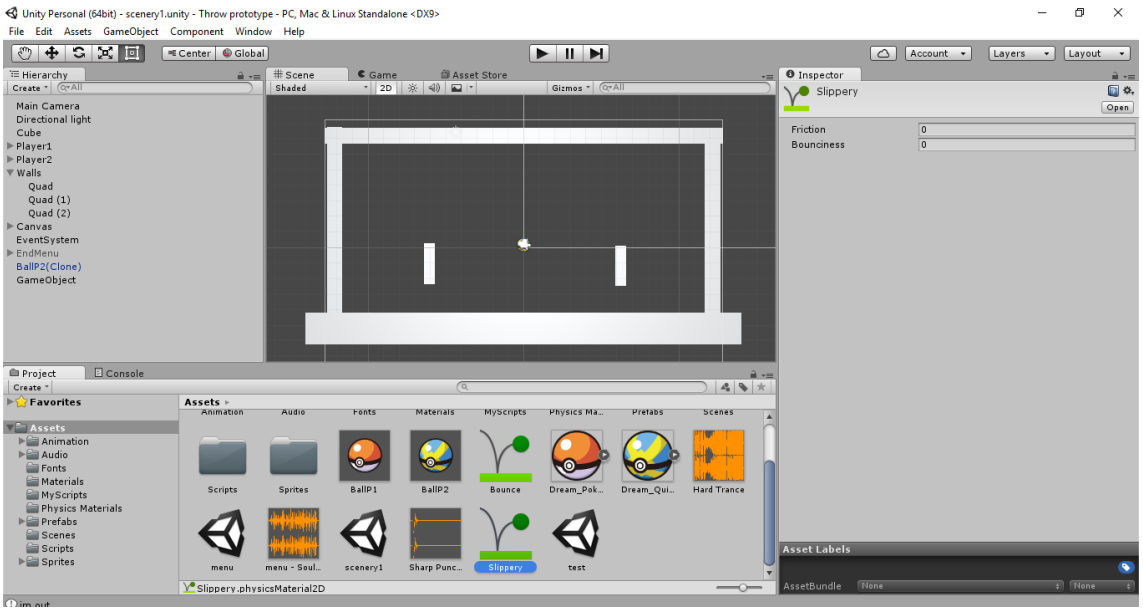
Hablaremos ahora de los elementos del escenario. Tenemos un recinto cerrado con suelo, techo y paredes. En este caso ningún objeto tendrá un Rigidbody porque esto supondría que los objetos estarían sometidos a las físicas y caerían. Al no suceder de esta forma, los elementos “flotan” en la posición que se les ha asignado.

Todos los objetos de este escenario cuentan con un Box Collider 2D para detectar colisiones con los jugadores. De esta forma los jugadores no podrán atravesar paredes o caerse por el suelo.



El resto de elementos son similares a los descritos previamente.

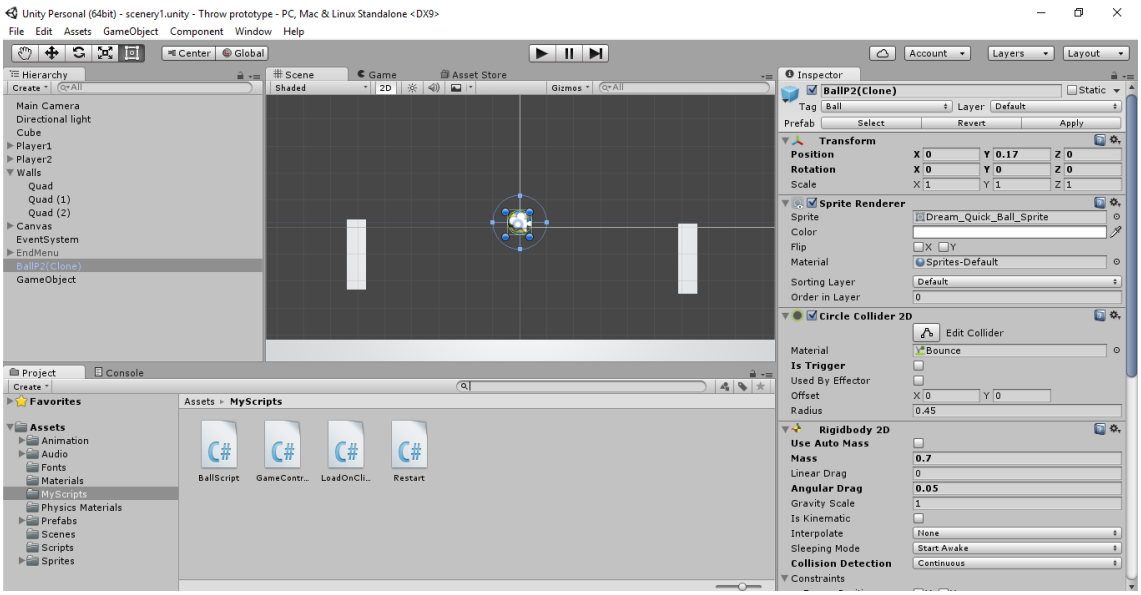
Cabe destacar que el material de las paredes (no del suelo o el techo) es Slippery, resbaladizo, esto es debido a que inicialmente cuando los jugadores saltaban y chocaban contra las paredes quedaban “adheridos” a ellas y caían muy lentamente, debido al rozamiento.

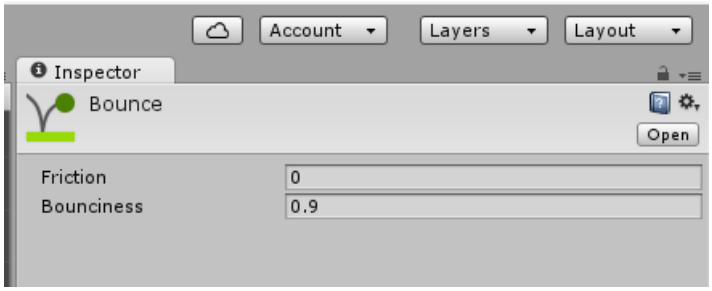


El material Slippery tiene una fricción de 0 y una capacidad de rebote de 0 también. De esta forma conseguimos que los jugadores al saltar no pierdan velocidad al chocarse con la pared ni reboten contra ella.

9.3.3. Pelota

Nuestra pelota es similar a los jugadores en cuanto a sus componentes. En este caso la pelota es un sprite que tiene un collider circular con un material concreto. En este caso bounce (rebote).





El material bounce tiene una capacidad de rebote de 0.9, a 1 el objeto no pierde fuerza y rebota permanentemente y en 0 no rebota en absoluto. Con 0.9 la pelota pierde lentamente su fuerza hasta dejar de botar.

La pelota también tiene un RigidBody por lo que es susceptible a físicas.

El código de la pelota es muy sencillo y simplemente establece la velocidad máxima de la misma.

```
public class BallScript : MonoBehaviour {
    private float maxSpeed;
    public float speed;



    // Use this for initialization
    void Start () {
        maxSpeed = 30f;
    }

    // Update is called once per frame
    void Update () {
        //Speed Control
        if(GetComponent<Rigidbody2D>().velocity.magnitude > maxSpeed){
            GetComponent<Rigidbody2D>().velocity =
                Vector3.ClampMagnitude(GetComponent<Rigidbody2D>().velocity, maxSpeed);
        }

        speed = GetComponent<Rigidbody2D>().velocity.magnitude;
        //print (speed);
    }

}
```

A continuación, se hablará de las mecánicas y de las mejoras que se desarrollaron posteriormente.

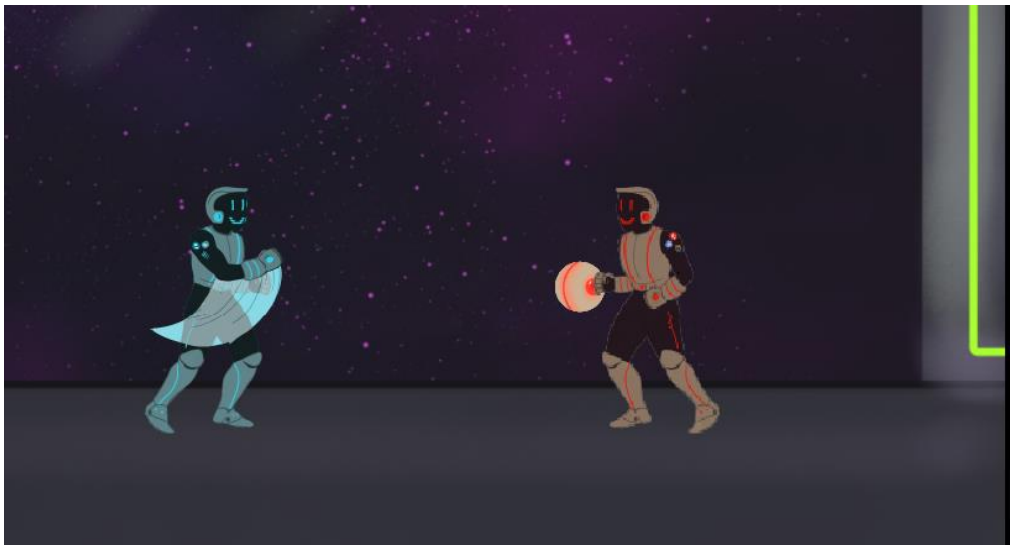
El salto de un personaje en un juego 2D lateral es importante puesto que marca los tiempos y la posición del jugador en cada momento. Inicialmente el salto que codificamos resultaba poco vistoso además de que tenía algunos fallos de comportamiento. Se mejoró el salto utilizando un tipo de salto mucho más fluido y con la misma fuerza.

9.3.4. Golpe de la pelota

Cuando la pelota está en el área del jugador este puede golpear la pelota para devolverla e intentar golpear con ella a su enemigo, además es una forma de defenderse. Inicialmente

habíamos planteado la opción de que el jugador pudiera controlar la dirección en la que se devolvía la pelota, pero finalmente se optó por realizarlo de otra forma.

En nuestro caso la dirección en la que se envía la pelota se calcula encontrando el vector de dirección entre la pelota y el jugador en el momento del golpe.



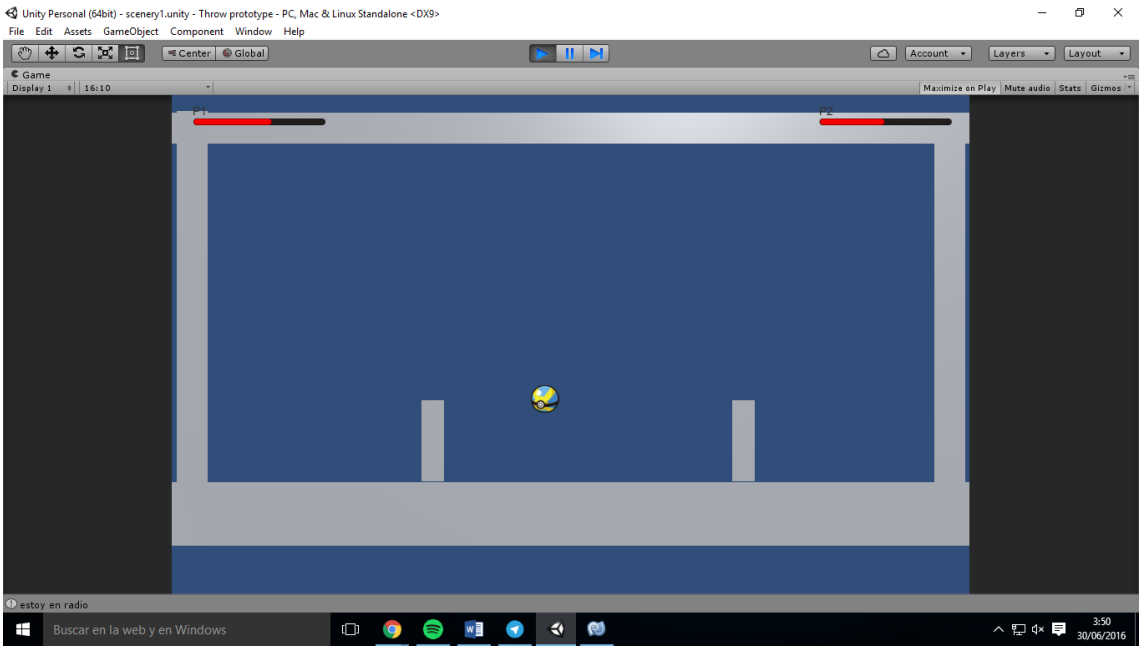
9.3.5. Condiciones de victoria y derrota.

La condición de victoria es que el enemigo tenga su barra de vida a 0. En el caso contrario, la condición de derrota es que la vida del jugador llegue a 0. En ambos casos el juego se reinicia a los pocos segundos después de mostrar un mensaje de victoria del jugador correspondiente.

En el primer prototipo la vida no se mostraba en la interfaz con barras de vida.

9.3.6. Pelota, 3 estados, 3 colores y daño según el color.

Inicialmente la pelota solo tenía un estado visualmente, y dos internamente según quién hubiera golpeado la misma la última vez. Es decir, si el jugador 1 había golpeado la pelota ésta no le hacía daño a sí mismo, pero sí al contrario y viceversa.



En iteraciones posteriores se pasó a modificar la pelota de forma que tuviera tres estados, azul, roja y verde. La pelota se vuelve azul cuando el jugador 1 la golpea, roja cuando el jugador 2 hace lo propio y verde cuando la pelota pierde suficiente fuerza como para no suponer una amenaza (no resta vida en este estado).

9.3.7. Barras de vida

Barras de vida se restan según la velocidad de la pelota en lugar de siempre un daño fijo. Mediante el uso de un canvas se añadieron barras de vida. Las barras de vida se implementaron con un slider y se actualizan según la variable correspondiente en código.

Posteriormente se decidió que según la velocidad de la pelota se restaría más o menos vida. La barra de vida tiene 100 puntos y se va restando de 10 en 10. En caso de tratarse de un golpe a alta velocidad se le restan 30 puntos.

```
void LoseHealth () {
    if (hit == false) {
        if ((myColour != ballStatus) && (ballStatus != "green")) {
            if (ballSpeed > 30f) {
                health -= 30f;
            } else {
                health -= 10f;
            }
            Invulnerability ();
            GetComponent<AudioSource> ().Play ();

            if (health < 0) {
                health = 0;
            }
            if (health == 0) {
                //Debug.Break ();
                StartCoroutine (Reload ());
            }
        }
    }
}
```

En este código podemos ver que contamos con la variable hit que nos permite saber si se puede restar vida o no (si el personaje es o no invulnerable).

Entonces se determina si el estado de la pelota no corresponde con el color del jugador y si el estado de la pelota es distinto de verde. Pues si no se cumpliera alguna de estas condiciones la pelota no haría daño.

Comparamos entonces la velocidad de la pelota durante el golpe, si es mayor de 30 restamos 30 de vida y el personaje pasa a ser invulnerable unos segundos. Esto evita que se detecten múltiples colisiones en un solo contacto y por lo tanto se reste vida varias veces por un solo golpe. También reproducimos el sonido del impacto.

Si la pelota es más lenta el código es similar, pero restamos menos vida.

Comprobamos además cuanta salud le queda al personaje, si es menor que 0 la ponemos a 0, si es igual a 0 recargamos el juego comenzando una corrutina.

La velocidad de la pelota se ha ido ajustando en distintas iteraciones del proyecto hasta el resultado actual.

También se estableció un límite de velocidad para la pelota para evitar que la pelota cogiera demasiada velocidad en determinadas circunstancias.

9.3.8. Pruebas y feedback

El juego fue probado por varios compañeros y compañeras durante la Ludum Dare. El feedback fue mayoritariamente positivo, aunque se propusieron mejoras.

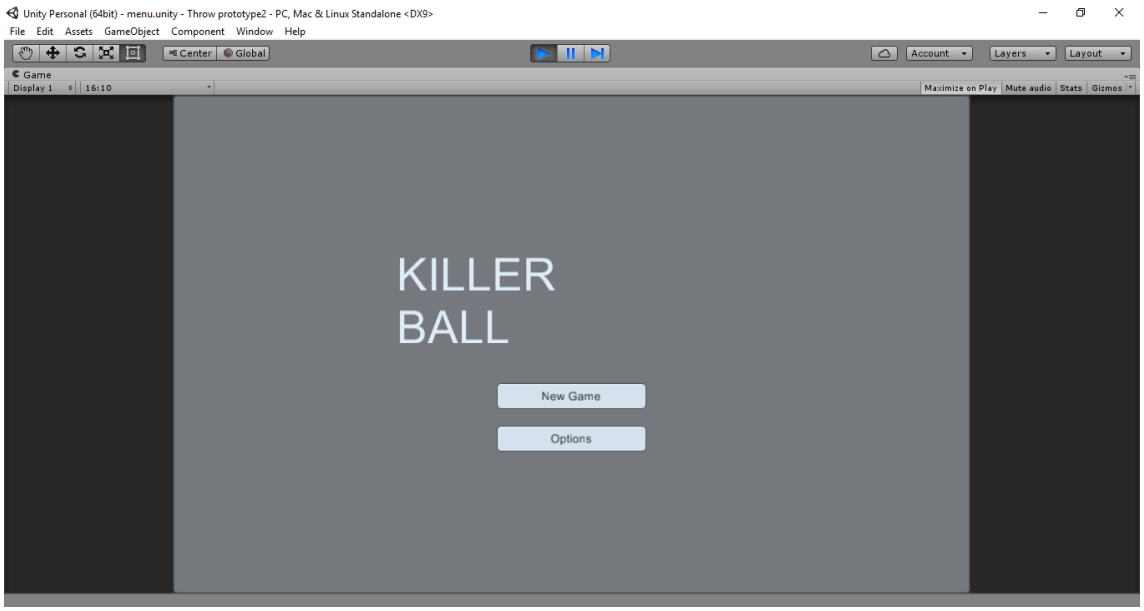
La principal dificultad se encontraba en los controles diseñados inicialmente. Los controles estaban diseñados para probarse por una sola persona, cada uno de los controles se alcanzaban con una mano. No obstante, la mayoría de los jugadores encontraron dificultad en jugar con una

sola mano y trataban de usar las dos. Por lo tanto, se propuso cambiar los controles para facilitar que dos jugadores pudieran jugar a la vez con un mismo teclado sin entorpecerse.

Otro detalle es que en esta versión temprana del juego los jugadores no se distinguían entre sí, especialmente después de haberse movido por el entorno. Puesto que ambos eran idénticos. Esto se solucionó temporalmente coloreando la apariencia de los personajes de azul y rojo respectivamente.

9.3.9. Audio, menús y nueva vista de derrota o victoria

Se añadió un menú principal en el que se puede elegir si empezar un juego nuevo o modificar las opciones de juego. En este caso las opciones de juego consisten solo en regular el volumen general del juego.



Para esto utilizamos de nuevo un slider, en este caso contamos con la ayuda de un Script que nos permite recoger el valor asignado por el jugador en el slider y traducirlo en el volumen de juego. Por lo tanto, a la derecha del todo tendremos el volumen al máximo y si lo desplazáramos a la izquierda del todo pondríamos el juego en silencio.

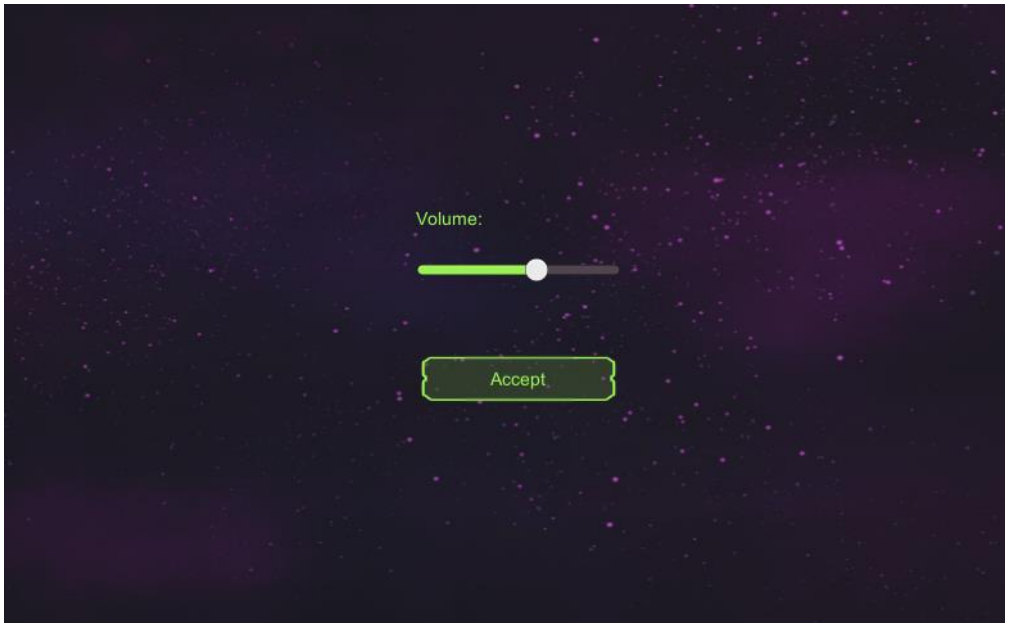
Script de audio

```
public class VolumeScript : MonoBehaviour {
    public static float volumeValue;
    public Slider volumeSlider;

    public static VolumeScript volumeScriptInstance;
    void Awake(){
        DontDestroyOnLoad (gameObject);
        volumeSlider.value = AudioListener.volume;
    }

    void Update () {
        volumeValue = volumeSlider.value;
    }

    public void OnGUI()
    {
//      //print (volumeSlider.value);
        AudioListener.volume = volumeSlider.value;

    }
```

En el script del volumen, utilizamos el DontDestroyOnLoad en el objeto que lo contiene. De esta forma este objeto, con su respectivo script, se mantendrá en las distintas escenas que carguemos. En el Awake asignamos también al slider de volumen el valor del volumen del juego. De esta forma siempre que se ejecute el script el volumen se actualizará de acuerdo al volumen actual.

En el update asignamos el valor del slider a una variable, de esta forma cuando se cliquemos el botón de aceptar asignamos el último valor del slider a la variable general de audio. Cambiando el volumen general del juego.

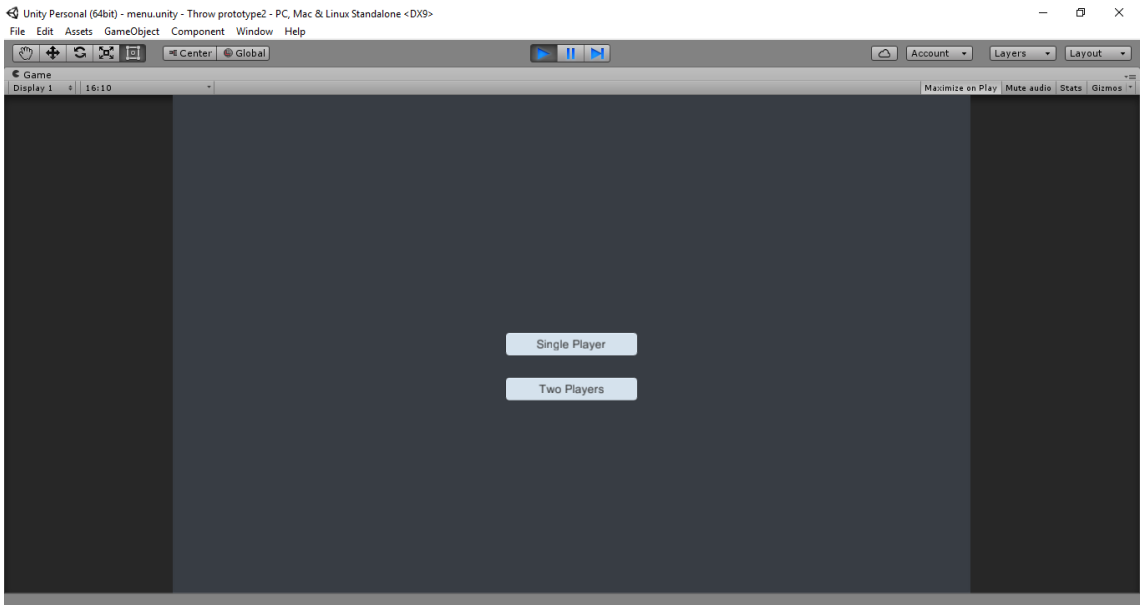
En este juego el audio consiste en la música de los menús, así como los golpes de la pelota al rebotar o el quejido de los jugadores al recibir un impacto de la pelota.



Captura del juego con los gráficos actualizados.

9.3.10. Menús

Una vez hayamos hecho click en el menú de juego nuevo aparecerá otro menú que nos permitirá elegir si queremos jugar con un jugador o dos jugadores. El modo un jugador aún no está implementado puesto que para ello primero tenemos que desarrollar una IA, tema del que hablaremos en el próximo apartado.



Para asociar métodos a botones en Unity es necesario crear un objeto que tenga asociado en el juego ese script, el objeto se asigna después a un botón y se elige la función correspondiente. Esto significa que cuando el botón se pulse, la función se pondrá en marcha.

En todos los menús usamos un mismo script con las funciones de los botones, puesto que en muchos casos (Menú principal o Salir del juego, por ejemplo) las funciones son las mismas.

El script consta de los siguientes métodos:

```
    public void playingMode(){
        //Application.LoadLevel (1);
        panelSec.SetActive(true);
        panelMain.SetActive(false);
    }

    public void goBack(){
        //Application.LoadLevel (1);
        panelSec.SetActive(false);
        panelMain.SetActive(true);
    }

    public void singlePlayer(){
        gameMode.SendMessage ("setToSingle");
        SceneManager.LoadScene (1);
    }


    public void twoPlayerGame(){
        gameMode.SendMessage ("setToPair");
        SceneManager.LoadScene (1);
    }

    public void optionsMenu(){
        SceneManager.LoadScene (2);
    }

    public void mainMenu(){
        SceneManager.LoadScene (0);
    }


     public void restart(){
         Application.LoadLevel (Application.loadedLevel);
     }

     public void hidePanel () {
         pausePanel.SetActive(false);
         Time.timeScale = 1;
     }

     public void quit () {
         Application.Quit ();
     }
}
```

Los distintos menús en Unity se pueden realizar creando escenas nuevas o mediante paneles. En este caso se ha hecho uso de paneles, uno de los cuales está desactivado por defecto. Al hacer click en “Juego nuevo” activamos el método playingMode que desactiva el panel principal y activa el secundario, que consiste en un menú en el que se puede seleccionar el modo de juego.

El método goBack realiza justo la función contraria. También tenemos funciones para los modos de juego, en un jugador tenemos que mandar un mensaje al controlador de modo de juego, para avisar de que se jugará con un solo jugador. Acto seguido se carga el nivel, en este caso corresponde con el número 1. En el modo dos jugadores la acción es similar, advirtiendo también al controlador que se va a jugar con dos jugadores.

De vuelta al menú principal podemos acceder a opciones o salir del juego con el método quit. También tenemos otros métodos similares relevantes en el menú de pausa como el restart.

Respecto al controlador de modo de juego, tiene un código muy simple y destaca el método DontDestroyOnLoad que nos permite mantener la información de este script entre escenas o niveles.

```
void Awake () {
    if (!created) {
        DontDestroyOnLoad (this.gameObject);
        created = true;

    } else {
        Destroy (this.gameObject);
    }
}
```

```
void setToPair () {
    twoPlayers = true;
    singlePlayer = false;
    easy = false;
    medium = false;
    hard = false;
}

void setToSingleEasy () {
    singlePlayer = true;
    twoPlayers = false;
    easy = true;
    medium = false;
    hard = false;
}

void setToSingleMedium () {
    singlePlayer = true;
    twoPlayers = false;
    easy = false;
    medium = true;
    hard = false;
}

void setToSingleHard () {
    singlePlayer = true;
    twoPlayers = false;
    easy = false;
    medium = false;
    hard = true;
}
```

Así pues, según el estado de las variables al comienzo del juego, es establecerá el modo de juego y la dificultad.

9.4. Desarrollo de la IA enemiga.

Hasta ahora el juego se había desarrollado para dos jugadores, puesto que es más sencillo realizar un prototipo de esta forma. No obstante, la idea era tener varios modos de juego, entre ellos el de un solo jugador y poder enfrentarse a una IA enemiga. Para ello se estudiaron varias fuentes sobre Inteligencia Artificial y se desarrolló una máquina de estados.

La máquina de estados consta de 5 estados, Wait, FindBall, AvoidBall, HitBall y CheckBallPos. En el start establecemos el estado, en este caso FindBall y arrancamos la corrutina de la máquina de estados.

```
//
enum AIenum
{
    Wait,
    FindBall,
    AvoidBall,
    HitBall,
    CheckBallPos,
    StateMachine
};

private AIenum state;
public float seconds;

void Awake (){
    Time.timeScale = 1;
    // ajustando los segundos podemos conseguir que el enemig
    seconds = 0.5f;
    rigidb = GetComponent<Rigidbody2D>();

}
// Use this for initialization
void Start()
{
    state = AIenum.FindBall;
    StartCoroutine(FSM());
}
```

FindBall primero establece la velocidad de la pelota, si la pelota es muy rápida y pertenece al rival, entonces se lanza la corrutina AvoidBall. En caso contrario se busca la posición futura de la pelota mediante una función y si la pelota está en el aire, y el jugador está en el suelo, la variable jump se activa, y el jugador salta al dirigirse hacia la pelota. También se envía un mensaje al componente animator para activar el trigger Jump, de forma que la correspondiente animación tenga lugar. Si por otra parte simplemente el jugador está en el suelo, aplicamos una fuerza al mismo para que se desplace, pero sin activar la variable del salto.

```
IEnumerator FindBall()
{
    print ( "go");
    // if the ball is fast and is the enemy ball
    if (gamectrl.ball.GetComponent<Rigidbody2D> ().velocity.magnitude > 20f
        && ballStatus == "blue") {

        state = AIenum.AvoidBall;
        yield return null;
    }
    //actual position of the ball

    speed = new Vector3 (ball.transform.position.x - transform.position.x,
        ball.transform.position.y - transform.position.y);

    // if future ball position is high, jump and go close, else go close

    if (FutureBallPosition ().y > 1.3f && gamectrl.grounded == true) {
        gamectrl.jump = true;
        GetComponent<Animator> ().SetTrigger ("Jump");
    }
    GetComponent<Rigidbody2D>().AddForce(FutureBallPosition() * 5f);

    if (ball.transform.position.x - transform.position.x > 0) {
        GetComponent<SpriteRenderer> ().flipX = true;
    }
    if (ball.transform.position.x - transform.position.x < 0) {
        GetComponent<SpriteRenderer> ().flipX = false;
    }
```

También tenemos en cuenta que, según la posición de la pelota, el sprite tiene que orientarse a una dirección u otra.

```
    if (gamectrl.inradius == true) {
        print (" en el radio prueba");
        //if the ball is low and the player is grounded,
        // jump to position itself pass the ball

        state = AIenum.CheckBallPos;
    }
    yield return null;
}
```

Para finalizar si encontramos la pelota en el radio del personaje, el estado pasa a ser CheckBallPos. Si no nos mantenemos en el mismo estado.

A continuación, hablaremos del estado HitBall. Para empezar, establecemos un porcentaje de error, en este caso del 30%. En caso de que tenga éxito, se activa el trigger del animator para que realice la animación correspondiente y se envía un mensaje al controlador de juego para que realice la función “Counter”, que golpea la pelota.

Los distintos niveles de dificultad juegan con este porcentaje de acierto. Además, contamos con la IA agresiva, ésta se diferencia en las demás en que no le importa la posición del jugador para

golpear la pelota, si ésta está en su rango intentará golpearla. Este modo solo se activa con la dificultad al máximo cuando al enemigo le queda poca vida.

El estado pasa a ser wait para evitar golpes múltiples.

```
IEnumerator HitBall()
{
    //print ( "hit ball");
    //added error 0.3
    int random = Random.Range (0, 9);
    if (random < 7) {
        GetComponent<Animator> ().SetTrigger ("Punch");
        gamectrl.SendMessage ("Counter");
    }
    state = AIenum.Wait;
    yield return null;
}
```

```
IEnumerator AvoidBall()
{
    print ( "run from ball");
    //gamectrl.SendMessage ("RunAway");
    speed = new Vector3 (transform.position.y - ball.transform.position.y,
        transform.position.x - ball.transform.position.x);

    if (FutureBallPosition ().y < 1.3f && gamectrl.grounded == true) {
        gamectrl.jump = true;
        GetComponent<Animator> ().SetTrigger ("Jump");
    }
    GetComponent<Rigidbody2D>().AddForce(FutureBallPosition() * 5f * -1);

    state = AIenum.FindBall;
    yield return null;
}
```

Con el AvoidBall desplazamos el personaje en dirección opuesta a donde se encuentra la pelota.

```
    IEnumerator CheckBallPos () {

        if (isPlayerInFront ()) {
            state = AIenum.HitBall;
        } else {

            //state = AIenum.PositionYouself;
            if (ball.transform.position.y < -1.60 && gamectrl.grounded == true) {
                print ("salto para pasar por encima de la pelota");
                gamectrl.jump = true;
                state = AIenum.FindBall;
            }
            state = AIenum.FindBall;
        }

        yield return null;
    }
```

Con el CheckBallPos llamamos a la función isPlayerInFront que nos permite saber si la pelota está entre el jugador enemigo y la IA. En caso afirmativo cambiamos el estado a HitBall. Si no es

así, si la pelota está en el suelo saltamos para pasar al otro lado para evitar que el propio personaje empuje la pelota contra la pared y se quede atascado. Si la pelota está botando simplemente cambiamos el estado a FindBall.

```
IEnumerator Wait()
{
    print ( "wait");


    yield return new WaitForSeconds(seconds);

    if ((ball.transform.position - transform.position).magnitude < 20f) {
        state = AIenum.FindBall;
    } else if ((ball.transform.position - transform.position).magnitude < 1f) {
        state = AIenum.AvoidBall;
    }
}
```

En el estado Wait, hacemos una llamada a la función WaitForSeconds. En este caso esperamos 0.5. Pasado este tiempo establecemos según la velocidad de la pelota si debemos buscarla o evitarla.

```
Vector3 FutureBallPosition () {
    // x = ut + 1/2at^2, where x = distance, a = friction
    // u = start velocity

    //ut

    Vector3 ut = ball.GetComponent<Rigidbody2D>().velocity *
        Time.deltaTime;
    float half_a_t_squared = 0.5f * 1 * Time.deltaTime * Time.deltaTime;
    Vector3 scalarToVector = half_a_t_squared *
        ball.GetComponent<Rigidbody2D>().velocity.normalized;

    return (ball.transform.position + ut + scalarToVector);

}

//function to know if the ball is between the player and the
bool isPlayerInFront () {
    //si el jugador tiene una x más pequeña que la pelota,
    // y la pelota una x más pequeña que la IA la pelota una
    // x más pequeña que la IA, significa que la pelota
    // está entre el jugador y la IA, golpea y la IA, golpea
    // si el jugador tiene una x más grande que la pelota, y la
    // pelota una x más grande que la IA x más grande que la IA,
    // significa que la pelota está entre el jugador y la IA, golpea


    if ((player.transform.position.x < ball.transform.position.x &&
        ball.transform.position.x < transform.position.x) ||
        (player.transform.position.x > ball.transform.position.x &&
            ball.transform.position.x > transform.position.x)) {

        return true;
    }
```

Finalmente tenemos las dos funciones auxiliares. FutureBallPosition utiliza una fórmula para calcular la posición futura de la pelota en base a su velocidad y dirección. Devuelve un vector de dirección.

Por otra parte, isPlayerInFront nos permite saber si la pelota está entre los jugadores o no, calculando para ello la diferencia en el eje x entre la pelota y el jugador, y la pelota y la IA.

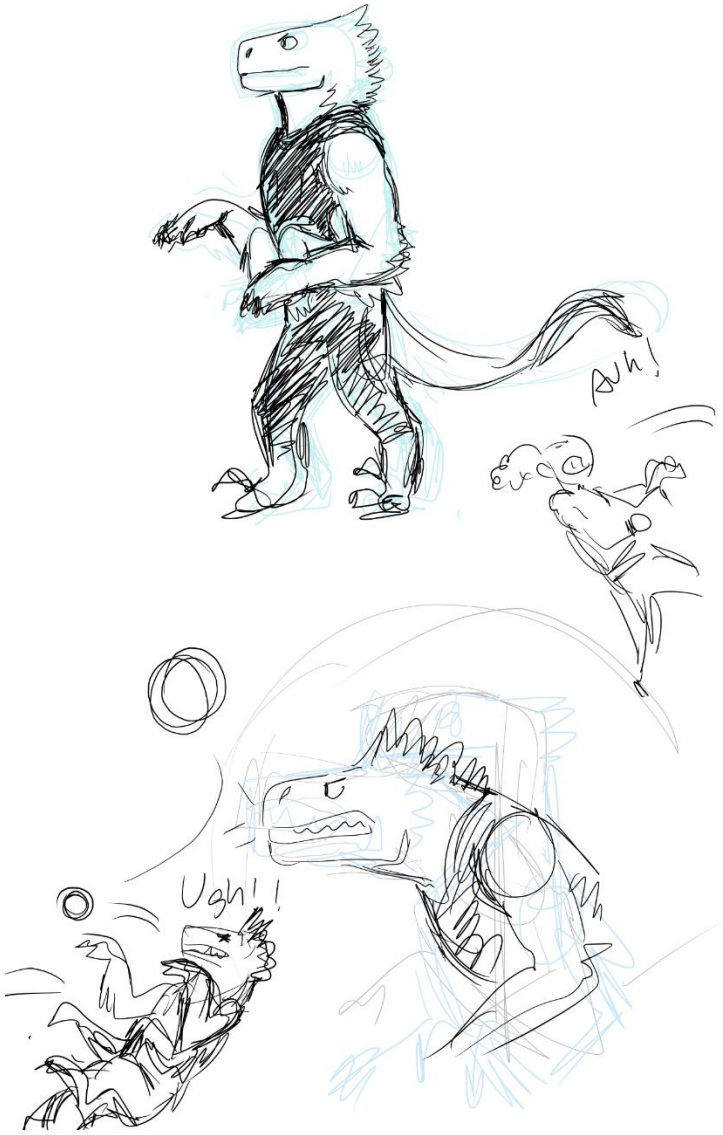
La primera versión de la IA evidenciaba demasiado su condición de IA. En ocasiones el enemigo se atascaba en las esquinas persiguiendo la pelota y golpeándola a la vez, y no podía salir de ese estado, por poner un ejemplo. Esto se solucionó mejorando el comportamiento de la IA y añadiendo nuevos estados.

9.5. Diseño, arte y animación

9.5.1. Concepto e ilustraciones

Para que el juego resultara más atractivo se le añadieron gráficos en su mayoría creados por la alumna. A excepción de botones y sliders, los primeros sacados de la assets Store de Unity (gratuitos) y los segundos utilizando herramientas propias de Unity.

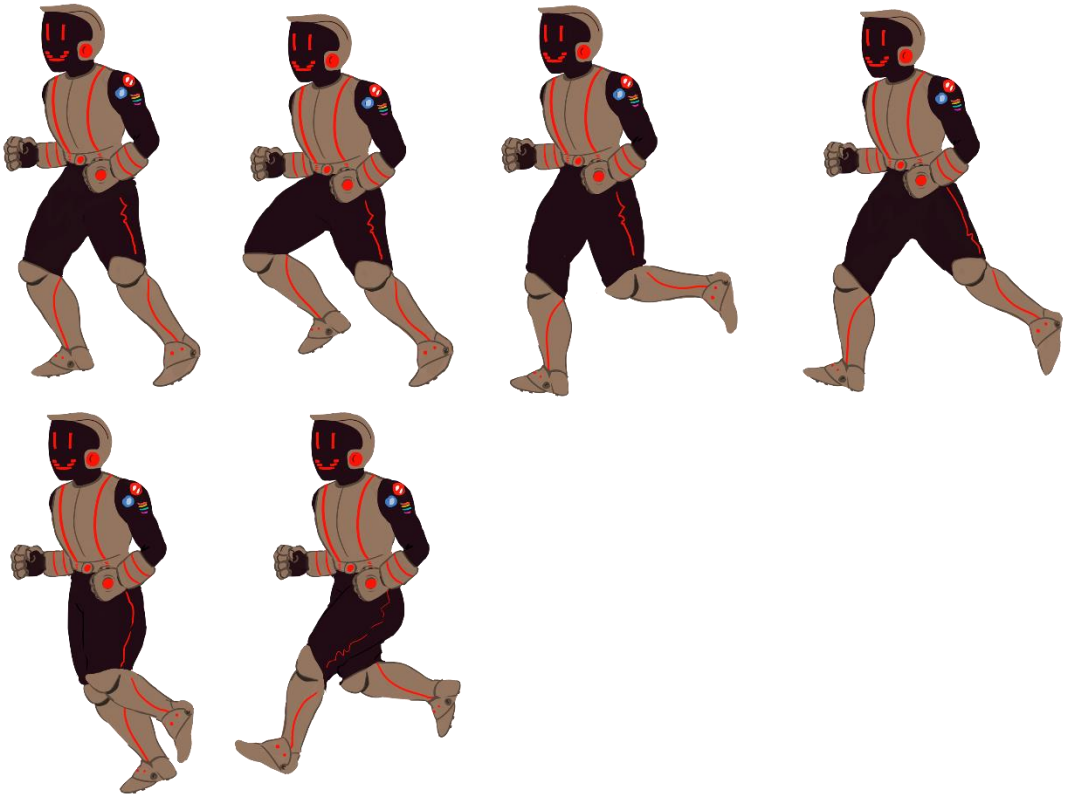
A continuación, mostramos algunos diseños sobre el concepto de juego:



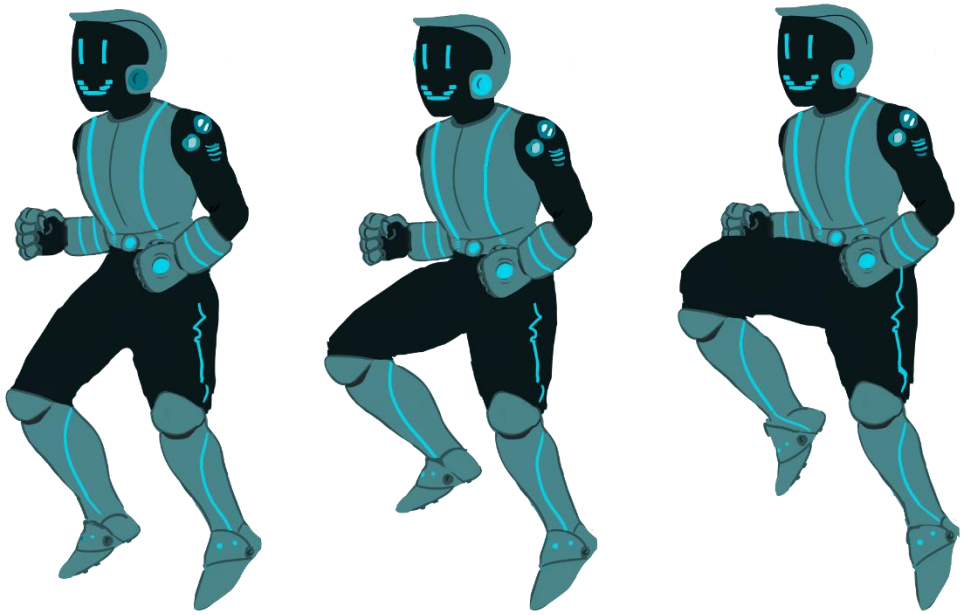
Inicialmente se pensó diseñar dos personajes, un alien y un humano, para ambientar el juego en el mundo de ciencia ficción que se había concebido. No obstante, los factores de tiempo obligaron a que se llevara a cabo el diseño de un único personaje, que fue evolucionando hasta el resultado actual.

Se realizaron distintos bocetos para estudiar los movimientos de los personajes, así como los golpes y saltos. El personaje principal pasó de ser claramente un ser humano a convertirse en una especie de androide como vemos en los bocetos a continuación. También se desarrolló la idea de los “guantes” empleados en este juego de pelota. Al golpear la pelota los guantes cambian el color de la energía de la misma al color del jugador, de esta forma la pelota no hiere al jugador que la golpeó por última vez. Si la pelota pierde energía vuelve al estado neutro, codificada con un color verde. Los diseños se pueden ver a continuación.

A continuación, veremos algunos de los diseños definitivos:

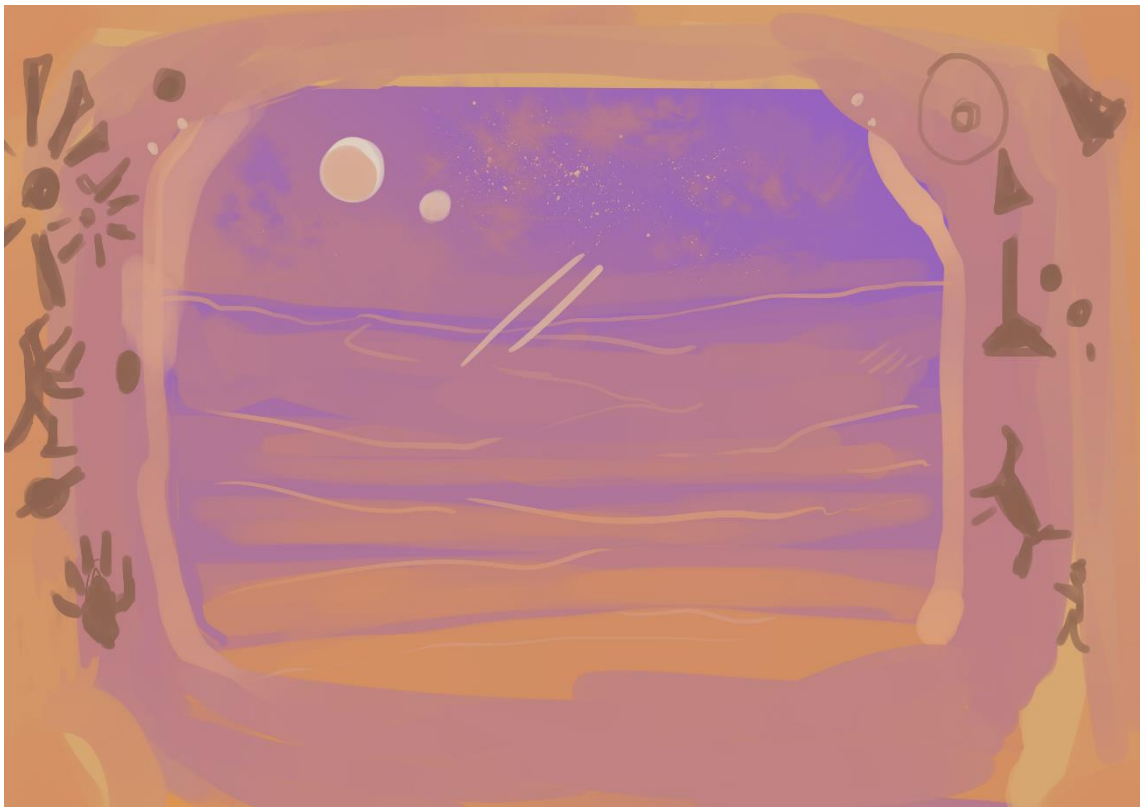


Sprites del jugador rojo corriendo.



Sprites del jugador azul saltando.

Diseño del escenario del juego.



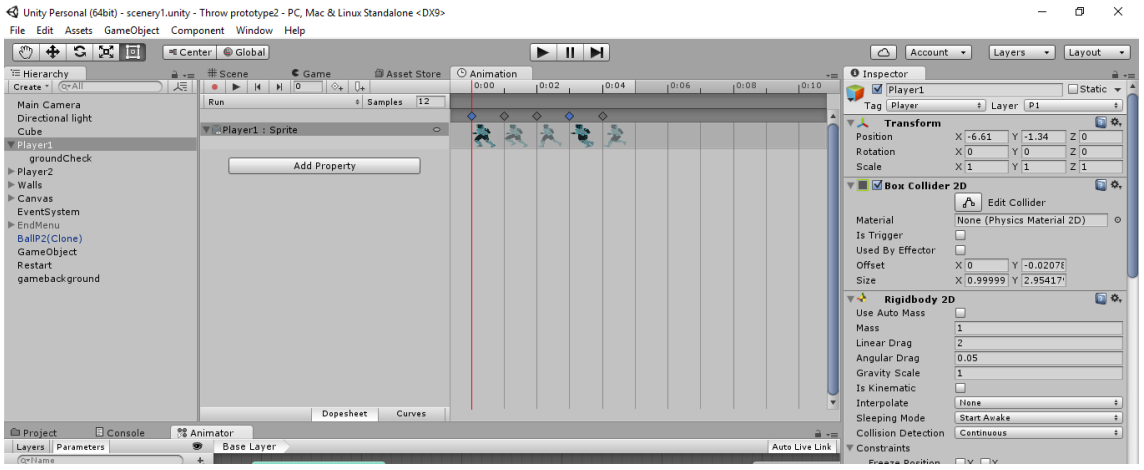
Otro escenario del juego.

Ilustración del menú principal del juego.

9.5.2. Animación

Unity cuenta con varias herramientas para facilitar la realización de animaciones de juego. Principalmente, la ventana de animación y el animator.

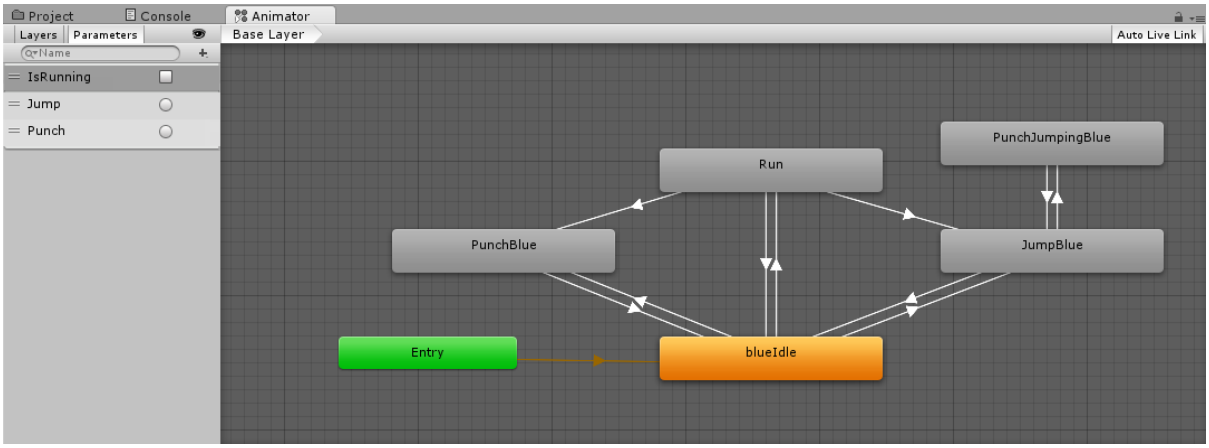
Al importar nuestros assets, en este caso cada uno de los sprites de una animación, podemos hacerlo en un mismo archivo. Puesto que Unity detecta los distintos sprites y los separa con una simple operación llamada “slice”. Una vez separados cada uno de los sprites, se arrastran a la ventana de animación para crear un nuevo clip. Aquí se ajustará entre otras cosas la duración del clip y los frames por segundo. A continuación, vemos una captura de la ventana de animación.



En cuanto al animator, representa una máquina de estados en la que cada uno de los estados se corresponde con una de las animaciones. En este caso en el estado inicial estamos en modo ocioso, el personaje se mueve ligeramente arriba y abajo, en un movimiento que pretende imitar a los videojuegos clásicos. De ahí, según el detonante, pasamos a saltar, correr o a golpear. Una vez concluida la animación volvemos al estado inicial en el caso del golpe, pero no así en el caso del salto, que vuelve al estado ocioso si el personaje toca el suelo.

En el caso de estar en el estado de correr volvemos al estado ocioso si la velocidad del personaje es 0, esto se notifica mediante un booleano. De este estado podemos pasar también a golpear o saltar.

Finalmente, mientras estamos saltando podemos golpear, este es un nuevo estado pues tiene una animación distinta a la de golpear estando en el suelo. De nuevo, se regresa al estado anterior al finalizar el golpe.



9.5. Depuración de errores

Una vez desarrollada la IA enemiga, existían ciertos problemas. Por ejemplo, la velocidad del enemigo era muy superior a la del jugador.

Otro bug se daba cuando el enemigo estaba constantemente "flotando" debido a que no se había tenido en cuenta en la IA si ya se había producido un salto. Esto se solucionó accediendo a la variable "grounded" del GameController y comprobando su estado, si tenía un valor de cierto, entonces se permitía el salto. Al saltar el controlador pone la variable a falso, por lo tanto, la próxima vez que se intente saltar no se permitirá.

Otro error se producía debido a que la pelota colisionaba varias veces por frame con un jugador en determinadas ocasiones, causando daño múltiple. Para solucionarlo se implementó una corrutina a la que se llama tras cada golpe recibido. Esta corrutina hace que se tenga que esperar 3 segundos antes de poder recibir daño de nuevo.

Cuando se implementó la capacidad de anticiparse de la IA inicialmente el jugador enemigo prácticamente se "teletransportaba" a la posición futura de la pelota, en lugar de desplazarse hacia ella. En varias ocasiones atravesaba incluso las paredes o el suelo, de forma que el jugador enemigo se perdía. La solución se encontró usando la variable del objeto velocidad, en lugar del

transform.traslade. No obstante, esto conllevó una pega, y es que el enemigo se desplazaba con inusitada lentitud. Para corregirlo se tuvieron que aumentar los valores asignados para desplazamiento hasta alcanzar el equivalente a los obtenidos con el transform.traslade. Hablamos de aplicar fuerzas de valor 500 en lugar de 5, por ejemplo.

También se encontraban bugs en el menú de opciones. La música no se regulaba correctamente y producía un error. Para solucionarlo se hizo uso del DontDestroyOnLoad que permite que un objeto se mantenga incluso durante cambios de escena. Por lo tanto, al asignarle un valor al AudioListener (el cual controla el volumen de todos los sonidos del juego globalmente), el valor se mantenía en cualquiera de las escenas.

Una de las dificultades que se encontró a lo largo de todo el proyecto fue determinar la velocidad de la pelota. Se tuvo que ajustar la misma en numerosas ocasiones hasta obtener el resultado actual.

También se encontraron dificultades respecto a la animación con Mechanim en Unity. Se considera el sistema poco intuitivo. En el animator, que contiene la máquina de estado de las animaciones, no se encontraban los parámetros en las condiciones o funcionaban de manera no deseada.

Otro gran problema durante este proceso fue el importar sprites a Unity. La calidad de los sprites se reducía muchísimo respecto al dibujo original. Para solucionar esto se tuvo que reducir la imagen original, así como evitar que la imagen fuera comprimida al importar. No obstante, incluso así consideramos que no se alcanzó la calidad deseada.

El juego se sometió a más pruebas y feedback de los usuarios una vez considerado terminado. Se encontraron varios errores, entre ellos, al jugar al modo un jugador, se podía tomar el control del jugador 2 con sus teclas correspondientes. Otro fallo que se detectó es que el controlador del modo de juego se duplicaba. Para solucionarlo hubo que comprobar que el objeto no estuviera instanciado previamente.

10. Normativa y Legislación

El rápido crecimiento en el campo de las tecnologías de la información ha propiciado también la aparición de nuevos métodos a la hora de delinquir. En el presente trabajo, al desarrollarse dentro de este campo, abarcaremos distintos aspectos de la ley que conciernen a este proyecto.

Para empezar, entendemos como delito informático aquel que alude a actividades ilícitas realizadas por medio de dispositivos tecnológicos, ya sea ordenadores, smartphones, tabletas, internet, etc.

Entre los riesgos existentes encontramos aquellos que vulneran la privacidad, el fraude, robo y estafa, falsificación y malversación de canales públicos. Los daños ocasionados por este tipo de delito son superiores a los de la delincuencia tradicional a la vez que resulta mucho más complicado descubrir a los culpables.

La ONU (Organización de Naciones Unidas) reconoce los delitos informáticos mencionados a continuación:

- Fraudes cometidos mediante manipulación de ordenadores
- Manipulación de datos de entrada
- Daños o modificaciones de programas o datos computarizados.

Legislación Española

Aunque los delitos informáticos no están contemplados como un tipo especial de delito en la legislación española, existen varias normas relacionadas con este tipo de conductas:

- Ley Orgánica de Protección de Datos de Carácter Personal.
- Ley de Servicios de la Sociedad de la Información y Comercio Electrónico.
- Reglamento de medidas de seguridad de los ficheros automatizados que contengan datos de carácter personal.
- Ley General de Telecomunicaciones.
- Ley de Propiedad Intelectual.
- Ley de Firma Electrónica.

Además de estas normas, en el Código Penal español, se incluyen multitud de conductas ilícitas relacionadas con los delitos informáticos. Las que más se aproximan a la clasificación propuesta por el "Convenio sobre la Ciberdelincuencia" se reflejan en los siguientes artículos:

- **Delitos contra la confidencialidad, la integridad y la disponibilidad de los datos y sistemas informáticos:**
    - El **Artículo 197** contempla las penas con las que se castigará:
        - A quien, con el fin de descubrir los secretos o vulnerar la intimidad de otro, se apodere de cualquier documentación o efecto personal, intercepte sus telecomunicaciones o utilice artificios de escucha, transmisión, grabación o reproducción de cualquier señal de comunicación.

        - A quien acceda por cualquier medio, utilice o modifique, en perjuicio de terceros, a datos reservados de carácter personal o familiar, registrados o almacenados en cualquier tipo de soporte.
        - Si se difunden, revelan o ceden a terceros los datos o hechos descubiertos.

        En el **artículo 278.1** se exponen las penas con las que se castigará a quien lleve a cabo las mismas acciones expuestas anteriormente, pero con el fin de descubrir secretos de empresa.

    - El **Artículo 264.2** trata de las penas que se impondrán al que por cualquier medio destruya, altere, inutilice o de cualquier otro modo dañe los datos, programas o documentos electrónicos ajenos contenidos en redes, soportes o sistemas informáticos.
- **Delitos informáticos:**
    - Los artículos 248 y 249 tratan las estafas. En concreto el **artículo 248.2**considera las estafas llevadas a cabo mediante manipulación informática o artificios semejantes.
    - Los **artículos 255 y 256** mencionan las penas que se impondrán a quienes cometan defraudaciones utilizando, entre otros medios, las telecomunicaciones.
- **Delitos relacionados con el contenido:**
    - El **artículo 186** cita las penas que se impondrán a aquellos, que por cualquier medio directo, vendan, difundan o exhiban material pornográfico entre menores de edad o incapaces.
    - El **artículo 189** trata las medidas que se impondrán quien utilice a menores de edad o a incapaces con fines exhibicionistas o pornográficos, y quien produzca, venda, distribuya, exhiba o facilite la producción, venta, distribución o exhibición de material pornográfico, en cuya elaboración se hayan utilizado menores de edad o incapaces.
- **Delitos relacionados con infracciones de la propiedad intelectual y derechos afines:**
    - El **Artículo 270** enuncia las penas con las que se castigará a quienes reproduzcan, distribuyan o comuniquen públicamente, una parte o la totalidad, de una obra literaria, artística o científica, con ánimo de lucro y en perjuicio de terceros.
    - El **artículo 273** trata las penas que se impondrán a quienes sin consentimiento del titular de una patente, fabrique, importe, posea, utilice, ofrezca o introduzca en el comercio, objetos amparados por tales derechos, con fines comerciales o industriales.

En el ámbito internacional solo algunos países cuentan con una legislación apropiada. Entre ellos encontramos a Francia, Inglaterra, Alemania, España, Holanda, Estados Unidos y Chile.

- **Francia**. Su Ley 88/19 del 5 de enero de 1988 contempla:
    - Acceso fraudulento a un sistema de elaboración de datos.

    - Sabotaje Informático.
    - Destrucción de datos
    - Falsificación de documentos informatizados.
- **Alemania**. En Alemania cuentan con la Segunda Ley contra la Criminalidad Económica del 15 de mayo de 1986. Esta ley reforma el Código Penal para contemplar delitos como espionaje de datos, estafa informática, falsificación de datos probatorios, alteraciones de datos y el sabotaje informático entre otros.
- **Inglaterra**. Tras varios ataques de hacking, en agosto de 1990 comenzó a regir la Computer Misuse Act (Ley de abusos Informáticos) por la cual cualquier intento de alterar datos informáticos con intención criminal se castiga con hasta 5 años de cárcel y multas de cuantías sin límite en función del delito cometido.
- **Holanda**. Hasta el 1 de marzo de 1993, Holanda era un paraíso para los hackers. A partir de ese día entró en vigencia la Ley de Delitos Informáticos. Esta ley contempla artículos específicos sobre técnicas de Hacking y los virus se consideran de forma especial.
- **Estados Unidos**. Este país adoptó en 1994 el Acta Federal de Abuso Computacional que modificó el Acta de Fraude y Abuso Computacional de 1986. En julio de 200 también se establece el Acta de Firmas Electrónicas en el Comercio Global y Nacional.
- **Chile**. Chile fue el primer país latinoamericano en establecer una Ley contra Delitos Informáticos. La ley 19.223 publicada el 7 de junio de 1993 señala que la destrucción o inutilización de un sistema de tratamiento de información puede ser castigado con prisión desde un año y medio a cinco años. También trata temas como el hacking o la divulgación de información privada.

10.1. Leyes que afectan a este proyecto

El proyecto no maneja datos de carácter personal tampoco manipula software o dispositivos de terceros. Por lo tanto, los aspectos legales se refieren al uso de licencias de los programas empleados, así como los recursos utilizados.

En cuanto a Unity, la licencia que se ha utilizado este proyecto es una licencia gratuita llamada Unity Personal. Bajo esta licencia el autor es dueño del contenido que crea, además se puede usar para proyectos de desarrollo personal de hasta 100.000$ al año. Esta versión no puede ser utilizada por:

- Entidad comercial que haya o bien alcanzado ingresos brugos que superen los 100.000 $ al año o que hayan recaudado fondos que superen los 100.000$ durante el año fiscal más reciente.
- Entidad no comercial con un presupuesto total anual que supere los 100.000.
- Una persona (que no actúe en nombre de una Persona Jurídica) o un propietario único que haya alcanzado ingresos brutos anuales que superen los 100.000 $ por el uso de Unity durante el año fiscal más reciente, que no incluya ningún ingreso devengado que no esté relacionado con el uso de Unity por parte de dicha persona.

La música empleada tanto para el menú como para el juego son de libre uso.

Los efectos de sonido correspondientes al golpe en la pared y los quejidos de los personajes son de libre uso, en la bibliografía se adjuntan las páginas que se usaron para obtener estos recursos.

El resto de recursos utilizados son los propios de Unity o han sido creados por la alumna para este proyecto.

Clasificación de videojuegos

España es uno de los 29 países europeos en los que la venta de videojuegos se regula mediante un sistema de catalogación por edades que impulsó la propia industria (PEGI, Pan European Game Information). Se trata de un sistema de orientación, que no obliga a los vendedores a requerir la documentación del comprador.

La clasificación de un juego confirma que es adecuado para jugadores que han cumplido una determinada edad. Así pues, un juego PEGI 7 solo será adecuado para quienes tengan 7 o más años de edad y un juego PEGI 18 solo será apto para adultos mayores de 18 años. La clasificación PEGI tiene en cuenta la idoneidad de la edad de un juego, no su nivel de dificultad.

El sistema PEGI se utiliza y cuenta con el respaldo de la Comisión Europea.

11. Conclusión y mejoras futuras

La realización de este proyecto ha permitido adquirir una gran cantidad de competencias y conocimientos relacionados con el diseño y creación de videojuegos y sus distintas fases, así como ámbitos. Comenzando por la idea pasando por el análisis, el desarrollo, la parte técnica, el apartado visual, etc. Se ha aprendido a manejar nuevas herramientas de software, a conseguir los assets necesarios para llevar a cabo un proyecto en Unity con pleno conocimiento del tipo de licencias existentes. Además, consideramos que se han afianzado competencias y conocimientos adquiridos previamente.

En cuanto a futuras mejoras del juego se plantean las siguientes. Algunas de las cuales no se llegaron a implementar por falta de tiempo:

- Añadir más personajes y skins, de forma que el jugador pueda elegir el personaje con el que quiere jugar. Esto conllevaría la creación de un menú de elección de personaje. Algunos personajes podrían ser desbloqueados tras alcanzar ciertos logros.
- Golpe especial, sería interesante que cada jugador tuviera un golpe especial que tras cargar una barra de energía se pudiera usar para dar mayor potencia a la pelota.
- Añadir modos de juego distintos, como un torneo con dificultad incremental en el que el jugador tiene que superar las distintas fases para ganar el mismo.
- Mejoras de la Inteligencia Artificial, por ejemplo, que pudiera anticiparse más allá de la primera colisión de la pelota.
- Otra posibilidad interesante sería adaptar el juego a otras plataformas. Esto conllevaría realizar ciertos cambios, sobre todo si se quisiera adaptar a móvil.
- Pulir el sistema de dificultad, añadiendo más variables.
- Modo de juego con múltiples pelotas simultaneas.

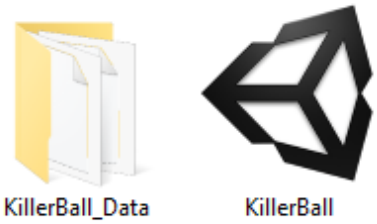
12. Manual de usuario

12.1. Contexto y objetivo

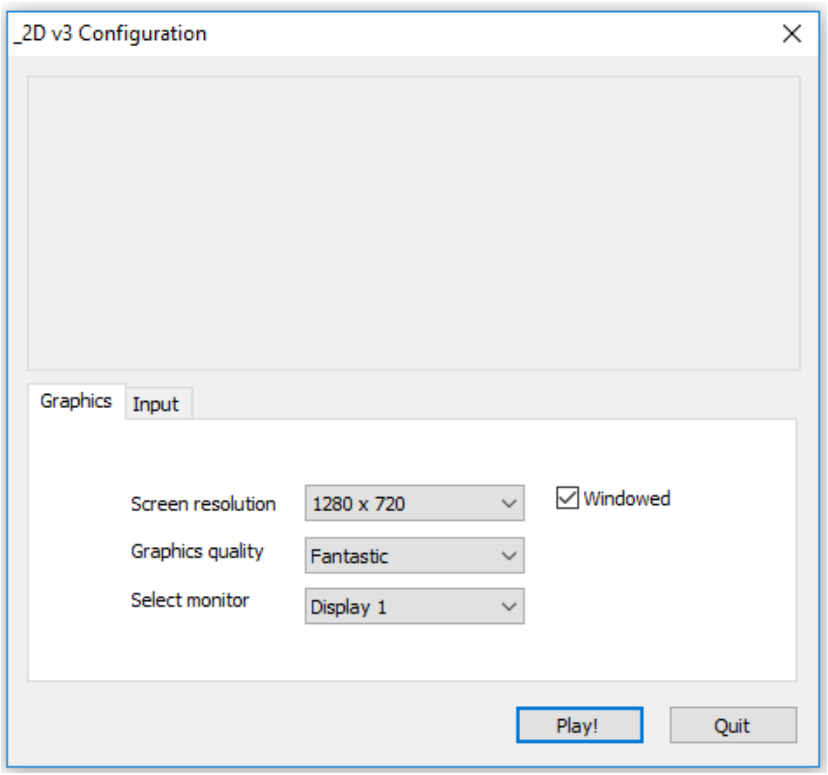
El juego está ambientado en un mundo de ciencia ficción habitado por humanos, alienígenas, androides y robots. Se desarrolla durante un torneo de un deporte algo peligroso en el que los participantes llevan trajes especiales para manipular la energía de la pelota. La pelota cambia de color según quién la ha golpeado, volviendo a estado neutro si pierde potencia. Tiene tres estados, azul (jugador 1), rojo (jugador 2) y verde (estado neutro). Cuando la pelota tiene el color verde (estado neutro) o el mismo color que el jugador, éste jugador no recibe daño en caso de que la pelota impacte contra él. Sólo se le resta vida si el color de la pelota es el mismo que el del rival. El objetivo del juego es derrotar al rival, esto sucede cuando la barra de vida del mismo llega a cero.

12.2. Ejecutando el juego

Disponemos de dos archivos, el KillerBall_Data que contiene los ficheros del juego y el KillerBall que es el fichero ejecutable. Para empezar el juego necesitamos hacer doble click sobre este segundo archivo.



A continuación, nos encontramos con la siguiente ventana emergente:



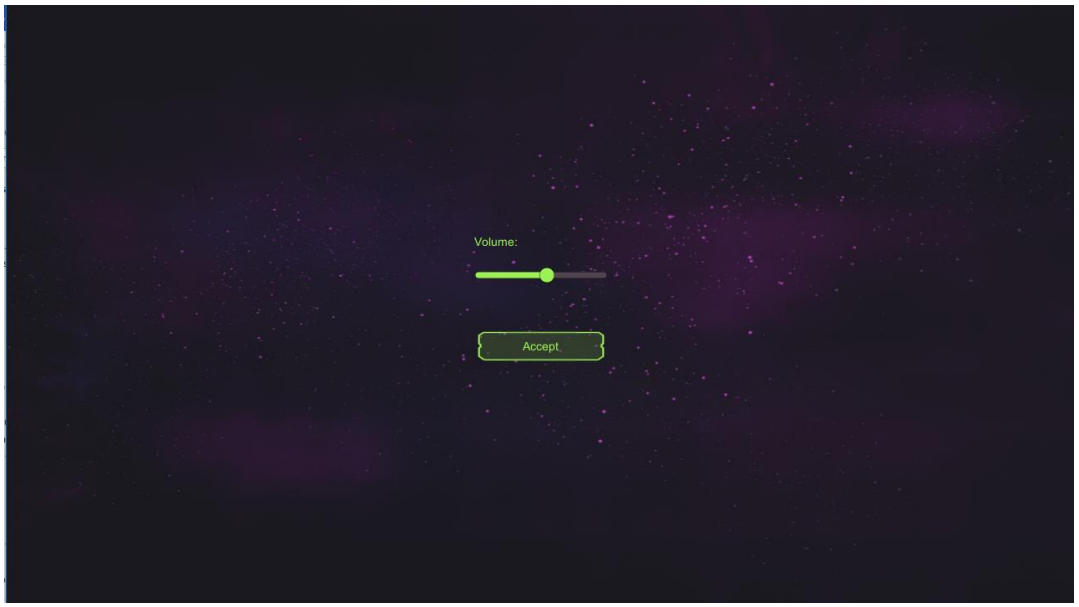
Aquí podremos modificar la resolución de pantalla, la calidad de gráficos, el monitor en el que se visualizará o si ejecutar el juego en pantalla completa entre otras opciones. Pulsaremos entonces el botón *Play!* para comenzar el juego. A continuación, veremos el logo de Unity y pasaremos al menú inicial del juego.

12.3. Menú de inicio

Una vez arrancado el juego veremos la pantalla del menú principal. En este menú podemos acceder a las opciones de juego en el botón de *Options*. Para comenzar a jugar tendremos que pulsar el botón de *New Game*.

12.3.1. Opciones

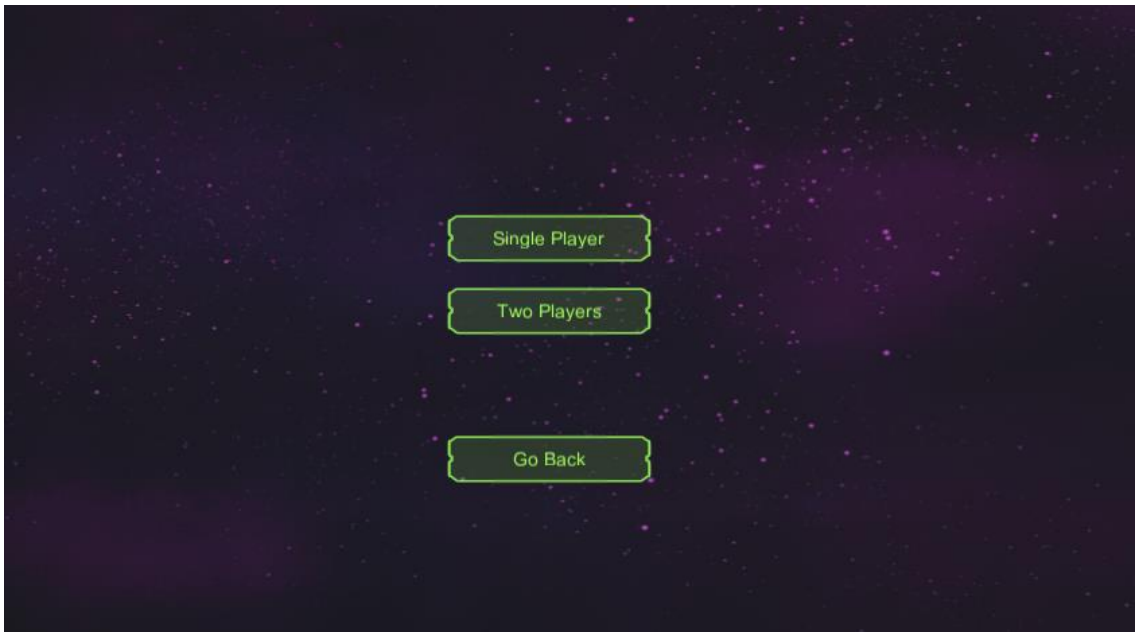
Si pulsamos el botón de opciones nos aparecerá la pantalla a continuación:



En este submenú de momento solo está implementado el volumen del juego, que podremos aumentar o reducir, e incluso silenciar. Hacemos click en aceptar para volver al menú principal.

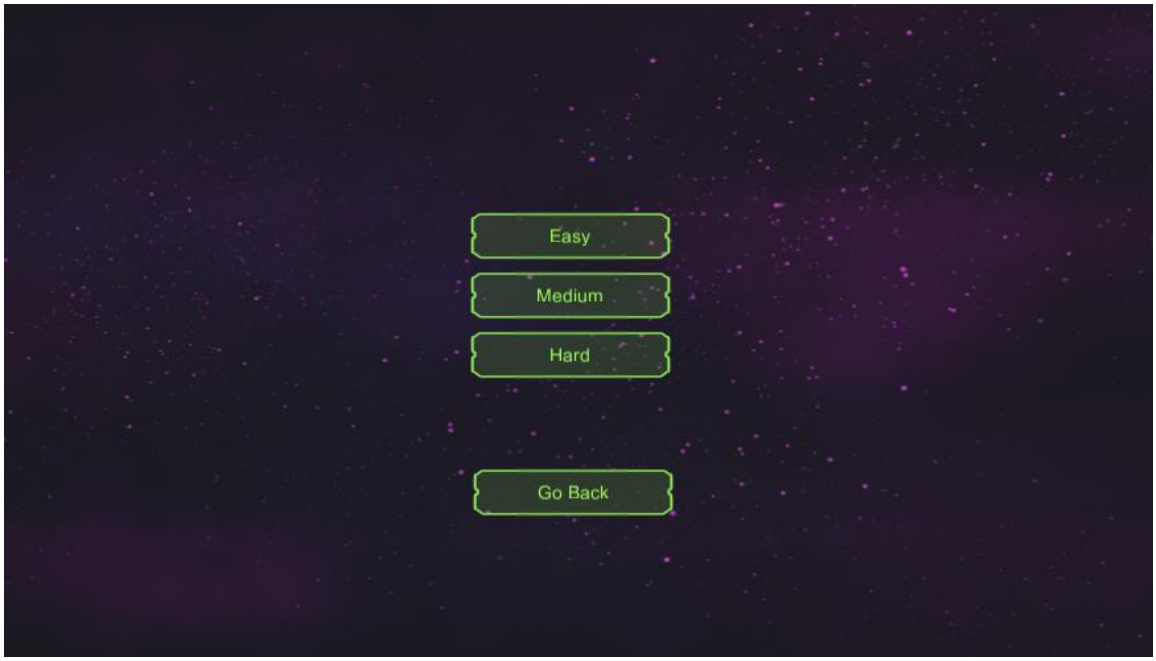
12.3.2. Modo de juego

Si hacemos click en *New Game* en el menú inicial nos aparecen los dos modos de juego, a saber, de un solo jugador (*Single Player*) o de dos jugadores (*Two Players*). En el primer modo el jugador se enfrentará a una Inteligencia Artificial. En el segundo modo se enfrentan dos jugadores humanos entre sí.



12.4. Dificultad

A continuación, accedemos al menú de dificultad y podemos seleccionar el nivel de dificultad del juego. Fácil, medio o difícil.

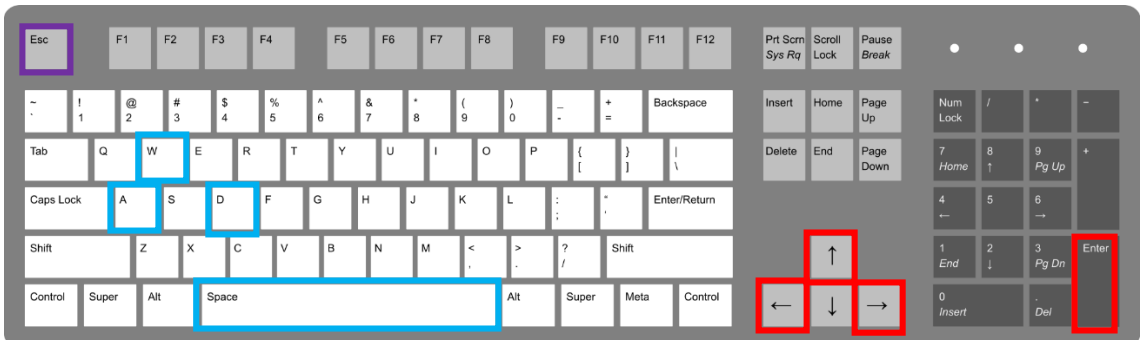


12.5. Controles

En el modo *Single Player, e*l jugador se mueve con las teclas awd de la siguiente forma. A -> izquierda, D -> derecha, W-> salto. Para golpear se usa la tecla E.

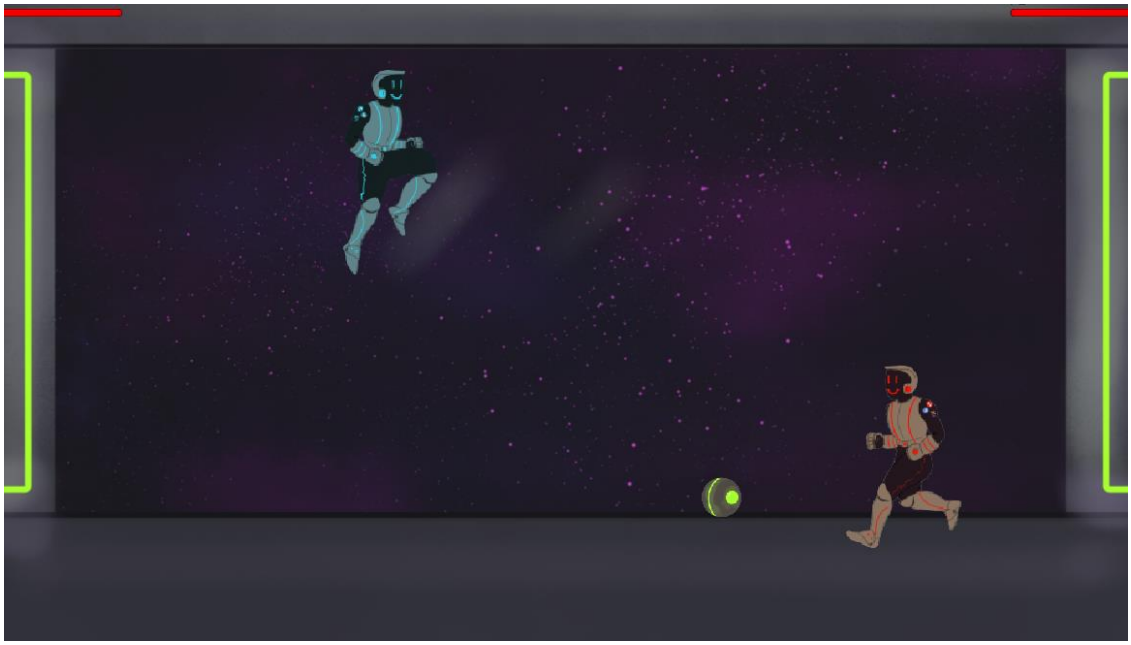
En el caso de dos jugadores el jugador azul tiene los controles AWD además del E para golpear. Mientras que el jugador rojo tiene los botones de dirección, flecha izquierda para ir a la izquierda, flecha derecha para ir la derecha, flecha arriba para saltar y el enter para golpear la pelota.

Durante el juego en marcha se puede pulsar la tecla escape para salir del juego.



En azul las teclas de movimiento y de golpeo del jugador 1. En rojo las teclas de movimiento y golpeo del jugador 2. En violeta la tecla de escape para pausar el juego.

Captura del juego con los personajes en estado ocioso



Captura del juego con los personajes saltando y corriendo

13. Glosario

GameObject: En Unity, es la clase base de todos los elementos de una escena. No confundir con gameObject (empezando por minúscula) que es un tipo de variable.

Corrutina: Subrutinas que se invocan una a otra de manera que, en la segunda llamada o en llamadas sucesivas, la ejecución se reanuda en el punto en el que se había interrumpido por última vez.

Animator: En Unity, es un componente que hace de interfaz de control respecto a la animación en Mecanim.

Trigger: En Unity, en el ámbito de animación y en el contexto de esta práctica, son unos disparadores que permiten activar un evento, en este caso una animación. Cuando termina la misma el trigger vuelve al estado original.

Sprite: Gráficos de juegos en 2D. Representan desde personajes a objetos y efectos en un juego.

Mesh renderer: Malla poligonal, consta de una serie de vértices relacionados entre sí formando triángulos.

Assets: Recursos de Unity. Unity cuenta además con la Assets Store en la que se pueden encontrar recursos gratuitos o de pago.

Frames: Fotogramas.

Audio Listener: El Audio Listener en Unity funciona como un micrófono. Recibe la entrada de todos los sonidos de la escena y los reproduce a través de los altavoces del ordenador.

Skins: En videojuegos suele denominarse así a las variaciones de sprite de un mismo personaje.

Prefab: En Unity, Permite almacenar un GameObject con sus componentes y propiedades. De esta forma se facilita su reutilización.

Shader: Es cualquier programa de ordenador que permite sombrear, es decir, proporcionar los niveles adecuados de color a una imagen, además hoy en día también permite producir efectos especiales o hacer post procesamiento de vídeo.

Renderizado: Es un término usado para referirse al proceso de generar una imagen o vídeo mediante el cálculo de iluminación GI partiendo de un modelo en 3D. Se desarrolla con el fin de generar en un espacio 3D formado por estructuras poligonales. Una simulación realista del comportamiento tanto de luces, texturas y materiales

Rigidbody: En Unity, un componente que se puede asociar a un GameObject y que permite aplicarle físicas.

Canvas: En Unity, un elemento de la Interfaz de juego que permite que se le agreguen imágenes, botones, texto, etc.

Panel: En Unity, es un elemento de la Interfaz de juego asociado al Canvas. Puede haber varios por canvas. En lugar de usar directamente el Canvas podemos usar los panels para agregar imágenes, botones, etc.

Collider: Definen la superficie de un objeto para poder detectar colisiones físicas, aunque en ocasiones también pueden definir un área.

BoxCollider: También existe el CircleCollider entre otros, estos collider se pueden combinar. Hay también distinciones entre 2D y 3D.

Raycast: Genera un rayo desde el punto de origen en dirección determinada durante una distancia máxima hasta que encuentra algún collider en la escena.

Quad: Similar a un plano, pero tiene una unidad de profundidad. Su ventaja respecto al plano es que sólo se compone de dos triángulos mientras que el plano contiene 200.

14. Bibliografía

Historia

https://es.wikipedia.org/wiki/Historia_de_los_videojuegos#Nuevas_consolas.2C_Handhelds_y_otros_formatos

https://en.wikipedia.org/wiki/Early_history_of_video_games

https://en.wikipedia.org/wiki/History_of_video_games

https://es.wikipedia.org/wiki/Industria_de_los_videojuegos

https://psicogamer.com/articulos/los-videojuegos-como-herramientas-para-el-desarrollo-de-la-sociedad/

https://en.wikipedia.org/wiki/Unity_(game_engine)

http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/

http://blogs.unity3d.com/es/2016/06/16/evolution-of-our-products-and-pricing/

https://www.cryengine.com/get-cryengine

https://www.unrealengine.com/what-is-unreal-engine-4

http://docs.unity3d.com/Manual/Navigation.html?_ga=1.116319232.780478527.1455900866

http://docs.unity3d.com/Manual/LearningtheInterface.html

Cursos de Unity

Básicas del Editor

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/editor-basics

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/introduction-to-unity-via-2d

Game Objects

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/gameobjects

Modelos y Materiales

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/models-and-materials

Usando Cámaras

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/cameras

Usando Luces

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/lights

Partículas

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/particle-systems

Scripting con Unity

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/scripting-primer

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/scripting-primer-continued

Fun with Lasers

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/fun-with-lasers

Mallas de Navegación

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/navmeshes

Ejemplo de Juego Runner:

http://uººnity3d.com/learn/tutorials/modules/beginner/live-training-archive/infinite-runner

Ejemplo Juego 2D:

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/2d-catch-game-pt1

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/2d-catch-game-pt2

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/2d-catch-game-pt3

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/2d-catch-game-qa

Ejemplo Angry Birds:

http://unity3d.com/es/learn/tutorials/modules/beginner/live-training-archive/making-angry-birds-style-game?playlist=17093

http://unity3d.com/es/learn/tutorials/modules/beginner/live-training-archive/making-angry-birds-style-game-pt2?playlist=17093

Juegos 2d de Vista Superior

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/top-down-2d

Animaciones con Mecanim

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/animate-anything

Persistencia

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/persistence-data-saving-loading

Audio

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/audio

Granadas Teleportadores

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/teleportation-grenades

Diversión con Explosiones

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/fun-with-explosions

Explosiones Cinemáticas

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/cinematic-explosions

Basic Platformer

https://unity3d.com/es/learn/tutorials/projects/mini-projects/creating-basic-platformer-game

Legislación y normativa

http://www.gitsinformatica.com/legislacion.html#lgs

http://www.delitosinformaticos.info/delitos_informaticos/legislacion.html

http://www.ocendi.com/in-motion/limitar-los-videojuegos-violentos-algunos-datos/

Sonidos

https://www.freesound.org/browse/tags/sound-effects/

http://soundbible.com/free-sound-effects-1.html

IA

Programming Game AI By Example por *Matt Buckland*

http://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

http://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-squad-pattern-using-steering-behaviors--gamedev-13638

http://gamedevelopment.tutsplus.com/tutorials/the-action-list-data-structure-good-for-ui-ai-animations-and-more--gamedev-9264

http://gameprogrammingpatterns.com

http://playmedusa.com/blog/a-finite-state-machine-in-c-for-unity3d/

http://playmedusa.com/blog/automata-finito-en-c-para-unity3d/

http://playmedusa.com/blog/simple-fsm-like-structure-with-coroutines-in-c-unity3d/