Diseño y desarrollo de un prototipo básico de un videojuego plataformas en 2D

Universidad de las Palmas de Gran Canaria

Escuela de Ingeniería Informática

Trabajo de Fin de Grado

Autora: Carlota Esteban Cazalla

Tutor: Agustín Trujillo Pino

Tutor: Adrián Rivero Pérez

Las Palmas de Gran Canaria, Diciembre de 2014

Contenido

1. Objetivos del proyecto 5
2. Justificación de las competencias 7
3. Aportaciones al entorno social 9
4. Situación actual 13
   4.1. Historia de los videojuegos 13
   4.2. Motores de videojuego 19
   4.3. Diseño de videojuegos 22
5. Análisis y prototipado 25
   5.1. Elección del motor 25
   5.2. Preproducción vs producción 26
   5.3. Concepción de la idea 28
   5.4. Nociones de Unity 30
   5.5. Fase de prototipado 31
      5.5.1. Prototipo movimiento de personaje 31
      5.5.2. Prototipo cambio de escenario 32
      5.5.3. Prototipo runner 36
      5.5.4. Prototipo enemigo pequeño 38
      5.5.5. Prototipo cajas 38
      5.5.6. Prototipo ataque 39
      5.5.7. Prototipo enemigo grande 42
      5.5.8. Prototipo artístico 43
      5.5.9. Prototipo artístico + cambio de escenario 44
      5.5.10. Prototipo de sonidos 44
      5.5.11. Nuevos obstáculos 44
      5.5.12. Versión alfa 46
   5.6. Casos de uso 46
6. Diseño y arquitectura 49
   6.1. Documento de diseño 49
   6.2. Arquitectura del software 49
   6.3. Diseño de niveles 53
7. Implementación 57
   7.1 Implementación del personaje 57
   7.2. Implementación de la cámara 61
   7.3. Implementación de enemigos 64

7.4. Implementación de cambio de mundo, entorno y obstáculos 66
7.5 Interfaz 71
7.5.1. Diseño de interfaz 71
7.5.2. Implementación de interfaz 73
7.6. Implementación de niveles 78
7.7. Implementación del sonido 78
8. Pruebas y mantenimiento 81
9. Trabajo futuro 83
10. Conclusiones 85
11. Normativa y legislación 87
12. Manual de usuario 91
Bibliografía 95
Anexo 1: Documento de diseño 97
Anexo 2: Arquitectura del software 101

1. Objetivos del proyecto

El objetivo principal de este Trabajo de Fin de Grado (referido también como TFG en futuras menciones) es realizar un prototipo básico de un videojuego. Con la elaboración de este proyecto se pretende cubrir todas las etapas que conllevan la realización de un videojuego comercializable. Se partirá de una idea original que irá evolucionando a medida que se vaya avanzando en el desarrollo del videojuego.

Para alcanzar el objetivo principal es necesario cumplir con una serie de objetivos más concretos:

- Estudio y aprendizaje de las herramientas de desarrollo para videojuegos actuales, para evaluar su idoneidad para la realización del proyecto.
- Estudio y aprendizaje de técnicas de diseño de videojuegos, debido a que se parte de una idea de juego original.
- Análisis y diseño de mecánicas del juego, prototipando cada mecánica planteada.
- Análisis y prueba de diferentes plataformas de control (mando, teclado y ratón), adaptando las mecánicas a cada una de ellas.
- Desarrollo de la arquitectura del programa, siguiendo los principios de la arquitectura de software.
- Implementación del videojuego siguiendo la arquitectura previamente establecida.

2. Justificación de las competencias

La realización de este Trabajo de Fin de Grado conlleva cubrir una serie de competencias profesionales. A continuación exponemos las competencias cubiertas en este TFG.

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.

Cubrimos esta competencia llevando a cabo el diseño y desarrollo de este TFG, cubierto en los capítulos de Análisis y prototipado, Desarrollo del primer jugable y Pruebas y mantenimiento. Todos los prototipos, versión alfa y versión jugable siguen los principios éticos y la legislación y normativa vigente.

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.

La planificación ha sido fundamental para la elaboración de este TFG. Se ha llevado a cabo una planificación temporal. A medida que se avanzaba en el proyecto, se iban evaluando todas las posibles mejoras y se añadían a la temporización.

CII18 – Conocimiento de la normativa y la regulación de la informática en los ámbitos nacional, europeo e internacional.

Esta competencia queda justificada en el capítulo de Normativa y legislación.

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.

Esta competencia se cubrirá una vez se realice la presentación de este Trabajo de Fin de Grado ante los miembros del tribunal establecidos.

3. Aportaciones al entorno social

El desarrollo de este Trabajo de Fin de Grado ha contribuido a lo siguiente:

- Mejorar los conocimientos adquiridos durante los años cursados en el Grado en Ingeniería Informática.
- Aprender a llevar a cabo la planificación, análisis, diseño, documentación y desarrollo de proyectos.
- Adquirir nuevos conocimientos relacionados con el diseño y desarrollo de videojuegos.
- Comprender las últimas tecnologías de la industria del videojuego y aprender a manejar nuevas herramientas de desarrollo.

Con respecto a las aportaciones sociales, la industria del videojuego en España se está consolidando como una de las industrias más dinámicas en el campo del contenido digital, posicionándose como una de las fuerzas principales para la economía de este ámbito. A nivel global, el sector del videojuego es la industria tecnológica con mayor crecimiento. En 2012 el mercado mundial del videojuego alcanzó 66,300 millones de dólares. Se prevé que la industria crezca a un ritmo anual del 6.7%, llegando a 86,100 millones de dólares en 2016. La irrupción de Internet como herramienta de distribución de videojuegos ha facilitado enormemente el crecimiento de este mercado.

Figura 1. Evolución de la facturación global del mercado del videojuego (millones de dólares). Fuente: Newzoo

Según más datos de Newzoo (ver Figura 1), en España existen 17 millones de jugadores de videojuegos en 2012. Si realizamos un estudio sobre las plataformas más usadas, las más usadas son consolas y dispositivos móviles, usadas por el 76% de los jugadores. Le siguen las páginas web de juegos casuales y las redes sociales, con un 68% y un 64% respectivamente. Las descargas, los juegos para PC y Mac y los MMO (videojuego multijugador masivo en línea) son utilizados por menos de la mitad de usuarios.

En la figura 2 podemos ver que el número de jugadores casuales es el más elevado de todas las plataformas presentadas: 16,6 millones de personas invierten su tiempo en páginas sociales o páginas web de juegos casuales. En la figura 3 apreciamos que el tiempo invertido en estas páginas es de casi el 30%.

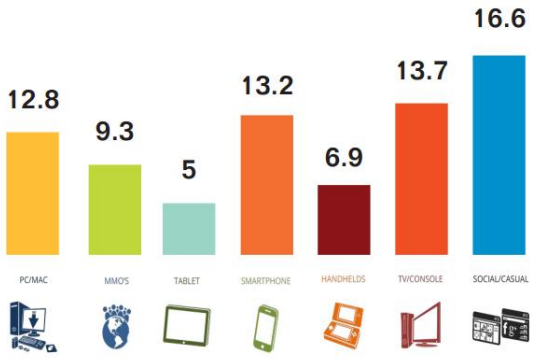


Figura 2. Número de jugadores por plataforma en España (unidades: millones de personas). Fuente: Newzoo

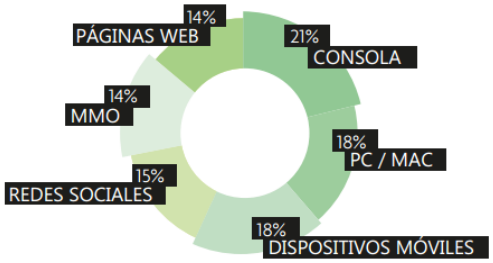


Figura 3. Tiempo invertido en videojuegos por plataforma. Fuente: Newzoo

Respecto a la inversión económica, los españoles gastaron 1.900 millones de euros en videojuegos en 2012, un 17% más que en 2011. Los videojuegos accesibles a través de Internet representan el 38% del mercado.

En la figura 4 vemos cómo ha crecido el mercado entre los años 2011 y 2012. El tiempo empleado en jugar a videojuegos se incrementó en un 30%. El porcentaje de jugadores sobre la población total aumentó considerablemente en España, un 13%. El aumento más significativo se ha producido en los jugadores que gastan dinero para jugar, que ha crecido un 44% entre 2011 y 2012.

Figura 4. Crecimiento del mercado del videojuego en el periodo 2011-2012. Fuente: Newzoo / Imagen: Univack Creative

Hablamos de una industria que crece de manera dinámica en España. En la figura 5 comprobamos que el sector está compuesto por 330 empresas, siendo Madrid, Cataluña y la Comunidad Valenciana las que aglutinan el 67,4% del total de empresas. Vemos que en Canarias se encuentran el 2,4% de las empresas de videojuegos españolas, un porcentaje bastante elevado con respecto a otras comunidades autónomas.

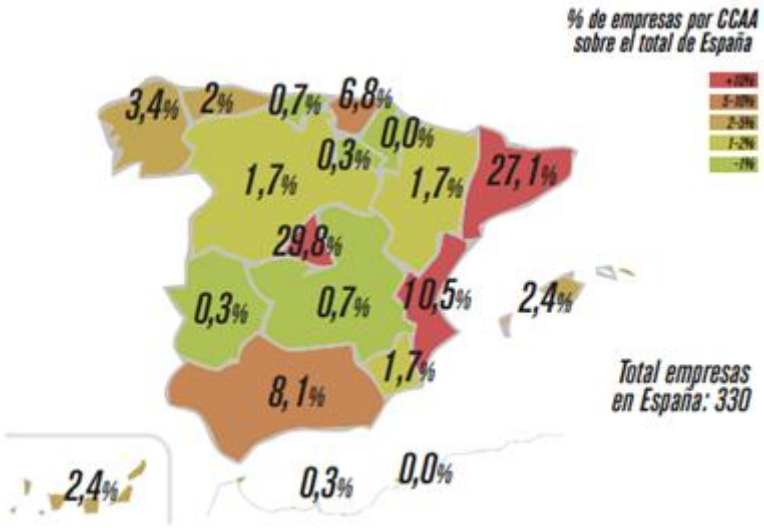


Figura 5. Distribución territorial de las empresas de videojuegos. Fuente: DEV (Asociación Española de Empresas Desarrolladoras de Videojuegos y Software de Entretenimiento)

Teniendo en cuenta todos estos datos, nuestro prototipo puede servir para aportar un nuevo producto a una industria que crece cada año. Además, el aprendizaje pretende servir como impulso a la formación en este sector, que constituye una salida profesional prometedora.

Al margen de las aportaciones sociales, una de las aportaciones fundamentales ha sido elegir para el desarrollo uno de los motores gráficos más utilizados actualmente, y llevar a cabo este TFG utilizando la última versión, siendo necesario el aprendizaje de dicha herramienta. En capítulos posteriores haremos un análisis de todos estos motores y seleccionaremos el más adecuado para nuestro proyecto.

4. Situación actual

4.1. Historia de los videojuegos

PRIMEROS AÑOS

De izquierda a derecha: el juego Tennis for Two, Ralph Baer y el juego Spacewar!

La historia de los videojuegos se remonta a 1947, cuando Thomas T. Goldsmith Jr. y Estle Ray Man construyeron un juego electrónico interactivo que permitía al jugador controlar un punto que representaba una mira, y disparar a unos aviones. Esta idea nunca se comercializó, pero se patentó en 1948.

Años más tarde, en 1950, Charly Adam creó el videojuego "Bouncing Ball" ("Pelota Saltarina"), un programa creado para el ordenador Whirlind del MIT. Fue el primer videojuego en mostrar señales de vídeo en tiempo real.

A pesar de empezar a tomar rumbo hacia lo que hoy se conoce como videojuegos, estos dos inventos no era completamente interactivos. Fue en 1951 cuando una de las figuras más reconocidas de la industria de los videojuegos, Ralph Baer[1], propuso utilizar instrumentos de calibración de equipos para ofrecer una experiencia interactiva a través de un televisor, como elemento innovador para distinguirse de la competencia. Esta idea fue rechazada de inmediato, y Baer no la recuperó hasta 15 años más tarde.

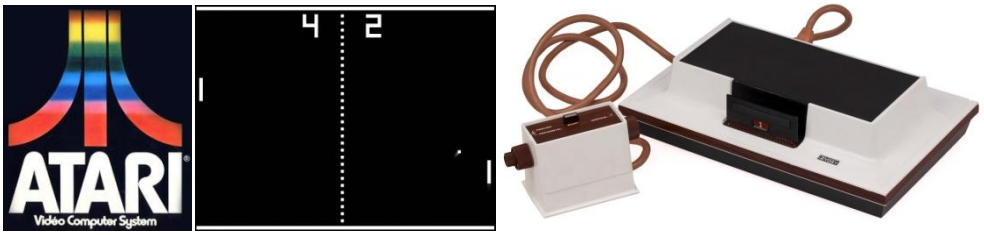
Pero antes de que Baer recuperara su idea, existieron dos proyectos de videojuegos que se disputarían el reconocimiento de "primer videojuego de la historia".

Por un lado, William Higinbotham desarrolló en 1958 el que se considera el primer videojuego de la historia: "Tennis for Two" ("Tenis para dos"). Mientras trabajaba en el Laboratorio de Brookhaven, y con la premisa de que las exhibiciones de ciencia no eran lo suficientemente interactivas, creó un sencillo simulador de tenis con un osciloscopio y dos diales para las visitas del Laboratorio. Cientos de visitantes esperaban su oportunidad para jugar a este juego electrónico, que se convirtió en el precursor de la industria que nacería a partir de entonces.

Por otra parte, un joven programador del MIT, Steve Russell, lideró al primer equipo de desarrollo de un videojuego para crear Spacewar, que consistía en una pantalla de tubo de rayos catódicos y un teclado como periférico de entrada.

[1] Ralph Baer falleció recientemente, durante la elaboración de este documento, el día 6 de diciembre de 2014.

PRIMERA GENERACIÓN



De izquierda a derecha: el logo de Atari, el juego Pong, y la consola Magnavox Odyssey

Sin embargo, ninguno de estos juegos aportó beneficios económicos a ninguno de sus creadores. El nacimiento de los videojuegos como industria comenzó en 1972, teniendo como protagonista a Nolan Bushnell. Bushnell decidió fundar su propia compañía de entretenimiento electrónico: Atari. Su cofundador fue Ted Basney. Bushnell y Ted contrataron a Alcorn para diseñar y crear en conjunto el hardware y software que daría soporte a uno de los videojuegos más conocidos de la historia: el Pong. Fue una inversión arriesgada, ya que nadie conocía entonces cuál sería la aceptación que tendría este juego en el público. Para descubrir su éxito, la máquina recreativa fue instalada en un bar local. Los clientes de este bar quedaron fascinados y al día siguiente, la máquina dejó de funcionar debido a que el depósito de monedas se había llenado por completo.

Una vez demostrada la aceptación del público, comenzó la fabricación en serie de las máquinas recreativas. Una vez vendidas unas 1000 máquinas por todo Estados Unidos, se comenzó a plantear la versión doméstica de Pong, que vio la luz en 1973. Esta versión planteó otro problema, ya que las jugueterías del país no querían vender la nueva versión. Fue una cadena de tienda de deportes y un solo anuncio los únicos elementos que bastaron para que todas las unidades de Pong se vendieran. Todo este éxito hizo que Atari creciera como empresa dedica al entretenimiento electrónico, llamando la atención de los mejores profesionales.

No obstante, en 1966, Ralph Baer puso una denuncia contra Atari por plagio. Hablábamos antes de la idea que tuvo Ralph Baer de generar una experiencia interactiva, y en 1966 consiguió recuperar esta idea y crear una máquina que permitía jugar a varios juegos, uno de ellos llamado Tennis. Este Tennis esta idea era muy parecida a Pong. Magnavox comercializó esta primera consola, llamada Odyssey, y fue presentada en una feria estatal, a la que Bushnell había asistido. Ambos llegaron a un acuerdo de licencia antes de pasar por los tribunales.

A pesar del éxito que tuvo Atari, y del crecimiento de Magnavox con la consola Odyssey, hubo una crisis económica de la industria de los videojuegos en 1977. Fabricantes de consolas obsoletas y clonadores del Pong vendieron sus sistemas con pérdidas solamente para vaciar sus stocks, saturando el mercado del entretenimiento electrónico. Muchas empresas tuvieron que cerrar, mientras las dos únicas grandes empresas, Atari y Magnavox, permanecieron en el mercado, pero con importantes pérdidas.

SEGUNDA GENERACIÓN

De izquierda a derecha: el juego Pong, la consola Atari VCS y el sistema Commodore 64

Esta situación duró hasta 1978, cuando Atari realizó una conversión de un juego arcade, que se convertiría en uno de los videojuegos más populares: Space Invaders. Con este lanzamiento la industria de los videojuegos revivió.

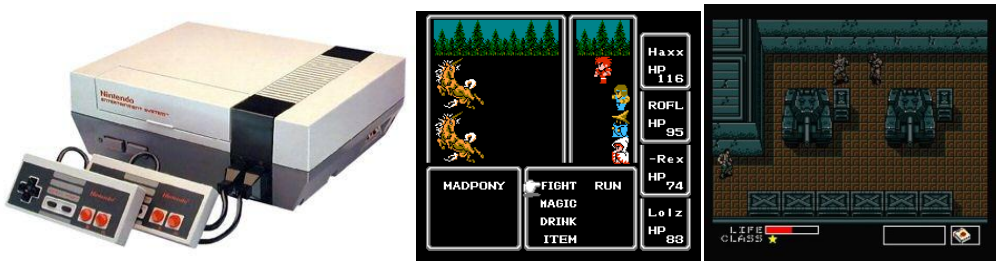
Atari lanzó otros juegos de gran impacto, como el Battle Zone, o el Asteroids, y se centró en un nuevo modelo de consola: la consola doméstica con cartuchos de ROM intercambiables. En 1979, Atari lanzó la Atari VCS. No solo era nuevo el concepto de cartucho intercambiable, sino que además incluía un periférico de control que marcaría un estándar, el joystick. Adaptó el Space Invaders para esta nueva consola y muchas personas compraron una consola Atari solamente para tener este juego en sus casas. Con esto nació una nueva tendencia para los fabricantes de consolas, que se disputarían los derechos de juegos arcade y dirigirían sus campañas de marketing a "llevar la experiencia de las recreativas a casa".

En 1980 Warner compró Atari y debido a las constantes disputas de los ejecutivos de Warner y Bushnell, este último abandonó la empresa, junto con otros diseñadores importantes de Atari. Uno de los trabajadores que abandonaron, David Crane, fundó Activision junto con otros antiguos empleados de Atari. Consiguieron los derechos de desarrollo de juegos en cartucho para la VCS de Atari. Con esto, Activision publicaba juegos manteniendo derechos de autor en las consolas de Atari, mientras Atari aumentaba su librería de videojuegos. Con esto nació la primera empresa Third-Party dedicada al desarrollo de videojuegos en exclusiva, sin necesidad de desarrollar sus propias consolas para publicar estos juegos.

Una vez apareció la primera empresa Third-Party, muchas empresas de software intentaron subirse al carro y desarrollaron juegos de escasa o nula calidad que tuvieron un impacto negativo en la industria. De nuevo, se produjo una crisis económica, Warner tuvo grandes pérdidas por el repentino colapso del sector, y vendió Atari en 1984.

De nuevo, hizo falta un nuevo producto para volver a resucitar el mercado. En Europa nació la Commodore 64, un ordenador doméstico con 64kB de RAM. Fue un éxito por dos motivos: su precio, muy competitivo con las consolas del momento; y el lenguaje de programación que utilizaba, el Commodore BASIC. Este lenguaje de programación permitía al usuario programar sus propios videojuegos. Las revistas especializadas publicaban código enviado por los lectores, y cualquiera podía copiarlo y tener ese juego en su casa. La capacidad de creación y compartir las creaciones impulsó esta consola. Nacieron muchas máquinas similares a la Commodore 64: ZX Spectrum, Amstrad CPC, etc.

TERCERA GENERACIÓN



De izquierda a derecha: la consola NES, el juego Final Fantasy y el juego Metal Gear

Mientras en Europa se concentraban en ordenadores personales, en Japón una empresa de barajas Hanafuda revolucionaría la industria. Este empresa es Nintendo, y la consola estrella que marca esta generación fue la Super Famicon, también conocida como NES. Ahora sería en Japón donde se daría una de las eras icónicas del mundo de los videojuegos. Gran parte del éxito de esta consola se debió al lanzamiento de juegos basados en personajes creados por otra de las figuras importantes de la industria: Shigeru Miyamoto. Miyamoto creó personajes emblemáticos como Mario, Donkey Kong o Link.

Otro de los grandes nombres que nacieron en ese momento es Hironobu Sakaguchi. Con la empresa Square al borde de la quiebra, decidió hacer un último juego para tratar de salvarla: Final Fantasy. Hoy en día es una de las franquicias más grandes y exitosas de la industria de los videojuegos.

Para la MSX2 nació el primer juego de sigilo de la historia, Metal Gear, que catapultó a la fama el nombre de su creador, Hideo Kojima, y dio nacimiento a otra de las grandes franquicias que sigue cosechando éxito en nuestras consolas actuales, Metal Gear Solid.

Con el cese de la fabricación de NES, terminó esta tercera generación, pero ya empezaban a verse los cimientos de lo que es la industria hoy en día.

CUARTA GENERACIÓN

De izquierda a derecha: las consolas Turbografx16, SNES y Mega Drive

La cuarta generación nació en 1987, aún cuando a la tercera generación de consolas le quedaban años de vida. El detonador de esta nueva generación fue Hudson Soft, con su Turbografx16, lanzada por la empresa NEC. A pesar del gran éxito que tenía la NES en Japón en ese momento, consiguió un buen número de ventas, pero en Estados Unidos cayó por detrás de la competencia.

En 1988, la empresa Sega, decidió apostar por la adaptación de su consola a la nueva Mega Drive y la lanzó al mercado con una versión del juego arcade Altered Beast. En Japón no consiguió desbancar a la Turbografx16, pero en territorios PAL cosechó un gran éxito.

En 1989, Nintendo creó la primera consola portátil, la Gameboy. Gracias al elemento innovador de la portabilidad, y el gran reconocimiento del que gozaba Nintendo, la Gameboy consiguió ser la tercera consola más vendida de todos los tiempos. Sin embargo, Nintendo perdía el mercado internacional debido a la Mega Drive, por lo que decidió entrar en esta nueva generación con la Super Famicon, conocida también como Super Nintendo (SNES).

Esta época se caracteriza por la extensa proliferación de títulos exclusivos para una consola. A mitad de esta generación nacieron tecnologías como la de Compact Disc, de éxito escaso en esta generación ya que era tecnología muy cara en ese momento. Nintendo se asoció con Sony para añadir un complemento de CD para la SNES, pero el trato quedó cancelado cuando Sony demandaba más control. Sony utilizaría su trabajo con Nintendo para sentar las bases de una de las consolas estrella de la siguiente generación.

QUINTA GENERACIÓN

De izquierda a derecha: las consolas Sega Saturn, Playstation y PC-FX

En 1993 dos competidores decidieron entrar en la batalla SNES - Mega Drive - Turbografx 16. Nacieron las consolas 3DO y Atari Jaguar, que a pesar de ser de mayor potencia que las tres consolas competidoras, salieron a un precio muy elevado, por lo que no supusieron una amenaza para las ventas de Sega, Nintendo y NEC.

En 1994, tres nuevas consolas de quinta generación fueron lanzadas en Japón: Sega Saturn, Sony Playstation y PC-FX, esta última de NEC. Un año después saltarían al mercado internacional.

Playstation tuvo un enorme éxito. Nintendo consiguió mantenerse en el mercado gracias a su consolidación previa en el sector con su SNES, pero intentó reaccionar a la fuerte aceptación de Sony presentando en 1996 la Nintendo 64.

De esta manera Sony y Nintendo se mantuvieron como las dos grandes empresas de la industria, dejando fuera de la competencia a Sega y NEC. En esta época nacieron nuevos géneros, como el survival horror gracias a Resident Evil de Playstation o el shooter gracias a GoldenEye de Nintendo. Ambas empresas ofrecían grandes catálogos de videojuegos, pero poco a poco las empresas Third-party apostaron por Sony ya que el formato de cartuchos ROMs de Nintendo 64 era más caro de producir y tenía menos memoria que los CDs de Playstation.

SEXTA GENERACIÓN

De izquierda a derecha, las consolas Playstation 2, Xbox y Gamecube

Con la llegada de esta generación Sega lanzó al mercado la Dreamcast en 1998. En 2001 dejó de fabricarlas, abandonó la competencia de consolas y se dedicó al desarrollo de videojuegos en exclusiva para Nintendo. La empresa americana Microsoft aprovechó el abandono de Sega para hacerse un hueco en el mercado de las consolas.

En 2000 Sony lanzó la Playstation 2, con un nuevo formato, el DVD, que mejoró de manera considerable las especificaciones técnicas de su predecesora. Nintendo, una generación más, tardó en incorporarse a la nueva generación y en 2001 presentó la Gamecube, con otro formato de soporte que estaría por detrás de los DVDs de Playstation 2, los mini DVDs. Las empresas Third-party apostaron de nuevo por el formato de Sony más que por el de Nintendo, y Sony conseguiría el mayor reconocimiento de esta época.

Microsoft entró en el mercado en 2001 con la Xbox. Consiguió atraer a muchos desarrolladores de PC con el NT Kernel y DirectX de su sistema operativo Windows. Con el tiempo ganó popularidad, sobre todo en Estados Unidos. Consiguió igualar a la Gamecube en ventas.

SEPTIMA Y OCTAVA GENERACIÓN

De izquierda a derecha: las consolas Playstation 4, Wii U y Xbox One

La séptima generación de consolas nace con el lanzamiento de versiones más potentes de las consolas de Microsoft y Sony. Microsoft lanza en 2005 la Xbox 360, y Sony en 2006 la PlayStation 3. Ambas se caracterizaban por una mejora sustancial de gráficos en alta definición, los discos duros para el almacenamiento de datos y la capacidad de jugar online.

Nintendo presentó más tarde su consola, pero a diferencia de sus competidoras, no tenía soporte para disco duro interno, disponía de una resolución máxima de 480p y creando un nuevo controlador, el Wii Remote. Este nuevo soporte de controlador fue la pieza clave para que Nintendo consiguiera mantenerse en el mercado con un gran éxito de ventas debido a la facilidad con la que veían el manejo de consolas las personas menos asiduas a los videojuegos.

Sony y Microsoft imitaron a Nintendo sacando al mercado dispositivos de movimiento, PlayStation Move y Kinect, pero Nintendo ya había establecido la fidelidad de los jugadores más casuales y no tuvieron gran éxito.

La generación actual es la octava generación. Sony con su Playstation 4, Microsoft con su Xbox One, y Nintendo con su Wii-U.

4.2. Motores de videojuego

3D Construction Kit

Los motores de videojuegos son sistemas software que permiten la creación y desarrollo de videojuegos, actuando como un software intermediario. Disponen de un motor de renderizado para gráficos 2D y 3D, un motor físico para la detección de colisiones, animación, sonidos, inteligencia artificial, scripting, etc. Todo esto son herramientas para que los desarrolladores de videojuegos simplemente adapten un mismo motor de videojuego para poder crear juegos diferentes, reduciendo los costes de desarrollo y mantenimiento.

Antes de que existieran los motores de videojuego, los juegos se programaban sin software intermediario. Los juegos tenían que diseñarse de manera ascendente, de manera que se aprovechara de manera óptima el hardware de visualización. Incluso cuando la visualización no era un problema, las restricciones de memoria impedían los intentos de crear los diseños de datos pesados que el motor necesitaba. Además la gran mayoría no se podía reutilizar. La rápida proliferación de hardware para arcade, que era la tecnología punta de la época, implicaba que la mayoría de código usado tenía que ser descartado más adelante, ya que las siguientes generaciones de juegos utilizarían nuevos diseños que tomarían ventajas de nuevos recursos. Por ello, la mayoría de los diseños de juegos de los 80 consistían en un conjunto de normas bien programadas con un pequeño número de datos de gráficos y pocos niveles. Desde la época dorada de los videojuegos arcade, fue más común desarrollar motores de videojuegos para la propia compañía y usarlos para software de la propia empresa.

Existieron algunos sistemas precursores antes de que el término "motor de videojuego" naciera. Todos ellos vienen de la década de los 80. Algunos ejemplos son:

- Adventure Game Interpreter de Sierra
- SCUMM de LucasArts
- Freescape Engine de Incentive Software

Motor SCUMM de Luca Arts

En la década de los 90, surge el término "motor de videojuego", sobre todo relacionado con los juegos de disparos en primera persona (First Person Shooters, FPS). Debido a la popularidad de juegos como Doom y Quake, en vez de trabajar desde cero, otros desarrolladores licenciaron las partes importantes del software y desarrollaron sus propios gráficos, personajes, armas y niveles. Todo esto se conoció

posteriormente como "contenido del juego" o "recursos del juego". La separación de datos y reglas especificas del videojuego y sus datos con respecto a conceptos básicos como la detección de colisiones y entidades del juego significaba que los equipos podían crecer y especializarse.

A mediados de los 90, el motor 3D de Quake fue un gran avance en la utilización de motores gráficos 3D, debido a que se aprovechaba las primeras tarjetas aceleradoras 2D.

A finales de los 90, surgió el Unreal Engine, que permitía realizar modificaciones. En 2004 nació el motor Source Engine, de Valve, y el motor CryEngine, que se utilizó para crear Far Cry.

Los motores de videojuegos actuales pueden llegar a ser de las aplicaciones más complejas jamás desarrolladas. La continua evolución de los motores ha creado una gran diferenciación entre renderizado, scripting, artwork y diseño de niveles.

Motores de acceso público

CRYENGINE

Es un motor de juego creado por la compañía desarrolladora de videojuegos Crytek. Fue introducido en el primer juego de FarCry. Está diseñado para ser usado en plataformas PC y consolas, incluyendo Playstation 4 y Xbox One. Tiene altas capacidades gráficas, a la par que Unreal Engine 4. También posee herramientas para el diseño de niveles. Es un motor de juego muy potente, pero requiere de mucho aprendizaje para comenzar a usarlo de manera productiva.

Su precio es de 8 euros al mes sin necesidad de aportar más dinero por las ganancias de los juegos creados.

UNREAL ENGINE 4

Es un motor gráfico lanzado por Epic Games, sucesor del UDK. Tiene grandes capacidades gráficas incluyendo herramientas como la creación de luces dinámicas y un nuevo sistema de partículas que puede manejar hasta un millón de partículas en una misma escena. Requiere de bastante aprendizaje para un buen uso. El lenguaje que utiliza Unreal Engine 4 es C++. Utiliza

un sistema de Blueprints. Diseñado para realizar juegos para PC, Mac, iOS, Android, Xbox One y Playstation 4.

Para utilizar este motor es necesaria una suscripción de 15 euros al mes y un 5% de los beneficios que generen tus títulos lanzados al público.

UNREAL DEVELOPMENT KIT

UDK es la versión gratuita de Unreal Engine 3, desarrollado por Epic Game. Tiene un gran nivel de capacidades gráficas y puede ser usado para juegos en dispositivos móviles. Tiene un conjunto de herramientas para el diseño de niveles directamente desde el motor. Fue desarrollado principalmente para juegos FPS, pero puede adaptarse a otros géneros. Utiliza un lenguaje de scripting llamado UnrealScript, orientado a objetos, similar a Java y C++. Puede crear juegos para iOS, Android, Windows Phone 8, Xbox 360, Playstation 3, Playstation Vita y Wii U, pero si queremos publicar en alguna plataforma que no sea PC o Mac, tenemos que adquirir la licencia completa.

Este motor es completamente gratuito hasta el momento en que quieras publicar tu juego. Es necesario pagar una licencia de 80 euros y el 25% de los beneficios hasta que hayas generado 50.000$ de tu juego.

GAMEMAKER

GameMaker es conocido por ser un motor de videojuegos para principiantes o para personas que hagan del desarrollo de videojuegos su hobby. A pesar de ello, existen muchos títulos lanzados al mercado desarrollados bajo este motor. Tiene su propio lenguaje llamado GML, que unido a su sistema de eventos predefinidos permite customizar los juegos.

Este motor es completamente gratuito.

UNITY 3D

Unity ofrece un gran conjunto de características y una interfaz amigable. Los juegos desarrollados pueden ser portados fácilmente a Android, iOS, Windows Phone 8 y Blackberry, haciéndolo un buen motor gráfico para el desarrollo de videojuegos en dispositivos móviles. También tiene la capacidad de desarrollar para Playstation 3, Xbox360, Wii U y navegadores web. No existe restricciones para los archivos de aplicaciones 3D ya que tiene soporte para la mayoría: Blender, Maya, 3ds Max, etc. También tiene capacidad de desarrollo de 2D nativa. Sin embargo no tiene gran capacidad de edición dentro del editor. Todos los modelos 3D deberían ser modelados en aplicaciones 3D de terceros. Sin embargo, tiene una gran librería de recursos que pueden ser descargados o comprados.

Existe una versión gratuita y una versión de pago, Unity Pro, que cuesta 1.200 euros o 60 euros al mes. Esta versión ofrece algunas características más que la versión gratuita, tales como iluminación global, render a textura y efectos de postprocesamiento.

4.3. Diseño de videojuegos

Todo juego comienza con un concepto. Se plantea una propuesta de juego en un equipo de desarrollo. Los diseñadores son los que hacen que este concepto tome forma.

El diseño de juegos consiste en tomar decisiones sobre cómo va a ser el juego que vamos a crear. No solamente se trata de una decisión, se trata de cientos de decisiones. Todas estas decisiones abarcan las siguientes áreas, entre otras:

- Reglas del juego
- Historia y tramas
- Estética
- Temporización
- Ritmo
- Toma de riesgos
- Recompensas
- Penalización

Hablamos del diseño de juegos de manera genérica, ya que se lleva a cabo tanto para juegos de cartas, como para juegos de tablero, de recreo, o videojuegos. El diseñador de videojuegos es un rol dentro del desarrollo de videojuegos. El diseñador de videojuegos debe estar implicado con el desarrollo del videojuego ya que las decisiones deben tomarse una vez se ven las ideas en acción. El trabajo del diseñador de videojuegos es comunicar a las diferentes áreas del desarrollo para que puedan llevar a cabo de la mejor manera posible su trabajo. Es un trabajo similar al de un analista de software.

El objetivo de un diseñador de videojuegos es crear una experiencia para el jugador. El juego no es la experiencia en sí, es solo un vehículo para hacer llegar la experiencia. La herramienta para llegar a transmitir esta experiencia es el juego.

5. Análisis y prototipado

5.1. Elección del motor

Antes de comenzar la fase de diseño del juego, es necesario hacer un estudio de los diferentes motores de desarrollo. En la sección previa hemos estudiado las diferencias de motores de videojuegos disponibles para el público general, y ahora es necesario hacer un análisis para elegir uno de ellos, el motor con el que se llevará a cabo este proyecto.

Para ello, seleccionamos tres factores fundamentales para el estudio:

- Usabilidad: Interfaz, sencillez de uso, documentación y comunidad.
- Funcionalidad: Características principales de cada motor.
- Costes: Cuánto nos cuesta desarrollar en dicho motor.

Motor gráfico	Usabilidad	Funcionalidad	Costes
CryEngine	Interfaz compleja. Curva de aprendizaje difícil. Escasa disponibilidad de documentación.	Gráficos muy potentes. Acabados de calidad. Soporte completo para 3D. Soporte incompleto para 2D. Programación en Lua/C++/Visual Scripting.	No disponemos de licencia. 8 euros/mes. Sin royalties.
Unreal Engine 4	Interfaz intuitiva. Curva de aprendizaje moderada. Gran disponibilidad de documentación Comunidad activa.	Gráficos muy potentes. Acabados de calidad. Soporte completo para 3D. Soporte incompleto para 2D. Programación en UnrealScript.	Disponemos de licencia. 15 euros al mes. 5% de royalties.
UDK	Interfaz intuitiva. Curva de aprendizaje moderada. Gran disponibilidad de documentación Comunidad activa.	Gráficos muy potentes. Acabados de calidad. Soporte incompleto para 2D. Programación en UnrealScript.	Gratuito si no se publica el trabajo realizado.
Unity	Interfaz intuitiva. Fácil aprendizaje. Gran disponibilidad de documentación Comunidad activa.	Apartado gráfico menos potente. Soporte completo para 3D. Soporte completo para 2D. Programación en JavaScript/C#/Boo.	Gratuito.

GameMaker	Interfaz muy intuitiva. Aprendizaje muy sencillo. Comunidad reducida.	Apartado gráfico menos potente. No tiene soporte para 3D. Soporte completo para 2D. Poco customizable programáticamente. Scripting en GML (Game Maker Language).	Gratuito.

Debido a que este TFG consiste en realizar un prototipo de videojuego no necesitamos gran potencia de gráficos. Además, el juego será diseñado para plataformas web por dos motivos:

- Contará con mecánicas complejas de implementar e incómodas de jugar en tablets y móviles
- Se pretende realizar un estudio para implementar las mecánicas con diferentes periféricos de control: mando, teclados y ratón.

También tenemos que tener en cuenta la curva de aprendizaje. Para poder terminar el prototipo en un número establecido de horas necesitamos realizar el proyecto en un motor que nos facilite el aprendizaje, ya que no se tiene mucha experiencia con este tipo de software.

Otro de los aspectos fundamentales es el soporte que ofrece cada motor para la realización de juegos 2D y 3D. Con motivo de aprovechar las ventajas de ambos, y aunque la idea original es realizar un videojuego en 2D, es recomendable que el motor también tenga soporte para videojuegos en 3D, para no limitar las posibilidades (por ejemplo, la utilización de gráficos 3D).

Teniendo todas estas restricciones en cuenta, decidimos que el mejor motor para el desarrollo del TFG es Unity 3D, por aportar las siguientes ventajas:

- Es gratuito.
- Es sencillo de utilizar.
- Está recomendado como uno de los mejores motores para realizar los primeros videojuegos.
- Aporta al usuario mucha documentación para su uso, y existe una comunidad muy activa para resolver problemas.
- Tiene soporte para juegos en 2D y 3D.
- Utiliza lenguajes de programación conocidos.

Escogemos C# como lenguaje de programación debido a que es muy similar a Java en la sintaxis básica. La preferencia por Java viene por ser uno de los lenguajes que más se han trabajado durante el Grado.

5.2. Preproducción vs producción

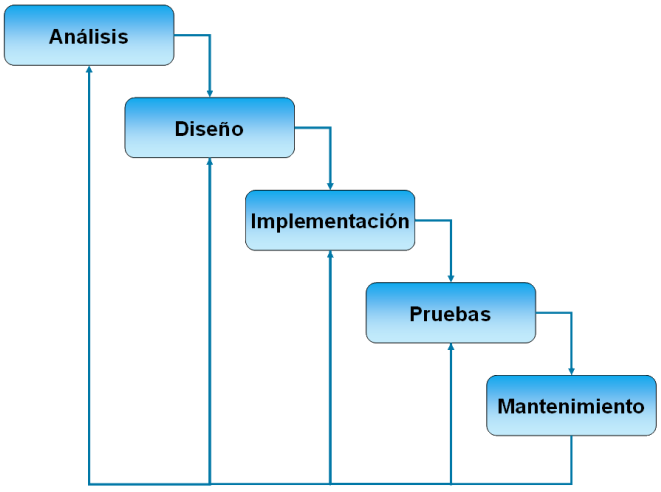
Mark Cerny es una figura de la industria de los videojuegos que ha trabajado como diseñador de videojuegos, programador, productor y ejecutivo de negocios. Actualmente es el arquitecto principal de Playstation 4 y Playstation Vita.

Cerny ha defendido abiertamente el método "METHOD" para el desarrollo de videojuegos. Este método divide el desarrollo en dos partes:

- Preproducción: Fase en la que se produce el diseño de juegos. Se caracteriza por tomar todas las decisiones relacionadas con cada aspecto del juego, llevar a cabo muchos prototipos para descubrir qué funciona y buscar aquello que diferencia al juego a desarrollar de la competencia.
- Producción: Fase en la que construimos el juego. Todas las decisiones se han tomado y se han demostrado que funcionan, siendo un producto viable para su comercialización.

Durante todo este TFG nos centraremos en la preproducción, ya que es necesario tomar todas las decisiones de desarrollo antes de comenzar a construir el juego. Nuestro producto final será el prototipo denominado First Playable (Primer Jugable). Este First Playable tiene que tener dos niveles aproximadamente, todas las características que hemos prototipado y hemos decidido incluir en nuestro juego, y debe tener una calidad suficiente para que parezca publicable.

Para la preproducción seguiremos un modelo de desarrollo en cascada. Este enfoque se puede adaptar fácilmente al desarrollo de videojuegos. La fase de análisis será la fase de prototipado de todas las mecánicas y elementos que queremos que aparezcan en nuestro juego. A continuación realizaremos el documento de diseño, en el que especificaremos las bases y la arquitectura de nuestro juego. Continuaremos con la implementación, las pruebas y el mantenimiento.



La temporización de nuestro proyecto se expone en la siguiente tabla:

Etapa del desarrollo	Duración
Análisis y prototipado	80 horas
Diseño y arquitectura	30 horas
Implementación del juego	120 horas
Pruebas y mantenimiento	20 horas

5.3. Concepción de la idea

La idea original de la que partimos para la realización del proyecto consiste en el desarrollo de un juego de plataformas en 2D. Realizamos un documento de análisis de esta idea con el fin de establecer los objetivos y mecánicas principales.

Temática

La temática escogida es el tema de los sueños. Existe un mundo real, que es el que vemos a diario, y existe un mundo que está en nuestras mentes, y que vemos al dormir. Estos dos mundos están relacionados, ya que las experiencias que nos ocurren a diario en nuestra realidad interfieren en los sueños, haciendo que imaginemos elementos o personas relacionados con los que hayamos presenciado o conocido ese día.

Definición aproximada del juego

Tecnología. La tecnología elegida es Unity 4.6, que se encuentra en este momento en fase beta. Unity presenta herramientas para el desarrollo de juegos en 2D y muchos assets (recursos) que pueden ser de utilidad para el prototipado y el desarrollo del videojuego. El juego se desarrollará de manera que sea aconsejable el uso de mando.

Género. Será un juego de plataformas 2D. Los videojuegos de plataformas son un género que se caracteriza por tener que caminar, correr, saltar o escalar sobre una serie de plataformas y acantilados. Existen enemigos, mientras se recogen objetos para poder completar el juego. Aprovecharemos los recursos que nos ofrece Unity para darle una estética 3D a algunos elementos del juego.

Mecánicas. La mecánica principal del juego se basa en la necesidad de cambiar el escenario en el que estamos, es decir, cambiamos de mundo en el que se encuentra el personaje principal.

Habrá un único personaje principal que podrá:

- caminar
- correr
- saltar
- cambiar de mundo

Habrá enemigos, pero no se les podrá atacar.

Estética

La estética debe diferenciar claramente cuándo se está en un mundo o en otro.

Música: En el mundo real la música será la misma que en el nivel de los sueños pero muy baja, apenas audible; en el mundo de los sueños habrá una música acorde a la estética

Visual: El mundo real es más monótono, se buscará una saturación negativa, vistas grisáceas aunque se aprecie color; el mundo de los sueños será de tonos muy saturados, y que no concuerden con la realidad (árboles y plantas de otros colores).

MUNDO REAL

MUNDO DE LOS SUEÑOS

Recursos aproximados

Personaje. Personaje SIN ARMAS 3D, ya que no va a combatir.

Este modelo tiene las animaciones necesarias para las mecánicas del personaje, así que puede ser un modelo a utilizar: https://www.assetstore.unity3d.com/en/#!/content/17222

Música. Chris Zabriskie "Cylinder One", de la página freemusicarchive.org.

Diferencias y similitudes entre los mundos

REALIDAD	SUEÑOS
Fotorrealista, los colores son apagados y que concuerdan con la realidad.	Colores que no concuerdan con la realidad.
Leyes físicas comunes	La gravedad puede ser ligeramente menor.
No hay enemigos.	Hay enemigos.
Hay objetos para presenciar y "llevar" a los sueños.	Los objetos presenciados en la realidad serán utilizados en este mundo.

El personaje se verá de la misma manera en ambos mundos, y las habilidades serán las mismas, pero existirán zonas de diferente gravedad.

Los enemigos solo se encontrarán en el mundo de los sueños, impidiendo el paso o dificultando el avance.

Referencias a juegos de dos mundos

Giana Sisters: Twisted Dreams, Of Light & Shadow y Schein.

Referencias a juegos de plataformas

Super Mario Bros 3., Yoshi's Island y Ducktales.

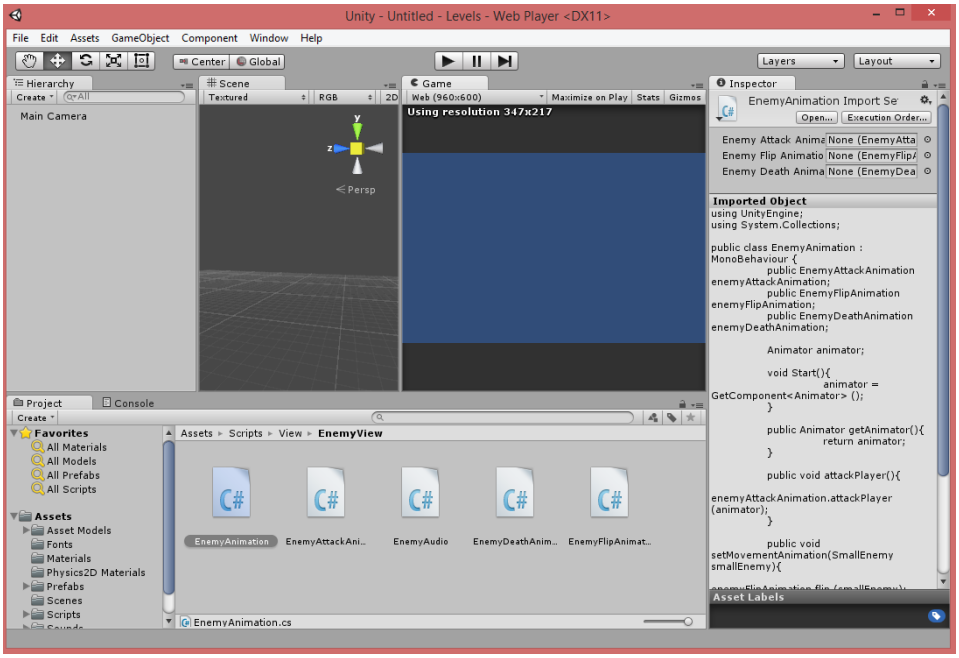
Una vez realizado el documento, se llevó a cabo un estudio de referencias exhaustivo. Se jugó a todos estos juegos para estudiar:

- Qué mecánicas tienen
- Cómo están diseñados los niveles
- Cómo se producen los cambios de mundo
- Diferencia estética de los cambios de mundo
- Cómo son los enemigos
- Cómo son las plataformas sobre las que tiene que moverse el jugador

Con las referencias estudiadas y el documento de análisis realizado, podemos comenzar la fase de prototipado.

5.4. Nociones de Unity

Para poder entender toda la terminología utilizada durante la fase de preproducción, vamos a explicar los conceptos básicos de Unity para facilitar el entendimiento de esta memoria.



Esta es la vista principal de Unity. Cada pestaña es denominada Vista. Existen diferentes vistas esenciales:

- Project: Este es el navegador del proyecto. Podemos acceder a todos los recursos de nuestro proyecto.
- Hierarchy: La jerarquía contiene todos los GameObject de la escena actual.
- Scene: La escena es nuestro sandbox interactivo. Se utiliza para seleccionar y posicionar todos los GameObjects de la jerarquía. Los niveles se construyen manipulando los GameObjects en la escena.

- Game: Esta vista muestra el aspecto final de nuestro juego. Lo que muestra es lo que ve nuestra cámara principal. En el modo de juego podemos pulsar Play y vemos nuestro proyecto en acción
- Inspector: Nos muestra las propiedades y componentes del GameObject seleccionado.

Existen diferentes elementos que son parte de nuestro proyecto y escenas.

- Assets: Recuerdos de nuestro proyecto: texturas, modelos, sonidos, scripts. Tienen representación en el disco duro.
- GameObject: Los GameObject son los elementos fundamentales de nuestro proyecto. Podemos verlo, trabajar e interactuar con él. Puede ser un cubo, una luz, un sistema de partículas. Éstos no tienen representación en el disco duro, solo se encuentran en la escena. Pero en esencia, todos los GameObject son contenedores vacíos. Lo que los diferencian son sus componentes.
- Componentes: Los componentes son añadidos a los GameObjects para aportarles propiedades. Es lo que hace a los GameObject útiles, lo que aporta funcionalidad.

5.5. Fase de prototipado

La fase de prototipado se caracteriza principalmente por realizar pruebas de todas las ideas que hayamos establecido anteriormente y mejorarlas progresivamente a medida que establezcamos su funcionalidad.

5.5.1. Prototipo movimiento de personaje

Uno de los elementos principales que no pueden faltar en un juego es uno o varios elementos que puedan ser controlados por el jugador. Por eso, una de las mecánicas imprescindibles de cualquier videojuego es el movimiento de los elementos que controla el jugador, ya que sin movimiento el juego sería imposible. En nuestro caso, tendremos un personaje controlable, el protagonista de la historia, que va a ser humano y que podrá realizar las siguientes acciones para cambiar la posición en la que se encuentre:

- Caminar
- Correr
- Saltar

Los movimientos del personaje estarán limitados por estas tres acciones, por lo que nuestro primer prototipo consistirá en crear este personaje controlable por el jugador. Para ello, añadimos algunas plataformas sobre las que pueda moverse el jugador y programamos el comportamiento del personaje.

El objetivo de este prototipo es concretar las físicas que van a influir sobre el personaje: velocidad a la que se moverá, fuerza del salto, gravedad, etc. Aunque estos valores pueden

variar en el futuro, debemos hacer que estas variables puedan ser fácilmente modificables para que puedan ser fácilmente adaptables en un futuro.

5.5.2. Prototipo cambio de escenario

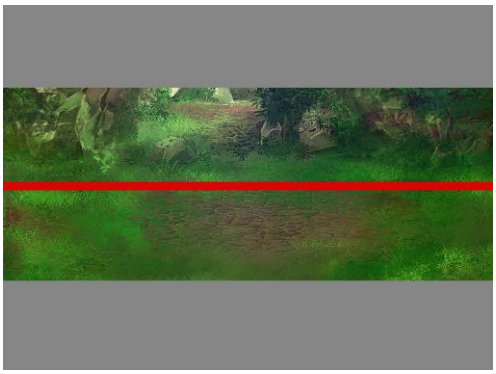
La mecánica de cambio de escenario es la mecánica más compleja de las que hemos planteado. Por ello, hemos planteado la interacción del jugador para llevar a cabo esta mecánica de varias maneras:

1. **Pulsación de botones arriba y abajo**
2. **Puzzle de gatillos**
3. **Efecto de aumento**
4. **Mundos transparentes**

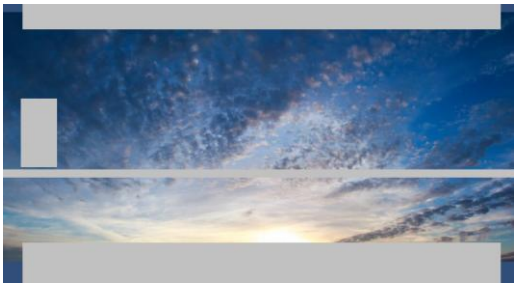
Explicaremos en detalle cada una de estas propuestas, su prototipado y las decisiones tomadas sobre cada una de ellas.

1. Pulsación de botones arriba y abajo

En esta propuesta de cambio de mundo, tenemos dos paneles que simularían unos párpados. Uno de los párpados se moverá hacia la parte superior de la pantalla, y el otro párpado se moverá hacia la parte inferior de la pantalla. Al pulsar el botón arriba y abajo, se moverá el párpado correspondiente. Para que el cambio de mundo se produzca de manera correcta, ambos párpados deben alinearse en la mitad de la pantalla.



Para ello, controlamos cada párpado con un botón diferente, de manera que al pulsar uno de ellos, uno de los párpados vaya bajando progresivamente y pare a mitad de pantalla. La idea detrás de esta mecánica de cambio de mundo es pulsar los dos botones al mismo tiempo, para que ambos párpados bajen de manera equitativa hasta la mitad de la pantalla. Una vez ambos párpados coincidan, se cambiará de mundo, pero los párpados seguirán en la misma posición si el botón sigue pulsado. Una vez se deje de pulsar el botón, cada párpado volverá a su posición original de manera progresiva.

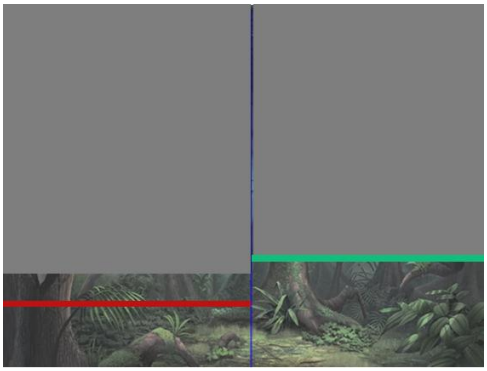


Antes de realizar este prototipo pensábamos que esta mecánica era viable ya que entraba dentro de la temática de sueños (es como si cerraras los ojos y despertaras en otro mundo), es descartada por dos razones:

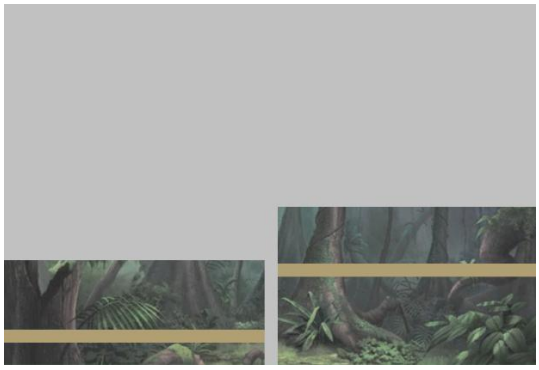
- Resulta muy incómoda de llevar a cabo (no es sencillo pulsar los botones de arriba y abajo al mismo tiempo)
- Esperar a que ocurra la transición resulta monótono para el jugador.

2. Puzzle de gatillos

El cambio de mundo se realizará con los gatillos del mando de XBOX 360, simulando unos párpados. El jugador tendrá que pulsarlos de manera que queden alineados con una línea que aparecerá en pantalla al comenzar a pulsarlos.



Cuando ambos "párpados" estén en la línea, se terminará de pulsar los dos gatillos al máximo y al soltarlos habrá un efecto de barrida hacia arriba del negro que pasará a ser el mundo contrario.



Esta era una de las mecánicas favoritas para el cambio de mundo, ya que implicaba la utilización de los gatillos de manera original. Sin embargo, con el prototipo descubrimos que las primeras veces que el jugador utilizara esta mecánica podría ser divertido, pero en fases posteriores el cambio sería lento y tedioso, por lo que empezamos a buscar algo más dinámico y rápido, pero que nos siguiera dando juego.

3. Efecto de aumento

Una vez prototipadas varias mecánicas de cambio de mundo con efecto de párpado, y al concluir que este tipo de mecánicas era muy lenta, decidimos empezar a plantear mecánicas

de cambio rápido: pulsar un botón y que el cambio ocurra rápidamente, sin ninguna clase de puzzle.

La primera mecánica de este tipo que planteamos consistía en un efecto de aumento para el cambio de mundo: al pulsar el botón correspondiente, se cambiaba de un mundo a otro desde el centro de la pantalla formando un círculo, y se extendía este cambio de mundo hacia los bordes.

Este efecto lo conseguimos gracias a los shaders. Los shaders son efectos gráficos que asociamos a las texturas, de manera que modifican el resultado visual de un material renderizado.

En este caso, programaremos un shader que funcione como una máscara de profundidad, de manera que prevenga dibujar objetos cuando estén ocluidos por la máscara. De esta manera, el mundo real no dejará ver los objetos del mundo de los sueños a no ser que se pulse un botón. Para ello, necesitamos dos objetos:

1. Un objeto que utilice esta máscara de profundidad. Será dibujado después de objetos opacos, y no dejará que otros objetos se dibujen detrás.

2. Objetos que vayan a ser enmascarados.

El objeto que utilice la máscara deberá tener el siguiente *shader*:

```
Shader "Masked/Mask" {
	SubShader {
		Tags {"Queue" = "Geometry+10" }
		ColorMask 0
		ZWrite On
		Pass {}
	}
}
```

Este shader renderiza la máscara detrás de toda la geometría regular, pero antes de toda la geometría enmascarada y los elementos transparentes. Además, no dibujamos en los canales RGBA, sino en el buffer de profundidad de elementos.

Los objetos que vayan a ser enmascarados tienen el siguiente script:

```
using UnityEngine;

[AddComponentMenu("Rendering/SetRenderQueue")]
```

```
public class SetRenderQueue : MonoBehaviour {

	[SerializeField]
	protected int[] m_queues = new int[]{3000};

	protected void Awake() {
		Material[] materials = renderer.materials;
		for (int i = 0; i < materials.Length && i <
	m_queues.Length; ++i) {
			materials[i].renderQueue = m_queues[i];
		}
	}
}
```

Con este código, establecemos el RenderQueue del objeto. El RenderQueue cambia el orden de los objetos que se van a renderizar. Esto instanciará los materiales, por lo que no interferirá con otros renderers que referencien el mismo material.

A pesar de hacer estas pruebas sobre el fondo de nuestro nivel, el efecto no era el esperado, el círculo no se apreciaba de manera correcta, y nos parecía demasiado complejo llevar este método a todos los elementos del escenario, por lo que descartamos la idea por no tener suficiente documentación y no proporcionar el efecto que buscábamos.

4. Mundos transparentes

La cuarta dinámica de cambio de mundo que planteamos consistía en lo siguiente: si pulsamos el gatillo rápidamente, el cambio de mundo se produce de manera inmediata. Sin embargo, si pulsamos el gatillo lentamente, y lo dejamos a media pulsación, podemos ver los elementos del mundo en el que no estamos semi transparentes. Como seguimos en el mismo escenario, los elementos del mundo contrario no son accesibles para el jugador, pero éste se puede hacer una idea de qué va a ocurrir si pulsa el gatillo completamente.

Para ello, ponemos una barra de carga, que indica cuánto más tienes que pulsar el gatillo para que el cambio se produzca. Si dejamos de pulsar cuando los objetos siguen siendo semi transparentes, el cambio no se producirá.

Para este cambio de mundo tenemos que realizar una diferenciación entre la parte visual y la parte tangible de una plataforma:

- Las plataformas solo serán tangibles en su mundo, es decir, que el jugador puede saltar y seguir moviéndose sobre ellas.
- Mientras cambiamos de mundo lentamente podremos ver las plataformas del mundo contrario semi transparentes, pero no estarán activas, por lo que no podremos saltar y movernos sobre ellas.

PLATAFORMA INTANGIBLE

(NO SE PUEDE PISAR)

PLATAFORMA TANGIBLE

(SE PUEDE PISAR)

La barra superior nos indica que cuando la barra naranja llegue a tapar por completo la barra violeta, se cambiará de mundo. Los elementos en violeta son los del mundo real, los elementos en naranja son los del mundo de los sueños. La superposición de dos elementos que están en ambos mundos da lugar a un color rojizo.

Este cambio de mundo da mucho más juego que los anteriores, ya que podremos basar el diseño de niveles con ciertas "trampas" y puzzles en los que el jugador deba anticiparse a lo que viene a continuación con el cambio de mundo. Además, es una mecánica rápida y que puede utilizarse de dos maneras completamente diferentes, adaptándose a la jugabilidad de cada persona.

5.5.3. Prototipo runner

En la fase de toma de decisiones sobre nuestro juego era necesario estudiar toda clase de mecánicas disponible. Además de que el juego fuera un juego plataformas en 2D, otra de las opciones era que fuera un juego de tipo runner. Los juegos runners son juegos plataformas en esencia, la diferencia radica en que el personaje nunca se está quieto, lo ves en movimiento todo el rato y no puedes pararlo. El jugador debe encargarse de superar los obstáculos que se encuentren en el camino del personaje. Este tipo de juegos simplifica las mecánicas considerablemente y aporta un componente de desafío, el jugador buscará llegar cada vez más lejos. Algunos ejemplos de este tipo de juegos son Temple Run, BIT.TRIP Runner y Robot Unicorn Attack.

Con este prototipo queremos conocer la viabilidad de realizar un juego de este tipo. Teniendo en cuenta la mecánica estrella de nuestro juego, el cambio de mundo, y adaptándola a los juegos de tipo runner, establecemos los siguientes elementos para nuestro prototipo:

- Un personaje cuyo único movimiento disponible es el salto.
- Plataformas que estarán en el mundo real
- Plataformas que estarán en el mundo de los sueños
- Una acción que permita cambiar de mundo, de manera que las plataformas del mundo nuevo se activan y las del mundo anterior se desactivan.

En primer lugar prototipamos el personaje y sus movimientos. Añadimos el personaje que habíamos prototipado anteriormente, pero lo modificamos de manera que solo responda al control de salto. También tenemos que hacer que el personaje siempre esté moviéndose hacia la derecha, y hacer que la cámara siempre siga al personaje, de manera que nunca pueda moverse más allá de lo que vemos en pantalla.

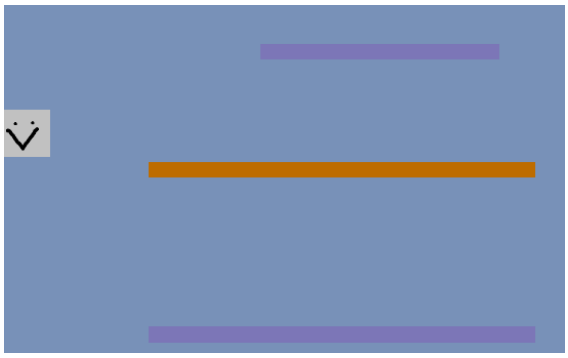
A continuación añadimos las plataformas. Creamos seis elementos que se estarán creando de manera infinita en nuestro escenario:

- Plataformas que solamente se puedan pisar en el mundo real, de tres tamaños diferentes.
- Plataformas que solamente se puedan pisar en el mundo imaginario, de tres tamaños diferentes.

Con la diferencia de tamaños de las plataformas añadimos un componente más aleatorio, ya que no puedes anticipar qué tipo de plataforma vendrá a continuación.

El único elemento que nos falta es la acción de cambio de mundo. Programamos los componentes para que con la pulsación de un botón, cambiamos automáticamente de mundo. El cambio de mundo se podrá diferenciar gracias a las plataformas: cuando cambiamos de mundo, activamos las plataformas nuevas y desactivamos las anteriores. Pero hacerlas aparecer y desaparecer simplemente haría el juego muy difícil, ya que el jugador no podría calcular donde están las plataformas del otro mundo y no sabría si debería cambiar o no.

Programamos el prototipo de manera que existan tres puntos en el escenario donde se estarán generando las plataformas, de manera aleatoria, antes de que el personaje pueda llegar a ellas.



Plataformas tangibles : naranjas

Plataformas tangibles: violetas

Realizamos tres pruebas de controles para ver cuáles son más intuitivos, divertidos, complicados, etc.

- Control 1: Salto simple. El salto y el cambio de mundo se realizan con el mismo botón.
- Control 2: Salto simple. El salto se realiza con un botón y el cambio de mundo se realiza con otro botón diferente.
- Control 3: Salto doble. El salto se realiza con un botón y el cambio de mundo se realiza con otro botón diferente.

Con estos tres prototipos diferentes de controles llegamos a las siguientes conclusiones:

- Sobre los controles: El control 1 es muy intuitivo y muy sencillo, pero da menos libertad al jugador. Sin embargo, los controles 2 y 3 aportan un componente más estratégico y dan más posibilidades al jugador. El salto simple hace que el juego sea ligeramente más complicado.
- Sobre la idea de desarrollar un juego runner: Un juego runner de por sí solo con esta mecánica no es del todo viable ya que el jugador probablemente se canse rápido de repetirlas infinitamente. Podría ser una opción viable si a nuestro juego añadimos algunas fases de este estilo, pero no parece factible basar todo el juego como un runner.

5.5.4. Prototipo enemigo pequeño

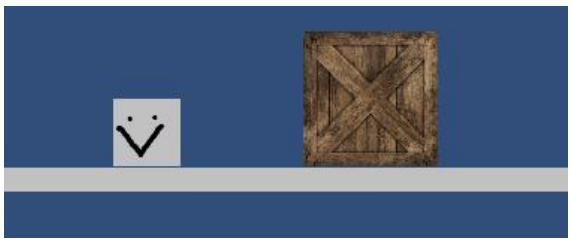
Una vez concluidas las mecánicas principales, necesitamos comenzar a prototipar elementos que nos permitan aportar dinamismo al juego: obstáculos. Uno de los obstáculos clásicos de los videojuegos que comenzamos a prototipar primero es el enemigo.

Programamos todas las físicas que le afectan, y añadimos las colisiones pertinentes para la interacción con el jugador: al estar en contacto el personaje principal y el enemigo, perdemos una vida. Además, el enemigo se moverá de derecha a izquierda continuamente una distancia predeterminada.

Con este prototipo, además de determinar cómo queremos que se comporten los enemigos, nos empieza a dar elementos con los que poder jugar cuando vayamos a diseñar los niveles.

5.5.5. Prototipo cajas

Otro de los elementos con los que queríamos contar en nuestro juego era el uso de cajas para alcanzar sitios donde el jugador a priori no puede llegar. Para ello, establecemos este nuevo elemento en nuestro prototipo. Este elemento incluye físicas para que el jugador sea capaz de arrastrar la caja con el personaje mientras éste se está moviendo, pero que tenga una velocidad de empuje moderada. Además, no se ve afectada por los enemigos, y éstos no pueden empujarla.



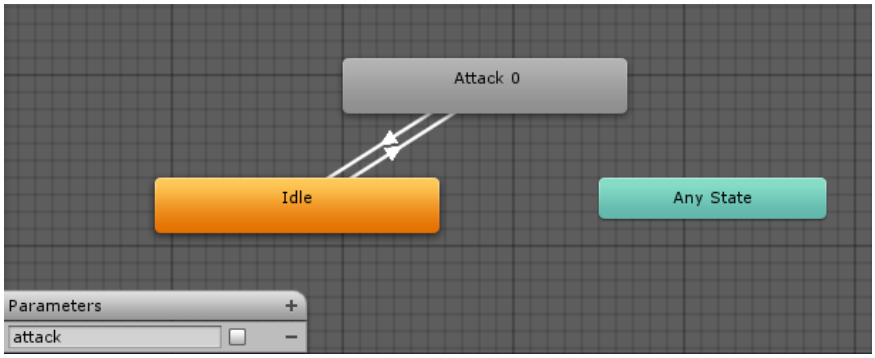
Este es otro de los elementos primordiales que incluiremos en nuestro diseño de niveles para proporcionar al jugador de diferentes componentes con los que interactuar.

5.5.6. Prototipo ataque

Aunque en nuestro esbozo inicial no se contemplaba la posibilidad de que el personaje incorporara una mecánica de ataque, en este punto de la fase de prototipado decidimos experimentar con esta mecánica para conocer las posibilidades que nos podía ofrecer.

Utilizando como elemento principal el personaje que habíamos prototipado antes, añadimos una animación de ataque sencilla. A continuación explicaremos cómo llevamos a cabo la animación de ataque, ya que no acabó utilizándose esta animación en el desarrollo.

Para realizar la animación fue necesario utilizar el sistema de animaciones implementado en Unity: Mecanim. Con esta herramienta, podemos controlar las animaciones de nuestros objetos y realizar transiciones entre dichas animaciones cuando se cumplen ciertas condiciones. Para ello, debemos programar una máquina de estados. Esta máquina la representamos en Unity con el recurso "Animator Controller".

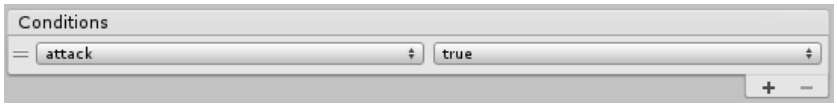


Con esta máquina de estados estamos definiendo dos estados: Idle y Attack.

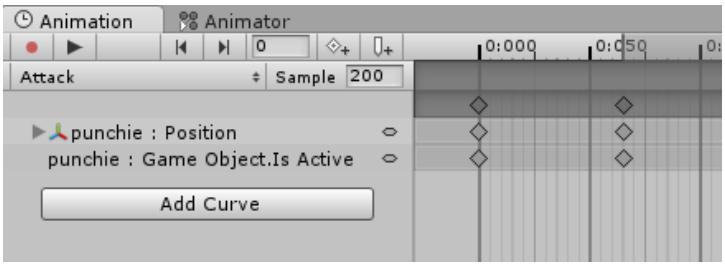
- Idle es el estado en el que se encuentra el personaje cuando está parado o moviéndose.
- Attack es el estado en el que se encuentra el personaje cuando se pulsa el botón de ataque.

Establecemos una nueva variable de tipo booleano denominada *attack,* que podemos modificar desde el código para llevar a cabo las transiciones.

Para cambiar de estado, necesitamos especificar en el editor bajo qué condiciones se pasará de una a otra, por lo que añadimos una nueva transición de Idle a Attack bajo la siguiente condición: cuando el booleano *attack* sea verdadero, se dará la transición. Especificamos el caso contrario para transicionar de Attack a Idle.



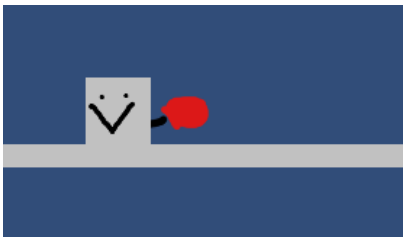
Como no disponemos de modelos 3D para nuestros prototipos aún, necesitamos crear una nueva animación para el ataque, por lo que utilizamos la herramienta Animator de Unity para crear una nueva animación utilizando GameObjects.



En Sample, debemos elegir cuánto tiempo durará nuestra animación, en este caso medio segundo. Modificaremos dos componentes del personaje en el tiempo.

El primer elemento a modificar es la posición de *punchie*¸ nuestro sprite que representa un guante de boxeo. En medio segundo se moverá hacia adelante, para que parezca que el personaje mueve su brazo para realizar el ataque.

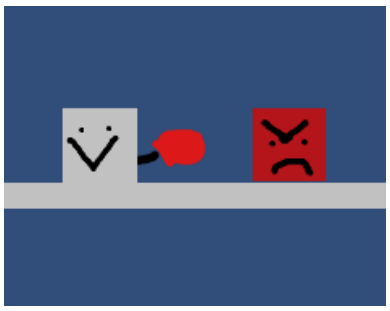
El segundo elemento es que el *punchie* esté activado solamente durante la animación. Es decir, que el sprite estará desactivado siempre y se activará solamente cuando el jugador pulse el botón correspondiente. Con esto evitamos que se estén dando colisiones indeseadas mientras no deberían activarse.



Una vez terminamos nuestra animación, realizamos los cambios programáticos para modificar la variable booleana *attack* que habíamos establecido previamente en nuestra máquina de estados.

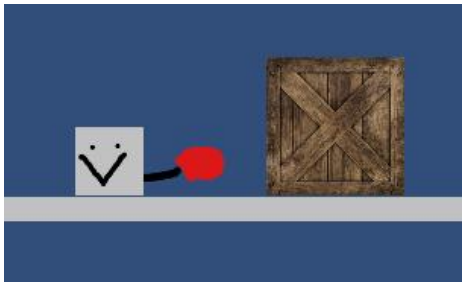
Una vez establecida la máquina de estados y la nueva animación, comprobamos la velocidad del ataque y que no existan problemas con la detección de colisiones con el jugador.

A continuación, añadimos un nuevo elemento para interactuar con el ataque: el enemigo previamente establecido. Con este prototipo comprobamos que podemos responder bien al enemigo con este ataque.



Es necesario tener en cuenta que la capacidad de noquear al enemigo debe ocurrir una vez se activa la animación del ataque y no antes, por lo que tenemos que tener en cuenta que las físicas de colisiones no actúen mientras no se está atacando para que funcione correctamente.

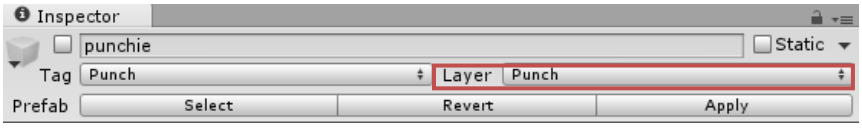
Por último, para terminar de prototipar el ataque, consideramos la posibilidad de que el jugador interactúe con las cajas que habíamos implementado anteriormente con la animación de ataque, para simular un empuje de las cajas mayor, así que añadimos este elemento a nuestro prototipo.



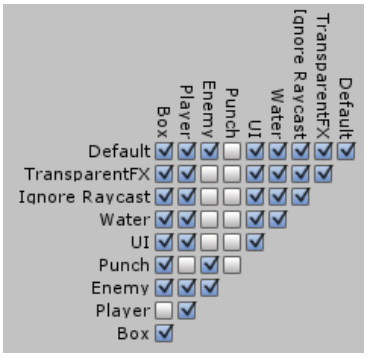
La manera de que esta mecánica funcionara correctamente era que el jugador necesitara de alguna manera recurrir al ataque para mover la caja, en lugar de empujar la caja con el simple movimiento del personaje. Es importante establecer una relación entre las físicas de la caja y la fuerza del ataque para que sea mucho más ágil mover la caja con esta mecánica que con el simple movimiento del personaje, añadiendo una fuerza proporcional a ambos elementos solamente cuando ocurre el ataque.

Para este prototipo fue necesario realizar modificaciones en la matriz de físicas 2D, la cual especifica qué tipos de objetos pueden colisionar o no con otros objetos de la escena. Utilizaremos el sistema de capas de Unity para dicho propósito.

En la vista Inspector de Unity, podemos especificar la capa en la que queremos que se encuentre cada objeto:



La matriz de físicas 2D modifica las colisiones entre diferentes capas, por lo que seleccionamos para cada elemento principal del prototipo la capa en la que se encuentra. A continuación, establecemos qué colisiones se pueden dar y qué colisiones se ignoran entre capas:



En este caso, especificamos que todos los objetos que se encuentren en la capa Punch (puñetazo) solo puedan colisionar con enemigos y cajas. También podemos especificar que el jugador no pueda colisionar con las cajas, de manera que solamente puede empujarlas con el ataque.

Al realizar este prototipo, nos dimos cuenta de que esta mecánica aportaba más dinamismo al conjunto de mecánicas disponibles, ya que aportaba al jugador una nueva opción para superar estos dos obstáculos:

- Al enfrentarse a un enemigo, el jugador puede esquivarle o atacarle.
- Al mover una caja, el jugador puede empujarla lentamente con el movimiento horizontal del personaje o empujarla rápidamente con el ataque.

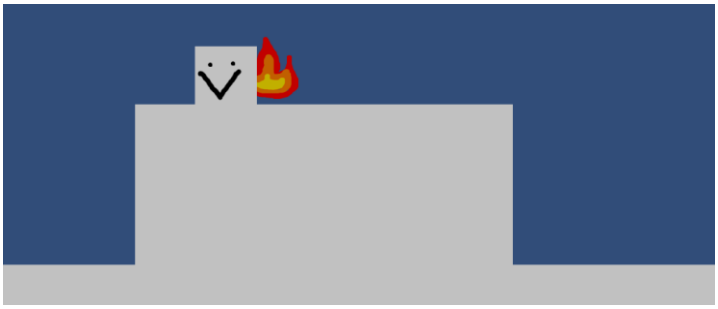
5.5.7. Prototipo enemigo grande

Siguiendo con la búsqueda de elementos de los que disponemos para poblar un nivel, barajamos la opción de añadir interruptores. La nueva mecánica planteada consiste en que el jugador debe encontrar un interruptor que le permita pasar por un sitio que a priori estaba bloqueado.

Para alejarnos de la idea utilizada en la mayoría de videojuegos de "una palanca abre una puerta", desarrollamos la idea de que la "puerta" que bloquea el avance del personaje sea un nuevo enemigo mayor al que el jugador no se pueda enfrentar con el ataque y que tampoco pueda esquivar con un salto. De esta manera, la "palanca" sería un objeto que el jugador puede recoger y que quitaría de en medio al enemigo.

Para testear esta idea, añadimos dos elementos nuevos: un nuevo enemigo, más grande y que no es afectado por el ataque; y una representación del objeto que es necesario recolectar para superar a este enemigo, en este caso un fuego.

De esta manera, prototipamos las reacciones del personaje con el enemigo, siendo inmune al ataque del jugador e infligiendo un punto de daño cuando el personaje se acerca. También tenemos en cuenta que el personaje pueda recoger el fuego, y la reacción del enemigo al recoger el fuego. Esta mecánica aporta dos nuevos elementos a los escenarios del juego, reforzando las mecánicas de puzzle de los escenarios al ser necesario buscar un objeto para continuar el avance.



5.5.8. Prototipo artístico

Hasta este momento, nos habíamos centrado en la parte programática del juego, prototipando todo tipo de mecánicas y elementos para dotar al juego de cierta consistencia. Una vez establecidas las mecánicas principales, y los diferentes elementos que queríamos ver en nuestros escenarios, comenzamos a pensar en la parte artística. En esta fase de prototipado fue necesario buscar todo tipo de recursos gratuitos para comenzar a darle el acabado artístico que queríamos a nuestro juego: modelos 3D y sprites (mapas de bits dibujados).

La primera decisión que tomamos es realizar una mezcla visual y utilizar modelos 3D para personajes, enemigos y elementos de decoración del escenario. Aunque nuestro juego fuera 2D, queríamos utilizar componentes 3D por dos motivos:

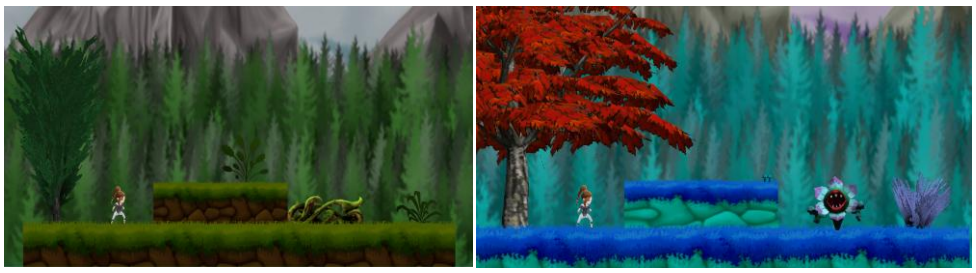
- Como herramienta de aprendizaje
- Como diferenciación visual con respecto a otros juegos 2D.

En primer lugar tuvimos que buscar al protagonista. Nos decantamos por un modelo que incluía las animaciones que necesitábamos para el juego: correr, saltar y atacar. Además, este modelo no incluía un arma, lo cual era idóneo para nuestro juego, debido a que buscábamos que el protagonista diera la apariencia de no ser extremadamente poderoso.

En segundo lugar, decidimos la ambientación del juego. Elegimos un bosque por varios motivos:

- Existía una gran cantidad de recursos como árboles, plantas, plataformas, etc. ambientadas en un ambiente forestal.
- Podíamos acceder a enemigos que encajaran mejor con esta temática: animales, plantas, árboles, etc.
- El cambio de mundo nos podía dar más juego ya que podíamos poner plantas comunes en el mundo real y cambiar los colores de manera fácil para el mundo de los sueños.

Por último, tomamos una decisión con respecto a los enemigos: solo estarían en el mundo de los sueños. Con esto proporcionamos una ilusión de realismo mayor a nuestro mundo real y de irrealidad a nuestro mundo de los sueños.



Este prototipo no constaba de ninguna clase de mecánicas ni elementos anteriormente citados, simplemente establecimos diferentes recursos en el escenario hasta que consiguiéramos el efecto visual que buscábamos.

5.5.9. Prototipo artístico + cambio de escenario

Cuando estuvimos conformes con el prototipo artístico de nuestro juego, queríamos ver cuáles de los sistemas de partículas resultaba más adecuado para nuestro cambio de escenario.

Concluimos que el cambio de escenario se veía mejor con el sistema de partículas amarillas, ya que daba una sensación de *flash* más adecuada.

5.5.10. Prototipo de sonidos

Los sonidos es uno de los apartados artísticos que se suelen dejar para el final, pero es uno de los apartados que causan más impresión en los usuarios. Por ello, queríamos incorporar desde el principio al prototipo artístico la música que acompañaría al juego. Realizamos una selección de música acorde a la estética del juego, música tranquila pero no demasiado pausada, y probamos una colección de canciones disponibles de manera gratuita. Después de mucho descartar, seleccionamos la música llamada "Air Hockey Saloon", de Chris Zabriskie.

5.5.11. Nuevos obstáculos

Teniendo preparado todo el apartado artístico, queríamos prototipar nuevos elementos inspirados en algunos recursos visuales que encontramos.

PUENTES

Partiendo de unos modelos 3D de puentes encontrados, decidimos convertirlo en un tipo de plataforma disponible en el mundo de los sueños. No hubo que prototipar ningún comportamiento nuevo, pero sí adaptarlo para que funcionara como una plataforma.

ESPINAS

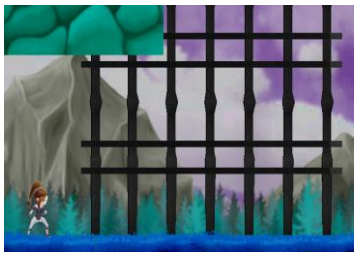
Encontramos unos sprites que representaban una hilera de espinas y decidimos prototipar éstos como un obstáculo, de manera que el personaje recibiera daño al colisionar con ellos.

NUBES

Al buscar fondos que se vieran tras el escenario, encontramos unos sprites de nubes. Con ellos, prototipamos un nuevo tipo de plataforma: una plataforma que se podía atravesar si se saltaba desde abajo, pero que nos permitía quedarnos sobre ella cuando la cruzábamos.

PUERTAS VERJA

Otro de los recursos que encontramos interesantes para añadir a nuestra lista de elementos eran las verjas: puertas que podían encontrarse abiertas o cerradas dependiendo del mundo donde te encontraras. Prototipamos este elemento para que cambiara su posición en función del mundo donde nos encontráramos.



ZONAS DE CAÍDA LIBRE

El último obstáculo nuevo que nos planteamos era la caída libre: una zona donde la cámara se alejaba del personaje y mostraba un hueco por donde podíamos caer para continuar el nivel. Nuestra idea era combinarlo con los pinchos para crear nuevas posibilidades para el posterior diseño de niveles. Para llevar a cabo este prototipo, fue necesario generar zonas donde al entrar, la cámara hiciera un efecto de zoom out, es decir, que se alejaran del personaje, con la ayuda de disparadores (*triggers*).

5.5.12. Versión alfa

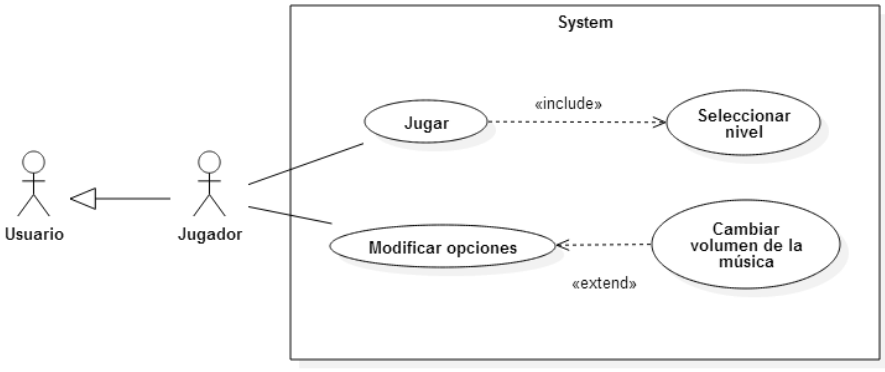
Terminados todos los prototipos individuales, llevamos a cabo la primera versión Alfa del juego. Para ello, comenzamos a unir todos los elementos y mecánicas prototipados durante esta fase y los incluimos en esta versión. Además, transformamos el prototipo en una prueba de nivel, diseñando un protonivel corto que abarcara todos los prototipos anteriores.

Con este protonivel concluimos:

- Nuestras mecánicas funcionan de manera correcta.
- Nuestros elementos de juego funcionan correctamente.
- Todo se adecua a la estética del juego.

5.6. Casos de uso

Realizamos un análisis de los casos de uso



La simplicidad de los casos de uso viene dada por la naturaleza de la aplicación. El jugador lo único que busca es jugar, y es el objetivo principal. El usuario además podrá modificar las opciones del juego. La única opción de este prototipo será cambiar el volumen de la música.

Si el jugador comienza a jugar, será obligatoria la selección de nivel.

Especificación de casos de uso

Nombre del caso de uso	Jugar
Descripción	Comienzo de partida
Actores	Jugador
Precondiciones	-
Postcondiciones	-
Flujo básico	1. El usuario selecciona "Jugar" 2. INCLUDE Caso de Uso Seleccionar nivel 3. Fin del caso de uso

Nombre del caso de uso	Modificar opciones
Descripción	Modificación de las opciones de partida
Actores	Jugador
Precondiciones	-
Postcondiciones	-
Flujo básico	1. El usuario selecciona "Opciones" 2. El sistema muestra las opciones disponibles 3. EXTENDS Caso de Uso Cambiar volumen de la música

Nombre del caso de uso	Seleccionar nivel
Descripción	Selección de nivel a jugar
Actores	Jugador
Precondiciones	Haber comenzado una partida
Postcondiciones	-
Flujo básico	1. El sistema muestra los niveles disponibles 2. El jugador selecciona uno de los niveles disponibles 3. Fin de caso de uso

Nombre del caso de uso	Cambiar volumen de la música
Descripción	Cambio del volumen de la música del juego
Actores	Jugador
Precondiciones	-
Postcondiciones	Volumen de la música modificado
Flujo básico	1. El sistema muestra el nivel actual de la música 2. El jugador modifica el nivel de la música. 3. Fin de caso de uso

6. Diseño y arquitectura

En esta etapa comenzamos con el diseño y la arquitectura de nuestro primer jugable.

6.1. Documento de diseño

Antes de comenzar con el desarrollo, establecemos un documento de análisis de nuestro juego, que nos ayudará a delimitar el alcance de nuestro juego y cómo implementaremos cada elemento. Podemos encontrar este documento en el Anexo 1: Documento de diseño.

Durante la fase de preproducción hemos tomado decisiones sobre nuestro juego. Todas estas decisiones se encuentran recopiladas en este documento, al que necesitaremos atenernos para desarrollar el videojuego.

6.2. Arquitectura del software

Antes de comenzar la implementación, necesitamos establecer la arquitectura que tendrá nuestro videojuego. En nuestro caso, hemos optado por una arquitectura muy similar al patrón Modelo-Vista-Controlador (MVC, Model-View-Controller), adaptado a las peculiaridades de Unity (. El patrón MVC nos permite separar los datos de la lógica de nuestro software.

- **Modelo:** Es la representación de los datos con los que nuestro sistema interviene. En nuestro caso, serán todas aquellas clases que representen una entidad en nuestro videojuego (personaje, enemigo, plataformas, etc.).
- **Vista:** Es la presentación del modelo. En nuestra arquitectura, serán todas aquellas clases que cambien de manera visual nuestros modelos o generen sonidos (animaciones del personaje, animaciones de enemigos, partículas de cambio de mundo, sonidos, etc.).
- **Controlador:** Los controladores responden a eventos (inputs, selecciones en el menú, eventos enviados por colisiones y disparadores) y modifican los modelos y vistas en base a estos eventos. Tendremos diversos controladores en nuestro juego (controladores de nivel, controladores de personaje, controladores de cambio de mundo, etc.)

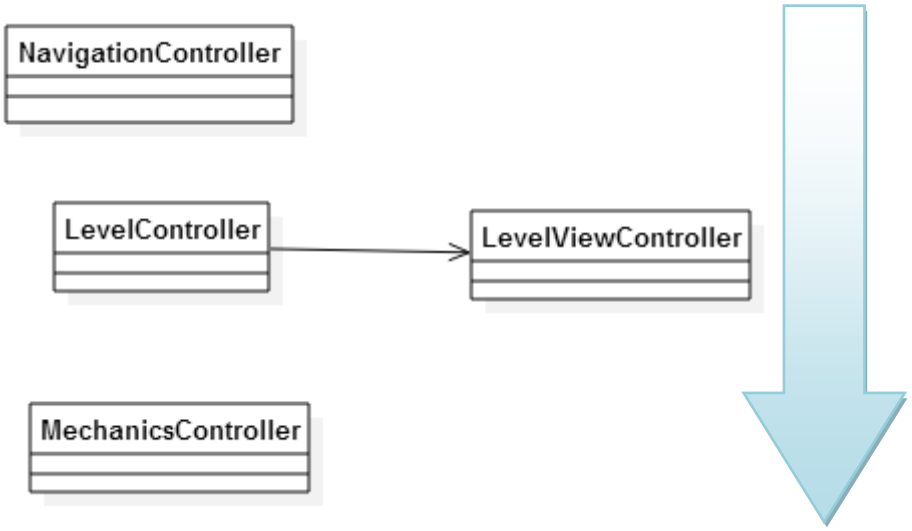
La arquitectura completa se encuentra en el Anexo 2: Arquitectura del software. En este apartado desglosaremos los aspectos básicos de nuestra arquitectura.

CONTROLADORES

Los controladores son clases que se encargan de responder a diversos eventos. Estos eventos pueden ser de varios tipos:

Entradas de los periféricos de control que utilice el usuario.
Eventos lanzados por colisiones y disparadores.
Selecciones del usuario en la interfaz del juego.

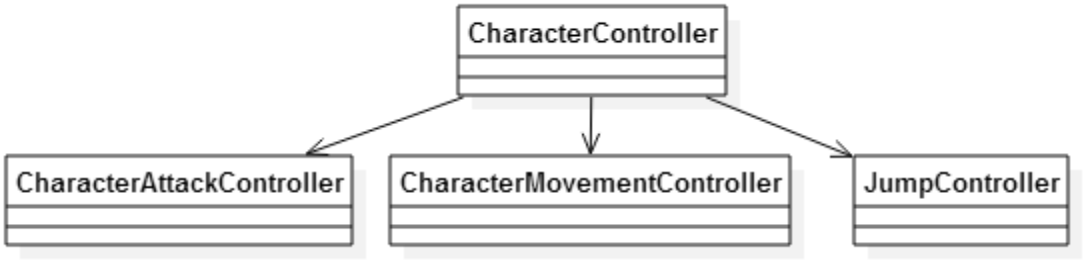
Los tres controlares principales de los que dispondremos serán los siguientes:



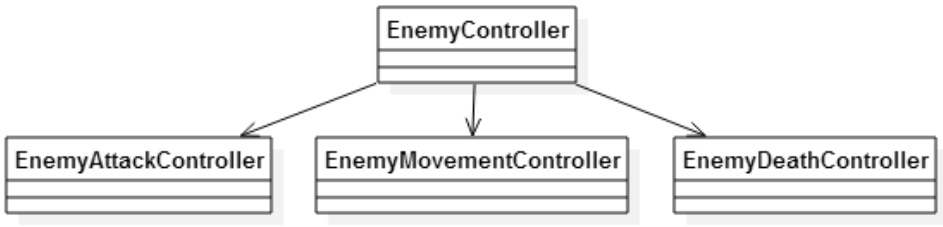
Estos controladores, a pesar de no tener asociación directa, establecen una jerarquía, ya que los controladores superiores controlan aspectos más generales de nuestra aplicación, y los controladores inferiores controlan aspectos concretos.

- NavigationController: Se encarga de transicionar entre escenas, cargar los diferentes niveles de nuestro videojuego y guardar los niveles desbloqueados.
- LevelController: Se encarga de controlar todos los aspectos de cada nivel: vidas disponibles, tiempo que lleva el jugador en ese nivel, puntos de control activados por el jugador, etc.
- LevelViewController: Se encarga de mostrar en pantalla la información necesaria relacionada con el nivel.
- MechanicsController: Este controlador engloba diferentes controladores que se encargan de las mecánicas y elementos principales de nuestro juego.

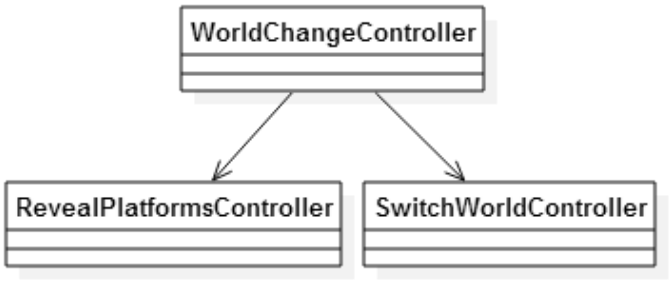
Veamos un detalle de cada MechanicController:



- CharacterAttackController: Gestiona los datos del modelo del personaje relacionados con el ataque.
- CharacterMovementController: Modifica la posición del personaje en el eje horizontal.
- JumpController: Determina si el personaje puede realizar un salto o no, y en caso afirmativo modifica la posición del personaje.



- EnemyAttackController: Hace que el enemigo realice el ataque si se cumplen las condiciones.
- EnemyMovementController: Actualiza la posición del personaje
- EnemyDeathController: Elimina la visualización y las referencias al enemigo.

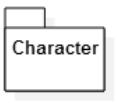


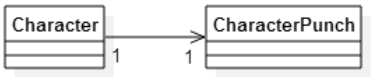
- RevealPlatformsController: Muestra las plataformas de ambos mundos al mismo tiempo, variando sus valores alfa para que sean más o menos transparentes.
- SwitchWorldController: Modifica todos los elementos implicados en el cambio de mundo: plataformas tangibles, luz, fondo, elementos decorativos, etc.

MODELOS

Tenemos cuatro clases de modelos, dependiendo de los elementos que estemos representando en nuestro juego:

Estos modelos serán la representación lógica del personaje, los enemigos, los elementos del entorno y los elementos intangibles. Explicaremos con poco grado de detalle en este apartado cada uno de ellos.

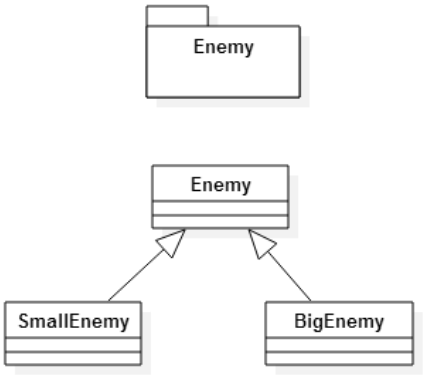




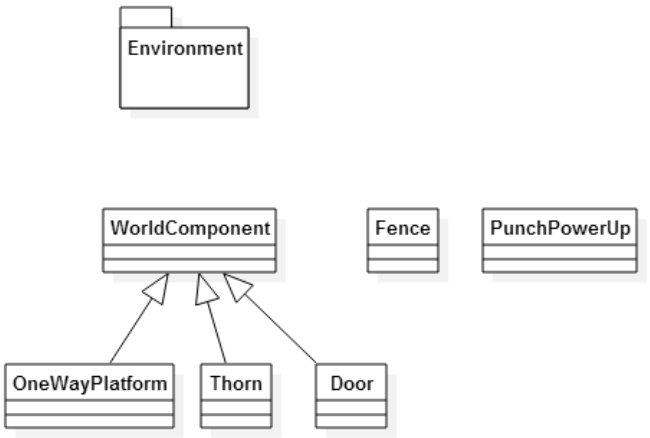
El personaje vendrá representado por dos modelos:

- Character: Contendrá todos los datos relacionados con el personaje: velocidad máxima, capacidad de salto, lado al que está mirando, etc.

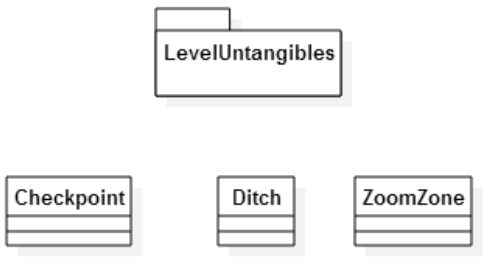
- CharacterPunch: Contendrá todos los datos relacionados con el ataque del personaje: si está activado o no, si tiene el power up, etc.



- Enemy: Datos relacionados con el enemigo: velocidad, lado al que está mirando, etc.
- SmallEnemy: Datos relacionados con los enemigos pequeños.
- BigEnemy: Datos relacionados con los enemigos grandes.



- WorldComponent y clases que heredan de ella: Modelo de los diferentes elementos del mundo que están afectados por el cambio de mundo.
- Fence: Puertas verja que cambian de posición según el mundo en el que se encuentren.
- PunchPowerUp: Power up que el jugador recoge y aumenta su capacidad de ataque.

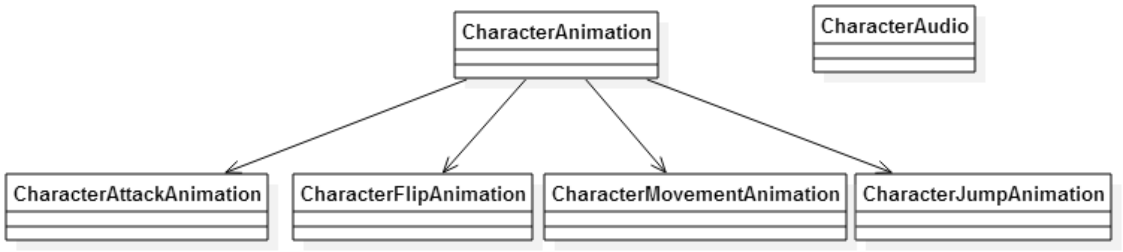


- Checkpoint: Representa los puntos de control para llevar al jugador a una posición intermedia si pierde todas las vidas, y no devolverlo al principio del nivel.
- Ditch: Representa los fosos en los que puede caer el jugador y que hace que pierda todas sus vidas.

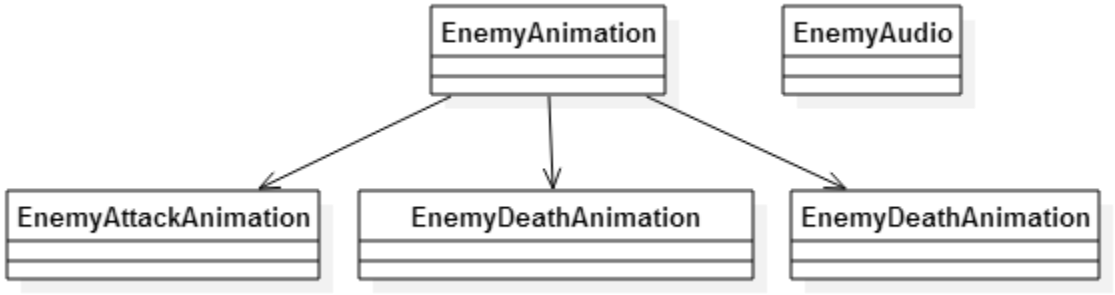
- ZoomZone: Representa las zonas de zoom en las que el jugador entra y la cámara se aleja para dejarle ver mejor los obstáculos a los que se tiene que enfrentar.

VISTAS

Las vistas representan los elementos visuales y auditivos que modificaremos bajo ciertas condiciones.



El personaje tendrá dos clases: una que controle las animaciones y otra que controle sonidos.



Los enemigos serán similares al personaje principal, tendrán sus propias clases que controlen animaciones y otra clase que controle los sonidos.

El cambio de mundo tendrá tres clases de vista: el fondo del mundo, las partículas que acompañan al cambio de mundo y el sonido del nivel.

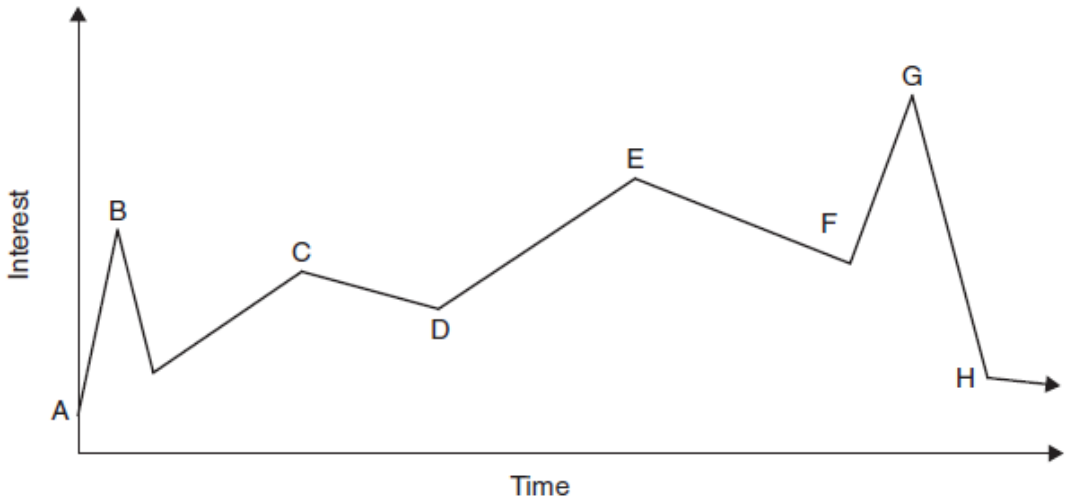
6.3. Diseño de niveles

Para llevar a cabo el diseño de niveles, primero fue necesario llevar a cabo un estudio previo de los buenos hábitos del diseño de niveles. Empezaremos a estudiar los diferentes apartados a tener en cuenta para un buen diseño de niveles:

CURVA DE INTERÉS

La calidad de una experiencia de entretenimiento viene dada por el interés que despierta una cadena de eventos en el público. Esta regla se da en cualquier experiencia cuyo objetivo es captar al público, ya sea una obra de teatro, un concierto, un videojuego o una película. El nivel de interés durante la experiencia puede ser representada en una curva de interés. A

continuación ponemos un ejemplo de una curva de interés para una experiencia que tenga éxito.

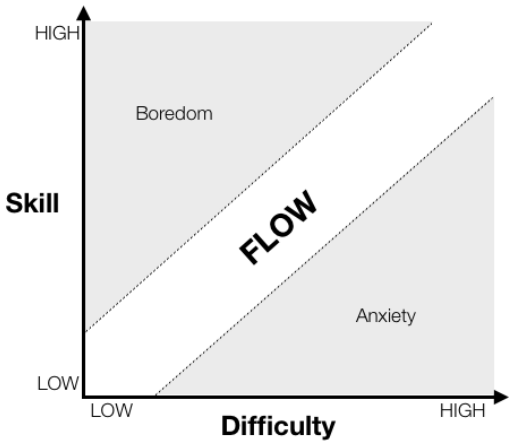


En el punto A, el espectador comienza la experiencia con un cierto grado de interés. Este interés puede venir de la publicidad, las recomendaciones, etc. No es recomendable que este interés inicial sea muy alto, o la experiencia será menos interesante.

Una vez comienza la experiencia, debemos tener un evento que sirva de gancho para captar la atención del público (punto B), algo que les haga suponer qué viene a continuación. Si la experiencia está bien organizada, veremos varios puntos cumbre de interés (puntos C y E), y varios puntos donde baja el interés (puntos D y F).

Finalmente, en el punto G, debe haber un punto álgido de interés, un evento que deje al usuario con buena sensación al final de la experiencia. Cuando se haya terminado, el usuario debería mantener cierto grado de interés, si se hace bien, ligeramente más alto que al principio.

Además de tener en cuenta la curva de interés, también debemos tener en cuenta el flow. El flow es el resultado de dos características que se relacionan: la habilidad del jugador y la dificultad del desafío. Cando interactúan surgen diferentes estados emocionales y cognitivos. Cuando la habilidad es muy baja y el desafío, las personas comienzan a sentir ansiedad. Por el contrario, cuando el desafío es muy fácil y la habilidad es muy alta, las personas se aburren. Pero si la habilidad y el reto son proporcionales, entran en un estado de fluidez (flow).



Además, existen cuatro características que encontramos en los desafíos que llevan a un equilibrio entre dificultad y habilidad, aumentando la probabilidad de estados de flow. Estos desafíos son:

Ten objetivos concretos con reglas manejables.
Demanda acciones para alcanzar objetivos que sean acordes a las capacidades de las personas.
Ten feedback constante sobre el rendimiento y el logro de objetivos.
Disminuye distracciones, a la vez que facilitas la concentración.

Teniendo las curvas de interés y el flow en cuenta, comenzamos a diseñar nuestros niveles.

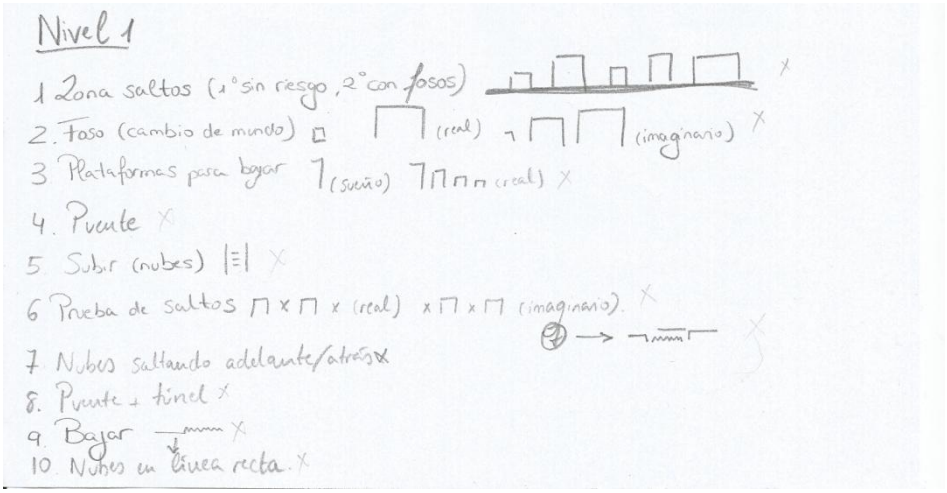
DISEÑO DE NIVELES

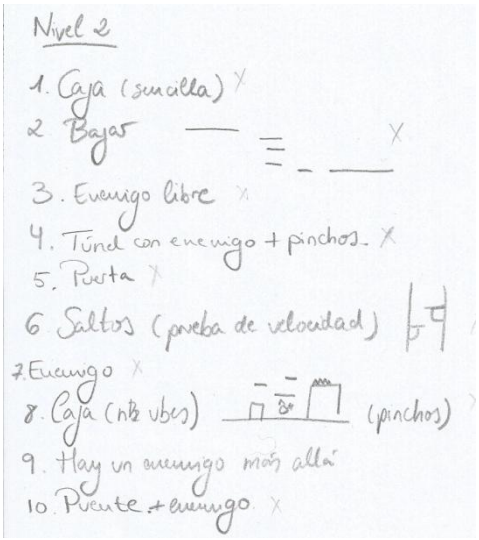
Nuestro primer objetivo era listar todos los elementos y obstáculos que podían encontrarse en nuestros niveles, y hacer una selección de los elementos que queríamos que aparecieran desde el primer nivel y los elementos que queríamos que aparecieran en niveles posteriores. Reservar obstáculos para otros niveles nos ayudaría a mantener la curva de interés a la vez que aumentábamos la dificultad.

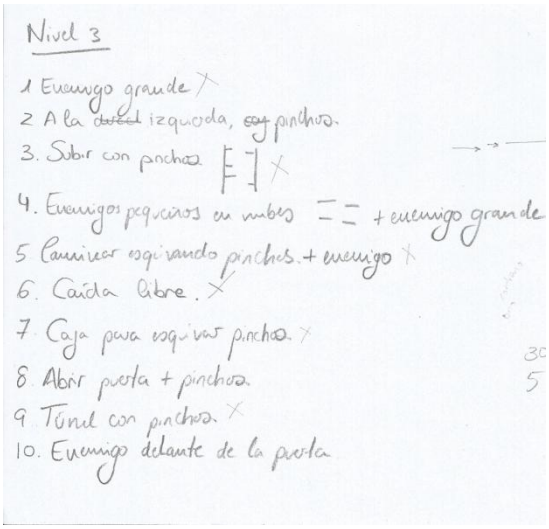
Elementos que encontraremos en los niveles:

- Plataformas: Nivel 1
- Espinas: Nivel 1
- Puente: Nivel 1
- Nubes: Nivel 1
- Cajas: Nivel 1
- Puertas verja: Nivel 2
- Enemigo pequeño: Nivel 2
- Enemigo grande: Nivel 3
- Caída libre: Nivel 3

Para diseñar los niveles, teníamos que incluir de manera progresiva estos elementos mientras aumentábamos la dificultad de algunos puzzles que utilizaran elementos vistos en anteriores niveles. La disposición de los niveles se planteó de la siguiente manera:


Nivel 1
1. Zona saltos (1° sin riesgo, 2° con fosos)
2. Foso (cambio de mundo)
(real)
(imaginario)
3. Plataformas para bajar
(sueño)
(real)
4. Puente
5. Subir (nubes)
6. Prueba de saltos (real)
(imaginario)
7. Nubes saltando adelante/atrás
8. Puente + túnel
9. Bajar
10. Nubes en línea recta

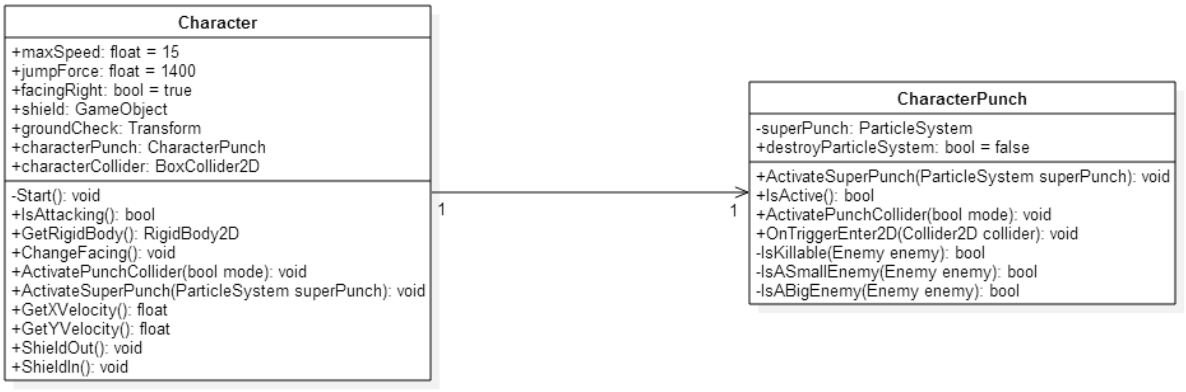

Nivel 2
1. Caja (sandía)
2. Bajar
3. Enemigo libre
4. Túnel con enemigo + pinchos.
5. Puerta
6. Saltos (prueba de velocidad)
7. Enemigo
8. Caja (nubes)
(pinchos)
9. Hay un enemigo más allá
10. Puente + enemigo


Nivel 3
1 Enemigo grande
2 A la izquierda, pinchos.
3. Subir con pinchos
4. Enemigos pequeños en nubes + enemigo grande
5 Caminar esquivando pinchos + enemigo
6. Caída libre.
7 Caja para esquivar pinchos.
8 Abrir puerta + pinchos.
9 Túnel con pinchos.
10. Enemigo delante de la puerta

7. Implementación

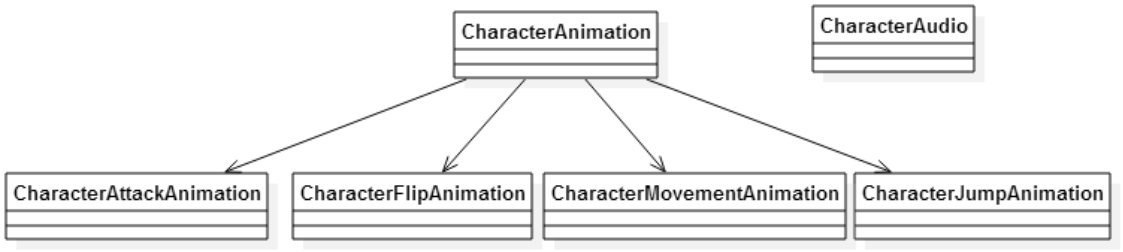
7.1 Implementación del personaje

La implementación del personaje comenzará con la programación del modelo de personaje y del puñetazo de ataque.

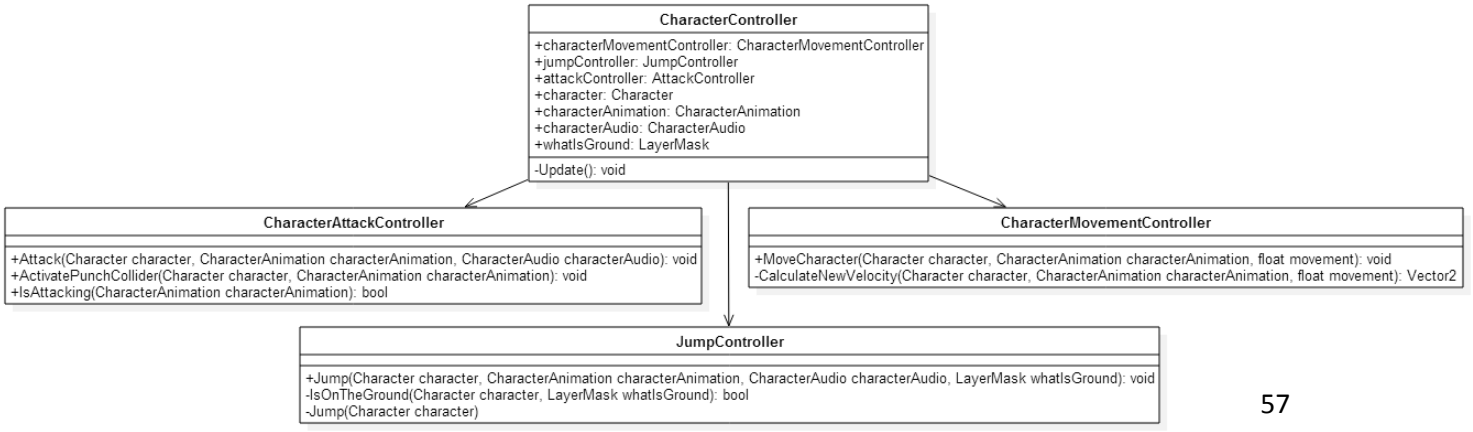


Estos serán los modelos de personaje que serán accedidos y modificados por los controladores. La primera clase es Character, que contiene toda la lógica relacionada con el personaje: velocidad máxima, fuerza de salto, dirección a la que está mirando, acceso al escudo que determina cuántas oportunidades tiene, el groundCheck (objeto que determina si el personaje está en el suelo o en el aire), el objeto que representa el ataque del personaje y el detector de colisiones. Por otro lado, el CharacterPunch es el objeto que representa el ataque del personaje. Debe ir separado del Character ya que tiene lógica propia, colisiones independientes y efectos de partículas.

Programamos las clases de vista para que activen animaciones y sonidos correspondientes.



Para modificar estas variables, debemos crear las clases que controlen al personaje.



Veamos con detalle cómo funciona la clase CharacterController:

```
void Update(){
        float movement = Input.GetAxis ("Horizontal");
        characterMovementController.MoveCharacter (character,
characterAnimation, movement);
        if(Input.GetButtonDown("Jump"))
            jumpController.Jump(character, characterAnimation,
characterAudio, whatIsGround);
        if(Input.GetButtonDown ("Attack")){
            attackController.Attack(character,characterAnimation,
characterAudio);
        }
        attackController.ActivatePunchCollider (character,
characterAnimation);
    }
```

Esta clase CharacterController es el controlador padre que se encarga de recoger todas las entradas que realice nuestro usuario desde el teclado, ratón o mando. Dependiendo de la entrada llamará a uno de los tres controladores hijo.

A continuación vemos un ejemplo de uno de los controladores hijo, el CharacterMovementController.

```
public class CharacterMovementController : MonoBehaviour {
    public void MoveCharacter(Character character, CharacterAnimation
characterAnimation, float movement){
        character.GetRigidBody ().velocity = new Vector2 (movement *
character.GetMaxSpeed(), character.GetRigidBody().velocity.y);
        characterAnimation.setMovementAnimation (character,movement);

    }
}
```

En este caso, el CharacterMovementController lleva a cabo el movimiento del personaje cuando el usuario ha pulsado los botones correspondientes a derecha o izquierda. Actualizamos la velocidad del Rigidbody[2] del personaje en función de la velocidad máxima del personaje y de cuánto haya pulsado el usuario (en los joysticks, mientras menos movimiento de éste se detecte, menos se moverá el personaje).

Además de los datos del modelo, actualizamos la animación del personaje para que se corresponda con los cambios ocurridos. Se hace una llamada al método SetMovementAnimation de la clase CharacterAnimation.

```
public void setMovementAnimation(Character character, float movement){
        characterMovementAnimation.setMovementAnimation (animator,
movement, isAttacking());
        characterFlipAnimation.flip (character, movement);
    }
```

En este método, se llama a dos métodos diferentes:

[2] El Rigidbody es el componente de un objeto de Unity que permite controlar dicho objeto a través de simulaciones física

- characterMovementAnimation.setMovementAnimation: Este método determina si el personaje está quieto (se activa la animación de reposo) o en movimiento (se activa la animación de correr).
- characterFlipAnimation.flip: Este método da la vuelta al modelo 3D de nuestro personaje, dependiendo de si esté mirando a izquierda o derecha.

Una vez realizada toda la programación de nuestro personaje, tenemos que añadirlo a nuestro juego. Asociamos todos los scripts al modelo 3D.

DELEGADOS

En ambos modelos del personaje, hemos tenido que hacer uso de delegados y eventos. El objetivo de utilizar eventos y delegados es evitar las referencias circulares en nuestra arquitectura, ya que no es recomendable que las clases de modelo llamen a métodos de las clases de controlador.

Los delegados son objetos que pueden llamar a métodos. Es el equivalente a un puntero a función de C o C++.

```
public delegate void PlayerChangesFacing();
public static event PlayerChangesFacing StopCamera;
```

En este segmento de código de Character, declaramos un delegado llamado PlayerChangesFacing. Este delegado será compatible con cualquier método que no devuelva nada y no tenga parámetros. A continuación, debemos declarar el evento de tipo PlayerChangesFacing, denominado StopCamera. De esta manera, podemos lanzar el evento StopCamera.

```
public void changeFacing(){
        facingRight = !facingRight;
        if (StopCamera != null)
            StopCamera();
    }
```

Cuando se hace una llamada a ChangeFacing, se lanzará el evento StopCamera. las clases que se suscriban a este evento tendrán el siguiente código:

```
void OnEnable(){
        Character.StopCamera += StopCamera;
    }
void StopCamera(){
        flipDistance = 0;
    }
```

Con el operador "+=" nos suscribimos al evento StopCamera de Character. Cuando este evento se lance, esta clase ejecutará el método StopCamera, que en este caso pondrá la variable flipDistance a 0.

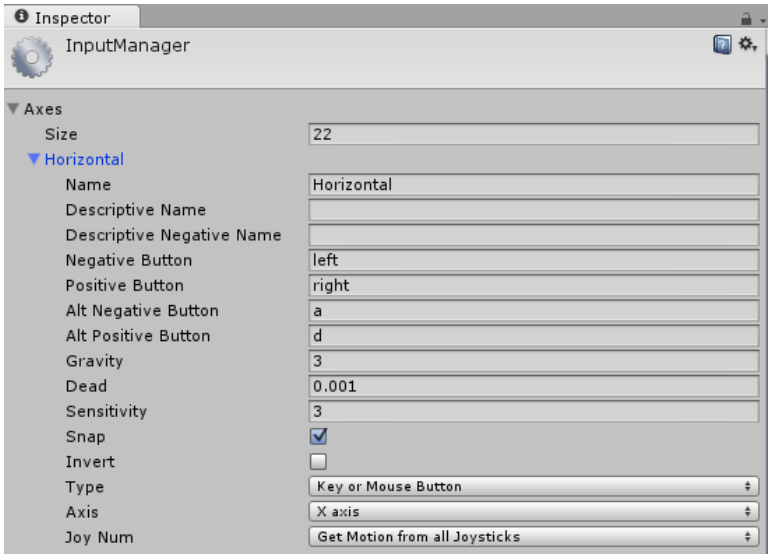
ENTRADAS DEL USUARIO

Para que la programación del personaje esté completa, necesitamos asociar a cada acción del personaje una entrada del usuario. En nuestro caso, se podrá jugar a este prototipo de dos maneras:

- Teclado + ratón

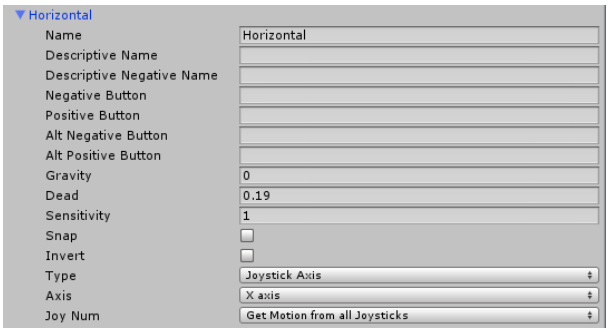
- Mando de XBOX 360

Tenemos que asociar a cada acción una tecla o un botón, lo cual lo haremos a través del InputManager de Unity, que nos permite definir una serie de botones o axis a cada acción que añadamos.



En este detalle de nuestro InputManager, hemos asociado el movimiento horizontal a los botones del teclado izquierda, derecha, A y D. Establecemos también los valores de sensibilidad, gravedad y un valor mínimo para que comience a registrar la entrada.

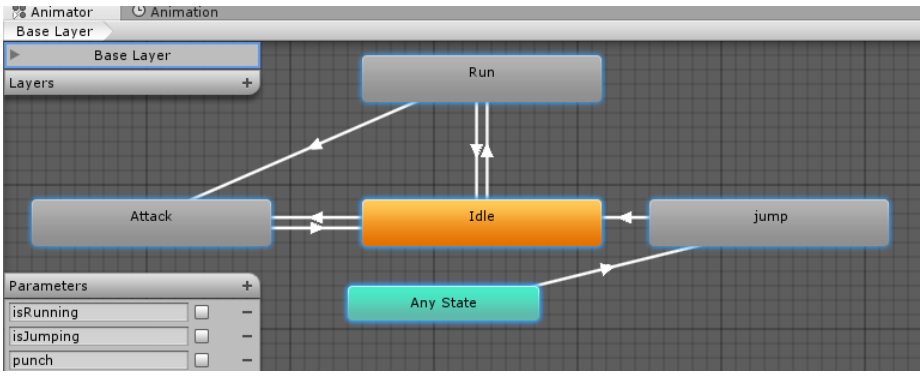
Para el caso homólogo del mando de XBOX 360, necesitamos conocer qué valores tiene cada botón accediendo al manual correspondiente. En este caso, asociamos el movimiento horizontal al axis X.



Realizamos esta operación para los tres movimientos básicos del personaje. En el manual de usuario se encuentra la configuración final.

ANIMACIONES

Para las animaciones del personaje trabajaremos con Mecanim, que explicamos en el apartado de prototipado. El modelo 3D que encontramos en nuestra fase de prototipado es el elegido para implementar el personaje. Accedemos a Mecanim para conocer en detalle cómo funciona la máquina de estados del personaje principal.



En esta máquina de estados, vemos que tenemos las siguientes animaciones: Idle (reposo), Attack (ataque), Jump (salto) y Run (correr). Cada una de ellas tiene las transiciones que se pueden dar de una a otra.

También tenemos tres parámetros: isRunning (booleano que determina si el personaje se está moviendo), isJumping (booleano que determina si el personaje está saltando) y punch (un disparador que se activa cuando el personaje debe atacar). Todas estas variables son las condiciones que se tienen que dar para que se produzcan los saltos en las transiciones, y las modificamos a través de código. Las clases a las que les hemos dado la responsabilidad de modificar estas variables y de activar los disparadores son las clases CharacterAnimation que vimos anteriormente. Un ejemplo es el código que implementa JumpAnimation.

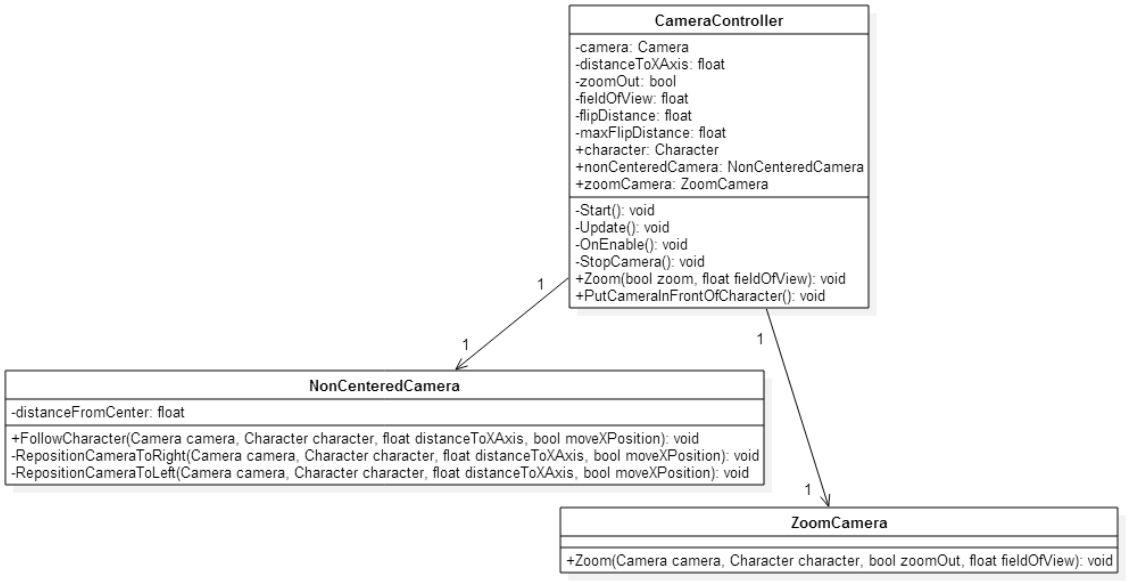
```
public class CharacterJumpAnimation : MonoBehaviour {

    public void setJumpAnimation(Animator animator){
        animator.SetTrigger ("isJumping");
    }
}
```

El GameObject Character tiene un componente Animator asociado, que es el encargado de almacenar esta máquina de estados y realizar las transiciones. A través del componente Animator, hacemos una llamada al método SetTrigger, que activa el disparador isJumping. De esta manera, se produce la transición desde cualquier estado al estado Jump.

7.2. Implementación de la cámara

La cámara no fue integrada dentro de ningún controlador, ya que es una entidad muy compleja que necesitamos separar del resto.

Para implementar la cámara, necesitamos un controlador que vaya actualizando los valores de posición de la cámara en función de la posición del personaje.



Vemos que existen dos tipos de cámaras: NonCenteredCamera y ZoomCamera. El controlador de la cámara (CameraController) se encarga de llamar a los dos métodos principales de cada tipo de cámara cada frame a través del método Update().

```
void Update () {
            nonCenteredCamera.FollowCharacter(camera, character,
distanceToXAxis, Mathf.Abs(flipDistance)>maxFlipDistance);
            zoomCamera.Zoom (camera, character, zoomOut, fieldOfView);
    }
```

La cámara descentrada funciona de la siguiente manera:

Determina hacia dónde se está posicionando el personaje: si está girado hacia la derecha, el personaje se posicionará por la izquierda, y viceversa.
Determina si el personaje se acaba de girar: si se acaba de girar, la cámara no se moverá en el eje X. Si lleva una distancia recorrida después del giro, la cámara deja al personaje a izquierda o derecha dependiendo de hacia dónde mire.
Mueve la cámara en el eje X en función de la velocidad del personaje.
Aunque el personaje no se mueva en horizontal, la cámara sigue al personaje en el eje Y.

La clase NonCenteredCamera es la que se encarga de llevar a cabo todos estos cálculos. A continuación vemos un detalle del código:

```
public void FollowCharacter(Camera camera, Character character, float
distanceToXAxis, bool moveXPosition){
        if (!character.isFacingRight ()) {
            RepositionCameraToLeft (camera, character,
distanceToXAxis, moveXPosition);
            if (Mathf.Abs (camera.transform.position.y -
character.transform.position.y) > 3) {
                camera.transform.position = new Vector3
(camera.transform.position.x, Mathf.Lerp
(camera.transform.position.y,character.transform.position.y +
distanceToXAxis,
Time.deltaTime*character.getYVelocity()),
camera.transform.position.z);
```

```
            }
        } else if (character.isFacingRight ()) {
            RepositionCameraToRight (camera, character,
distanceToXAxis, moveXPosition);
            if (Mathf.Abs (camera.transform.position.y -
character.transform.position.y) > 3) {
                camera.transform.position = new Vector3
(camera.transform.position.x, Mathf.Lerp (camera.transform.position.y,
character.transform.position.y + distanceToXAxis,

Time.deltaTime*character.getYVelocity()),
camera.transform.position.z);
            }
        }
    }

    void RepositionCameraToRight(Camera camera, Character character,
float distanceToXAxis, bool moveXPosition){
        float xPosition;
        if(!moveXPosition)
            xPosition = character.transform.position.x+15;
        else
            xPosition = camera.transform.position.x;
        camera.transform.position = Vector3.Lerp
(camera.transform.position, new Vector3(xPosition,
character.transform.position.y+distanceToXAxis,
camera.transform.position.z),
Time.deltaTime*(character.getXVelocity()/5));
        if(character.transform.position.x -
camera.transform.position.x <=-distanceFromCenter){
            camera.transform.position = new
Vector3(character.transform.position.x+distanceFromCenter, Mathf.Lerp
(camera.transform.position.y,
character.transform.position.y+distanceToXAxis,

Time.deltaTime), camera.transform.position.z);
        }
    }
```

En este código vemos como funciona en detalle los cambios de cámara. En el primer método, FollowCharacter, preguntamos hacia qué lado está mirando el personaje. A continuación, reposicionamos la cámara en función de su orientación llamando a los métodos RepositionCameraToRight o RepositionCameraToLeft. Además, preguntamos si el personaje se ha movido en el eje Y, para actualizar la posición de la cámara con respecto a éste.

En esta porción de código también vemos cómo funciona RepositionCameraToRight. Comprobamos si el personaje acaba de girarse sobre sí mismo. De esta manera, sabremos si la cámara debe quedarse fijo en caso de que acabe de girar en el eje X o si la cámara debe recolocarse. A continuación, actualizamos los valores de posición de la cámara con "camera.transform.position" y haciendo uso de las funciones Lerp, que permiten una transición suave de posición. Además, si la cámara ya se ha colocado perfectamente para dejar al personaje a la izquierda, no realizamos una transición suave, sino que actualizamos automáticamente la posición de la cámara en el eje X con respecto a la del personaje, ya que de otra manera el efecto de movimiento se ve inadecuado.

La clase ZoomCamera determina si nos encontramos en una zona en la que la cámara deba hacer un zoom out, y alejarse del personaje.

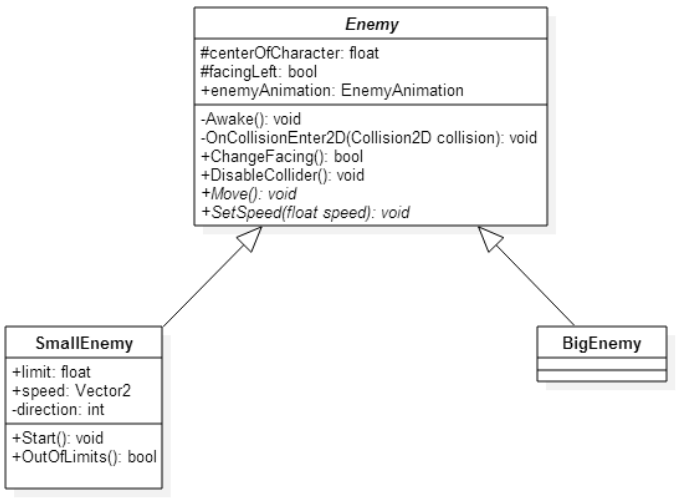
```
public void Zoom(Camera camera, Character character, bool zoomOut,
float fieldOfView){
        if (zoomOut) {
            Physics2D.gravity = new Vector2(Physics2D.gravity.x, -5f);
            character.setJumpForce(50);
            camera.fieldOfView = Mathf.Lerp (camera.fieldOfView,
90+fieldOfView, Time.deltaTime);
        }else{
            Physics2D.gravity = new Vector2(Physics2D.gravity.x, -
30f);
            character.setJumpForce(1400);
            camera.fieldOfView = Mathf.Lerp (camera.fieldOfView, 90,
Time.deltaTime);
        }
    }
```

En las zonas donde se haga un zoom out, que corresponden a las zonas donde el jugador tenga que realizar una caída libre, cambiamos la gravedad y la fuerza de salto del personaje, para hacer que caiga más lentamente y no salte de manera exagerada. Además, modificamos el fieldOfView (el campo de vista) de la cámara, de manera que se aleje y se aprecien más elementos. Si salimos de la zona de zoom, los valores vuelven a sus valores predeterminados.

7.3. Implementación de enemigos

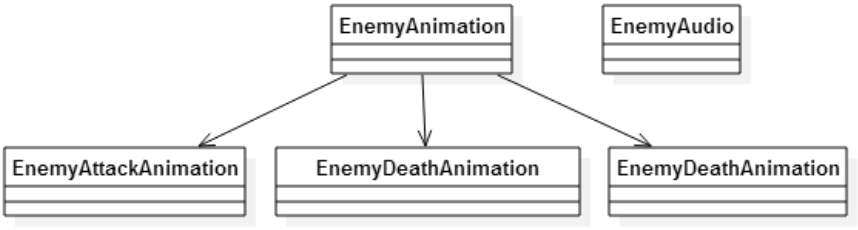
Para la implementación de enemigos necesitamos lo mismo que para el personaje principal: clases de modelo, vista y controlador.

Las clases de modelo se implementan de la siguiente manera:

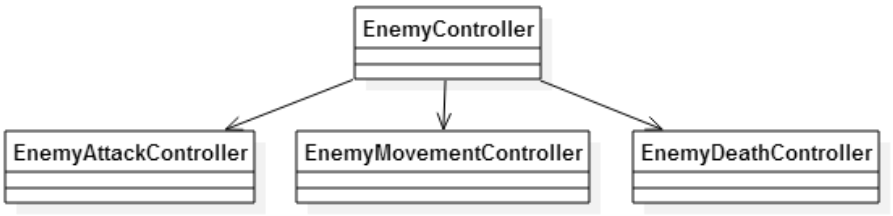


Definiremos una clase abstracta denominada Enemy, que tenga dos métodos abstractos que implementarán el resto de clases que deriven de ella: Move y SetSpeed. Tendremos dos tipos de enemigos: enemigos pequeños y enemigos grandes. La principal diferencia radica en que los enemigos pequeños se mueven, y los grandes no.

Las clases de vista se encargarán de las animaciones de los enemigos.



Las clases de controlador se encargan de llamar al modelo y vista correspondientes en cada caso.



Entremos en los detalles de la clase EnemyController:

```
void Update () {
        enemyMovementController.MoveEnemies(enemies);
        if (enemiesToBeDeleted.Count != 0) {
            DeleteEnemies();
        }
    }
```

En el Update de EnemyController, se hace una llamada por frame al método MoveEnemies de la clase EnemyMovementController, pasando por parámetro la lista de enemigos previamente establecida.

```
public class EnemyMovementController : MonoBehaviour {

    public void MoveEnemies(List<Enemy> enemies){
        foreach (Enemy enemy in enemies) {
            if(IsSmallEnemy(enemy)){

enemy.GetEnemyAnimation().SetMovementAnimation((SmallEnemy)enemy);
                enemy.Move();
            }
        }
    }

}
```

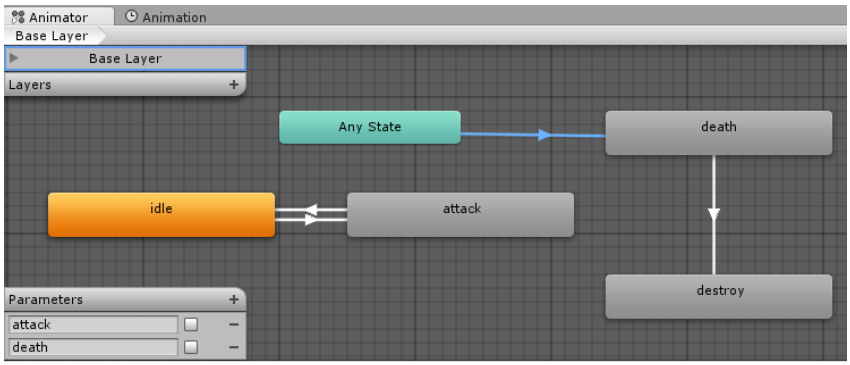
En esta clase, el método MoveEnemies itera por todos los enemigos disponibles en el mapa y si es un enemigo pequeño, modifica su posición haciendo una llamada al método Move, y establece la animación a través de GetEnemyAnimation().SetMovementAnimation. De manera homóloga, los otros controladores realizan llamadas a las clases de modelo y vista en función de las acciones que se quieran llevar a cabo.

DELEGADOS

Los siguientes eventos son recogidos por el EnemyController:

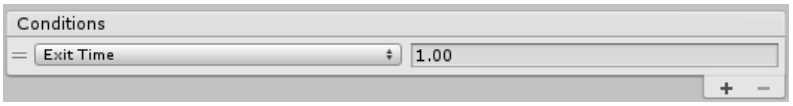
- EnemyAttacks, activado por un enemigo, que se dispara cuando el personaje, sin haber atacado antes, colisiona con el enemigo. El resultado es restar una vida al jugador.
- EnemyDies, activado por el objeto de puñetazo del jugador, que se dispara cuando el puñetazo colisiona con un enemigo. El resultado es preparar al enemigo para que realice las animaciones correspondientes y eliminarlo del escenario.

ANIMACIONES



La máquina de estados de las animaciones de un enemigo comprende cuatro estados: idle (reposo), attack (ataque), death(muerte) y destroy(destruir). En la imagen podemos ver las transiciones que se pueden dar entre ellas. También vemos los diferentes parámetros: attack (disparador que se activa si el enemigo ataca al jugador) y death (disparador que se activa si el jugador ataca al enemigo).

La diferencia con respecto a otros estados que habíamos visto en el personaje son los estados death y destroy. Se transiciona al estado death si el jugador ataca al enemigo, y de ahí se transiciona al estado destroy cuando termina la animación de muerte, lo cual especificamos estableciendo la siguiente condición:



La diferenciación entre estos dos estados se ha realizado por lo siguiente: cuando el enemigo muere, primero tiene que terminar de ejecutar la animación de muerte, y luego el GameObject que corresponde al enemigo muerto debe eliminarse del escenario. Sabemos que está listo para eliminarse si ha terminado de ejecutar dicha animación, por lo que solo se eliminarán enemigos que se encuentren en el estado destroy.

7.4. Implementación de cambio de mundo, entorno y obstáculos

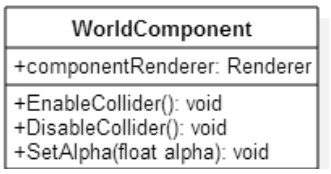
La siguiente fase de producción se corresponde a la mecánica principal de nuestro juego: el cambio de mundo. Además, en este punto del desarrollo programaremos todos los componentes del escenario que se vean afectados por el cambio de mundo, así como los obstáculos que nos podemos encontrar en la escena.

Lo primero que tenemos que tener en cuenta es el alfa de los objetos. El alfa es el componente que define la transparencia de un objeto.

- Cuando el alfa de un objeto es 1, el objeto es completamente opaco.

- Cuando el alfa de un objeto es 0, el objeto es completamente transparente.

Primero veremos un detalle de la clase de modelo principal, de la que derivarán los diferentes obstáculos, WorldComponent.

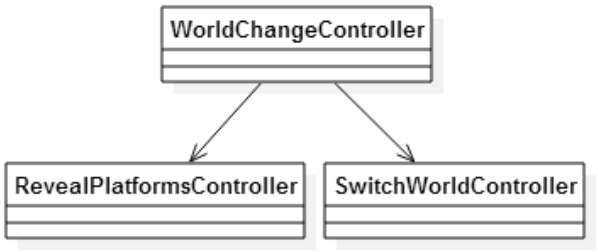


Cualquier GameObject que tenga como componente este script será un elemento del escenario que se vea afectado por el cambio de mundo, es decir, que estará activo en uno de los dos mundos únicamente. Para que se vea afectado por el cambio de mundo, debemos activar o desactivar sus colliders dependiendo si están en el mundo activo o en el contrario. Además, el método SetAlpha cambia la transparencia de los objetos, haciendo que sean parcialmente visibles durante el cambio de mundo.

Las clases de vista son las siguientes:

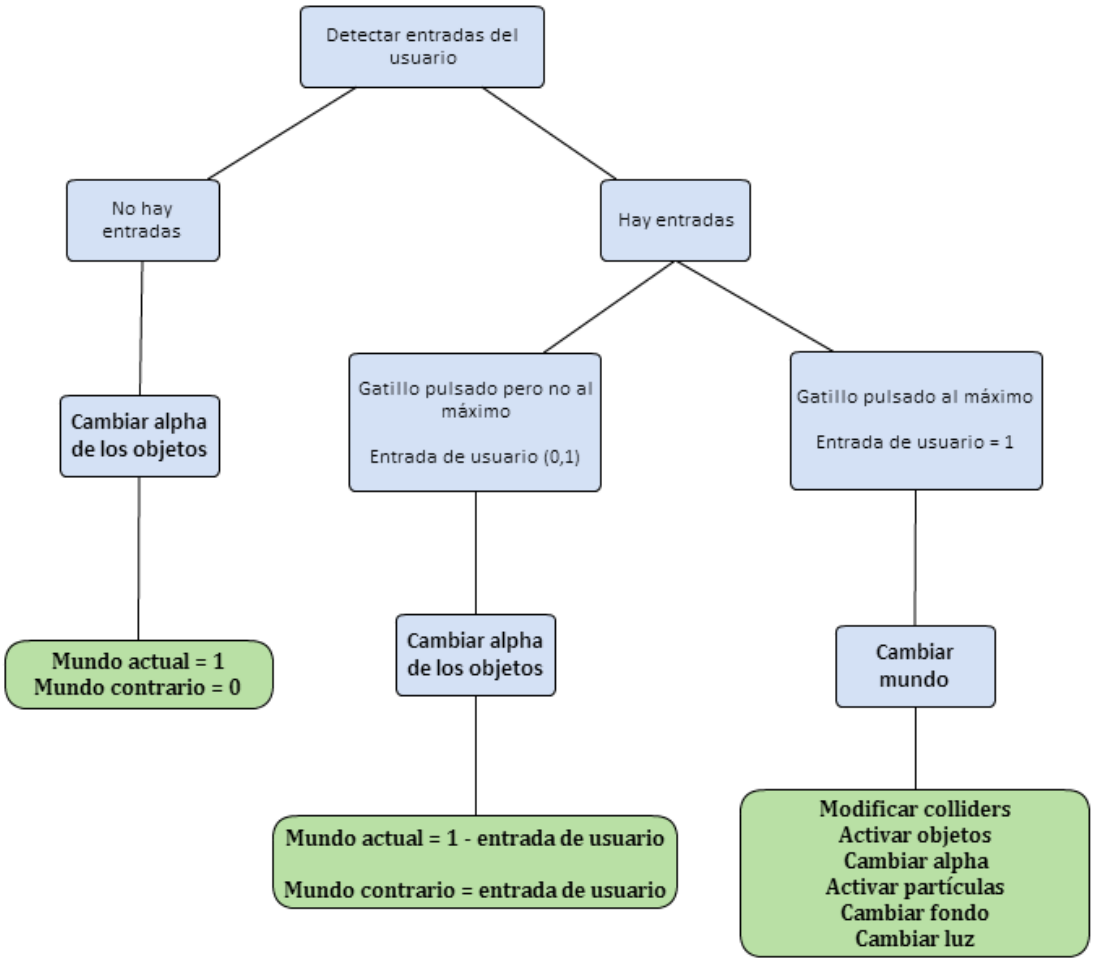
El Background es la clase que se encarga de cambiar el fondo dependiendo del mundo donde nos encontremos. ChangeWorldParticle se encarga de activar las partículas de cambio de mundo.

Explicaremos cómo funcionan los controladores de cambio de mundo:



El controlador padre es WorldChangeController, que se encarga de determinar si el usuario activa el mundo contrario o si simplemente está echando un vistazo a los cambios que se pueden producir. El controlador hijo RevealPlatformsController cambia el valor alfa de los elementos para que se vuelvan semitransparentes, pero no activa ninguno de los elementos del mundo contrario. En cambio, SwitchWorldController realiza todos los cambios necesarios para pasar de un mundo a otro.

WorldChangeController tiene dos parámetros importantes: actualWorldObjects y secondWorldObjects. Ambos son una lista de objetos de tipo WorldComponent. Los dos controladores hijos se encargan de recorrer estas listas y realizar los cambios necesarios. A continuación mostramos un flujo de cambio de mundo dependiendo de las entradas que aporte el jugador en los periféricos de control:

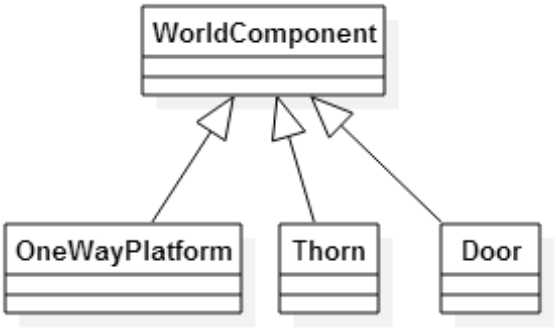


Si no existen entradas, actualizamos el alfa de los objetos (mundo actual completamente opaco, mundo contrario completamente transparente). En caso de que se detecte entrada de usuario, si este valor se encuentra entre 0 y 1, ninguno de ellos incluido, cambiamos el alfa de los objetos: el mundo actual comienza a hacerse más transparente y el mundo contrario comienza a hacerse más opaco. Para estos dos casos, el controlador encargado es el RevealPlatformsController.

Si realizamos el cambio del mundo, es decir, la entrada de usuario es igual a 1, el controlador padre se encargará de llamar al controlador SwitchWorld, que llevará a cabo todas las tareas de cambio de mundo descritas en el flujo de WorldChangeController.

ELEMENTOS QUE DERIVAN DE WORLDCOMPONENT

A continuación explicamos todos los elementos que están afectados por el cambio de mundo:



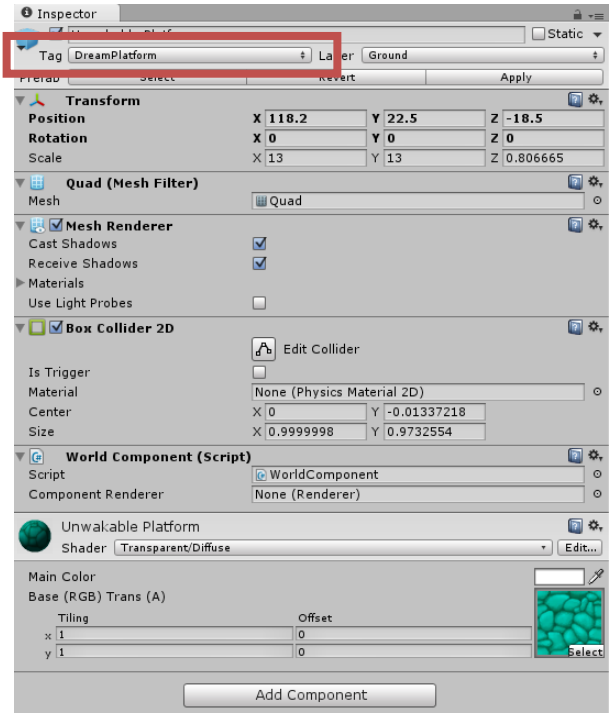
- OneWayPlatform: Es el componente que irá asociado a las plataformas con forma de nubes, las cuales son plataformas tangibles pero que pueden ser atravesadas por el personaje si éste salta desde abajo.
- Thorn: Es el componente que va asociado a las espinas, elementos del escenario que quitan un punto de vida al jugador si éste llega a tocarlas.
- Door: Es el componente que va asociado a la puerta del final del nivel, que indica que el jugador ha conseguido acabar con éxito el escenario y que solo estará disponible en el mundo de los sueños.

ELEMENTOS QUE NO ESTÁN ASOCIADOS A WORLDCOMPONENT

- Fence: Puerta en forma de verja que se encuentra abierta en el mundo real y cerrada en el mundo de los sueños
- PunchPowerUp: Poder que aumenta el ataque del personaje, permitiéndole atacar a los enemigos grandes.

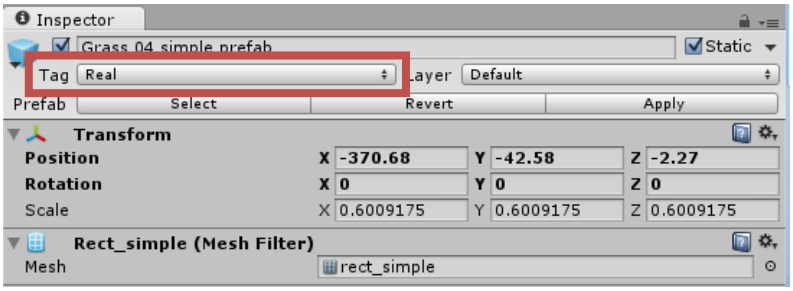
ESTABLECER MUNDO AL QUE PERTENECE CADA WORLDCOMPONENT

Para cada WorldComponent, establecemos una etiqueta que indique en qué mundo se encuentra. Para el mundo real elegimos "RealPlatform" y para el mundo de los sueños seleccionamos "DreamPlatform".



DECORACIÓN DEL MUNDO

Existen elementos decorativos, como árboles, plantas y hierba, que no son elementos con los que el jugador pueda interaccionar, sino que aportan vida al escenario. Es necesario tener en cuenta estos elementos, ya que cuando cambiamos de mundo deben activarse o desactivarse según corresponda.



De manera similar a los WorldComponent, diferenciamos en qué mundo se encuentran por las etiquetas "Real" o "Dream". Los controladores se encargan de buscar todos los elementos según su etiqueta y de activarlos o desactivarlos.

PERSISTENCIA

La primera vez que el usuario comience a jugar, no estarán disponibles todos los niveles. Los niveles se tendrán que desbloquear a medida que el jugador vaya terminando los anteriores. Utilizaremos los datos persistentes para conseguir que los datos de desbloqueo de niveles se almacenen.

PlayerPrefs es una clase que permite guardar información recuperable aún cuando se cierre el juego. Veremos cómo programar el almacenamiento de datos persistentes:

```
using UnityEngine;
using System.Collections;

public class NavigationController : MonoBehaviour {
    int currentLevel;

    void Start(){
        currentLevel = int.Parse(Application.loadedLevelName);
    }

    void OnEnable(){
        Door.finishLevel += FinishLevel;
    }

    void FinishLevel(){
        currentLevel++;
        PlayerPrefs.SetInt("Level"+ currentLevel, 1);
        Application.LoadLevel (currentLevel);
    }
}
```

La clase NavigationController es la encargada de cargar el siguiente nivel. Antes de cargar el nivel, guardamos en una variable que corresponde al nivel que vamos a cargar el valor 1, como si de un booleano se tratara. Así desbloquearemos el nivel desde el menú principal.

```
    public void LoadLevel(int level){
        if(level!=1){
            if(PlayerPrefs.GetInt("Level"+level) == 1)
                Application.LoadLevel (level.ToString());
        }else
            Application.LoadLevel (level.ToString());


    }
```

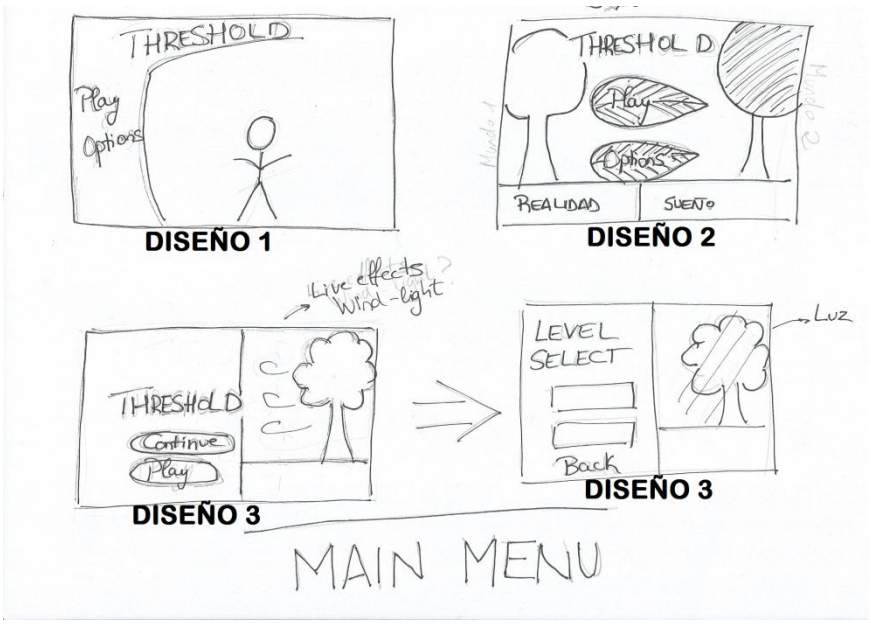
En el MenuView, la clase que se encarga de lanzar eventos según el botón de la interfaz seleccionado, al seleccionar uno de los tres niveles, primero preguntará si la variable correspondiente al nivel se encuentra a 1. Si es así, significa que el jugador habrá desbloqueado el nivel y podrá jugarlo.

7.5 Interfaz

La principal aportación de este TFG consistía en el desarrollo de este prototipo en la versión de Unity 4.6, que incluía un nuevo sistema de interfaces.

7.5.1. Diseño de interfaz

En primer lugar comenzamos con el diseño del menú principal. Planteamos diversos diseños que se muestran a continuación:

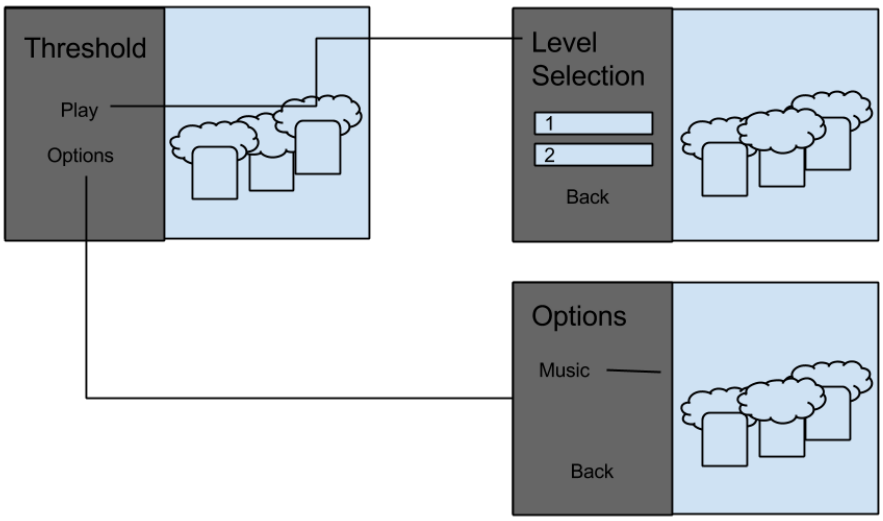


Nos decantamos por el tercer diseño por varios motivos:

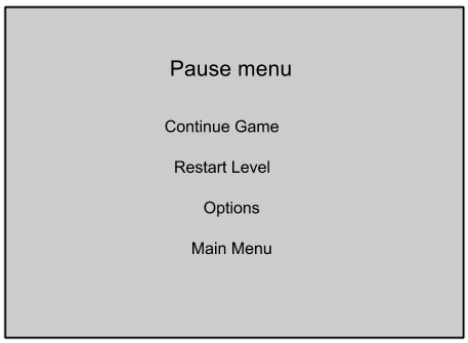
- Los dos primeros requerían de elementos visuales muy específicos.
- El tercer diseño podía incluir efectos de partículas, de manera que el menú estuviera en movimiento y llamara la atención del usuario desde el primer momento.

Realizamos el diseño con mayor detalle de nuestra interfaz. Tendremos tres ventanas en el menú principal:

- La pantalla principal, donde elegiremos si empezar a jugar o modificar las opciones.
- La selección de nivel, donde elegimos el nivel que queramos jugar.
- Las opciones, donde modificaremos ciertos parámetros del juego.



Para el menú de pausa, llevaremos a cabo un diseño mucho más simple para no distraer al jugador.



7.5.2. Implementación de interfaz

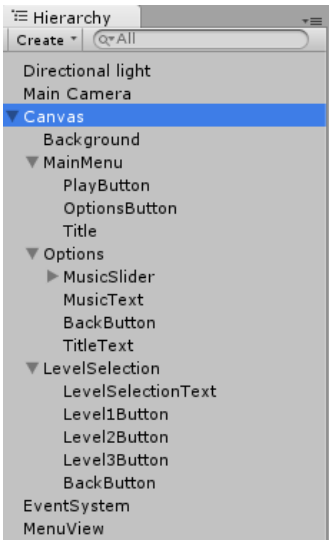
Implementación del menú principal

Antes de comenzar con la interfaz construimos nuestro escenario en movimiento con los modelos 3D y sistemas de partículas escogidos para decorar la escena.

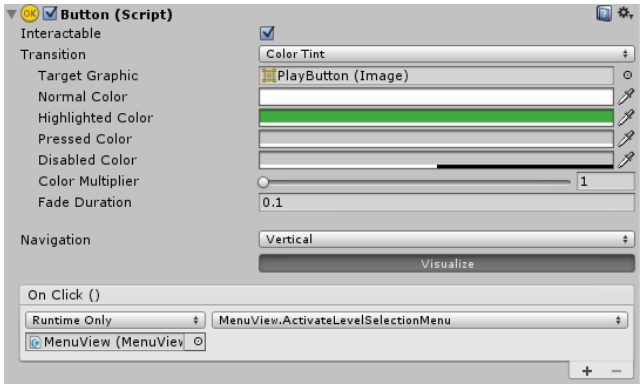
En Unity 4.6, las interfaces se desarrollan utilizando dos elementos nuevos:

- Canvas: Área que contendrá todos los elementos de la interfaz de usuario.
- EventSystem: Maneja las entradas de usuario y envía eventos. Es necesario para que el jugador pueda navegar por la interfaz.

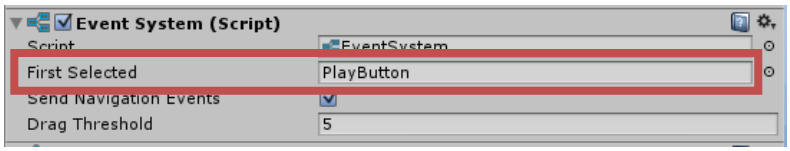
Tenemos que añadir un Canvas como un nuevo objeto de la escena, y automáticamente se añadirá un nuevo EventSystem asociado a este Canvas. Añadimos un fondo sobre nuestro escenario, de manera que la parte de la izquierda se oscurezca para poder apreciar mejor los elementos de la interfaz. También añadimos tres paneles, uno para vista principal, otro para opciones y un último para selección de nivel. Cada uno de ellos contendrá los elementos de su vista. Por último añadimos todos los botones, textos y sliders necesarios.



Para que el usuario pueda navegar con el mando de Xbox, necesitamos modificar la navegación de los botones. En el apartado Navigation de nuestro componente Button, tenemos que seleccionar el tipo de navegación que queremos. En nuestro caso, como los botones están en una columna, tenemos que seleccionar una navegación vertical.



Cuando tenemos todos los botones, tenemos que actualizar nuestro Event System y seleccionar el botón Play Game como First Selected. Esto hace que el Event System considere este botón como el que está seleccionando el jugador cuando carga el menú principal. Esto es necesario para que los usuarios que utilicen el mando puedan comenzar a navegar por dicho menú desde que carga.



Con esto terminamos de añadir y modificar nuestros elementos de la interfaz, y comenzamos la parte programática. Necesitamos tener una clase MenuView, que permita movernos entre las diferentes vistas del menú y ejecutar acciones seleccionadas. Esta clase se encargará de mostrar solamente los elementos de la vista en la que nos encontramos, y ocultará los elementos restantes. Veamos un ejemplo de cómo funciona esta navegación.

```
public void ActivateMainMenu(){
        mainMenu.SetActive (true);
        optionsMenu.SetActive(false);
        levelSelectionMenu.SetActive (false);
        eventSystem.SetSelectedGameObject(mainMenu.transform.Find
("PlayButton").gameObject);
    }
```

El método ActivateMainMenu permite activar el panel de vista principal, a la vez que desactiva las otras dos vistas. Además, cada vez que activamos una vista, tenemos que elegir un nuevo objeto seleccionado por el Event System, para que el navegador tenga un botón inicial sobre el que empezar a navegar en dicha vista.

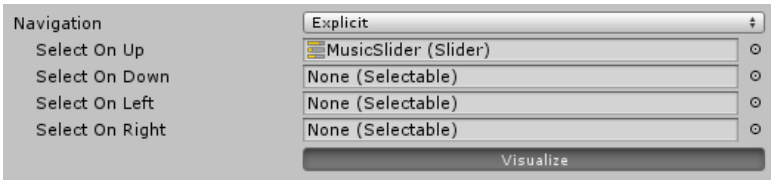
Para cada botón, seleccionamos qué método de nuestro script debe ejecutar seleccionando en el inspector de Unity un evento que se produzca al clicar (OnClick).

El resultado final es el que vemos a continuación.

Implementación del menú de pausa

El menú de pausa se implementa de manera similar al menú principal. Añadimos un Canvas como un nuevo objeto de la escena, que vendrá acompañado por el Event System. Añadimos un filtro negro que ocupe toda la pantalla, para que el jugador centre su atención en los elementos del menú. Creamos dos paneles, uno para la vista principal y otro para las opciones. Agregamos todos los botones, textos y sliders necesarios.

Volvemos a modificar la navegación del menú. Esta vez probamos con el modo explícito, en el cual no es automático, sino que debes seleccionar a cuál de los otros elementos de la interfaz transiciones cuando pulsas arriba, abajo, izquierda o derecha.



Pasamos a la parte de código. Creamos una nueva clase denominada PauseView, que tendrá una función similar al MenuView, nos permite movernos entre las vistas y ejecutar acciones seleccionadas.

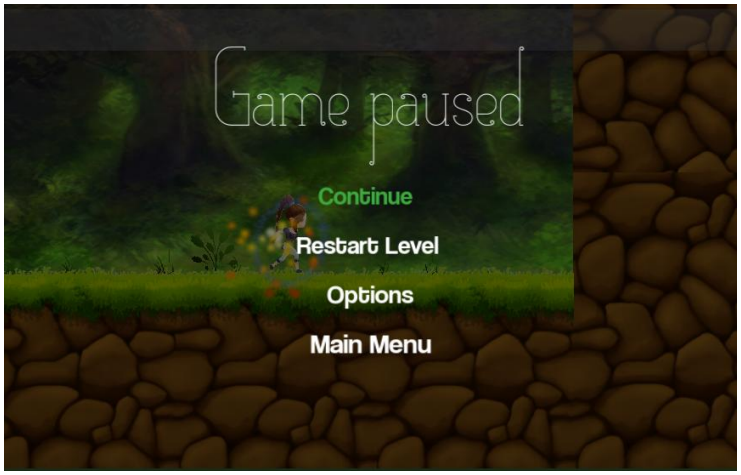
```
public void PauseGame(){
        gamePaused = !gamePaused;
        pauseMenu.SetActive (gamePaused);
        if(gamePaused){
            Time.timeScale = 0;
            mainPausePanel.SetActive(true);
            optionsPanel.SetActive(false);
            eventSystem.SetSelectedGameObject(null);
            eventSystem.SetSelectedGameObject(GameObject.Find
("ContinueButton").gameObject);
        }
        else
            Time.timeScale = 1;
    }
```

En esta porción de código especificamos como se hace la pause. El LevelController detecta que se ha pulsado el botón de pausa y se llama al método PauseGame de la clase PauseView. Se

cambia el estado de pausa, y se activa o desactiva el menú de pausa correspondiendo a este estado. Si el juego se pausa, ocurren los siguientes cambios:

- Se actualiza el timeScale a 0, de manera que ningún elemento de la pantalla se moverá.
- Se activa la vista principal del menú de pausa, y se oculta la vista de opciones.
- Se resetea el objeto seleccionado por el Event System, poniendo el foco sobre el botón Continue.

De nuevo, para cada botón se establece la acción que llevará a cabo a través del editor con la opción On Click(). Una vez terminado todos estos cambios, el menú de pausa estará completo.



Implementación del timer

Para añadir un componente de reto a nuestro juego, añadimos un pequeño timer en la parte superior de la pantalla. Este timer será un incentivo para el jugador, que podrá rejugar el nivel para mejorar su marca.

Al Canvas que habíamos añadido a nuestros niveles para el menú de pausa añadimos un nuevo panel, que contendrá el timer y lo mostrará de manera continua. Para contabilizar el tiempo que tarda el jugador en jugar la fase, modificamos el Update del LevelController.

```
void Update(){
        if(Input.GetButtonDown("Pause")){
            pauseView.PauseGame();
        }else{
            timer += Time.deltaTime;
            string minutes = Mathf.Floor(timer / 60).ToString("00");
            string seconds = (timer % 60).ToString ("00");
            levelView.UpdateTimer (minutes, seconds);
        }
    }
```

Tenemos una variable timer a la que sumamos cuánto tiempo ha pasado desde el último frame en cada frame. A partir de este timer extraemos los minutos y segundos, y actualizamos el contador visual de la pantalla.

Implementación de barra de carga de mundo

Como último elemento de nuestro HUD añadiremos una barra de carga. Esta barra de carga aparecerá cuando empecemos a cambiar de mundo de manera lenta, y se irá llenando conforme los elementos del otro mundo se vean más. De esta manera, el usuario sabrá cuánto falta para que el cambio de mundo se produzca.

Como método de aprendizaje del desarrollo de interfaces antes de la versión 4.6 de Unity, no utilizaremos el Canvas en este caso. Programaremos esta barra de carga haciendo uso de la función OnGUI, función que se llama para renderizar y manejar los eventos de interfaz.

Añadiremos la función OnGUI a nuestro LevelView, que será el encargado de todos los elementos de la HUD.

```
void OnGUI () {
        if (Input.GetAxis("Trigger1") != 0.00f){
            GUI.color = new Color(GUI.color.r, GUI.color.g,
GUI.color.b, Input.GetAxis("Trigger1")*2);
            GUI.DrawTexture(new Rect(frameMarginLeft, frameMarginTop,
frameWidth, frameHeight), realTexture);
            GUI.BeginGroup(new Rect (frameMarginLeft, frameMarginTop,
frameWidth * Mathf.Clamp01(Input.GetAxis("Trigger1")), frameHeight));
            GUI.DrawTexture(new Rect(0, 0, frameWidth, frameHeight),
dreamTexture);
            GUI.EndGroup();
        }
    }
```

En primer lugar, especificamos que solo dibujaremos este elemento si se está realizando el cambio de mundo por parte del usuario. Dibujamos la barra de carga más opaca conforme el cambio de mundo vaya avanzando. A continuación dibujamos la barra de carga vacía, seleccionando su altura y anchura (frameWidth, frameHeight) y la posición donde queremos que se dibuje (fframeMarginLeft, frameMarginTop).

Para dibujar la barra de carga llena establecemos un nuevo grupo, de manera que el sistema de coordenadas se transforma y sitúa el punto (0,0) en la esquina superior izquierda del grupo. Esto nos servirá para dar el efecto de "rellenar" la barra de carga, estableciendo la anchura de la barra de carga llena en función de cuánto queda para completar el cambio de mundo (frameWidth * Mathf.Clamp01(Input.GetAxis("Trigger1")).

El resultado final es el siguiente:

Este sería el resultado de pulsar el gatillo del mando de Xbox a la mitad o de haber completado el 50% del cambio de mundo con los botones del ratón.

7.6. Implementación de niveles

Para implementar los niveles, añadiremos objetos en la jerarquía de Unity, construyendo los niveles en base al diseño que planteamos.

UI: Objetos de la interfaz de usuario - EventSystem, canvas, botones, texto, etc.
Controllers: Todos los controladores necesarios para el nivel - NavigationController, LevelController, CharacterController, WorldChangeController, EnemyController.
Cameras: Los dos tipos de cámaras de los que disponemos - Cámara principal y cámara de fondo.
Character: Personaje jugable
Light: Luz direccional que abarca toda la escena
WorldComponents: Todos los componentes que forman un nivel: plataformas, decoración, puertas, espinas, enemigos, etc.
LevelAudio: Música del nivel.

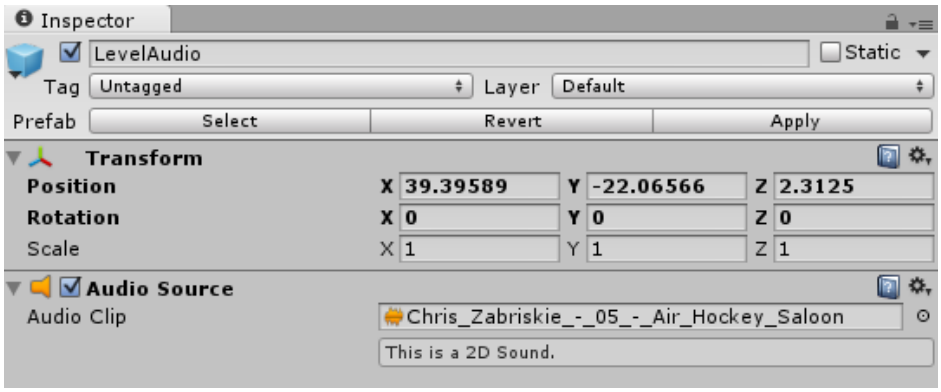
7.7. Implementación del sonido

El sonido forma parte de la vista de la arquitectura, ya que es un elemento que modifica lo que el usuario aprecia. Tenemos tres tipos de sonidos:

MÚSICA DE NIVEL

Asociamos la música del nivel a un nuevo objeto de la jerarquía de Unity. A este objeto añadimos un nuevo componente: una fuente de sonido. Asociamos a esta fuente de sonido la música escogida, y escribimos un script que suba el volumen cuando el jugador se encuentre en el mundo de los sueños y lo vuelva a bajar cuando regresa al mundo real.

```
public class LevelAudio : MonoBehaviour {
    int audioVolume;

    void Start () {
        audioVolume = 1;
    }

    public void ChangeAudio(){
        audio.volume = audio.volume + audioVolume * 0.5f;
        audioVolume = audioVolume * -1;
    }
}
```



Llamaremos al método ChangeAudio desde el controlador que se encarga del cambio de mundo, SwitchWorldController.

SONIDO DE PERSONAJE

Para el sonido del personaje, añadimos un nuevo objeto como hijo del objeto Character. Este hijo tendrá tres componentes de tipo AudioSource, cada uno de ellos será un sonido que se active cuando: el personaje reciba daño; el personaje salte; el personaje ataque. De manera similar a la música de nivel, necesitaremos asociar a este objeto un script que permita reproducir los sonidos correspondientes. A continuación vemos una porción del código de este script, denominado CharacterAudio.

```
public class CharacterAudio : MonoBehaviour {

    public void playDamageSound(){
        damage.Play ();
    }

    public void playExertSound(){
        exert.Play ();
    }

    public void playPunchSound(){
        punch.Play ();
    }
}
```

Los controladores que utilizarán estos métodos serán CharacterAttackController, JumpController y LevelController.

SONIDO DE ENEMIGOS

De manera similar al personaje principal, los enemigos tendrán dos tipos de sonidos: sonido de ataque y sonido de muerte. Con un objeto que emparentamos a cada enemigo, añadimos dos componentes AudioSource y un script que reproduzca dichos sonidos.

Este script será accedido por el EnemyAttackController y el EnemyDeathController.

8. Pruebas y mantenimiento

Las pruebas no han sido algo independiente del desarrollo. Cada vez que se implementaba una mecánica, se realizaban pruebas para saber que funcionaba correctamente. Además, cuando se pasaba a la implementación de un apartado nuevo, se llevaba a cabo las pruebas de integración, comprobando que ninguna de las mecánicas dejara de funcionar cuando se integraran.

El resultado final del prototipo se llevó a unos pocos usuarios que testearon los niveles diseñados y aportaron feedback del juego para mejorarlo. Incluyendo los bugs que pudieron encontrarse mientras se jugaba, la mayoría del feedback aportado estaba relacionado con la dificultad del nivel (zonas muy difíciles para el nivel primero, zonas muy fáciles para el nivel tercero), por lo que los niveles se fueron mejorando con toda esta información recibida.

En cuanto al mantenimiento, tras cada prueba que realizaban los usuarios, se llevaba a cabo un mantenimiento correctivo de los bugs encontrados durante las sesiones de juego. Muchos de los bugs estaban relacionados en su mayoría con los siguientes elementos:

- Zonas a las que no se podía llegar con las mecánicas planteadas
- Problemas con el salto del personaje
- Colisión del personaje con los enemigos

9. Trabajo futuro

En este apartado cubriremos todas las posibles mejoras que, por falta de tiempo, han quedado fuera del alcance de este TFG, pero que se plantearán como futuras aportaciones a este proyecto.

- Diseño de más niveles: Para nuestro prototipo, y siguiendo la metodología METHOD de Mark Cerny, el número recomendable de niveles para un First Playable es entre dos y tres. Para que el prototipo se convierta en un producto comercializable, debería tener un mayor número de niveles.
- Agregación de más elementos para los niveles: Existen algunos elementos que no pudieron añadirse por falta de tiempo: llaves, otras clases de enemigos, representación de algunos elementos del mundo real en el mundo de los sueños, etc. La agregación de nuevos elementos viene de la mano del desarrollo de más niveles, ya que es fundamental ir incorporando nuevas mecánicas para hacer la experiencia de juego más amena.
- Mejora de interfaz: En este TFG se cubren tres elementos de interfaz: menú principal, menú de pausa y timer del nivel. Podríamos cubrir más pantallas de interfaz que hagan la experiencia más agradable al jugador: pantalla de carga, confirmación de las selecciones que no se puedan deshacer, pantalla de final de nivel, etc.
- Feedback: Es necesario realizar un feedback a fondo del juego. Este juego ha sido testeado por algunas personas, pero no ha sido probado con un gran número de personas. Además, la edad de los testers ha sido entre 20 y 30 años, por lo que necesitamos hacer pruebas con un público más amplio para que evalúen el juego.
- Mejora de la estética: La estética visual de nuestro juego ha sido creada a partir de recursos gratuitos que hemos encontrado en la Asset Store de Unity y en páginas de gráficos a disposición del público. Debido a esto, no todos los elementos del escenario concuerdan entre ellos y no tienen una estética general que los aúne completamente. Una de las mejoras más notables que podrían realizarse sería que un diseñador logre crear una estética única para este prototipo.

10. Conclusiones

El desarrollo de este TFG me ha enseñado las nociones básicas del diseño y desarrollo de un videojuego. En un primer momento, pensaba que la implementación era una de las fases más complejas y largas del desarrollo total, pero aproximadamente un 30% del tiempo invertido en este proyecto ha sido para la fase de análisis y prototipado. Ha requerido una valoración especial y una fuerte carga de toma de decisiones.

Es necesario despegarse de la valoración subjetiva de nuestro juego, ya que muchas de las mecánicas planteadas en un primer momento que se contemplaban como perfectamente viable fueron rápidamente valoradas negativamente por resultar aburridas o tediosas. En estos casos siempre es mejor descartar estas ideas que pueden perjudicar a la calidad final del juego y comenzar a plantear nuevas ideas que se alejen de las primeras.

Por regla general el juego ha tenido un feedback positivo en los jugadores. Además, la mecánica del juego se adaptaba perfectamente a diferentes jugadores: algunos jugadores precavidos preferían realizar el cambio de mundo de manera lenta para no arriesgarse, mientras que otros jugadores más frenéticos buscaban en todo momento el riesgo, haciendo los cambios de mundo a ciegas para ver qué se encontraban. La capacidad de un juego de adaptarse al estilo de un jugador y no obligarle a jugar de cierta manera se considera positivo y aporta riqueza a nuestro producto final.

Este prototipo será mejorado en un futuro fuera de las horas establecidas para el TFG, ya que ofrece muchas oportunidades de mejora.

Este videojuego no está planteado para generar resultados económicos, pero puede servir como videojuego de presentación para oportunidades de trabajo.

11. Normativa y legislación

Legislación y delitos informáticos

El delito informático implica actividades criminales que los países han tratado de encuadrar en figuras típicas de carácter tradicional, tales como robos, hurtos, fraudes, falsificaciones, perjuicios, estafas, sabotajes. Sin embargo, debe destacarse que el uso de las técnicas informáticas han creado nuevas posibilidades del uso indebido de los ordenadores, lo que ha creado la necesidad de regulación por parte del derecho.

Tipos de delitos informáticos

La Organización de Naciones Unidas (ONU) reconocen los siguientes tipos de delitos informáticos:

1. Fraudes cometidos mediante manipulación de ordenadores

   - Manipulación de los datos de entrada.
   - Manipulación de programas
   - Manipulación de los datos de salida.
   - Fraude efectuado por manipulación informática.

2. Manipulación de los datos de entrada

   - Como objeto.
   - Como instrumento.

3. Daños o modificaciones de programas o datos computarizados

   - Sabotaje informático.
   - Acceso no autorizado a servicios y sistemas informático.
   - Reproducción no autorizada de programas informáticos de protección legal.

Además de estos tipos de delitos reconocidos, el XV Congreso Internacional de Derecho ha propuesto todas las formas de conductas lesivas de la que puede ser objeto la información.

- "Fraude en el campo de la informática.
- Falsificación en materia informática.
- Sabotaje informático y daños a datos computarizados o programas informáticos.
- Acceso no autorizado.
- Intercepción sin autorización.
- Reproducción no autorizada de un programa informático protegido.
- Espionaje informático.
- Uso no autorizado de un ordenador.
- Tráfico de claves informáticas obtenidas por medio ilícito.
- Distribución de virus o programas delictivos."

Legislación nacional

En España, los delitos informáticos son un hecho sancionable por el Código Penal en el que el delincuente utiliza, para su comisión, cualquier medio informático. Estas sanciones se recogen en la Ley Orgánica 10/1995, de 23 de Noviembre en el BOE número 281, de 24 de noviembre de 1995. Éstos tienen la misma sanción que sus homólogos no informáticos.

La legislación completa puede encontrarse en el siguiente enlace:

http://delitosinformaticos.com/legislacion/espana.shtml

Legislación internacional

Alemania

En Alemania, para hacer frente a la delincuencia relacionada con la informática, el 15 de mayo de 1986 se adoptó la Segunda Ley contra la Criminalidad Económica. Esta ley reforma el Código Penal (art. 148 del 22 de diciembre de 1987) para contemplar nuevos delitos como el espionaje de datos, la falsificación de datos probatorios y el sabotaje informático.

Austria

Según la Ley de reforma del Código Penal del 22 de diciembre de 1987, se contemplan los siguientes delitos:

- Destrucción de datos (art. 126).
- Estafa informática (art. 148).

Chile

Chile fue el primer país latinoamericano en sancionar una Ley contra Delitos Informáticos. La ley 19223 publicada en el Diario Oficial (equivalente del Boletín Oficial argentino) el 7 de junio de 1993 señala que la destrucción o inutilización de un sistema de tratamiento de información puede ser castigado con prisión de un año y medio a cinco.

Como no se estipula la condición de acceder a ese sistema, puede encuadrarse a los autores de virus. Si esa acción afectara los datos contenidos en el sistema, la prisión se establecería entre los tres y los cinco años.

El hacking, definido como el ingreso en un sistema o su interferencia con el ánimo de apoderarse, usar o conocer de manera indebida la información contenida en éste, también es pasible de condenas de hasta cinco años de cárcel; pero ingresar en ese mismo sistema sin permiso y sin intenciones de ver su contenido no constituye delito.

Dar a conocer la información almacenada en un sistema puede ser castigado con prisión de hasta tres años, pero si el que lo hace es el responsable de dicho sistema puede aumentar a cinco años. Esta ley es muy similar a la inglesa aunque agrega la protección a la información privada.

Estados Unidos

El primer abuso de un ordenador se registró en 1958. En la primera mitad de la década del 70, mientras los especialistas y criminólogos discutían si el delito informático era el resultado de una nueva tecnología o un tema específico, los ataques computacionales se hicieron más frecuentes. Para acelerar las comunicaciones, enlazar compañías, centros de investigación y transferir datos, las redes debían (y deben) ser accesibles, por eso el Pentágono, la OTAN, las universidades, la NASA, los laboratorios industriales y militares se convirtieron en el blanco de los intrusos.

Pero en 1976 dos hechos marcaron un punto de inflexión en el tratamiento policial de los casos: el FBI dictó un curso de entrenamiento para sus agentes acerca de delitos informáticos y el Comité de Asuntos del Gobierno de la Cámara presentó dos informes que dieron lugar a la Ley Federal de Protección de Sistemas de 1985.

Esta ley fue la base para que Florida, Michigan, Colorado, Rhode Island y Arizona se constituyeran en los primeros estados con legislación específica, anticipándose un año al dictado de la Computer Fraud y Abuse Act de 1986.

Este se refiere en su mayor parte a delitos de abuso o fraude contra casas financieras, registros médicos, ordenadores de instituciones financieras o involucradas en delitos interestatales. También especifica penas para el tráfico de claves con intención de cometer fraude y declara ilegal el uso de passwords ajenas o propias en forma inadecuada. Pero sólo es aplicable en casos en los que se verifiquen daños cuyo valor supere el mínimo de mil dólares.

En 1994 se adoptó el Acta Federal de Abuso Computacional (18 U.S.C. Sec 1030), modificando el Acta de 1986. Aquí se contempla la regulación de los virus (computer contaminant) conceptualizándolos, englobando todas las instrucciones designadas a contaminar otros grupos de programas o bases de datos.

Modificar, destruir, copiar, transmitir datos o alterar la operación normal de los ordenadores, los sistemas o las redes informáticas es considerado delito. Así, esta ley es un acercamiento real al problema, alejado de argumentos técnicos para dar cabida a una nueva era de ataques tecnológicos.

Francia

Aquí, la Ley 88/19 del 5 de enero de 1988 sobre el fraude informático contempla:

- Acceso fraudulento a un sistema de elaboración de datos.
- Sabotaje Informático.
- Destrucción de datos.
- Falsificación de documentos informatizados.

Holanda

Hasta el día 1 de marzo de 1993, día en que entró en vigencia la Ley de Delitos Informáticos, Holanda era un paraíso para los hackers. Esta ley contempla con artículos específicos sobre técnicas de Hacking y Phreacking.

El mero hecho de entrar en un ordenador en la cual no se tiene acceso legal ya es delito y puede ser castigado hasta con seis meses de cárcel. Si se usó ese ordenador hackeado para acceder a otro, la pena sube a cuatro años.

El daño a la información o a un sistema de comunicaciones puede ser castigado con cárcel de seis meses a quince años. Cambiar, agregar o borrar datos puede ser penalizado hasta con dos años de prisión pero, si se hizo vía remota aumenta a cuatro.

Los virus están considerados de manera especial en la ley. Si se distribuyen con la intención de causar problemas, el castigo puede llegar hasta los cuatro años de cárcel; si simplemente se "escapó", la pena no superará el mes.

Inglaterra

Luego de varios casos de hacking surgieron nuevas leyes sobre delitos informáticos. En agosto de 1990 comenzó a regir la Computer Misuse Act (Ley de Abusos Informáticos) por la cual cualquier intento, exitoso o no de alterar datos informáticos con intención criminal se castiga con hasta cinco años de cárcel o multas sin límite.

El acceso ilegal a un ordenador contempla hasta seis meses de prisión o multa de hasta dos mil libras esterlinas.

El acta se puede considerar dividida en tres partes: hackear (ingresar sin permiso en un ordenador), hacer algo con la computadora hackeada y realizar alguna modificación no autorizada.

El último apartado se refiere tanto al hacking (por ejemplo, la modificación de un programa para instalar un backdoor), la infección con virus o, yendo al extremo, a la destrucción de datos como la inhabilitación del funcionamiento del ordenador.

Bajo esta ley liberar un virus es delito y en enero de 1993 hubo un raid contra el grupo de creadores de virus. Se produjeron varios arrestos en la que fue considerada la primera prueba de la nueva ley en un entorno real.

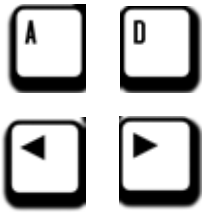
12. Manual de usuario

Los sueños son irreales. Son fantasías, meras representaciones mentales que desaparecen al despertar. No podemos repetirlos, ni evitarlos. Sin embargo, la joven Ava es capaz de hacerlo: puede acceder a ese lugar que los demás sólo visitan mientras duermen. Pero uno nunca elige lo que sueña. Y a veces los sueños pueden convertirse en verdaderas pesadillas. Acompaña a Ada a buscar la salida del bosque y de sus sueños.

CONTROLES

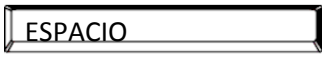
MANDO DE XBOX

TECLADO



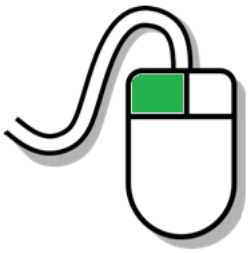
Mover personaje

Saltar



Atacar

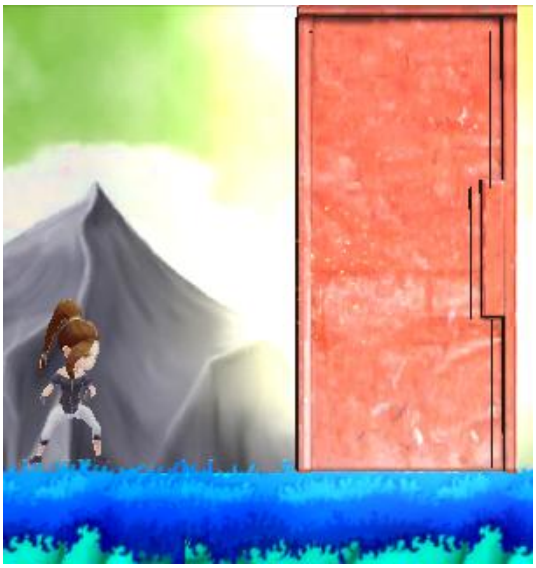
Pausa



Cambiar de mundo (instantáneo)

Cambio de mundo(lento)

Encuentra la puerta de salida de cada nivel



¡Cuidado con los enemigos!

El escudo te protegerá de un golpe, pero si lo pierdes, el próximo golpe te devolverá a un punto de control

Cambia de mundo de manera lenta, y podrás ver qué va a pasar

Bibliografía

Libros e informes

- Jesse Schell. *The Art of Game Design: A book of lenses.*
- DEV. *Libro blanco del desarrollo español de los videojuegos:* http://www.dev.org.es/images/stories/docs/LibroBlancoDEV%20alta_compr.pdf

Páginas web

- http://www.segu-info.com.ar/delitos/delitos.htm
- https://es.wikipedia.org/
- http://stackoverflow.com/questions/1901251/component-based-game-engine-design/3495647#3495647
- http://www.gamasutra.com/view/feature/166972/cognitive_flow_the_psychology_of_.php?print=1
- http://articulos.softonic.com/que-es-un-motor-grafico-o-motor-3d
- http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/

Apuntes de asignaturas cursadas en la ULPGC

- Apuntes de Ingeniería del Software I
- Apuntes de Ingeniería del Software II
- Apuntes de Diseño de Interfaces de Usuario
- Apuntes de Gestión del Software I
- Apuntes de Herramientas profesionales para desarrollo de software

Blogs y foros

- https://unity3d.com/es/community
- http://gamasutra.com/

Documentación Unity

Manual de Unity

- http://docs.unity3d.com/Manual/

Tutoriales de Unity

- http://unity3d.com/learn/tutorials/

Recursos

Sonidos

- http://freesound.org/
- http://www.freesfx.co.uk/
- http://freemusicarchive.org/

Gráficos

- http://opengameart.org/
- Asset Store de Unity

Anexo 1: Documento de diseño

TÍTULO DEL JUEGO: THRESHOLD

1. Historia y guión

Los sueños son irreales. Son fantasías, meras representaciones mentales que desaparecen al despertar. No podemos repetirlos, ni evitarlos. Sin embargo, la joven Ava es capaz de hacerlo: puede acceder a ese lugar que los demás sólo visitan mientras duermen. Pero uno nunca elige lo que sueña. Y a veces los sueños pueden convertirse en verdaderas pesadillas.

2. Personaje

El personaje será una joven llamada Ava que contará con los siguientes movimientos:

- Cambiar de mundo
- Saltar
- Correr
- Atacar con puñetazo

Existen dos mundos. La diferencia entre estos dos mundos viene dada por el diseño del nivel (existirán obstáculos en uno de los mundos que necesiten sortearse cambiando al otro mundo):

- Mundo real, de estética realista y colores oscuros.
- Mundo de los sueños, de estética surrealista y colores vivos.

El personaje podrá realizar las siguientes acciones:

- Saltar obstáculos
- Subir a plataformas
- Atacar enemigos
- Recoger objetos
- Sortear obstáculos cambiando de un mundo a otro

3. Enemigos

Existirán dos clases de enemigos: enemigos pequeños y enemigos grandes.

- Los enemigos pequeños se moverán por una zona restringida del mapa, repitiendo el mismo patrón, y el personaje podrá atacar para eliminarlos del mapa o esquivarlos con el salto.
- Los enemigos grandes no se moverán, atacarán si el personaje se acerca y no recibirán daño de ataques normales. Se podrán eliminar del mapa recogiendo un *power up* (mejora) que aumentará el ataque y atacando al enemigo con este nuevo poder.

Los enemigos solamente existen en el mundo de los sueños.

4. Obstáculos

Los escenarios constarán de una secuencia de obstáculos. El jugador tendrá que encontrar la manera de pasar a través de ellos utilizando los diferentes movimientos del personaje. Algunos de los posibles obstáculos serán:

- Puentes que aparecen y desaparecen al cambiar de mundo.
- Plataformas que aparecen y desaparecen al cambiar de mundo.
- Enemigos pequeños en movimiento.
- Plataformas demasiado grandes para saltar que requieran de cajas para llegar a ellas.
- Enemigos grandes que no puedan atacarse y necesiten eliminarse con un poder especial.
- Zonas de caída libre.
- Pinchos estáticos que aparecen y desaparecen al cambiar de mundo.
- Puertas que se abren y cierran al cambiar de mundo.

5. Temática de niveles

Los niveles estarán ambientados en el bosque, ya que se pueden utilizar muchos recursos de vegetación. El arte será una mezcla de 2D y 3D:

- Personaje, enemigos, y elementos de decoración en 3D.
- Plataformas y fondos en 2D.

6. Interfaz (menú, HUD)

Los elementos de la interfaz serán:

- Menú: En el menú podrás modificar las opciones y comenzar una partida. Al comenzar una partida podrás elegir un nivel, pero la primera vez que se juegue todos los niveles excepto el primero estarán bloqueados. A medida que el jugador supere niveles se irán desbloqueando los demás niveles.
- Pausa: En el menú de pausa se podrá continuar con el juego, volver al menú principal y modificar las mismas opciones que estaban disponibles en el menú.
- HUD: El único elemento que aparecerá será la barra de carga del mundo cuando se esté pasando de un mundo a otro.

7. Condiciones para ganar/perder

El objetivo del juego es llegar al final del nivel. Se avanzará caminando hacia la derecha. El jugador tendrá que llegar al final del nivel para continuar al siguiente nivel.

El jugador dispondrá de dos vidas. La primera vida se simbolizará con un escudo que rodeará a la protagonista, y al recibir el primer golpe desaparecerá, dando a entender al usuario que no puede recibir más golpes. Cada elemento hostil quitará una vida, por lo que a los dos golpes que reciba el jugador se reiniciará el nivel o se llevará a un punto de control.

Los siguientes elementos serán elementos hostiles:

- Pinchos
- Enemigos pequeños
- Enemigos grandes

Además, caer al vacío hará que el jugador pierda todas las vidas automáticamente y se reiniciará el nivel o se llevará al punto de control.

8. Sonido

Música: La música sonará más fuerte en el mundo de los sueños que en el mundo real. Se aumentará o disminuirá el volumen dependiendo del mundo en el que se encuentre. La música escogida es "Air Hockey Saloon", de Chris Zabriskie.

Sonidos: Existirán sonidos para los siguientes elementos:

- Cambio de mundo completado
- Cambio de mundo en progreso
- Salto del personaje
- Ataque del personaje
- El personaje recibe daño
- El enemigo ataca
- El enemigo muere

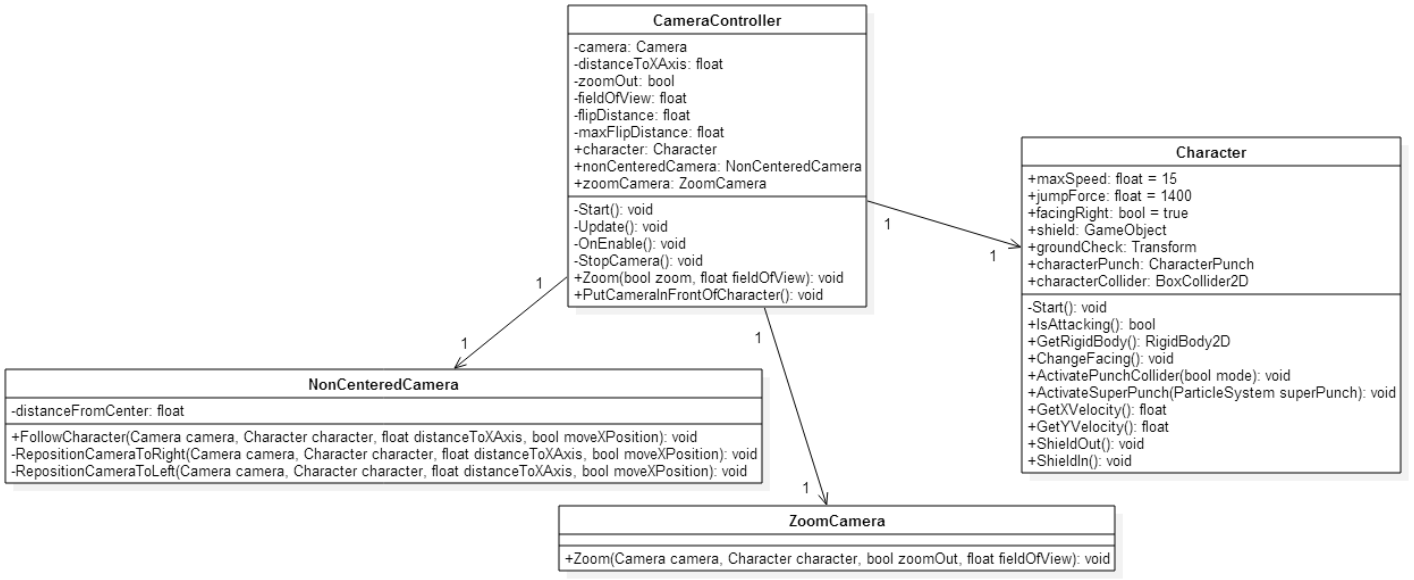
9. Cámara

La cámara se desplazará con el personaje en todo momento. El personaje se verá descentrado de la pantalla, es decir, que estará posicionado hacia la izquierda o derecha, dependiendo de hacia dónde esté mirando.

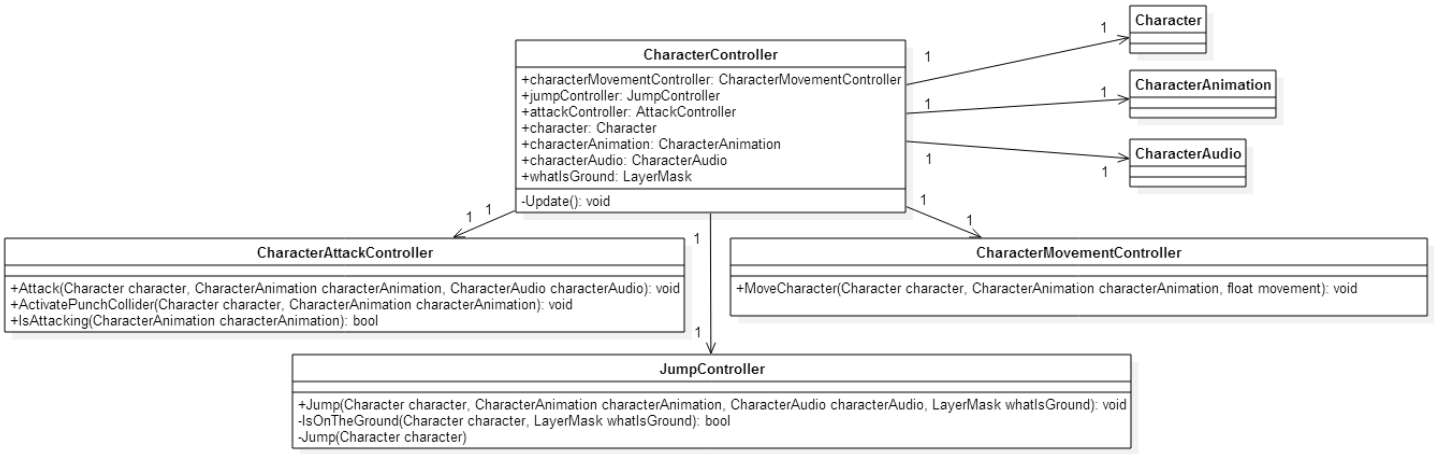
Existirán movimientos de zoom in y zoom out para las caídas libres, para observar mejor los obstáculos que existan.

Anexo 2: Arquitectura del software

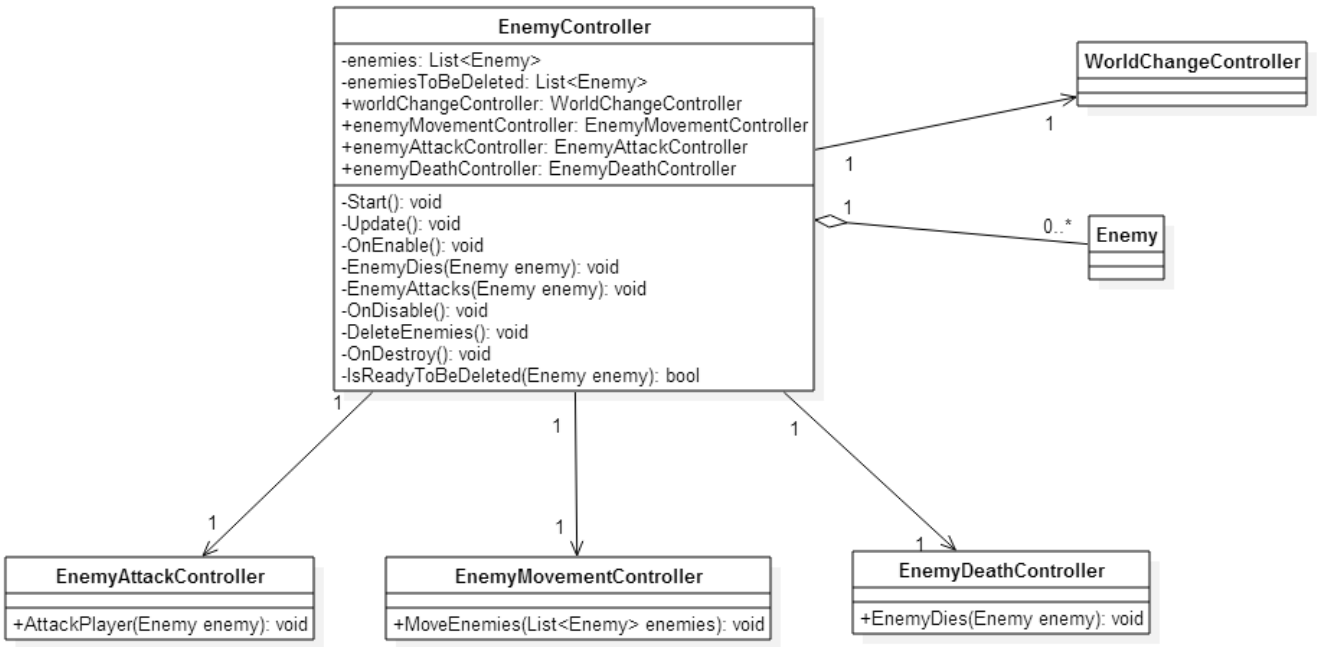
1. Cámaras



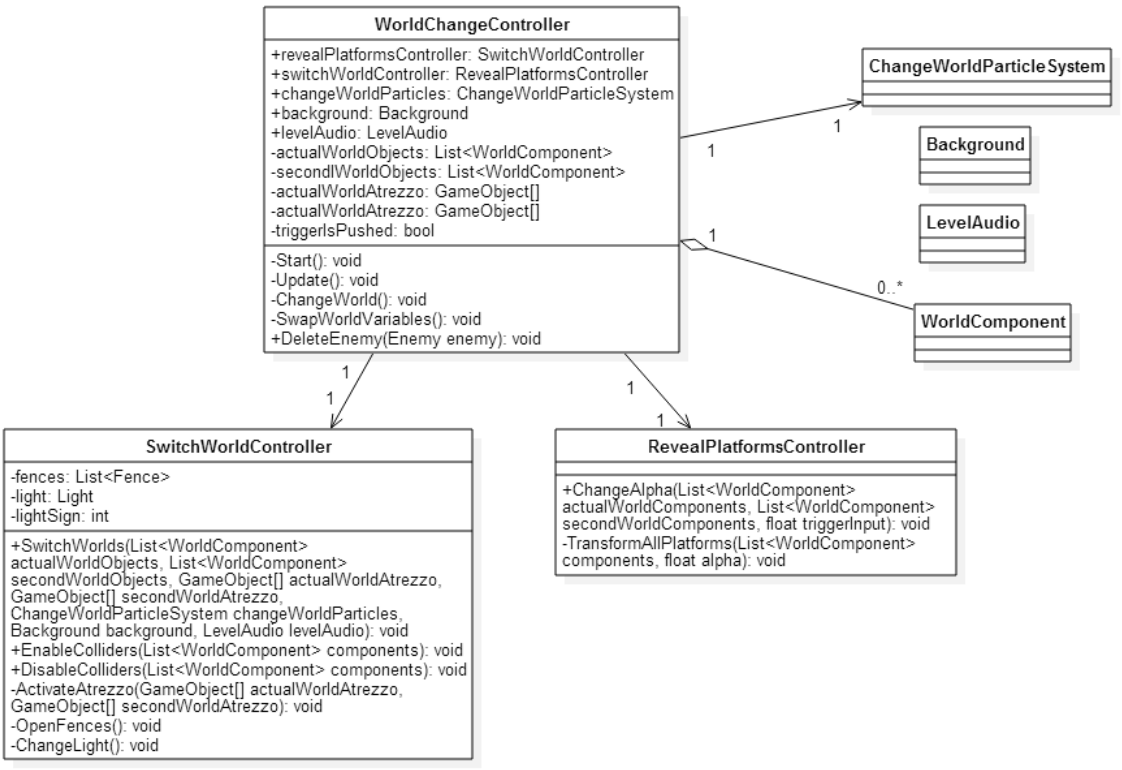
2. Character Controller



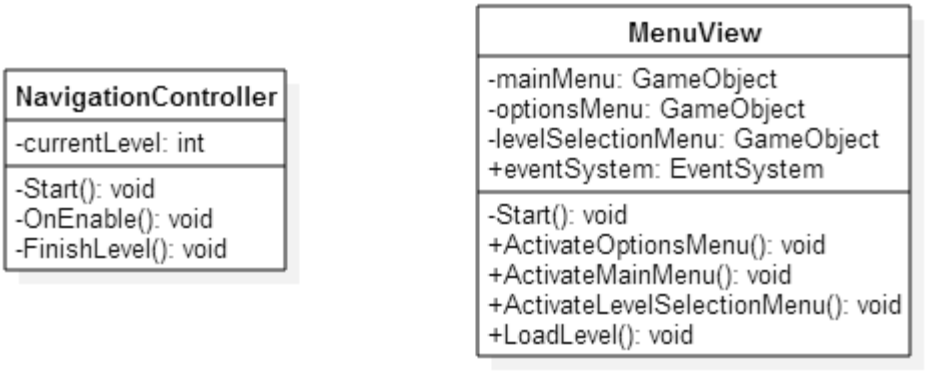
3. Enemy Controller



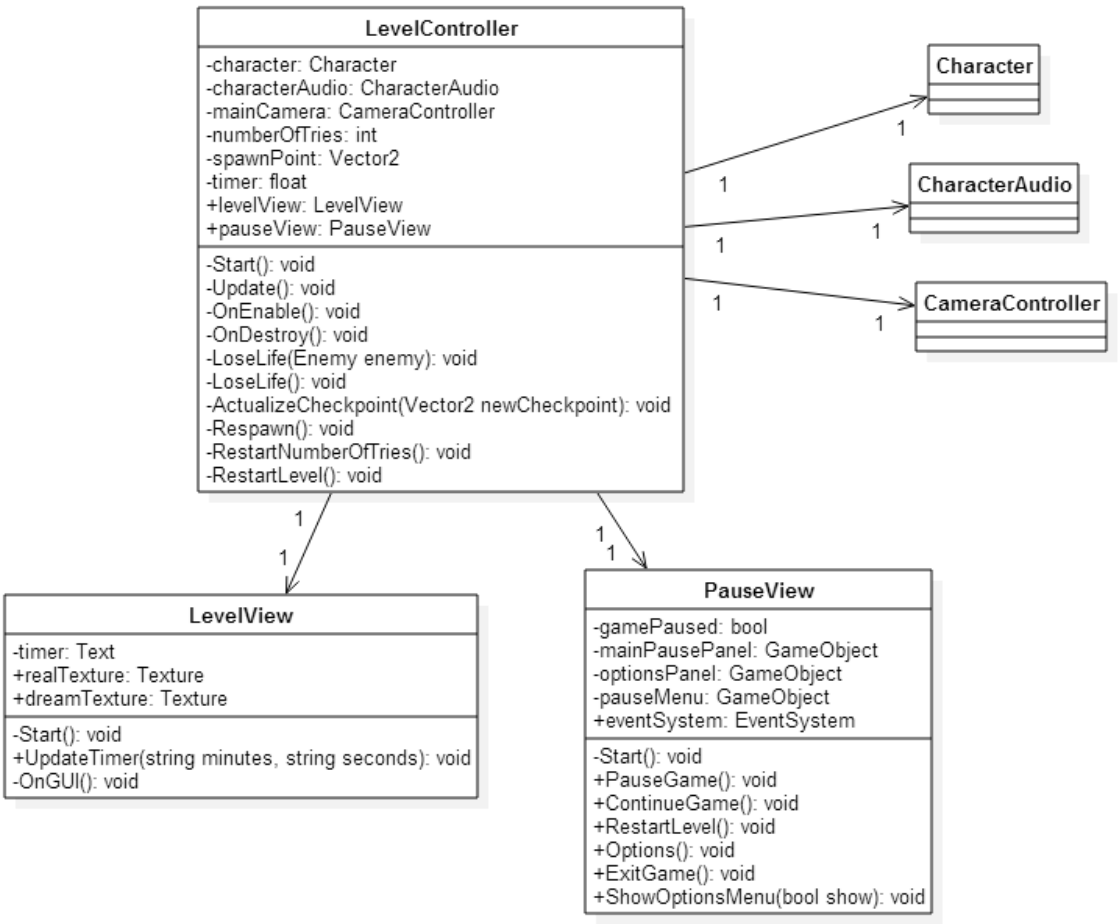
4. World Change Controller



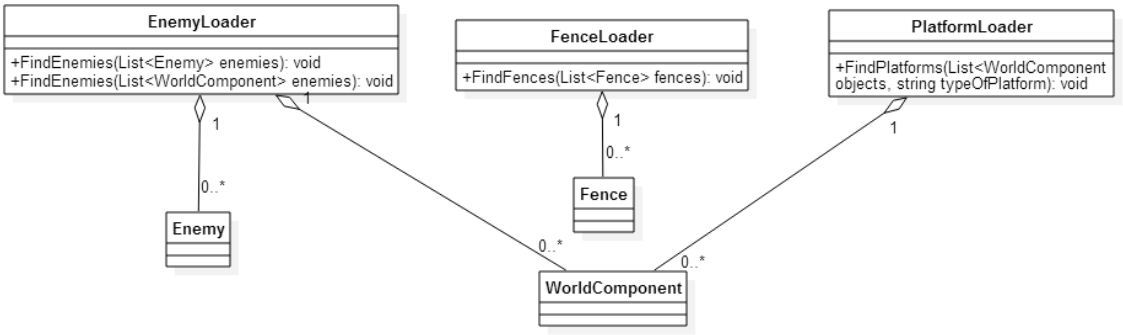
5. Navigation Controller



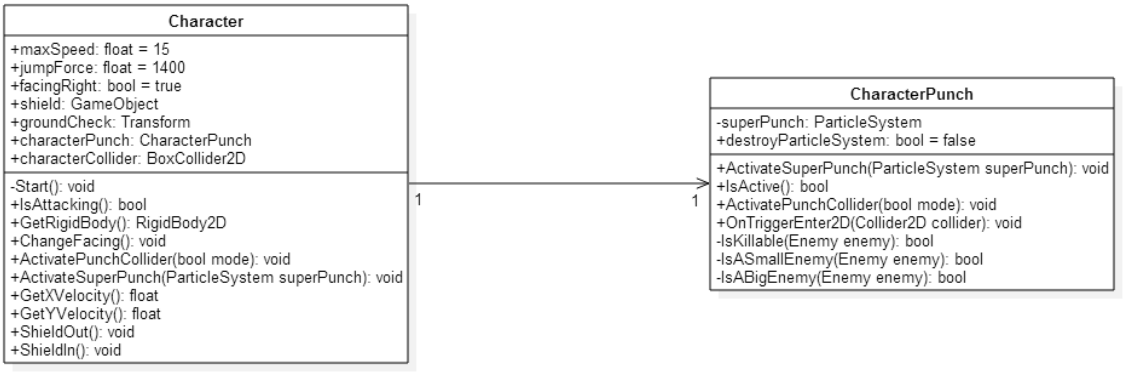
6. Level Controller



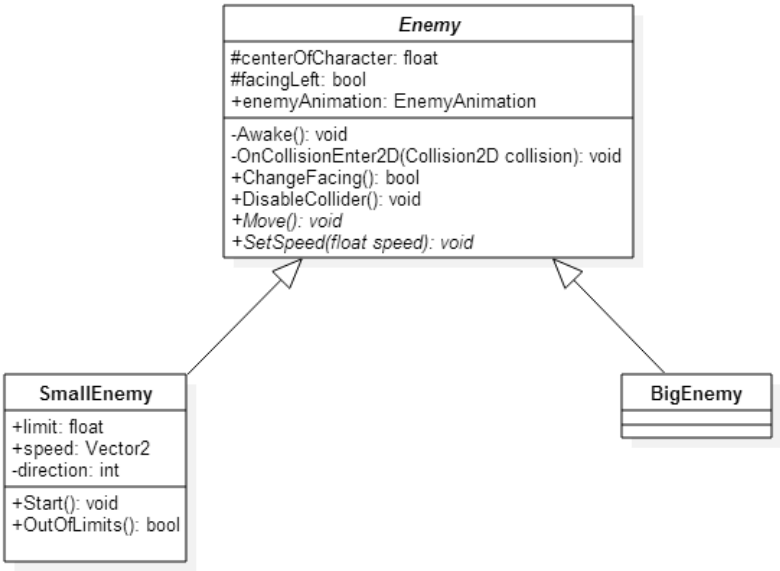
7. Loaders



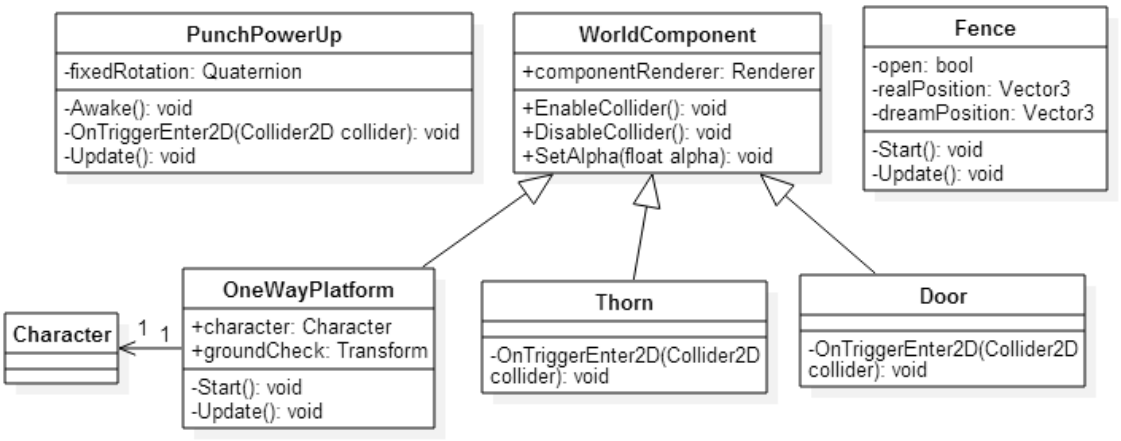
8. Character Model



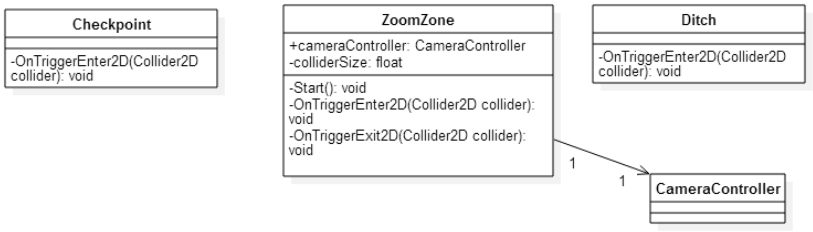
9. Enemy Model



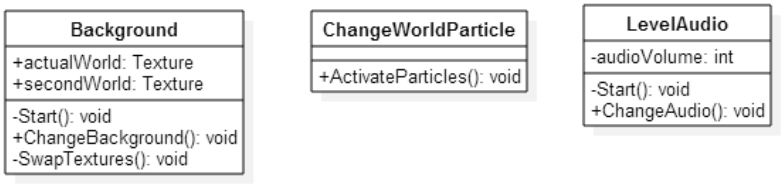
10. Environment Model



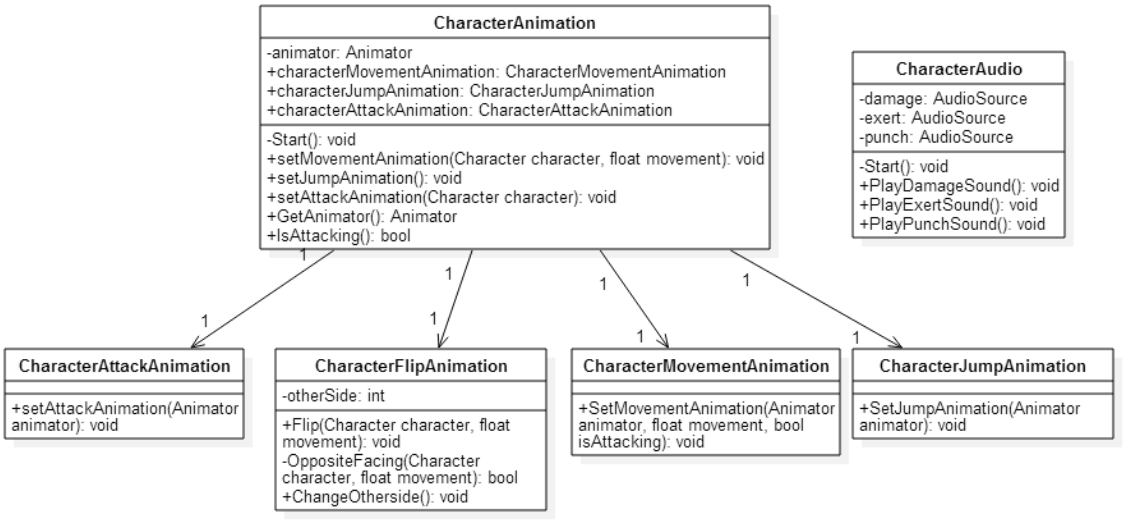
11. Untangible Model



12. World Change View



13. Character View



14. Enemy View