UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE GRADO

Journey of the Mechanic Village
VideoJuego para Plataformas Móviles

Juan Carlos Fernández Durán

Diciembre, 2016

JOURNEY OF THE MECHANIC VILLAGE
VIDEOJUEGO PARA PLATAFORMAS MÓVILES

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Tecnologías y Sistemas de Información

TECNOLOGÍA ESPECÍFICA DE COMPUTACIÓN

TRABAJO FIN DE GRADO

Journey of the Mechanic Village
VideoJuego para Plataformas Móviles

Autor: Juan Carlos Fernández Durán

Director: David Vallejo Fernández

Diciembre, 2016

Juan Carlos Fernández Durán

Ciudad Real – Spain

E-mail: fdz.juancarlos@gmail.com
Teléfono: 696 364 171

TRIBUNAL:

Presidente:

Vocal:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL	SECRETARIO
Fdo.:	Fdo.:	Fdo.:

Resumen

El presente documento es un ejemplo de memoria del Trabajo de Fin Grado según el formato y criterios de la Escuela Superior de Informática de Ciudad Real. La intención es que este texto sirva además como una serie de consejos sobre tipografía, LaTeX, redacción y estructura de la memoria que podrían resultar de ayuda. Por este motivo, se aconseja al lector consultar también el código fuente de este documento.

Este documento utiliza la clase LaTeX *esi-tfg*, disponible como paquete Debian/Ubuntu, consulta:

`https://bitbucket.org/arco_group/esi-tfg`.

Si encuentra cualquier error o tiene alguna sugerencia, por favor, utilice el *issue tracker* del proyecto *esi-tfg* en:

`https://bitbucket.org/arco_group/esi-tfg/issues`

El resumen debería estar formado por dos o tres párrafos resaltando lo más destacable del documento. No es una introducción al problema, es decir, debería incluir los logros más importantes del proyecto. Suele ser más sencillo escribirlo cuando la memoria está prácticamente terminada. Debería caber en esta página (es decir, esta cara).

Abstract

English version of the previous page.

Agradecimientos

Escribe aquí algunos chascarrillos simpáticos. Haz buen uso de todos tus recursos literarios porque probablemente será la única página que lean tus amigos y familiares. Debería caber en esta página (esta cara de la hoja).

Juan[1]

[1] Sí, los agradecimientos se firman

A alguien querido y/o respetado

Índice general

Índice de cuadros

Índice de figuras

Índice de listados

Listado de acrónimos

GDD	Game Document Design
TFG	Trabajo Fin de grado

Capítulo 1

Introducción

DESDE 1972 con la puesta en producción de la máquina recreativa Pong, la industria del videojuego ha avanzado de manera progresiva a lo largo de los años viviendo la aparición de la consola de sobremesa donde cada persona podía obtener un dispositivo asequible donde poder disfrutar de los videojuegos sin que fuera necesario gastar dinero en las máquinas recreativas, más tarde vivimos la capacidad de poder disfrutar de esa potencia de una manera portable para poder disfrutar en cualquier lugar de los videojuegos con la llegada de la **Game Boy** en el año 1989.

Por aquel entonces la industria estaba enfocada a personas muy jóvenes, pero la industria fue madurando tanto junto con el consumidor contándonos historias más adultas como con la tecnología, siempre a la vanguardia de los últimos avances, los desarrolladores se esforzaban por exprimir al máximo la capacidad técnica de la que disponían para poder ofrecer una mejor experiencia a sus jugadores.

Para continuar con la introducción, primero debemos introducir a qué nos referimos con el término **videojuego** desde una perspectiva técnica. Un videojuego es una aplicación de renderizado en tiempo real, la cual posee una importante componente interactiva. Esta aplicación está basado en un bucle en el que cada iteración se debe renderizar una imagen, se captura una entrada por parte del usuario y en función de ese conjunto de acciones, la aplicación se retroalimenta mostrando una u otra salida [DV15].

La industria del videojuego está actualmente en auge, moviendo en todo el mundo un volumen de dinero equivalente a aproximadamente 99,6 miles de millones de dólares en 2016 desglosados en la TODO FIGURA 1 [Bla15].

Actualmente podemos desglosar el mercado del videojuego en varios sectores de enorme importancia entre los que se incluyen los desarrollos en consola, en PC, y en móviles. De manera tradicional, los videojuegos siempre han sido relevantes en el sector de las consolas tradicionales, sin embargo, con el paso de tiempo y la llegada de los smartphones, el sector de móviles y tablet ha ido ganando una gran relevancia.

Actualmente en las grandes plataformas de distribución de aplicaciones como Google Play para Android o AppStore para iOS, el 41 % de las descargas de aplicaciones fueron juegos, y

el 59 % restante son aplicaciones de otro tipo, sin embargo, el 85 % del gasto se debe a esos videojuegos de plataformas móviles [Bla15].

Sin embargo el desarrollo de un videojuego no es un componente trivial, ya que involucra un equipo multi-disciplinar en el que se ven involucradas varios tipos de perfiles profesionales entre los que se incluyen informáticos, diseñadores, artistas, músicos así como otros perfiles especializados en la conciliación de estas áreas como los productores o recursos humanos. Además tampoco es fácil deducir qué juego puede gustar el público y cual no, a diferencia del desarrollo de una aplicación tradicional, aquí perdemos la figura del cliente el cual nos especifica todos aquellos requisitos de nuestra aplicación, si no que debemos innovar con un producto totalmente nuevo y presentarlo al público con el correspondiente riesgo que eso puede llevar.

El ámbito del desarrollo móvil es un campo relativamente nuevo en el que se juega de diferente manera y por lo tanto, debemos de ir más allá de los diseños tradicionales del videojuego y pensar en otros juegos pensados para la plataforma móvil, en la que destaca la corta duración en la que el jugador interactúa con el móvil y que la forma de interacción entre el usuario y el juego debe estar especialmente cuidada debido a que el usuario sólo puede interactuar con el juego en la propia pantalla táctil.

El modelo de distribución ha ido evolucionando a lo largo de todos estos años, beneficiando en último término a los desarrolladores al quedar cada vez menos intermediarios entre estos y los propios consumidores. A partir de 2010 el hasta ahora dominante formato físico fue cediendo terreno ante la distribución digital, proporcionando un 95 % de beneficios a los desarrolladores frente al 20 % que alcanzaban en formato físico [Bla15]. Sin embargo después de tanta actividad, todos los datos apuntan a que la industria del videojuego tiene camino por recorrer, alcanzando en 2014 una facturación de 413 millones de euros en España, con expectativas de que alcanzará los 998 millones de euros, creciendo a una tasa anual compuesta del 24,7 % [Bla15]. De los 101 miles de millones de dólares que factura la industria del videojuego, 21,6 miles de millones pertenecen a la industria móvil del videojuego [Bla15] de manera que, como podemos observar, este es un sector muy atractivo que no para de crecer y que requerirá una gran demanda de desarrolladores de videojuegos.

Journey of the Mechanic Village es un paso más en la historia de los videjuegos, distribuido de manera gratuita, combina el género **Runner** [Gre09] , en el que tenemos a un personaje que corre por un camino determinado, en el que es necesaria la superación de diferentes obstáculos, junto con el añadido de un **Sistema de Mejoras**. Este sistema de mejoras está basado en la gestión de recursos y el tiempo. El jugador ordenará la mejora de una parte del robot concreta y esta requerirá recursos virtuales del juego y tiempo. En los primeros niveles requerirá minutos, pero conforme el juego avance y requerirá más tiempo para completar las mejoras como minutos, horas, o incluso días. El objetivo del juego consiste en recorrer este camino atravesando adversidades e intentar llegar al final. Llegar al final del camino será

muy difícil incluso para los jugadores que tengan a su personaje muy mejorado, por lo tanto el objetivo además de llegar al final es llegar lo más lejos posible para competir de manera asíncrona contra otros jugadores, de manera que los jugadores que más puntuación obtengan al día serán recompensados con recursos especiales del juego. Este ciclo de recompensa se repetirá cada día incentivando a los jugadores a seguir jugando cada día.

Con esta premisa, Journey of the Mechanic Village propone una solución fácil para entretener en esos lapsos de tiempo de tiempo donde el jugador en su vida diaria necesita esperar, como en cualquier sala de espera, viajando en un medio de transporte público o en cualquier otra situación. La portabilidad y la potencia de los dispositivos móviles proporcionan las herramientas necesarias para que cualquier persona en cualquier situación pueda entretenerse con tan solo utilizar su móvil.

Al igual que los dispositivos se han hecho con el tiempo más potentes, también se ha mejorado progresivamente con el tiempo las metodologías y técnicas de carácter ingenieril para mejorar la calidad del producto final. Antiguamente con cada nuevo juego se tenía que plantear un nuevo desarrollo Software que cubría todos los aspectos del juego, sin embargo se fué abstrayendo aquello que era común a todos los videojuegos dando lugar al concepto de **Motor de juego** [Gre09]. Este motor actúa como base sobre la que construir en cualquier videojuego, proporcionando una solución a los problemas comunes a todos los desarrollos, como la renderización 3D, la detección de teclas pulsadas por el usuario, la parte sonora o las físicas. Además los motores más modernos proporcionan editores que pueden ser utilizados tanto por programadores como por artistas y diseñadores de juego. Estos diseñadores por ejemplo pueden crear las pantallas de juego fácilmente, y al trabajar todo el equipo sobre el mismo motor, permite el trabajo colaborativo entre todas las partes involucradas en el desarrollo de un videojuego. f

Cabe destacar que el motor de juego seleccionado para el desarrollo de este proyecto es **Unity**, el cual dispone de un buen editor así como una abstracción a nivel de programación donde el juego se desarrolla a partir del paradigma de la *programación orientada a componentes* mediante scripts construidos en *C#*, *UnityScript* o *Boo*, en el caso del presente proyecto, *C#* dada la cercanía del lenguaje con la sintaxis de *Java*.

La *programación orientada a componentes* enfatiza la separación de funcionalidad en distintos scripts independientes, de manera que cuando tengamos una entidad de juego, podemos componer esa entidad de diferentes scripts que definen esa entidad, como por ejemplo, que un enemigo posee vida, puede moverse y es susceptible de ser atacado por el jugador. Esta práctica persigue una **arquitectura débilmente acoplada**, de manera que sus componentes se pueden reutilizar tanto en el mismo desarrollo como en otros desarrollos.

Además el juego contará con una capacidad procedural de combinar partes de niveles de manera procedimental dada una arquitectura previamente diseñada el cual combinará peque-

ñas partes del escenario y las irá combinando de manera inteligente a medida que el jugador avanza en el nivel, solucionando gran parte de la problemática del diseño de nivel del propio juego y proporcionando variedad y admitiendo en cualquier momento integrar nuevas piezas de escenario en su plantilla permitiendo una mayor disposición de piezas de las que hacer el nivel.

A la hora de desarrollar un videojuego, es importante el conocimiento de diseño y arquitectura, proporcionando grandes beneficios como la reutilización de código, un código mantenible y mayor eficiencia a nivel de código. Esto no solo es importante en el desarrollo de videojuegos si no también en cualquier desarrollo software. Un buen diseño puede ahorrar muchos problemas en el día del mañana, pero también hay que tener cuidado con el **Overengineering**, ya que hacer el diseño de tu producto más complicado de lo necesario puede ser perjudicial para el ciclo de vida de tu proyecto.

1.1 Estructura del documento

A continuación se muestra la estructura de este documento, se nombran cada uno de los capítulos con su correspondiente descripción sobre el contenido de los mismos.

[TODO NOTA DE AUTOR: Se irá detallando esta sección conforme se desarrollen los capítulos]

Capítulo 1 Introducción

Proporciona un vistazo global al contexo sobre el que se desarrolla el proyecto y una breve introducción al mismo, así como la estructura del documento.

Capítulo 2 Objetivos

En este capítulo se presenta el objetivo general del trabajo fin de grado así como una descripción de los objetivos específicos que se deben cumplir para dar paso a cumplir el hito general.

Capítulo 3 Estado del Arte

Se mostrará una visión global sobre los proyectos de características similares así como sus diferencias.

Capítulo 4 Metodología

En este capítulo se expone el método de trabajo seguido a lo largo del ciclo de vida del proyecto.

Capítulo 5 Arquitectura del Sistema

Se mostrará cómo está compuesta la arquitectura software del proyecto.

Capítulo 6 Resultados

Se expondrán el resultado del proyecto.

Capítulo 7 Conclusiones

Capítulo 8 Trabajo Futuro

Capítulo 2

Objetivos

En el presente capítulo se establecerán los principales objetivos del Trabajo fin de grado, así como una descripción de los mismos con un correspondiente desglosamiento en objetivos específicos.

2.1 Objetivo general

El hito final consiste en el desarrollo de un Videojuego para plataforma móvil cumpliendo con los requisitos básicos para que nuestra aplicación interactiva pueda ser usable de cara al público.

El completo desarrollo del Videojuego comprende el diseño del mismo, en un documento el cual contendrá las especificaciones y requisitos del mismo, el cual llamaremos de ahora en adelante como GDD. Este documento especifica cómo es nuestro juego y por lo tanto, cómo divierte al usuario. La primera fase del proyecto será actuar como un Stakeholder y especificar aquello que nuestra aplicación contendrá para que la misma sea divertida para los usuarios. Este artefacto será necesario en la fase de desarrollo.

Para el desarrollo de cualquier Software es necesario contar con los requisitos y necesidades del cliente para saber qué es lo que se quiere desarrollar. En nuestro caso, este documento será el GDD.

Con dicho documento podemos comenzar la fase de desarrollo, en la cual podemos planificar los requisitos en base a la metodología que usemos y comenzar con el desarrollo del mismo.

2.2 Objetivos específicos

A continuación se presentarán los objetivos específicos del presente TFG

2.2.1 Definición formal del Videojuego

Es necesario formalizar desde la perspectiva del jugador qué es el videojuego y de qué se va a componer. Esto consiste de la redacción de un documento en el que se especifica de qué está compuesto el juego y que servirá de guía a la implementación del mismo ya que

representa las necesidades del jugador para entretenerse. A partir de esta formalización se podrá comenzar el desarrollo del videojuego. Este artefacto es conocido como GDD

2.2.2 Usabilidad y Rendimiento del videojuego

Para el desarrollo de cualquier aplicación interactiva es necesario generar una tasa de generación de imágenes mínima para producir una sensación de movimiento en el usuario. También es necesario tener una capacidad de respuesta mínima ante la entrada de un usuario.

Esto requiere de una tasa de refresco para nuestra aplicación de 30 a 60 frames por segundo. A pesar de que se percibe una sensación de movimiento con una tasa de refresco menor, al tratarse de una aplicación en la que el usuario interacciona en tiempo real con ella, es necesario mantener esa tasa por encima de como mínimo a 30 frames por segundo.

2.2.3 Arquitectura débilmente acoplada

Mediante la orientación a componentes que propone el sistema de Scripting de Unity, se pretende conseguir una arquitectura débilmente acoplada basada en componentes, de manera que los scripts que representen una función (es enemigo, se puede matar, se puede mover) se convierta en un script modular que aplique dicha función.

Con esta aproximación podemos lograr un código mantenible y que permitirá su reutilización en trabajos futuros.

2.2.4 Transparencia y mantenibilidad de servicios

Es necesario proveer una transparencia a la hora de acceder al servidor de manera que la dirección de acceso sea la misma para todas las máquinas. Así mismo es necesario tener localizado y mantener el servicio central que atenderá a las peticiones web de las aplicaciones móviles.

2.2.5 Despliegue de servicios

Se debe realizar un despliegue en 2 plataformas claramente diferenciadas, por un lado, se debe realizar el despliegue del videojuego en una plataforma de distribución digital de aplicaciones móviles. La aplicación servidora se debe desplegar en un contenedor de aplicaciones, localizado en un servidor público y accesible.

Capítulo 3

Estado del Arte

En el presente capítulo haremos una breve revisión de los actuales motores de juego, la trayectoria de los videojuegos móviles así como proyectos con similitudes a Journey of the Mechanic Village.

3.1 Videojuegos en la actualidad

El videojuego es una de las industrias de ocio y entretenimiento líder del mundo, en el mercado mundial llegó a generar en 2015 unas ventas de 91.800 millones de dólares según datos de Newzoo, esto supera en más del doble a la asociación cinematográfica y en más de seis veces al mercado de la música. A pesar de la gran cantidad de dinero que mueven a cantidad mundial, aún está tan socialmente aceptado como sí lo está la industria del cine o de la música, debido a que es una industria aún muy jóven cuya historia está ligada a la historia de la informática.

Sin embargo, uno de los datos más impactantes es que se prevee que el mercado global seguirá creciendo a una tasa anual del 6,6 % llegando en 2019 a los 118.600 millones de dólares (ver Figura 3.1.

Todo este mercado está dividido en 3 plataformas principales.

- **Plataforma Móvil**, hoy en día cada persona dispone de un móvil, un pequeño ordenador en potencia que además de todo lo que permite hacer, es ya de por sí una estación de juego robusta y utilizada en todo el mundo. Dada la cantidad de usuarios, es un mercado masivo tan joven como los Smartphones.
- **Consolas**, plataformas cerradas diseñadas específicamente para reproducir juegos, normalmente pocas empresas son lo suficientemente grandes como para diseñar una plataforma cerrada y atraer una gran cantidad de esfuerzo en desarrollo en ofrecer un catálogo de juegos variado. Actualmente, Sony, Microsoft y Nintendo son las principales empresas que controlan el mercado de las consolas.
- **PC**, el ordenador personal lleva más tiempo que los Smartphones asentado en las casas de cada persona, uno de sus innumerables beneficios es la capacidad de reproducir juegos, siempre limitados a la potencia del propio PC.

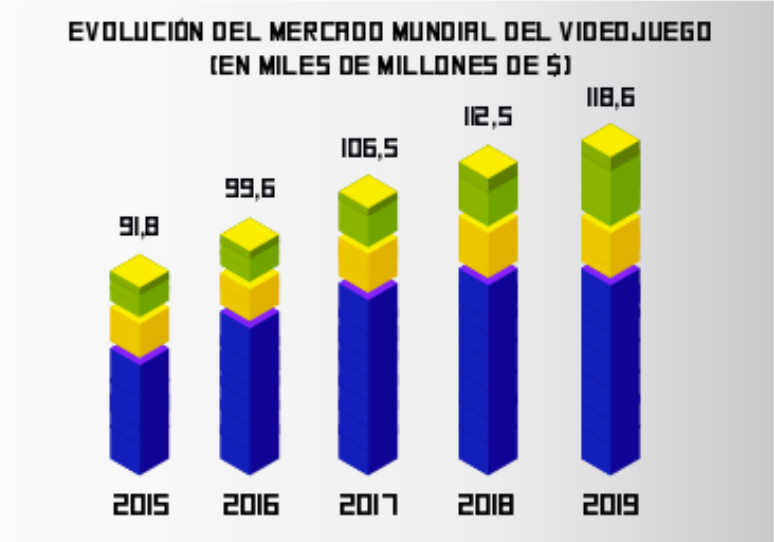

Figura 3.1: Gráfica de Newzoo publicada en el Libro Blanco de desarrollo.

En función de las regiones del mundo podemos encontrar importantes diferencias entre los consumidores de juegos, por ejemplo en países como Norteamérica la disposición es homogénea, sin embargo en otras regiones como Europa y países asiáticos se hace evidente que la mayor parte de los jugadores juegan en plataformas móviles y en PC (ver Figura 3.2).

Aunque sabemos que el PC y el móvil son las plataformas que más jugadores tienen, esto no quiere decir que sean los que más ingresos generan. Las plataformas cerradas con pocos usuarios como las consolas generan una cantidad de ingresos medios por usuario muchísimo mayor que las plataformas como el PC o los móviles, que a pesar de generar menos ingresos, estos cuentan con un mercado mucho más extenso.

Asia lidera el mercado con una previsión de 46.600 millones de dólares, esto representa un 47 % del total de los ingresos globales, seguido de Norte América y Europa. Los paises que con diferencia mueven más dinero son China y Estados unidos con una amplia diferencia respecto a otros paises.

Los videojuegos reprentan la principal fuente de ingresos en los desarrollos móviles. En 2015 los juegos generaron unos ingresos de 34.800 miles de millones de dólares cuando las aplicaciones de móvil generarían 6.300 miles de millones de dólares, es decir, los juegos representarían un 85 % del gasto lobal a pesar de que fueron menos descargados que las aplicaciones. Se prevee para 2020 un enorme crecimiento en ambos sectores (ver Figura 3.3).

Como se puede observar, los videojuegos ya no son un juguete que divierten a niños y adolescentes, es una industria que ha crecido y madurado junto con su público, y no para de crecer debido a los esfuerzos de la industria en ser accesibles a todo el mundo. La in-

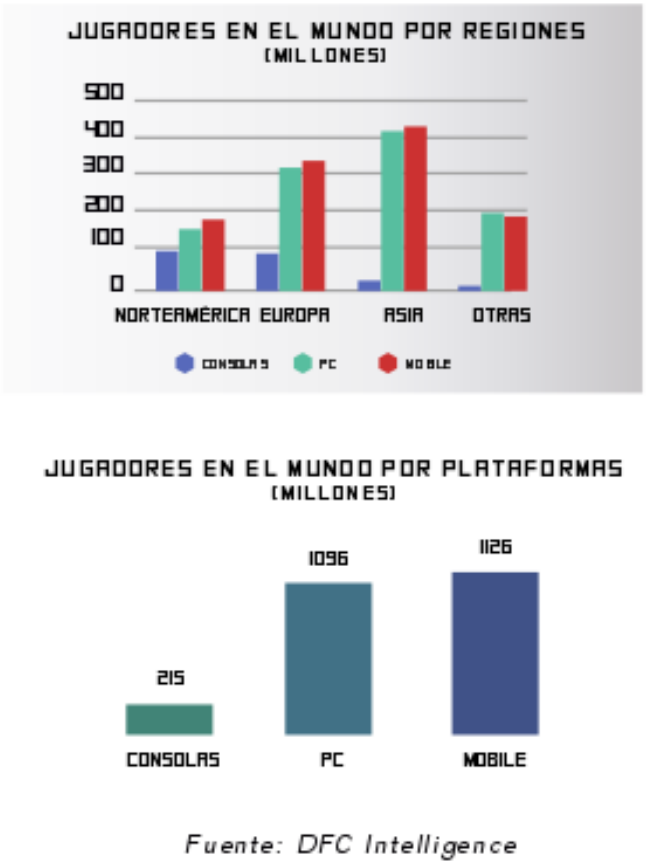

Figura 3.2: Gráfica de DFC Intelligence publicada en el Libro Blanco de desarrollo.

dustria del videojuego hoy en día tiene una enorme importancia, y aún se estima un muy importante crecimiento en los últimos años, por lo tanto se demandarán una gran cantidad de profesionales especializados en esta área.

3.2 Motores de Juegos

El motor de juego es la base de todo desarrollo de un videojuego. Su existencia se debe a que hay unas determinadas funcionalidades comunes a todos los videojuegos como es la renderización de imágenes, la gestión de la entrada y salida, el motor de físicas o la gestión del sonido entre otras cosas, por lo tanto trabajar en esa base es muy rentable, ya que todos tus desarrollos incorporarán esa misma arquitectura, y una optimización de cualquier tipo en el motor de juego repercutirá en todos tus posteriores desarrollos de manera positiva. Este motor nos servirá de base para la construcción de nuestro juego, facilitando e integrando distintos módulos necesarios para nuestro juego.

Cada motor de juego presenta sus peculiaridades, las cuales normalmente definirán nuestra forma de trabajar con el videojuego, por lo tanto es importante tener un conocimiento básico del mismo.

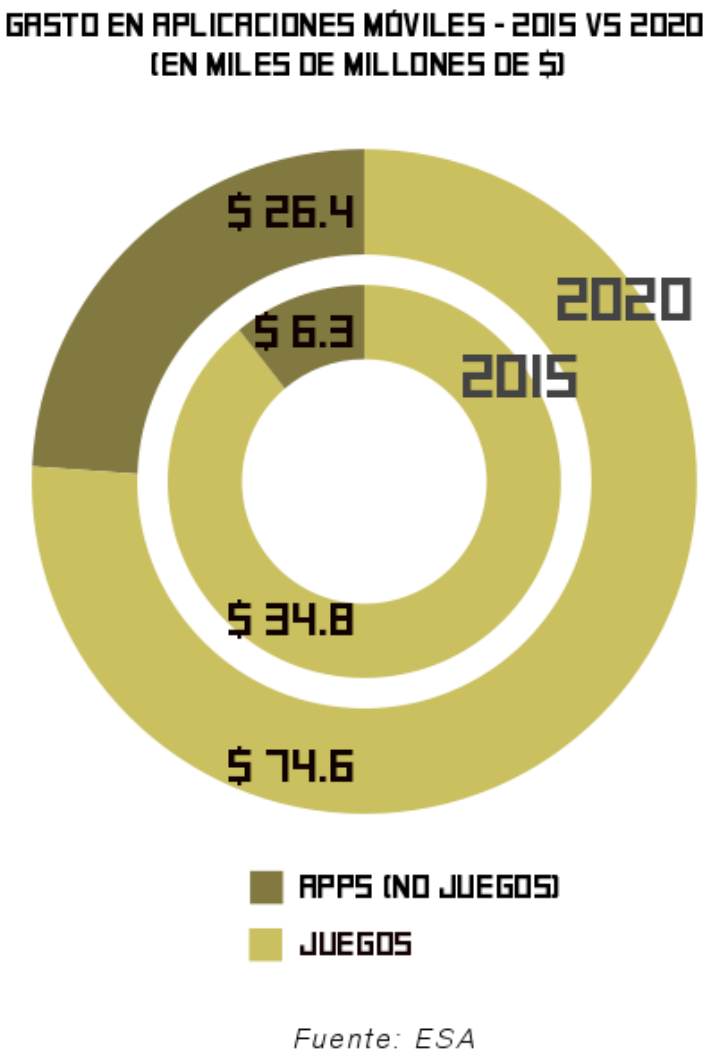

Figura 3.3: Gráfica de ESA publicada en el Libro Blanco de desarrollo.

La opción más común para desarrolladores independientes y estudios pequeños es partir de un motor de juego disponible para el público mediante alguna licencia o de forma gratuita según el caso. Los estudios de videojuegos más grandes en ocasiones desarrollan y mantienen su propio motor de juego como por ejemplo **Frostbite** del estudio DICE, reconocido internacionalmente por su apartado gráfico (ver Figura 3.4), el cual siempre pueden adaptar dependiendo de los desarrollos que tenga en mente el estudio a largo plazo.

El desarrollo de un nuevo motor gráfico desentraña una gran complejidad técnica que escapa al alcance de este TFG, por lo tanto haremos un estudio de los motores de juegos disponibles actualmente.

Figura 3.4: Frostbite en Star Wars:Battlefront

3.2.1 Unity

Motor de juego desarrollado por **Unity Technologies** (ver Figura 3.5), es un motor que se ha popularizado con los últimos años debido a su enfoque multiplataforma. Una de las mayores fortalezas de Unity es su capacidad de poder hacer un único desarrollo y que el motor sea capaz de exportar el desarrollo hasta a 21 plataformas, de entre las que se incluyen **Android** y **iOS**.

Unity favorece principalmente el *desarrollo orientado a componentes*. Esto es debido a que todos sus elementos se representan como una **Entidad de juego** las cuales puedes componer de otros componentes como scripts, materiales, entidades tridimensionales, sonidos...

Figura 3.5: Logotipo de Unity

El editor de Unity es bastante potente, proporcionando la posibilidad no solo de editar la propia escena si no también de crear animaciones, personalizar y crear interfaz de usuario

así como incorporar inteligencia artificial o pathfinding a tus entidades.

Este motor también proporciona también poder trabajar en 2 dimensiones, configuración y edición de gráficos, programación de shaders, físicas, implementación de networking, sonido, soporte para realidad virtual...

Otra de las mayores ventajas es la gran comunidad de desarrolladores que respaldan el motor, debido principalmente a su fácil acceso y a sus licencias gratuitas, proporcionando una gran cantidad de información en internet como manuales, libros, tutoriales o videos que ayudan a entender mejor la herramienta.

Una gran cantidad de juegos de géneros muy dispares han sido desarrollados en Unity, algunos de los más conocidos son:

- Fallout Shelter (Estrategia, Móviles)
- Kerbal Space Program (Simulación espacial, PC)
- Pillars of Eternity (RPG, PC, Linux, OS X)
- Firewatch (Aventura en primera persona, PS4, Xbox One, Linux, PC, OS X)
- Pokémon GO (Realidad aumentada basada en geolocalización, Móviles)
- Hearthstone (Juego de Cartas coleccionable, PC y Móviles)

Como se puede observar, tanto las plataformas como los géneros pueden variar mucho, además algunos de estos juegos son éxitos que pueden llegar a ingresar millones de dólares cada mes. (TODO alguna referencia de internet que me respalde?)

3.2.2 OGRE3D

OGRE (ver Figura 3.6) es el acrónimo en inglés de *Object-Oriented Graphics Rendering Engine*, Este es un motor de renderizado en 3D de software libre bajo la licencia MIT.

Figura 3.6: Logotipo de OGRE3D

El propósito principal es solucionar la problemática de la renderización en tiempo real de la escena del videojuego. OGRE3D proporciona un gran control sobre el juego debido a que tan solo se centra en la renderización 3D, y por lo tanto eres libre de incorporar o implementar otros módulos según los necesites, por ejemplo, es posible incorporar motores especializados en resolver la problemática de la simulación física como Havok, Bullet o PhysX.

Un mayor control sobre lo que está ocurriendo en el código también implica un mayor

tiempo de desarrollo ya que el desarrollador debe de preouparse por todos los módulos que componen su juego.

Además, contamos con el proyecto **Glue Editor** que proporciona un editor de escenas a OGRE.

OGRE dispone también de un soporte multiplataforma, entre ellos Windows, Linux, OS X, Android e iOS. Además algunos proyectos han sido portados a PS3 y Xbox360.

Algunos de los juegos más famosos construidos por OGRE son la saga Torchlight, Jack Kaene, Venetica y Zero Gear.

3.2.3 Unreal Engine

Unreal Engine (ver Figura 3.7) es un motor creado inicialmente en 1998 por *Epic Games* que ha evolucionado junto con la industria de los videojuegos a lo largu de sus sucesivas versiones.

Figura 3.7: Logotipo de OGRE3D

Inicialmente este fué un motor propio de Epic Games con los que desarrolló su propio juego como la saga **Unreal Tournament** que es a la que da nombre el motor.

A lo largo del tiempo otras compañías han hecho uso del motor Unreal mientras que el estudio se ocupaba de mantener y proporcionar soporte.

Está programado en C++, proporciona un editor de escenas completo. La programación del videojuego se puede realizar tanto en C++ como en una modalidad de programación visual conocida como **Blueprints**.

Actualmente se encuentra en la versión 4, con diversas licencias para los desarrolladores, los cuales pagan en función de los beneficios. Permite la exportación a numerosas plataformas como Xbox One, PlayStation 4, Android, iOS así como los principales sistemas operativos de PC.

Algunos de los juegos más conocidos desarrollados para este motor son Deus Ex (2000),

Bioshock (2007), Mass Effect (2007), Daylight (2014), así como otros títulos aún en desarrollo como Kingdom Hearts III.

3.3 Otros Desarrollos TODO otro titulo

En la presente sección se realizará un breve estudio de otros proyectos similares al desarrollo propuesto por este TFG.

3.3.1 Subway Surfers

Juego móvil desarrollado por el estudio Kiloo en 2012 es uno de los mayores éxitos del género Runner en plataformas móviles.

El juego consiste en avanzar a través de 3 calles sin golpearse con ningún obstáculo en nuestro camino, pudiendo recoger por el camino monedas que nos servirán más tarde para comprar potenciadores.

El juego ganó popularidad gracias a la gestión de records, en los que jugadores competían de manera semanal por obtener las mejores puntuaciones, obteniendo premios dentro del juego.

Subway Surfers fue desarrollado en Unity y se estima, gracias a sus técnicas de monetización que actualmente generan una cantidad de 33,613$ sólo en plataformas iOS según la página thinkgaming.com (TODO referencias a páginas?)

3.3.2 Fallout Shelter

Juego móvil desarrollado por Bethesda publicado en 2015 de género estratégico para juegos móviles. Su planteamiento está basado en la gestión de recursos, así como la gestión y la inversión de tiempos.

Figura 3.8: Pantalla de juego de Fallout Shelter

La dinámica principal de Fallout Shelter es la gestión de los recursos, incluyendo entre los mismos la gestión del tiempo del propio jugador, dependiendo de su previsión sobre cuando va a volver a jugar a abrir dicha aplicación, premiando al jugador por hacer ciclos cortos de espera en el juego y proporcionando una menor recompensa si se hacen ciclos más largos. Mantener de manera prolongada en el tiempo la atención del jugador es una estrategia básica para un juego de móvil, y para ello se recurren a determinados diseños en el juego como el mecionado anteriormente para captar y mantener el interés del jugador.

Capítulo 4

Metodología

En este capítulo se presentará la metodología aprendida y aplicada en el presente TFG así como las herramientas Software y Hardware.

4.1 Metodología de Desarrollo

La metodología que desarrollamos a lo largo de este proyecto está basada en una metodología ágil llamada **eXtreme programming**. Esta es una metodología ágil, la cual se adapta a nuevas necesidades, en lugar de preveer los cambios.

Esto es una propiedad fundamental a la hora del desarrollo de un videojuego, ya que ante la figura difusa de un cliente con una idea de producto clara, tenemos por un lado los distribuidores, los roles administrativos, los propios desarrolladores y por último los diseñadores quienes tienden a tomar más decisiones, quienes dan pie a esas necesidades y esto genera grandes cambios en el producto.

Es de fundamental importancia la generación de diversos prototipos en la fase de incepción de un juego, ya que es necesario comprobar si aquello que se ha pensado en un primer momento tiene futuro y puede divertir al jugador o si por el contrario, se detectan problemas y por lo tanto se toman medidas en consecuencia a esos problemas.

Al realizar iteraciones de desarrollo cortas, se prueba y se comprueba si aquello que se ha desarrollado es lo que realmente se necesita, y tener una versión prematura del producto permite poder **reaccionar** ante el.

4.1.1 Extreme Programming

Esta metodología tiene como base los principios del manifiesto de desarrollo ágil de Software, los cuales serán las lineas maestras de nuestra metodología, pasamos a comentar brevemente cada uno de los principios.

- Individuos e interacciones sobre procesos y herramientas. Este principio recalca la importancia de que ante todo están las personas y comunicarnos con ellas antes que darle más importancia a las herramientas como puede ser comunicarle a un compañero de trabajo que debe de realizar una tarea y transmitirlo personalmente en lugar de tan solo registrarlo en la herramienta de gestión de proyectos.

- Software funcionando sobre documentación extensiva. La prioridad ante todo es que nuestro producto funcione, priorizando sobre realizar unos extensos documentos sobre el diseño.
- Colaboración con el cliente sobre negociación contractual. En lugar de ver al cliente un día, firmar en un contrato y reunirse con el cliente un año después con el producto ya acabado, tiene más valor realizar reuniones con dicho cliente, de manera que tenga la capacidad de que si algo en su software no le gusta, y por lo tanto sea necesario un cambio en los requisitos, tenga la oportunidad de poder modificar su producto para que el resultado final se adapte a sus necesidades, en lugar de darse cuenta una vez todo el desarrollo ya está acabado.
- Respuesta ante el cambio sobre seguir un plan. En la metodología ágil, uno debe estar siempre preparado ante los cambios en los requisitos en su software que con mucha probabilidad sucederán

Esto plantea una forma diferente de ver el desarrollo software en el que se tienen en cuenta los problemas que se han visto en el desarrollo software tradicional.

Esta disciplina procura minimizar el coste de cambios en los requisitos teniendo múltiples ciclos de desarrollo cortos. A demás incluye principios y prácticas dentro del marco de desarrollo.

Las actividades se dividen principalmente en:

- Programación, proceso verdaderamente importante por el cual obtenemos el producto funcional.
- Testing, por el cual podemos asegurar funcionalidad básica en nuestro código mediante test unitario. A un nivel más general, los Test de aceptación verifican un requisito del cliente.
- Escuchar, los programadores deben escuchar y entender a los clientes aquello que es necesario implementar.
- Diseñar, se puede programar hasta cierto punto sin que el desarrollo se vuelva demasiado complejo en si.

Además se establecen varias normas en este contexto, como por ejemplo:

- El stakeholder está siempre disponible
- Se programa primero el test unitario
- Se escriben las historias de usuario
- Se realizan pequeñas releases
- Todo el código debe tener test unitarios

4.1.2 Diferencias

En este proyecto partiremos de una base formada a partir de elementos de Extreme Programming con el que formaremos nuestra propia metodología.

En el desarrollo, antes de utilizar cualquier norma, prima ante todo el sentido común, por ello es necesario revisar la metodología cuando se va a aplicar a un desarrollo donde el equipo de programadores como el conjunto de stakeholders están formados por la misma persona.

Los test son una herramienta básica que descubren los estados de regresión, que son aquellos elementos en el código que antes funcionaban y ante un determinado cambio han dejado de hacerlo. Esto permite detectar fallos ante un cambio de entorno (se despliega la aplicación en un entorno completamente nuevo) un cambio en la tecnología (una nueva librería) o cualquier otra situación adversa. Los test de aceptación tienen también relevancia y se utilizan a menudo en empresas grandes, en el que mediante un proceso de integración continua se comprueba que un programador no ha hecho un cambio en el código que podría ser completamente nocivo en el juego, como una bajada de rendimiento importante o directamente el fallo de la aplicación.

Dado el poco alcance del presente proyecto, no se considera importante la cobertura de Test durante el ciclo de vida del actual proyecto, sin embargo se considera relevante para trabajos futuros (TODO puedo ser tan contundente respecto a test? hago unos test unitarios para contentar? porque el valor que me aportan para este proyecto es 0)

El stakeholder siempre estará disponible ya que el interesado es el propio programador, de manera que no tendrá dudas acerca del producto que se desea desarrollar.

4.2 Recursos

A continuación se listarán aquellos bienes que han sido utilizados durante el desarrollo de este TFG

4.2.1 Hardware

Se ha utilizado un portatil, un dispositivo Android para testar y comprobar el rendimiento en móviles y una pantalla adicional para extender el entorno de trabajo.

4.2.2 Software

Unity

Para el desarrollo del propio juego, con su correspondiente editor de niveles. Además de integrar todo aquello

Monodevelop

Entorno de desarrollo integrado de Unity para la edición de scripts en C#. Se encuentra

configurado con funciones "*auto-complete*"de las librerías de Unity. Con el objetivo de ahorrar tiempo en configuraciones con otros IDEs, Monodevelop es una buena opción para la edición de scripts.

Maven

Para el desarrollo de la aplicación servidora, será necesario Maven, la cual es una herramienta software para la gestión y construcción de proyectos Java con un modelo de configuración basado en XML. La cómoda gestión de dependencias y su integración con otros programas software de desarrollo lo convierten en una opción imprescindible a la hora de realizar un desarrollo Java.

Eclipse

Entorno de desarrollo integrado para proyectos Java. Su integración con Maven así como su fácil uso lo convierte en uno de las mejores opciones a la hora de trabajar con Java.

Spring

Framework para el desarrollo de aplicaciones y contenedor de inversión de dependencias. Además de proporcionar inyección de dependencias, ofrece otros muchos aspectos como la autenticación, programación orientado a aspectos o servicios de comunicación.

Se utilizará principalmente para el desarrollo de Web Services RESTful en la aplicación servidora, con el objetivo de poder hacer una llamada web a través de un DNS.

Tomcat 7

Contenedor de Servlets Java open-source desarrollado por Apache Software Foundation. Tomcat implementa varias especificaciones de Java EE.

Figura 4.1: Logo de Apache Tomcat

Se utilizará en el proyecto como infraestructura para realizar el despliegue de la aplicación servidora. Debido a su fácil accesibilidad así como una demostrada potencia, Apache Tomcat es una buena opción a la hora de realizar dicho despliegue.

Lenguajes de Programación

C#

Lenguaje de propósito general a alto nivel desarrollado por Microsoft para su plataforma

de .NET, utilizado en Unity como herramienta para scripting.

C# es el lenguaje principal con el que se construirá la lógica de juego.

Java

Lenguaje orientado a objetos compilado en Bytecode con la característica de poder ejecutarse en cualquier plataforma que soporte Java.

Java es el principal lenguaje con el que se construirá la aplicación servidora.

LaTeX

LaTeX es un sistema de composición de textos orientados a una gran calidad tipográfica, facilitando al usuario poder crear documentos de gran calidad delegando los problemas de estilo en dicho Software.

LaTeX se utilizará para la generación de documentación del proyecto.

Capítulo 5

Arquitectura

En este capítulo se presentará la arquitectura aplicada así como los distintos esfuerzos teniendo en cuenta el diseño del software.

5.1 Arquitectura del Sistema

A continuación describiremos varios componentes clave en la arquitectura del software desarrollado en el presente trabajo.

5.1.1 Arquitectura orientada a Componentes

Por imposición de la tecnología, el juego debe de ser desarrollado bajo el paradigma de arquitectura orientada a componentes. Esto permite desarrollar perqueñas piezas de funcionalidad, con su correspondiente modularidad de manera que obtenemos un código mantenible y reutilizable.

El esquema de Unity es la de construir la lógica de juego a partir de pequeños scripts individuales que se asocian a una o varias entidades de juego definidas previamente por el diseñador de niveles.

Cada uno de estos scripts están compuestos principalmente por 2 métodos, un método TODO Start, el cual es ejecutado al inicio, previo al primer frame con el cometido de inicializar las variables de manera que el script no empiece a funcionar en un estado inconsistente.

El segundo método se trata de Update el cual será llamado en cada frame. Este método por lo general comprueba el estado del juego y actúa en base a el mismo, pudiendo no hacer nada ser una actuación válida.

Un caso práctico es un el script de un enemigo. Los enemigos se inicializan con una vida base y comprueban constantemente si entran en colisión con un objeto del juego por el que no pueden cruzar, en cuyo caso son eliminados del juego, o si por el contrario son disparados por el jugador en cuyo caso restarían vidas a su total máximo de vidas.

En el desarrollo tradicional de videojuegos, esta lógica podría estar codificada en un bucle infinito en el que se agruparía toda la lógica del juego, y en una parte de dicho bucle se haría un recorrido linea de cada enemigo al cual se le aplicaría esta lógica de negocio, sin embargo,

la encontramos distribuida en cada enemigo con su propio script, esto añade mantenibilidad, ya que es más fácil identificar aquella parte del código correspondiente al enemigo tanto por separarlo en un script como por evitar el recorrido lineal de enemigos que ya hace por nosotros automáticamente el motor de juego.

Además, esta lógica de juego está encapsulada en dos scripts llamados .Enemyz "Killable"de manera que si esta lógica es necesaria en otra parte del juego, o necesitamos reutilizar dicha lógica en posteriores desarrollos, podremos reutilizar el código fácilmente.

Esta es la arquitectura orientada a componentes con el que podemos llegar a obtener unas clases débilmente acopladas entre sí.

A continuación se describirán los componentes que componen la lógica del juego. Cada componente está asociado a una entidad de juego, y esta pieza de código puede estar replicada en diferentes entidades, como por ejemplo, el script Enemy está replicado en cada uno de los enemigos, o por el contrario, puede ser un script genérico de control del juego como el creador de escenarios.

Lógica del jugador

El movimiento del jugador se encuentra principalmente encapsulado en el script de TODO EMPH Character Movement, este script tan solo lo posee el jugador, y controla varios aspectos del juego.

Desde este script se controla el movimiento del jugador, comprobando diferentes entradas por parte del jugador y respondiendo ante ellas. El movimiento del Juego es muy restrictivo, solo se permite el movimiento entre 3 calles delimitadas.

Además del movimiento, se hace un control de las animaciones del jugador. Unity tiene un motor de animación para facilitar esta labor a los desarrolladores de videojuegos llamado Animator (ver Figura 5.1). En esta parte del motor podemos montar fácilmente las transiciones entre las animaciones.

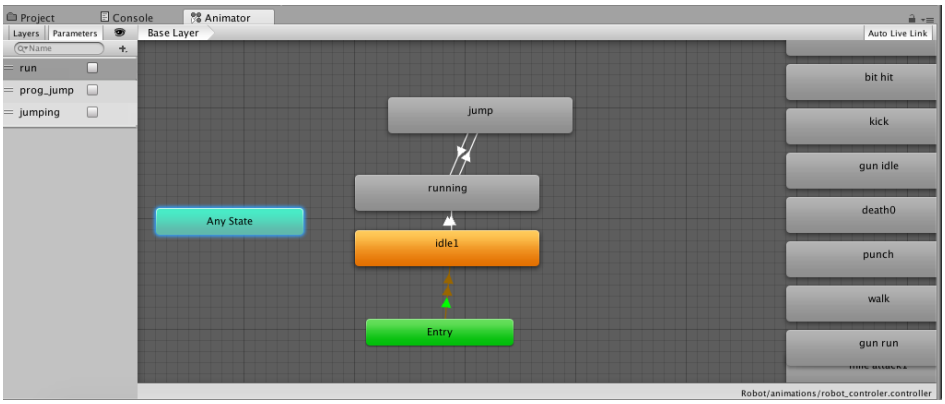

Figura 5.1: Animator de Unity

Para ello nos podemos ayudar de variables definidas en el propio Animator, por ejemplo, permitimos la transición de caminar a la animación de salto cuando la condición booleana "jumping.ésté activa. Desde código, activamos esas variables, permitiendo al personaje saltar cuando realmente está saltando. Esta máquina finita de estados es una buena abstracción para plantear las animaciones del personaje.

Desde el script, podemos editar dichas variables (ver código en 5.1) para que nuestro personaje realice la animación adecuada. También se pueden configurar otro tipo de variables como enteros o números de punto flotante y utilizarlos en la máquina finita de estados con el objetivo de proporcionar una animación más fiel. Normalmente esta parte requiere una labor importante de coordinación entre la dirección artística y el equipo de desarrollo del videojuego.

```
if(animJump){
        animatorComponent.SetBool("delayedJumping", animJump);
        animatorComponent.SetBool("prog_jump", true);
        StartCoroutine(stopdelayedJumping());
}
```

Listado 5.1: Modificación de variables del Animator

Sin embargo, la principal responsabilidad de este script es realizar el movimiento del personaje. Para ello se ha parametrizado la mayoría de las variables de manera que tenemos siempre el control de cual es la abertura de las calles, cómo de rápido pasa de una calle a otra...

Cuando el jugador aplica la entrada necesaria para el cambio de calle, entonces el script responde a ello en función de en que calle se encuentra originalmente. Mientras el jugador no quede por encima o por debajo de una determinada coordenada X, se aplicará una traslación hacia el sentido en el que se está desplazando y el script esperará a ser llamado en el siguiente frame, de manera que al final queda animada de forma programática el desplazamiento hacia los laterales. Siempre debemos tener en cuenta hacia donde nos movemos y de donde partimos para ejecutar la animación de manera correcta, utilizamos una variable de tipo enumerado para recordar qué calle es la actual del robot. En 5.2 podemos ver un fragmento del script en el que se puede observar el ciclo completo desde que el jugador decide ir a la izquierda hasta que el robot para de desplazarse hasta la izquierda. Como se puede observar en el código, no es tan sencillo como aplicar solo una transformación de las coordenadas del jugador, si no que hay que tener en cuenta todos los posibles casos en los que el jugador ha podido decidir desplazarse a la izquierda, tanto desde cualquier posición, como si el jugador se estaba desplazando hacia la derecha en ese momento.

```
        //Side movement
        if (Input.GetKey (KeyCode.LeftArrow) && !blockInput && !sidePenalizationMovement ||
```

```
    movingLeft){
        //Play Animation Conditions
        if(!jumping && !movingLeft && Input.GetKey (KeyCode.LeftArrow) && enumPosition !=
            enPosition.Left)
                animatorComponent.SetBool("movL", true);
        movingLeft = true;
        if (Input.GetKey (KeyCode.LeftArrow)){
                movingRight = false;
        }
        moveLeft ();
        playSound("blip");
}

void moveLeft(){
        switch(enumPosition){
        case enPosition.Right:
                if(gameObject.transform.position.x < firstPosition.x){
                        movingLeft=false;
                        enumPosition = enPosition.Center;
                        sidePenalizationMovement = false;

                }else{
                        translationLeft();
                }
                break;

        case enPosition.Center:
                if(gameObject.transform.position.x < firstPosition.x - horizontalSlope){
                        movingLeft=false;
                        sidePenalizationMovement = false;

                }else{
                        enumPosition = enPosition.Left;
                        translationLeft();
                }
                break;

        default:
                if(gameObject.transform.position.x < firstPosition.x - horizontalSlope){
                        movingLeft=false;
                        enumPosition = enPosition.Left;
                        sidePenalizationMovement = false;

                }else{
                        translationLeft();
                }
                break;
```

```
        }
}

void translationLeft(){
        Vector3 translation = new Vector3 (-1, 0, 0) * horizontalSpeed * Time.deltaTime;
        gameObject.transform.Translate (translation);
        transform.Find("/TestCharacter/PlayerCamera").SendMessage("doTranslate", -
            translation);
}
```

Listado 5.2: Desplazamiento hacia la izquierda

Desde este código se controla también si el jugador entra en contacto con un elemento el cual debe esquivar, como por ejemplo un barril o un enemigo y si es así, se aplica una penalización y se mueve al jugador de sitio en el caso de que el mismo tenga más de 1 vida. Para programar esta lógica nos ayudamos de los métodos de retrollamada de Unity. Uno de los elementos más usados en los videojuegos la comprobación de colisiones contra otros objetos. Una colisión es el momento en el que una entidad definida por un espacio, en nuestro caso, por un cubo de 8 puntos definido en el espacio, entra en colisión con otra entidad de juego. Es muy frecuente que este impacto en el juego tenga como resultado un movimiento de los dos entes, por ejemplo un jugador golpeando un barril podría desembocar en el barril siendo desplazado o tumbado por el jugador. Sin embargo otro uso muy frecuente es el de disparadores o "triggers". En nuestro juego, el motor de físicas no se utiliza ya que no nos conviene, prácticamente todos los movimientos son desencadenados a raíz de nuestro script. Cuando una entidad colisiona con otra entidad, se hace una llamada al método .ºnTriggerEnter", desde este método disponemos de toda la información del objeto con el que ha colisionado y podemos reaccionar en consonancia, en nuestro caso, comprobamos si el trigger tiene un nombre clave, de manera que comprobamos si el jugador ha colisionado con un enemigo o contra un obstáculo y si es así, aplicamos una penalización al jugador y corregimos su posición. En el fragmento de código 5.3 se puede observar la lógica relativa a la comprobación de colisiones respecto a otras entidades de juego.

```
// ======== Unity Callbacks ========
void OnTriggerEnter(Collider other) {
        if(!crouched && other.name.Equals("crouch"))
                applyPenalization();
        else if(other.name.Equals("wall") || other.name.Equals ("wall_rock") ||  other.name.
            Contains ("Enemy")){
                applyPenalization();

                //RayCast Forward
                bool center=true, left=true, right=true;
                RaycastHit hit;
```

```
Vector3 positionCenter = new Vector3(firstPosition.x, transform.position.y
    +2.4f, transform.position.z-1);
Ray forwardRay = new Ray(positionCenter, Vector3.forward);
if(Physics.Raycast(forwardRay, out hit) && hit.distance < 4){
        center=false;
}
//Raycast Left
Vector3 positionLeft = new Vector3(firstPosition.x - horizontalSlope,
    transform.position.y+2.4f, transform.position.z-1);
forwardRay = new Ray(positionLeft, Vector3.forward);
if(Physics.Raycast(forwardRay, out hit) && hit.distance < 4){
        left=false;
}
//Raycast Right
Vector3 positionRight = new Vector3(firstPosition.x + horizontalSlope,
    transform.position.y+2.4f, transform.position.z-1);
forwardRay = new Ray(positionRight, Vector3.forward);
if(Physics.Raycast(forwardRay, out hit) && hit.distance < 4){
        right=false;
}
sidePenalizationMovement = true;
if(enumPosition == enPosition.Center){
        if(right){
                movingRight = true;
                movingLeft = false;
        }else if (left){
                movingLeft = true;
                movingRight=false;
        }
}else if(enumPosition == enPosition.Right){
        if(center){
                movingLeft = true;
                movingRight = false;
        }else if (left){
                movingLeft = true;
                movingRight = false;
                forceTwoLeft = true;
        }

}else if(enumPosition == enPosition.Left){
        if(center){
                movingRight = true;
                movingLeft = false;
        }else if (right){
                movingRight = true;
                movingLeft = false;
                forceTwoRight = true;
```

```
                        }

                }

                if(other.name.Equals("Enemy")){
                        Destroy (other.transform.gameObject);
                }


        }

}


void OnCollisionEnter(Collision other){
        if(other.gameObject.name.Equals ("jump") &&
           other.gameObject.GetComponent<Collider>().bounds.max.y-0.1f > transform.
                GetComponent<Collider>().bounds.min.y){
                heightToReach = other.gameObject.GetComponent<Collider>().bounds.max.y +0.1f
                     ;
                applyPenalization();
        }
}

void applyPenalization(){
        Debug.Log ( "Lifes: " + lifes);
        if(!invencible){
                timerIdle=0;
                invencible = true;
                lifes = lifes-1;
                if(lifes == 0){
                        originalSpeed = speedForward;
                        speedForward = 0;
                        movingLeft=false;
                        movingRight=false;
                        forceTwoLeft=false;
                        forceTwoRight=false;
        StartCoroutine(saveRecord());
                        enemyDisappears();
                }else{
                        StartCoroutine(stopInvencible());
                }
        }
}
```

Listado 5.3: Comprobación de penalizaciones

Generación procedural de escenarios

En el juego a desarrollar, el nivel es infinito, el objetivo es sobrevivir el mayor número de segundos posible, de manera que no importa lo largo que hagamos el nivel, siempre habrá un final. Otro problema que nos encontramos si realizamos un único gran nivel es que empezaremos a memorizarlo y a conocerlo, esto puede tener repercusiones negativas en el diseño del juego ya que el jugador se puede aburrir fácilmente si conoce todos los obstáculos durante los primeros 20 segundos de juego. Para solucionar estos problemas, se ha creado un pequeño motor de generación de niveles procedural. Este motor tiene en cuenta la posición del jugador y en función de donde se encuentra, construye el nivel o destruye piezas del escenario que el jugador ya ha dejado atrás para optimizar recursos. El nivel se construye en base a un conjunto de piezas definido por el desarrollador del juego, en el Script se puede modificar el tamaño de ese conjunto de piezas (ver Figura 5.2) permitiendo construir el de manera dinámica con más o menos piezas, resultando en una mayor o menor variedad para el jugador.

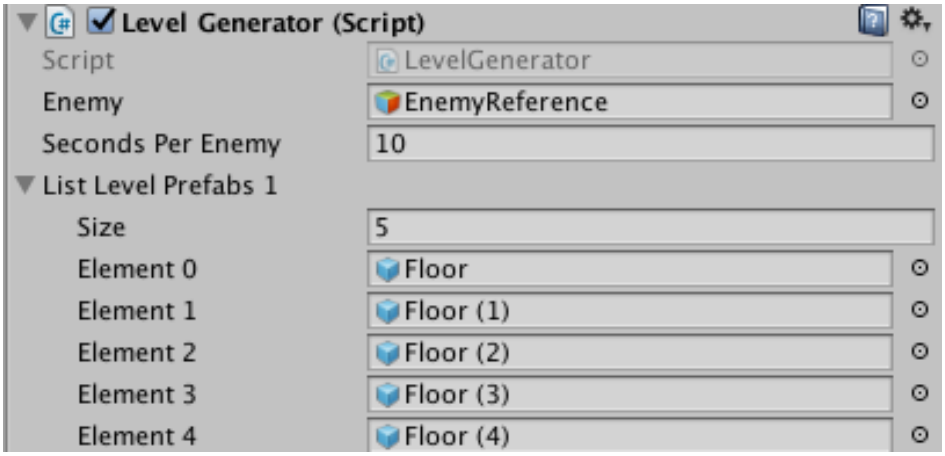

Figura 5.2: Script del generador en el editor de niveles

Este script se encarga también de crear enemigos de manera aleatoria por el nivel para complicar la partida al jugador. El script se configura para que dado un número arbitrario de segundos se genere un nuevo enemigo en alguna de las 3 calles disponibles. Debido a la aleatoriedad, es posible que suponga un problema para el jugador o no ya que se puede generar en una de las calles con obstáculos y si el enemigo colisiona con un obstáculo, este desaparecerá automáticamente. A continuación se puede ver en la Figura 5.4 la lógica elemental para la generación procedural de niveles.

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

        //Update Timers
        enemyTimer += Time.deltaTime;
```

```
        enemySpawn();

        if(scenarioSteps > listScenario.Count){
                addNewPiece();
        }

        float zPlayer = player.transform.position.z;
        float zScenario = ((BoxCollider)actualScenario.GetComponent<BoxCollider>()).bounds.
            max.z;
        if(zPlayer > (zScenario -10)){
                addNewPiece ();
                actualScenario = listScenario[listScenario.IndexOf(actualScenario)+1];
        }



        //Check to delete the first one
        zScenario = ((BoxCollider)listScenario[0].GetComponent<BoxCollider>()).bounds.max.z;
        if(zPlayer > (zScenario + 10)){
                removeFirstPiece();
        }

}

void addNewPiece(){
        GameObject newPiece = (GameObject)Instantiate(getNewPiece(), Vector3.zero,
            Quaternion.identity);
        Bounds objectBounds =  ((BoxCollider)newPiece.GetComponent<BoxCollider>()).bounds;
        float zSize = objectBounds.max.z - objectBounds.min.z;
        Vector3 pos = listScenario[listScenario.Count-1].transform.position;
        Vector3 finalPos =  new Vector3(pos.x, pos.y, pos.z + zSize);
        newPiece.transform.position = finalPos;
        listScenario.Add(newPiece);
}

void removeFirstPiece(){
        GameObject.Destroy(listScenario[0]);
        listScenario.RemoveAt (0);
}

GameObject getNewPiece(){
        int random = Random.Range (1, listLevelPrefabs1.Count);
        return listLevelPrefabs1[random];
}
```

Listado 5.4: Generación procedural de niveles

Se puede observar en el código referenciado que se tiene en cuenta la anchura de la última

pieza colocada para solapar la nueva pieza con exactitud al final de la última, de manera que todos los niveles quedan solapados de manera uniforme.

Componentes de los enemigos

La lógica de cada enemigo está formada por 2 componentes, Enemy y Killable. La lógica de Enemy controla el movimiento de la entidad del juego, siempre en dirección opuesta al jugador, controla la altura a la que está del suelo y detecta si colisiona con una pared, en cuyo caso, muere. El componente Killable sin embargo controla que la entidad de juego se pueda matar. Para ello controla si el jugador está pulsando sobre la entidad de juego, si es así, recibe daño. Llegado al punto en el que la vida es igual o menor que 0, se activa el sonido de muerte y la animación de muerte programática, reduciendo la escala de la entidad de juego progresivamente hasta eliminarlo.

Propiedades del jugador

Una de las funcionalidades del juego es ir mejorando progresivamente a tu personaje. Es necesario mantener guardados los datos del progreso del jugador, para ello nos tenemos que apoyar en el sistema de guardado de Unity. El motor ofrece una API que se encarga de persistir y recuperar datos de manera que la plataforma en la que se persistan dichos datos es transparente para el programador. Es ideal para guardar y recuperar pocos datos ya que se utiliza una estructura de diccionario por la cual mediante una clave, en este caso una cadena de caracteres, podemos recuperar la variable que deseemos.

Para este desarrollo necesitamos guardar 2 tipos de datos, uno es el nivel de una habilidad determinada, como por ejemplo, el daño que hace el personaje a los enemigos. Otro dato guardado es una fecha. Para guardar la fecha es necesario guardarla en formato String ya que las variables Integer y Long son demasiado pequeñas. Cuando el jugador decide mejorar una habilidad, se registra la fecha exacta en la que se solicitó la mejora de manera que independientemente de si la aplicación está abierta o cerrada, el juego sabrá en todo momento si ha pasado el suficiente tiempo como para otorgar la mejora.

5.1.2 Arquitectura de niveles

En el desarrollo de cualquier juego, una parte importante del desarrollo está compuesta únicamente por la composición de niveles, los cuales el jugador deberá superar para finalizar el reto que se le plantea. Esto es un trabajo que debe realizar normalmente el diseñador de niveles, y el planteamiento y la construcción de estos escenarios debe de estar acordado y pactado con la sección de desarrollo.

El diseño de niveles de este desarrollo se plantea de una forma totalmente modular, el diseñador de niveles puede crear tantas piezas de nivel como crea oportuno, ya que el motor procedural es el encargado de ir montando las piezas conforme el juego avance. Para expresar

por qué partes un jugador puede pasar y qué otras partes se considera un obstáculo, se utilizan etiquetas en las propias entidades de juego. De esta manera el diseñador de niveles solo tiene que preocuparse de poner la etiqueta correspondiente y el programador la recuperará en código. Como se aprecia en la Figura 5.3, la composición de niveles se puede realizar cómodamente desde el mismo motor, facilitando mucho la labor del diseño de niveles.

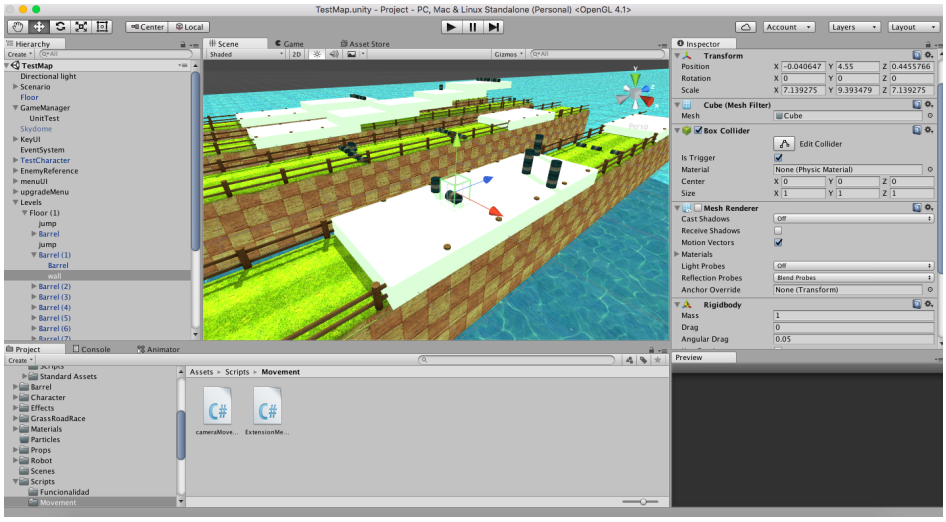

Figura 5.3: Diseño de niveles en Unity

5.1.3 Arquitectura servidor - cliente

Dado que la gestión de Records es un servicio que debe de ser compartido por todos los clientes de la aplicación, se ha planteado el modelo de arquitectura cliente-servidor para la implementación de la gestión de Records, de manera que todos los clientes conocen al servidor y el mismo les provee del servicio de la consulta y el guardado de records.

Servidor

Para la implementación del Servidor se ha utilizado el framework Spring para incorporar fácilmente una modelo TODO acrónimo REST. La decisión del uso de REST es una mayor comodidad para la implementación de la aplicación.

Para la implementación del servidor, se ha dividido la lógica en 3 capas.

- **Endpoint**. Encargado de recibir la orden REST, es el punto de conexión del servidor con el propio juego. Spring delega las peticiones GET en el método que designemos mediante anotaciones 5.5. Tras obtener una respuesta, la devuelve como un método normal y Spring se encarga de devolver dicha respuesta a el cliente.
- **Controller**. Incorpora toda la lógica del servidor, en nuestro caso, recupera los mejores records o los guarda. Realiza una gestión de persistencia a nivel de fichero dada la simplicidad de los datos.

- **Entidad**. La entidad de negocio, en nuestro caso Record, compuesta por un identificador y un indicador de tiempo entero.

```
@Controller
public class RecordEndpoint {

	private RecordManagerBean recordMgt = new RecordManagerBean();



	@RequestMapping(value="/getRecordRequest", method = RequestMethod.GET)
	public Record getRecord(@RequestParam(value="name", defaultValue="1338") String request) {
		Record response = recordMgt.retrieveRecord(request);

		return response;
	}

	@RequestMapping(value="/storeRecord", method = RequestMethod.GET)
	public Record storeRecord(@RequestParam(value="name", defaultValue="1338") String uid,
			@RequestParam(value="seconds", defaultValue="0") String seconds) {
		Record storedResponse = new Record(uid,Integer.parseInt(seconds));
		recordMgt.storeRecord(storedResponse);

		return storedResponse;
	}

	@RequestMapping(value="/getTop", method = RequestMethod.GET)
	public Record[] storeRecord() {
		Record[] response = recordMgt.getTop();

		return response;
	}



}
```

Listado 5.5: Endpoint de conexión al juego

La compilación del proyecto está gestionado por **Maven**, permitiendo descargar y controlar las versiones de aquellas librerías que utilicemos así como aplicar testing automáticamente, establecer propiedades, y otras muchas utilidades.

Marshalling y Unmarshalling

Se le conoce a **Marshalling** como el proceso de transformación de un objeto en la memoria del programa a un formato distinto que represente dicho objeto y permita guardarlo en el disco duro, transferirlo por la red... Para llevar a cabo el proceso de Marshalling en la aplicación servidora, delegamos dicha responsabilidad en Spring 5.6. En Spring, mediante

los descriptores XML decidimos a qué clase se le debe aplicar el Marshalling y de qué tipo, en nuestro caso, utilizaremos el formato JSON.

```
<?xml version="1.0" encoding="UTF-8"?>

        <mvc:annotation-driven>
                <mvc:message-converters>
                        <bean
                                class="org.springframework.http.converter.xml.
                                    MarshallingHttpMessageConverter">
                                <property name="marshaller" ref="xstreamMarshaller" />
                                <property name="unmarshaller" ref="xstreamMarshaller" />
                        </bean>
                </mvc:message-converters>
        </mvc:annotation-driven>

        <!-- Configures the different content types the servlet can handle. -->
        <bean name="viewResolver"
                class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">

                <property name="ignoreAcceptHeader" value="true" />
                <property name="favorParameter" value="true" />
                <property name="favorPathExtension" value="true" />
                <!-- if no content type is specified, return json. -->
                <property name="defaultContentType" value="application/json" />

                <property name="mediaTypes">
                        <map>
                                <entry key="json" value="application/json" />
                                <entry key="xml" value="application/xml" />
                        </map>
                </property>
                <property name="defaultViews">
                        <list>
                                <bean
                                        class="org.springframework.web.servlet.view.json.
                                            MappingJacksonJsonView">
                                        <!-- see http://static.springsource.org/spring/docs/3.1.x/
                                            javadoc-api/org/springframework/web/servlet/view/json/
                                            MappingJacksonJsonView.html#
                                            setExtractValueFromSingleKeyModel(boolean) -->
                                        <property name="extractValueFromSingleKeyModel" value="true"
                                             />
                                </bean>
                                <bean class="org.springframework.web.servlet.view.xml.
                                    MarshallingView">
                                        <constructor-arg ref="xstreamMarshaller" />
```

```
                                <!-- see http://static.springsource.org/spring/docs/3.1.x/
                                     javadoc-api/org/springframework/web/servlet/view/xml/
                                     MarshallingView.html#setModelKey(java.lang.String) -->
                                <property name="modelKey" value="responseObject" />
                        </bean>
                </list>
        </property>
</bean>

<bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
        <property name="autodetectAnnotations" value="false" />
        <property name="aliases">
                <map>
                        <entry key="record"
                                value="journey.server.ws.spring.model.Record" />
                </map>
        </property>
</bean>
</beans>
```

Listado 5.6: Configuración Spring para Marshalling

El proceso de **Unmarshalling** es justamente el contrario, convertimos algo en un formato pre-establecido en un objeto de memoria. En este caso, recibiremos una respuesta por parte del servidor en formato JSON y debemos convertir esa respuesta en una serie de objetos. En el caso del servidor utilizamos la tecnología de Spring, pero en el cliente utilizamos tecnología totalmente diferente, por lo que debemos realizar el Unmarshalling de otra forma.

El motor de Unity incorpora una utilidad llamada JSONUtility con la que podemos realizar fácilmente el Unmarshalling, proporcionando la clase que representa el objeto y el mensaje en sí. Con esta utilidad podemos transformar un mensaje JSON en una composición de objetos de tipo Record que podemos manipular y representar fácilmente en este juego.

La comunicación se establece cuando muere el jugador durante un nivel 5.7, primero se llama al servicio REST mediante una llamada GET normal con unos parámetros construidos de forma dinámica. Una vez se ha obtenido la respuesta del servidor, se procesa con la utilidad JSON y obtenemos los objetos como objetos de entidad del propio juego. Una vez tenemos todos los objetos los metemos en una lista ordenada, proporcionando el criterio de ordenación, en nuestro caso, por segundos.

```
IEnumerator saveRecord(){
        string uid = SystemInfo.deviceUniqueIdentifier;
        int record = Mathf.RoundToInt(recordTimer);
        manager.GetComponent<CharacterProperties>().saveMoney(record);
```

```
        string ip = "http://10.211.55.4:8080";
        string conxtext = "/saniprintServer";

        string petition = "/storeRecord?"
           + "name=" + uid
           +"&seconds=" + record;

        string url = ip + conxtext + petition;

        WWW saveRecord = new WWW(url);
        yield return saveRecord;

        //Receive top
        petition = "/getTop";
        url = ip + conxtext + petition;
        saveRecord = new WWW(url);
        yield return saveRecord;

        record[] listOfRecords = JsonHelper.getJsonArray<record>(saveRecord.text);
        List<record> orderedList = new List<record>(listOfRecords);
        orderedList.Sort(delegate(record a, record b) {
                return (b.seconds).CompareTo(a.seconds);
        });

        //Print top
        for (int i = 0; i < orderedList.Count && i < 5; i++)
        {
                Text text = GameObject.Find("record" + (i + 1)).GetComponent<Text>();
                text.text = "" + orderedList[i].seconds;
                if (orderedList[i].uid.Equals(uid))
                {
                        text.color = Color.red;
                }
        }
}
```

Listado 5.7: Comunicación con el servidor

Capítulo 6

Resultados

En el presente capítulo presentaremos un ejemplo de uso para el usuario final, utilizando total o parcialmente todos los elementos desarrollados durante este TFG. TODO acronimo

6.1 Ejemplo de uso

Una vez el usuario carga la aplicación, se podrá encontrar directamente con la pantalla de nivel ya cargada y con el protagonista listo para correr y desempeñar el nivel (ver Figura 6.1), antes se presentará un menú en el cual el jugador escogerá si quiere primero aplicar mejoras al personaje para desempeñar con más facilidad el nivel.

Figura 6.1: Estado inicial

Si el jugador pulsa sobre Üpgrades"(ver Figura 6.2), accederá al menú de mejoras que le permitirá comprar diferentes mejoras para su personaje, entre ellas, hacer más daño a los enemigos, saltar más alto o reducir la velocidad a la que acelera de velocidad progresivamente de nivel. Estas mejoras se traducen en mayores facilidades para el jugador para llegar más lejos en el nivel, si son utilizadas correctamente. Estas mejoras incrementan su coste tanto en tiempo como en dinero. Para obtener dinero lo único que necesita el jugador es jugar y aguantar tantos segundos como le sea posible. Para esperar el tiempo basta tanto esperar, como jugar al juego o como simplemente apagar el móvil hasta que dicho tiempo

transcurra.

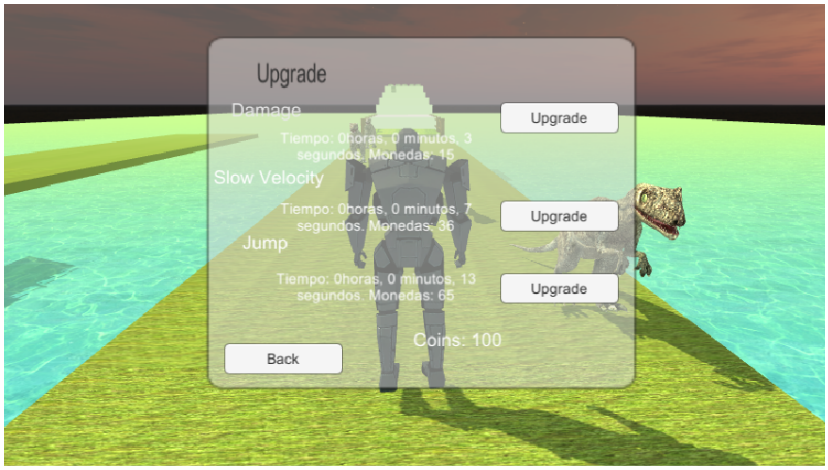

Figura 6.2: Menú de mejoras

Una vez el jugador decide empezar el nivel, el robot comenzará a andar a una determinada velocidad que se irá incrementando poco a poco de manera indefinida para que sea cada vez más difícil superar el nivel. Delante del jugador se va generando de manera procedural el nivel, compuesto en su mayoría por obstáculos que obligan al jugador a saltar, esquivar obstáculos o apoyarse encima de plataformas superiores. Adicionalmente el nivel genera enemigos aleatorios que caminan en dirección opuesta al jugador, en ese momento podrá decidir si quiere pulsar sobre el para eliminarlo o si por el contrario prefiere tan solo esquivarlo y continuar el nivel.

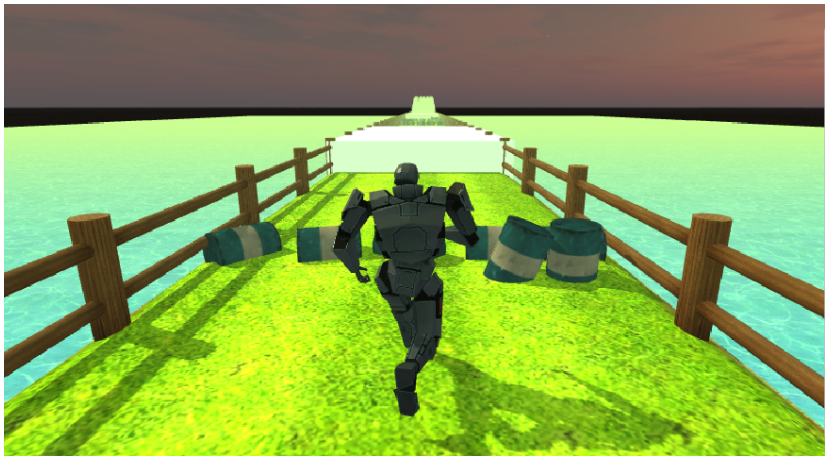

Figura 6.3: Imagen del juego

Una vez el jugador muere, se envía la puntuación al servidor y se solicita los mejores 5 records que se han alcanzado en el juego, si uno de los records es el del jugador, se imprimirá en rojo. En ese momento el jugador reinicia el nivel volviendo al punto de partida en el que puede mejorar a su personaje o bien volverlo a intentar.

Capítulo 7

Conclusiones y trabajos futuros

En el presente capítulo haremos una reflexión acerca de aquello que se puede aprender del desarrollo de este proyecto así como consideraciones a tener en cuenta para un segundo ciclo de desarrollo en este proyecto.

En lo relativo a los objetivos, se puede observar cómo se ha alcanzado el objetivo final realizando efectivamente todo el desarrollo del videojuego a partir de la especificación formal. El juego cumple con las especificaciones técnicas permitiendo al jugador jugar de forma fluida, el cual es un requisito indispensable en el campo de la informática interactiva.

Este trabajo en lo relativo a lo teórico me ha enseñado mucho tanto de conceptos teóricos relativos a la informática gráfica como prácticos, destacando el uso de un motor mundialmente conocido. En lo relativo a la arquitectura del Software he aprendido como una alternativa a un buen diseño, el paradigma de la arquitectura orientada a componentes, otro de los diseños Software que no es necesariamente mejor o peor que otros, es bueno cuando es conveniente usarlo, y aprender cuándo se debe aplicar el diseño es igual de importante que cómo implementarlo.

El desarrollo de videojuegos es un área multidisciplinar, requiere un importante esfuerzo no sólo en lo relativo a el buen diseño software si no también en otros 2 campos igual de importantes. Uno de ellos es el **diseño de juego**, en campo de la informática tradicional, el momento de toma de requisitos cliente es crítica para el desarrollo, ya que un error en esta parte repercute en mucho esfuerzo inútil en etapas más tardías. Los videojuegos no son distintos, establecer un buen diseño y saber de qué manera tu juego va a ser divertido es vital, ya que aunque tu juego sea muy sólido, estable, y técnicamente brillante, no le va a interesar a nadie si su función más básica que es divertir, no es lo suficientemente satisfactoria. El pilar restante es el **apartado artístico** del juego, es importante que exista una colaboración estrecha entre desarrollo y el área de arte, ya que unas animaciones póbremente programadas puede quitar inmersión al jugador, y eso repercute negativamente en la experiencia. En este trabajo he tenido que lidiar no sólo con la parte técnica, si no también cubrir los campos básicos del diseño y el arte que compone un juego.

Este trabajo también me ha permitido crecer en el ámbito de los servicios red, creando una aplicación web en Java bajo el marco de Spring gestionado por Maven que más tarde será

desplegado por un contenedor de aplicaciones como Tomcat, permitiendo la comunicación mediante servicios REST, utilizando JSON como formato para la codificación y descodificación de los datos tanto en el lado del servidor como del cliente. Esto tiene un gran valor tanto por el número de tecnologías aprendidas y aplicadas como por practicar con un caso real, estableciendo una conexión real entre dos aplicaciones con un uso claramente definido.

7.1 Trabajos Futuros

Gracias a la naturaleza iterativa e incremental del proyecto, es fácilemente expandible para generar nuevas revisiones. Algo muy común en el mundo de los desarrollos de los videojuegos es comenzar escribiendo la definición del juego, empezar a describir un enorme abanico de funcionalidades que hagan más completo y más variado al videojuego en cuestión, sin embargo, cada funcionalidad requiere un esfuerzo importante de los 3 principales campos del desarrollo de un videojuego que son Arte, Software y Diseño. Por lo tanto es común llegar a un punto en el que el juego puede considerarse como un producto finalizado, sacarlo al mercado y empezar a trabajar en una segunda parte mejorada con ideas que han quedado en el tintero.

Journey of the Mechanic Village no es una excepción a este caso, y existe una gran variedad de funcionalidades que se pueden aplicar de cara el futuro para proporcionar una mejor experiencia, aquí se listarán algunas de ellas.

- **Record identificado por usuarios**, actualmente los records identifican a cada jugador por un identificador único gestionado únicamente por el motor de Unity, y reconoce los records de su UID como suyos. Es interesante plantear un sistema de reconocimiento de usuarios, tal vez soportado por una plataforme como Google en el que los jugadores puedan reconocerse unos a otros mediante sus nombres.
- **Extender la jugabilidad**, se puede plantear muchas cosas para añadir una mayor variedad al juego o incrementar la sensación de progresión como por la compra de mejoras, objetivos en cada nivel, enemigos especiales más poderosos, mayor variedad de escenarios...
- **Análisis de uso**, se puede enviar información al servidor a cerca de cómo los jugadores interactúan con la aplicación para estudiarla y mejorar el videojuego en sí.

Además de estas mejoras, también se pueden aplicar importantes esfuerzos en el diseño del juego, haciéndolo más divertido para el jugador como en el apartado artístico, dándole una coherencia propia a el universo de Journey of the Mechanic Village.

ANEXOS

Anexo A

Ejemplo de anexo

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Referencias

[Bla15] Libro Blanco del Desarrollo Español de Videojuegos. http://www.news.cornell.edu/Chronicle/00/5.18.00/wireless_class.html, 2015. DEV Asociacion espñaola de Empresas Productoras y Desarrolladoras de Videojuegos y Software de entretenimiento.

[DV15] Cleto Martín David Vallejo. *Desarrollo de Videojuegos:Un enfoque Práctico. Volumen 1. Arquitectura del Mo*. 2015.

[Gre09] Jason Gregory. *Game Engine Architecture*. A K Peters, Natick, MA, USA, 2009.

Este documento fue editado y tipografiado con LaTeX empleando
la clase **esi-tfg** (versión 0.20170108) que se puede encontrar en:
`https://bitbucket.org/arco_group/esi-tfg`