UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de un videojuego Tower Defense usando Unity3D y una API para la IA en lenguaje C#

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Enrique Gonjar Verdejo

Tutor: Ramón Pascual Mollá Vayá

2018/2019

Resumen

Este proyecto consiste en el desarrollo de un videojuego de estrategia tipo Tower Defense utilizando el motor gráfico Unity. Se hace uso de una API para la gestión de la inteligencia artificial basada en el uso de máquinas de estados finitos. Desarrollada en el departamento de Sistemas informáticos y Computación de la Universitat Politècnica de València. El videojuego se creará desde cero y se lanzará para los sistemas operativos Android y Windows. El código fuente será implementado en el lenguaje de programación C#.

Palabras clave: Unity, lenguaje C#, Android, FSM, videojuego, estrategia, IA.

Abstract

This project consists of the development of a Tower Defense strategy video game using the Unity graphics engine. It makes use of an API for the management of artificial intelligence based on the use of finite-state machines. Developed in the Computer Systems and Computing Department of the Universitat Politècnica de València. The game will be created from scratch and will be launched for Android and Windows operating systems. The source code will be implemented in the C# programming language.

Keywords: Unity, lenguage C#, Android, FSM, strategy, videogame, IA.

Tabla de contenidos

1 Introducción

En este documento se describe el videojuego que se ha realizado, llamado Forest Defense. Se trata de un videojuego de estrategia en tiempo real en 3D, con vista desde arriba, realizado en el motor de Unity (versión 2018). Se ha hecho uso de una API para la realización de la Inteligencia Artificial de los enemigos basada en máquinas de estados finitos que fue realizada por José Alapont Luján que se desarrolló en un TFM del IARFID[1].

Se ha desarrollado con **fines académicos** dado al uso de recursos[2] sin licencia para comercializar, aunque no se descarta que a largo plazo se pueda comercializar obteniendo la licencia para ello y competir contra otros juegos del mismo género. Las herramientas escogidas para su realización se decidieron debido a la gran documentación que tienen y al conocimiento previo autodidacta obtenido de ellas.

1.1 Motivación

A lo largo de mi trayecto por la carrera, he tenido la posibilidad de cursar asignaturas donde he podido realizar varios videojuegos sencillos para diferentes plataformas, como Windows, Android e incluso para una consola portátil como la Nintendo DS. La experiencia adquirida en las asignaturas ha sido la principal motivación para seguir creciendo en el diseño y desarrollo de videojuegos.

Desde la última década, la industria de los videojuegos está experimentando un gran crecimiento. Tanto ha sido su crecimiento que España ya es el quinto mercado de videojuegos más grande de Europa. Aunque el principal canal son los juegos en consolas como fuente de ingresos, se espera que se experimente un mayor crecimiento en los juegos casuales que son aquellos basados en apps para smartphones o *tables* o accesibles desde un navegador.[3]

Este crecimiento es debido a la introducción de herramientas como Unity o Unreal Engine que facilitan la creación de videojuegos de alta calidad e incluso aplicaciones con el uso de muy pocos recursos. Esto implica que realizar juegos pequeños de manera independiente está al alcance de todos.

Además, el interés por la inteligencia artificial en los videojuegos, ha sido una motivación extra.

---

[1] Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen digital.
[2] Recursos tales como modelos 3D y efectos de partículas que se han utilizado para facilitar el desarrollo del videojuego con fines académicos.
[3] Aevi, La industria del videojuego http://goo.gl/r5XYg8

El desarrollo de un videojuego necesita de conocimientos en muchos de los campos que un ingeniero informático debe conocer, como son la programación, ingeniería de software, inteligencia artificial, teoría de autómatas o redes de computadores. Así como otros más generales, como son la física o las matemáticas, el diseño de niveles e iluminación. Por todo esto se considera un proyecto completo y una forma adecuada de demostrar los conocimientos que se han aprendido en la carrera.

1.2 Objetivos

El objetivo de este proyecto es el desarrollo de un videojuego de estrategia Tower Defense en 3D, con la ayuda del motor gráfico de Unity3D ya que permite realizar un mismo proyecto en distintas plataformas. Entre estas plataformas destacan, Windows y Android, son las más conocidas e importantes en el mercado actual y en las que está implementado este proyecto.

No se pretende realizar el videojuego completo, sino un prototipo para mostrar las posibilidades de las herramientas elegidas. El prototipo consta de la jugabilidad clásica para este tipo de juego de estrategia y además de añadir ciertos comportamientos inteligentes a los enemigos.

Se desea ampliar los conocimientos previos de las herramientas elegidas y con ello ganar experiencia en el proceso de creación de un videojuego en general.

El uso y aprendizaje de la API de gestión de Inteligencia Artificial de José Alapont Luján es otro de los objetivos de este proyecto. Se busca crear comportamientos inteligentes controlados por las distintas máquinas de estados que proporciona esta API.

1.3 Estructura de la obra

En este apartado se describe la estructura del documento. El documento comienza con una breve introducción explicando los objetivos que se quieren conseguir y las motivaciones que han llevado a la realización del mismo.

A continuación, se presenta el apartado del estado del arte donde se expone brevemente la situación actual del género del juego, junto a un estudio de la competencia donde se analiza los juegos más exitosos del género hasta la fecha. En último lugar, se describen las herramientas que hay en el mercado para el desarrollo de videojuegos y concluye el apartado con la selección de las herramientas utilizadas en este proyecto.

En el siguiente apartado se aborda el diseño del videojuego de una manera resumida y visual.

La planificación es el siguiente apartado, donde se describe la distribución del tiempo para el desarrollo del proyecto.

Después la implementación constituye el siguiente apartado donde se explica todas las partes del desarrollo del proyecto. El documento sigue con el apartado conclusiones donde se analiza el resultado final del proyecto.

Finalmente se presentan los apartados agradecimientos, trabajos futuros, bibliografía, el apéndice y un anexo que contiene el documento de diseño.

1.4 Esquema de trabajo

Para la realización de este proyecto se ha seguido el siguiente esquema de trabajo:

- Creación del documento de diseño del videojuego (GDD).
- Creación del diagrama de Gantt para planificar el proyecto.
- Lectura y compresión de la API para la IA.
- Búsqueda de recursos audiovisuales.
    - Búsqueda de modelos 3D para enemigos.
    - Búsqueda de recursos para la decoración de los niveles.
    - Búsqueda de modelos 3D para las torres.
- Programación del videojuego.
    - Programación de la IA de los enemigos.
    - Programación de las torres.
    - Programación de la interacción del jugador con el nivel.
    - Programación del movimiento de la cámara.
        - Controles para Windows con teclado.
        - Controles para Android con controles en pantalla.
    - Programación de menús e interfaces.
        - Pantalla de inicio.
        - Pantalla de opciones.
        - Pantalla de controles
        - Pantalla de selección de niveles.
        - Pantalla de carga.
        - Interfaz del juego.
- Creación de los niveles definitivos.
    - Creación de los niveles.
    - Iluminación y decoración
    - Inclusión de audio.
- Corrección de errores, ajuste de dificultad del juego y ajustes finales.
- Testeo en varias plataformas.
    - Testeo en varios dispositivos con sistema operativo Windows.
    - Testeo en varios dispositivos Android.
- Redacción de la memoria.

1.5 Metodología

De las diferentes metodologías estudiadas para el desarrollo de software durante la carrera se ha optado por seguir una metodología basada en un modelo de desarrollo iterativo e incremental.

Este modelo combina el modelo en cascada donde las fases de desarrollo (análisis, diseño, implementación y pruebas) se realizan de forma secuencial. Empleando el desarrollo incremental se obtiene, al final de cada iteración, una versión funcional del producto de esta manera el sistema se desarrolla poco a poco y se obtiene siempre un *feedback* continuo con el usuario.

Se ha elegido esta metodología porque es la que mejor se adapta y la que más se suele emplear en proyectos destinados al desarrollo de videojuegos. Este método se ha aplicado en cada una de las subtareas asignadas a las tareas principales descritas en el apartado planificación.

2 Estado del arte

En este apartado se hará un estudio del mercado actual de videojuegos en dispositivos móviles, profundizando en los Tower Defense, para valorar las posibles opciones que se ofrecen en este género y sus puntos fuertes. También se identificarán las características comunes para plantear mejoras.

Para finalizar, se describirán las herramientas que se han utilizado para la realización de este proyecto.

2.1 Situación del género Tower Defense

El género Tower Defense o TD está clasificado en el mercado de juegos para dispositivos móviles como un subgénero de estrategia en tiempo real.

Los videojuegos de estrategia se caracterizan porque requieren que el jugador ponga en práctica sus habilidades de planeamiento y pensamiento para maniobrar, gestionando recursos de diverso tipo como materiales, humanos, militares… para así lograr la victoria.[1]

En general los videojuegos de estrategia toman uno de cuatro posibles modelos dependiendo de si el juego es por turnos, en tiempo real y si el juego se enfoca en estrategia en todos los ámbitos o sólo en una, preferentemente la militar (táctica).

Los términos de táctica y estrategia suelen utilizarse como sinónimos, aunque la estrategia es un esquema que se implementa para intentar alcanzar los objetivos y la táctica es la forma prevista para alcanzar dichos objetivos. Por ejemplo: el

objetivo de una guerra es conquistar el territorio enemigo. La estrategia puede consistir en sitiar[4] la región para impedir la llegada de ayuda, mientras que las tácticas empleadas incluyen acciones específicas como bombardear los puentes o colocar minas en las carreteras.[5]

Los videojuegos de estrategia se dividen en subgéneros de acuerdo a la dinámica y al predominio de la táctica y la estrategia y estos subgéneros son los siguientes:[4]

Estrategia en tiempo real

Son videojuegos de estrategia en los que la acción transcurre de forma continua en el tiempo y no hay turnos. Están pensados para ser jugados de forma muy dinámica y rápida. Se caracterizan por estar más trabajados en su apartado gráfico, ya que al tener terrenos de juego más pequeños que otros subgéneros, se pueden representar más texturas sin que el rendimiento se vea alterado. A diferencia de los basados en turnos no precisan un planteamiento tan pausado y se centran muy a menudo en la acción militar. En cuanto a la recolección de recursos son siempre materias primas.[3]

Las batallas se representan a una escala de refriega, aunque hay títulos donde se centran en representar batallas multitudinarias con militares como unidades en el terreno. Uno de los títulos más populares de este subgénero es:

Clash of Clans[6]: es un videojuego multijugador y también de un solo jugador para dispositivos móviles con plataformas Android o IOS con fecha de lanzamiento el 2 de agosto de 2012 creado por Supercell compañía de videojuegos con sede en Helsinki, Finlandia. El juego consiste en que el usuario debe crear una aldea, entrenar tropas y elaborar hechizos para atacar a otras aldeas de jugadores en línea. También puede realizar campañas de un jugador para conseguir recursos como oro, elixir y elixir oscuro para poder mejorar a las tropas y para protegerse de ataques de enemigos.

[4] Definición: Poner cerco a un lugar para lograr su rendición.
[5] Definición de táctica. https://definicion.de/tactica/
[6] Link de descarga de Clash of Clans
https://play.google.com/store/apps/details?id=com.supercell.clashofclans&hl=es

Ilustración 1. Juego *Clash of Clans*

Táctica en tiempo real[7]

Este subgénero comparte aspectos de los juegos de simulación y juegos de guerra además de que se enfocan en aspectos operacionales y control de guerra. A diferencia de los juegos de estrategia en tiempo real, el manejo económico y de recursos y la construcción de edificios no forman parte de las batallas.

Un ejemplo es el título de *World in Conflict*[8] que fue desarrollado por la empresa sueca Massive Entertainment y publicado en 2007 por Sierra Entertainment para PC y Xbox 360. El juego se ambienta en 1989, donde la Unión Soviética en vez de caer decide iniciar una Tercera Guerra Mundial. El jugador puede controlar en el modo campaña a los ejércitos de Estados Unidos y a varias naciones europeas miembros de la OTAN (Reino Unido, Italia, Francia, Alemania Occidental). Su característica principal es su motor gráfico que hasta ordenadores de gama baja consigue unos efectos de explosión y detalles nunca vistos en el mundo de la estrategia.

[7] Videojuegos de táctica en tiempo real
https://es.wikipedia.org/wiki/Videojuego_de_t%C3%A1ctica_en_tiempo_real
[8] Página donde se puede encontrar el juego World in conflict
https://www.gog.com/game/world_in_conflict_complete_edition

Ilustración 2. Juego *World in Conflict*

Estrategia por turnos

El término de "juego de estrategia por turnos" se aplica generalmente para distinguirlos de los juegos de estrategia en tiempo real. Se caracteriza por que el usuario posee un periodo de análisis para realizar una acción. Los juegos por turnos vienen en dos formas dependiendo si, en un turno los jugadores juegan simultáneamente o juegan sus turnos en secuencia.

Un ejemplo de este género es *Empire-Strike*[9] que es un juego de estrategia ambientado en un mundo de fantasía donde el jugador podrá tener el control total de un imperio y elegir entre diferentes razas como orcos, humanos, elfos, no muertos y elfos oscuros.

Ilustración 3. Juego *Empire-Strike*

[9] Página oficial del juego Empire-Strike: https://www.empire-strike.com/

Los videojuegos de estrategia también se pueden categorizar por su temática independientemente de que sean en tiempo real o por turnos, estratégicos o tácticos. Estas son las temáticas más importantes:

Videojuegos de construcción de imperios: también conocidos como 4x, son juegos donde se debe explorar, explotar, expandir y exterminar. [2] Pueden ser por turnos o en tiempo real. Un título muy conocido es *Age of Empires***.** Este videojuego fue desarrollado por Ensemble Studios y más tarde por Skybox Labs y publicado por Microsoft Games Studios**.**

Videojuegos de artillería: generalmente son juegos por turnos en los que se utilizan las armas como tanques para atacarse unos con otros. Un ejemplo clásico de este género es *Worms.*[10] Este videojuego de estrategia militar fue desarrollado por Team17 y lanzado en 1995. El jugador controla a un equipo de gusanos contra otros equipos de gusanos controlados por una computador u oponente humano. El objetivo es utilizar varias armas para matar a los gusanos de los otros equipos. Podemos encontrar diferentes versiones[11] que se han ido distribuyendo desde mediados de los años 1990.

Ilustración 4. Juego de Worms

[10] Link donde se encontrar el juego Worms: https://store.steampowered.com/app/327030/Worms_WMD/?l=spanish

[11] Las diferentes versiones pueden ser categorizadas en varias generaciones según el modo de juego. En este link se puede encontrar las diferentes generaciones y distintas versiones del juego: https://es.wikipedia.org/wiki/Worms_(serie)

MOBA[12]: los juegos MOBA (videojuego multijugador de arena de batalla en línea) enfatizan el juego de equipo. El objetivo es destruir la estructura principal de los oponentes con la ayuda de unidades que no son controladas por el jugador y que se generan periódicamente y marchan hacia la estructura principal por unos senderos llamados "carriles" [5]. El jugador controla a un "héroe" que tiene varias habilidades. Se diferencia de los juegos los juegos de estrategia en tiempo real en que no hay construcción de unidades (estas se generan solas) y los jugadores sólo controlan a un personaje. El ejemplo más popular de este género es *League of Legends*[13] que es un juego desarrollado por Riot Games para Microsoft Windows y OS X.

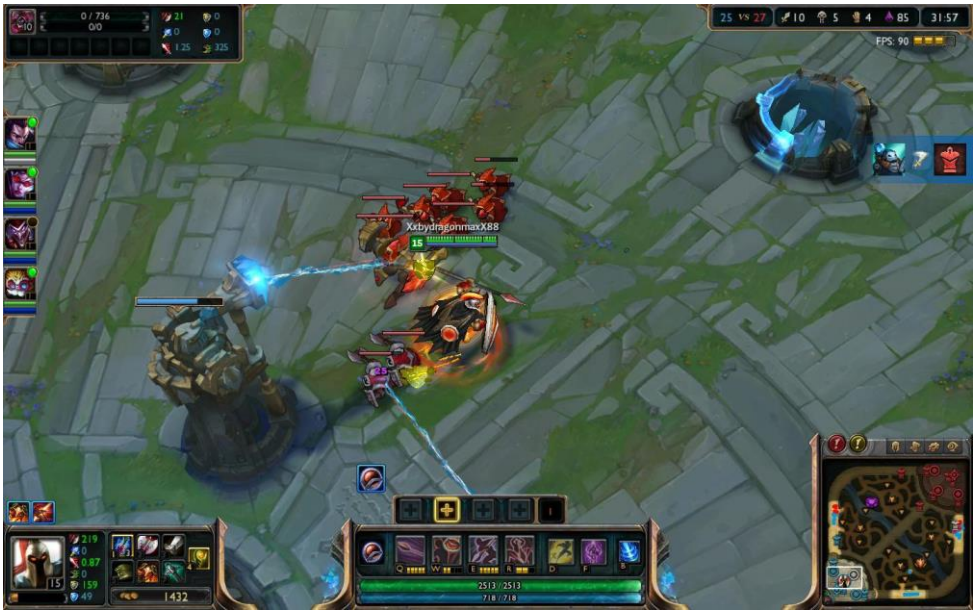

Ilustración 5. Juego *League of Legends.*

Tower defense: es el subgénero de estrategia en tiempo real en el que se centra este proyecto. El objetivo de los juegos Tower defense es lograr que las unidades enemigas no lleguen a cruzar el mapa, para lograrlo se deben construir torres que las atacan al pasar. Los enemigos y las torres poseen habilidades y cuando se elimina a una unidad enemiga se reciben puntos o dinero para así poder mejorar las torres. Los juegos más populares son *Plants vs zombies*, *Kingdom Rush* entre otros. Todos ellos se analizarán en el siguiente apartado.

A modo de resumen se muestra una tabla comparativa de todo lo descrito anteriormente.

[12] ¿Qué es un MOBA?
https://es.wikipedia.org/wiki/Videojuego_multijugador_de_arena_de_batalla_en_línea
[13] League of Legends es gratuito y se puede encontrar en la página oficial de Riot Games
https://play.euw.leagueoflegends.com/es_ES

Temática	Estrategia	Títulos	Plataformas
Construcción de imperios	turnos o tiempo real	Age of Empires	Windows, PlayStation 2, Mac
Videojuegos de artillería	por turnos	Worms	PlayStation
MOBA	juego en equipo	League of Legends	Windows ,OS X
Tower Defense	tiempo real	Plants vs Zombies	Android

Ilustración 6. Tabla resumen

2.2 Estudio de la competencia

Por su simplicidad de su mecánica de juego, trabajar en el género Tower Defense se ha convertido en un arma muy útil para que empresas primerizas se den a conocer. A continuación, se describen algunos de los principales Tower Defense del mercado actual.

Kingdom Rush[14]: un juego muy adictivo que resalta por la calidad de sus gráficos y la gran variedad de enemigos que existen, cada uno con sus características particulares. Ha sido desarrollado por Ironhide Game Studio y publicado por Armor Games, lanzado como juego de navegador gratuito. Posteriormente fue lanzado para más plataformas como Android e IOS.

Está ambientado en fantasía medieval. Cada nivel presenta una ruta predeterminada con ranuras vacías alrededor, llamadas “puntos de estrategia” donde el jugador puede construir las torres. Hay cuatro tipos de torres para crear y evolucionar: cuarteles de soldados, bombarderos, arqueros y magos. Todas ellas se construyen usando monedas que se obtienen eliminando a los enemigos y al inicio de cada nivel. También el jugador dispone de un héroe en cada nivel (que podrás elegir entre los gratuitos y los de pago) y elementos que se pueden comprar con gemas, como bombas y dinamita. Cuando se completa por primera vez un nivel se le recompensa al jugador con una o más estrellas. Estas estrellas se pueden usar para desbloquear mejoras pasivas que mejoran la efectividad de las distintas torres y hechizos. Se pueden obtener hasta tres estrellas por nivel si el jugador consigue superar el nivel con 18 vidas de 20. Esto permite

[14] Link de descarga en steam de Kingdom Rush
https://store.steampowered.com/app/246420/Kingdom_Rush/?l=spanish
Link para jugar en navegador web: https://www.miniclip.com/games/kingdom-rush/es/

desbloquear diferentes modos de juego como el modo heroico y el modo desafío de hierro. Estos modos se desarrollan en el mismo nivel donde se han desbloqueado y consisten en superar el nivel con la restricción de tener solamente una vida. Si se consigue superar el jugador obtendrá una estrella más. El juego está disponible en *Steam,* navegador web y en *PlayStore*.[15]

Ilustración 7. Juego *Kingdom Rush*

Plants vs Zombies[16]: desarrollado y publicado el 5 de mayo de 2009 por PopCap Games para Microsoft Windows, dispositivos móviles y consolas de la anterior generación como PlayStation 3 y Xbox 360. El juego consiste en defender una casa de una horda de zombis que quieren devorar los cerebros de los residentes. Para ello el jugador dispone de diferentes plantas de defensa o ataque que le ayudarán a defenderse de los zombis. Este Tower Defense plantea una perspectiva diferente para el jugador, situando a los enemigos en diferentes caminos que se defienden de manera independiente. Los zombis presentan diferentes atributos o habilidades, como cavar por debajo de las plantas o saltar por encima de ellas mediante el uso de pértigas, llevar diferentes objetos como cascos o conos, escaleras, globos o invocar a cuatro zombis para realizar la coreografía inspirada en “Thriller” en la que aparece un zombi caracterizado como Michael Jackson.

[15] Kingdom Rush Frontiers, análisis de un TD increíble.
https://www.vix.com/es/btg/gamer/4986/kingdom-rush-frontiers-analisis-de-un-td-increible

[16] Link de descarga de Plants vs Zombies
https://play.google.com/store/apps/details?id=com.ea.game.pvzfree_row&hl=es_419

El principal modo de juego es el modo aventura en el que se van presentando de forma progresiva las distintas plantas y zombis. Según se va progresando se van desbloqueando diferentes opciones o modos de juegos extra.

Para finalizar este título ha tenido numerosas nominaciones a premios Fue nominado para el Juego Casual del Año y Logro en diseño de juego Interactive Achievement Awards por la Academia de Artes y Ciencias Interactivas. En 2010, recibió nominaciones a Mejor Diseño de Juego Innovación y Mejor juego de descargas en los Game Developers Choice Awards.

Ilustración 8. Juego *Plants vs Zombies*

Jelly Defense[17]**:** este videojuego se sumerge en un mundo imaginario situado en la "Nación de Jalea" (*jelly nation*) donde el jugador tiene que defender este reino y las gemas que guarda. Las mecánicas siguen las bases de un tower defense clásico. Tenemos una serie de torres, cada una elimina a un tipo de enemigo dentro de un determinado rango y que también podremos mejorar o vender para sustituirlas por otro tipo de torre. Con esto destruimos las oleadas de enemigos y con ello evitar que se lleven las gemas del reino además que el jugador puede lanzar meteoritos y tornados para limpiar las oleadas con más rapidez. Los enemigos se diferencian por dos colores: rojos y azules. En cuanto a las torres hay diferentes tipos: las torres de color rojo que atacan a los

[17] Link de descarga en google play Jelly Defense
https://play.google.com/store/apps/details?id=pl.idreams.jellydefense

enemigos de color rojo, las torres azules para los enemigos azules y un tipo de torre híbrido que ataca a todos.[18]

El juego cuenta con gráficos muy buenos, con unos paisajes coloridos y una música envolvente que se complementa con una jugabilidad que pone a prueba la destreza y el ingenio del usuario. El juego cuenta con 17 niveles y ha sido desarrollado por Infinite Dreams[19]. Tiene un coste de 3,68€ y se puede comprar y descargar en Google Play.

Ilustración 9. Juego *Jelly Defense*.

Bloons Tower Defense[20]: en este videojuego los enemigos son globos que tienen como objetivo completar el recorrido de cada nivel. Para impedirlo el jugador debe colocar como defensa a unos monos. Cada mono tiene características concretas como son: artilleros, portadores de boomerangs, monos congelantes y capaces de lanzar pegamento para ralentizar a los globos. Además de los monos el jugador cuenta con múltiples recursos como colocar a lo largo del recorrido clavos para explotar a los globos o lanzar bombas. Cada vez que se consigue explotar un globo se obtiene dinero para mejorar cada una de las defensas.[21]

[18] Jelly Defense https://www.actualidadiphone.com/jelly-defense-un-divertidisimo-juego-de-defender-la-torre/

[19] Link de la página oficial del juego Jelly Defense https://www.idreams.pl/JellyDefense/

[20] Bloons tower defense 6 https://store.steampowered.com/app/960090/Bloons_TD_6/

[21] Bloons Tower Defense Review. https://www.ign.com/articles/2011/11/11/bloons-tower-defense-review

Ilustración 10. *Bloons Tower Defense*

Al contrario que *Jelly Defense*, *Bloons* TD deja al margen el apartado audiovisual para, centrarse en la jugabilidad proponiendo una gran cantidad de opciones y diferentes niveles y retos para los jugadores de cada versión de la saga. El juego ha sido desarrollado por *Ninja Kiwi* y actualmente tiene 6 versiones. Los cambios en cada una de las versiones se aprecian en el aumento de las actualizaciones por torre, actualización de los gráficos la capacidad de guardar el juego actual como en *Bloons Tower Defense 4.*

A continuación, se presenta a modo de resumen una tabla comparativa de los videojuegos anteriormente mencionados.

Título & Características	Kingdom Rush	Plants vs Zombies	Jelly Defense	Bloons Tower Defense
Gráficos	Muy cuidados	Estética original y colorida	Muy buenos y coloridos	De calidad baja
Nº de jugadores	1	1	1	1
Tipos de enemigos	Orcos, trolls y otros monstruos	Diferentes tipos de zombies	Alienígenas de color rojo y azul	Globos de diferentes colores

Tipos de torre	Arqueros, magos, bombarderos	Diferentes tipos de plantas	Rojas, azules e híbridas	Diferentes tipos de monos.
Habilidades extra	Colocar tropas, lanzar meteoritos	-	Lanzar tornados	Colocar clavos y bombas
Nº de niveles	26	50	17	45
Modos de juego	Heroico, desafío de hierro, campaña	Aventura, minijuegos, puzzles	Aventura	Aventura
Plataformas	Navegador web, Android, IOS, Windows, Linux	Xbox 360, one S, Android, Windows, Nintendo DS,3DS, Steam OS, IOS, Mac OS X, PlayStation 3, Vita	IOS, Android, Mac OS X	IOS, Android, navegador (flash) Mac OS X, Windows, Nintendo DSi, switch PlayStation 4, portable
Empresa desarrolladora	Ironhide Game Studio	PopCap Games	Inifinite Dreams	Ninja Kiwi

Ilustración 11. Tabla resumen de los juegos analizados.

Se puede concluir que cada juego mencionado ha sabido explotar un aspecto determinado del género en su propio beneficio, bien centrándose en la jugabilidad y dejando de lado los gráficos o proponiendo perspectivas diferentes como la del título *Plants vs Zombies.*

Es por ello que en este proyecto se quiere desarrollar un tower defense con las características que debería tener el mejor juego de este género. Estas características son las siguientes:

- Que presente una jugabilidad bien cuidada buscando un equilibrio en la dificultad para evitar que el jugador pueda quedarse bloqueado en un nivel.
- Evitar los puntos muertos haciendo que el jugador siempre esté en constante interacción con el entorno.
- Algunos de los juegos mencionados anteriormente presentan una IA algo simple donde los enemigos solo van de un punto a otro y apenas tienen interacción con el entorno. Es aquí donde entra el propósito de este proyecto que pretende desarrollar una IA algo más avanzada donde los enemigos sean capaces de tener más interacción con el entorno. Por ejemplo, la posibilidad de atacar a las torres y los obstáculos que se encuentren en su camino.

2.3 Motores gráficos

En la actualidad existen una gran cantidad de motores gráficos para el desarrollo de videojuegos. Cada uno de ellos presentan unas características que se deben valorar a la hora de realizar un videojuego. A continuación, se describen algunos de los motores más importantes, así como sus ventajas y desventajas.

2.3.1 Unreal Engine

Es el motor de Epic Games. Estuvo orientado solo para desarrolladores. Desde 2015 es una herramienta totalmente gratuita con el único requisito es que hay que pagar un 5% al comercializar un producto realizado con este motor. Se usa C++ como lenguaje de programación, también cuenta con un editor visual de *scripting,* también conocido como sistema de *blueprints.* Como ventajas cabe destacar la calidad gráfica y el poder crear juegos complejos, además de ser gratuito. Sin embargo, su curva de aprendizaje es complicada y cuenta con una comunidad inferior con respecto a otros motores, siendo su principal desventaja.

2.3.2 Unity

Es probablemente el motor de juegos más usado en la actualidad. Permite desarrollar juegos en 2D y en 3D, cuenta con una documentación muy amplia y una inmensa comunidad. Cuenta con una versión gratuita y una de pago, aunque sus diferencias son mínimas. Soporta el lenguaje C# con sintaxis similar a Java y es un editor multiplataforma permitiendo exportar a muchas plataformas distintas incluyendo hasta consolas. Puede usarse con juegos comerciales siempre y cuando no se superen unos ingresos de 100.000 dólares al año. Recibe constantes actualizaciones cada cierto periodo de tiempo y su curva de aprendizaje es sencilla.

2.3.3 RPGMaker

Es un software para hacer juegos hecho como base para los usuarios que no saben programación. Aunque si se poseen conocimientos de programación

puede ser útil para desarrollar *plugins* usando JavaScript. Este software sirve para desarrollar juegos mayoritariamente del género RPG orientado a 2D, ya que posee un editor para la creación de escenarios y personajes mediante *sprites*. Fue desarrollado por ASCII Corporation.

2.3.4 GameMaker Studio

Tuvo su origen en los años 90, cuando Mark Overmars (su creador) empezó a crear una herramienta de animación para ayudar a sus estudiantes, con el paso del tiempo se ha vuelto una herramienta muy popular para la creación de videojuegos, principalmente en 2D. Este motor utiliza su propio lenguaje GML, el cual es muy flexible, su sintaxis es comparable con la de C++. Es capaz de exportar a una gran cantidad de plataformas distintas, móviles o escritorio, sin embargo, el editor sólo funciona en Windows. La versión gratuita es limitada ya que solo permite la exportación a Windows.

2.3.5 Godot

Es un motor de código abierto, totalmente gratuito, para desarrollar juegos en 2D y 3D. Creado por OKAM Studio y funciona en Windows, OS X y Linux entre otros. Hace uso de un lenguaje de scripting propio basado en Python. Permite exportar a plataformas móviles y de escritorio. La documentación es limitada y está únicamente en inglés, es una desventaja en comparación con otros motores que sí tienen su documentación traducida a otros idiomas.

2.3.6 CryEngine

Es un motor de videojuegos lanzado en 2002 y creado por un estudio alemán llamado *Crytek*. Fue lanzado originalmente para hacer una demo técnica para la empresa de tarjetas gráficas Nvidia. Tras su gran éxito los mismos creadores del motor acabaron desarrollando el videojuego *FarCry*. En 2016 pasarían a pertenecer a la empresa Ubisoft los derechos de dicho motor. El motor está programado con Lua, C++ y C#. Las primeras versiones del motor tenían una licencia de pago. Actualmente con la versión 5 el motor es totalmente gratuito y se limita a las donaciones que los usuarios quieran realizar de forma voluntaria. Se utiliza una programación basada en nodos muy parecida a los *blueprints* del motor Unreal. Cuenta con una comunidad pequeña con una amplia documentación que se puede encontrar en su página oficial[22], además de una gran cantidad de videotutoriales en Youtube. Toda esta documentación se encuentra en inglés.[23]

[22] Enlace a la página oficial: https://www.cryengine.com/

[23] ¿Qué es CryEngine? https://rubi3d.com/65-que-es-cryengine/

A continuación, se muestra una tabla comparativa de todos los motores mencionados y sus características más importantes

Motor gráfico & Atributos	Unity	Unreal Engine	RPGMaker	GameMaker Studio	Godot	CryEngine
Orientación	2D y 3D	2D y 3D	2D	2D	2D y 3D	3D
Sistema Operativo	Windows, Mac	Windows	Windows	Windows	Windows, Linux, OS X	Windows, Linux, OS X
Lenguaje de Programación	Lenguaje C#	Lenguaje C++, Editor visual Blueprints	Javascript	Lenguaje propio GML	Lenguaje de scripting	Programación basada en nodos
Licencia	Gratuita	De pago	De pago	Versión gratuita limitada	Gratuita	Gratuita en la versión 5
Documentación	Muy amplia	Escasa	Escasa	Amplia	limitada	Amplia
Curva de aprendizaje	Sencilla	Complicada	Sencilla	Media	Media	Media

Ilustración 12. Tabla comparativa de los motores mencionados

2.4 Selección de las herramientas

2.4.1 Motor gráfico

Para la elección del motor se han optado por dos opciones (Unity o Unreal Engine). Para decidir una de las dos opciones se ha realizado una tabla comparativa donde se muestran las diferentes ventajas e inconvenientes de ambos motores.

	Unity	Unreal Engine
Ventajas	● Curva de aprendizaje sencilla. ● Soporta el lenguaje C# ● Gran facilidad para realizar juegos en 3D o 2D ● Cuenta con una comunidad muy amplia y documentación muy detallada.	● Buen rendimiento y gráficos realistas. ● Soporta el lenguaje C++ y también el uso de Blueprints. ● Control total del código fuente del motor.
Inconvenientes	● Calidad gráfica menor que Unreal Engine ● No es de código abierto por lo que los errores no los puede solucionar el propio programador.	● Comunidad pequeña con pocos recursos. ● La dificultad del lenguaje C++

Teniendo en cuenta el estudio realizado de ambos motores, Unity es el mejor posicionado para la elaboración de este proyecto. Pese a que Unreal tiene un mejor rendimiento y su nivel gráfico es mejor que Unity, la poca experiencia que se tiene sobre este motor dificultará considerablemente este proyecto. Además, la API que se va a usar en este proyecto está implementada en C#, lenguaje que no es compatible con el motor Unreal ya que soporta C++.

Sin embargo, Unity soporta el lenguaje C#, tiene una amplia comunidad y una cantidad de recursos disponibles en la red que facilitan el desarrollo de cualquier juego y se cuenta con una experiencia previa con la herramienta. Es por ello que finalmente se ha decidido utilizar Unity.

2.4.2 Control de versiones

El control de versiones es una herramienta imprescindible para proyectos de este calibre, ya que permite evitar pérdidas de información y llevar un control de las modificaciones que se hacen durante el desarrollo de un proyecto. A continuación, se presentan diferentes herramientas para el control de versiones:

Unity Collaborate,[24] permite equipos pequeños de desarrollo guardar, compartir, sincronizar un proyecto de Unity en un entorno alojado en la nube. Esta herramienta es bastante interesante ya que permite tener un control de versiones. También permite recuperar archivos individuales o el proyecto en un estado anterior. Esta herramienta está disponible a partir de la versión de Unity 2017.4 y cuenta con dos licencias. Una gratuita que permite almacenar en la nube hasta un 1Gb y otra de pago que cuesta 9 dólares y permite almacenar hasta 25Gb.

Git[25], es un sistema de control específico de versión de fuente abierta creada por Linus Torvalds en el 2005. Específicamente, Git es un sistema de control de versión distribuida, lo que quiere decir que la base del código entero y su historial se encuentran disponibles en la computadora de todo desarrollador, lo cual permite un fácil acceso a las bifurcaciones y fusiones.

GitHub[26], es una compañía sin fines de lucro que ofrece un servicio de hosting de repositorios almacenados en la nube. Esencialmente hace que sea más fácil para los desarrolladores usar Git como la versión de control y colaboración.[27]

SVN, (Apache Subversion) es una herramienta de control de versiones de código abierto basada en un repositorio cuyo funcionamiento se asemeja enormemente al de un sistema de ficheros. Es un software libre bajo una licencia de tipo Apache/BSD[28].

Comparando cada una de las herramientas se ha elegido utilizar *Git* y *GitHub* por ser herramientas gratuitas y no tiene límite de almacenamiento en la nube. Al contrario que en *Unity Collaborate* que sí tiene un límite de almacenamiento y es de pago. Sin embargo, la alternativa de *SVN* al no tener apenas experiencia con la herramienta queda descartada.

2.4.3 Plugins y librerías

Unity cuenta con una gran cantidad de herramientas que mejoran algunas características del motor y también permiten facilitar las tareas de implementación. A continuación, se muestran diferentes herramientas que se han utilizado en este proyecto:

Log Viewer[29], esta herramienta permite verificar fácilmente los registros de la consola del editor dentro del juego. Esto permite encontrar rápidamente los errores que se puedan producir en tiempo de ejecución. Para poder mostrar esta

[24] Se puede encontrar más información en la documentación oficial de Unity https://docs.unity3d.com/es/2017.4/Manual/UnityCollaborate.html
[25] Página de descargar de Git https://git-scm.com/
[26] Página oficial a GitHub https://github.com/
[27] ¿Qué es GitHub? https://kinsta.com/es/base-de-conocimiento/que-es-github/
[28] Definición de licencia Apache https://es.wikipedia.org/wiki/Apache_License
[29] Se puede encontrar en tienda oficial de unity https://assetstore.unity.com/packages/tools/integration/log-viewer-12047

consola lo que se tiene que hacer es un gesto circular con el mouse en Windows (hacer clic y arrastrar) o el dedo (tocar y arrastrar) en la pantalla del dispositivo móvil para mostrar todos esos registros. Esta herramienta es gratuita hay otras alternativas como son *Reporter* o *LogCat Viewer* son herramientas de pago más completas pero su funcionalidad es la misma por lo que la herramienta gratuita es suficiente para el proyecto.

3 Diseño

3.1 Diseño conceptual

"*Forest Defense*" es un juego Tower Defense en 3D donde el jugador deberá superar diferentes oleadas de enemigos en cada nivel. El juego está ambientado en un mundo fantástico donde existen diferentes reinos y uno de ellos es el reino del bosque, lugar donde transcurre la gran mayoría de los escenarios. Los habitantes del bosque viven en paz y en armonía, pero han sido descubiertos por unos invasores que quieren arrasar con el bosque. Con la ayuda de los habitantes que nos proporcionan armas y la del propio bosque con la formación de muros de piedra debemos combatir la invasión.

Los enemigos aparecen en un punto de inicio siguiendo un camino predefinido hasta su punto final. El jugador deberá evitar que estos enemigos lleguen a su punto de destino utilizando armas que podrá comprar en una tienda. Estas armas se encargan de destruir a los enemigos y una vez destruidos le otorgará al jugador dinero para poder seguir comprando torres o mejorarlas.

Cada nivel dispone de diferentes torres distribuidas por el mapa donde se colocarán las armas. Estas armas se podrán mejorar o vender que tendrán un nivel de mejora y también existe otro tipo de defensa que son los muros de piedra que el jugador podrá colocar en el nivel para obstaculizar el avance de los enemigos y permitir a las torres destruir a los enemigos con más facilidad.

3.2 Diseño estructural

El diseño estructural abarca las diferentes clases empleadas para la implementación del proyecto. Estas clases o *scripts* se pueden dividir en cuatro grupos: las clases para la IA, los managers, las clases para el control de la interfaz gráfica y las clases dedicadas a la optimización.

Las clases para la IA se encargan de gestionar el comportamiento de los enemigos mediante las distintas máquinas de estados. En este grupo también se encuentran los distintos ficheros *XML* donde se describen cada una de las máquinas y las clases encargadas para la carga de las mismas.

Los managers se encargan de la gestión del sonido, del control de los estados en los que se encuentra el juego y de la interacción del jugador con el entorno a la hora de seleccionar y colocar las diferentes armas en las torres distribuidas

por cada nivel. Este tipo de clases utilizan el patrón de diseño *Singleton* que se encarga de asegurar que existe una única instancia de cada una las clases que lo implementa en toda la aplicación, permitiendo un acceso global desde otras clases.

Las clases dedicadas a la interfaz gráfica se encargan de actualizar y mostrar los mensajes y menús al usuario.

En cuanto a los scripts de optimización son los que implementan el patrón de diseño *object pool* que se encarga de reutilizar las instancias de los objetos, como los disparos de las armas para evitar problemas de rendimiento sobre todo en los dispositivos móviles.

3.3 Diseño de la IA

Los enemigos son los encargados de dar vida a los escenarios del juego mediante sus comportamientos. Estos comportamientos se han definido empleando la API de gestión de máquinas de estados. El diseño de los enemigos estará más detallado en el GDD que se encuentra en el apartado anexo de este documento.

A continuación, se describirán las diferentes máquinas de estados que definen los comportamientos de los enemigos. Se han diseñado cuatro máquinas de estados, una determinista y tres probabilísticas. Se ha decidido que sea así por cuestiones de diseño, aunque se han tenido otras opciones, pero finalmente han sido descartadas porque no se acoplan al comportamiento de los personajes.

3.3.1 El fantasma

Este enemigo cuenta con una máquina de estados determinista con dos transiciones y tres estados. Su comportamiento es el siguiente:

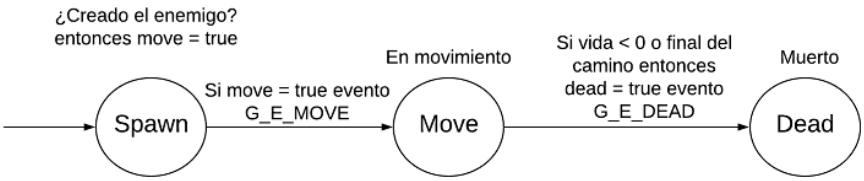

Ilustración 13**.** Máquina de estados del fantasma.

La máquina empieza con el estado inicial ***spawn*** donde se ejecuta la función que se encarga de hacer aparecer al enemigo y prepararlo para que empiece a

moverse. Mediante una variable booleana activará la condición de salida que permite lanzar el evento *g_e_move*[30] para cambiar al estado ***move***.

El estado ***move*** realiza la acción del movimiento del fantasma hasta que su vida llega a cero o llega a su destino. En ambos casos se usa una variable booleana para dar la condición de salida que permita cambiar al estado ***dead*** mediante el evento *g_e_dead* [31] donde realiza la acción de destruir al enemigo.

Todas las máquinas utilizan una o más variables booleanas para las condiciones de activación de los eventos que permiten las transiciones de un estado a otro. Se ha decidido así por comodidad y cuestiones de diseño.

3.3.2 El golem

El comportamiento del *golem* está gestionado por una máquina probabilística. Esta máquina contará con cinco estados y varias transiciones. Su funcionamiento se describe a continuación:

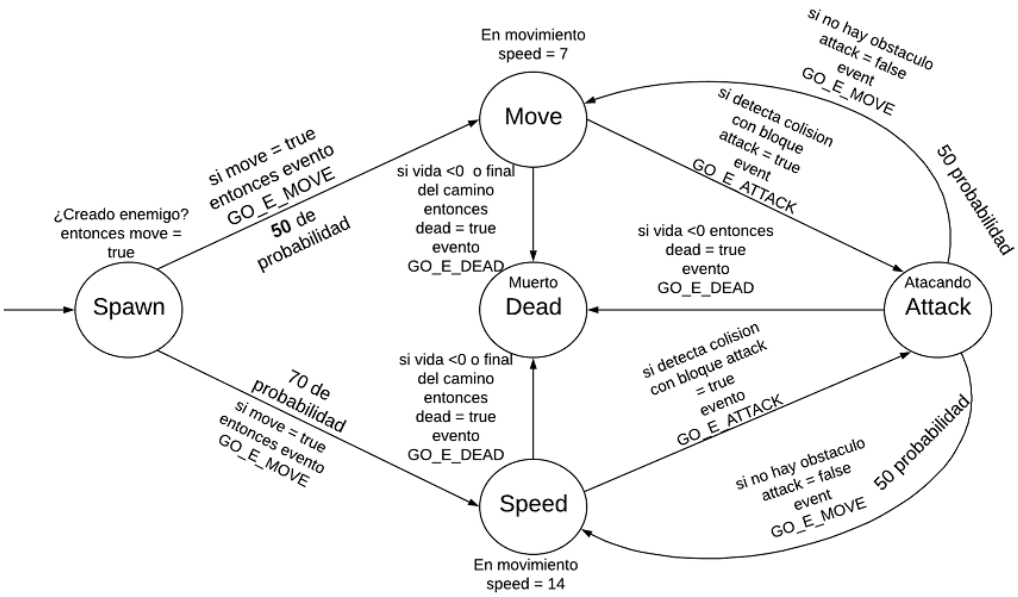

Ilustración 14. Máquina de estados del golem.

El enemigo parte del estado ***spawn*** que se encarga de la creación del enemigo en su punto inicial. Tras crearse el enemigo, se activará la condición de salida del estado ***spawn*** mediante la variable booleana correspondiente que tendrá el valor de cierto. Esto permitirá cambiar al estado ***move*** o al estado ***speed***

[30] Nomenclatura g_e_move g: ghost indica el enemigo, e: el evento y move el estado al que se va a hacer la transición tras el evento.

[31] Nomenclatura g_e_dead: g: ghost e: evento dead: el estado al que se va hacer la transición tras el evento.

dependerá de la probabilidad asignada que tiene un valor de 70% para la transición al estado ***speed*** y 50% para el estado ***move***.

El estado ***move*** permitirá mover al enemigo a una velocidad normal y el estado ***speed*** hará que el enemigo incremente su velocidad. Hay que aclarar que del estado inicial sólo podrá cambiar de estado una vez y sólo puede ser o el estado ***speed*** o el estado ***move***, en ningún momento los dos estados se ejecutan a la vez.

Se ha empleado una máquina probabilística para hacer que el enemigo tome decisiones y porque es la que mejor se acopla a este comportamiento.

El enemigo se mantendrá en uno de los dos estados hasta que se cumplan una de estas tres condiciones:

- La **primera** es que la vida del enemigo llegue a cero entonces en ese caso el valor de la variable booleana correspondiente valdrá cierto y en ese momento se activa el evento *go_e_dead*[32] que permite la transición al estado ***dead*****.**
- La **segunda condición** es que el enemigo llegue al último punto del camino entonces ocurrirá exactamente lo mismo que en la primera condición.
- Y la **última condición** es que el enemigo detecte un obstáculo entonces cambiará al estado ***attack*** mediante el evento *go_e_attack* y permanecerá en el estado ***attack*** hasta que el obstáculo sea destruido o cuando la vida del enemigo sea cero.

Permaneciendo en el estado ***attack***, si la vida del enemigo no llega a cero, pero el obstáculo sí que es destruido entonces se vuelve a uno de los dos estados anteriores que son: ***move*** o ***speed***. La probabilidad de que se escoja una de las dos transiciones es de 50%.

Para finalizar el estado ***dead*** se activará cuando la vida del enemigo llegue a cero o llegue a su punto de destino. Estas dos condiciones se pueden cumplir en los estados ***move*** o ***speed*** y en el estado ***attack*** sólo cambiará al estado ***dead*** cuando la vida llegue a cero ya que mientras está atacando el enemigo está parado y no puede llegar a su destino hasta que el obstáculo sea destruido.

3.3.3 El mago

La máquina de estados que se encarga del mago cuenta con seis estados. Es una máquina probabilística. Con el objetivo de que el esquema en la imagen sea

[32] Indica el evento que se activa cuando se produce una transición. La nomenclatura consta de tres partes separadas por guiones bajos. La primera parte indica el nombre del enemigo que en este caso es golem. La segunda parte indica que es un evento en caso de ser una acción aparecería la letra A o si es una transición la palabra TO. Y la tercera parte indica el estado al que se va hacer la transición en este caso al estado dead.

lo más claro posible se han unificado las transiciones de los estados de ataque al estado ***dead***.

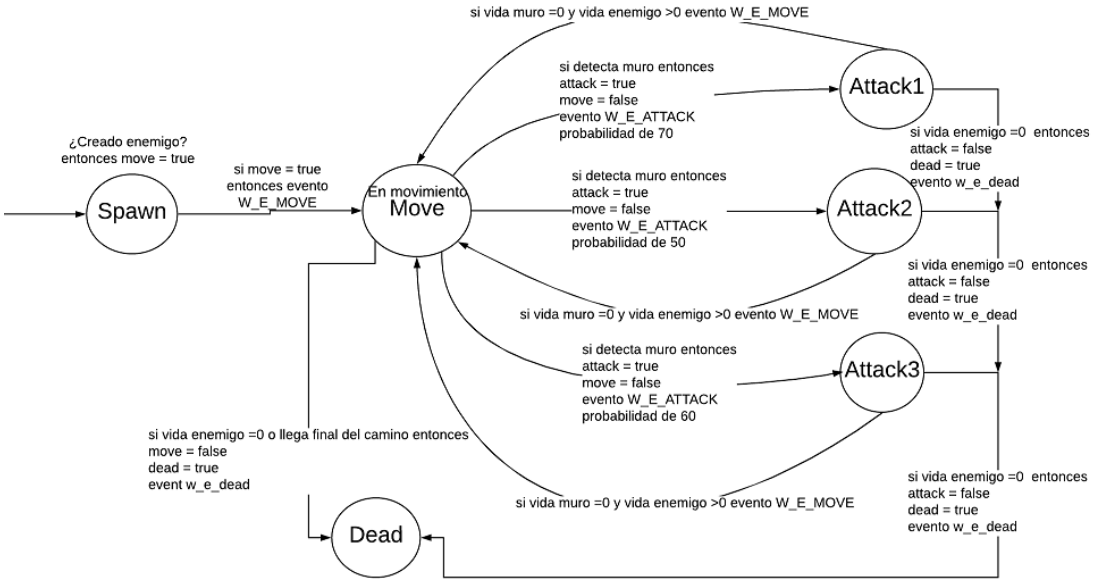

Ilustración 15. Máquina de estados del mago.

El mago comienza en el estado ***spawn*** que como se ha comentado en apartados anteriores realiza la misma función que los enemigos ya comentados.

Cuando cambia al estado ***move*** el enemigo se mantendrá en este estado hasta que se cumplan una de estas dos condiciones:

- La **primera** es que el enemigo detecte un obstáculo, condición para que se cambie a uno de los tres estados de ataque (***attack1***, ***attack2*** o ***attack3***).
- La **segunda condición** es que la vida del enemigo llegue a cero o que el enemigo haya llegado al punto de destino entonces es cuando se cambia al estado ***dead*** donde el enemigo es destruido.

Por otro lado, el enemigo permanecerá en uno de los tres estados de ataque hasta que se destruya el obstáculo al que está atacando o la vida del enemigo llegue a cero. Si se destruye el obstáculo y la vida no es cero entonces se cumple la condición para volver al estado ***move***. Pero si la vida es cero entonces, se cambiará al estado ***dead***.

3.3.4 El dragón

El dragón tiene un comportamiento similar al mago con la diferencia de que el dragón ataca a las torres cuando su vida se ve reducida considerablemente. La máquina de estados que se encarga de su comportamiento es una máquina probabilística inercial que añade pequeños retardos entre los estados.

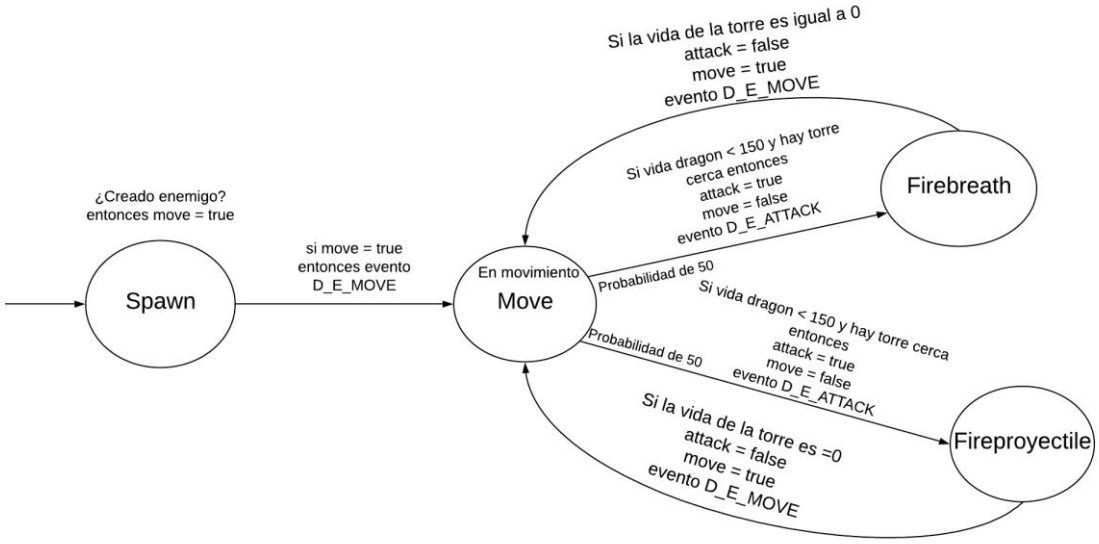

Ilustración 16. Máquina de estados del dragón.

Su comportamiento en el estado ***spawn*** es idéntico que el de todos los enemigos ya mencionados. Cuando el enemigo está en el estado ***move*** y su vida se ve reducida entonces cambia a uno de los dos estados, mediante una variable booleana, que se encargan de atacar a la torre más cercana al enemigo mediante una bola de fuego con el estado ***fireprojectile*** o una llamarada con el estado ***firebreah***. Cuando la torre es destruida la variable booleana se pone a valor falso y esto permite volver al estado ***move*** donde seguirá moviéndose hasta que encuentre otra torre a la que atacar.

4 Planificación

Para el correcto desarrollo del proyecto se ha llevado a cabo una planificación muy necesaria ya que sin ella sería muy complicado llegar a acabar un proyecto.

Para este proyecto se ha estimado una duración de tres meses empezando a finales de febrero de 2019 y terminarlo a finales de julio de 2019.

Para cumplir con los plazos establecidos la planificación se ha dividido en varias partes fundamentales:

4.1 Parte 1 - Lectura y compresión de la API utilizada y búsqueda de recursos

Antes de comenzar el proyecto, previamente, se realizó el documento de diseño o GDD en el que se incluyen los puntos como la descripción de los personajes, los niveles, jugabilidad, ambientación, historia y todo lo relacionado con el diseño de personajes, niveles y de la interfaz. El GDD se encuentra en el anexo de esta memoria.

Después se realizó una búsqueda intensiva por internet de recursos o *assets* que conformarán los elementos del terreno, decoraciones, efectos en los disparos en forma de partículas, los modelos 3D para las torres y los enemigos.

Y por último la lectura y compresión de la documentación de la API para la inteligencia artificial desarrollada por José Alapont Luján.

Para la duración de esta parte se ha previsto de dos semanas para realización de todas las tareas descritas.

4.2 Parte 2 - Programación del videojuego

En esta parte se va a realizar toda la programación del videojuego desde la programación de enemigos hasta la programación de las interfaces.

Esta parte ha sido la más costosa y duradera, pero es en la que más se ha aprendido.

Para esta parte se ha previsto una duración de un mes y medio.

Para la realización de esta parte se han llevado diferentes tareas que se describen a continuación:

- **Creación de un nivel provisional:** para ello se utilizó elementos sencillos como un plano para representar el suelo, esferas para representar a los enemigos y cubos para las plataformas donde el jugador podrá colocar las torres.
- **Creación de los ficheros XML para cada una de las máquinas:** estos documentos se encargan de describir las máquinas de estados que

gestionan el comportamiento de los enemigos. En estos ficheros se incluyen los estados que componen cada una de las máquinas, así como las transiciones, los eventos que permiten la transición de un estado a otro y las acciones que van a realizar en cada uno de los estados.

- **Creación de un sistema de waypoint,** para definir el camino que van a seguir los enemigos durante el transcurso de la partida.
- **Programación de los enemigos** además de su testeo para su correcto funcionamiento. Serán cuatro enemigos.
- **Depuración de posibles errores en la programación de los enemigos.**
- **Programación de las torres o defensas,** mediante un script que se encarga de controlar la detección de los enemigos y su disparo.
- **Programación de la interacción del jugador con el entorno.**
- **Solución a posibles errores en la programación de las interacciones con el entorno.**
- **Programación del movimiento de la cámara,** así como los controles para Windows y Android y su correspondiente testeo en las diferentes plataformas.
- **Creación del menú de pausa.**
- **Creación de la pantalla de victoria:** es la pantalla que aparece cuando el jugador completa el nivel.
- **Creación de la pantalla *Game Over*:** es la pantalla que aparece cuando el jugador se queda sin vidas y no completa el nivel.
- **Creación del menú principal y la pantalla de selección de niveles.**
- **Creación del menú de opciones.**
- **Testeo de los menús en búsqueda de posibles errores.**

4.3 Parte 3- Creación de los niveles definitivos

Al tener toda la programación realizada y su correcto funcionamiento, lo siguiente es ponerse a realizar los dos niveles definitivos que tendrá el videojuego.

La duración de esta parte ha sido de tres semanas y se han realizado las siguientes tareas:

- **Creación de los niveles.** Se ha utilizado un terreno compuesto por texturas para la hierba y la arena.
- **Integración de toda la programación en los niveles creados.**
- **Decoración y detalles para los niveles.** Se han hecho uso de diferentes recursos para la decoración como plantas, árboles además de unas plataformas bien diferenciadas donde el jugador podrá colocar las diferentes torres.
- **Iluminación y la incorporación de niebla.**

- **Inclusión de la canción principal y efectos de sonidos** además de las canciones de cada nivel.
- **Testeo de posibles errores con las colisiones de los objetos,** como por ejemplo las colisiones de los ataques del mago contra los obstáculos.

4.4 Parte 4 - Testeo y redacción de la memoria

Con toda la funcionalidad del proyecto terminada, en esta parte nos centraremos en el testeo y corrección de errores y ajustes finales para el correcto funcionamiento del juego.

La duración de esta parte ha sido de diez semanas repartidas en: dos semanas para el testeo y ocho semanas para la redacción de la memoria.

Para esta parte se han realizado las siguientes tareas:

- **Más decoración en los niveles.**
- **Animación del título del juego.**
- **Exportación del juego a plataformas de escritorio.**
- **Exportación a la plataforma Android.** Se ha realizado un testeo comprobando la interacción del usuario con el entorno como la colocación de las torres en cada una de sus plataformas además se ha comprobado el correcto funcionamiento de los menús y las diferentes pantallas
- **Ajustes del nivel del juego,** así como la vida de cada uno de los enemigos, el daño que reciben de las torres.

A continuación, se muestra una tabla con las diferentes tareas realizadas y sus correspondientes fechas.

Actividad	Fecha Inicio	Fecha Fin
Parte 1 Búsqueda de recursos y compresión de la API	**11/02/2019**	**28/02/2019**
Búsqueda de recursos	11/02/2019	15/02/2019
Lectura y compresión de la API	17/02/2019	21/02/2019
Creación del GDD (descripción de niveles, ambientación..)	24/02/2019	28/02/2019
Parte 2 Programación del videojuego	**04/03/2019**	**19/04/2019**
Creación de un nivel provisional	04/03/2019	04/03/2019
Creación de los ficheros XML para las máquinas de estado	05/03/2019	05/03/2019
Diseño del sistema de waypoint y programación	05/03/2019	07/03/2019
Programación de la IA de los enemigos y depuración de errores	11/03/2019	21/03/2019
Programación de la lógica de las armas y testeo	25/03/2019	28/03/2019
Programación de la interacción del entorno y su testeo	1/04/2019	05/04/2019
Creación de los menús y programación	08/04/2019	11/04/2019
Búsqueda de posibles errores finales en toda la fase de programación	15/04/2019	19/04/2019
Parte 3 Creación de los niveles definitivos	**22/04/2019**	**10/05/2019**
Creación del primer nivel y segundo nivel	22/04/2019	22/04/2019
Integración de toda la programación hecha en los niveles y su testeo	24/04/2019	28/04/2019
Iluminación de los niveles	29/04/2019	30/04/2019
Inclusión de la música y efectos de sonido	01/05/2019	01/05/2019
Testeo de las colisiones con los obstaculos	03/05/2019	03/05/2019
Comprobación final de la programación en los niveles definitivos	06/05/2019	10/05/2019
Parte 4 Ajustes finales y redacción de la memoria	**20/05/2019**	**31/07/2019**
Ajustes nivel del juego y parámetros de los enemigos	20/05/2019	21/05/2019
Exportación a Windows y Android y comprobación de funcionamiento	22/05/2019	26/05/2019
Búsqueda de errores y reparación	27/05/2019	31/05/2019
Redacción de la memoria	03/06/2019	31/07/2019

Ilustración 17. Tabla con el reparto de tiempo de las tareas.

A modo de resumen se muestra un diagrama de Gantt que muestra los cuatro bloques en los que se divide la planificación.

Ilustración 18. Diagrama de Gantt Febrero-mayo

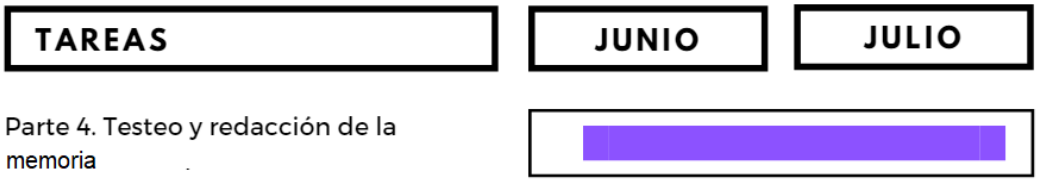

Ilustración 19. Diagrama de Gantt Junio-Julio

5 Implementación

En este apartado se va a explicar la implementación de las partes más importantes del proyecto que son: el funcionamiento de los enemigos, el sistema de *waypoints*, el funcionamiento de las armas y el funcionamiento de los managers del juego.

5.1 Los enemigos

En este apartado se va a explicar la implementación de la IA de los enemigos del juego. Para la implementación de estos enemigos se han hecho uso de tres scripts: un script para controlar la IA, otro para el movimiento del enemigo y por último un script para configurar los parámetros de la vida y el daño.

5.2 La clase AI_Control

Antes de explicar la implementación de los enemigos es necesario hacer uso de la clase abstracta *AI_Control.* Esta clase ha sido reutilizada del código de ejemplo de la propia API. Tiene como propósito facilitar el uso y creación de las máquinas de estados. Esta clase define varias funciones que deben ser implementadas por sus clases hijas. Las funciones que contiene esta clase son las siguientes:

La función *Start* que es la encargada de cargar y crear la correspondiente máquina de estados asociada a cada enemigo. Esta función se implementa en las clases de los enemigos.

La función *Update* se encarga de actualizar la máquina en cada iteración mediante la llamada al método *UpdateFSM* de la máquina. Este método es el encargado de llamar al método especificado en la etiqueta *callback* del documento *XML*, método llamado *CheckEvents*. *CheckEvents* se encarga de generar los eventos para realizar las transiciones además de comprobar el estado en el que se encuentra la máquina. Una vez que se generan dichos eventos la función *UpdateFSM* devuelve una lista de acciones identificadas por un número entero especificado en la clase *Tags*. Esta lista se recorre y en caso de que exista alguna acción a realizar se ejecuta la función *ExecuteAction*.

En la imagen siguiente se puede ver el funcionamiento descrito de la función *Update* y la implementación de la clase *AI_Control*.

```
public abstract class AI_Control : MonoBehaviour {
    //Variable tipo FSM_Machine que sirve para poder actualizar la máquina.
    public FSM_Machine FSM { get; protected set; }
    //Lista de enteros donde se guardan los identificadores de las acciones
    protected List<int> doActionsList;
    //Lista que almacena los eventos.
    protected List<int> eventList = new List<int>();

    protected virtual void Start() {}

    protected virtual void Update()
    {
        doActionsList = FSM.UpdateFSM();
        //Se recorre la lista de acciones
        foreach (int action in doActionsList)
            //Si existe una acción a realizar.
            if (action != Tags.UNKNOWN) ExecuteAction(action);
    }
    //Metodos a implementar en las clases que heredan de esta.
    protected abstract List<int> CheckEvents();
    //Método ExecuteAction
    protected abstract void ExecuteAction(int action);
}
}
```

Ilustración 20. Código de la clase *AI_Control.*

5.2.1 El fantasma

Este enemigo utiliza una máquina de estados determinista descrita en el archivo *EnemyDeterministic.xml*.

La clase *Ghost,* que hereda de *AI_Control,* se encarga de gestionar el comportamiento del fantasma. La clase *Ghost* se fundamenta en varias funciones: la función *Start* que se encarga de inicializar las variables necesarias además de crear la máquina de estados del fantasma.

La función *Update* que se actualiza cada *frame* por segundo y comprueba si el enemigo ha muerto o no. Hay que puntualizar que estas dos funciones descritas son funciones propias del motor y que están en todos los scripts que se creen ya que heredan de la clase *MonoBehaviour.*

El comportamiento de este enemigo se describe de la siguiente manera:

El fantasma nada más crearse entra en su estado inicial que es ***spawn*** ejecutando la función *Spawn* que se encarga de asignar el punto al que tiene que ir de su recorrido mediante la llamada a la función *Init* que se encuentra implementada en la clase *EnemyMovement.* Después mediante el uso de una variable booleana, que se verifica en la función *CheckEvents*, permite añadir a la lista de eventos el evento correspondiente para la transición al estado ***move***, donde la función *ExecuteAction* se encarga de ejecutar la función que corresponde a la acción del estado ***move***.

En la imagen siguiente se puede ver un ejemplo de implementación de la función *ExecuteAction*. Mediante el parámetro *action* (identificador de la acción) se comprueba que acción tiene que realizar la máquina en cada estado.

```
protected override void ExecuteAction(int action)
{
    switch (action)
    {
        case Tags.B_A_SPAWN:
            Spawn();
            break;
        case Tags.B_A_MOVE:
            Move();
            break;
        case Tags.B_A_DIED:
            Died();
            break;
    }
}
```

Ilustración 21. Ejemplo código de la función *ExecuteAction* de la máquina de estados del fantasma.

La función *Move* es la encargada del movimiento del enemigo que se ejecuta hasta que el fantasma ha llegado al punto final del camino o su vida ha llegado a cero. Cuando se dan una de las dos condiciones se activa otra variable booleana encargada de que se cambie al estado ***dead*** donde el fantasma queda eliminado. Cuando se elimina al fantasma o a cualquier enemigo aparecerá en pantalla un mensaje mostrando el dinero que el jugador ha ganado, además de la reproducción de un sonido.

5.2.2 El golem

La máquina que se encarga del comportamiento de este enemigo es una máquina probabilística y está descrita en *GolemProbabilistic.xml.*

```
<Transitions>
    <Transition>
    <T_Name>GO_TO_MOVE</T_Name>
    <T_Origin>GO_SPAWN</T_Origin>
    <T_Destination>GO_MOVE</T_Destination>
    <T_Action>NULL</T_Action>
    <Events>
      <Event>
        <ID>GO_E_MOVE</ID>
        <Type>BASIC</Type>
      </Event>
    </Events>
    <Probability>50</Probability>
  </Transition>
  <Transition>
    <T_Name>GO_TO_SPEED</T_Name>
    <T_Origin>GO_SPAWN</T_Origin>
    <T_Destination>GO_SPEED</T_Destination>
    <T_Action>NULL</T_Action>
    <Events>
      <Event>
        <ID>GO_E_MOVE</ID>
        <Type>BASIC</Type>
      </Event>
    </Events>
    <Probability>70</Probability>
  </Transition>
  <Transition>
```

CÓDIGO XML TRANSICIONES GOLEM

Ilustración 22. Código *XML* (máquina golem) de las transiciones a los estados *move* y *speed*.

En la imagen adjunta se puede ver como se ha implementado las transiciones a los estados ***move*** y ***speed*** y los valores que tienen asignados en el elemento *probability* de la etiqueta *Transition*. Los demás elementos dentro de *Transition* corresponden al nombre de la transición (T_Name), el estado origen de la transición y el estado destino (*T_Origin* y *T_Destination*). Si se realiza alguna acción durante la transición se especifica en el elemento *T_Action*, en caso contrario se pone el valor *null.* La etiqueta Events se encarga de especificar el evento que se produce para esa transición y se identifica por un nombre y el tipo de evento.

La clase que implementa la lógica de este enemigo es *Golem*. Al contrario que el fantasma, el golem es capaz de detectar los obstáculos que encuentre a lo largo de su trayecto. Para que el enemigo pueda detectar estos obstáculos se ha hecho uso de la función *RayCast* de Unity. Esta función se encarga de proyectar un rayo que permite detectar las colisiones con los objetos del entorno utilizando etiquetas o *tags* para identificar a los objetos con los que colisiona. En este caso los obstáculos llevan la etiqueta llamada *obstacule* y esto permite detectar si el rayo colisiona con un muro o no. En caso de que colisione con un obstáculo entonces la variable booleana asociada se pone a valor cierto y esto permite cambiar al estado ***attack***. El enemigo se mantendrá atacando hasta que el obstáculo haya sido destruido o la vida del enemigo llegue a cero. Si obstáculo

ha sido destruido entonces se cambiará al estado ***move*** o al estado ***speed*** dependiendo de la máquina ya que tiene asignada una probabilidad de 50% en estas dos transiciones.

En cuanto al movimiento del enemigo sigue una lógica similar a la del fantasma con la diferencia de que el estado inicial tiene dos transiciones con una probabilidad asignada que tiene un valor de 50% para cada transición. Esto permite que la variable de la velocidad de movimiento del enemigo se vea alterada dependiendo si cambia al estado ***move*** que ejecuta la función *Move* con una velocidad normal (con un valor de 7) o al estado ***speed*** donde se ejecuta la función *Speed* encargada de aumentar la variable velocidad a un valor de 14.

5.2.3 El mago

El mago cuenta con una máquina probabilística descrita en el fichero *WizardProbabilistic.xml.* Este enemigo se caracteriza por tener múltiples ataques que se ejecutan en diferentes estados de la máquina. Las transiciones a los estados de los ataques tienen una probabilidad que se indica en la etiqueta *probability* del fichero *XML*.

Se puede observar en la imagen como cada transición tiene una probabilidad. Para ir del estado ***move*** al estado ***attack1*** tiene una probabilidad de 70%. Del estado ***move*** al estado ***attack2*** tiene una probabilidad de 50% y para ir al estado ***attack3*** tiene una probabilidad de 60%. Estos valores se han ajustado para evitar que no se ejecute siempre el mismo estado, aunque a veces puede llegar a repetirse.

```
<Transition>
  <T_Name>W_TO_ATTACK1</T_Name>
  <T_Origin>W_MOVE</T_Origin>
  <T_Destination>W_ATTACK1</T_Destination>
  <T_Action>NULL</T_Action>
  <Events>
    <Event>
      <ID>W_E_ATTACK</ID>
      <Type>BASSIC</Type>
    </Event>
  </Events>
  <Probability>70</Probability>
</Transition>
<Transition>
  <T_Name>W_TO_ATTACK2</T_Name>
  <T_Origin>W_MOVE</T_Origin>
  <T_Destination>W_ATTACK2</T_Destination>
  <T_Action>NULL</T_Action>
  <Events>
    <Event>
      <ID>W_E_ATTACK</ID>
      <Type>BASSIC</Type>
    </Event>
  </Events>
  <Probability>50</Probability>
</Transition>
<Transition>
  <T_Name>W_TO_ATTACK3</T_Name>
  <T_Origin>W_MOVE</T_Origin>
  <T_Destination>W_ATTACK3</T_Destination>
  <T_Action>NULL</T_Action>
  <Events>
    <Event>
      <ID>W_E_ATTACK</ID>
      <Type>BASSIC</Type>
    </Event>
  </Events>
  <Probability>60</Probability>
</Transition>
```

CÓDIGO XML TRANSICIONES MÁQUINA MAGO

Ilustración 23. Implementación de las transiciones del estado move (máquina mago) a los estados de ataque.

Estos ataques son controlados por una variable que controla la velocidad de los disparos y otra variable que controla el daño que producen los disparos. El primer ataque dispara una vez por segundo, el segundo dispara dos veces por segundo y el tercero dispara tres veces por segundo.

En la imagen (ilustración 24) se muestran las tres funciones encargadas de ejecutar cada uno de los ataques. Dependiendo del estado en el que se encuentre el enemigo, se llamará a una de estas tres funciones que a su vez llama a una función común llamada *Attack* pasando por parámetro el nombre que identifica al disparo mediante un *array* de *string* y el daño que realiza cada uno de los ataques mediante un array de tipo *float*. Toda la lógica de este enemigo se encuentra implementada en el script *Wizard.*

```
public void Attack1()
{
    //Ejecuta la animación projectile Attack
    controller.SetTrigger("Projectile Attack");
    //Llama a la función attack pasando el nombre que identifica al disparo y el daño asociado.
    Attack(nameObject[0],damage[0]);

}
public void Attack2()
{
    //Ejecuta la animación projectile Attack
    controller.SetTrigger("Projectile Attack");
    //Llama a la función attack pasando el nombre que identifica al disparo y el daño asociado.
    Attack(nameObject[1],damage[1]);
}
public void Attack3()
{
    //Ejecuta la animación projectile Attack
    controller.SetTrigger("Projectile Attack");
    //Llama a la función attack pasando el nombre que identifica al disparo y el daño asociado.
    Attack(nameObject[2], damage[2]);
}
```

Ilustración 24. Funciones de los ataques de los ataques del mago.

5.2.4 El dragón

El dragón utiliza una máquina probabilística inercial que permite añadir latencias en la ejecución de cada uno de los estados. Su máquina se encuentra descrita en *DragonProbabilistic.xml.* El comportamiento de este enemigo es muy similar al de los anteriores enemigos en cuanto a movimiento, pero en el ataque hay diferencias ya que el ataque del enemigo no se basa en destruir los obstáculos del camino, sino que centra todo su ataque en las torres.

Para ello se ha implementado una función llamada *Target* que se encarga de localizar a la torre más cercana al enemigo. El enemigo realiza su ataque cuando su vida se ve reducida a la mitad y es cuando la función *Target* asigna como objetivo del ataque a la torre más cercana. Al tener el enemigo asignada una máquina probabilística le permite realizar dos ataques diferentes con una probabilidad de 50% para cada transición. El script que se encarga de todo el comportamiento es *Dragon.cs.*

```
<FSMtype Probabilistic="YES">INERTIAL</FSMtype> <!--Probabilistic has to be "YES" (PROBABILISTIC) or "NO"
<FSMid>DragonProbabilistic</FSMid>
<Fsm>
  <Callback>CheckEvents</Callback> <!--Method for events that concern to this FSM -->
  <States>
    <State Initial="YES">
      <S_Name>D_SPAWN</S_Name>
      <S_Action>D_A_SPAWN</S_Action>
      <S_inAction>NULL</S_inAction>
      <S_outAction>NULL</S_outAction>
      <S_Fsm></S_Fsm>
      <S_Latency>1000</S_Latency>
    </State>
    <State Initial="NO">
      <S_Name>D_MOVE</S_Name>
      <S_Action>D_A_MOVE</S_Action>
      <S_inAction>NULL</S_inAction>
      <S_outAction>NULL</S_outAction>
      <S_Fsm></S_Fsm>
      <S_Latency>500</S_Latency>
    </State>
    <State Initial="NO">
      <S_Name>D_FIRE_BREATH</S_Name>
      <S_Action>D_A_FIRE_BREATH</S_Action>
      <S_inAction>NULL</S_inAction>
      <S_outAction>NULL</S_outAction>
      <S_Fsm></S_Fsm>
      <S_Latency>500</S_Latency>
    </State>
    <State Initial="NO">
      <S_Name>D_PROJECTILE_ATTACK</S_Name>
      <S_Action>D_A_PROJECTILE_ATTACK</S_Action>
      <S_inAction>NULL</S_inAction>
      <S_outAction>NULL</S_outAction>
      <S_Fsm></S_Fsm>
      <S_Latency>500</S_Latency>
    </State>
  </States>
```

CÓDIGO XML MÁQUINA DE ESTADOS DEL DRAGÓN

Ilustración 25. Código XML de los estados (máquina del dragón)

Se puede observar en la imagen adjunta los estados del dragón definidos en el fichero *XML*. El elemento *latency* indica el retardo en milisegundos. En el apéndice de este documento se puede encontrar información más detallada de los elementos de la etiqueta *State*.

En la siguiente imagen se muestra cómo se han implementado las transiciones en el fichero *XML*. Con la etiqueta *probability* se puede asignar una probabilidad para que se produzca esa transición.

```
<Transition>
  <T_Name>D_TO_FIRE_BREATH</T_Name>
  <T_Origin>D_MOVE</T_Origin>
  <T_Destination>D_FIRE_BREATH</T_Destination>
  <T_Action>NULL</T_Action>
  <Events>
    <Event>
      <ID>D_E_ATTACK</ID>
      <Type>BASIC</Type>
    </Event>
  </Events>
  <Probability>50</Probability>
</Transition>
<Transition>
  <T_Name>D_TO_MOVE</T_Name>
  <T_Origin>D_FIRE_BREATH</T_Origin>
  <T_Destination>D_MOVE</T_Destination>
  <T_Action>NULL</T_Action>
  <Events>
    <Event>
      <ID>D_E_MOVE</ID>
      <Type>BASIC</Type>
    </Event>
  </Events>
  <Probability>100</Probability>
</Transition>
<Transition>
  <T_Name>D_TO_PROJECTILE_ATTACK</T_Name>
  <T_Origin>D_MOVE</T_Origin>
  <T_Destination>D_PROJECTILE_ATTACK</T_Destination>
  <T_Action>NULL</T_Action>
  <Events>
    <Event>
      <ID>D_E_ATTACK</ID>
      <Type>BASIC</Type>
    </Event>
  </Events>
  <Probability>50</Probability>
</Transition>
<Transition>
  <T_Name>D_TO_MOVE</T_Name>
  <T_Origin>D_PROJECTILE_ATTACK</T_Origin>
  <T_Destination>D_MOVE</T_Destination>
  <T_Action>NULL</T_Action>
  <Events>
```

CÓDIGO XML TRANSICIONES DE LA MÁQUINA DE ESTADOS DEL DRAGÓN

Ilustración 26. Implementación en *XML* de una parte de las transiciones de la máquina del dragón.

5.3 Sistema de waypoint

Este sistema se basa en crear diferentes puntos distribuidos por el nivel para formar un camino para que los enemigos lo sigan. Estos puntos son *GameObject* vacíos es decir que no son visibles al jugador y se añaden como hijos a un *GameObject* padre donde se le asigna un script. El script creado para ello se llama Waypoints que contiene una variable estática que es array de tipo *Transform* que almacena todos los puntos con el componente *Transform*. Esta variable es pública y estática para que pueda ser accesible desde otras clases sin necesidad de crear una instancia de la clase. También en esta clase se ha implementado una función que permite que el enemigo siempre esté mirando al punto del camino que tiene como objetivo. Esta función se llama *Turn* que calcula la dirección hacia donde se va a girar y a partir de ella el enemigo gira.

```
public class Waypoints : MonoBehaviour
{
    public static Transform[] points;
    //Este metodo se ejecuta mucho antes que el start
    private void Awake()
    {
        //Crea un array de Transform donde guarda cada punto por el que el enemigo tiene que pasar
        points = new Transform[transform.childCount];
        for (int i=0; i<points.Length; i++)
        {
            //Obtiene todos los hijos del objeto padre
            points[i] = transform.GetChild(i);
        }
    }
    //Metodo que sirve para girar y mirar hacia el objetivo
    public static void turn(Transform target, Transform origin, float speed)
    {
        //Calcula la direccion
        Vector3 dir = target.position - origin.position;
        //Rota hacia esa direccion
        Quaternion rotation = Quaternion.LookRotation(dir);
        //Y guarda la rotacion.
        origin.rotation = Quaternion.Slerp(origin.rotation, rotation, 5 * Time.deltaTime);
    }

}
```

Ilustración 27 Código de la clase *Waypoints.*

Con la clase Waypoints terminada es hora de pasar con el script encargado de todo el movimiento de los enemigos. Este script llamado *EnemyMovement* se le asigna a cada uno de los enemigos y se centra en tres funciones fundamentales. La primera llamada *Init* se encarga de inicializar el array ya explicado en el script *Waypoints,* es decir asigna como objetivo el punto almacenado en la posición cero del array. Seguidamente se llama a la función *Turn* para que el enemigo gire hacia ese punto.

La siguiente función se llama *Movement* y se encarga de actualizar la posición actual mediante la función *MoveTowards* (de la clase *Vector3* perteneciente a Unity) donde se le pasa como parámetro la posición actual, la posición del objetivo y una variable velocidad que se multiplica por la función *DeltaTime* para darle movimiento al enemigo. Antes de actualizar la posición se comprueba si se ha llegado al punto objetivo mediante el cálculo de la distancia entre la posición actual y la posición del objetivo. Cuando esta distancia es

inferior 0.2 se llama a la siguiente función *NextPoint.* La función se encarga de comprobar que el punto actual no sea el último almacenado en el array, de ser así se realizan una serie de comprobaciones utilizando las referencias de cada uno de los scripts de los enemigos permitiendo acceder a una variable booleana que permite indicar que el enemigo debe ser destruido y se le resta una unidad a la variable encargada de la gestión de las vidas del jugador almacenada en el script *Player.* En caso de que el punto actual no fuese el último se actualiza el índice del array sumando una unidad y se asigna el siguiente punto como objetivo.

5.4 Armas

La lógica de las armas está implementada en el script *Turret.* Este script está compuesto por varias funciones entre ellas la más importante es la función *Target* que realiza la búsqueda del objetivo más cercano a la torre. Lo primero que hace es buscar todos los objetos que hay en la escena que tengan la etiqueta *enemy* mediante la función *FindGameObjectsWithTag* que devuelve un array de *GameObject* y se inicializan las variables necesarias para el cálculo de la distancia y almacenar el objetivo encontrado. Después se recorre el *array* y se calcula la distancia entre la posición de la torre quedándose solamente con la distancia más pequeña entre el objetivo y la torre. Por último, se comprueba que la distancia encontrada está dentro de un rango asignado. Si se cumple entonces se asigna el objetivo encontrado y se dispara hasta que el objetivo sale del rango. En caso contrario se asigna a la variable objetivo el valor *null* y se vuelve a buscar el siguiente objetivo a disparar.

Además de la función target también se han implementado otras funciones secundarias como por ejemplo *LookAt* que su misión es seguir al objetivo que se quiere disparar y la función *Shoot* que realiza la instanciación de las balas en la escena. Para controlar los disparos se ha utilizado una variable que gestiona la velocidad de disparo, es decir cuantas balas por segundo se instancian cada vez que se realiza un disparo. Esta variable se llama *fireRate* y se modifica cuando el jugador realiza una mejora a la torre.

La instanciación de las balas y destrucción de ellas es un proceso costoso y puede llegar a ralentizar el juego. Es por ello que se ha hecho uso del patrón de diseño *ObjectPooling* mediante el cual se pretende reducir el consumo de memoria RAM y a su vez mejorar el uso de la CPU. Esto se realiza mediante la reutilización de los objetos instanciados que, en vez de ser destruidos, son almacenados temporalmente para volver a utilizarse. Para la implementación de este patrón han sido necesario la implementación de dos scripts. El primer script se llama *Pool* y define los atributos de los objetos que se van a instanciar como son el número de objetos, el nombre que identifica al objeto y el propio objeto.

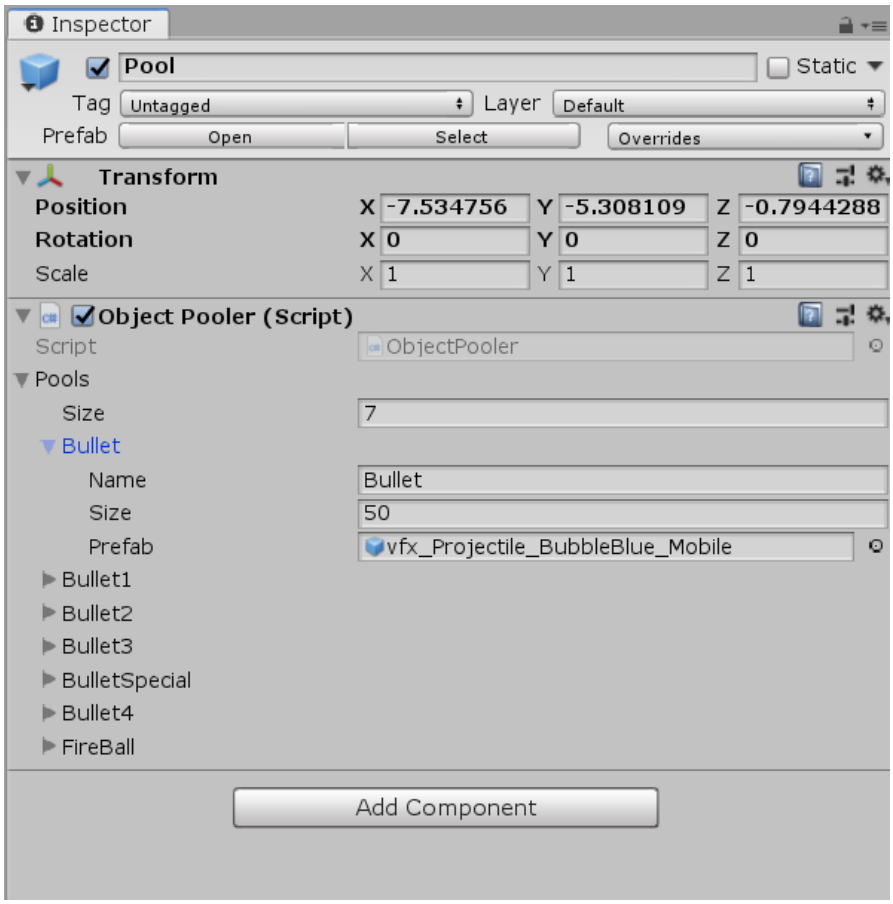

Ilustración 28. Vista en el inspector del script *ObjectPooler*.

En la imagen se puede ver el siguiente script *ObjectPooler* donde se muestran los atributos públicos en el inspector. El atributo *Pools* es una lista de objetos donde se agrupan los objetos que se van a instanciar y que se identifican por el nombre, el número de objetos a instanciar y el objeto a instanciar.

El script *ObjectPooler* funciona de la siguiente manera: en el método *Start* se inicializa un diccionario que tiene como clave el nombre del objeto y como valor una cola de los objetos que se van a utilizar. Mediante el primer bucle se recorre la lista de objetos y el segundo bucle se encarga de instanciar tantos objetos como indique el atributo *size* del objeto *pool*.

Una vez instanciado el objeto se utiliza el método *SetActive* que se encarga de activar o desactivar un objeto en la escena. En este caso se pasa como parámetro el valor false y después el objeto se almacena en la cola. Finalmente se guarda en el diccionario el nombre del objeto como clave y la cola donde están todos los objetos instanciados.

```
//Se encarga de instanciar todos los objetos en la escena al iniciar el juego y prepararlos para su uso
private void Start()
{
    //Se crea el diccionario que tiene como clave un string y como valor una cola de GameObject
    poolDictionary = new Dictionary<string, Queue<GameObject>>();

    foreach (Pool pool in pools)
    {
        Queue<GameObject> objectPool = new Queue<GameObject>();
        for(int i=0; i<pool.size; i++)
        {
            //Se instancia el objeto
            GameObject obj = Instantiate(pool.prefab);
            //Se desctiva el objeto
            obj.SetActive(false);
            objectPool.Enqueue(obj);
        }
        //Se añade al diccionario la clave que es el nombre del objeto y como valor el objeto
        poolDictionary.Add(pool.name, objectPool);
    }
}
```

Ilustración 29. Método *Start* de la clase *ObjectPooler*.

Además de este método también se ha implementado el método *Spawn* que es el encargado de poder utilizar los objetos que se han instanciado y guardado en el diccionario. Su explicación se puede ver en la siguiente imagen.

```
//Metodo que devuelve un gameobject y tiene como parametro un string para identificar el nombre, un vector3 para la posicon
//y un Quaternion para la rotación
public GameObject Spawn(string name, Vector3 position, Quaternion rotation)
{
    //Si no contiene la clave no se hace nada
    if (!poolDictionary.ContainsKey(name))
        return null;
    //Se desencola el objeto
    GameObject obj = poolDictionary[name].Dequeue();
    //Se activa
    obj.SetActive(true);
    //Se guarda su posicion
    obj.transform.position = position;
    //Se guarda su rotacion
    obj.transform.rotation = rotation;
    //Se vuelve a enconlar
    poolDictionary[name].Enqueue(obj);
    //Se devuelve el objeto obtenido.
    return obj;
}
```

Ilustración 30. Método *Spawn* de la clase *ObjectPooler.*

Su funcionalidad es la siguiente: primero se comprueba si la clave existe en el diccionario de objetos. Si existe la clave se desencola el objeto y se activa, además de establecer una posición y la rotación. En caso contrario se devuelve el valor *null*. Para finalizar, se vuelve a encolar en el diccionario y se devuelve el objeto para que pueda ser utilizado. Este método se llama cuando se producen los disparos de las torres y de los enemigos.

5.5 Managers

Estas clases se encargan de gestionar el flujo del juego además de controlar el audio y la persistencia de datos. Gracias a ellas permiten escalar el proyecto fácilmente, ya que es muy importante en un videojuego a la hora de introducir nuevas características en el futuro. Todos los managers de este proyecto hacen uso del patrón de diseño *Singleton* que se encarga de asegurar que solamente existe una única instancia del objeto en cuestión y permitiendo un acceso global a dicha instancia para que pueda ser utilizada en cualquier script.

GameManager se encarga de gestionar los diferentes estados en los que se encuentra el juego: pausa, en juego, menú principal, menú de nivel completado, menú fin del juego. Los estados se guardan como constantes en un tipo *enum*[33] y mediante una función implementada en el script se encarga de realizar la acción correspondiente dependiendo en el estado en el que se encuentre el juego.

BuildManager este script implementa las funciones necesarias para la interacción del usuario con la interfaz de la tienda. Estas funciones indican la selección del arma que se quiere colocar en la torre.

AudioManager es el encargado de gestionar el audio del juego. Implementa diferentes funciones que permiten reproducir o detener la música del juego y los efectos de sonido.

[33] Se utiliza para declarar una enumeración, un tipo distinto que consiste en un conjunto de constantes con nombre denominado lista de enumeradores.

6 Conclusiones

Para finalizar esta memoria, se exponen las conclusiones extraídas de la realización de este proyecto.

En el apartado técnico, este proyecto me ha servido para ampliar mis conocimientos y aprender nuevas características sobre el motor Unity. También me ha ayudado a mejorar en el apartado de programación en el uso de patrones de diseño estudiados en la carrera como es el *Singleton* o el *ObjectPool*.

Por otro lado, la utilización de la API, me ha servido como un ejemplo de aplicación del uso de máquinas de estados para hacer la inteligencia artificial. Una rama que me ha resultado interesante durante la carrera y que gracias a este proyecto he conseguido comprender mejor el funcionamiento de las máquinas de estados y su aplicación. Los resultados obtenidos con la API han sido bastante satisfactorios porque los comportamientos de los personajes cumplen con lo esperado.

En cambio, en el aspecto personal, este proyecto me ha enseñado a afrontar las dificultades que tiene diseñar un videojuego desde cero y utilizar una *API* que no conocía. El proceso de redacción de la memoria ha sido a nivel personal el más costoso por el estrés y frustración que supone bloquearte a la hora de redactar. Pero finalmente he conseguido salir adelante con tranquilidad y paciencia. A pesar de todo ha sido una experiencia muy positiva donde he aprendido a superar esas dificultades y seguramente me va ayudar a la hora de volver afrontar nuevos proyectos.

Con respecto a la planificación, se estimó una duración de 25 semanas finalmente han sido 27 semanas. Ese tiempo adicional se ha empleado para completar lo mejor posible la redacción de este documento. Además, se han realizado ajustes en la dificultad del juego, modificaciones en el daño de las armas y ajustes en la colocación de los menús y botones.

Por último, los resultados de este proyecto han sido bastante satisfactorios cumpliéndose la gran mayoría de los objetivos propuestos, aunque solo se han completado dos niveles de los seis que se tenían previsto desarrollar.

7 Trabajos futuros

El juego se diseñó como un prototipo, que presenta las mecánicas del género Tower Defense y sienta las bases para que el juego pueda ser ampliado y más completo.

La estructura del proyecto está preparada para que se implementen mejoras que por la falta de tiempo no se han podido llevar a cabo. Como son la inclusión de nuevos modos de juego, enemigos, niveles, nuevas armas y el rediseño de la interfaz gráfica. Mejoras que serían muy costosas para una única persona y por ello sería necesario de un equipo para poder llevarlas a cabo y completar el juego en el futuro. El proyecto está preparado para que estas mejoras se realicen con el menor número de cambios.

En lo que respecta a la API utilizada, propongo mejoras necesarias para la reducción de tiempo a la hora de definir las máquinas de estados. Considero que definir máquinas de estados con un número reducido de estados es bastante fácil de implementar y el tiempo que se dedica a ello es reducido. Pero en el momento en que esas máquinas de estados tienen un número considerable de estados el tiempo dedicado para definir los ficheros *XML* puede llegar a ser muy elevado. Por eso sería necesario el uso de una interfaz intuitiva para que agilice todo el proceso de definición.

Finalmente, sería necesario automatizar la clase *Tags* o al menos la función *StringToTags*. Su implementación es un proceso tedioso y repetitivo, teniendo que realizarlo con cada modificación de la máquina.

8 Agradecimientos

A mi tutor Ramón Pascual Mollá Vayá, por la gran libertad que me ha dado para realizar este proyecto, por su paciencia y consejos en los momentos más difíciles. Agradezco la oportunidad de usar la API de José Alapont ya que me ha permitido aprender y afianzar los conocimientos existentes en las máquinas de estados.

Bibliografía

[1] Rollings, Andrew; Ernest Adams (2003). *Andrew Rollings and Ernest Adams on Game Design*. New Riders Publishing. pp. 321-345.

[2] Emrich, A. (1993). MicroProse' Strategic Space Opera is Rated XXXX! ComputerGaming World, 110, pp. 92-93.

[3] Suarez, Sandy; Bautista, Raúl. (2013). *Crea un mundo y diviértete en él "tipología de los videojuegos".* pp. 61-68

[4] Gil Juárez, Adriana; Vida Mombiela, Tere. (2007) *Los videojuegos.*

[5] Gavrilova, Nuangjumnong (22 de enero de 2016). «The Effects of Gameplay on Leadership Behaviors: An Empirical Study on Leadership Behaviors and Roles in Multiplayer Online Battle Arena Games». *Transactions on Computational Science XXVI: Special Issue on Cyberworlds and Cybersecurity* (en inglés). Springer. p. 147.

Apéndice

Introducción a Unity

Unity se fundamenta principalmente en los *GameObjects.* Estos objetos son los elementos que contienen una escena en Unity. Una escena es aquella donde se representan todos los elementos que componen nuestro juego, así como los personajes, la cámara, el terreno… todos ellos están dentro de la misma.

Todo *GameObject* que se encuentra dentro de la escena está compuesto por un componente llamado *Transform*. Este componente es único y lo poseen todos los *GameObjects*. El componente *Transform* se encarga de dar la posición, rotación y escala del objeto representado en la escena. También existen otros componentes como el *Renderer* que se encarga de hacer visible a los objetos en la escena, además de dar un color y una textura, el *Rigidbody* y el *Collider*, que gestionan las colisiones con otros elementos y las características de la física gestionada por el propio motor. Otro componente importante el *Camera*, toda escena en Unity tiene un objeto llamado *MainCamera* que se encarga de renderizar la escena.

Interfaz

La interfaz de Unity es muy intuitiva y consta de una serie de ventanas principales que el usuario puede colocar a su gusto. Las ventanas principales son las siguientes:

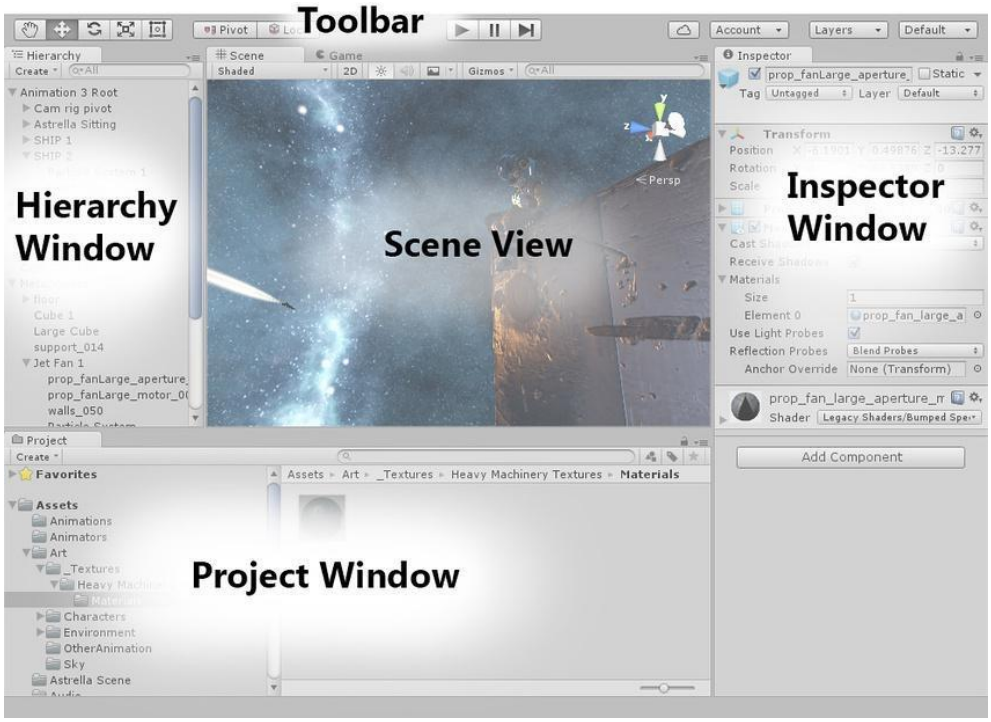

Ilustración 31. Interfaz de Unity.

La **ventana del proyecto** consta de un explorador de archivos que nos permite añadir, eliminar, organizar elementos todos ellos llamados *assets*. Esta ventana consta de dos pestañas, en la izquierda se encuentra toda la jerarquía de carpetas de nuestro proyecto cuyo directorio raíz es *Asset*. Este directorio se sincroniza en tiempo real con la carpeta del sistema: si se realiza cualquier cambio, Unity actualizará el proyecto. En la pestaña de la derecha se encuentran todos los archivos que aparecen dentro la carpeta seleccionada de la jerarquía de carpetas. También cuenta con un buscador para facilitar la búsqueda de archivos o carpetas en caso de que el usuario no recuerde dónde se encuentran.

La **ventana de Hierarchy (Jerarquía)** contiene una lista de cada *GameObject* (también llamados "*Objects*") en la escena. Estos objetos pueden ser instancias directas de archivos de *assets*, y otras son instancias de *prefabs*, que son objetos personalizados que una vez creados en la escena al desplazarse a la ventana de proyecto se convierten en *prefabs.* Del mismo modo cuando el usuario importa modelos 3D estos son prefabs que cuando son desplazados a la escena se convierten en GameObjects y estos aparecen en la ventana jerarquía en orden de creación. Esta ventana también permite crear objetos y formar una jerarquía con solo arrastrar el objeto hijo al objeto padre.

La **ventana de escena** es la vista interactiva del mundo que se está creando. La vista escena se utiliza para seleccionar y posicionar paisajes, personajes, luces y todos los demás objetos que componen el juego.

La **ventana de juego** muestra el resultado representado en la escena mediante una o más cámaras que se encargan de renderizar los objetos que hay en ella. Desde el editor se pueden realizar diferentes acciones como poder pausar el juego, ajustar el juego a distintas resoluciones o ejecutar el juego *frame* a *frame* para mayor detalle.

El **inspector** se encarga de mostrar de forma detallada toda la información asociada al *GameObject* seleccionado en la escena o en la ventana jerarquía. Incluye información de todos los componentes asociados y sus propiedades. También permite modificar la funcionalidad del *GameObject* y añadir nuevos componentes.

La **ventana de animación,** donde se pueden realizar clip de animación desde cero. Consta de una línea del tiempo donde se puede crear cualquier animación.

La **ventana del animator** nos permite crear máquinas de estados y asignar animaciones a cada uno de ellos.

Scripts

Los scripts son un componente más de Unity, se crean para darle un comportamiento a los objetos que se crean en la escena de nuestro juego. Para crear un script solo hay que hacer click con el botón derecho del ratón dentro de la ventana proyecto y se desplegará un menú con varias opciones. La primera opción es *create* y desde ahí se despliega otro menú donde aparece la opción *C#* script. Una vez creado aparecerá un fichero con el símbolo de *C#* pidiendo que se introduzca un nombre al script. Es importante saber que el nombre del script es el nombre de la clase que se acaba de crear y es por ello que tiene que coincidir ya que de no ser así se producirá un error de compilación.

Otra forma de crear un script es seleccionando el objeto al que se le quiere asignar el script y en la ventana inspector aparece un botón llamado *add component* el cual si se hace *click* se despliega una serie de opciones entre las cuales se encuentra la opción script.

Todos los scripts creados heredan de la clase MonoBehaviour que contiene todas las clases y funciones necesarias para poder programar nuestro juego. El ciclo de vida de un script se compone de varias funciones principales que son las siguientes:

- ***Awake*.** Este método se ejecuta antes de cualquier función *Start* y también justo después de que un *prefab* es instanciado. Hay que tener en cuenta que si un *GameObject* está inactivo durante el comienzo, *Awake* no es llamado hasta que se vuelva activo.
- ***Start*.** Es llamado antes de la primera actualización de frame sólo si la instancia del script está activada. Este método se ejecuta después del *Awake.* Por defecto *Start* siempre aparece cuando se crea un script. Se suele utilizar para la inicialización de variables.
- ***Update*.** Se llama una vez por *frame*. Es la función principal para las actualizaciones de *frames*. Esta función aparece siempre por defecto cuando se crea un script. El tiempo que tarda en ejecutarse puede variar dependiendo de las imágenes por segundo en ese preciso instante.
- ***LateUpdate*.** Es llamada una vez por *frame* después de que *Update* haya finalizado. Cualquier cálculo que sea realizado en *Update* será completado cuando *LateUpdat*e comience. Un uso común de *LateUpdate* sería una cámara en tercera persona que sigue.
- ***FixedUpdate*.** Es invocado en cada iteración del bucle de físicas. Esta función se utiliza para todo lo relacionado con las físicas de nuestro juego. La diferencia con la función *Update* es que esta función sí que tiene un tiempo fijo entre llamada. Se debe tener en cuenta que la sobrecarga de esta función puede repercutir negativamente en el rendimiento de nuestro juego.

Lo explicado anteriormente no es todo lo que puede ofrecer Unity. Unity tiene una gran cantidad de funciones y de características que se pueden encontrar en la documentación oficial[34] y en su página web.

API para la gestión de máquinas de estados

Como ya se ha mencionado, la IA de los enemigos se hará usando la *API* de José Alapont.

Esta herramienta permite crear y gestionar máquinas de estados de diferentes tipos de una forma sencilla.

Creación de las máquinas de estados mediante XML

En primer lugar, hay que elaborar un documento XML donde se definen las características que va a tener la máquina en cuestión, como los estados que va a tener, las transiciones y las acciones que se llevarán a cabo en los mismos. Para ello partimos de las plantillas proporcionadas por la *API* y también el uso de la herramienta *FSM_Parser*[35] que facilita el proceso de carga de nuestro fichero *XML.*

Lo primero que debemos especificar es el tipo de máquina que queramos implementar entre los siguientes tipos:

- **Clásica-Determinista:** Es la máquina más simple y general.
- **Clásica-Probabilística:** Esta máquina permite introducir probabilidades de activación entre las transiciones de un estado a otro. Es una máquina indeterminista.
- **Inercial:** Permite introducir cierta latencia a los estados de la máquina.
- **Basada en pilas:** Permite la interrupción de la ejecución de un estado cuando la acción entrante es más prioritaria y después volver al estado en que se encontraba anteriormente. Esto permite otorgar cierta memoria a la máquina.
- **Máquina con estados concurrentes:** Esta máquina permite ejecutar varios estados al mismo tiempo.

Una vez elegido el tipo de máquina a implementar lo siguiente que encontraremos es la etiqueta “States” donde se encuentra cada uno de los

[34] Documentación oficial de unity: https://docs.unity3d.com/es/current/Manual/UnityManual.html

[35] FSM: Finite State Machine. Máquina de estados finitos.

estados de la máquina. Dentro de la etiqueta “*States*” encontramos la etiqueta “*State*” donde quedan definidos cada uno de los estados y es hija de “*States*”.

Haciendo uso del atributo “initial” podemos especificar cuál es el estado inicial. A continuación, se muestran los elementos que se encuentran dentro de la etiqueta “*State*”:

- **S_Name:** Indica el nombre que identifica al estado.
- **S_Action:** Indica el nombre de la acción que se realizará mientras la máquina se encuentre en el estado. Este valor puede ser nulo.
- **S_inAction:** Indica el nombre de la acción que se realizará al entrar a un estado. Puede ser nulo.
- **S_outAction:** Indica el nombre de la acción que se realizará al salir del estado.
- **S_Fsm:** Enlaza a una máquina de estados si se desea crear estados jerárquicos. Se puede dejar vacío.

Con los estados definidos, lo siguiente que nos encontraremos son las transiciones que se identifican con la etiqueta “Transitions” y tienen como elemento “Transition” donde se describen los siguientes atributos:

- **T_Name:** Nombre de la transición.
- **T_Origin:** Indica el nombre del estado origen de la transición.
- **T_Destination:** Indica el nombre del estado destino de la transición.
- **T_Action:** Indica el nombre de la acción que se realizará en la transición. Puede ser un valor nulo.

Además de todos los elementos descritos anteriormente encontramos el elemento **Events** que alberga una lista de eventos, cada uno definido por la etiqueta “Events” con los siguientes elementos:

- **ID:** Es el nombre que identifica al evento.
- **Type:** Es el tipo de evento, que puede ser “BASIC”, es el elemento por defecto en todas las máquinas de estados o “STACKABLE” que se emplea en las máquinas basadas en pilas para indicar que apila al estado origen.

La clase Tags.

La clase Tags es una clase estática donde se encuentran definidas las constantes numéricas que identifican los estados, las transiciones, las acciones y los eventos de cada una de las máquinas. En esta clase se encuentra el método *StringToTag* que hay que completar con cada una de estas constantes. Hay que destacar que estas constantes son únicas y deben coincidir con los nombres que aparecen en los ficheros XML.

La carga de las máquinas

Tras los pasos anteriores, toca el proceso de carga de las máquinas para su correcto funcionamiento. Para el proceso de carga se ha usado la herramienta *FSM_parser*. Para ello se ha usado la clase *pbFSM_Manager* que crea una instancia de *FSM_parser* y busca todos los archivos xml que se encuentran en el directorio StreamingAssets de Unity y los carga al principio de la ejecución.

Esta búsqueda de archivos es sencilla cuando se trata de exportar a la plataforma Windows ya que se puede acceder a la carpeta StreamingAssets sin ningún problema. Sin embargo, en Android es diferente ya que los archivos están contenidos dentro de un fichero comprimido *.jar.*[36] Sino se tiene en cuenta esto puede llegar a producir errores en el proceso de carga de los archivos xml en la plataforma Android.

En este script se muestra como se hace el proceso de búsqueda de archivos *xml* tanto en Windows como en Android.

En primer lugar, tenemos el método *Start* que es un método de Unity que se ejecuta solamente una vez al inicio de la ejecución. Este método se divide en dos partes dependiendo en la plataforma en la que se ejecute. Si se ejecuta en Android se hace una comprobación previa y después se recorre un array de String que contiene el nombre de los ficheros *xml* a cargar. Utilizando una corrutina se le pasa como parámetro el nombre del fichero y se define un *path.* Haciendo uso de la clase *UnityWebRequest*[37] se accede al fichero de la ruta definida anteriormente y se descarga para su uso.

En la siguiente imagen se muestra el fragmento de código del script *pbFSM_Manager* dónde se realizan las acciones descritas anteriormente.

[36] Extensión de archivo JAR: Java Archive

[37] Información de la clase UnityWebRequest
https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.html

```
private void Start()
{
    xmltest.FSM_Parser parser = new xmltest.FSM_Parser();
    fsm_manager = new FSM_Manager(parser);

    if (Application.platform == RuntimePlatform.Android)
    {
        for(int i=0; i<names.Length; i++)
            StartCoroutine(loadFile(names[i]));
    }
    else
    {
        // Carga las máquinas que se encuentran en /Assets/StreamingAssets/AI/
        DirectoryInfo dir = new DirectoryInfo(Application.streamingAssetsPath + "/AI/");
        var files = dir.GetFiles("*.xml");

        for (int i = 0; i < files.Length; i++)
        {
            fsm_manager.addFSM(files[i].FullName);

        }

    }
}
IEnumerator loadFile(string names)
{
    string path = "jar:file://" + Application.dataPath + "!/assets/AI/"+names;
    UnityWebRequest wwwfile = UnityWebRequest.Get(path);
    yield return wwwfile.SendWebRequest();
    var filepath = string.Format("{0}/{1}", Application.persistentDataPath,names);
    File.WriteAllBytes(filepath, wwwfile.downloadHandler.data);
    Debug.Log(filepath);
    fsm_manager.addFSM(filepath);
}
```

Ilustración 32. Código de la clase pbFSM_Manager.

Anexo 1

Documento de diseño

Forest Defense

por Enrique Gonjar Verdejo

1 Información general

1.1 Concepto general

El juego consistirá en que el jugador deberá defenderse ante diferentes oleadas de enemigos colocando de manera estratégica diferentes defensas o armas en el mapa, para impedir que los enemigos lleguen a su destino.

1.2 Objetivo

El jugador tendrá como objetivo evitar que los enemigos lleguen al final de su camino utilizando armas que el jugador podrá mejorar o vender a lo largo de la partida.

1.3 Género

Es un Tower defense que pertenece al subgénero de videojuegos de estrategia en tiempo real.

1.4 Historia o sinopsis

Tras muchos años viviendo en paz y en armonía con la naturaleza, los habitantes del reino del bosque han sido descubiertos por una gran amenaza del exterior. Criaturas mágicas enviadas desde un reino desconocido con el objetivo de robar la denominada esencia del bosque. Esta esencia es la que permite que los habitantes junto al bosque existan y sin ella desaparecerían. Tras conocer esta amenaza los habitantes y con la ayuda del bosque, empiezan a construir diferentes armas para poder defender lo que más quieren, su hogar.

1.5 Estilo visual

El juego tendrá un estilo 3D, con gráficos con bajo poligonaje low poly.

1.6 Motor y editor

Se usará Unity en su versión 2018.3.10 y Visual Studio como IDE.

1.7 Núcleo del gameplay

El jugador podrá realizar las siguientes acciones:

- Seleccionar diferentes armas, mejorarlas, venderlas y colocarlas en los espacios destinados para su uso.
- Mover la cámara para mejorar la visión a la hora de colocar las armas.
- Interactuar con los menús, como el menú de pausa.
- Elegir el nivel y desbloquear nuevos niveles.
- Elegir entre diferentes opciones la calidad gráfica.

1.8 Público objetivo

Dirigido a jugadores que tengan experiencia en juegos de estrategia y jugadores casuales que les gusten los retos.

1.9 Características del juego

1.9.1 Ambientación

La historia se desarrolla en un mundo donde habitan varios reinos entre ellos está el del bosque donde transcurre toda la historia. No hay un tiempo definido, ni una época exacta

1.10 Alcance del proyecto

1.10.1 Ubicaciones del juego

La acción se desarrolla en el bosque y en sus alrededores.

1.10.2 Descripción de los enemigos

Encontraremos diferentes tipos de enemigos como son: el fantasma, el golem, el mago y el dragón. Tienen como objetivo conseguir la esencia del bosque. El fantasma es el más débil, pero cuenta con una gran velocidad. El golem utiliza su resistencia para acumular energía y usarla para aumentar su velocidad y poder destruir lo que tenga a su paso con sus puños. El mago usa sus potentes hechizos para destruir cualquier obstáculo e incluso encantar algunas armas para que dejen de funcionar durante algunos segundos. Por último, el dragón, sus alas le permiten volar e ir a una gran velocidad. Si se ve acorralado por los disparos, puede llegar a escupir fuego usando su potente llamarada o una simple pero potente bola de fuego que podrá acabar con cualquier torre que se encuentre a su paso.

1.10.3 Descripción de las armas o defensas

Las armas han sido forjadas por los herreros y encantadas por los magos que habitan en el bosque. Podemos encontrar diferentes armas, entre ellas se encuentra el cañón de plasma que lanza bolas mágicas que producen cierto daño según el tipo de enemigo. El cristal que se encarga de lanzar un rayo que permite ralentizar a los enemigos produciéndose daño. También se encuentra el cañón de misiles, un arma muy potente que causa un gran daño a los enemigos. La última arma es el cañón de fuego una vez colocado estará lanzando una llamarada constantemente para producir quemaduras a los enemigos.

1.10.4 Descripción de los niveles

Los niveles cuentan con un sendero de tierra por donde irán los enemigos. Alrededor de ese camino se encuentran las torres donde se colocarán las armas y también se encuentran ciertos muros de decoración. Además, cuentan con diferentes árboles alrededor de todo el nivel y algunos objetos todos ellos con una función decorativa.

2 Diseño conceptual del juego

2.1 Diseño de los controles

No habrá ningún personaje controlable, pero si se podrá mover la cámara por el nivel para tener una mejor visión a la hora de colocar las defensas. Además, el jugador podrá interactuar con el entorno pudiendo colocar las torres, mejorarlas y venderlas según la necesidad y transcurso de la partida.

En la imagen adjunta se muestran los controles para el movimiento de la cámara en plataformas de escritorio.

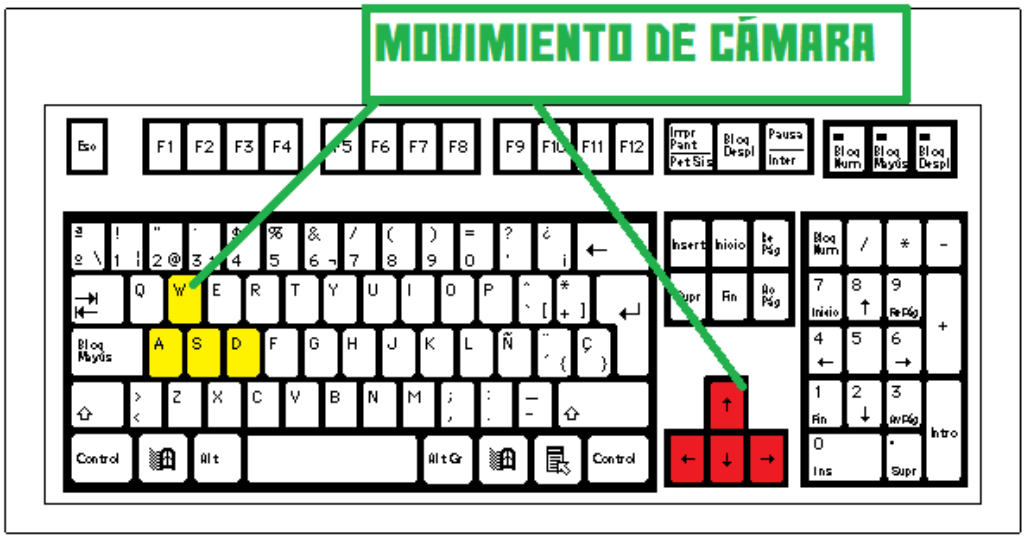

Ilustración 1. Controles de teclado para la cámara.

En la siguiente imagen se muestran los controles con el mouse usando el botón izquierdo para la selección en los menús e interacción con el entorno para colocar las torres en el nivel.

Ilustración 2. Control del mouse para seleccionar.

Los controles para dispositivos móviles se centran en el uso de la pantalla táctil para interactuar con la interfaz y entorno del juego. Para el movimiento de la cámara se hará uso exclusivo de los controles que aparecen en la parte inferior izquierda de la pantalla.

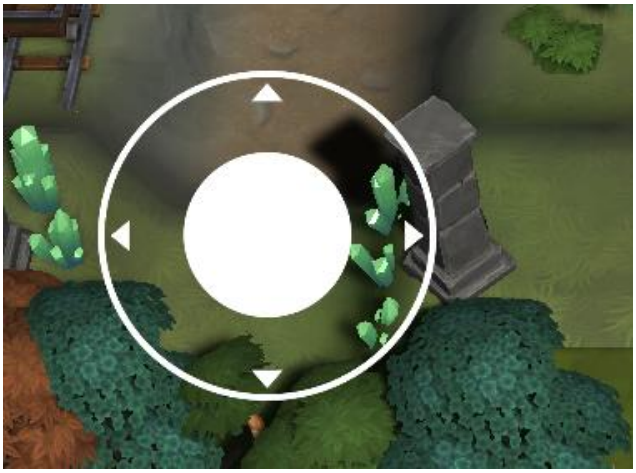

Ilustración 3. Imagen del control de la cámara en Android.

Para mover la cámara en Android se debe colocar el dedo en el círculo inferior de la imagen adjunta, moviéndolo hacia la dirección que indican las flechas.

2.2 Diseño de los personajes

2.2.1 Fantasma

Descripción: Es el enemigo más débil, pero cuenta con una velocidad que si te distraes puede que no te dé tiempo a reaccionar y colocar un arma cerca. Es un enemigo que no le afectan las colisiones cualquier muro lo atravesará y es inmune al fuego.

Velocidad: Tiene una velocidad de 6 puntos.

Puntos de vida: Tiene 100 puntos de vida.

Monedas: Al ser destruido dejará 50 monedas para el jugador.

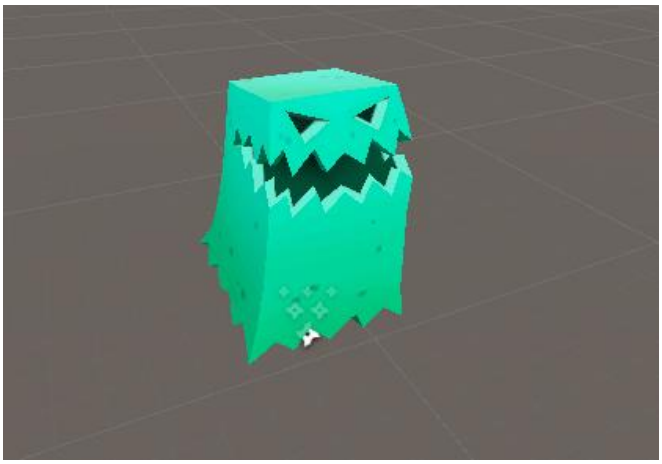

Ilustración 4. Fantasma.

2.2.2 Golem

Descripción: El golem se caracteriza por ser un enemigo resistente. Utiliza su energía para aumentar su velocidad y poder hacer más daño con sus puños, pero cuando se le agota la energía se vuelve un enemigo vulnerable porque se ve reducida su velocidad. Esto permite que los disparos sean más eficaces y producir más daño.

Velocidad: Velocidad normal con valor de 7. Velocidad rápida con valor 14.

Puntos de vida: Tendrá 150 puntos de vida.

Monedas: Al ser destruido dejará 100 monedas.

Ilustración 5. Golem.

2.2.3 Dragón

Descripción: Es el enemigo más temido y más poderoso. Tiene mucha vida además que es capaz de destruir las torres con su potente llamarada o con su bola de fuego.

Velocidad: velocidad de 8 puntos.

Puntos de vida: 500 puntos de vida.

Monedas: 200 monedas.

Ilustración 6. Dragón.

2.2.4 Mago

Descripción: El mago es fuerte por su poder mágico que es capaz de lanzar hasta tres hechizos diferentes para destruir los muros y cuentan con una gran potencia a veces llegando hasta paralizar los disparos de las armas.

Velocidad: Tendrá una velocidad de 6 puntos.

Ataque 1: Disparará 1 bala por segundo con un daño de 20. puntos de daño.

Ataque 2: Disparará 2 balas por segundo con un daño de 30 puntos de daño.

Ataque 3: Disparará 3 balas por segundo con un daño de 50 puntos de daño.

Puntos de vida: Tendrá 200 puntos de vida.

Monedas: Al ser destruido dejará 150 monedas.

Ilustración 7. Mago.

2.3 Diseño de las torres.

Las torres son las plataformas donde se colocarán las armas. Están distribuidas por cada nivel y son de varios tipos: las de madera donde se colocarán las armas como el cañón mágico, el cristal y el cañón de proyectiles. Y las de piedra donde se colocará el cañón de fuego. (ver ilustración 9).

Ilustración 8. Torre de madera.

Ilustración 9. Torre de piedra con el cañón de fuego colocado.

2.4 Diseño de las armas.

En este apartado se detalla las diferentes armas que el jugador podrá poner en el escenario del juego.

2.4.1 Cañón mágico

Descripción: El cañón mágico se encarga de disparar bolas mágicas que infligen daño al enemigo cuando éstas impactan con él. Esta arma permite una mejora la cual hará que los disparos sean más potentes y aumentará el número de balas por segundo.

Cadencia de disparo: un disparo por segundo sin la mejora y dos disparos por segundo con la mejora.

Daño: variable entre 10 y 20 puntos de daño.

Coste: 100 monedas.

Ilustración 10. Cañón mágico.

2.4.2 Cristal láser

Descripción: Es un cristal que lanza un rayo de luz que permite ralentizar a los enemigos y así combatirlos más rápido. Este rayo también produce daño dependiendo del enemigo. Cuando se mejora permite mejorar el daño y hacer que los enemigos vayan mucho más lentos.

Daño: variable entre 5 y 15 puntos de daño.

Coste: 150 monedas.

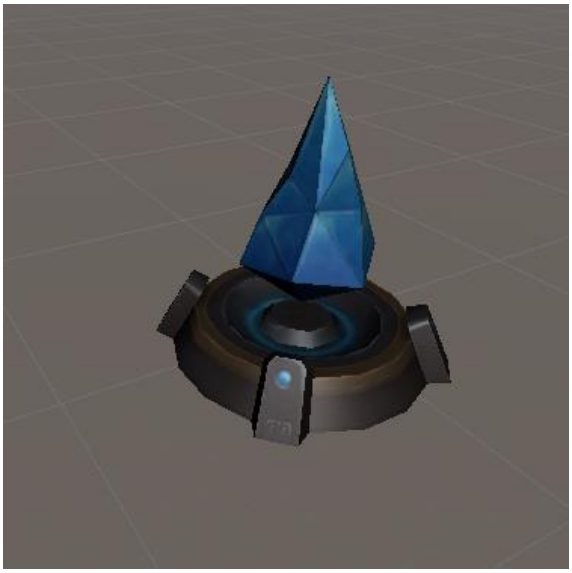

Ilustración 11. Cristal láser.

2.4.3 Cañón de proyectiles

Descripción: Es un potente cañón que dispara proyectiles. Estos proyectiles son los troncos de los árboles que han sido convertidos en misiles gracias a la magia que poseen ciertos habitantes. Cuando se mejora este cañón consigue disparar dos misiles a la vez aumentando el daño contra los enemigos.

Cadencia de disparo: dispara un misil sin utilizar la mejora. Utilizando la mejora dispara dos misiles.

Daño: daño variable que oscila entre 10 y 40 puntos.

Coste: 300 monedas.

Ilustración 12. Cañón de proyectiles.

2.4.4 Cañón de fuego

Descripción: Esta arma una vez colocada produce continuamente una gran llamarada que cuando los enemigos pasan por ella les inflige daño. Hay enemigos inmunes como son: el dragón y el fantasma.

Daño: Depende del enemigo. El golem recibe 5 puntos y el mago 10 puntos. El dragón y el fantasma son inmunes.

Coste: 175 monedas.

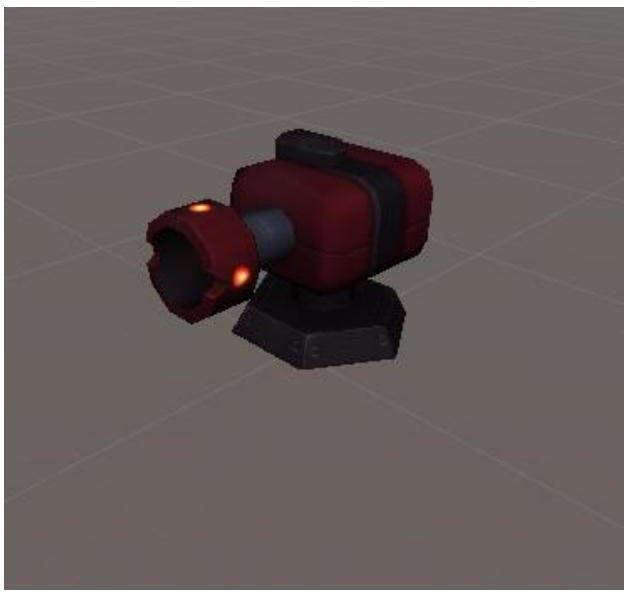

Ilustración 13. Cañón de fuego.

2.4.5 Muro o bloque

Descripción: No es considerado un arma, pero si es un elemento que el usuario puede colocar en el suelo para que los enemigos se detengan. No se puede mejorar. No tiene efecto sobre el fantasma y el dragón.

Puntos de vida: 100 puntos de vida.

Coste: 100 monedas.

Ilustración 14. Muro o bloque.

2.5 Diseño de la tienda

La tienda se encuentra en la parte inferior de la interfaz del juego. Este menú se compone de cinco botones donde el jugador podrá hacer click con el ratón si está en Windows o pulsar con el dedo si hace uso de Android.

Ilustración 15. Menú de la tienda.

Cuando se pulsa uno de esos botones el jugador podrá colocar el arma seleccionado en una de las torres que están repartidas por todo el mapa, con solo pulsar en la torre en la que se quiera colocar dicha arma. El último botón es para colocar bloques de piedra y estos tienen un lugar específico donde colocarlos.

El jugador podrá mejorar o vender las armas que desee para ello solamente tiene que pulsar en la torre donde se encuentre el arma que se quiera vender o mejorar. Aparecerá un menú con dos opciones seleccionables que son: vender o mejorar tal como se muestra en la imagen.

Ilustración 16. Menú para mejorar o vender un arma.

2.6 Diseño de niveles

2.6.1 Nivel 1

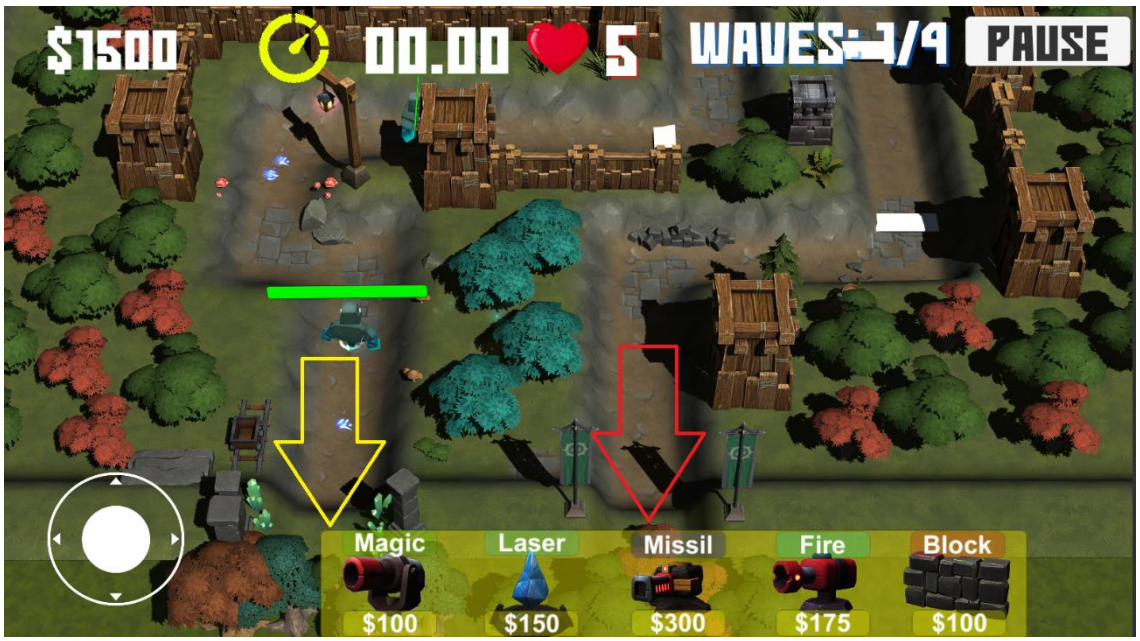

Ilustración 17. Nivel 1.

En la imagen se muestra como es el nivel 1. La flecha amarilla indica por donde aparecerán los enemigos y la flecha roja indica donde desaparecerán sino son destruidos antes. En el nivel 1 el jugador cuenta con 6 vidas. Las plataformas blancas que se observan en el suelo sirven para colocar los bloques de piedra.

2.6.2 Nivel 2

Este nivel transcurre en la noche para dar cierta dificultad. La flecha amarilla indica por donde saldrán los enemigos y la flecha roja donde desaparecerán. Se ha añadido niebla como efecto decorativo y se ha eliminado el cañón de fuego.

Ilustración 18. Nivel 2.

2.7 Diseño de la interfaz

2.7.1 Interfaz del juego

Una vez dentro del nivel se mostrarán diferentes elementos que estarán presentes en todos los niveles del juego.

Empezando por la parte superior izquierda encontramos el dinero disponible para poder comprar los diferentes elementos para colocar en el nivel.

Seguidamente se encuentra el contador de tiempo que transcurre entre oleada.

Siguiendo en la parte superior se encuentran las vidas del jugador junto con las oleadas que faltan para acabar el nivel y por último el botón de pausa.

En la parte inferior izquierda se encuentra el botón para controlar la cámara en Android. El jugador deberá colocar el dedo en el centro de la imagen y moverlo hacia la dirección en la que quiera mover la cámara.

Para finalizar la parte inferior encontramos la tienda de objetos que se constituye de cinco botones donde el jugador podrá seleccionar las diferentes armas que puede colocar en el mapa.

2.7.2 Menú de inicio

Cuando se inicia el juego aparecerá la pantalla del menú principal donde podemos observar el título del juego y tres botones con los que interactuar:

- ➔ **Play:** al darle te llevará a la pantalla de selección de nivel.
- ➔ **Controls:** aparecerá una pantalla donde se mostrarán los controles del juego.
- ➔ **Options:** ejecutará un menú donde aparecerán diferentes opciones como poder controlar el volumen de la música y los efectos y también elegir la calidad de los gráficos.
- ➔ **Exit:** la aplicación se cerrará y dejará de ejecutarse.

Ilustración 19. Menú principal.

2.7.3 Menú de pausa

El menú de pausa aparecerá cuando se pulse el botón de pausa que se encuentra en la parte superior derecha de la pantalla del juego. Cuando se pulse el botón el juego se detendrá y aparecerá un menú con las siguientes opciones:

- ➔ **Resume:** al darle te permite volver a la partida.
- ➔ **Reset:** permite reiniciar el nivel actual.
- ➔ **Main Menu:** permite ir a la pantalla de inicio donde está el menú principal.

- ➔ **Options:** al pulsar aparecerá una pantalla con el menú de opciones que se describe en el siguiente apartado.
- ➔ **Exit:** permite detener la aplicación y cerrarla.

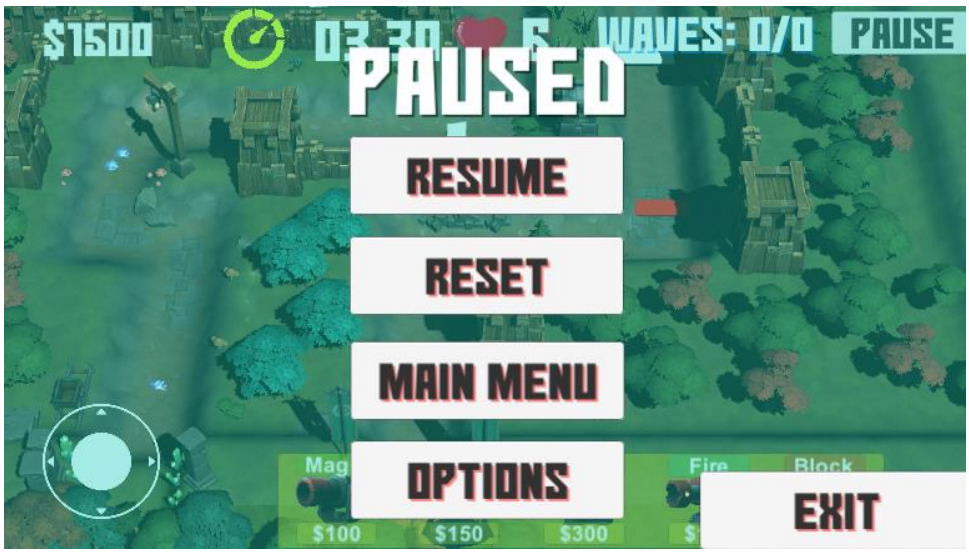

Ilustración 20. Menú pausa.

2.7.4 Menú de opciones

Al menú opciones se puede acceder desde el menú principal o desde el menú de pausa. Este menú cuenta con varias opciones:

- ➔ **Graphics:** permite configurar los gráficos del juego entre múltiples opciones.
- ➔ **Volume music:** el jugador podrá configurar el volumen de la música de fondo a su gusto moviendo la barra hacia la izquierda para bajar el volumen o derecha para subirlo.
- ➔ **Volume effect:** con esta opción se podrá bajar y subir el volumen de los efectos del juego, como los sonidos de los disparos o los de selección.
- ➔ **Back:** pulsando en este botón se volverá al menú anterior que puede ser el menú de pausa o el menú principal.

Ilustración 21. Menú opciones.

2.7.5 Menú nivel completado

Este menú aparecerá cuando el jugador haya eliminado a todas las oleadas o que la vida del jugador no haya llegado a cero. Este menú consta de las siguientes opciones:

- ➔ **Continue:** al pulsar se cargará el siguiente nivel del juego.
- ➔ **Menú:** al pulsar se volverá al menú principal del juego.
- ➔ **Exit:** permitirá cerrar la aplicación completamente.

Ilustración 22. Menú nivel completado.

2.7.6 Menú fin del juego

Si el jugador se queda sin vidas entonces aparecerá la siguiente pantalla con diferentes opciones:

- ➔ **Play again:** si se pulsa en este botón se reiniciará el nivel con las monedas por defecto y las vidas.
- ➔ **Main menu:** permitirá al jugador volver al menú principal.
- ➔ **Exit:** al pulsar en este botón se cerrará la aplicación.

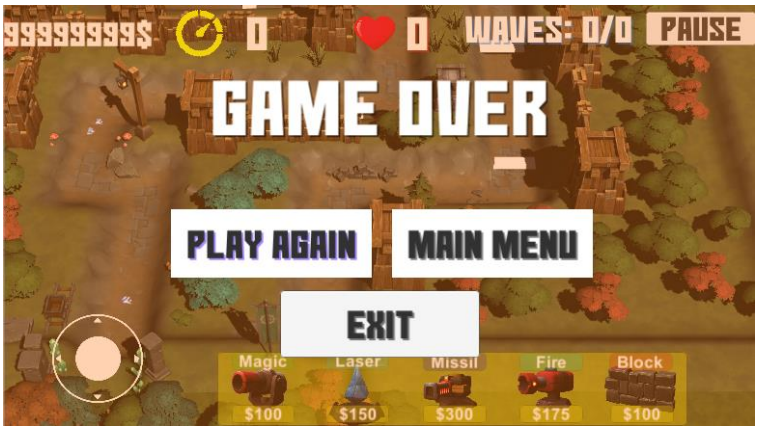

Ilustración 23. Menú fin de juego.

2.7.7 Menú selección de nivel

Al darle al botón *play* del menú principal seguidamente aparecerá la pantalla de selección de niveles.

En esta pantalla encontraremos diferentes botones que indican el número del nivel a seleccionar. Inicialmente sólo el nivel 1 estará disponible y el resto de niveles estarán bloqueados. El nivel 2 se desbloquea cuando se supere el nivel 1, el nivel 3 cuando se supere el nivel 2. Esta mecánica se emplea para el resto de niveles. Para superar los niveles hay que sobrevivir a las distintas oleadas de enemigos.

Ilustración 24. Menú selección de nivel.

2.7.8 Diagrama de flujo

En este diagrama se muestra cómo se relacionan cada uno de los menús descritos anteriormente. Nada más iniciar la aplicación nos encontramos en el menú principal donde tenemos diferentes opciones. Podemos ir al menú opciones o al menú donde se muestran los controles. Para iniciar el juego se debe pulsar el botón *play* que nos llevará a la pantalla de selección de niveles y ahí debemos seleccionar el nivel. Una vez seleccionado el nivel se mostrará una pantalla de carga con algunas instrucciones a seguir a la hora de colocar las armas en las torres. Cargado ya el nivel nos encontramos en el juego y pueden ocurrir los siguientes sucesos: que se complete el nivel por lo tanto aparecerá el menú de nivel completado y se podrá elegir entre continuar con el siguiente nivel, volver al menú principal o salir de la aplicación. Si no se completa el nivel entonces aparecerá la pantalla de fin del juego con diferentes opciones: salir del juego, reiniciar el nivel o volver al menú principal. Para finalizar, durante el juego el usuario siempre puede pausar la partida. El menú de pausa cuenta con varias opciones que son: volver al juego, ir a opciones, volver al menú principal y salir de la aplicación.

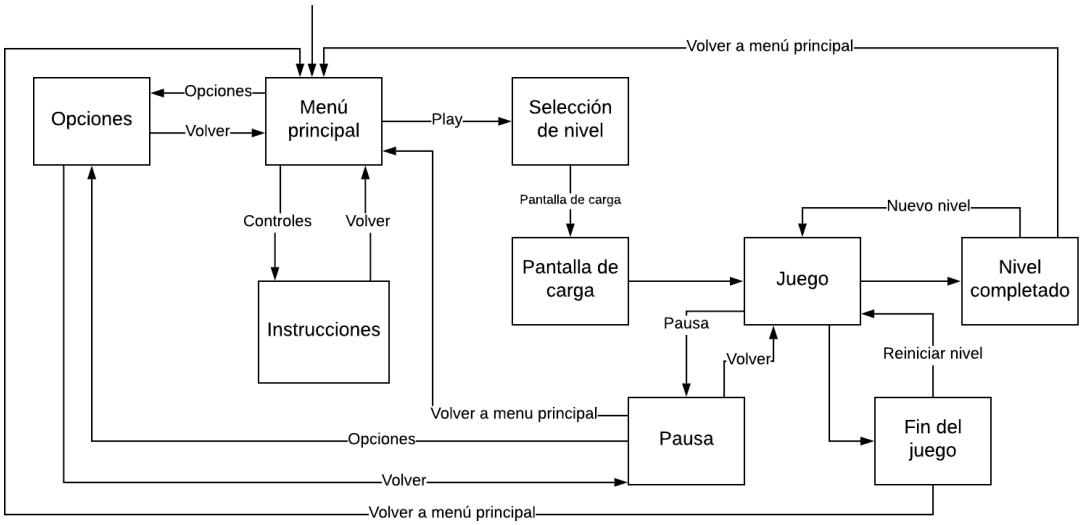

Ilustración 25. Diagrama de flujo del juego.

3 Recursos

La utilización de estos recursos se hace con fines académicos sin intención de comercializar.

3.1 Audios

- Música del menú *The True Story of Beelzebub*: https://patrickdearteaga.com/es/musica-libre-derechos-gratis/
- Música de los niveles *The Three Princesses of Lilac Meadow* : https://patrickdearteaga.com/es/musica-epica-orquestal-fantasia-medieval/
- Sonido de monedas: https://freesound.org/people/NenadSimic/sounds/171756/

3.2 Imágenes

- Imagen de fondo menú principal: https://cutt.ly/nCOyxi

3.3 Fuentes de texto

- Fuentes menús: https://www.dafont.com/no-continue.font

3.4 Modelos 3D

- Escenarios y armas versión demo utilizada: http://unityassetcollection.com/tower-defense-and-moba-free-download/
- Escenarios y armas versión original: https://assetstore.unity.com/packages/3d/environments/fantasy/tower-defense-and-moba-28234
- Packs enemigos versión original: https://assetstore.unity.com/packages/3d/characters/toon-enemies-pack-50421
- Packs enemigos versión demo utilizada: http://unityassetcollection.com/toon-enemies-pack-free-download/

3.5 Otros

- Efectos partículas para los disparos versión demo utilizada: http://unityassetcollection.com/unique-projectiles-volume-1-free-download/
- Efectos de partículas para los disparos fuente original: https://assetstore.unity.com/packages/vfx/particles/unique-projectiles-volume-1-124214
- Joystick para el control en Android: https://assetstore.unity.com/packages/tools/input-management/joystick-pack-107631