



## **ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

### **GRADO EN INGENIERÍA INFORMÁTICA ESPECIALIDAD EN INGENIERÍA DEL SOFTWARE**

#### **AR TOWER DEFENSE**

**Realizado por  
CARLOS JIMENO CORDERO**

**Dirigido por  
MANUEL DOMINGUEZ MORALES**

**Departamento  
ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES**

**Sevilla, septiembre 2019**

## Resumen

La industria del videojuego ocupa actualmente uno de los primeros puestos en ingresos anuales entre las industrias del entretenimiento. Dentro de esta, los estudios indie tienen cada vez más fuerza, estos suelen estar compuestos por grupos pequeños de personas y contar con escasos recursos, no obstante, producen juegos exitosos y muy cuidados. Cada vez más, están llegando a nuestras manos videojuegos de realidad virtual y aumentada que no requieren de equipos caros para ser jugados.

En este proyecto se llevará a cabo el desarrollo de un videojuego usando el motor de videojuegos Unity. El juego tendrá características propias de un juego tipo ‘tower defense’, es decir, habrá que colocar y mejorar torretas para defenderse de oleadas consecutivas de enemigos. Estará únicamente disponible para Android y orientado a ser usado en dispositivos móviles, además de incluir características que permitan jugar haciendo uso de realidad aumentada, gracias al uso del software de realidad aumentada Vuforia.

## ÍNDICE

1. Descripción del proyecto .....	4
1.1. Motivación.....	4
1.2. Objetivos del proyecto .....	7
1.3. Estado del arte.....	8
1.4. Elicitación de requisitos.....	13
1.4.1. Introducción al dominio del problema .....	13
1.4.2. Requisitos Generales .....	13
1.4.3. Actores del sistema .....	14
1.4.4. Casos de uso .....	15
1.4.5. Requisitos de información.....	26
1.4.6. Requisitos funcionales.....	28
1.4.7. Requisitos no funcionales.....	30
1.4.8. Asunciones y excepciones .....	31
2. Ejecución del proyecto .....	32
2.1. Diseño del Sistema .....	32
2.2. Implementación.....	35
2.2.1. Tecnologías empleadas .....	35
2.2.2. Desarrollo .....	43
2.3. Pruebas .....	72
3. Planificación del proyecto .....	73
3.1. Planificación temporal inicial.....	73
3.2. Planificación financiera inicial .....	74
3.3. Planificación temporal final.....	76
3.4. Planificación financiera final.....	77
4. Conclusiones.....	78
5. Trabajo futuro.....	79
Bibliografía.....	81
Anexos .....	82
A - Imágenes Vumark reconocibles para el juego. ....	82
B – Manual de instalación de la aplicación.....	83

## 1. Descripción del proyecto

### 1.1. Motivación

La industria de los videojuegos ocupa desde hace tiempo uno de los primeros puestos como producción del entretenimiento en cuanto a ingresos. Desde su inicio hasta ahora ha experimentado un crecimiento inmenso que aún sigue en marcha. Actualmente tiene un valor a nivel global de 152 Billones de dólares siendo el sector móvil con tablets y smartphones el dominante, aportando 68.5 Billones de dólares al año lo cual corresponde a un 45% del total [1].

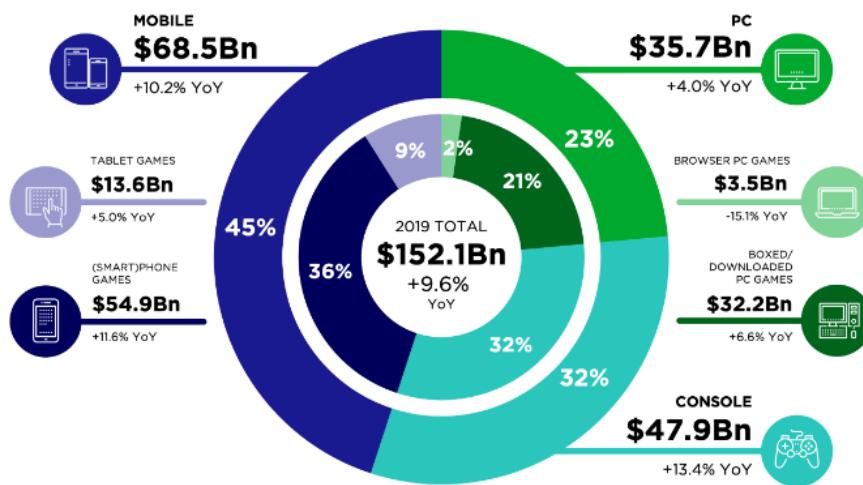


Imagen 1 - Ingresos industria del videojuego

Los videojuegos de realidad aumentada o virtual también están ocupando un lugar importante en la expansión de los videojuegos, títulos como Pokémon Go (Nintendo) han demostrado que no es totalmente necesario disponer de un dispositivo con grandes capacidades para poder disfrutar de algunas características que aporta la realidad aumentada. Esto nos deja entrever los retos y peculiaridades aún por explotar que supone integrar un videojuego con la realidad virtual/aumentada, sin necesidad de hacer grandes gastos como comprar unas gafas caras ni un equipo que sea compatible con ellas.

En esta última década, también ha sido notorio el auge en el campo de los videojuegos indie. Este término en algunas ocasiones da lugar a confusión, pensándose que se refiere a una estética concreta o un estilo, simplemente es una etiqueta que ha ido arrastrando muchos tópicos que no se relacionan con lo que es realmente. Los videojuegos incluidos dentro de este campo están caracterizados por ser independientes. Lo cual quiere decir que no están sujetos a dictados de una gran compañía que les indique qué camino coger, llevando normalmente al juego por un estilo más comercial. Normalmente, este estilo está

acompañado de pequeños grupos de desarrolladores con ayudas financieras ínfimas o nulas. Aunque pueda parecer a primera vista que los videojuegos indie son de menor calidad o valor que los videojuegos triple A [2], esto está muy lejos de la realidad. Existe una lista interminable de títulos indie que han demostrado destacar entre el resto de los videojuegos principalmente por innovar y cuidar mucho los detalles, algunos ejemplos son: Enter the Gungeon (Dodge Roll), Hollow Knight (Team Cherry), Celeste (Matt Makes Games), Cuphead (Studio MDHR), Terraria (Re-Logic), ... y muchos más.



Imagen 2 - Juegos indie famosos

Este boom de los videojuegos indie se debe en su mayor parte a lo fácil que es distribuir los videojuegos de manera online ya sea por plataformas como Steam, Play Store o App Store y asimismo a la gran cantidad de recursos disponibles de forma gratuita para el desarrollo de videojuegos.

Entre las plataformas de desarrollo más conocidas se encuentra Unity, que es un motor de desarrollo completamente integrado y que tiene abundantes funciones para la creación de contenido 3D interactivo. Además, puede ofrecer contenido de alta calidad y rendimiento en múltiples plataformas y tiene una gran variedad de usuarios, tanto pequeños desarrolladores como grandes empresas multinacionales, como estudiantes o aficionados. La razón de que sea usado un grupo tan amplio es su suave curva de aprendizaje y la rapidez con la que se pueden conseguir resultados decentes. Por otro lado, Unreal Engine, que es algo más complejo que Unity y produce resultados más conocidos por un increíble acabado gráfico. Ambas tecnologías cuentan con una gran comunidad online que proporciona

innumerables recursos de aprendizaje, haciendo posible para cualquiera adentrarse en el mundo del desarrollo de videojuegos.

El videojuego que se desarrolla en este proyecto se aprovechará del carácter innovador que tienen los juegos de realidad aumentada mezclándose con las características de un típico juego Tower Defense. La motivación personal del alumno es la creada por tener una vida rodeada de videojuegos, en la que la fuente principal de entretenimiento viene de estos. Además de la curiosidad de experimentar con aplicaciones de realidad aumentada conociendo su funcionamiento al menos a nivel básico.

A lo largo del proyecto se usarán numerosas fuentes de aprendizaje, estas servirán de guía y en algunos casos serán esenciales para el desarrollo del proyecto. Los recursos principales son: 'CatlikeCoding' y su sección de tutoriales de Unity [3], 'Brackeys' y su gran cantidad de videos relacionados con el desarrollo con Unity [4], y finalmente 'Sebastian Lague' con sus videos inspiradores e ilustrativos de desarrollo y experimentación con Unity [5]. Además de los recursos relacionados exclusivamente con Unity, se hará uso durante un mes de la plataforma de pago 'CG Cookie' para reforzar el aprendizaje de Unity y Blender [6].

## 1.2. Objetivos del proyecto

Los objetivos del proyecto son:

### Unity 3D

Diseño e implementación de videojuego para Android utilizando la plataforma Unity. Se creará un juego de temática ‘tower defense’, es decir, habrá que implementar torretas, enemigos, oleadas, etc. Se seguirá inicialmente una etapa de aprendizaje de la herramienta y posteriormente su diseño y desarrollo. Dentro de este objetivo se incluye:

- Comprender a nivel básico / medio todo el flujo de creación de videojuegos con Unity.
- Familiarizarse con su interfaz haciendo uso de esta.
- Conocer el funcionamiento del ciclo de vida de programación de los objetos en Unity.
- Importación y uso de paquetes de utilidades ‘Assets’.
- Diseño de niveles / escenas.
- Flujo de creación /edición de ‘prefabs’.
- Diseño y creación de interfaz básica.
- Hacer uso de C# y la herramienta Visual Studio para desarrollar todos los scripts del juego.
- Poner en práctica algunos de los conceptos aprendidos en la carrera para conseguir resultados limpios y optimizando el rendimiento todo lo posible.
- Aprender técnicas o trucos que sean de utilidad con la integración de elementos de Unity.

### Etiquetas visuales para realidad aumentada

Estudio y elaboración de etiquetas visuales para su integración en el sistema de realidad aumentada. Se deberá diseñar y experimentar con la creación de etiquetas visuales para su posterior uso con el software de realidad aumentada Vuforia.

- Experimentar a la par con Unity reconocimientos de imágenes, ‘vumarks’, modelos 3D, botones, etc. para conocer las capacidades del software.
- Generación de imágenes codificadas ‘Vumark’ con la utilidad que proporcionan para ello.

### Vuforia

Integración de la detección de objetos de realidad aumentada mediante Vuforia en el juego. Establecer un sistema de detección de elementos de realidad aumentada dentro del juego.

- Comprender y poner en práctica los conceptos básicos del funcionamiento de Vuforia.
- Conseguir utilizar la herramienta para poder renderizar el juego sobre una imagen reconocible.
- Diseño de interacciones entre el usuario y el juego mediante uso de botones o gestos con realidad aumentada.

### 1.3. Estado del arte

#### ¿Qué es un juego Tower Defense?

Tower Defense es un género de juegos que trata sobre estrategia, gestión de recursos y planificación. En ellos el jugador debe impedir que oleadas de enemigos lleguen a su destino, para ello dispone de recursos que puede gastar en comprar torretas. Estas torretas son las que se encargan de hacer daño a los enemigos y eliminarlos antes de que alcancen su objetivo.

Esa sería la definición básica de este género, luego existen muchas características que diferencian a unos de otros. Algunas de las características que más se repiten son:

- Mejoras de torretas: se pueden mejorar las características de una torreta gastando más recursos en ella.
- Diversidad de torretas: cada torreta tiene un tipo de ataque distinto que puede ser más o menos útil según donde se coloque y contra algún tipo de enemigo específico.
- Diversidad de enemigos: cada enemigo tiene unos atributos que lo diferencia del resto, algunos de estos pueden ser velocidad, vida, armadura, resistencia a algún tipo de torreta, etc.
- Camino preestablecido: los enemigos tienen un camino fijo por donde deben pasar y existen posiciones a lo largo del mismo donde colocar las torretas.
- Progresión de niveles: el usuario suele empezar por los niveles básicos y a medida que los completa, se van desbloqueando niveles más difíciles.
- Modo sin fin: son niveles que no tienen un número de oleadas límite que el usuario tenga que superar para completar el nivel. El objetivo es aguantar el máximo número de oleadas posible.

Actualmente existen infinidad de juegos del género ‘Tower Defense’ tanto para PC y consolas como para móvil. Sin embargo, diría que este género tiene su mayor público en el mercado para móviles, ya que son juegos normalmente de partidas rápidas y casuales. Es por ello por lo que nos hemos centrado en los productos existentes en este sector y además, porque el juego se desarrollará sólo para móvil por la comodidad de la cámara para la realidad aumentada. A continuación, mencionamos algunos de los juegos más importantes y por qué destacan o pueden servir de referencia para el desarrollo del juego.

#### Kingdom Rush [7]

Uno de los mejores juegos de este género, disponible para Android, iOS y PC (navegador). Cuenta con una estética cartoon 2D y ambientación medieval muy característica. Destaca por su gran cantidad de contenido además de poseer guerreros que te ayudan y poderes que puedes invocar en mitad del tablero cuando lo necesites.

Los mapas son siempre con caminos preestablecidos y hay huecos fijados a lo largo de los mismos para colocar torretas. Muchos niveles cuentan con múltiples caminos de entrada/salida de enemigo, esto hace que el usuario deba gestionar bien sus recursos, cubriendo lo mejor posible todos los caminos y usando las posiciones mejor cualificadas para un tipo de torreta u otro.

Características a tener en cuenta:

- Estética simpática y original.
- Sistema de oleadas muy bien desarrollado.
- Interfaz sencilla e intuitiva.
- Gran variedad de torretas y enemigos.
- Poderes activables por el usuario en zonas concretas del mapa.
- Héroes /guerreros que se pueden mover por el mapa ayudando a las torretas.
- Sistema de mejoras permanentes, al acabar una partida se consiguen puntos que se pueden gastar en mejoras permanentes para las torretas, poderes o héroes.
- Modo campaña: tiene un conjunto de niveles que van progresando en dificultad y van haciendo de tutorial enseñando al jugador los distintos elementos que puede utilizar poco a poco.



Imagen 3 - Kingdom Rush

### Plantas contra Zombis [8]

Quizás el más conocido de todos los Tower Defense que se puedan nombrar. Plantas contra Zombis es un juego sencillo y aun así, muy exitoso y divertido. A diferencia de otros juegos de este género, la disposición y el funcionamiento del tablero es un tanto peculiar, se dispone de varias líneas horizontales que los zombis atacarán independientemente, si consiguen destrozar varias de ellas, has perdido. La mayoría de torretas hacen daño en su línea horizontal aunque existen excepciones que te permiten disparar en diagonal o hacer daño en área.

Además, las plantas o torretas deben ser compradas usando energía solar que se obtiene también con plantas que la generan e interactuando con ella cuando aparece. Esto hace que no sólo tengas que estar atento a que plantas de daño colocar sino también qué plantas de generación de energía colocar y dónde. En otros tower defense, lo normal suele ser obtener esta energía / dinero eliminando enemigos.

Características a tener en cuenta:

- Gran variedad de torretas, cada una con sus peculiaridades. Limitan el número de torretas que puedes usar en cada nivel y suelen dejar que sea el usuario quien elija cuales usar antes de comenzar el nivel.
- Gran variedad de enemigos, algunos sólo pueden ser derrotados con torretas específicas.
- Gran variedad de niveles, cada uno con nuevas características, que hacen que el usuario disfrute y no se aburra fácilmente del juego.
- Disposición del mapa peculiar, con filas horizontales que defender.



Imagen 4 - Plantas contra Zombis

## AR Defender 2

Entramos ahora en el mundo de los Tower Defense con Realidad Aumentada. AR Defender 2 fue publicado en 2012 por BulkyPix y está disponible únicamente para iOS. Tiene características interesantes como poder jugar en multijugador local por wifi. También permite habilitar o deshabilitar la realidad aumentada y controlar a personajes dentro de la partida.

Para la realidad aumentada usan una imagen reconocible por el dispositivo donde aparecería un generador que hay que defender. No tiene un mapa establecido, el usuario es libre de colocar torretas donde desee, y los enemigos aparecen por distintas zonas alrededor de la base.

Características a tener en cuenta:

- Libertad para colocar torretas donde el usuario quiera.
- Manejo de personajes / poderes por parte del usuario.
- Realidad aumentada con reconocimiento de imagen: funcionamiento “centro del mundo”.
- Diseño más realista con estética cartoon 3D.
- Modo multijugador local.



Imagen 5 - AR Defender 2

### GeoDefense Swarm

Publicado en 2009 por Critical Thought Games únicamente para iOS, es una secuela del juego GeoDefense publicado en 2008. Fue un juego no tan conocido como Plantas contra Zombis, pero aún así, también tuvo bastante éxito en su momento. No tiene realidad aumentada, pero es el juego que inspira este proyecto y en el que se basará, casi en su totalidad, el juego que hemos desarrollado.

A diferencia del juego que lo precede, este cuenta con un mapa de casillas hexagonales de distintos tipos. Estas permiten al jugador colocar torretas de distintos tipos estableciendo él mismo el camino que van a tener que seguir los enemigos para llegar a su destino (las torretas bloquean el camino de los enemigos).

Tiene una estética 2D con estilo arcade mezclado con texturas de neón y muchos efectos de partículas que le dan al juego un aspecto bastante llamativo.

Características a tener en cuenta:

- Oleadas de enemigos bien diseñadas.
- Progresión de niveles con dificultad creciente.
- Diseño de niveles bien conseguido y bastante diversidad.
- Estética aparentemente simple pero interesante y llamativa.
- Mapa con casillas hexagonales de distintos tipos que aumentan las posibilidades de estos.

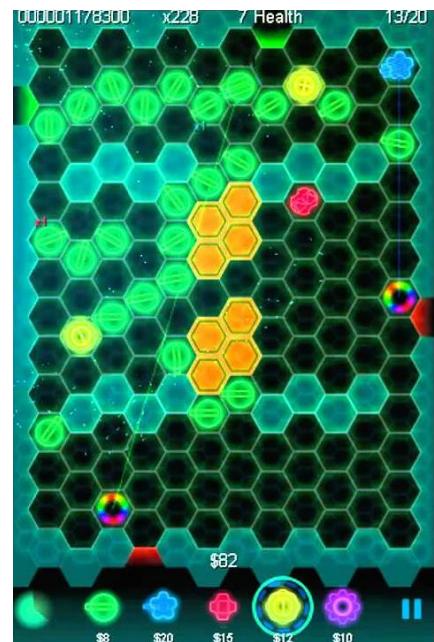


Imagen 6 - GeoDefense Swarm gameplay

El proyecto se basará principalmente en este juego, por tanto, cabe destacar las diferencias que aportará con respecto al mismo:

- Entorno 3D: usará modelos 3D para representar el entorno, las torretas y los enemigos.
- Realidad Aumentada: permitirá jugar con realidad aumentada usando una imagen de referencia que haga de 'centro del mundo'.

## 1.4. Elicitación de requisitos

Esta sección trata de los requisitos identificados para el juego, describiendo por un lugar las exigencias generales del sistema, los casos de uso, los requisitos de información y los funcionales. Podremos guiar el desarrollo del proyecto a partir de todos los datos definidos en esta sección.

### 1.4.1. Introducción al dominio del problema

El juego que vamos a desarrollar en este proyecto pretende ser un juego de estrategia Tower Defense, donde el objetivo principal es sobrevivir a oleadas de enemigos que intentarán llegar a su destino. Para sobrevivir, el jugador debe colocar torretas a lo largo del camino o de forma que alteren el camino que deben seguir los enemigos. Estas torretas se encargan de forma automática de atacar a los enemigos que tengan a su alcance. Para poder colocar torretas hay que gastar monedas del juego que se consiguen eliminando enemigos, estas también pueden ser usadas para mejorar las torretas ya colocadas.

Además, el juego tendrá opción de jugarse con Realidad aumentada, permitiendo ver la acción sobre una imagen reconocible en la vida real.

### 1.4.2. Requisitos Generales

A continuación, se describen los requisitos generales del sistema.

<b>RG-01</b>	Aplicación Android
<b>Descripción</b>	El sistema deberá estar disponible únicamente para la plataforma Android.

<b>RG-02</b>	Realidad Aumentada
<b>Descripción</b>	El sistema deberá integrar funcionalidades de realidad aumentada haciendo uso de la cámara del dispositivo Android.

<b>RG-03</b>	Flujo del Juego
<b>Descripción</b>	El sistema deberá controlar el progreso del juego haciendo uso de variables, estructuras y métodos de control diseñados con ese propósito.

<b>RG-04</b>	Controles Táctiles
<b>Descripción</b>	El sistema deberá estar adaptado para ser controlado mediante gestos táctiles.

<b>RG-05</b>	Tablero de juego
<b>Descripción</b>	El sistema deberá generar un tablero de juego de casillas hexagonales.

<b>RG-06</b>	Distintos tipos de casillas
<b>Descripción</b>	El sistema deberá soportar distintos tipos de casillas.

<b>RG-07</b>	Distintos tipos de enemigos
<b>Descripción</b>	El sistema deberá soportar distintos tipos de enemigos.

<b>RG-08</b>	Distintos tipos de torretas
<b>Descripción</b>	El sistema deberá soportar distintos tipos de torretas.

<b>RG-09</b>	Control del tiempo
<b>Descripción</b>	El sistema deberá soportar el control de la velocidad con la que pasa el tiempo dentro del juego.

<b>RG-10</b>	Búsqueda de caminos
<b>Descripción</b>	El sistema deberá tener implementar algún mecanismo que permita a los enemigos determinar el camino más corto hacia un punto de destino.

#### 1.4.3. Actores del sistema

Únicamente existe un tipo de actor en el sistema.

<b>A-01</b>	Jugador
<b>Descripción</b>	Este actor representa al jugador o usuario del sistema. Será la persona que esté haciendo uso del juego.

#### 1.4.4. Casos de uso

A continuación, se detallan los casos de uso identificados para el sistema.

<b>CU-01</b>	Iniciar Juego	
<b>Precondición</b>	El jugador debe tener instalado el juego en su dispositivo Android.	
<b>Descripción</b>	El juego debe iniciarse y abrir la primera pantalla de selección de niveles cuando el jugador pulse sobre el ícono de la aplicación.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el ícono de la aplicación.
	2	El sistema carga el juego y la pantalla de selección de niveles.
<b>Postcondición</b>	El jugador se encuentra en la pantalla de selección de niveles.	

<b>CU-02</b>	Seleccionar un nivel	
<b>[Dependencias]</b>	CU-01	
<b>Precondición</b>	El jugador se encuentra actualmente en la pantalla de selección de niveles.	
<b>Descripción</b>	El jugador debe ser capaz de elegir el nivel que desea jugar.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre alguno de los niveles disponibles en el juego.
	2	El sistema carga el nivel seleccionado y comienza ese nivel.
<b>Postcondición</b>	El jugador se encuentra en el nivel seleccionado previamente.	

<b>CU-03</b>	Pausar / Reanudar el tiempo	
<b>[Dependencias]</b>	CU-02	
<b>Precondición</b>	El jugador se encuentra jugando algún nivel actualmente.	
<b>Descripción</b>	El sistema debe permitir al jugador pausar / reanudar la partida.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el botón de pausar / reanudar.
	2	El sistema pausará / reanudará el juego según sea el caso.
<b>Postcondición</b>	La partida se encuentra pausada o reanudada según la situación previa de la misma.	

<b>CU-04</b>	Aumentar velocidad del tiempo	
<b>[Dependencias]</b>	CU-02, CU-03	
<b>Precondición</b>	El jugador se encuentra jugando algún nivel actualmente.	
<b>Descripción</b>	El sistema debe permitir al jugador aumentar la velocidad de la partida hasta un máximo.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el botón de aumentar velocidad.
	2	Si el jugador había realizado previamente el CU-03 pausando el juego.
	2.1	El sistema reanuda el tiempo en la partida.
	3	Si el sistema no se encuentra actualmente a la velocidad máxima.
	3.1	El sistema aumenta la velocidad del juego una cantidad determinada.
	En caso contrario.	
	3.2	El sistema reinicia la velocidad del juego a la velocidad normal.
<b>Postcondición</b>	La partida ha aumentado su velocidad o se ha reiniciado a la velocidad base.	

<b>CU-05</b>	Acceder al menú Opciones	
<b>[Dependencias]</b>	CU-02	
<b>Precondición</b>	El jugador se encuentra jugando algún nivel actualmente.	
<b>Descripción</b>	El sistema debe proporcionar al usuario un menú opciones donde aparecerán distintas acciones que este podrá realizar.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el botón de Opciones.
	2	El sistema pausa la partida si no estaba pausada ya.
	3	El sistema muestra la pantalla de Opciones.
<b>Postcondición</b>	El jugador se encuentra en la pantalla de Opciones.	

<b>CU-06</b>	Continuar la partida	
<b>[Dependencias]</b>	CU-05	
<b>Precondición</b>	El usuario ha realizado el CU-05.	
<b>Descripción</b>	El sistema debe permitir al jugador la opción de continuar la partida desde el menú Opciones.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el botón de continuar partida.
	2	El sistema cierra el menú de opciones.
	3	El sistema deja el tiempo de la partida como estaba antes de realizar el CU-05
<b>Postcondición</b>	El jugador se encuentra de nuevo jugando la partida en la que se encontraba.	

<b>CU-07</b>	Reintentar nivel	
<b>[Dependencias]</b>	CU-05	
<b>Precondición</b>	El usuario ha realizado el CU-05 o ha perdido la partida.	
<b>Descripción</b>	El sistema debe dar la opción de reintentar el nivel actual.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el botón de reintentar.
	2	El sistema vuelve a cargar el nivel inicial con las opciones iniciales del mismo.
<b>Postcondición</b>	El jugador ha comenzado desde 0 el nivel en el que se encontraba.	

<b>CU-08</b>	Volver a selección de niveles	
<b>[Dependencias]</b>	CU-05	
<b>Precondición</b>	El usuario ha realizado el CU-05 o ha perdido la partida.	
<b>Descripción</b>	El sistema debe dar la opción de volver a la pantalla de selección de niveles.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el botón de menú.
	2	El sistema vuelve a cargar la pantalla de selección de niveles.
<b>Postcondición</b>	El jugador se encuentra de nuevo en la pantalla inicial de selección de niveles.	

<b>CU-09</b>	Salir del juego	
<b>[Dependencias]</b>	CU-01, CU-05, CU-08	
<b>Precondición</b>	El jugador se encuentra en la pantalla de selección de niveles o en el menú de opciones.	
<b>Descripción</b>	El sistema debe dar la opción de cerrar el juego.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el botón de cerrar.
	2	El sistema cierra la aplicación.
<b>Postcondición</b>	El jugador ya no se encuentra dentro del juego.	

<b>CU-10</b>	Activar / Desactivar la Realidad Aumentada	
<b>[Dependencias]</b>	CU-02	
<b>Precondición</b>	El jugador se encuentra jugando algún nivel actualmente.	
<b>Descripción</b>	El sistema debe permitir al jugador activar y desactivar la realidad aumentada durante la partida.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el botón de Realidad Aumentada.
	2	El sistema activa / desactiva la realidad aumentada según el estado previo de esta.
<b>Postcondición</b>	El sistema tiene la realidad Aumentada activada / desactivada según el estado previo de esta.	

<b>CU-11</b>	Jugar sobre una imagen reconocible con realidad aumentada.	
<b>[Dependencias]</b>	CU-10	
<b>Precondición</b>	La realidad aumentada se encuentra activada.	
<b>Descripción</b>	El sistema debe permitir al jugador jugar la partida sobre una imagen reconocible por el juego.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador apunta con la cámara de su dispositivo a una imagen reconocible.
	2	El sistema reconoce la imagen y muestra la partida actual sobre la misma.
<b>Postcondición</b>	La partida está transcurriendo actualmente sobre una imagen reconocible.	

<b>CU-12</b>	Controlar la posición y el zoom de la cámara del juego.	
<b>[Dependencias]</b>	CU-10	
<b>Precondición</b>	La realidad aumentada se encuentra desactivada.	
<b>Descripción</b>	El sistema debe permitir al jugador controlar la posición y el zoom de la cámara.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	Si el jugador realiza un gesto de movimiento con los dedos en la pantalla.
	1.1	El sistema cambia la posición de la cámara según la dirección y la cantidad de movimiento del gesto realizado.
	2	Si el jugador mantiene pulsado dos dedos y los desliza acercándolos o alejándolos.
	2.1	El sistema aumenta o disminuye el zoom de la cámara según el gesto realizado.
<b>Postcondición</b>	El sistema ha modificado las propiedades de la cámara.	

<b>CU-13</b>	Seleccionar un tipo de torreta	
<b>[Dependencias]</b>	CU-02	
<b>Precondición</b>	El jugador se encuentra jugando algún nivel actualmente.	
<b>Descripción</b>	El sistema debe permitir al jugador seleccionar distintos tipos de torretas.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre el botón asignado a un tipo de torreta concreto.
	2	El sistema cambia el tipo de torreta seleccionado previamente a el que acaba de pulsar el jugador.
<b>Postcondición</b>	El jugador tiene el tipo de torreta que desea seleccionado.	
<b>Comentarios</b>	Al iniciar un nivel, por defecto está seleccionado el tipo de torreta básica.	

<b>CU-14</b>	Colocar una torreta	
<b>[Dependencias]</b>	CU-02, CU-13	
<b>Precondición</b>	El jugador se encuentra jugando algún nivel actualmente y ha seleccionado la torreta que desea colocar.	
<b>Descripción</b>	El sistema debe permitir al jugador colocar torretas en las casillas vacías del tablero de juego.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre una casilla vacía del tablero de juego.
	2	Si el jugador dispone de las monedas necesarias para comprar la torreta que tiene seleccionada.
	2.1	El sistema coloca una torreta en la casilla que ha seleccionado el jugador.
	2.2	El sistema quita al jugador las monedas que ha costado comprar esa torreta.
<b>Postcondición</b>	Se ha colocado una nueva torreta en el tablero de juego.	

<b>CU-15</b>	Mejorar una torreta	
<b>[Dependencias]</b>	CU-14	
<b>Precondición</b>	El jugador tiene alguna torreta colocada en el tablero.	
<b>Descripción</b>	El sistema debe permitir al jugador mejorar torretas aumentando las capacidades de estas. Hay un máximo de mejoras por torreta.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre una torreta colocada previamente.
	2	El jugador pulsa sobre el botón de mejorar que aparece sobre la torreta seleccionada.
	3	Si el jugador dispone de las monedas necesarias para mejorar la torreta seleccionada.
	3.1	El sistema mejora la torreta seleccionada.
	3.2	El sistema quita al jugador las monedas que ha costado la mejora.
<b>Postcondición</b>	Se ha colocado una nueva torreta en el tablero de juego.	

<b>CU-16</b>	Vender una torreta	
<b>[Dependencias]</b>	CU-14	
<b>Precondición</b>	El jugador tiene alguna torreta colocada en el tablero.	
<b>Descripción</b>	El sistema debe permitir al jugador vender torretas colocadas previamente en el tablero.	
<b>Secuencia Normal</b>	<b>Paso</b>	<b>Acción</b>
	1	El jugador pulsa sobre una torreta colocada previamente.
	2	El jugador pulsa sobre el botón de vender que aparece sobre la torreta seleccionada.
	3	El sistema elimina la torreta del tablero.
	4	El sistema devuelve al jugador una cantidad concreta de monedas.
<b>Postcondición</b>	Se ha eliminado una torreta del tablero.	

#### 1.4.5. Requisitos de información

A partir de los casos de uso y los requisitos generales podemos deducir los siguientes requisitos de información del sistema:

<b>RI-01</b>	Estado del tablero
<b>Descripción</b>	El sistema deberá almacenar y conocer en todo momento el estado actual del tablero, conociendo el contenido de cada una de las casillas, si hay una torreta seleccionada, qué torretas hay colocadas, etc.

<b>RI-02</b>	Vidas
<b>Descripción</b>	El sistema deberá gestionar y conocer en todo momento la vida del jugador y comportarse de una forma u otra de acuerdo a esta.

<b>RI-03</b>	Monedas
<b>Descripción</b>	El sistema deberá gestionar y conocer en todo momento las monedas del jugador y actualizarse de acuerdo con las acciones que las afectan.

<b>RI-04</b>	Velocidad del juego
<b>Descripción</b>	El sistema deberá gestionar y conocer en todo momento la velocidad actual del juego.

<b>RI-05</b>	Realidad aumentada
<b>Descripción</b>	El sistema deberá conocer en todo momento el estado actual de la Realidad aumentada, para poder permitir activarla y desactivarla.

<b>RI-06</b>	Enemigos
<b>Descripción</b>	El sistema deberá gestionar y conocer en todo momento el estado de los enemigos presentes en el tablero.

<b>RI-07</b>	Torretas y proyectiles
<b>Descripción</b>	El sistema deberá gestionar y conocer en todo momento el estado de las torretas colocadas en el tablero, así como sus proyectiles.

<b>RI-08</b>	Niveles
<b>Descripción</b>	El sistema deberá almacenar de alguna forma los distintos niveles disponibles del juego.

<b>RI-09</b>	Oleadas
<b>Descripción</b>	El sistema deberá gestionar y conocer en todo momento la información referente a las oleadas de enemigos.

#### 1.4.6. Requisitos funcionales

A continuación, describimos los requisitos funcionales identificados a partir de los requisitos generales y los casos de uso.

<b>RF-01</b>	Selección y carga de niveles
<b>Descripción</b>	El jugador debe ser capaz de seleccionar un nivel en la pantalla de inicio y este se debe cargar adecuadamente.

<b>RF-02</b>	Controlar velocidad del juego. Pausar, continuar, incrementar.
<b>Descripción</b>	El jugador debe ser capaz de controlar la velocidad del juego, esto incluye pausar la partida, reanudarla, y acelerar la velocidad de la misma.

<b>RF-03</b>	Salir del juego
<b>Descripción</b>	El jugador debe ser capaz de salir del juego cerrando la aplicación desde dentro.

<b>RF-04</b>	Reintentar nivel
<b>Descripción</b>	El jugador debe disponer de una opción que le permita reiniciar el nivel actual.

<b>RF-05</b>	Realidad Aumentada
<b>Descripción</b>	El jugador debe ser capaz de jugar al juego con y sin realidad aumentada activada, haciendo uso de los elementos necesarios para esto.

<b>RF-06</b>	Torretas
<b>Descripción</b>	El jugador debe ser capaz de comprar torretas, mejorarlas y venderlas. Siempre haciendo uso de las monedas actuales que disponga. Además, debe poder ver el alcance de las torretas.

<b>RF-07</b>	Cámara
<b>Descripción</b>	El jugador debe ser capaz de controlar el movimiento y zoom de la cámara del juego durante una partida.

<b>RF-08</b>	Generación del tablero de juego
<b>Descripción</b>	El sistema debe ser capaz de generar un tablero de casillas hexagonales de tamaño configurable.

<b>RF-09</b>	Tipos de casillas
<b>Descripción</b>	El sistema debe ser capaz de soportar los siguientes tipos de casillas: <ul style="list-style-type: none"> <li>- Casilla vacía.</li> <li>- Casilla bloqueante. No permite la interacción del jugador.</li> <li>- Pared. Bloquea el paso de enemigos.</li> <li>- Generador de enemigos.</li> <li>- Destino de enemigos.</li> </ul>

<b>RF-10</b>	Tipos de enemigos
<b>Descripción</b>	El sistema debe ser capaz de soportar distintos tipos de enemigos.

<b>RF-11</b>	Tipos de torretas
<b>Descripción</b>	<p>El sistema debe ser capaz de soportar los siguientes tipos de torretas (basados en el juego de referencia):</p> <ul style="list-style-type: none"> <li>- Torreta básica: dispara proyectiles a un único objetivo.</li> <li>- Torreta láser: dispara un láser que daña a todos los objetivos que atraviese.</li> <li>- Torreta misiles: dispara misiles perseguidores que explotan al alcanzar a un enemigo.</li> <li>- Torreta área: hace daño en área a todos los enemigos a su alcance.</li> <li>- Torreta ralentizadora: ralentiza a todos los enemigos a su alcance.</li> </ul>

<b>RF-12</b>	Diseño de niveles
<b>Descripción</b>	El sistema debe proporcionar alguna forma de diseño de niveles, ya sea dentro del juego o dentro de la herramienta Unity.

<b>RF-13</b>	Perder partida
<b>Descripción</b>	El sistema debe controlar el flujo del juego haciendo que este se detenga cuando las vidas del jugador lleguen a 0. Tras esto, debe dar la opción de reiniciar el nivel o volver a la selección de niveles.

#### 1.4.7. Requisitos no funcionales

<b>RNF-01</b>	Android
<b>Descripción</b>	El sistema debe ser portado únicamente a Android generando un archivo .apk

RNF-02	Permisos de aplicación
<b>Descripción</b>	El sistema debe solicitar al jugador los permisos necesarios de acceso a la cámara para poder hacer uso de esta con la realidad aumentada.

#### 1.4.8. Asunciones y excepciones

Se considera fuera del alcance y los objetivos del proyecto:

- Comercialización del juego, subida a Play Store, búsqueda de clientes potenciales y estudio de mercado.
- Todo el contenido utilizado de terceros debe proceder de fuentes sin copyright o de fuentes que autoricen su uso dando crédito a los creadores.
- Debido a la nula experiencia del desarrollador en el sector de música y audio. Se considera opcional añadir contenido auditivo al juego.

## 2. Ejecución del proyecto

### 2.1. Diseño del Sistema

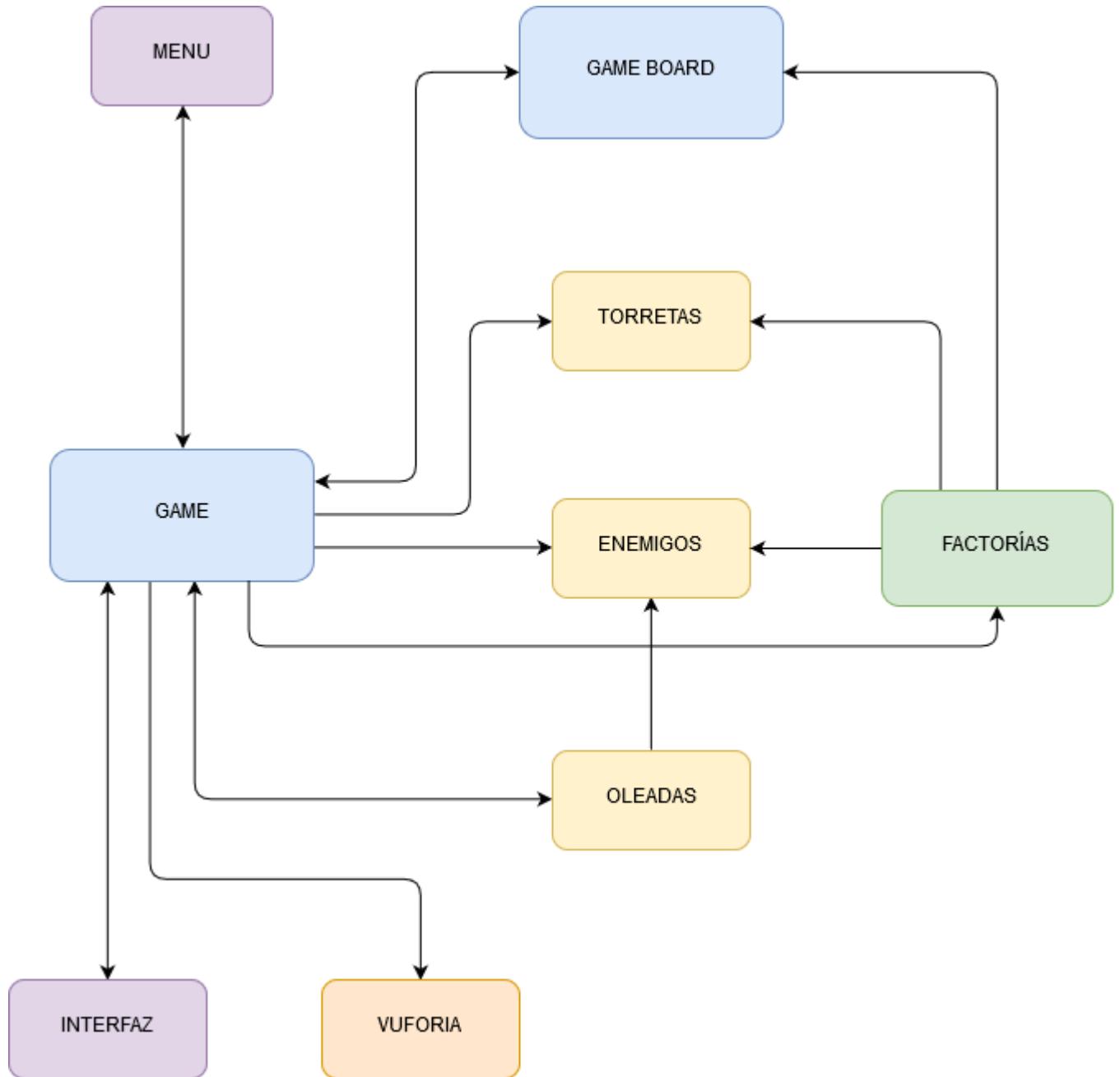


Imagen 7 - Esquema general del sistema desarrollado

Los componentes principales del sistema y la función de estos son:

**Menu:**

Es el elemento de entrada al iniciar el juego y permite la selección de los distintos niveles de este.

**Game / Juego:**

Es el componente principal del sistema, se encarga de gestionar el transcurso de la partida y orquestar el orden en el que se ejecutan los distintos elementos de esta. Almacena gran cantidad de datos relevantes para el transcurso de la partida y está conectado con prácticamente todo el resto de los componentes del juego. También hace de puente entre distintos componentes del sistema. Proporcionándoles información y otros recursos de utilidad, como por ejemplo, instancias de proyectiles o información sobre el número de enemigos vivos actualmente.

**Game Board / Tablero de juego:**

Se encarga de toda la gestión del tablero de casillas del juego. Es el encargado de la generación del tablero de juego y el contenido de las casillas de este: puntos de aparición, destinos de enemigos, paredes, torretas, casillas bloqueantes, etc.

También coordina el sistema de búsqueda de caminos para los enemigos, el cual se recalcula cada vez que se produce un cambio que lo requiere. Este algoritmo tiene algunas restricciones de validación como veremos más adelante.

**Factorías:**

Son las encargadas de gestionar y reunir de forma ordenada y lógica los distintos elementos del juego que se van generando y eliminando sobre la marcha durante la partida.

Existen 3 factorías en el juego: Enemigos, Contenido de casillas, Proyectiles. Cada una de ellas tiene referencias a instancias de objetos creados previamente en el editor, estos son los que serán instanciados y recolectados cada vez que sea necesario.

**Enemigos:**

Dirigidos por el componente Game, su objetivo es alcanzar una casilla de destino. Todos los enemigos generados tienen un funcionamiento similar, sólo se diferencian entre ellos por los valores de los atributos que rigen su comportamiento.

**Torretas:**

Dirigidas por el componente Game, su objetivo es impedir la llegada de los enemigos a una casilla de destino.

## Interfaz

Se encarga de permitir la interacción entre el jugador y el juego. Además, muestra información relevante sobre la partida. Dentro de esta, se encuentra también la selección de torretas, que es la interfaz de torretas colocadas en el tablero, se encarga de mostrar al jugador el rango de las torretas colocadas y permitir su mejora o eliminación usando monedas.

## Sistema de oleadas:

Es el componente encargado de gestionar el sistema de oleadas, su generación y transcurso. Está formado por componentes de tipo Wave, que reúnen toda la información de las oleadas de forma independiente.

## Vuforia:

Se encarga de controlar el funcionamiento de la realidad aumentada del juego.

## 2.2. Implementación

### 2.2.1. Tecnologías empleadas

Esta sección trata de las tecnologías utilizadas en el proyecto y las características de estas.

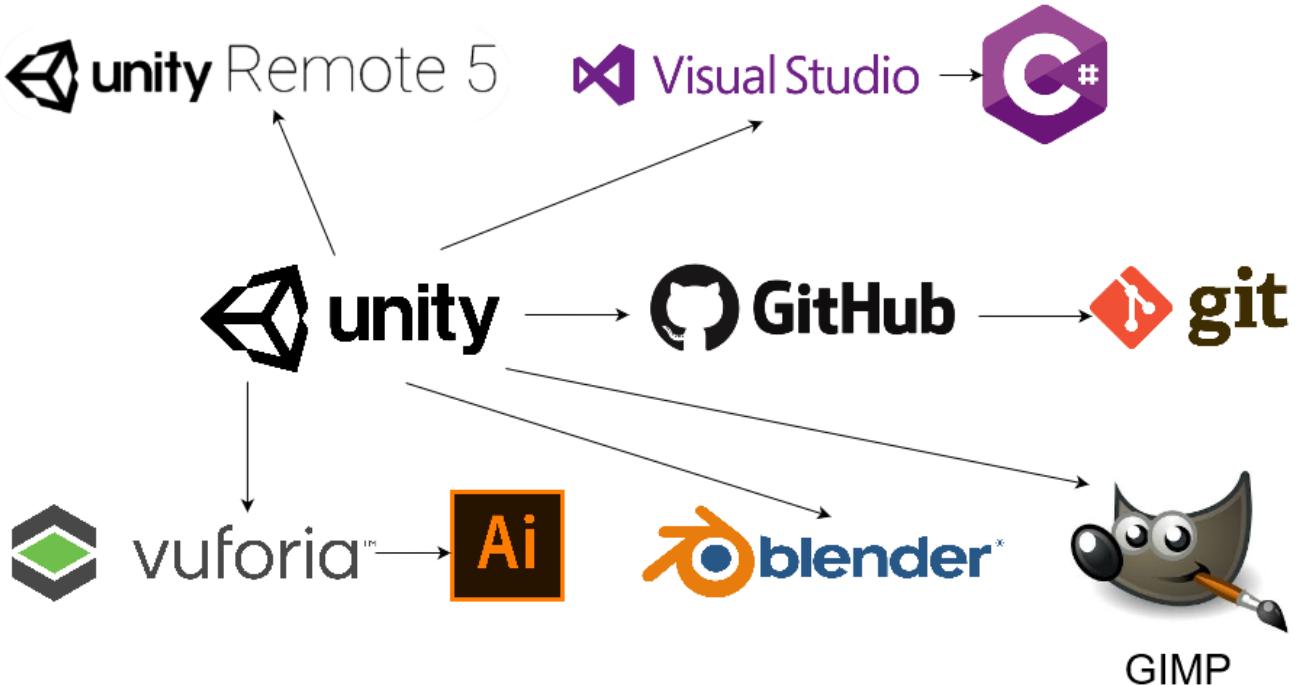


Imagen 8 - Mapa de herramientas utilizadas

#### Unity 3D

Es una plataforma de desarrollo de videojuegos, aunque también es bastante utilizada para realizar proyectos de investigación y de otras áreas no relacionadas con los videojuegos. Esto es porque ofrece un entorno bastante amigable para los usuarios y es bastante sencillo y rápido prototipar lo que se tenga en mente. Se puede utilizar de forma gratuita siempre y cuando tus ingresos recaudados o autofinanciados no superen los \$100.000 al año.

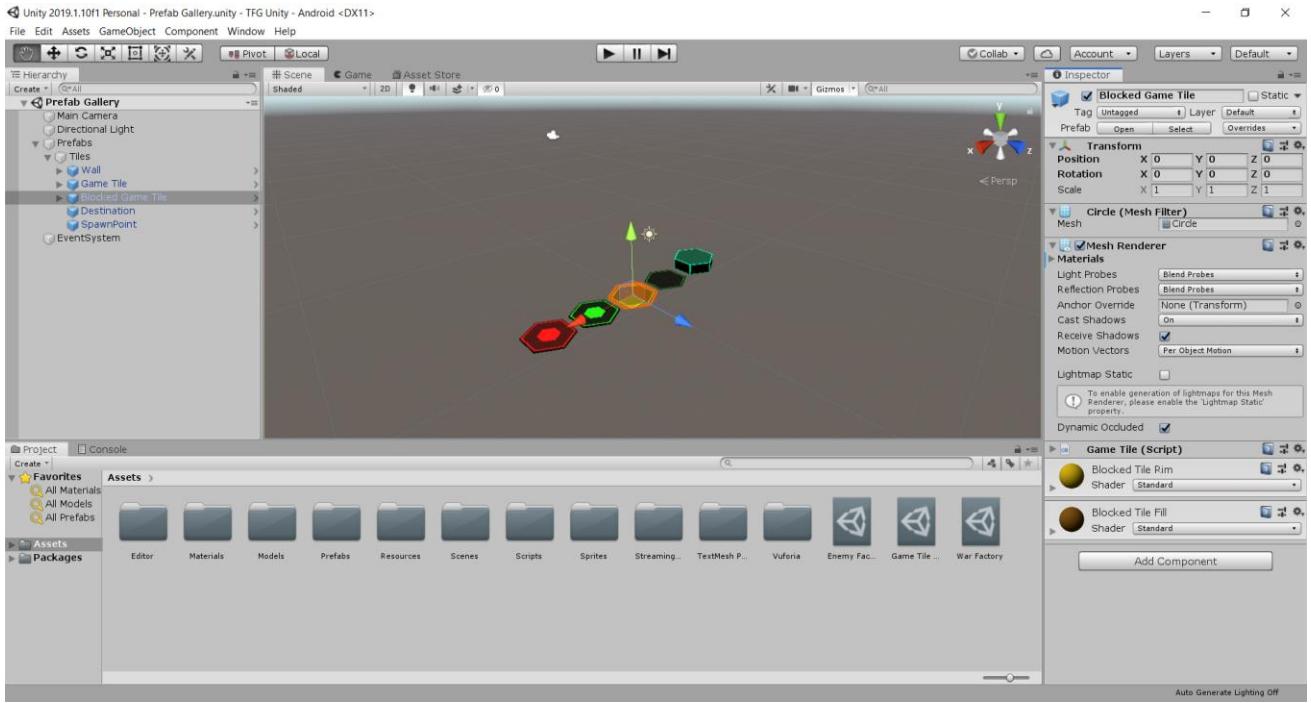


Imagen 9 - Interfaz de Unity

Es un editor todo en uno, disponible para Windows, Mac y Linux. Incluye gran cantidad de herramientas tanto para artistas como desarrolladores, permitiéndoles crear experiencias envolventes y lógicas de juego de alto nivel y rendimiento. Unity da soporte al desarrollo de entornos 2D y 3D, aportando funcionalidades específicas para cada uno de estos.

Gracias al uso de los llamados ‘Prefabs’, que son ‘Game Objects’ preconfigurados, se puede seguir un flujo de trabajo flexible y eficiente, permitiendo configurar estos objetos sin necesidad de cambiar los valores en todas las instancias de estos de forma individual.

Los Objetos de los juegos en Unity (‘GameObjects’ a partir de ahora) funcionan con componentes que se pueden incluir en los objetos, estos pueden ser activados y desactivados durante el transcurso del juego. Unity trae por defecto una gran cantidad de componentes que puedes añadir sin necesidad de haber importado ningún paquete de contenido previamente.

Los scripts creados para los GameObjects que quieran funcionar como componentes siempre deben heredar de ‘MonoBehaviour’, permitiendo el acceso a unos métodos específicos que son llamados en el flujo de programación de los juegos de Unity. A continuación, se muestra una imagen con algunos de estos métodos y el orden en el que son llamados.

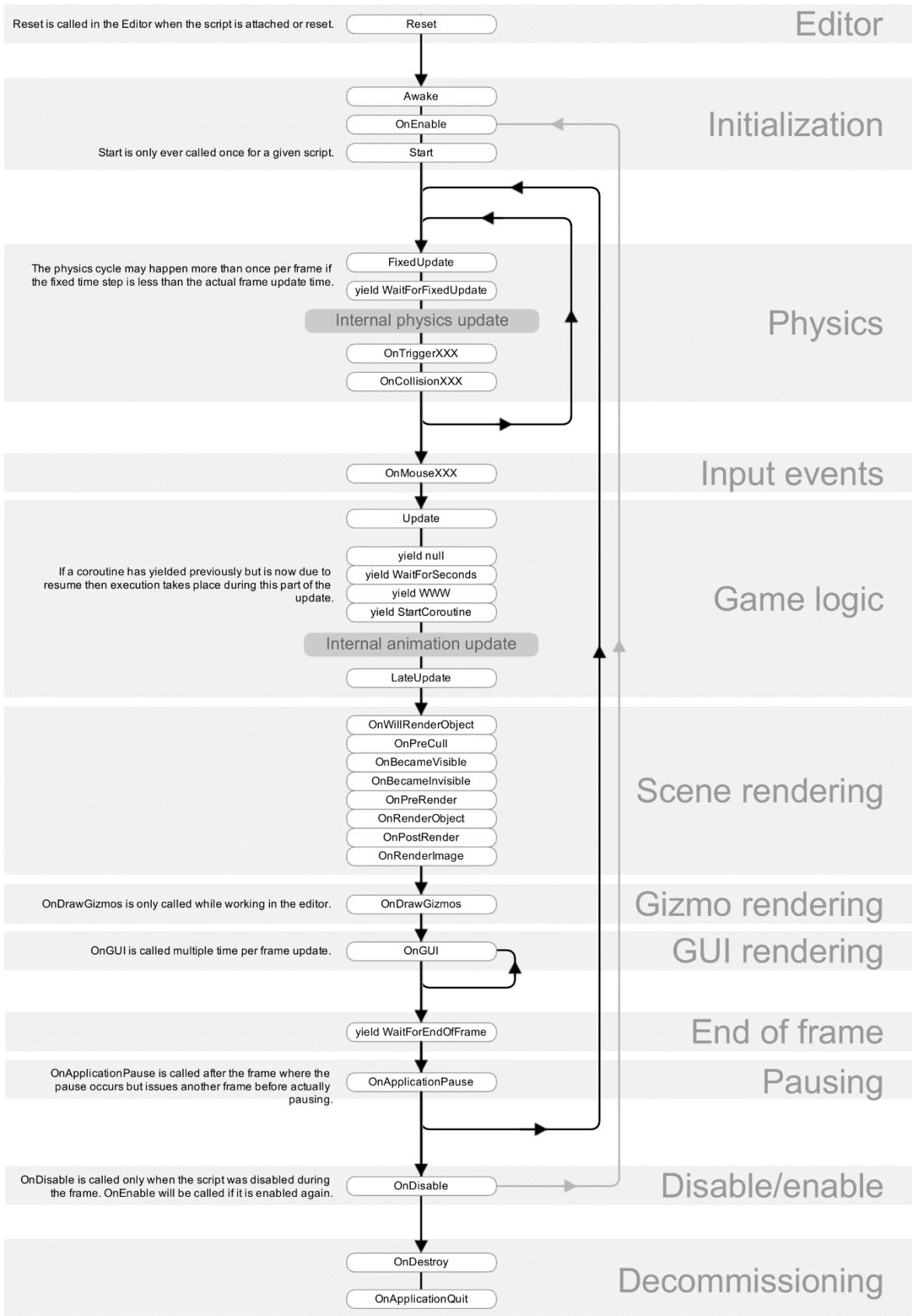


Imagen 10 - Flujo de llamadas a métodos de Unity

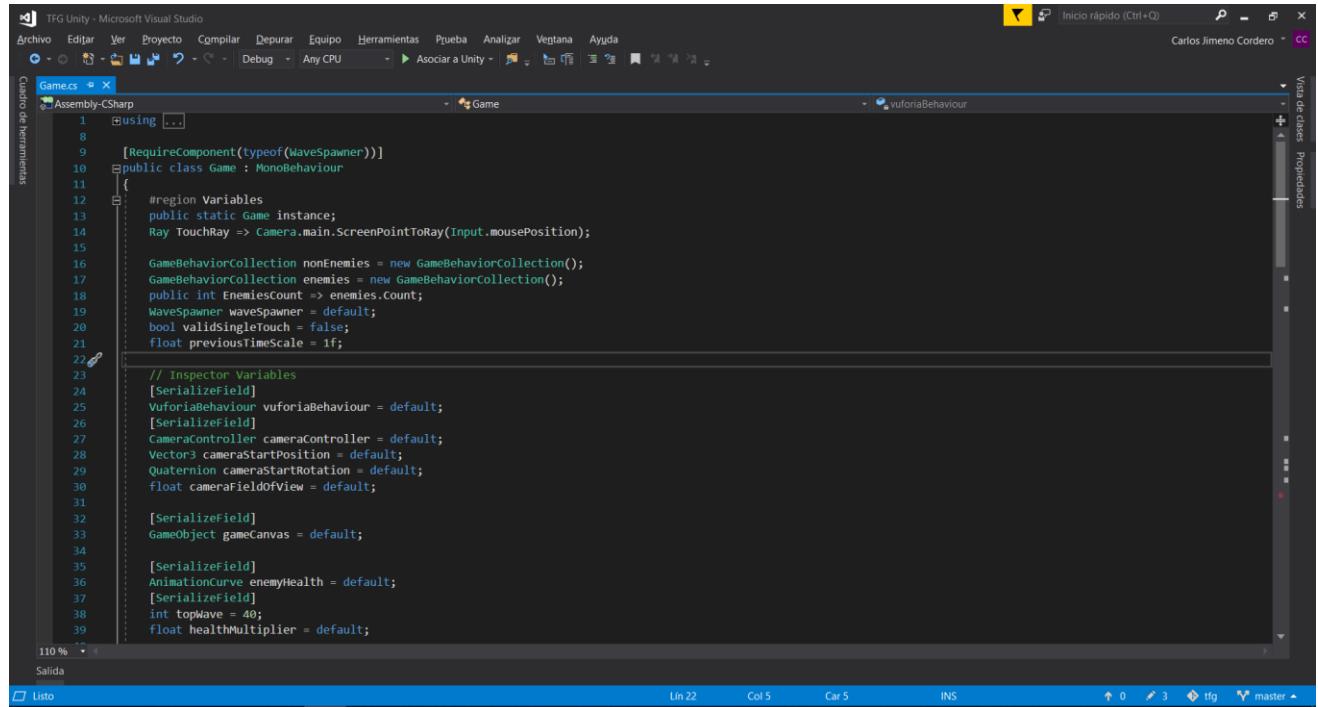
## Visual Studio Community

Es un entorno de desarrollo integrado compatible con gran cantidad de lenguajes de programación. Al instalarlo junto con Unity se nos ofrece la posibilidad de instalar herramientas de integración con Unity ‘Visual Studio Tools for Unity’, estas incluyen funcionalidades como:

- Abrir scripts y proyectos Unity.
- Acceder rápidamente a la documentación de código de Unity resaltando antes lo que deseamos buscar.
- Intellisense para las librerías de Unity, basta con comenzar a escribir para que Visual Studio sugiera soluciones a lo que estás escribiendo, facilitando el desarrollo y aportando información sobre qué hace lo que estás escribiendo.

Además, permite la depuración de código junto con Unity pulsando en ‘Asociar a Unity’ y poniendo puntos de ruptura donde deseemos. Nos ofrece la posibilidad de realizar evaluaciones sobre la marcha una vez se haya parado donde queremos.

También incluye snippets o plantillas para estructuras básicas de control, bucles for, foreach, while, if, y autocompletado de estructuras switch case en caso de que te reconozca todas las posibilidades, por ejemplo, usando los valores de un tipo Enum.



```

1  using UnityEngine;
2
3  [RequireComponent(typeof(WaveSpawner))]
4  public class Game : MonoBehaviour
5  {
6      #region Variables
7      public static Game instance;
8
9      Ray TouchRay = Camera.main.ScreenPointToRay(Input.mousePosition);
10
11     GameBehaviorCollection nonEnemies = new GameBehaviorCollection();
12     GameBehaviorCollection enemies = new GameBehaviorCollection();
13     public int EnemiesCount => enemies.Count;
14     WaveSpawner waveSpawner = default;
15     bool validSingleTouch = false;
16     float previousTimeScale = 1f;
17
18     // Inspector Variables
19     [SerializeField] VuforiaBehaviour vuforiaBehaviour = default;
20     [SerializeField] CameraController cameraController = default;
21     Vector3 cameraStartPosition = default;
22     Quaternion cameraStartRotation = default;
23     float cameraFieldOfView = default;
24
25     [SerializeField] GameObject gameCanvas = default;
26
27     [SerializeField] AnimationCurve enemyHealth = default;
28     [SerializeField] int topWave = 40;
29     float healthMultiplier = default;
30
31
32
33
34
35
36
37
38
39

```

Imagen 11 - Interfaz Visual Studio Community

## C#

Es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft. Su sintaxis deriva de C/C++ y utiliza un modelo de objetos similar al de Java. Fue diseñado para la infraestructura del lenguaje común (CLI), lo que hace que sea un lenguaje cómodo de ejecutar en distintas plataformas sin necesidad de reescribir o recompilar.



Imagen 12 - Logo C#

## Vuforia

Software de Realidad Aumentada integrado con Unity, permite crear experiencias de realidad aumentada de manera sencilla y rápida. Incluye funcionalidades como:

- Reconocimiento de objetivos: imágenes, modelos, cilindros, objetos escaneados, vumarks.
- Plano del suelo: reconoce superficies horizontales que se podrán usar como base donde colocar el contenido.
- Botones interactivos: botones que se pueden pulsar virtualmente.

En el portal de desarrollo permite crear bases de datos con imágenes reconocibles que podrás usar más adelante en tus proyectos. Incluye proyectos de prueba que puedes utilizar para comenzar rápidamente el aprendizaje de la tecnología.

Además, proporcionan algunas herramientas de utilidad como generadores de imágenes 'vumark', que vienen a ser imágenes codificadas que representan algo, por ejemplo, un número, una cadena de texto, etc.



Imagen 13 - Logo vuforia

## Unity Remote 5

Unity Remote es una aplicación de Unity que te permite probar los desarrollos tal y como se haría desde un PC dándole al modo Play, pero con la comodidad de poder usar un teléfono u otro dispositivo, pudiendo así probar funcionalidades que no se pueden probar desde el ordenador, como controles táctiles.

Por desgracia, la depuración en modo Play de Vuforia no permite usar la cámara del dispositivo cuando se está usando Unity Remote, por lo que obliga a usar una webcam o tener que montar una 'build' si quieras probar algo con la cámara del móvil.



Imagen 14 - Interfaz Unity Remote 5

## Blender

Es una herramienta de software libre que permite modelar, animar, renderizar, componer gráficos 3D y 2D. Actualmente es la herramienta de diseño 3D gratuita más conocida y usada en el mundo, tiene características que la hacen equiparable a el resto de las herramientas de pago existentes.

Llevaba bastantes años en la versión 2.7, que era una versión bastante estable e incluía ya gran cantidad de operaciones de todos los géneros. No obstante, era una versión quizás un poco intimidante por su apariencia e interfaz. Hace muy poco, sacaron oficialmente la versión 2.8, que llevaba un tiempo en una versión beta. Con esta nueva versión, le han dado un nuevo enfoque a la aplicación invirtiendo mucho más en la interfaz y cómo de intuitiva es esta.

*Imagen 15 - Logo Blender*

### Gimp

Es un programa de edición de imágenes digitales en forma de mapa de bits, es libre y gratuito. Incluye características como edición por capas, filtros, transformaciones, dibujo de formas libres, convertir a distintos formatos de imagen, etc. Se usará a nivel básico para editar imágenes que serán usadas principalmente para la interfaz del juego.

*Imagen 16 - Logo Gimp*

### Adobe Illustrator

Es un programa de dibujo vectorial, no es gratis, pero incluye una prueba de 30 días gratis. Lo usaremos únicamente para crear una plantilla de generación de imágenes VuMark.

*Imagen 17 - Logo Adobe Illustrator*

## Github

Es una plataforma de desarrollo colaborativo que hace uso del sistema de control de versiones Git. Se utilizará para alojar el proyecto en su totalidad, subiendo iterativamente los cambios para poder volver atrás si fuera necesario y llevar el historial de avance del proyecto. Se utilizará únicamente una rama, la rama master, ya que solo estoy yo programando y no hay intención de divergir la dirección del proyecto durante el desarrollo. Permite entre otras cosas crear repositorios privados para grupos pequeños de forma gratuita.



*Imagen 18 - Logo Github*

## Git

Sistema de control de versiones. Junto con Github, nos permitirá crear un repositorio privado para el proyecto, clonarlo en nuestra máquina local, añadir elementos para subir, eliminar elementos existentes, renombrar, hacer commits reflejando el estado actual del proyecto, y muchas cosas más. En nuestro caso no vamos a hacer nada “elegante” fuera de lo normal, los comandos que usaremos serán:

- git clone: clona el repositorio remoto en la ruta que deseemos.
- git add: añade cambios a la etapa previa antes de hacer un commit.
- git commit -m “título del commit”: crea un commit con los cambios añadidos actualmente.
- git push: sube los cambios a remoto.
- git reset --hard: descarta los cambios en local desde el último commit.



*Imagen 19 - Logo git*

## 2.2.2. Desarrollo

En esta sección explicaremos de forma detallada todos los componentes del juego y el funcionamiento de cada uno de ellos.

### GameTile

Las casillas son el componente base del tablero, tienen conocimiento de sus vecinos y también de cuál es la mejor dirección para alcanzar una casilla de destino. Es el objeto GameBoard el encargado de computar el camino más corto entre cada casilla y la casilla más cercana de destino. A nivel visual, se le añadió a cada casilla una flecha que más adelante apuntaría hacia el vecino con la distancia más corta. Esta funcionalidad no se usa en el juego final, aunque está disponible en todo momento en el editor de Unity pulsando la tecla V en el modo de juego.

Además de la lógica de caminos y vecinos, albergan la relación con el contenido que tienen (si lo tienen). Más adelante seguiremos hablando de esto.

Existen dos tipos de casillas, las normales y las bloqueantes, estas últimas impiden la interacción del jugador con las mismas, pero no impiden a los enemigos pasar sobre ellas.

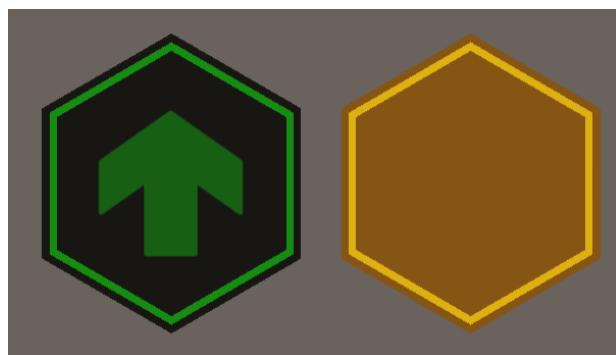
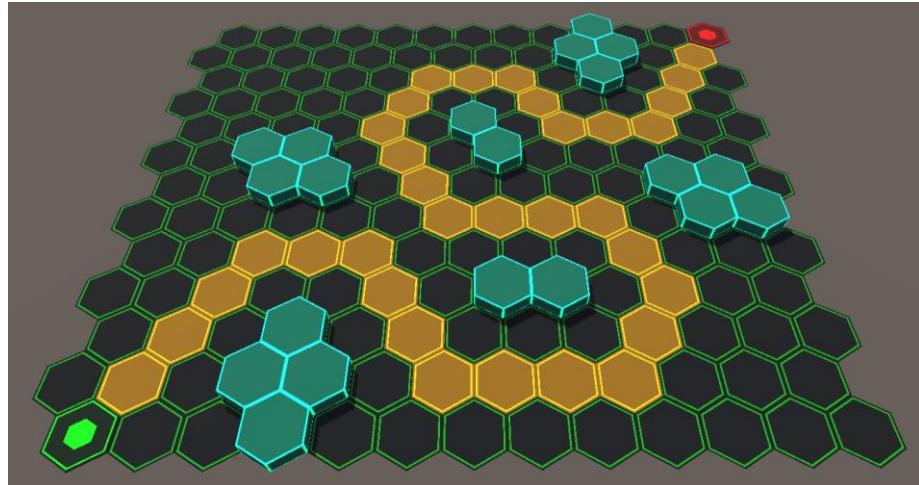


Imagen 20 - Casilla normal y bloqueante

## GameBoard



*Imagen 21 - Tablero del juego*

Es un elemento esencial para el sistema, sin él no hay juego, está dirigido por el Objeto GameBoard, que se encarga de la generación, actualización y gestión de las casillas y su contenido. Al comenzar una partida el objeto Game le solicitará su creación e inicialización a partir de unos parámetros de tamaño horizontal y vertical que representan el número de casillas en cada dirección. Este creará un tablero de casillas vacías y actualizará su contenido para que haya al menos un punto de salida de enemigos y un destino. En la inicialización del tablero, se asocia a cada casilla con sus vecinos y eso es algo que no va a ser alterado durante el resto de la partida.

A el algoritmo de inicialización se le añadió también la funcionalidad de utilizar una cadena de texto como plantilla, esta cadena debe encajar a la perfección con el tamaño del tablero, es decir, debe tener ‘ancho’ x ‘alto’ número de caracteres y debe generar un tablero válido, lo cual definiremos más adelante en el algoritmo de búsqueda de caminos. Cada carácter de la cadena representa el contenido de cada casilla. Estos son: ‘e’ para una casilla vacía, ‘s’ para una casilla con contenido de tipo aparición de enemigos, ‘d’ para una casilla con contenido de tipo destino, ‘b’ para una casilla de tipo bloqueante y ‘w’ para una casilla con contenido de tipo pared.



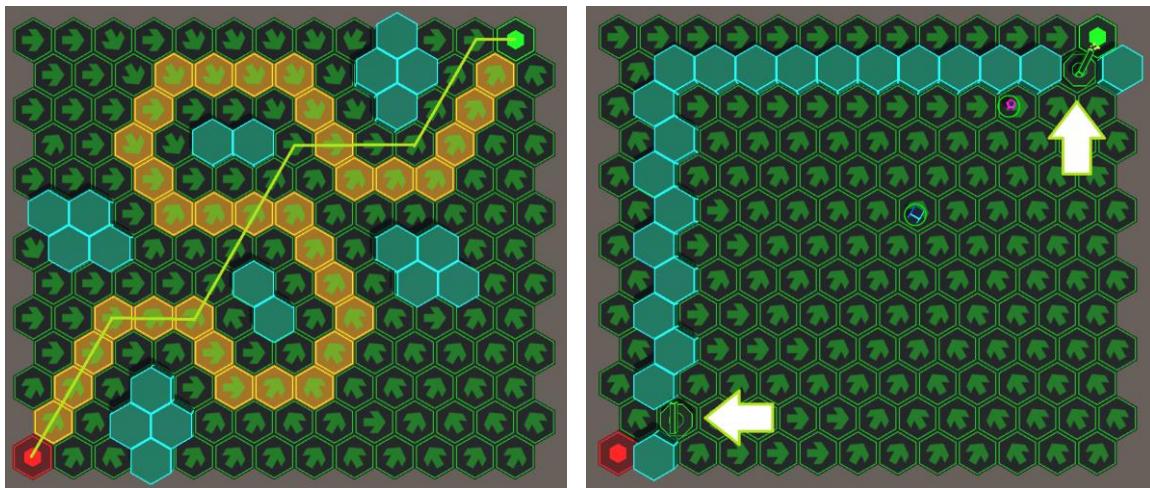
*Imagen 22 - Casillas del juego y algunos contenidos*

En la imagen anterior, además de las casillas normales y bloqueantes, se muestran tres tipos de contenidos estáticos, de izquierda a derecha son:

- Contenido – Pared: bloquea el avance de los enemigos.
- Casilla vacía.
- Casilla bloqueante.
- Contenido - Destino: sirve de ubicación de destino para los enemigos.
- Contenido – Generador de enemigos: sirve de punto de aparición para enemigos.

Una vez se han creado las relaciones y todas las casillas están colocadas con su contenido establecido, el tablero pasa a ejecutar el algoritmo de búsqueda de caminos, este es un método que devuelve verdadero o falso, según si la búsqueda ha tenido éxito o no. El funcionamiento es el siguiente:

1. Añadir las casillas de destino a una lista frontera y las establece como casillas destino a nivel interno.
2. Limpia el camino que tengan establecido las casillas que no sean destino.
3. Si la lista frontera está vacía para la búsqueda y devuelve falso.
4. Por cada casilla de la frontera, se expande su camino hacia las casillas vecinas y se reordena la frontera utilizando como valor la distancia más cercana hacia una casilla de destino. Se pretende que las casillas con menor distancia sean las primeras en expandirse. Las casillas con contenido de tipo Pared bloquean la expansión sobre ellas mismas, pero no hacia ellas.
5. Una vez finalizada la expansión, se comprueba por cada punto de aparición de enemigos, que haya un camino libre hasta al menos una casilla de destino. No se comprueba en la dirección opuesta ya que las casillas no están diseñadas para apuntar hacia puntos de aparición. Esto permitiría al jugador bloquear uno de los puntos de destino totalmente forzando a los enemigos a ir a otros puntos que sí estén libres. En el caso de que algún punto de aparición no disponga de un camino hacia una casilla de destino, se devuelve falso.
6. Por último, en caso de que estemos jugando una partida, se comprobará si el jugador ha intentado encerrar a un enemigo en una región. Para ello se mirará las casillas que no tengan camino y se les establecerá a fuerza bruta el camino que tenían inicialmente al empezar la partida. Este paso se ignora en la inicialización.
7. Como extra, si la funcionalidad visual de las flechas de las casillas está activada, se actualizará la dirección de estas y se mostrarán.

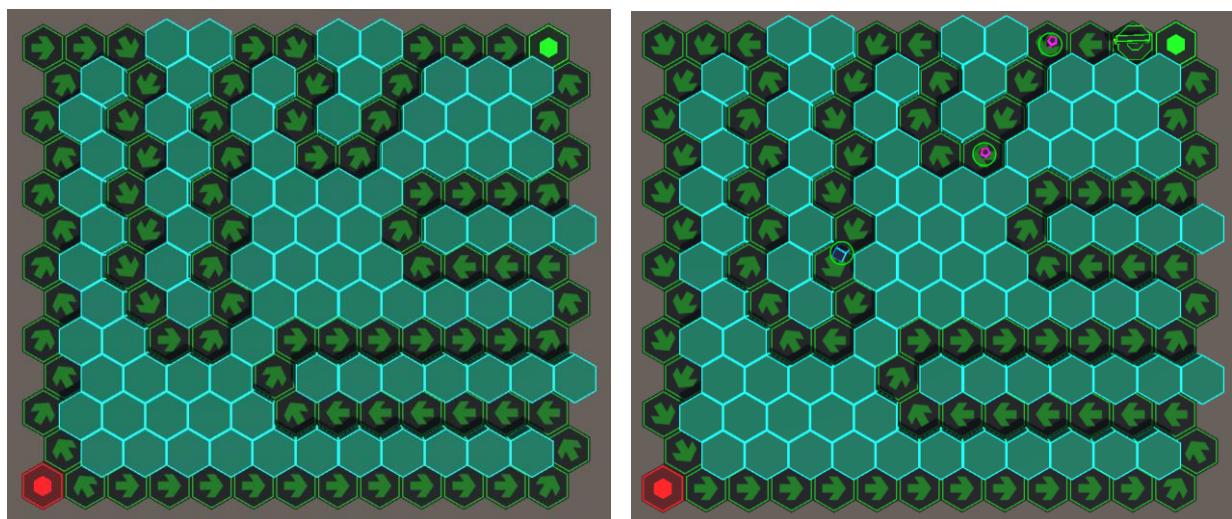


*Imagen 23 - Ejemplo de búsqueda de caminos y bloqueo a enemigos*

Este algoritmo fue un poco problemático al principio ya que en lugar de los puntos 5 y 6, se comprobaba si todas las casillas tenían camino y en caso contrario se devolvía falso. Esto impedía crear conjuntos grandes de paredes, o torretas que son los dos tipos de contenido que bloquean caminos. Es decir, una casilla no podía ser rodeada totalmente por contenido bloqueante.

Finalmente se decidió sustituir esta comprobación por los puntos 5 y 6 que dejan al jugador total libertad para colocar torretas donde quiera siempre y cuando no bloquee los caminos desde la aparición a los destinos.

El algoritmo también tiene una vulnerabilidad claramente visible que es que el jugador puede ‘marear’ constantemente a los enemigos actualizando la dirección en la que tienen que ir, esto se haría colocando torretas en sitios estratégicos y eliminándolas. Contra este problema no se ha llevado a cabo ninguna acción.



*Imagen 24 - Ejemplo 'mareo' de enemigos*

## Enemigos

Su único objetivo es el de llegar desde la casilla de aparición donde nazcan hasta una casilla de destino, si lo consiguen el jugador perderá una vida. ¿Cómo consiguen llegar hasta allí?, pues haciendo uso del camino calculado que tienen las casillas del tablero. Cada enemigo es consciente de qué casilla viene, cuál es la siguiente casilla a la que va y del progreso que lleva en la transición entre esas dos casillas.

Los enemigos serán los primeros elementos del juego que veamos que tienen su ciclo de vida definido en un método personalizado llamado ‘GameUpdate’, este funcionaría igual que el método ‘Update’ de Unity que se llama en cada frame del juego, solo que nosotros decidiremos cuándo se llama a este. Es decir, contendrá el comportamiento normal del enemigo en cada frame.

Este tipo de mecánica la hemos incluido en otros objetos del juego y la hemos agrupado en ‘GameBehaviourCollection’, que gestiona dentro de sí misma una lista de componentes de tipo ‘GameBehaviour’. Este último es tan solo una clase de la que heredan todos los demás y tiene un método virtual vacío llamado ‘GameUpdate’ que debe devolver un booleano, cada uno de los elementos de la lista deberá este método sobrescribir adaptándolo a sus necesidades.

El objeto de tipo ‘GameBehaviourCollection’ también tiene un método llamado ‘GameUpdate’, que llamará uno a uno al de los elementos de la lista, y en caso de devolver falso, removerá ese elemento de la lista de forma eficiente. Intercambiándolo con el último elemento de la lista y borrando la última posición de esta.

El flujo de vida de los enemigos siempre es el mismo:

1. Compruebo si mi vida es mayor a 0, en caso contrario, me destruyo y devuelvo falso.
2. Aumentamos el progreso que llevemos por (velocidad x Time.deltaTime). Variable explicada más adelante.
3. Mientras el progreso sea mayor que 1, actualizamos la siguiente casilla a la que tenemos que ir y restamos 1 al progreso, sólo se resta uno para que en caso de habernos pasado, por ejemplo 1.2f, no perdamos ese 0.2f restante creando movimientos extraños. En caso de que hayamos llegado a una casilla destino, nos destruimos, hacemos una llamada al objeto Game indicando que el jugador debe perder una vida y finalmente devolvemos falso. Este bloque se hace con un bucle ‘while’ para permitir que podamos atravesar varias casillas en un mismo frame.
4. Por último, actualizamos nuestra posición basándonos en el progreso actual

Esta variable ‘Time.deltaTime’ es proporcionada por Unity en cada frame y representa el tiempo que ha transcurrido desde el anterior frame en segundos. Se utiliza normalmente en Unity para hacer los juegos estables y que determinadas acciones duren lo mismo independientemente de la plataforma en la que se esté ejecutando el juego. Puede que en un ordenador muy potente la duración entre cada frame sea menor que en un ordenador con capacidades menores, pero en ambos el resultado será el mismo.



Imagen 25 - Tipos de enemigos

Como ya se habrá podido deducir de la función de comportamiento anterior, los enemigos tienen dos atributos que determinan su estado actual: vida y velocidad. Estos además tienen copias: vida inicial y velocidad base. Estas copias son necesarias para controlar la barra de vida actual del enemigo y poder ser ralentizados. Se han creado tres tipos distintos de enemigos alternando estos valores.

Por un lado está el enemigo **básico** que tiene valores medios de vida y velocidad, el enemigo **rápido** que tiene una velocidad alta y vida baja y, todo lo contrario, el enemigo **pesado**, que tiene vida alta y velocidad baja.

Los enemigos tienen además un pequeño colisionador con forma de esfera 3D situado en su centro, esto es lo que se usará cuando queramos obtener los enemigos que hay en el alcance de una torreta. Además, todos los objetos que componen a los enemigos están en una capa de Objetos llamada ‘Enemy’, esto hará que la búsqueda de ellos sea mucho más rápida y eficiente.

Disponen de una función pública que les permite recibir daño, una cantidad determinada que se establece en la llamada del método. Una vez se recibe daño, se llama inmediatamente a la actualización de la barra de vida. Esta se muestra con un Sprite con forma de aro y una opción de Unity que permite a las imágenes ser rellenadas de distintas formas, en este caso de forma radial. Este transformador usa un valor de tipo float entre 0 y 1. Y es ahí donde entra en juego la vida inicial del enemigo para calcular el porcentaje de vida restante del enemigo.

## Torretas

Son los elementos del juego encargados de ayudar al jugador, atacando o ralentizando a los distintos enemigos que tengan a su alcance. Se ha creado una clase base 'Tower' que contiene todas las variables y funcionalidades que son comunes entre los distintos tipos de torretas. Algunos de estos atributos se han acabado usando en algunas torretas para cosas que no representan su nombre, por ejemplo, en la torreta ralentizadora se usa el daño de la torreta como factor de ralentización.

Entre las variables comunes están el rango de la torreta, el daño, la cadencia de disparo, el coste de comprar esa torreta, el coste de mejorarla y el dinero por venderla. Entre los métodos, se encuentra la inicialización, obtener un objetivo, seguir a un objetivo, mejorar la torreta y venderla.

Para el sistema de mejoras se ha creado el componente 'Upgrade', en él se recoge toda la información necesaria para las mejoras de las torretas.

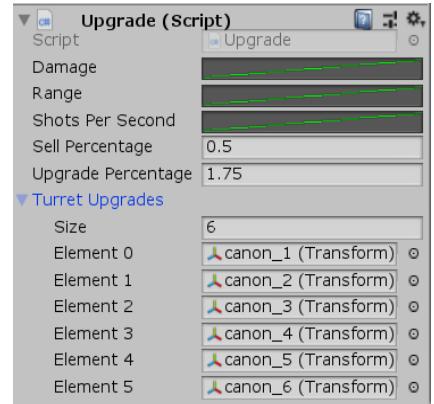


Imagen 26 - Ejemplo valores de mejora

La mayoría de los atributos relacionados con el funcionamiento de las torretas se ha recogido en forma de curva. De esta forma es bastante más fácil modelar la progresión de los atributos de las mejoras de torreta, con una curva que va de 0 a 1 en el eje horizontal y que verticalmente puede tomar los valores que nosotros queramos añadiendo varios puntos intermedios si lo deseamos y modificando la curvatura de cada vértice. Estas curvas se utilizan a nivel de código evaluándolas con un valor de 0 a 1 usado como coordenada 'x' y obteniendo el resultado, este valor se determina a partir del nivel de mejora actual de la torreta.

Además, se ha tratado el dinero devuelto al vender y el coste de mejorar con variables de tipo float, por ejemplo, la mayoría de las torretas devuelven el 50% del coste actual de las mismas.

Por último, se ha creado un array de tamaño configurable donde almacenar las referencias a las mejoras de los cañones de las torretas, que estéticamente serán los únicos que cambien.

En caso de que algún tipo de torreta concreta necesite modificar algún otro atributo cuando se le aplica una mejora, será necesario añadir un atributo de tipo curva u otro para ello en ese tipo de torreta.

Existen 5 tipos distintos de torretas, cada una con su funcionalidad y comportamiento diferenciado, cada una dispone de 5 mejoras posibles que aumentan sus capacidades. A continuación, se explica el funcionamiento de cada una.

### Torreta básica

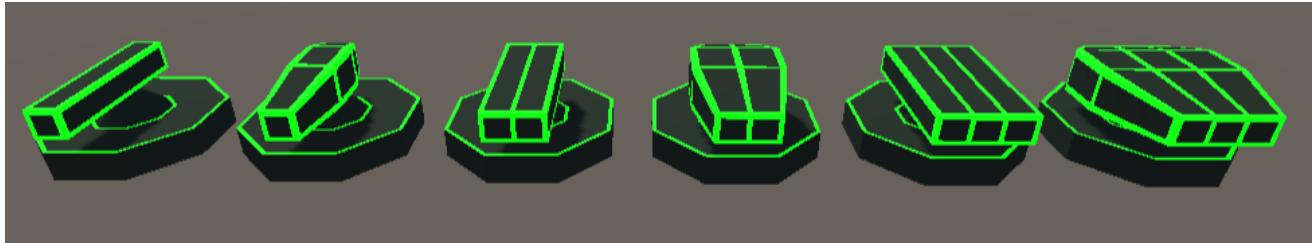


Imagen 27 - Torreta básica y mejoras

Dispara proyectiles a los enemigos, estos hacen un daño determinado al impactar y muestran un pequeño efecto de partículas sobre el objetivo. A medida que se van mejorando, aumenta el número de cañones. Se ha hecho que cada uno de ellos sea la salida de un proyectil y que se vaya alternando entre los disponibles actualmente. Es el método de mejorar el encargado de actualizar la posición de estos puntos, que han sido colocados manualmente desde el editor como objetos vacíos en cada una de las puntas de los cañones. Una vez encuentran un enemigo dentro de su rango, lo fijan como su objetivo actual y seguirán disparándole siempre que este siga en el alcance y no haya muerto.

Los proyectiles que usan son balas, esferas pequeñas sin colisionadores. Se simula el disparo, pero no se hace ningún cálculo de lanzamiento ni se usan físicas, de hecho, la bala es perseguidora, no va a fallar nunca. La velocidad a la que se mueve la bala hace que no se aprecie que es perseguidora.

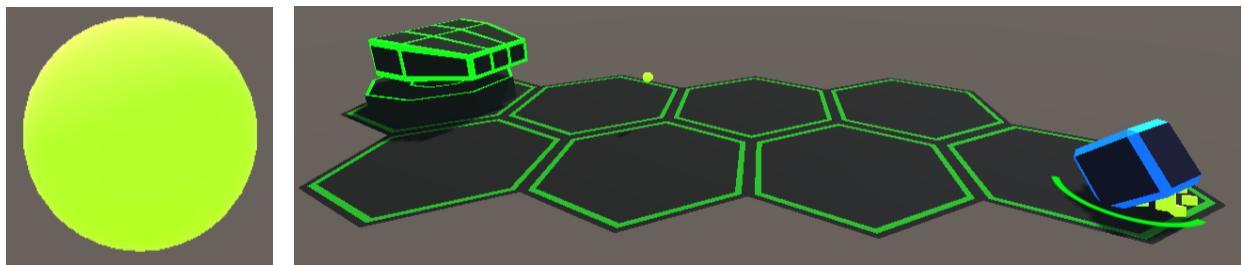
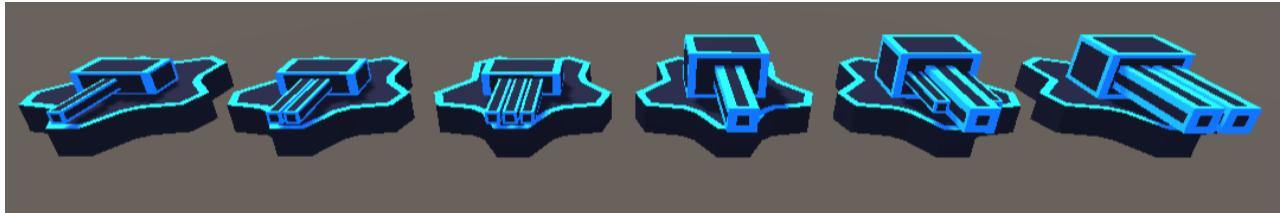


Imagen 28 - Proyectil bala y ejemplo de disparo con partículas de impacto

Al mejorar la torreta básica, se está aumentando el daño del proyectil, el alcance de la torreta y la cadencia de disparo, además de las posiciones por las que salen las balas.

## Torreta láser



*Imagen 29 - Torreta láser y mejoras*

Disparan un rayo ‘perforante’ que hará daño durante un breve periodo de tiempo a todos los enemigos que atravesese. Hacen un daño individual menor al de la torreta básica, pero si se colocan bien hacen mucho más daño, aprovechando que pueden dañar a varios enemigos a la vez.

Están continuamente apuntando cuando hay objetivos a su alcance, incluso si no está disparando. Una vez comienzan a disparar, permanecen un breve periodo de tiempo activas, su objetivo puede ir variando mientras están activas y si se queda sin enemigos a los que disparar dentro de su alcance, seguirá manteniendo el láser activo hasta que termine el tiempo que este debe permanecer activo. De esta forma es posible dañar a enemigos que ya no estén al alcance.

Para determinar qué enemigos son afectados por el láser, se ha usado la misma mecánica que para detectar a los enemigos en el alcance de las torretas, se busca a nivel de código los enemigos que colisionan con un cilindro de unas determinadas características.

En el caso de los enemigos en el rango de una torreta, se usa un cilindro de radio el rango de la torreta, situado en el centro de esta y de altura 3 (desde el 0 hasta el 3). De esta forma, aunque el juego tenga estética 3D realmente se simula un funcionamiento 2D. Para que fuera efectivamente un funcionamiento 3D habría que usar una esfera.

Para el láser es igual, sólo que el cilindro es de radio fijo: 0.25f y el cilindro comienza en el punto inicial del láser y termina 30 unidades más adelante en la dirección en la que esté apuntando el cañón.

Para el renderizado del láser se ha utilizado un componente de tipo ‘Line Renderer’ proporcionado por Unity. Se planteó la posibilidad de que a medida que se mejorase la torreta, esta disparara más láseres, como sugieren los cañones de mejora. Sin embargo, el componente no permite renderizar líneas discontinuas, de esa forma podríamos haber renderizado cada láser por su cuenta.

Tras hacer alguna prueba viendo si sería posible conseguir renderizar varios láseres con un solo componente, se vio que el resultado no era muy convincente. Por tanto, las opciones eran dejarlo con un solo láser a pesar de que los cañones sugieran que deben salir más láseres, o añadir a cada torreta de este

tipo varios componentes de tipo ‘Line Renderer’ y dibujar cada láser con un componente distinto. Finalmente, se decidió optar por la primera opción ya que es más simple y no merecía la pena la otra opción.

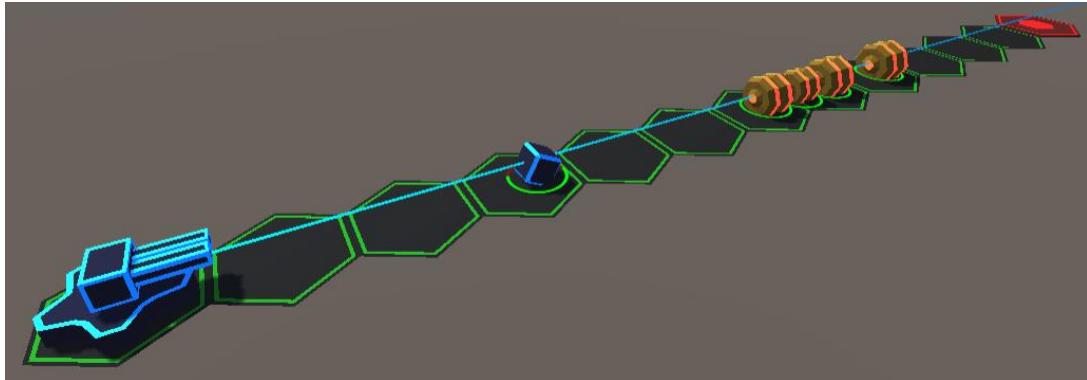


Imagen 30 - Ejemplo de disparo torreta láser

### Torreta misiles

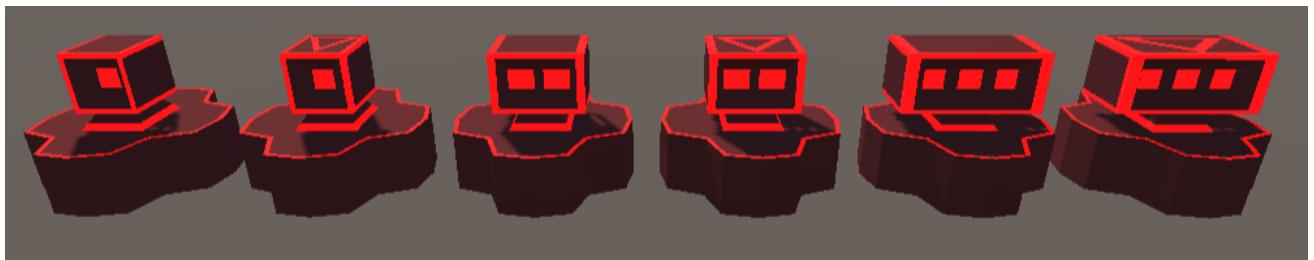


Imagen 31 - Torreta misiles y mejoras

Dispara misiles perseguidores que explotan al alcanzar a su objetivo dañando una cantidad determinada a todos los enemigos que haya en el área. Son perfectas para eliminar conjuntos concentrados de enemigos. No tienen ningún interés en fijar un objetivo de forma continua, una vez pueden disparar, apuntarán al primer enemigo que puedan y soltarán un proyectil de tipo misil que se encargará del resto.

Para las mejoras se ha usado el mismo sistema de puntos de aparición de proyectiles que para la torreta básica. Además, se ha añadido un atributo de tipo curva que es el radio de la explosión de los misiles.

Los proyectiles tienen una velocidad bastante menor que las balas y van dejando a su paso un rastro simulado con pequeñas explosiones que no hacen daño.

El funcionamiento de los misiles en su función ‘GameUpdate’ es comprobar si el enemigo que persiguen sigue vivo, en caso contrario explotar con una explosión menor que no hace daño. Si el enemigo sigue vivo, estos apuntan directamente hacia la posición del mismo. Para esto se ha utilizado la función proporcionada por Unity: ‘transform.LookAt(Vector3 worldPosition)’, transform en minúscula siempre hace referencia al componente transform del objeto en el que está el script. Este componente está en todos los objetos presentes en una escena de Unity, almacena información sobre la posición, rotación y escala del objeto. La función anterior se encargará de que el objeto tenga su vector azul ‘transform.forward’ en la dirección de la coordenada que le hemos pasado como parámetro.

Mientras se desplaza como comentamos antes, va creando pequeñas explosiones a su paso, estas tienen un radio de 0.1 y no hacen daño. Las explosiones son un objeto a parte que tiene métodos de inicialización y actualización. El efecto del tamaño creciente y la opacidad cambiante se crea mediante curvas que determinan la escala inicial y final y la opacidad inicial y final. Están activas durante un tiempo determinado y durante este tiempo se van actualizando. Sin embargo, el daño que realizan a los enemigos se aplica una única vez al iniciarse la explosión.

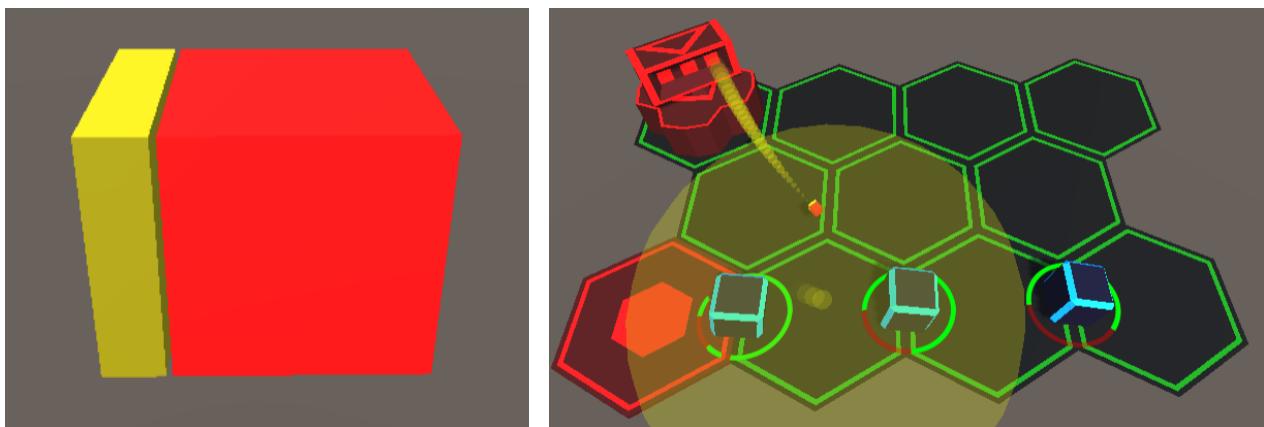


Imagen 32 - Proyectil misil y ejemplo de disparo con explosión

### Torreña daño en área



*Imagen 33 – Torreta daño en área y mejoras*

Realizan daño a todos los enemigos que se encuentren a su alcance a la vez. El daño que realizan es mucho menor que el de otras torretas a nivel individual, a nivel de conjunto hacen un daño más considerable que el resto de torretas si se colocan en sitios concretos, por ejemplo en una esquina que los enemigos tengan que bordear obligatoriamente estando dentro del área de la torreta durante más tiempo.

A nivel visual, se ha hecho que siempre que haya enemigos a su alcance, la cabeza de la torreta esté girando. Esto se ha conseguido con un componente sencillo ‘Rotator’ al que se le pasa el eje de rotación (0,1,0) en este caso, y los ángulos por segundo de la rotación. Por dentro se llama al método ‘transform.Rotate’ de Unity, que recibe un eje y un ángulo como parámetros y se encarga de realizar la rotación sobre ese eje.

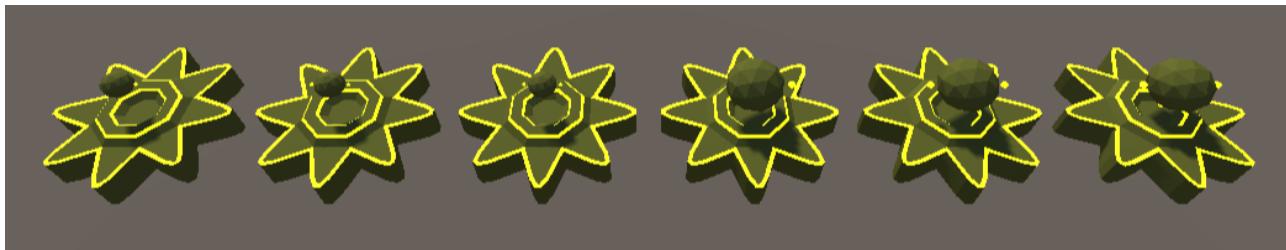
Para dar la sensación de que esta torreta hace daño de forma casi continua, se le ha establecido valores de cadencia de disparo más elevados, y los valores de daño son bajos teniendo esto en cuenta.

La detección de enemigos tiene un sistema similar al del daño de la torreta láser, se ha limitado el número de enemigos a los que pueden dañar a la vez por temas de eficiencia. Incluso así, es poco probable que una misma torreta tenga a 20 o más enemigos a su alcance a la vez.



*Imagen 34 - Torreña de daño en área en acción con alcance visible*

### Torreta ralentizadora en área



*Imagen 35 - Torreta ralentizadora y mejoras*

Tienen un funcionamiento prácticamente similar al de las torretas de daño en área, pero en lugar de hacer daño a los enemigos, los ralentizan. Aunque no se aprecie muy bien en la imagen, las cabezas de la torreta están compuestas por una esfera grande levitante y pequeñas esferas a su alrededor de color amarillo.

A nivel visual, al igual que con la torreta anterior, se ha hecho que cuando tengan enemigos a su alcance, las esferas pequeñas giren alrededor de la esfera grande. Esto se ha conseguido con otro Script ‘RotateAroundParent’ que es similar al anterior ‘Rotate’ pero este usa la función ‘transform.RotateAround’ que recibe una posición, un eje y un ángulo. La posición que le pasamos es la del padre que siempre es la esfera grande.

Al igual que con la anterior torreta, se le ha limitado el número de enemigos a los que puede ralentizar a la vez, estableciendo el máximo a 20 enemigos.

Cuando estuvimos hablando de los enemigos no entramos a discutir la forma en la que podían ser ralentizados. Además del método para recibir daño, los enemigos disponen de un método público ‘Slow’ que recibe un parámetro de 0 a 1, este parámetro representa el porcentaje de la velocidad base del enemigo que se le reducirá su velocidad.

Al principio se implementó este método haciendo que varias torretas de ralentización pudieran ralentizar al mismo enemigo a la vez, estableciendo una velocidad mínima para que fuera imposible parar a los enemigos totalmente. Se hicieron pruebas con esta funcionalidad y se veía demasiado poderosas a estas torretas. Por tanto, se decidió limitar el número de torretas que pueden ralentizar al mismo enemigo a una única torreta. La torreta con mayor porcentaje de ralentización será la que afecte al enemigo.

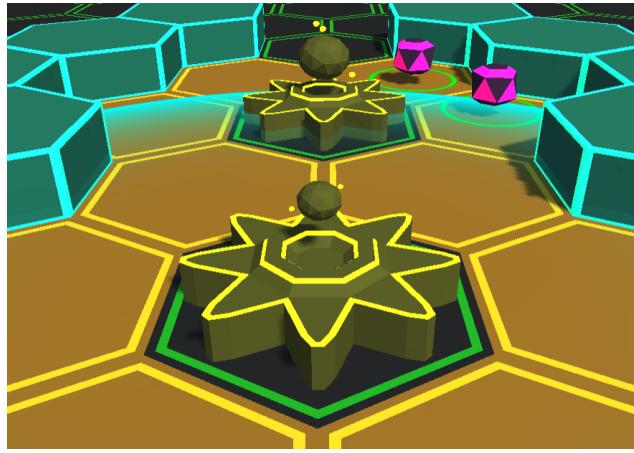


Imagen 36 - Torreta ralentizadora en acción con alcance visible

## Oleadas

El sistema de oleadas está coordinado por el componente ‘WaveSpawner’, este funciona como una máquina de estados, es consciente del estado en que se encuentra y actúa conforme a este. Tiene tres estados posibles:

- Spawning: está actualmente generando enemigos. (explicado al final de esta sección)
- Waiting: está esperando a que todos los enemigos de la oleada actual sean eliminados o lleguen al destino.
- Counting: está realizando la cuenta atrás para la siguiente oleada.

Se ha intentado hacer el sistema de oleadas suficientemente complejo como para generar distintos conjuntos de enemigos en una misma oleada. Toda la información sobre cada oleada está recogida en el componente ‘Wave’, que a su vez está compuesto de ‘waveSlices’ o fragmentos de oleada. Estos fragmentos son simplemente variables de tipo Vector3Int, en el que las coordenadas (x, y, z) se han utilizado como (punto de aparición, tipo de enemigo, cantidad). Los dos primeros parámetros deben tener un valor válido entre el número de puntos de aparición y el número de enemigos existentes en el juego. En caso de que tengan un valor no válido, se seleccionará un punto de aparición o enemigo aleatorio entre los disponibles.

Finalmente, se usan otros parámetros de control, como tiempo entre fragmentos y tiempo entre enemigos, para controlar la velocidad y el flujo con el que se generan los enemigos. Este componente está

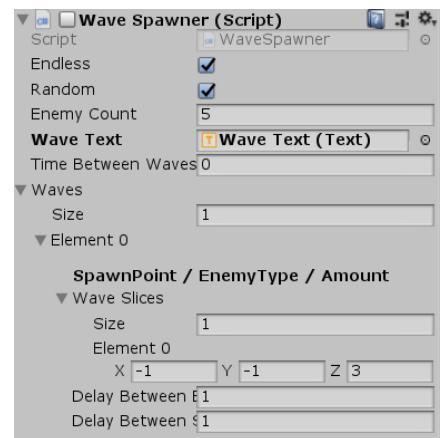


Imagen 37 - Ejemplo de valores del sistema de oleadas

ligado al mismo objeto que el componente ‘Game’ del juego y será el que le indique cuando funcionar o no. Se añadió además una funcionalidad ‘modo sin fin’, en esta se puede elegir si se desea que las oleadas existentes empiecen de 0 cada vez que se alcance la última oleada, o si en su lugar se deben generar oleadas aleatorias cada vez más grandes.

Las oleadas aleatorias se generan desde el componente ‘Wave’, llamando al método estático ‘RandomWave’ que recibe el número de enemigos que se desea que tenga la oleada aleatoria. Estas oleadas se generan creando conjuntos fragmentos de oleada de enemigos de distinto tipo. Además del sistema de oleadas, el componente ‘Game’ se encarga de hacer que el progreso de las oleadas sea cada vez más duro. Esto se hace aumentando poco a poco la vida con la que aparecen los enemigos de las oleadas, de nuevo, se ha usado una curva para controlar esta progresión de la vida de los enemigos a medida que avanzan las oleadas. Al iniciar una partida se ha establecido un valor de 0.5, significando esto que los enemigos de las primeras oleadas tienen un 50% de la vida original. Tras 100 oleadas, se ha establecido un máximo de 3000% la vida original de los enemigos, es poco probable que los jugadores lleguen tan lejos, y si lo hacen deben tener una disposición del tablero muy buena y óptima.

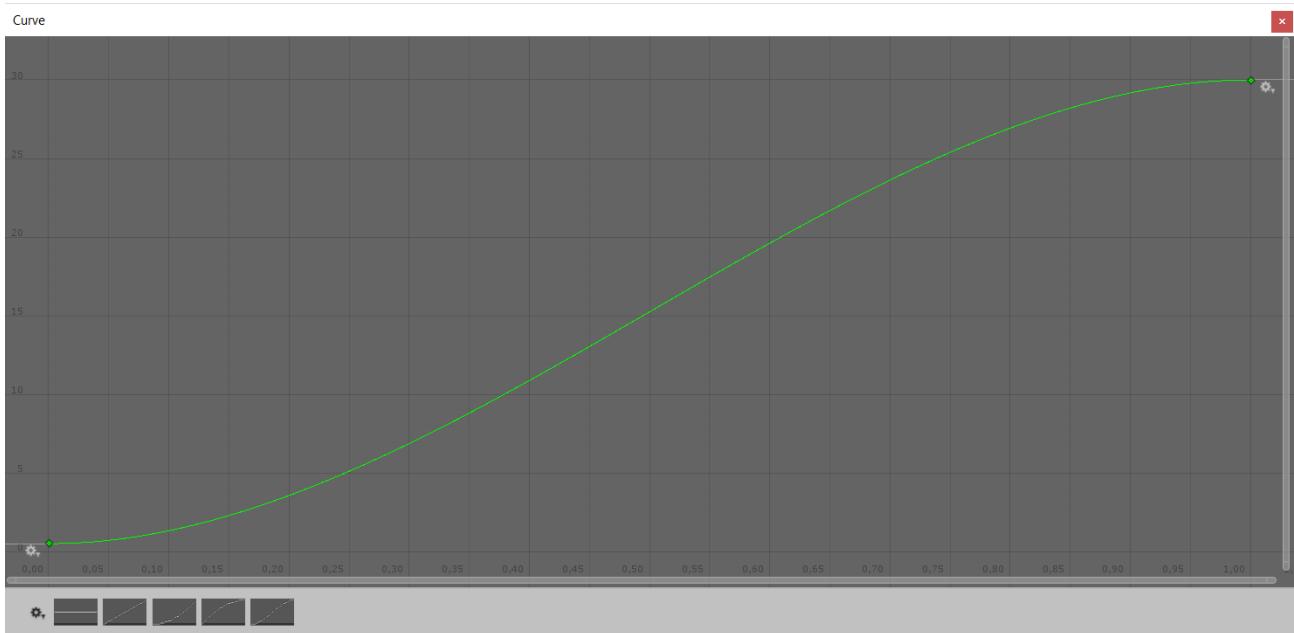


Imagen 38 - Ejemplo de valores de la curva de vida de los enemigos

La lógica interna del componente WaveSpawner hace uso de las llamadas corutinas, estas son funciones que pueden suspender su ejecución con la palabra reservada ‘yield’ hasta que la instrucción que la acompañe termine. Son muy cómodas para funciones que no se deben ejecutar en un solo frame del juego o que hacen uso de pausas temporales por motivos de control. Esta corutina es usada en el método `SpawnWave(Wave wave)`, que mostramos a continuación. Finalmente se termina la ejecución usando ‘yield break’.

```
IEnumerator SpawnWave(Wave wave)
{
    state = SpawnState.SPAWNING;
    for (int sliceIndex = 0; sliceIndex < wave.waveSlices.Length; sliceIndex++)
    {
        Vector3Int waveSlice = wave.waveSlices[sliceIndex];
        int spawnIdx = waveSlice.x;
        int enemyIdx = waveSlice.y;

        for (int i = 0; i < waveSlice.z; i++)
        {
            Game.SpawnEnemy(spawnIdx, enemyIdx);
            if (i < waveSlice.z - 1)
                yield return new WaitForSeconds(wave.delayBetweenEnemies);
        }

        if (sliceIndex < wave.waveSlices.Length - 1)
            yield return new WaitForSeconds(wave.delayBetweenSlices);
    }

    state = SpawnState.WAITING;
    yield break;
}
```

Imagen 39 - Ejemplo de método de tipo coroutine

## Interfaz del juego



Imagen 40 - Explicación interfaz del juego

La interfaz durante el juego se ha desarrollado para ser simple y se ha intentado que sea intuitiva. Se han cogido ideas de juegos existentes, por ejemplo, el panel de información de arriba a la izquierda es prácticamente idéntico al de 'Kingdom Rush'. El resto son simplemente botones que utilizan el componente de Unity 'Button'.

Este componente incluye información sobre la apariencia, cómo cambiará cuando se pase por encima de él, cuando se le pulse, cuando esté inactivo, seleccionado, etc. También se pueden utilizar animaciones en lugar de transiciones de color, en este proyecto no se ha utilizado esta funcionalidad. Además de la apariencia, los botones permiten editar su función de forma cómoda dentro del editor de Unity.

En la imagen vemos la sección 'On Click ()', donde podemos arrastrar objetos de la escena y seleccionar funciones públicas o estáticas de los componentes de estos. En el ejemplo, vemos los valores para el botón del menú de opciones, este al ser pulsado:

1. Pausa el juego si no estaba pausado ya.
2. Activa la interfaz del menú pausa.
3. Desactiva la interfaz del juego.

Configurar los botones en Unity es bastante sencillo y cómodo, en caso de que se quisiera hacer algo más complejo, se puede crear un método público en algún componente del juego y llamar a esta función desde el botón. Es el caso por ejemplo del paso 1, usa la función 'ToggleTime' que está definida en el componente Game.

Para crear una interfaz en Unity, se empieza añadiendo un objeto de tipo 'Canvas', que será el marco donde incluiremos el resto de los componentes. Al crear un objeto de este tipo por primera vez, Unity crea automáticamente otro objeto de llamado 'EventSystem', que será el encargado de controlar la interacción del usuario con la interfaz. Todos los elementos interactivos de la interfaz tienen un atributo "Raycast Target" donde se indica si se le debe considerar un objetivo para la interacción con el usuario. Los paneles de información tienen esta opción desactivada.

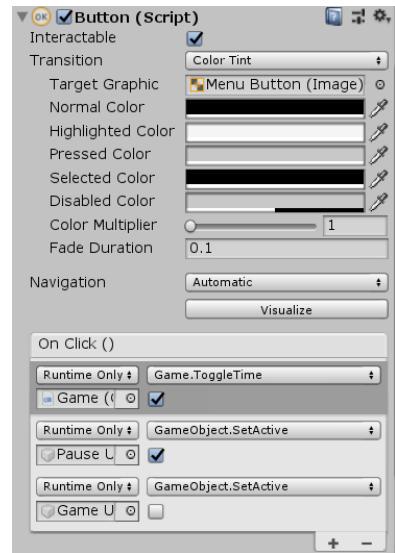


Imagen 41 - Ejemplo componente Button

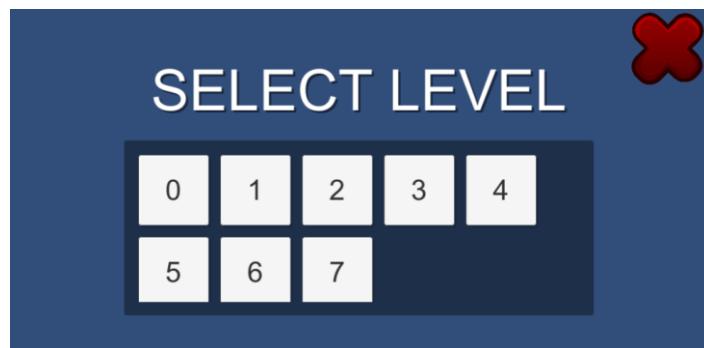


Imagen 42 - Interfaz del menú inicial / selección de niveles

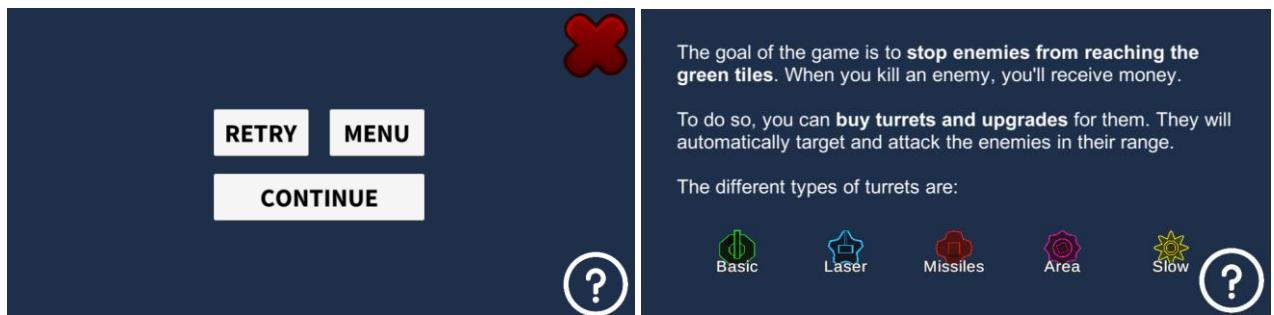


Imagen 43 - Menú de pausa y ventana de ayuda

Con la interfaz de usuario han surgido varios problemas durante el desarrollo. Por un lado, había que tener en cuenta los distintos tamaños de pantalla de los dispositivos Android, y por defecto, el 'Canvas' de Unity tiene un tamaño fijo y no adapta su contenido al tamaño de la pantalla. Para solucionar este problema, se ha modificado el componente 'Canvas Scaler' que trae por defecto el Canvas de Unity y se ha seleccionado la opción 'Scale with screen size' usando una resolución de referencia de 800 x 600 y un modo de igualación a la pantalla 'Match width or height' intentando que se adapte por igual a ambas medidas.

El otro problema que surgió tuvo que ver con la interacción con el tablero del juego, lo veremos más adelante.

### Selección de torreta

Este objeto aparece cuando el jugador selecciona una torreta existente en el tablero. Cuando esto pasa, el objeto Game le indica la torreta seleccionada y este se inicializa automáticamente en la posición de esta. Incluye accesos la función de mejorar y vender la torreta, que son botones iguales que los de la interfaz de juego, con la única diferencia de que estos en lugar de estar posicionados en la pantalla del dispositivo, están situados en el entorno 3D del juego.



Esto le proporciona un enfoque distinto a la hora de tener que interactuar con ellos. Con la activación de la cámara del móvil surgió la necesidad de que los botones se adaptaran a la posición del jugador con respecto al tablero. Para resolver esta necesidad, se les añadió un componente 'LookAtCamera' cuyo funcionamiento interno es similar al proporcionado por Unity 'transform.LookAt', pero en lugar de mirar directamente a la cámara se mira a su posición en el plano y = 0.

Imagen 44 - Interfaz de selección de torreta

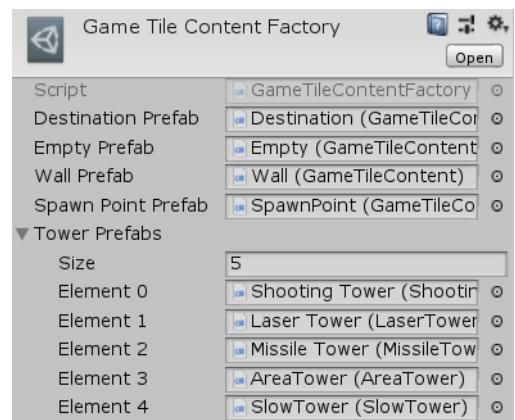
El botón de mejorar la torreta está visible siempre y cuando la torreta seleccionada no esté ya en su máximo nivel de mejora, y el botón se pondrá ‘inactivo’ cuando el jugador no disponga de las monedas necesarias para realizar la compra de la mejora.

Además de los botones, este componente refleja el alcance actual de la torreta seleccionada, que se actualiza al seleccionar una torreta o mejorar la que esté seleccionada. Esto se consigue modificando la escala del objeto que contiene la imagen circular, su tamaño por defecto es 1f, por tanto usando el rango de la torreta como escala este círculo debería representar con exactitud el rango de la torreta.

## Factorías

La idea original tras el uso de las factorías no era solamente el de mantener la creación e inicialización de los contenidos y proyectiles en lugares ordenados y de manera óptima. El propósito principal del uso de estas era el de crear un recolector de “basura” manual, que permitiera la recuperación de objetos que temporalmente no son necesarios y que más adelante sí volverían a serlo, pero en otro lugar. Por poner un ejemplo, con las balas disparadas por las torretas básicas el sistema habría consistido en sustituir el método ‘Destroy(gameObject)’ de Unity, por el reciclamiento de la bala, almacenándola en una lista auxiliar de balas inactivas. Cada vez que fuera necesaria una nueva bala, en lugar de instanciar un nuevo objeto de tipo bala, en su lugar comprobaríamos si existe una bala inactiva en la partida actualmente, en caso afirmativo, la retiraríamos de la lista auxiliar, la activamos y la inicializamos en el lugar de aparición y con las nuevas condiciones, daño, velocidad, etc.

Esta mecánica y otras más de manipulación de objetos con Unity, guardado de datos, carga de datos, gestión de diversidad, etc. viene muy bien recogida y explicada en los tutoriales de ‘Catlike Coding’ en su sección de tutoriales de Unity ‘Object Management’ [9], que se siguieron al inicio del proyecto hasta la mitad, y finalmente se vio que se escapaba del alcance del proyecto.



Finalmente, las factorías del juego han quedado a medio camino de lo que podrían haber sido, su única funcionalidad actual es la de instanciar objetos nuevos cuando sean necesarios, y borrar los ya existentes cuando ya no sean necesarios.

Imagen 45 - Valores de la factoría de contenido de casillas

## Game

Como ya comentamos antes, es el encargado de orquestar el flujo de las partidas, controlar los gestos de entrada del jugador, llamar a la actualización de los componentes activos en la escena cuando sea necesario y en el orden adecuado, creación de nuevos mapas desde el editor, y gestión de todos los métodos de los botones en la interfaz de usuario con comportamientos más complejos y personalizados.

Comenzaremos hablando de la entrada táctil, el componente Game tiene presente en todo momento el tipo de torreta seleccionada en la interfaz (comienza siendo la torreta básica al iniciar una partida) y también tiene presente si actualmente hay una torreta seleccionada del tablero.

Al comenzar con el desarrollo, se utilizaba siempre la función de Unity “`Input.GetMouseButtonDown(int button)`” con el valor 0, que devolvía verdadero en el frame en el que el usuario hacía clic izquierdo o pulsaba la pantalla, si se pulsaba, se creaba un rayo usando la función de Unity “`Camera.main.ScreenPointToRay(Input.mousePosition)`” el cual puede ser usado para detectar una colisión con objetos que contengan componentes de tipo ‘`Collider`’.

Más adelante se comenzó a implementar la funcionalidad de poder mover la cámara y hacer zoom con controles táctiles, por tanto, se tuvo que adaptar lo anterior a las nuevas necesidades. Para ello, en el componente Game se creó una especie de máquina de estados que comprueba si el toque táctil del jugador es individual (un solo dedo) y sin desplazamiento. Esto se hizo gracias a las herramientas que proporciona Unity para el control de entrada táctil, entre otras funcionalidades, proporciona el número de toques en la pantalla en el frame actual, y el estado de cada uno de estos:

- `TouchPhase.Began`: el dedo acaba de comenzar a tocar la pantalla.
- `TouchPhase.Moved`: el dedo se ha desplazado desde el último frame.
- `TouchPhase.Stationary`: el dedo no se ha movido.
- `TouchPhase.Canceled`: se ha cancelado el seguimiento del dedo, por ejemplo, por haber colocado la mano.
- `TouchPhase.Ended`: el dedo se ha levantado de la pantalla.

Teniendo en cuenta todo lo anterior, se creó un método para llevar el seguimiento de los dedos en la pantalla y determinar si ha habido una pulsación simple sin movimiento:

1. Si el usuario pulsa una primera vez la pantalla se comienza el seguimiento.
2. Mientras el número de dedos en la pantalla sea de 1, se sigue manteniendo el seguimiento.
3. Mientras el dedo no se haya movido en la pantalla o cancelado, se sigue manteniendo el seguimiento.

4. Si el usuario está levantando el dedo que tenía seguimiento y el seguimiento del dedo sigue activo, se determina el toque como un toque válido.

En caso afirmativo, se proyecta un rayo usando la función mencionada anteriormente ('ScreenPointToRay') y se detecta la colisión con cualquier 'Collider' del juego, el tablero del juego dispone de un 'collider' plano con forma cuadrada que lo cubre por completo, y además las torretas y paredes también disponen de sus propios 'collider'.

Usando la coordenada 3D de esa colisión se transforma a coordenadas hexagonales, y de estas se transforman a fila y columna del tablero de juego. Si la fila y columna resultantes encajan dentro del número de casillas definidos en el tablero, se obtiene el índice de la casilla seleccionada y se recupera la misma solicitándosela al objeto GameBoard.

Una vez recuperada la casilla que se ha seleccionado, se hace con ella lo que se deseé. En caso de que hubiera una torreta seleccionada, se desactiva la selección y se ignora el toque del jugador para colocar una torreta.

Con esta lógica de tocar el tablero de juego surgió el problema que comentábamos en la sección de la interfaz de usuario. El problema que tuvimos que resolver es que al tocar un botón de la interfaz que detrás tuviera una casilla del juego, este botón no bloquea esa interacción. Es decir, pulsamos el botón y también se coloca una torreta o se interactúa con la casilla que hay detrás de este, lo cual, es poco intuitivo y simplemente da sensación de mal diseño.

Tras investigar varias soluciones encontradas en foros de ayuda [10], se estuvo usando la solución 'EventSystem.current.IsPointerOverGameObject()' proporcionada por Unity, que detectaba si el ratón o el toque está actualmente sobre un objeto de la interfaz, si devolvía verdadero, ignorábamos el toque. Esta solución era perfectamente válida siempre y cuando no se quisiera detectar un dedo que se acababa de levantar (TouchPhase.Ended), a nivel interno, este método ignora esos toques.

Finalmente, se acabó usando una función que aparece en las respuestas del enlace anterior. Esta recoge la posición del mouse o el dedo independientemente de su estado, y proyecta un rayo en la interfaz



Imagen 46 - Funcionamiento de selección de casillas en una fase temprana del proyecto

del juego detectando si ha habido alguna colisión. Si la ha habido, se devuelve verdadero, y se ignoraría el clic/toque.

Tanto con el objeto Game como con el objeto GameBoard se ha adoptado el patrón ‘singleton’ que les hace ser instancias únicas y estáticas a lo largo de la partida. Es decir, dentro de cada script tienen una referencia estática a sí mismos, esta referencia se inicializa en el método ‘OnEnable()’ de Unity que se llama cada vez que se activa el gameObject que los contiene y al iniciarse el juego si están activos. Una vez establecida esta referencia, otros objetos pueden hacer llamadas a métodos estáticos de estos scripts sin necesidad de tener una referencia de ese componente.

Por ejemplo, desde la clase Enemy, cuando un enemigo alcanza un destino, llama al método estático ‘Game.ReceiveDamage()’, este a su vez utiliza los valores de la referencia estática de sí mismo, realizando la lógica que sea necesaria. En la imagen, ‘instance’ es la referencia estática de sí mismo.

```
public static void ReceiveDamage()
{
    instance.health -= 1;
    instance.healthText.text = instance.health.ToString();

    if (instance.health <= 0)
        instance.GameOver();
}
```

*Imagen 47 - Ejemplo de método estático haciendo uso de la instancia única del componente Game*

Este enfoque ha permitido realizar comunicación entre objetos de forma sencilla y sin perder eficiencia. El componente Game maneja también toda la lógica detrás de los botones de la interfaz del juego, se encarga de actualizar la información de las vidas, las monedas y las oleadas. También controla la velocidad del tiempo modificando la variable de Unity ‘Time.timeScale’, que por defecto tiene un valor de 1f, si se aumenta, la velocidad del juego aumenta, para detener el juego, se establece a 0f. Incluso cuando el tiempo está parado, la interfaz del juego sigue funcionando sin tener esto en cuenta.

Para la creación de niveles se decidió separar la lógica del juego con la de edición de niveles usando un atributo booleano que permite en el editor de Unity, modificar el tablero del juego y posteriormente imprimir por la consola una cadena de texto con la representación del estado actual del mismo. En este modo edición gestionado por los objetos Game y GameBoard, se permite modificar el tablero de juego añadiendo y quitando los 4 tipos de contenidos para las mismas: casilla bloqueante, punto de aparición, punto de destino y pared.

Durante la edición del tablero se sigue comprobando la validez del mismo (usando las reglas explicadas en la sección tablero). Con esta funcionalidad disponible únicamente en el editor de Unity, se ha

podido prototipar y crear distintos niveles del juego de forma relativamente rápida. Una vez se imprime el mapa por la consola de Unity, almacenamos esa plantilla en el objeto Game, en un array de cadenas.

Desde el menú inicial del juego cada botón establece que plantilla desea utilizar usando la funcionalidad de Unity para establecer preferencias del jugador que son persistentes entre distintas escenas. Esto se hace con instrucciones como ‘PlayerPrefs.SetInt("key", value)’, con esto, establecemos la clave “templateIndex” al índice de la plantilla de mapa que deseamos jugar. Es el componente ‘Game’ el encargado de comprobar el índice establecido en esa variable antes de generar el tablero de juego.

### **Camera controller**

La cámara del juego dispone de un componente que se encarga de controlar el desplazamiento paralelo al plano y = 0 y el zoom de la cámara cuando esta no tiene la realidad aumentada activada dándole al jugador libertad para moverse a su antojo siempre dentro de unos límites establecidos. Esta funcionalidad se ha implementado utilizando las mismas herramientas que para detectar si el toque es simple y sin movimiento que se usaba en el componente ‘Game’.

Si hay un dedo pulsando la pantalla y este se mueve, aplicamos un movimiento en esa dirección a la cámara del juego.

Si hay dos dedos pulsando la pantalla y alguno de estos se mueve, calcular si se han acercado o alejado los dedos y aplicar un movimiento a la cámara según el resultado. Este componente al principio usaba los dos dedos para el movimiento y el zoom, pero hacía complejo determinar que estaba queriendo hacer el usuario. Además, el resultado era bastante torpe. Finalmente, por simplicidad y por hacerlo más cómodo, se decidió separarlo en dos gestos distintos.

### **Vuforia**

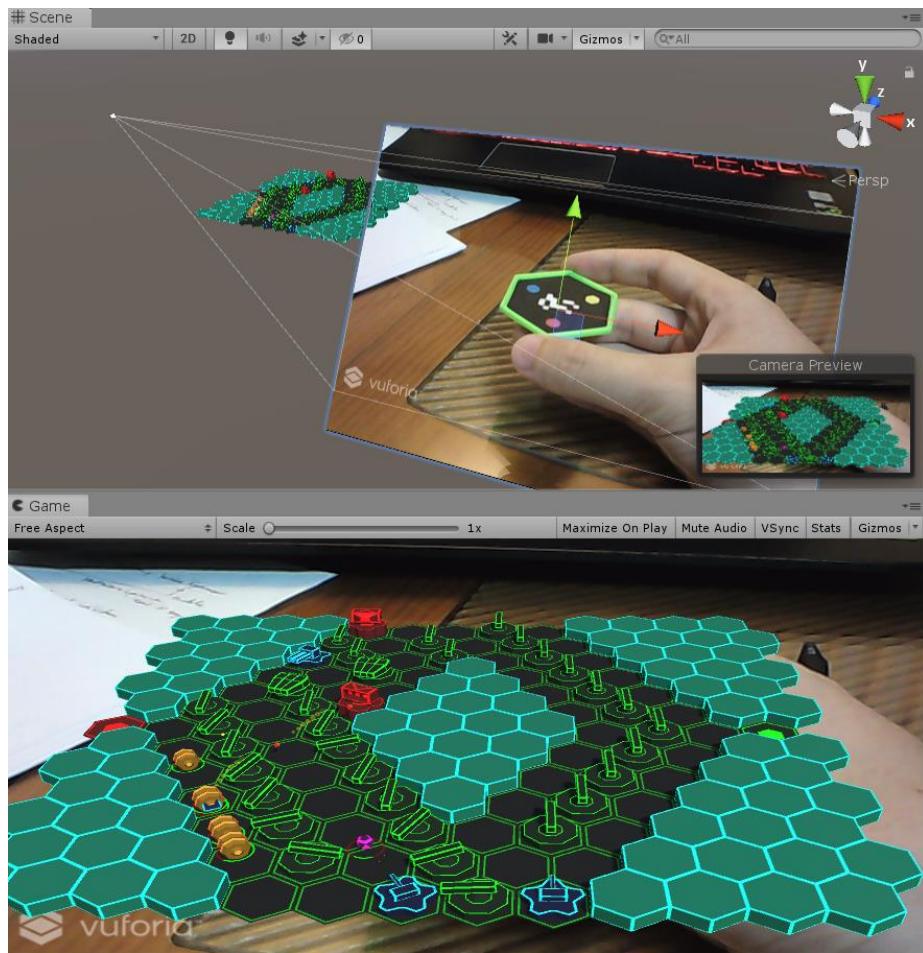
Es el encargado de hacer que funcione la realidad aumentada del juego. El proceso de montaje ha sido el siguiente:

1. Nos creamos una cuenta en ‘<https://developer.vuforia.com>’.
2. Creamos una clave de desarrollo gratuita en la pestaña License Manager.
3. Añadimos una base de datos de tipo Vumark en la pestaña Target Manager, seleccionando la licencia anterior.
4. Seleccionamos la base de datos y pulsamos en añadir objetivo. Seleccionamos el archivo ‘.svg’ que contiene los gráficos de nuestra imagen vumark. Añadimos la anchura en metros que deseamos

que tenga la imagen, en nuestro caso lo establecimos a 0.035 m ó 3.5 cm. Y finalmente le damos un nombre.

5. Desde Unity nos vamos a File/Build Settings, seleccionamos como plataforma Android y pulsamos en ‘Player Settings..’, al final de la sección ‘Player’, en la subsección ‘XR Settings’ marcamos la opción ‘Vuforia Augmented Reality Support’. Unity nos pedirá automáticamente importar los recursos necesarios en el proyecto para poder trabajar con Vuforia.
6. Eliminamos la cámara normal del juego y añadimos una de tipo Vuforia Engine/AR camera. En las opciones de esta, pulsamos sobre el botón ‘Open Vuforia Engine Configuration’.
7. En esta configuración debemos establecer la clave de licencia que generamos en la página web y en la sección de bases de datos seleccionar la base de datos creada anteriormente, esta se puede descargar desde la web e importarse en Unity como un paquete de contenido.
8. Por último, creamos un objeto de tipo Vuforia Engine/VuMark y le establecemos la base de datos y la imagen codificada a representar.
9. Si queremos que este objeto se use como centro del juego, debemos establecerlo en la cámara del juego, seleccionando como ‘World Center Mode’, ‘Specific target’ y arrastrando el objeto con la imagen codificada en el hueco que aparece justo debajo de esta opción.

Según el tamaño que se le establezca en la escena a la imagen codificada, el tablero saldrá más o menos grande en la realidad aumentada.



*Imagen 48 - Funcionamiento de Vuforia. Vista de escena y modo jugar de Unity*

Como se puede ver en la imagen, Vuforia se encarga durante el juego de crear un plano que usa como textura la imagen recibida por la cámara. Al usar este modo, de ‘centro del mundo’, la cámara adapta su posición y la posición del plano para que encaje con la de la imagen codificada.

El comportamiento de Vuforia es fácilmente activable y desactivable, basta con marcar o desmarcar el componente ‘Buforia Behaviour’ asociado a la cámara de realidad aumentada. En el juego al comenzar una partida la cámara comienza con este componente desactivado, y se almacena la posición, rotación y campo de visión inicial de esta. Cuando se activa el componente pulsando el botón de realidad aumentada, la cámara pasa a ser controlada totalmente por el jugador. Si se desactiva, el componente Game se encarga de establecerle a la cámara los valores iniciales de posición, rotación y campo de visión.

Para las imágenes codificadas se creó una plantilla siguiendo un tutorial proporcionado por Vuforia, haciendo uso de Adobe Illustrator [11]. Se decidió crear una de tipo entero, es decir, con los elementos de código, se puede codificar un número entero. En nuestro caso se pueden representar desde el 0 hasta el 999. Esto se hizo así con intención de añadir o usar más de una imagen en el juego, o quizás usar varias imágenes que sirvan de puntos de referencia para poder mover más libremente la cámara. Finalmente se

dejó en una sola imagen ya que las pruebas que se hicieron con varias imágenes a la vez eran muy inestables y el contenido central entre las mismas no paraba de temblar.

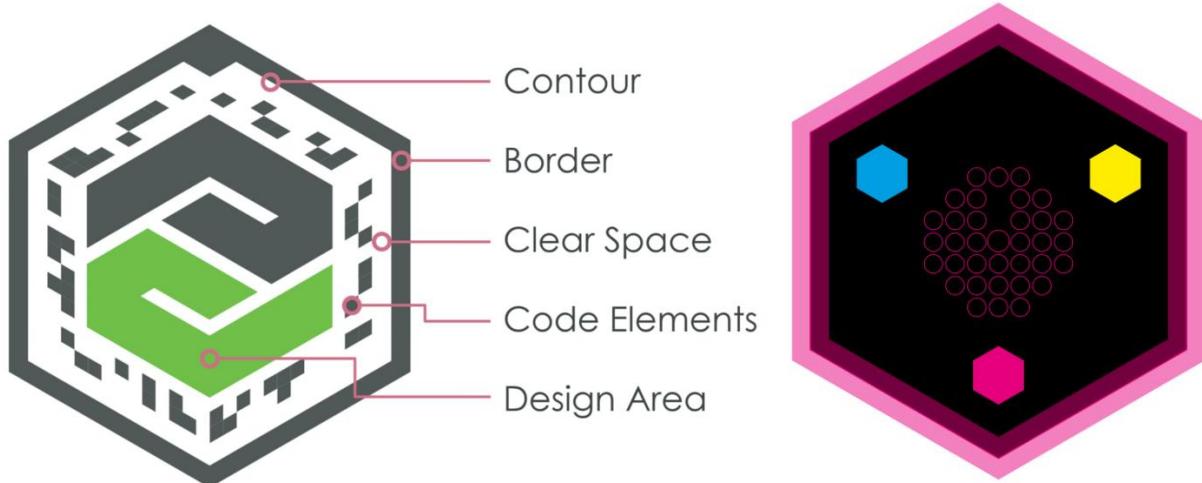
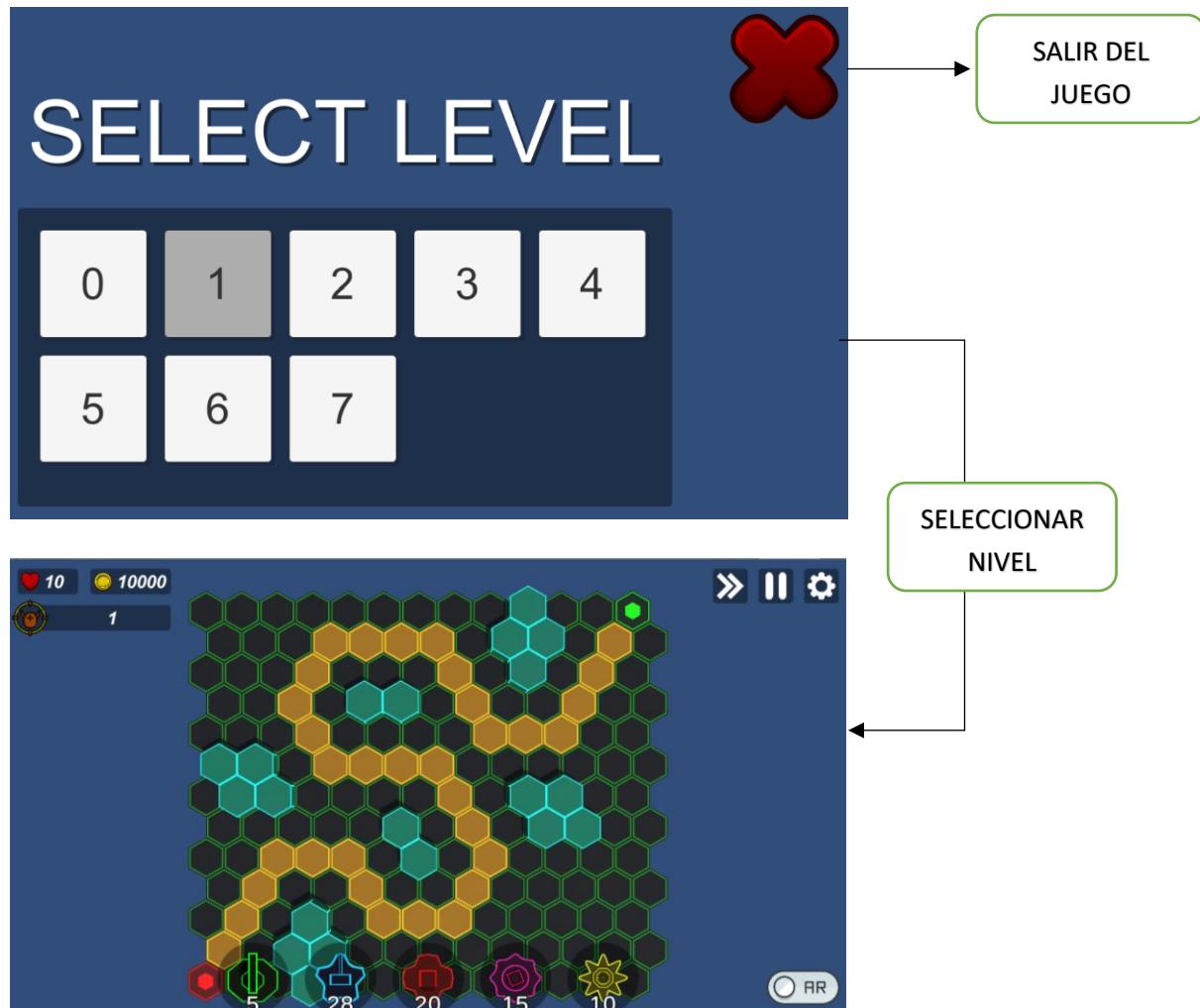


Imagen 49 - Ejemplo de VuMark y plantilla VuMark generada para el proyecto

Para poder probar el juego con la Realidad Aumentada se puede utilizar cualquiera de las imágenes codificadas independientemente del número que representen. Si hay varias imágenes presentes, se elegirá solo una de ellas y se ignora el resto. Al final del documento se adjuntarán algunas de ellas.

## Mapa de navegabilidad

Finalmente mostraremos el flujo de navegabilidad adoptado para el juego, explicando las opciones en cada pantalla.

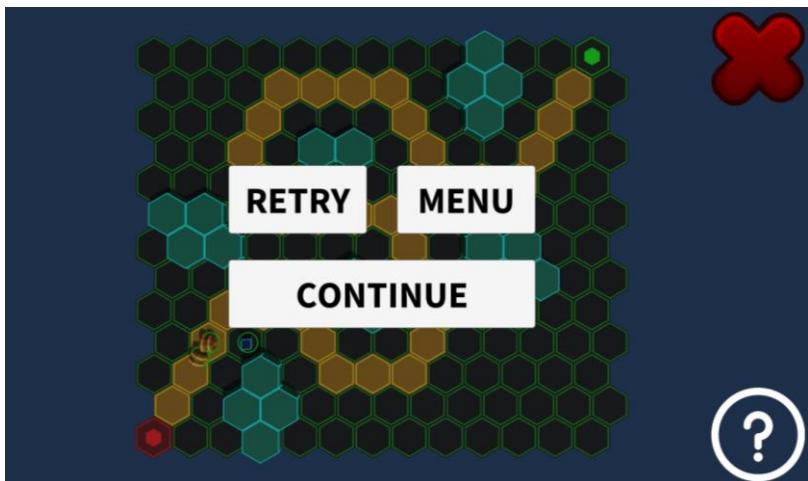




COMPRAR TORRETA  
Y SELECCIONARLA



ACTIVAR / DESACTIVAR  
REALIDAD AUMENTADA



ABRIR MENÚ OPCIONES  
/ CONTINUAR



ABRIR / CERRAR  
AYUDA

Se han mostrado únicamente las transiciones entre pantallas o escenas del juego, los botones que tienen cambios en la lógica del juego se explican en la primera imagen de la sección ‘Interfaz del juego’.

Aquí termina la sección de desarrollo, a continuación, se muestra una recopilación de casi todos los elementos generados o utilizados para el juego de forma ordenada.

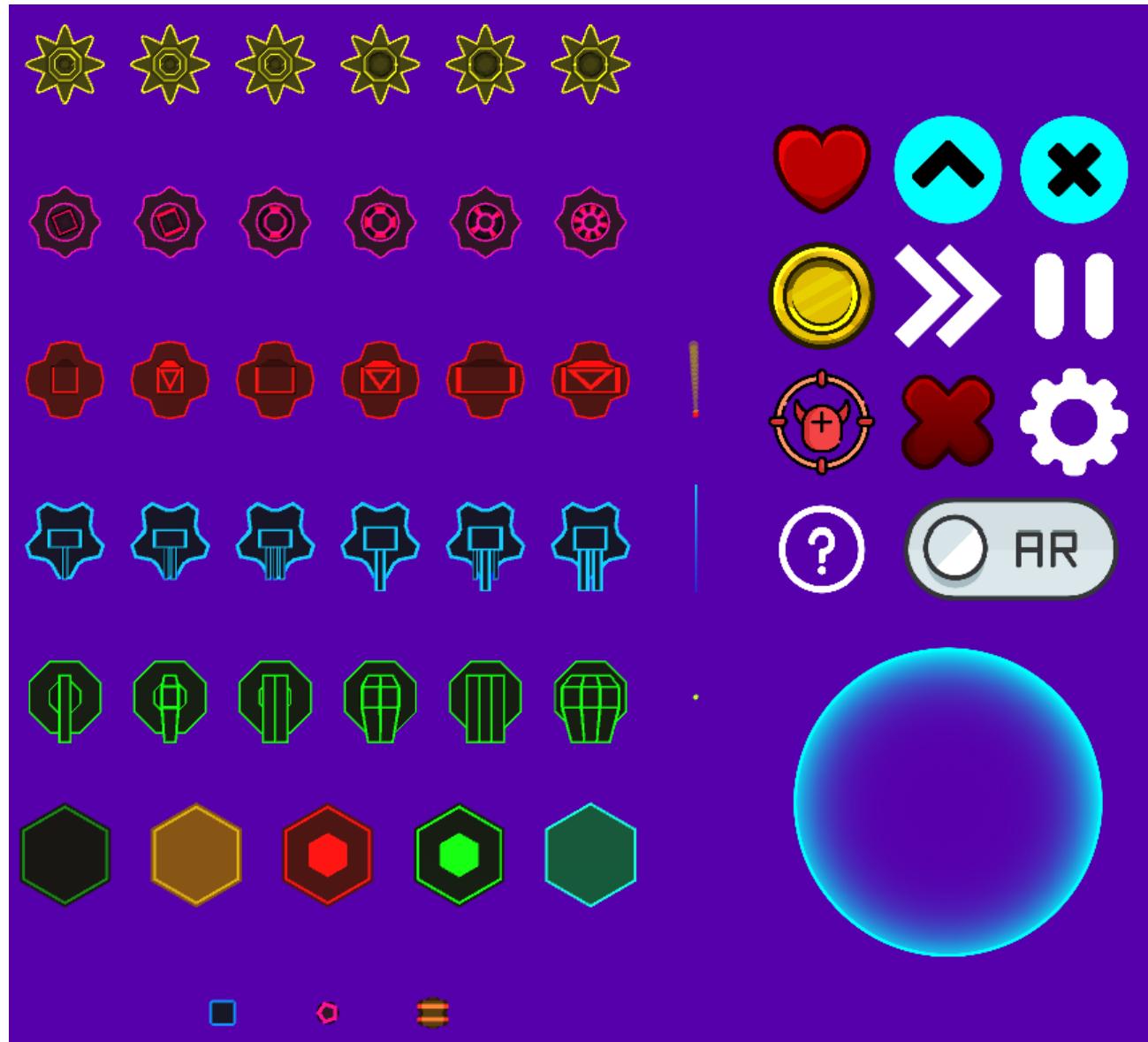


Imagen 50 - Recopilación de casi todo el contenido del juego

### 2.3. Pruebas

No se ha seguido ningún plan de pruebas concreto para probar el código producido para el videojuego. Sin embargo, se han utilizado herramientas para probar la aplicación de distintas formas, encontrar fallos de forma visual y depuración de estos:

- Modo 'Play' de Unity: sin ir más lejos, Unity permite probar el estado actual del juego de forma rápida y en tiempo real, consiguiendo el mismo resultado que tendría una 'build' de este.
- Modo 'Debug' de Visual Studio: pulsando sobre 'Asociar a Unity' hemos podido resolver bugs del juego que de otra manera no habríamos sido capaces de encontrar.
- Unity Remote 5: permite probar el juego directamente desde un dispositivo móvil, permitiendo controlar la entrada táctil sin necesidad de montar una 'build'. Para configurarlo se tuvo que hacer varios cambios tanto en Unity como en el móvil [12].
- Webcam externa: ha sido de gran utilidad para poder probar la funcionalidad de Vuforia apuntando con la cámara a las imágenes codificadas. Esta se puede seleccionar en la configuración de Vuforia.

Para pruebas de resolución, se han alterado las dimensiones de la pantalla de juego durante el mismo. Unity permite cambiar la relación de aspecto y la resolución de la pantalla en tiempo real. Se ha comprobado que la interfaz se adapta correctamente a los dispositivos móviles. Se decidió limitar la orientación de las pantallas a modo horizontal, no permitiendo jugar con el móvil vertical.

Incluso con las medidas usadas anteriormente, cómo más se han encontrado fallos ha sido construyendo el archivo de instalación .apk y probándolo en el móvil de forma aislada. Sobre todo, se encontraron fallos con respecto a la interfaz y el funcionamiento de esta.

Para las pruebas en móvil se han utilizado los siguientes dispositivos:

- Xiaomi Mi 8
- Xiaomi Redmi S2
- OnePlus 5T

### 3. Planificación del proyecto

#### 3.1. Planificación temporal inicial

El proyecto se ha realizado a lo largo de 3 meses, las etapas planificadas ordenadas cronológicamente son:

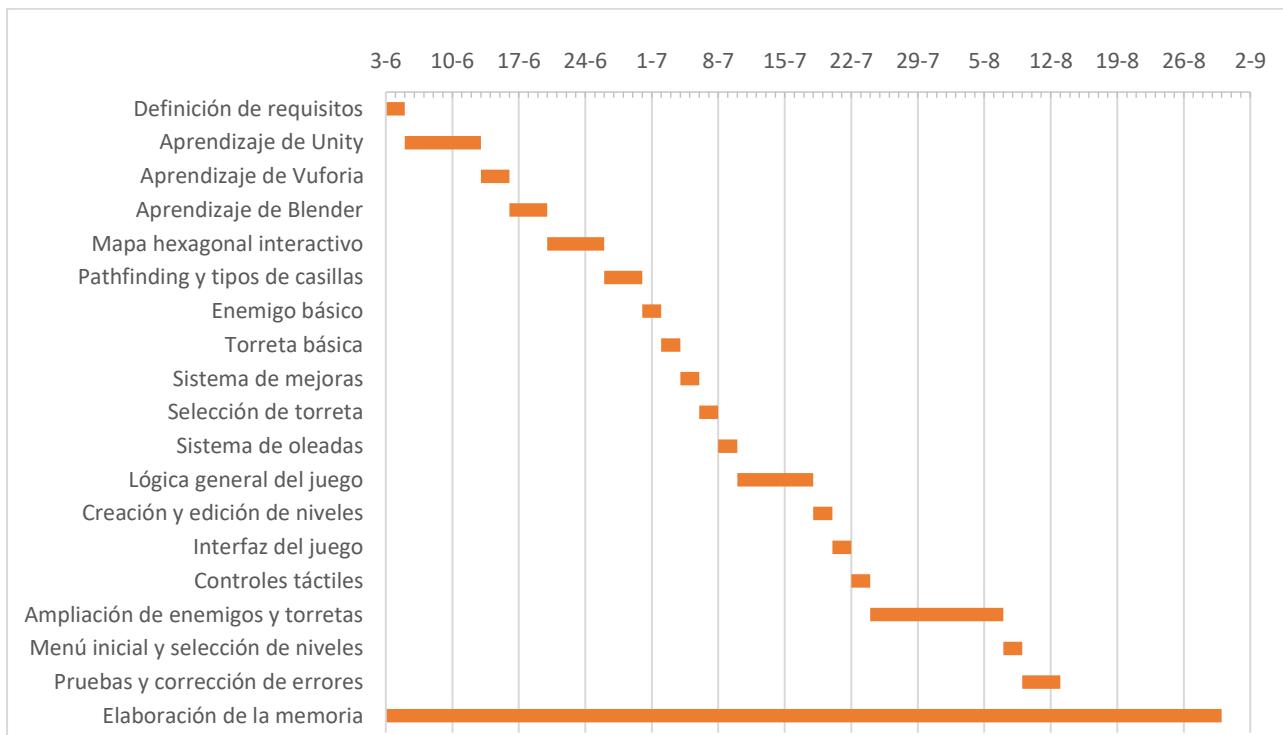
- Investigación y definición de requisitos del juego.
- Aprendizaje de tecnologías.
- Desarrollo del proyecto.
- Elaboración de la memoria.

El desglose de las tareas y la estimación temporal en horas de cada una:

*Tabla 1 - Estimación temporal inicial*

ID	Título	Tiempo estimado
1	Definición de requisitos	10
2	Aprendizaje de Unity	30
3	Aprendizaje de Vuforia	15
4	Aprendizaje de Blender	20
5	Mapa hexagonal interactivo	20
6	Pathfinding y tipos de casillas	15
7	Enemigo básico	5
8	Torreta básica	7
9	Sistema de mejoras	5
10	Selección de torreta	10
11	Sistema de oleadas	15
12	Lógica general del juego	30
13	Creación y edición de niveles	10
14	Interfaz del juego	10
15	Controles táctiles	10
16	Ampliación de enemigos y torretas	50
17	Menú inicial y selección de niveles	5
18	Pruebas y corrección de errores	20
19	Elaboración de la memoria	60
Total		337

Tabla 2 - Planificación temporal inicial - diagrama Gantt



Como aclaración, la tarea de elaboración de la memoria se ha representado abarcando la duración de todo el proyecto. Lo que vengo a decir con esto es que, aunque tenga una duración estimada de 60 horas, estas serán repartidas a lo largo del proyecto alternando con el resto de las tareas.

### 3.2. Planificación financiera inicial

#### Costes directos

Se ha considerado que la labor ejercida a lo largo del proyecto se puede equiparar a la de un programador junior debido a la escasa experiencia laboral previa en el sector de la industria del videojuego. El salario base según el BOE a partir del 01-01-2019 para un programador junior (Área 3 – Grupo E – Nivel 1) es de 14.817,89€ brutos al año.

Si a esto le calculamos las cotizaciones que tiene que pagar la empresa:

	COTIZACIONES SEGURIDAD SOCIAL				
	CONTINGENCIAS	DESEMPLEO	FOGASA	F.P.	TOTAL
EMPRESA	23,60%	5,50%	0,20%	0,60%	29,90%

$14.817,89 * (1 + 0,299) = 19.248,44$  € al año por parte de la empresa.

Esto al mes equivale a:  $19.248,44 / 12 = 1.604,04$  €, que, trabajando a jornada completa, son 160 horas al mes.

Por tanto, el coste por hora para la empresa sería de  $1.604,04 / 160 = 10,03$  € por hora.

Si tenemos en cuenta que la estimación temporal inicial suma 337 h, el resultado del coste de personal del proyecto es:  $10,03 * 337 = 3.380,11 \text{ €}$ .

### **Costes indirectos**

El equipo usado para el desarrollo es un portátil HP Omen 15-ce0xx con un coste de 1.200€, considerando el tiempo de amortización unos 4 años, la equivalencia de 3 meses es:  $3 / (4 * 12) = 0,0625 \Rightarrow 6,25\%$ . La amortización del portátil en estos 3 meses equivale a  $1.200 * 0,0625 = 75\text{€}$ .

También se ha utilizado un móvil como principal plataforma de pruebas: el modelo es un Xiaomi Mi 8 con un coste de 370€. Considerando el tiempo de amortización 2 años, el coste de amortización del mismo durante los 3 meses del proyecto es:  $370 * 3 / (2*12) = 46,25\text{€}$ .

Además de los equipos utilizados, se planea realizar una suscripción de 1 mes a la plataforma de aprendizaje ‘CG Cookie’ [6], para el aprendizaje de Unity y Blender al inicio del proyecto. El coste de esta suscripción es de 29\$ = 26,34€.

### Resumen de costes indirectos

Portátil HP Omen	75€
Móvil Xiaomi Mi 8	46,25€
Suscripción CG Cookie	26,34€
Total	147,59€

Finalmente, el coste del proyecto sin IVA es de  $3.380,11\text{€} + 147,59\text{€} = 3.527,7\text{€}$ .

Añadiéndole el IVA obtenemos un coste total estimado del proyecto de  $3527,7\text{€} * 1,21 = 4.268,52\text{€}$ .

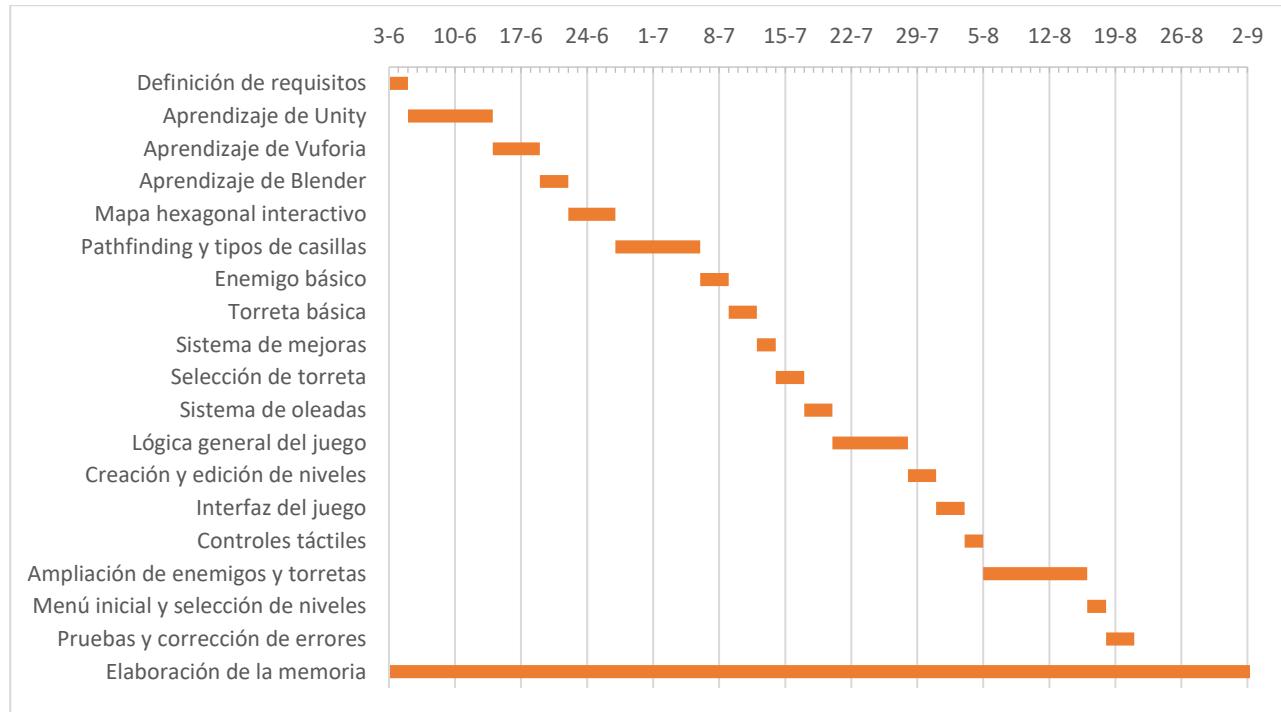
### 3.3. Planificación temporal final

A continuación, mostramos una tabla con el recuento temporal real. Este se ha tomado a ojo ya que no se ha utilizado ninguna herramienta de gestión del tiempo, por tanto, los números mostrados son estimaciones poco precisas.

*Tabla 3 - Planificación temporal final*

ID	Título	Tiempo Real
1	Definición de requisitos	8
2	Aprendizaje de Unity	28
3	Aprendizaje de Vuforia	15
4	Aprendizaje de Blender	10
5	Mapa hexagonal interactivo	17
6	Pathfinding y tipos de casillas	30
7	Enemigo básico	8
8	Torreta básica	11
9	Sistema de mejoras	5
10	Selección de torreta	10
11	Sistema de oleadas	10
12	Lógica general del juego	24
13	Creación y edición de niveles	8
14	Interfaz del juego	10
15	Controles táctiles	7
16	Ampliación de enemigos y torretas	35
17	Menú inicial y selección de niveles	5
18	Pruebas y corrección de errores	10
19	Elaboración de la memoria	70
Total		313

Tabla 4 - Planificación temporal final - diagrama Gantt



### 3.4. Planificación financiera final

Los cálculos para la planificación financiera son idénticos a los de la planificación financiera inicial, lo único que varía es el número de horas dedicadas al proyecto: 293h.

Usando de nuevo el coste por hora de un programador junior:  $10,03\text{€}/\text{h} * 313\text{h} = 3.139,39\text{€}$ .

Si le sumamos los mismos costes indirectos:  $147,59\text{€} + 3.139,39\text{€} = 3.286,98\text{€}$  sería el coste del proyecto sin IVA.

Con IVA, el coste total del proyecto es de:  $3.286,98\text{€} * 1,21 = \underline{3977,25\text{€}}$ .

## 4. Conclusiones

### Conclusiones del proyecto

La realización de un videojuego con Unity no habría sido posible sin los conocimientos previos de programación estudiados en la carrera y sin la fase de aprendizaje inicial del proyecto. Gracias a unos recursos de aprendizaje geniales, se ha conseguido consolidar un nivel básico en cuanto a manejo de la herramienta. Durante el proyecto se han practicado algunos de los flujos de desarrollo recomendados para Unity, por ejemplo, mediante el uso de ‘prefabs’, escenas, etc.

Para la integración con la realidad aumentada, a pesar de haber tenido una fase inicial de aprendizaje y experimentación de varias funcionalidades proporcionadas por Vuforia, al final del proyecto se ha terminado usando únicamente el reconocimiento de Vumarks. No obstante, la etapa inicial fue bastante útil para conocer las posibilidades del software.

Finalmente se ha conseguido desarrollar un videojuego que quizás con un poco más de pulido visual y efectos de sonido, podría ser válido para salir al mercado. Incluso con la única funcionalidad de poder jugar sobre una imagen Vumark, creo que el juego puede resultar entretenido a bastantes personas y también atractivo visualmente.

### Conclusiones personales

La ejecución de este proyecto ha supuesto una gran oportunidad para explorar las peculiaridades del mundo de desarrollo de videojuegos. Se han adquirido nuevos conocimientos sobre el proceso, tanto relacionados con la programación con C#, cómo con la herramienta Unity y su integración con Vuforia.

En definitiva, el proyecto ha supuesto una experiencia agradable, que ha permitido conocer levemente a nivel personal si realmente tengo la motivación o interés para seguir mi trayectoria profesional por esta rama del mundo de los videojuegos.

## 5. Trabajo futuro

Cómo comentaba en la sección anterior, durante el desarrollo del videojuego van surgiendo muchas ideas que se quedan en el tintero por falta de tiempo, conocimiento o recursos. A continuación, explicamos algunas de ellas:

### Mejorar sistema de oleadas

El sistema de oleadas actual funciona bien pero no permite crear conjuntos de oleada independientes entre sí que se ejecuten de forma asíncrona, es decir, que las oleadas estén compuestas por conjuntos que cada uno se inicia en un momento concreto independientemente del resto de conjuntos.

Además, el algoritmo de generación de oleadas aleatorias a partir de un número de enemigos no sigue ninguna lógica de diseño de oleadas, como por ejemplo crear oleadas en las que primero salgan enemigos resistentes y tras un tiempo grande (suficiente para que hayan avanzado una cantidad razonable del tablero), siguieran apareciendo otros conjuntos de oleadas de enemigos menos resistentes.

### Diseño de niveles

Incluir de alguna forma en el producto final un modo de diseño de niveles, en este, el jugador podrá crear niveles desde dentro del juego y guardarlos para poder jugarlos en el futuro.

### Sistema de desbloqueo de niveles

Implementar un sistema de desbloqueo de niveles, además de crear un repertorio de niveles más grande con sus oleadas establecidas y dificultad creciente. De esta forma el jugador aprendería poco a poco a jugar al juego.

### Colocación de torretas

Mejorar el sistema de colocación de torretas para que el usuario pueda ver dónde va a colocar una torreta y el alcance de esta en esa posición (similar al sistema usado en ‘GeoDefense Swarm’). También se podría añadir un sistema de confirmación, tanto para la colocación como para las mejoras y vender torretas (similar al usado en ‘Kingdom Rush’).

### Ampliación de enemigos

Aumentar la variedad de enemigos, creando nuevos tipos que cambien su funcionamiento respecto a los ya creados con mecánicas nuevas.

### Ajustar los valores de las torretas

Actualmente, los valores de daño, alcance, cadencia de disparo, coste, etc. de las torretas están establecidos por intuición. Convendría ajustar los valores para que tengan más sentido y se basen en datos, por ejemplo, realizar pruebas de simulación a nivel masivo con distintos valores y comparar los resultados. Pudiendo determinar así qué valores se ajustan mejor a la lógica de cada torreta.

### Añadir poderes especiales

Al igual que los poderes controlables del juego ‘Kingdom Rush’, se podrían crear herramientas similares para el jugador. Por ejemplo, en ese juego existen inicialmente un poder para invocar tropas en una posición que ralentizarán el paso de los enemigos y otro poder que invoca una sucesión rápida de meteoritos en la posición seleccionada haciendo gran cantidad de daño en área.

### Catálogo de elementos

Haciendo uso de la realidad aumentada, permitir al jugador explorar sujetando en su mano una imagen VuMark los distintos elementos del juego: casillas, enemigos, torretas y mejoras, etc. Funcionaría como un catálogo de elementos, en el que únicamente se renderiza un elemento en cada momento y se muestra información sobre el mismo.

### Música y efectos de sonido

Invertir tiempo en el aprendizaje y uso de música en videojuegos de Unity. Aunque no entre en el alcance del proyecto, si se hace bien, la música y el audio pueden aportar bastante ambientación al videojuego.

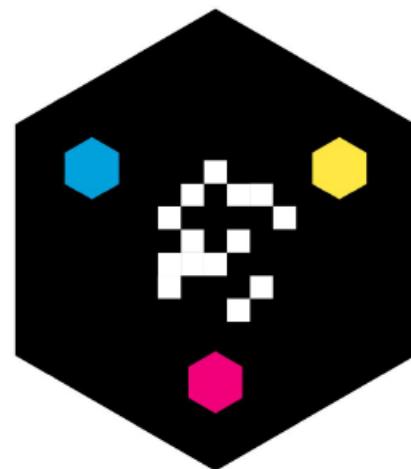
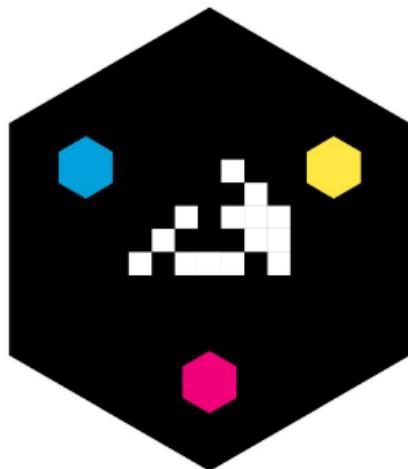
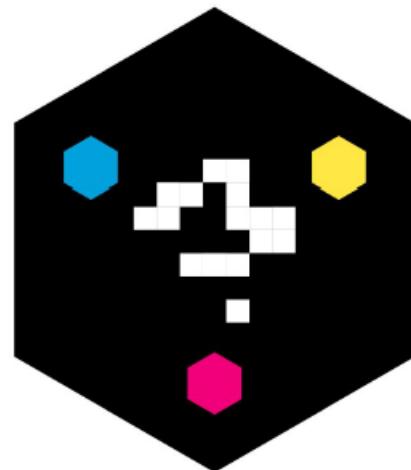
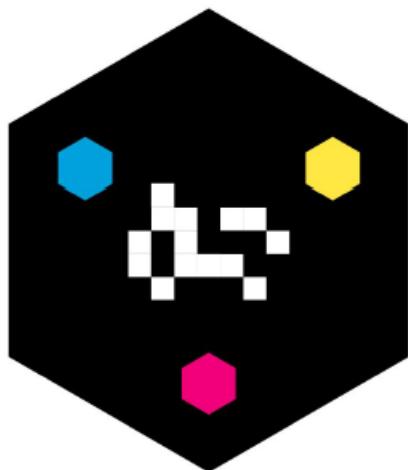
## Bibliografía

- [1] Newzoo, «Newzoo,» [En línea]. Available: <https://newzoo.com/key-numbers/>.
- [2] Wikipedia, «Juegos triple A,» [En línea]. Available: [https://es.wikipedia.org/wiki/AAA\\_\(industria\\_del\\_videojuego\)](https://es.wikipedia.org/wiki/AAA_(industria_del_videojuego)).
- [3] J. Flick, «Catlike Coding,» [En línea]. Available: <https://catlikecoding.com/unity/tutorials/>.
- [4] A. Thirlslund, «Brackeys,» [En línea]. Available: <https://www.youtube.com/user/Brackeys>.
- [5] S. Lague. [En línea]. Available: <https://www.youtube.com/user/Cercopithecan>.
- [6] «CG Cookie,» [En línea]. Available: [www.cgcookie.com](http://www.cgcookie.com).
- [7] G. Rodriguez, «vix,» [En línea]. Available: <https://www.vix.com/es/btg/gamer/4251/juegos-de-ipad-kingdom-rush>.
- [8] Jarkendia, «vidaextra,» [En línea]. Available: <https://www.vidaextra.com/analisis/plantas-contra-zombis-analisis>.
- [9] F. Jasper, «Catlike Coding - Object Management,» [En línea]. Available: <https://catlikecoding.com/unity/tutorials/object-management/>.
- [10] Karjalan, «Answers Unity,» [En línea]. Available: <https://answers.unity.com/questions/1073979/android-touches-pass-through-ui-elements.html>.
- [11] Vuforia, «Youtube - Vuforia,» [En línea]. Available: <https://www.youtube.com/watch?v=YXMiDRyvqzk>.
- [12] [En línea]. Available: <https://answers.unity.com/questions/198853/unity-remote-for-android-not-working-solution.html>.
- [13] [En línea]. Available: <https://www.gamecrate.com/statistically-video-games-are-now-most-popular-and-profitable-form-entertainment/20087>.
- [14] «appdevice,» [En línea]. Available: <https://appadvice.com/appnn/2009/09/review-geodefense-swarm>.

## Anexos

### A - Imágenes Vumark reconocibles para el juego.

Cualquiera de estas 4 imágenes es reconocible dentro del juego. Se recomienda imprimir una de ellas y colocarla en un plano horizontal. Para su correcto reconocimiento debe haber buena luz y la imagen debe permanecer plana.



## B – Manual de instalación de la aplicación

Para la instalación de la aplicación será necesario un móvil con Android y el archivo .apk que se adjuntará en los archivos del proyecto.

Antes de abrir la aplicación para su instalación, es necesario activar la opción de Instalar aplicaciones de orígenes desconocidos, se encuentra normalmente en Ajustes -> Seguridad. Esto permitirá instalar aplicaciones que no procedan de la tienda de aplicaciones. Una vez activada esa opción, podemos abrir el archivo de instalación y este procederá a su instalación de forma automática.

Al iniciar la aplicación por primera vez, se nos pedirá permisos para hacer uso de la cámara del móvil, esta opción habrá que habilitarla si se quiere usar la realidad aumentada en la aplicación.