Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Desarrollo de un videojuego utilizando Unity y C#

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

*Autor:* Radosvet Desislavov Georgiev
*Tutor:* Javier Lluch Crespo

Curso 2018-2019

# Resumen

Este trabajo trata sobre el desarrollo de un videojuego inspirado en los videojuegos de las máquinas *arcade*, concretamente en el género Shoot 'em up, el cual se caracteriza por tomar el control de un personaje e ir acabando con hordas de enemigos. En el videojuego desarrollado, el jugador controla una nave con la cual va destruyendo enemigos. El juego tiene diferentes modos de juego y se estructura en niveles. A pesar de que se inspire en juegos clásicos, la estética del juego es actual. Para desarrollarlo se ha usado el motor de videojuegos Unity y el lenguaje de programación C#. La plataforma objetivo es la Web, lo que significa que el juego se puede ejecutar desde cualquier navegador web con soporte a WebGL. Para que esto sea así se han tenido en cuenta las limitaciones en cuanto a capacidad de cómputo de esta plataforma y se ha adecuado el videojuego a ella.

**Palabras clave:** videojuego, Unity, C#, WebGL, Shoot 'em up

# Abstract

This work is about developing a video game inspired by arcade machine video games, specifically in the genre Shoot 'em up, which is characterized by taking control of a character and with it destroying hordes of enemies. In the developed video game, the player controls a ship with which he destroys enemies. The game has different game modes and it is structured in levels. Although the video game is inspired by classic video games the aesthetics are modern. To develop it, Unity game engine and C# programming language have been used. The target platform is the Web, this means that the game can be run from any web browser with WebGL support. To make this possible, the limitations in terms of computing capacity of this platform have been taken into consideration and the video game has been adapted to it.

**Key words:** video game, Unity, C#, WebGL, Shoot 'em up

# Índice general

**Índice general** v
**Índice de figuras** vii
**Índice de tablas** viii

**1 Introducción** 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 2
1.3 Metodología . . . 2
1.4 Estructura de la memoria . . . 2

**2 Estado del arte** 5
2.1 Motores de videojuegos . . . 5
2.2 Género Shoot 'em up . . . 7
2.3 Propuesta . . . 9

**3 Análisis del problema** 11
3.1 Requisitos . . . 11
3.2 Oportunidades de negocio . . . 12
3.3 Identificación y análisis de soluciones posibles . . . 13
3.4 Solución propuesta . . . 14

**4 Diseño de la solución** 17
4.1 Tecnología utilizada . . . 17
4.2 Diseño detallado de la solución . . . 22

**5 Desarrollo** 27
5.1 Creación de la nave del jugador . . . 27
5.2 Desarrollo del nivel del tutorial . . . 30
5.3 Desarrollo de los enemigos . . . 31
5.4 Generación de enemigos . . . 36
5.5 Nivel infinito . . . 38
5.6 Niveles 3D . . . 41
5.7 Jefe Final . . . 42
5.8 UI . . . 45
5.9 Persistencia . . . 47

**6 Pruebas** 49
6.1 Rendimiento con WebGL . . . 49
6.2 Dificultad y controles . . . 50
6.3 Desplazamiento de los enemigos . . . 50

**7 Resultados** 51
7.1 Rendimiento y optimización . . . 51
7.2 Dificultad y controles . . . 52
7.3 Desplazamiento de los enemigos . . . 52
7.4 Videojuego . . . 53

**8 Conclusiones** 55

**9 Trabajo futuro** **57**

**Bibliografía** **59**

**Anexo** **61**

**A Game Design Document** **61**

A.1 Descripción . . . . . . . . . . 61
A.2 Historia . . . . . . . . . . 61
A.3 Personajes . . . . . . . . . . 61
A.4 Jugabilidad . . . . . . . . . . 63
A.5 Niveles . . . . . . . . . . 63
A.6 Mecánicas . . . . . . . . . . 64
A.7 Habilidades del jugador . . . . . . . . . . 64
A.8 Sonido . . . . . . . . . . 65

# Índice de figuras


2.1 Space Invaders. . . . . . 7
2.2 Ikagura. . . . . . 8
2.3 Sky Force Reloaded. . . . . . 9
2.4 Nivel Shoot ’em up del Nier: Automata. . . . . . 9

3.1 Cuota de mercado. Datos obtenidos de StatCounter. . . . . . 12
3.2 Mercado videojuegos 2019. Datos obtenidos de Newzoo. . . . . . 13

4.1 Editor de Unity. . . . . . 18
4.2 Ventana Animator. . . . . . 18
4.3 Ventana Animation. . . . . . 19
4.4 Propiedades del componente Rigidbody. . . . . . 19
4.5 Propiedades del componente Box Collider. . . . . . 19
4.6 Relación entre escena, GameObject y componente. . . . . . 20
4.7 Leyenda del diagrama de flujo. . . . . . 20
4.8 Diagrama de flujo del ciclo de vida del *script*. . . . . . 21
4.9 Nave que controlará el jugador. . . . . . 22
4.10 Drones enemigos. . . . . . 22
4.11 Modelo 3D del jefe final del juego. . . . . . 23
4.12 Diagrama de clases UML. . . . . . 24
4.13 Esquema del patrón publicación-subscripción. . . . . . 26

5.1 Capa del Animator que controla la inclinación. . . . . . 28
5.2 Capa del Animator que controla la onda. . . . . . 28
5.3 Componentes que forman el GameObject de la nave del jugador. . . . . . 29
5.4 Código comentado de la clase PlayerCharacter. . . . . . 29
5.5 Jerarquía de GameObject de la nave del jugador. . . . . . 30
5.6 Captura del nivel del tutorial. . . . . . 31
5.7 Drones explosivos. . . . . . 32
5.8 Dron que lanza rayos. . . . . . 33
5.9 Dron explosivo con *gizmos*. . . . . . 34
5.10 Selección de capa en un GameObject. . . . . . 35
5.11 Pseudocódigo de la función usada para mover a un dron. . . . . . 35
5.12 Ejemplo de como se obtienen las coordenadas para ubicar enemigos. . . . . . 36
5.13 Cabecera de las funciones de creación de drones. . . . . . 38
5.14 Ventana del nivel infinito. . . . . . 39
5.15 Gráfica de la función. . . . . . 40
5.16 Captura de pantalla de uno de los niveles 3D. . . . . . 42
5.17 Máquina de estados del jefe final. . . . . . 43
5.18 Circunferencia unitaria. . . . . . 43
5.19 El jefe final disparando esferas en forma de circunferencia. . . . . . 44
5.20 Menú principal. . . . . . 45
5.21 Ventana de opciones en el menú principal. . . . . . 46
5.22 Ventana de opciones en el menú principal. . . . . . 46

7.1 Captura de pantalla del nivel 4. . . . . . . . . . . . . . . . . . . . . . . . . 53
7.2 Captura de pantalla del nivel 5. . . . . . . . . . . . . . . . . . . . . . . . . 54
7.3 Captura de pantalla del nivel 7. . . . . . . . . . . . . . . . . . . . . . . . . 54

A.1 Nave aliada. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.2 Dron normal. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.3 Dron avanzado. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.4 Dron explosivo. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.5 Dron que lanza rayos. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.6 Jefe final. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

# Índice de tablas

3.1 Requisitos funcionales. . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.2 Requisitos no funcionales. . . . . . . . . . . . . . . . . . . . . . . . . . 12

5.1 Probabilidades. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.2 Tabla de valores. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

# CAPÍTULO 1
# Introducción

En este primer capítulo se expondrán los motivos que me han llevado a realizar este trabajo, los objetivos que se pretenden conseguir, la metodología aplicada y la estructura que seguirá la memoria. El proyecto consiste en el desarrollo de un videojuego usando el motor de videojuegos Unity [1] y el lenguaje de programación C#. El videojuego consistirá principalmente en manejar una nave e ir acabando con los enemigos que se presentan, aunque también habrá niveles con mecánicas diferentes. El nombre elegido para el videojuego es *Attack of the Drones*.

## 1.1 Motivación

Hoy en día la industria del videojuego se encuentra en continuo crecimiento y cada año alcanza una mayor facturación. Como ejemplo para tener una referencia es interesante comentar que esta industria en 2016 facturó 99.600 millones de dólares [1]. El gran avance tecnológico de los últimos años ha hecho posible ejecutar videojuegos en una gran multitud de dispositivos, tales como teléfonos inteligentes y tabletas, y gracias a esto se ha llegado a un público mucho más amplio.

Otro motivo es la gran presencia de conceptos de informática en el desarrollo de un videojuego. Se necesitan usar diversas estructuras de datos, implementar inteligencia artificial, gestionar las conexiones entre los jugadores en caso de juegos en red, etc. Hoy en día hay gran variedad de motores de videojuegos que facilitan el desarrollo, aun así los videojuegos más relevantes se desarrollan durante años y requieren una planificación compleja.

La elección del tipo de juego es la que se presenta porque es un juego que se adaptará bien a plataformas con poca capacidad de cómputo, tales como dispositivos móviles o páginas web. Y es en estos sitios donde los videojuegos pequeños pueden llegar a tener éxito. Tras el desarrollo, este podría ser implantado en la web de Minijuegos [2], si llega a ser seleccionado para ello.

También está presente una motivación personal, ya que los videojuegos son algo que siempre me ha gustado y estaría interesado en trabajar en su desarrollo en un futuro. Realizando este trabajo podré obtener conocimiento y experiencia sobre cómo desarrollar videojuegos en Unity.

[1] Página web de Unity: https://unity.com/es
[2] Página web de Minijuegos: https://www.minijuegos.com/

## 1.2 Objetivos

Los objetivos que se buscan cumplir con este trabajo son los siguientes:

- Desarrollar un videojuego sencillo pero a la vez completo.
  - Implementar una jugabilidad satisfactoria.
  - Diseñar e implementar una interfaz de usuario intuitiva.
  - Añadir música y efectos de sonidos adecuados para cada nivel.
- Desarrollar un videojuego que se pueda jugar fluidamente en un navegador web.
- Crear un videojuego que sea entretenido de jugar.
  - Incluir diferentes modos de juego.
  - Introducir novedades en cada nivel.
- Familiarizarse con el entorno de desarrollo que proporciona el motor de videojuegos Unity y con el uso de C# en este motor.

## 1.3 Metodología

Dado que el modelado y el diseño no es el campo en el que se centra este trabajo, los modelos 3D, texturas, sonidos y música para el videojuego se obtendrán principalmente de la Asset Store [3] de Unity o en caso de que no haya nada adecuado disponible se recurrirá a otras fuentes de internet, siempre respetando los derechos de autor. Una vez obtenidos los elementos gráficos y de audio se importarán en Unity y se empezará con la parte de programación y diseño de niveles. El desarrollo empezará por los niveles más básicos y tras desarrollar un nivel del videojuego se probará para comprobar si el funcionamiento es correcto. En caso de que no lo sea se modificará antes de pasar al desarrollo del siguiente nivel ya que los niveles siguientes serán como una ampliación de los previos. Al final, cuando estén todos los niveles acabados, se harán más pruebas para evaluar diferentes aspectos del videojuego y modificarlos si es necesario.

## 1.4 Estructura de la memoria

Esta memoria está compuesta por 9 capítulos, la bibliografía y un anexo en el cual se puede encontrar el Game Design Document (GDD).

En el primer capítulo, llamado «Introducción», se expondrá el proyecto de forma global, desarrollando para ello las secciones de motivación, objetivos, metodología y estructura de la memoria. En el segundo capítulo: «Estado del arte», se empezará exponiendo brevemente la historia de los motores de videojuegos y se verán algunos motores de videojuegos actuales. Se seguirá con la historia del género de videojuegos al que pertenece el juego que se desarrollará y posteriormente se verán algunos ejemplos destacados de este género. El capítulo finalizará con una breve propuesta para el proyecto.

El tercer capítulo, el cual se titula «Análisis del problema», empezará con la especificación de los requisitos del proyecto, a continuación se estudiarán las oportunidades de negocio, y después se identificarán y analizarán diferentes soluciones posibles. Por último, se elegirá una de esas soluciones y se desarrollará en profundidad. Tras analizar el

[3] Dirección web de la Asset Store: https://assetstore.unity.com/

problema se seguirá con el capítulo «Diseño de la solución», en el cual primero se expondrán las herramientas que se van a utilizar en el desarrollo del proyecto y, a continuación, se detallarán diferentes aspectos del diseño de la solución, explicando la arquitectura de la solución entre otras cosas.

Después de todos los capítulos anteriores llega el turno al capítulo llamado «Desarrollo», el cual es el número 5. Este es el capítulo más extenso de la memoria y en él se explica cómo se han desarrollado las diversas partes que forman el videojuego, entrando en detalles en los aspectos más complejos de este. En el sexto capítulo, llamado «Pruebas», se proponen diversas verificaciones que se llevarán a cabo en el videojuego, estas pruebas se pueden separar en 3 categorías que son las secciones del capítulo: pruebas de rendimiento, pruebas de los controles y la dificultad y pruebas del algoritmo de desplazamiento de los enemigos. Después de esto se encuentra el capítulo «Resultados», en el cual se exponen los resultados de las pruebas propuestas en el capítulo anterior y los cambios que se han realizado tras las pruebas. Por último, dentro de ese capítulo se encuentra la sección llamada «Videojuego», en esta se expone brevemente el resultado final obtenido después del desarrollo y las modificaciones realizadas tras las pruebas.

El penúltimo capítulo son las conclusiones del trabajo, y el último capítulo, nombrado «Trabajo futuro», recoge algunas ideas para mejorar el proyecto. A continuación se encuentra la bibliografía de la memoria y el anexo con el Game Design Document.

# CAPÍTULO 2
# Estado del arte

El origen de los videojuegos se remonta a los años 50, muchos consideran como el primer videojuego a Nougth and Crosses, también conocido como OXO, que era un tres en raya que se podía jugar contra la máquina [2]. Este era ejecutado en el EDSAC [1] —que era una computadora británica terminada de construir en 1949—. Desde aquel entonces los videojuegos han evolucionado enormemente, pasando por las máquinas recreativas, videoconsolas con juegos 2D, etc. Hasta llegar a día de hoy que podemos ejecutar videojuegos 3D en multitud de diferentes dispositivos.

## 2.1 Motores de videojuegos

Hasta principios de los 90 los videojuegos se desarrollaban desde cero. Durante el desarrollo de la primera serie de Commander Keen sus programadores se encontraron con unos plazos de producción difíciles de cumplir, y debido a eso se les ocurrió reutilizar una parte del código en los 3 juegos de la serie. A esa parte común de los 3 videojuegos la llamaron Keen Engine. Este motor era capaz de ofrecer algunas funcionalidades genéricas que podían ser útiles en muchos juegos [3]. Posteriormente, intentaron licenciar el motor, pero no tuvieron éxito.

La verdadera revolución se dio en 1993 con el juego Doom[2], el cual usaba un motor de videojuegos bastante logrado para la época, llamado Doom Engine. Los gráficos que se conseguían con él parecían 3D aunque en realidad no lo eran, no se le puede considerar un motor 3D debido a ciertas limitaciones que tenía [3, 4]. Este fue posteriormente usado para la creación de más videojuegos. A partir de entonces se empezó a usar el termino motor de videojuegos, o *game engine* en inglés. En la actualidad todos los videojuegos comerciales se desarrollan con uno de estos, ya que los motores de videojuegos permiten un desarrollo más rápido y menos costoso. Además, dada la complejidad de los videojuegos actuales, se hace necesario disponer de una herramienta la cual proporcione funciones que faciliten el desarrollo y la reutilización de elementos.

Muchas empresas del sector de los videojuegos desarrollan sus propios motores, y no es posible utilizarlos a menos que trabajes en sus proyectos. Pero existen varios motores de videojuegos de buena calidad disponibles para cualquiera. A continuación se verán algunos de los más utilizados de estos a día de hoy.

[1] Acrónimo de Electronic Delay Storage Automatic Calculator

[2] https://es.wikipedia.org/wiki/Doom_(videojuego_de_1993)

### 2.1.1. Unity

La primera versión de Unity fue lanzada en junio de 2005, desde entonces el motor ha ido evolucionando y se han ido añadiendo más herramientas para los desarrolladores. En sus principios se podía programar en C#, UnityScript —lenguaje propio con la sintaxis de JavaScript— y Boo, pero hoy en día solo se pueden usar los 2 primeros aunque UnityScript está en desuso y se pretende dejar C# como único lenguaje posible en un futuro cercano.

En los últimos años, este motor ha ganado gran popularidad, sobre todo entre los desarrolladores *indie*. Esto es debido a varios factores, uno de ellos es que posee una curva de aprendizaje menor que otras alternativas disponibles, pero a pesar de ello se pueden obtener resultados de alta calidad. Además, es posible publicar los videojuegos desarrollados en gran número de plataformas. Unity tiene una comunidad muy grande con múltiples recursos disponibles para aprender, también está la Asset Store, donde se puede encontrar contenido gratuito y de pago para usar en los proyectos. Actualmente hay gran multitud de videojuegos desarrollados con Unity [3], aunque los proyectos más ambiciosos de la industria suelen desarrollarse con motores más avanzados.

Hoy en día hay disponibles tres versiones de Unity. La personal es la versión gratuita y más básica, pero esta no tiene grandes limitaciones respecto a las demás si no estás trabajando en equipo. Las otras dos son la Plus y la Pro, esta última es la versión más avanzada y la indicada para grandes equipos de desarrollo.

### 2.1.2. Unreal Engine

Es un motor de videojuegos creado por la compañía Epic Games. La primera versión de este fue lanzada en 1998. En este motor para programar los videojuegos se utiliza el lenguaje de programación C++, e igual que Unity permite exportar los proyectos a muchas plataformas distintas. La última versión —Unreal Engine 4— fue lanzada en 2014, y desde 2015 se puede descargar y usar gratuitamente con la condición de que si obtienes beneficios superiores a los 3000$ tienes que pagar a Epic Games un 5 % de lo ganado. El código fuente de Unreal Engine está escrito en C++ y es posible acceder a él para ver cómo funciona el motor o para modificarlo.

Es una herramienta para desarrolladores expertos, requiere bastante tiempo de aprendizaje, pero con esta se pueden conseguir resultados superiores a los que se obtendrían con Unity. Actualmente muchos títulos importantes usan este motor [4].

### 2.1.3. CryEngine

Creado por la empresa alemana Crytek originalmente como un motor de demostración para la empresa Nvidia —empresa principalmente dedicada a la fabricación de tarjetas gráficas—. Tras demostrar un gran potencial, fue anunciado oficialmente en 2002 y se usó por primera vez en el videojuego Far Cry, que se convirtió en una serie de videojuegos muy popular.

CryEngine ha pasado por diferentes versiones y por diferentes modelos de negocio, pero hoy en día es gratuito y permite el acceso a su código fuente, igual que Unreal Engine. Hay que pagar a la empresa creadora solo en el caso de que comercialicemos el videojuego y alcancemos cierto umbral de beneficios. Desde su creación diferentes

[3] https://unity.com/es/madewith

[4] https://www.unrealengine.com/en-US/blog/the-gaming-industry-gets-set-for-an-unreal-2018

versiones del motor han sido compradas por empresas para crear su propio motor a partir de la versión adquirida.

## 2.2 Género Shoot 'em up

El videojuego que se va a desarrollar pertenece al género Shoot 'em up, que en ocasiones se traduce al castellano como matamarcianos. En este género, el jugador controla un personaje u objeto, típicamente una nave espacial o un avión, que dispara contra hordas de enemigos que van apareciendo en pantalla. El movimiento se limita a dos ejes, aunque los gráficos sean 3D. Los juegos pertenecientes a este género requieren reacciones rápidas.

El primer videojuego del género es Spacewar!, que, además, es uno de los primeros videojuegos. Este fue desarrollado en 1961 y publicado en máquinas *arcade* a principios de la década de los 70. Aunque el juego Space Invaders, publicado en las máquinas japonesas en 1978, se acredita muchas veces como el creador del género debido al gran éxito que obtuvo. A medida que se iba desarrollando más el género aparecieron diversos subgéneros de este. Es interesante mencionar el subgénero *bullet hell*, ya que Attack Of The Drones coge algunas características de este. Los juegos pertenecientes a *bullet hell* se caracterizan por las enormes cantidades de proyectiles enemigos en pantalla lo que implica que requieran reacciones aún más rápidas [5].

### 2.2.1. Space Invaders

Es un videojuego de *arcade* diseñado por Toshihiro Nishikado y lanzado al mercado en 1978 en Japón y más tarde en Estados Unidos. Es un videojuego en dos dimensiones, en el cual el jugador controla un cañón que puede moverse hacia la derecha o la izquierda. El objetivo es ir destruyendo a los extraterrestres invasores que van acercándose a la tierra. Con el paso del tiempo estos se acercan más rápidamente. El juego acaba cuando el jugador pierde, y esto ocurre cuando los invasores llegan al cañón. Se han realizado gran cantidad de adaptaciones de este juego a lo largo de los años para diferentes plataformas. En la figura 2.1 se puede ver la versión original de Space Invaders.

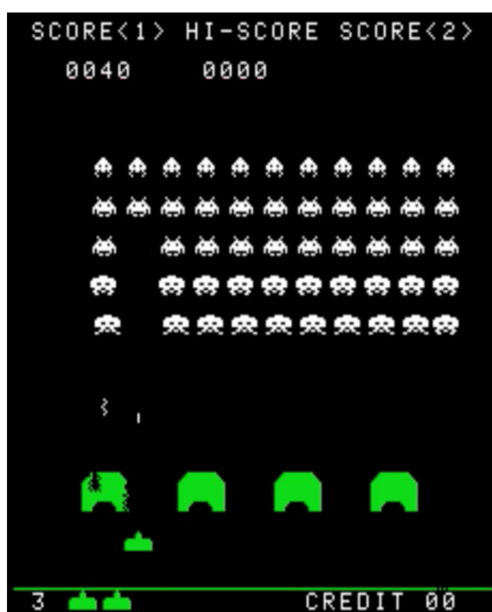


**Figura 2.1:** Space Invaders.

### 2.2.2. Ikagura

Ikagura fue desarrollado y publicado por Treasure. Este videojuego fue lanzado primeramente en máquinas recreativas en diciembre de 2001. Un año después se lanzó en Dreamcast, y en 2003 fue publicado en GameCube. A fecha de hoy se han lanzado varias versiones adaptadas de este juego a las plataformas actuales —está disponible en PS4, Nintendo Switch, Xbox 360 y Xbox One—.

Es un videojuego 3D, aunque como es normal en el género no es posible desplazarse en las tres dimensiones, sino que es solo posible moverse en 2 dimensiones. Este juego tiene algunas mecánicas distintas a los demás juegos del género Shoot 'em up. Los enemigos que se pueden encontrar tendrán dos polaridades diferentes: blanca o negra. La nave del jugador podrá cambiar su polaridad para enfrentar a los enemigos, si la polaridad de los disparos es opuesta a la de la nave del jugador, entonces pueden dañarla, en cambio, si los disparos son del mismo color que la nave, esta los absorbe y los transforma en energía, la cual recarga el arma especial. En la figura 2.2 se puede ver una pantalla del juego de una versión adaptada a las pantallas actuales.

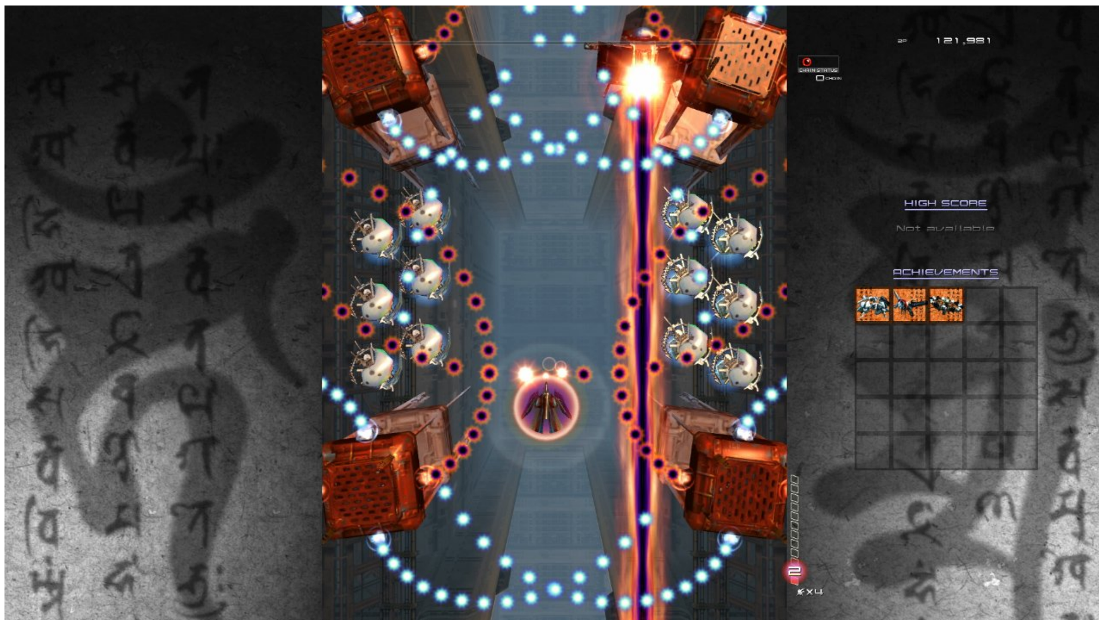


**Figura 2.2:** Ikagura.

### 2.2.3. Últimos años

Con el avance tecnológico, este género ha perdido la importancia que tenía en el pasado, dado que es posible crear videojuegos con mundos complejos. Pero se siguen pudiendo encontrar algunos juegos con gráficos actuales que apuestan por este género. El videojuego Sky Force Reloaded, el cual fue lanzado inicialmente en 2006, tiene un *remake* para los dispositivos actuales que fue lanzado en 2016 para móviles, en 2017 para PS4 y Xbox One y en 2018 para Nintendo Switch. En la figura 2.3 se puede ver cómo es la nueva versión de este. En el videojuego Nier: Automata, lanzado en el año 2017, se pueden encontrar algunos niveles de este género, aunque el juego no sea de este género principalmente —es un videojuego de rol de acción— tiene numerosos niveles de combate de aviones que sí lo son, un ejemplo se puede observar en la figura 2.4.

**Figura 2.3:** Sky Force Reloaded.

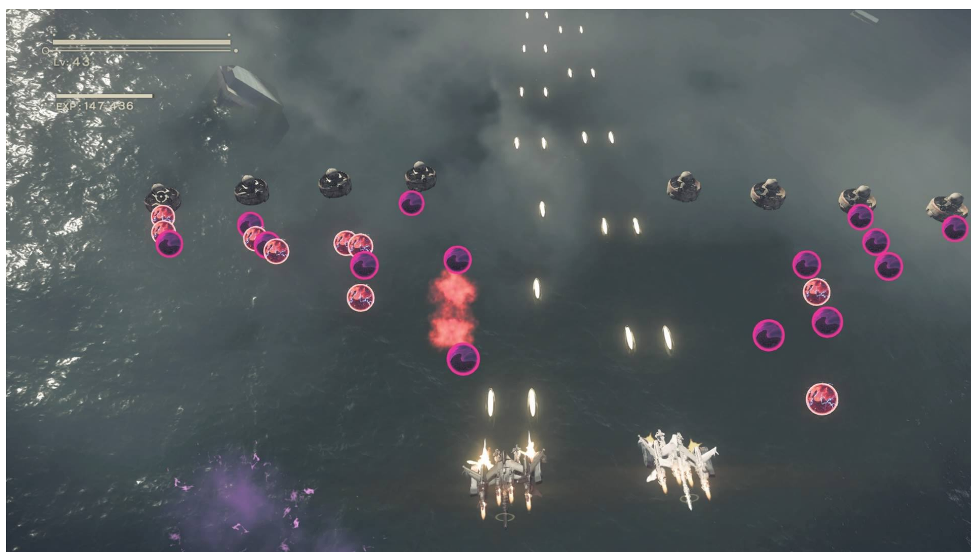

**Figura 2.4:** Nivel Shoot ’em up del Nier: Automata.

## 2.3 Propuesta

En cuanto al motor de videojuegos, en este trabajo se usará Unity, dado que tiene una curva de aprendizaje adecuada para este trabajo y ha sido utilizado en algunas asignaturas de la carrera. Además, utiliza el lenguaje C#, el cual es un lenguaje de programación orientado a objetos que tiene muchas similitudes con Java, lenguaje estudiado durante varios cursos del grado.

El videojuego será principalmente un juego del género Shoot ’em up, con modelos 3D y el movimiento de la nave restringido a 2 dimensiones. Pero también tendrá algunos pocos niveles que se alejen de este género en los cuales la cámara estará ubicada detrás de la nave y el objetivo será llegar al final del nivel sin chocar con ningún obstáculo. Esto aportará una forma totalmente nueva de jugar, ya que estos niveles se presentarán después de varios niveles Shoot ’em up y obligarán al jugador a adaptarse a la nueva jugabilidad haciendo el juego menos repetitivo y más desafiante.

# CAPÍTULO 3
# Análisis del problema

En este capítulo se empezarán analizando los diferentes requisitos que deberá cumplir el proyecto. A continuación, se analizarán las oportunidades de negocio y, por último, se identificarán diferentes soluciones posibles y se elegirá una de ellas como la más adecuada.

## 3.1 Requisitos

En las siguientes dos tablas se pueden observar los principales requisitos funcionales y no funcionales que deberá cumplir el videojuego.

| **Requisito** | **Descripción** |
|---|---|
| Guardado | Permitir guardar el progreso de la partida para que el usuario pueda continuar la partida tras cerrar la aplicación. |
| Nueva Partida | Se podrá empezar partida nueva aunque se tenga una partida guardada. |
| Pausa | Permitir pausar el videojuego en cualquier instante. |
| Opciones | Desde el menú principal se debe poder controlar el volumen del sonido y la sensibilidad del ratón. |
| Récord | Mostrar la mayor puntuación obtenida hasta el momento en los niveles que haya puntuación. |
| Niveles | El videojuego estará compuesto por varios niveles, los cuales el usuario deberá superar para poder avanzar a los siguientes. |
| Tutorial | Se le debe explicar al usuario como jugar dentro del juego, incluir nivel de aprendizaje. |

**Tabla 3.1:** Requisitos funcionales.

| Requisito | Descripción |
|---|---|
| Optimización | El videojuego deberá poder ejecutarse en plataformas con poca capacidad de cómputo. |
| Dificultad | Se buscará un equilibrio en la dificultad del juego, que no sea muy fácil ni muy difícil. |
| Diversidad | Los niveles deben ser distintos entre sí para evitar que se sienta como un juego repetitivo. |
| IU | La interfaz de usuario debe ser intuitiva y seguir las tendencias del sector para que no resulte confusa. |

**Tabla 3.2:** Requisitos no funcionales.

## 3.2 Oportunidades de negocio

Dado que es un juego relativamente simple, no puede competir contra los videojuegos desarrollados para las videoconsolas actuales. Las plataformas en las que los videojuegos de esta índole pueden tener éxito son los dispositivos móviles y las páginas web, en estas plataformas el usuario no busca jugar a videojuegos muy complejos, sino entretenerse durante un rato libre del que dispone.

Hoy en día si hubiese que elegir entre dispositivos móviles o páginas web, lo más razonable sería elegir desarrollarlo para dispositivos móviles, y concretamente para Android, ya que es el sistema operativo que mayor número de dispositivos usan —ver figura 3.1—. Además, es más sencillo publicar aplicaciones en su tienda de aplicaciones en comparación con la de iOS. Los dispositivos móviles son el segmento que más dinero genera en el mercado de los videojuegos, tal como se puede observar en la figura 3.2. Esto puede parecer extraño a primera vista, ya que los videojuegos de ordenador o videoconsolas tienen precios mucho más elevados, pero lo que hay que tener en cuenta es que en los dispositivos móviles los videojuegos se monetizan de otra manera. Es muy común poner anuncios dentro de estos o que se puedan hacer compras dentro del juego con dinero real.

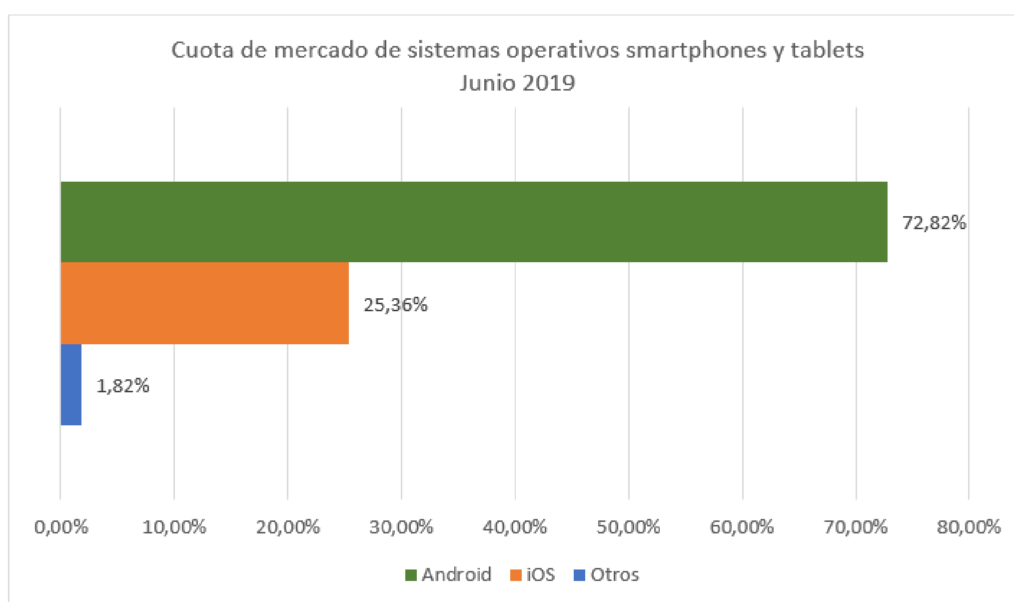


**Figura 3.1:** Cuota de mercado. Datos obtenidos de StatCounter.

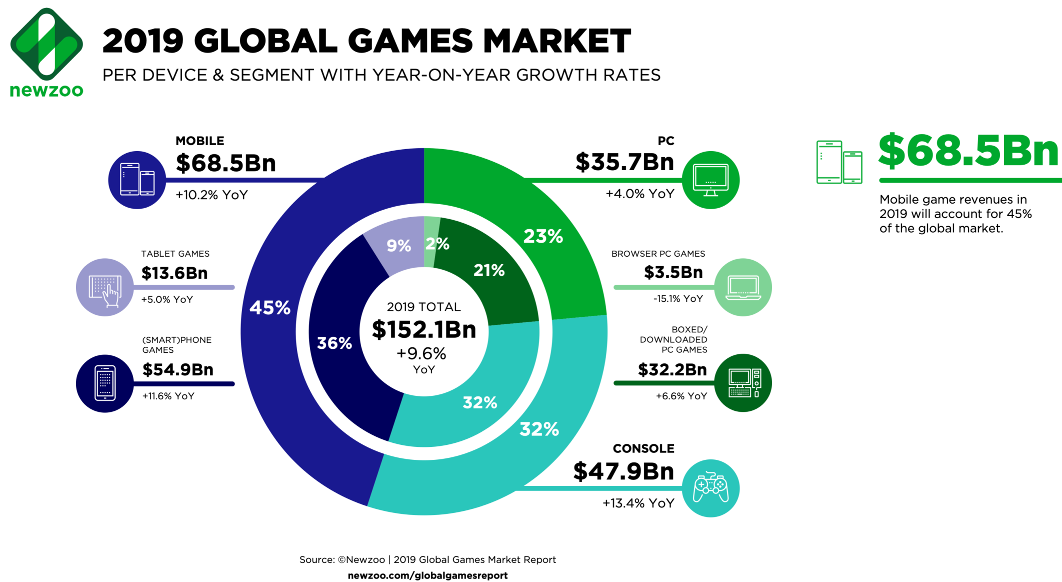


**Figura 3.2:** Mercado videojuegos 2019. Datos obtenidos de Newzoo.

Pero dado que el tutor ha comentado que se organizará un concurso para videojuegos WebGL, el videojuego se desarrollará para la Web. A pesar de esto, exportar el videojuego a Android no sería problemático, solo requeriría adaptar los controles, el rendimiento del juego no sería un problema, ya que los dispositivos móviles de hoy en día tienen suficiente potencia para poder ejecutar un videojuego que va bien en la Web. En resumen, tras el concurso el juego puede ser fácilmente adaptado para dispositivos móviles. Lo más conveniente en este caso sería permitir su descarga gratuitamente dentro de la tienda de aplicaciones, y una vez que se consiga un cierto número de jugadores poner anuncios dentro del juego.

También cabe comentar en este punto que los textos dentro del videojuego estarán en inglés. Lo más común en la industria es que el idioma principal del videojuego sea el inglés y tenga traducciones a distintos idiomas. Dado que traducir el juego en varios idiomas no sería el trabajo de un informático y requeriría bastante tiempo, el juego estará disponible en principio solo en inglés, ya que si se aspira a tener usuarios de todas las regiones es lo más adecuado.

## 3.3 Identificación y análisis de soluciones posibles

Teniendo en cuenta las horas que se deben dedicar a un trabajo fin de grado, se han barajado dos posibles opciones para el diseño del videojuego.

La primera opción consiste en que el videojuego se base en una campaña larga. Esta tendría muchos niveles y bastante duración. Pero en este caso habría que sacrificar la diversidad de los niveles y hacerlos más repetitivos, ya que desarrollar muchos niveles y muy distintos requeriría mucho tiempo y no sería viable para trabajos de esta índole.

La otra opción es que el videojuego se componga solo de unos 10 niveles y que sean más variados, que tengan diferentes escenarios y distintos tipos de enemigos. De esta forma el juego será más entretenido y se programarían más elementos diferentes. También en este caso se puede añadir un modo de juego distinto aparte de la campaña, se podría hacer un modo de juego el cual consista en acabar con el mayor número de enemigos

posibles y que no tenga fin. Así se promovería la rejugabilidad y la competición, los jugadores podrían competir con ellos mismos o con otros para obtener la puntuación más alta.

## 3.4 Solución propuesta

De las dos posibles soluciones propuestas anteriormente se ha elegido la segunda, aunque el videojuego no tenga una campaña muy larga, en esta solución se dan varias ventajas importantes que no están en la primera opción.

Al iniciar el programa la primera pantalla será el menú principal en el cual habrá varios botones. Se podrá elegir entre dos modos de juego, la campaña y el modo llamado Infinity Battle. También habrá un botón llamado *continue*, el cual permitirá reanudar la partida desde el nivel en el que se ha quedado el jugador en el caso de que haya jugado a la campaña previamente. Otro botón que es necesario que esté en el menú principal es el botón de opciones, desde el cual se podrá abrir una ventana en la cual se podrán configurar algunos parámetros.

La campaña consistirá en una sucesión de niveles los cuales habrá que ir superando para avanzar. En la primera pantalla de la campaña se explicará brevemente la historia del mundo en el que se desarrolla el videojuego. Al ir avanzando en la campaña cada vez aparecerán más tipos de enemigos y la dificultad aumentará.

Estructura de niveles de la campaña:

- Nivel 1: historia, se explicará un poco de trasfondo del mundo en el que se desarrolla el videojuego mediante texto.
- Nivel 2: tutorial, este nivel tiene como objetivo familiarizar al usuario con los controles y las mecánicas del videojuego.
- Nivel 3: nivel de combate, en estos niveles la cámara está ubicada encima del escenario y es ortográfica, el jugador se desplazará en 2 dimensiones para evitar los disparos enemigos y para disparar a los enemigos, la nave que se controla solo puede disparar hacia delante, este tipo de nivel es del género Shoot 'em up explicado anteriormente.
- Nivel 4: nivel de combate como el anterior pero con diferentes formaciones de enemigos y con más tipos de enemigos.
- Nivel 5: nivel de esquivar obstáculos, en este la cámara se posicionará atrás de la nave y será perspectiva. El objetivo es que el jugador llegue al final de un túnel lleno de obstáculos sin chocarse.
- Nivel 6: nivel de combate, en este nivel ya se usan todos los tipos de enemigos que se desarrollarán.
- Nivel 7: nivel de combate similar al anterior pero de mayor dificultad.
- Nivel 8: nivel de esquivar, como el nivel 5, pero con diferente escenario.
- Nivel 9: nivel del jefe final, un único enemigo difícil de vencer con comportamientos variados.

El modo Infinity Battle es un nivel de combate en el que irán apareciendo enemigos en posiciones aleatorias de la pantalla. Cada tipo de enemigo sumará una cantidad diferente

de puntos a la puntuación del jugador al ser destruido por este. Con el paso del tiempo se irán generando enemigos más rápidamente para dificultar la supervivencia del jugador. Existirá un objeto que al ser recogido devolverá puntos de vida a la nave, este objeto aparecerá después de un determinado tiempo que será preestablecido durante el desarrollo, así las condiciones siempre serán las mismas. La puntuación más alta alcanzada será almacenada y se mostrará durante las partidas, el objetivo es superar esa puntuación y establecer un récord.

Se empezará el desarrollo por el nivel del tutorial. A pesar de que es uno de los niveles más sencillos, para desarrollarlo hará falta implementar muchas partes del juego, que después se podrán reutilizar para los próximos niveles, como el fondo sobre el que ocurre la acción —el suelo—, la nave del jugador, los ataques de los que dispondrá, un enemigo con el que probar cómo funciona el combate, etc. Después de acabar este nivel se desarrollarán los demás niveles de combate de la campaña, para ello hará falta implementar más tipos de enemigos y diseñar diversas formaciones. Esto se puede ver como un desarrollo incremental, ya que se empieza creando los nivel más sencillo para continuar con los niveles que tendrán más elementos, los cuales son parecidos a los anteriores pero con más tipos de enemigos y formaciones más complejas.

Tras tener todos los niveles de combate implementados se procederá a crear los niveles de esquivar obstáculos, estos supondrán bastante trabajo, debido a que no se puede reutilizar mucho de los niveles de combate. En los niveles de esquivar, los controles serán totalmente distintos y hará falta desarrollar obstáculos, el túnel y más cosas, en resumen, toda la geometría 3D. Para concluir con el desarrollo de la campaña, se diseñará e implementará el jefe final, el cual tendrá un comportamiento mucho más complejo que el de los enemigos normales.

Por último, se desarrollará el nivel infinito, el menú principal y el menú de pausa. El nivel infinito presentará algunas dificultadas porque es necesario controlar en qué posiciones ya hay enemigos para no generar enemigos en esas mismas posiciones. Además, habrá que aumentar los enemigos con el paso del tiempo. Al finalizar todo el desarrollo se probarán todos los niveles con detenimiento para buscar posibles fallos y para comprobar si el nivel de dificultad del juego es el buscado.

# CAPÍTULO 4
# Diseño de la solución

En el presente capítulo se presentarán las herramientas utilizadas, prestando especial atención al motor de videojuegos Unity, ya que es importante saber cómo funciona para hacer un buen uso de este y para aprovechar las herramientas que proporciona durante el desarrollo. Además, el motor condiciona la estructura que seguirán los componentes del proyecto. Tras el apartado que trata sobre la tecnología utilizada se encuentra el apartado de diseño de la solución.

## 4.1 Tecnología utilizada

Las tres principales herramientas usadas para el desarrollo de este proyecto son Unity, GIMP y Visual Studio. A continuación se encuentra lo básico sobre estas herramientas. La información de los apartados 4.1.1 y 4.1.2 ha sido aprendida principalmente del manual de usuario de Unity [6] y de un libro sobre desarrollo de videojuegos con este motor [7].

### 4.1.1. El editor de Unity

La interfaz de Unity es bastante intuitiva y tiene muchas opciones de personalización para que cada usuario la adapte a sus necesidades y gustos. Durante las diferentes fases del desarrollo conviene tener distintas ventanas abiertas en el editor, a continuación se puede observar la distribución por defecto del editor de Unity, ver figura 4.1.

El editor tiene muchas ventanas que se le pueden añadir, las más importantes son:

- Scene View: es la ventana que se encuentra en medio de la imagen de la figura 4.1 y en ella se colocan los elementos que compondrán la escena, llamados GameObjects. Las escenas contienen los entornos del videojuego y los menús.
- Hierarchy: aquí se muestran los objetos que componen la escena actual. Los objetos se organizan jerárquicamente como un grafo de escena, esto quiere decir que pueden tener relaciones de padre/hijo entre sí.
- Project View: contiene todos los recursos del proyecto —llamados *assets*—, estos se suelen organizar en diferentes carpetas porque cuando el proyecto crece un poco si no se organizan bien el desarrollo es más complicado debido a la dificultad para encontrar los recursos que se necesitan. Normalmente se arrastran elementos de esta ventana a la ventana de escena.
- Inspector: en esta ventana se muestran las propiedades y los componentes que forman el GameObject seleccionado.

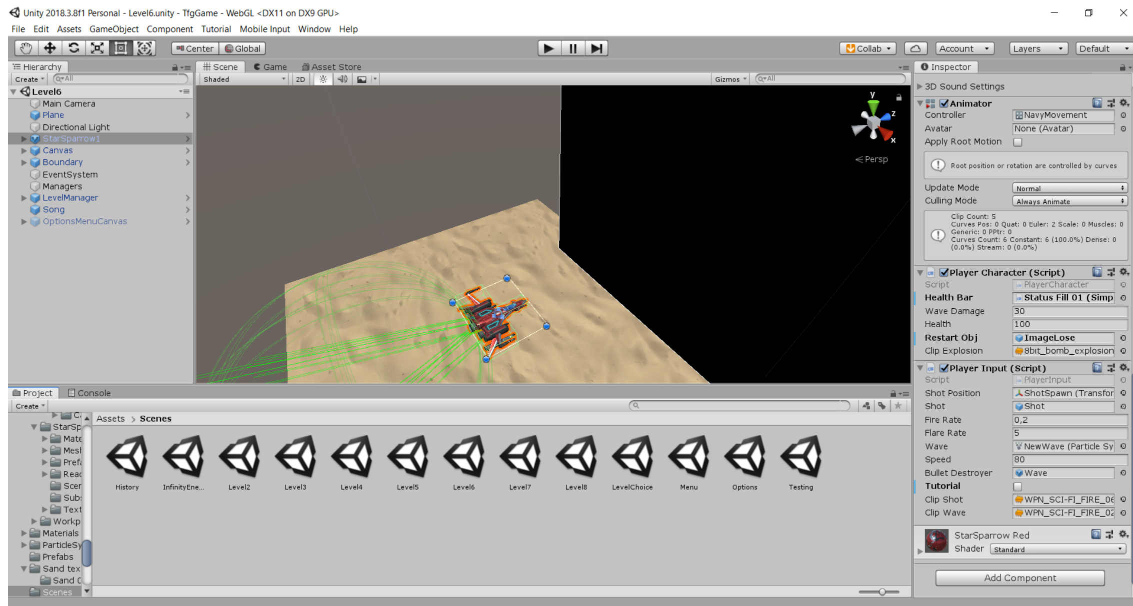


**Figura 4.1:** Editor de Unity.

- Game View: cuando se presiona el botón *play*, ubicado en la barra de herramientas que se encuentra en la parte superior de la interfaz, se ejecuta el videojuego. En esta ventana se puede jugar como si se tratará del videojuego final pero con la posibilidad de modificar parámetros en el inspector durante la ejecución y ver la consola, que es otra ventana disponible en Unity.

- Animator: ventana que permite crear y editar *assets* de tipo Animator Controller, como indica el nombre estos objetos se utilizan para controlar las animaciones y se componen de Animation Clips. El Animator Controller es un grafo dirigido, llamado máquina de estados, en el cual se pueden controlar las transiciones entre los diferentes clips mediante parámetros.

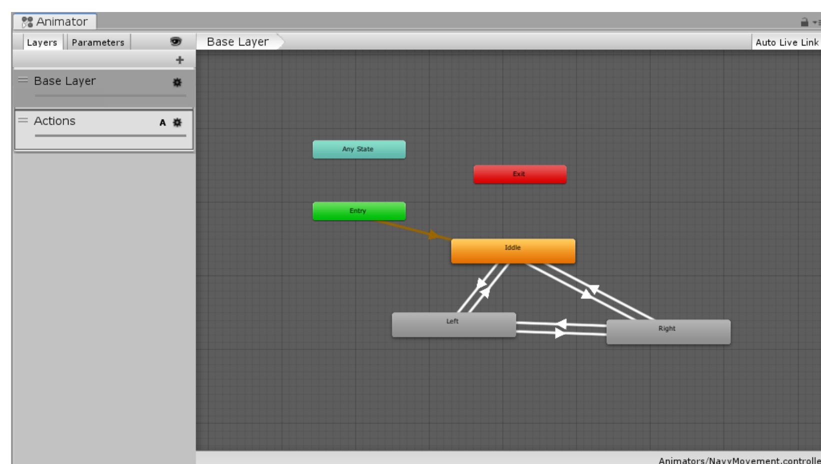


**Figura 4.2:** Ventana Animator.

- *Animation*: permite crear y editar Animation Clips, en estos se pueden variar características de los GameObjects con el paso del tiempo.

A parte de las ventanas comentadas anteriormente, disponemos de muchas otras que no se describirán, ya que son muchas y muy específicas y alargarían mucho este apartado.

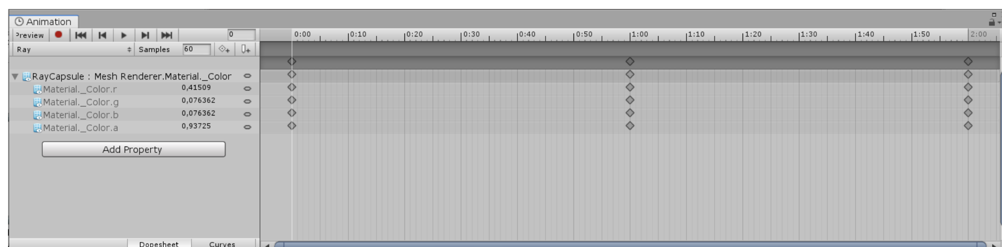


**Figura 4.3:** Ventana Animation.

### 4.1.2. Conceptos de Unity

#### GameObject, componentes y escenas

Cada objeto del videojuego es un GameObject: los personajes, la cámara, las luces, los efectos especiales, etc. A partir de ahora el término objeto se utilizará en ocasiones para referirse a GameObject. Un GameObject es un contenedor de componentes, el único componente que debe tener obligatoriamente un GameObject es el componente Transform, el cual gestiona la posición, la rotación y la escala del objeto, todos los demás componentes son opcionales, dependiendo del comportamiento y aspecto que se desee lograr se añadirán distintos componentes. Unity proporciona muchos componentes predefinidos que serán muy útiles para facilitar el desarrollo, también los *scripts* se consideran componentes de los GameObjects, así que cualquier cosa que haya que programar se hará desde *scripts* que irán adjuntos a un GameObject.

Algunos de los componentes predefinidos más importantes son:

- Rigidbody: permite que el objeto actúe bajo el control del motor de físicas. Este componente se puede ver en la figura 4.4.
- Collider: define la forma de un objeto que será tenida en cuenta para calcular las colisiones físicas. Existen muchos distintos tipos de Colliders, en la figura 4.5 se pueden observar los diferentes parámetros que se pueden configurar en un componente de tipo Box Collider.
- AudioSource: este componente hace posible que un objeto emita sonido.

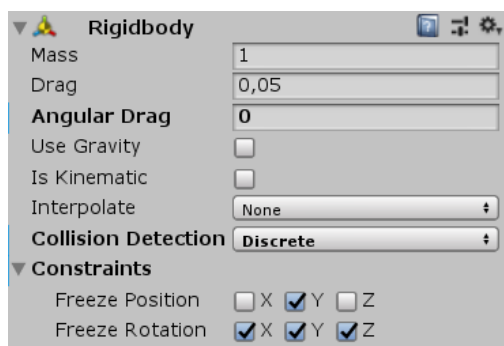


**Figura 4.4:** Propiedades del componente Rigidbody.

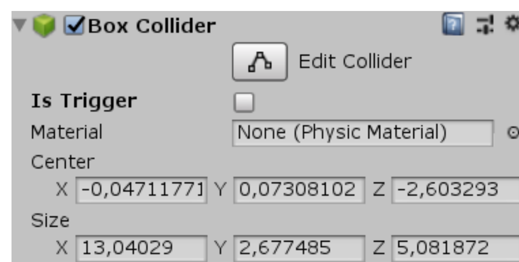


**Figura 4.5:** Propiedades del componente Box Collider.

En la figura 4.6 se puede ver como se compone una escena en Unity.

**Figura 4.6:** Relación entre escena, GameObject y componente.

**Prefabs**

Los Prefabs son como clases visuales, se puede crear un GameObject en la escena con sus respectivos componentes, valores e hijos y después arrastrarlo al Project View. Al hacer eso se tiene ese objeto guardado y se podrán crear el número de instancias que se desee de este desde el editor o desde código. Si se edita el Prefab original, todos los cambios se aplicarán automáticamente a cualquier instancia de este.

***Scripts***

Todos los *scripts* que se vayan a agregar a algún objeto deben heredar de MonoBehavior. Esta clase proporciona muchas funciones que se utilizarán durante el desarrollo. Al crear un *script* este tiene predefinidos dos funciones, Start y Update —que si no se necesitan se pueden borrar sin problema—. Start se ejecuta solo una única vez cuando se crea el objeto, mientras que Update se ejecuta en cada fotograma. Unity tiene muchas funciones de eventos que se ejecutan en un determinado orden, el cual es importante conocer para poder controlar correctamente el comportamiento de los objetos. En las figuras 4.7 y 4.8 se puede observar cómo es el orden en el que se ejecutan las funciones de Unity.

### 4.1.3. GIMP

GIMP son las siglas en inglés de GNU Image Manipulation Program. Es un programa de edición de imágenes digitales en forma de mapa de bits. Es gratuito y es posible modificar su código fuente. Se va a usar en el desarrollo del videojuego para modificar algunas texturas o crear imágenes necesarias para el juego.

### 4.1.4. Visual Studio

Es un entorno de desarrollo integrado —IDE— que está disponible para Windows, Linux y macOS. Es compatible con múltiples lenguajes de programación, tales como C++, C#, Visual Basic .NET, F#, etc. Es el IDE que usa Unity para la edición y la creación de los *scripts*.

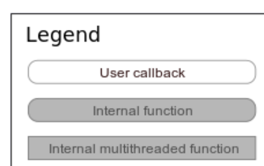


**Figura 4.7:** Leyenda del diagrama de flujo.

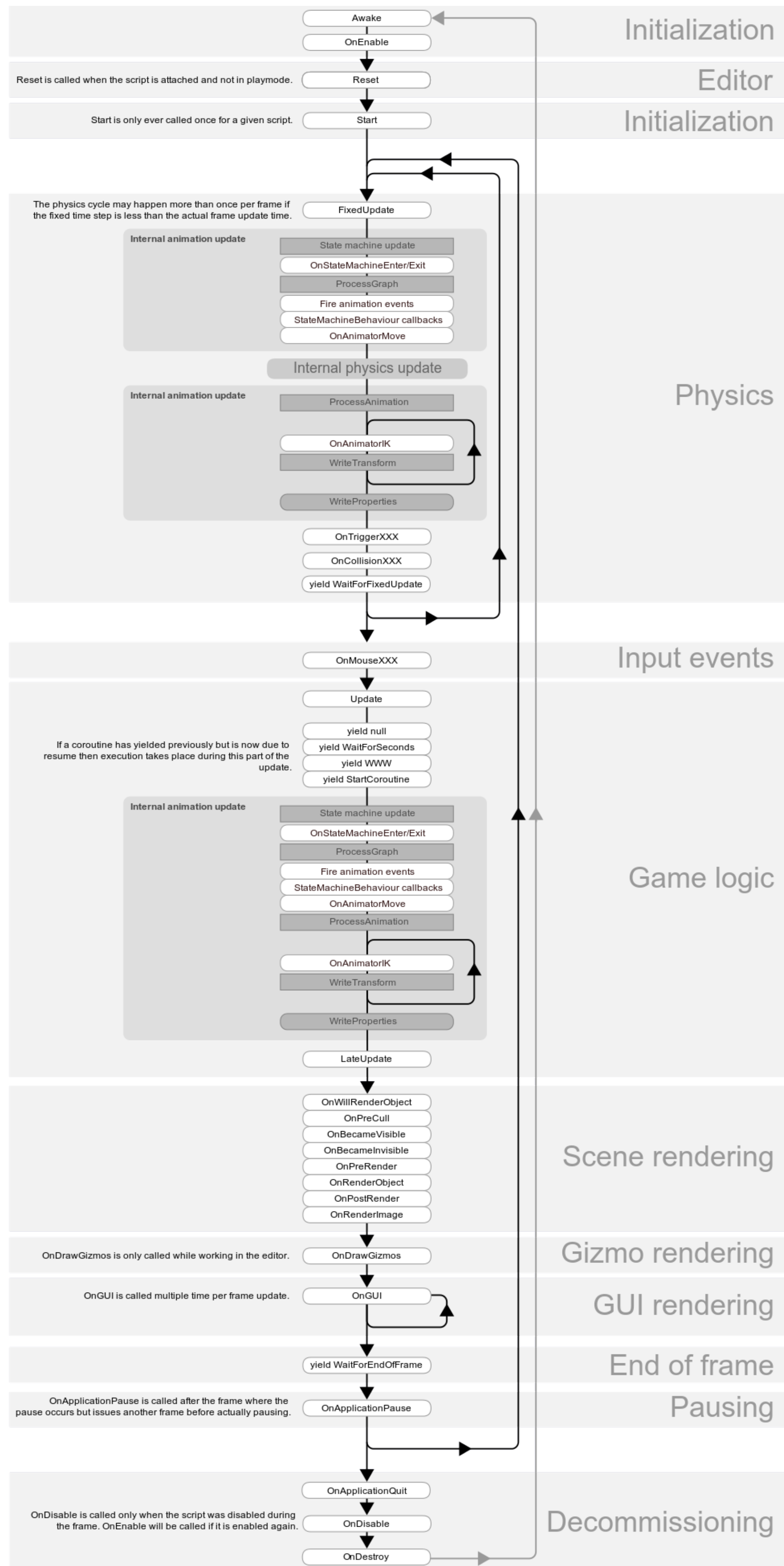


**Figura 4.8:** Diagrama de flujo del ciclo de vida del *script*.

## 4.2 Diseño detallado de la solución

### 4.2.1. Personajes

El jugador controlará una nave con la cual tendrá que ir destruyendo a los drones enemigos. El modelo de la nave ha sido descargado de la Asset Store y se le añadirán diferentes componentes para lograr el comportamiento y aspecto deseado. Algunos de estos componentes serán añadidos a GameObjects hijos para tener una mejor organización, ya que son necesarios gran cantidad de componentes. La nave tendrá efectos de partículas, algunas animaciones, emitirá sonidos, etc. En la figura 4.9 se puede ver el modelo 3D original. La nave tendrá dos posibles acciones —aparte de desplazarse por toda la pantalla—. Estas son disparar y activar una onda, la cual se expande desde el centro de la nave hasta una determinada distancia. La onda daña a los enemigos si impacta con ellos y destruye algunos proyectiles enemigos. Después de usar la onda hay que esperar un determinado tiempo antes de poder volver a usarla.

**Figura 4.9:** Nave que controlará el jugador.

En el juego habrá 4 tipos de enemigos, cada uno tendrá diferentes características y comportamientos. Todos ellos aparecerán fuera del rango de la cámara y entrarán en este desplazándose hasta un punto concreto de la zona de juego. En la figura 4.10 están los modelos de dichos enemigos y los nombres con los cuales me referiré a ellos posteriormente.

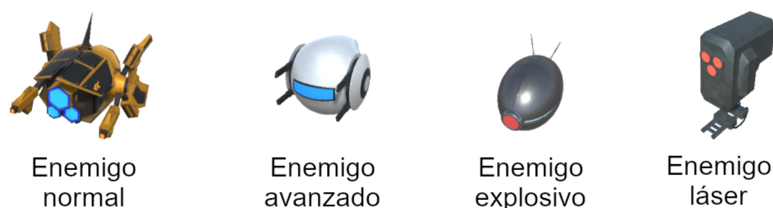


**Figura 4.10:** Drones enemigos.

- Enemigo normal: es el enemigo más básico y el más fácil de derrotar. Sus disparos son esferas de color naranja, las cuales podrán ser destruidas por el jugador si les dispara o si activa la onda y están en el rango de esta. Este dron podrá estar quieto o moverse entre dos puntos datos.

- Enemigo avanzado: muy similar al anterior, pero con más puntos de vida y sus disparos son esferas de color negro que no pueden ser destruidas por el jugador.

- Enemigo explosivo: este dron persigue al jugador hasta alcanzarle o hasta que sea destruido por este. Una esfera invisible definirá el área en el cual si el jugador entra, este dron se destruye a sí mismo quitando un gran número de puntos de vida al jugador con la explosión.

- Enemigo láser: este enemigo siempre se mueve entre dos puntos dentro de la zona de juego, y tras cierto tiempo lanza un rayo, este rayo quita vida al jugador mientras este se encuentre dentro del área del rayo.

Por último queda presentar al jefe final del juego, para el cual se usará el modelo de la figura 4.11. Este tendrá varios comportamientos diferentes que se irán turnando, estos serán controlados mediante una máquina de estados, esto será mejor explicado en el capítulo de desarrollo.

**Figura 4.11:** Modelo 3D del jefe final del juego.

### 4.2.2. Interfaz de usuario

La interfaz de usuario debe ser sencilla e intuitiva. Al iniciar el videojuego aparecerá un menú principal desde el cual se podrá elegir el modo de juego y se podrán configurar algunos aspectos del videojuego. Esta interfaz cumplirá todos los requisitos expuestos en el capítulo anterior. Mientras se esté jugando, en todo momento se podrá pausar la partida y en pantalla aparecerá una pequeña ventana que permitirá reanudar la partida, acceder a la ventana de opciones y volver al menú principal. La pulsación de la tecla Enter se interpretará como aceptar/acceder, esto es muy usual en los videojuegos actuales.

En los niveles de combate, se ubicará en la esquina superior izquierda una barra para mostrar la vida del jugador. Debajo de esta barra habrá un icono que estará visible cuando la onda que lanza la nave esté lista para usarse, si no puede lanzarse todavía donde debería estar el icono habrá una cuenta atrás con el tiempo que queda hasta que vuelva a estar disponible esta acción. Al completar un nivel, aparecerá texto en pantalla para informar y felicitar al jugador, mientras que si pierde, que ocurre cuando la vida del jugador llega a 0, se mostrará texto que indique la derrota.

### 4.2.3. Organización de las clases

Muchas de las clases —*scripts*— que se usarán no tendrán dependencias interesantes para representar en un diagrama de clases UML, pero sí que es relevante destacar la dependencia que existe entre algunas de ellas. Se ha creado una clase padre para todos los enemigos, y otra clase padre para las clases que se encargan de gestionar los niveles. En la figura 4.12 se pueden observar estas relaciones.

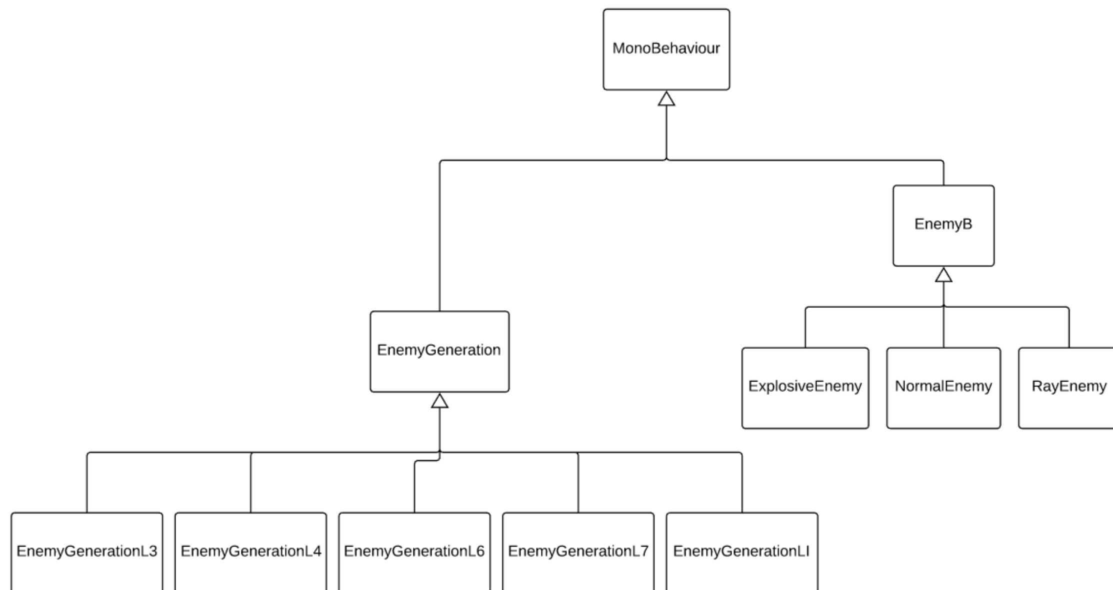


**Figura 4.12:** Diagrama de clases UML.

En el diagrama de clases no se han representado los atributos y los métodos de las clases debido a que son muchos. Hacer una clase padre para todos los tipos de enemigos es bastante útil, ya que casi todos los atributos son comunes a todos los enemigos y las funciones que permiten que un enemigo se desplace también son las mismas en todos los casos. Creando clases padre se evita repetir código y se permite una mejor escalabilidad en el caso de que en un futuro se implemente un tipo de enemigo nuevo. La clase NormalEnemy es la encargada de controlar al enemigo normal y al enemigo avanzado, dado que su comportamiento es el mismo no hace falta implementar clases distintas, lo que cambia entre estos enemigos es el aspecto, los puntos de vida que tienen y el tipo de disparo. Las clases ExplosiveEnemy y RayEnemy son las que irán asociadas al enemigo explosivo y al enemigo láser respectivamente.

Respecto a las clases de generación de niveles, en la clase padre —EnemyGeneration— se declaran todas las variables necesarias para la generación de los enemigos y las funciones que permiten crear instancias de los Prefabs de los enemigos en el juego. En esta clase se implementará el método Start heredado de MonoBehaviour. En él, entre otras cosas, se calculará una matriz, la cual contiene coordenadas en las que es posible ubicar enemigos. Hay que tener en cuenta que si volvemos a implementar la función Start en alguna clase hija, esta se superpondrá a la implementación de Start de EnemyGeneration y no tendremos la matriz con las posiciones calculada, algo que puede tener resultados catastróficos, así que cualquier cosa que se desee implementar en el Start tendrá que ponerse en la clase EnemyGeneration.

Las clases EnemyGenerationL3, EnemyGenerationL4, EnemyGenerationL6 y EnemyGenerationL7 serán las responsables de gestionar la generación de enemigos en los ni-

veles 3, 4, 6 y 7 respectivamente. Cuando el jugador destruya todos los drones que hay en pantalla las clases anteriores lo detectarán y generarán los enemigos correspondientes a la siguiente oleada, usando las funciones que crean instancias de enemigos ubicados en la clase padre EnemyGeneration. La clase EnemyGenerationLI será la encargada de gestionar la generación de enemigos en el modo infinito, esta tiene más complejidad dado que la generación será aleatoria y hay que verificar que la posición donde se quiere ubicar un enemigo no esté ocupada, también hay que aumentar el ritmo de generación de enemigos con el paso del tiempo.

### 4.2.4. Sistema de mensajería

En muchas ocasiones, al estar programando un objeto del videojuego, se hace necesario tener referencias de otros objetos, por ejemplo, cuando queremos que un enemigo se desplace hacia la dirección del jugador hay que disponer de la posición del jugador. Hay varias maneras de pasar referencias, se puede hacer declarando una variable pública dentro del *script* del enemigo —así, cuando añadimos el *script* a un objeto en el inspector podemos darle valor a esa variable— y arrastrar el objeto del que necesitamos la referencia desde la escena al inspector. También se puede hacer únicamente mediante código, usando el método Find de la clase GameObject, el cual toma un argumento de tipo *string*, que es el nombre del objeto buscado, y devuelve el GameObject si lo encuentra, o en caso contrario devuelve *null*. También hay más maneras de conseguir referencias de los GameObject que componen la escena, pero en algunos casos es mucho más fácil y recomendable usar un sistema de mensajería. Por ejemplo, si queremos detectar cuando un enemigo ha sido destruido en la clase que gestiona el nivel, es más conveniente que el enemigo envíe un mensaje a la clase gestora que comprobar constantemente si el enemigo sigue existiendo en la escena. También este sistema será muy útil para controlar la interfaz de usuario.

Además, usar un sistema de mensajería disminuye el acoplamiento entre los diferentes objetos permitiendo una mejor escalabilidad, ya que si se dispone de un gran número de clases que tienen dependencias entre sí al cambiar alguna de ellas es probable que haya que volver a asignar las referencias, debido a que ha cambiado algún nombre, hemos modificado alguna variable, etc.

Por esas razones se usará un sistema de mensajería, concretamente se usará el patrón de publicación-subscripción. En este hay una serie de mensajes predefinidos, los *scripts* interesados en ciertos mensajes pueden suscribirse a través del servidor de mensajes, llamado *broker*, y cuando algún *script* genere un mensaje de ese tipo y lo mande al *broker*, este último lo redistribuirá a todos los suscritos a ese mensaje. En la figura 4.13 está representado el funcionamiento de este patrón.

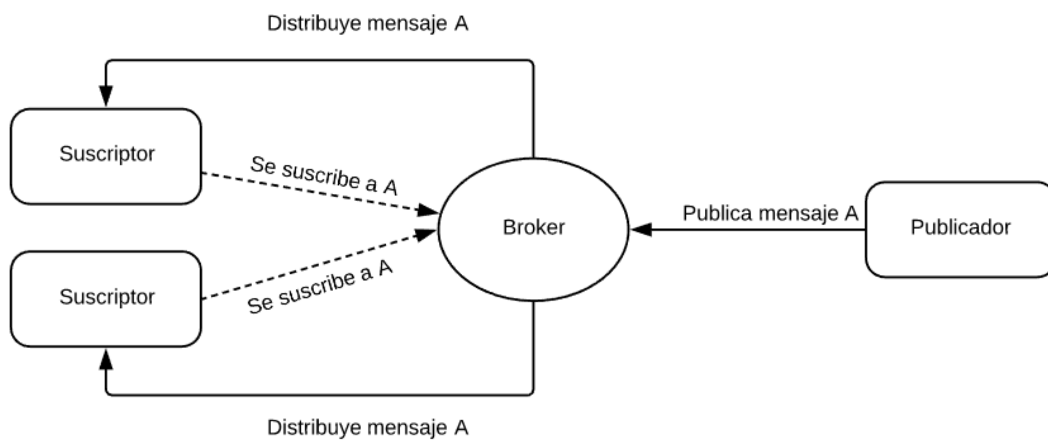


**Figura 4.13:** Esquema del patrón publicación-subscripción.

# CAPÍTULO 5
# Desarrollo

En este capítulo se explicará cómo se ha desarrollado el videojuego. Se verán las partes de programación que han resultado más complejas y que son interesantes de exponer. Se ha seguido un desarrollo incremental, primero se han implementado las partes más básicas y posteriormente se han ido añadiendo nuevos elementos, o se han mejorado los existentes. Durante el desarrollo se ha consultado en repetidas ocasiones la API de Unity [8] y la API de C# [9].

## 5.1 Creación de la nave del jugador

Lo primero que se ha desarrollado es que la nave pueda ser controlada por el jugador. Para ello se ha creado un *script* llamado PlayerInput.cs, esta clase será la encargada de gestionar toda entrada que esté relacionada con las acciones de la nave. Para los controles se ha usado la clase Input de UnityEngine, la cual proporciona diferentes funciones para detectar cuando se ha pulsado una tecla o para detectar el movimiento del ratón, entre otras cosas. Esta clase tiene funciones tales como GetButton, en las cuales se puede pasar un botón virtual como argumento, estos botones virtuales se pueden asignar a teclas o botones de un mando desde la pestaña Project Settings. Se hará uso estas funciones para detectar la entrada, dado que permiten gestionar la entrada de diferentes dispositivos con el mismo código. En cada fotograma se ejecutan las funciones encargadas de detectar la entrada, por consiguiente, las llamadas a las funciones se hacen en la función Update. Tras detectar la entrada se crea un vector que representa con qué velocidad se debe mover la nave en cada eje y se asigna este vector a la variable velocity del Rigidbody de la nave, lo que hace esto es asignar una velocidad al objeto que contiene el Rigidbody —recordar que Rigidbody es un componente que hace posible gestionar las físicas del objeto—.

Cuando la nave se está moviendo hacia la derecha o hacia la izquierda esta se inclina hacia esa misma dirección, esto se controla mediante un Animator y código. En el código se comprueba si el jugador está moviendo la nave en el eje horizontal —eje x— y si el movimiento llega a un determinado umbral, se establece la variable booleana correspondiente a la dirección del movimiento a verdadero. Estas variables que se establecerán por código son las que controlan las transiciones del Animator. Cada estado del Animator tiene un Animation Clip el cual reproduce la animación que corresponde. El Animator puede tener varias capas, se usarán 2 capas en este caso, una que controle la inclinación de la nave y otra que se encargará de controlar la onda que lanza la nave. En la figura 5.1 se puede observar la máquina de estados que controla la inclinación y en la figura 5.2 se puede ver la máquina de estados encargada de la onda.

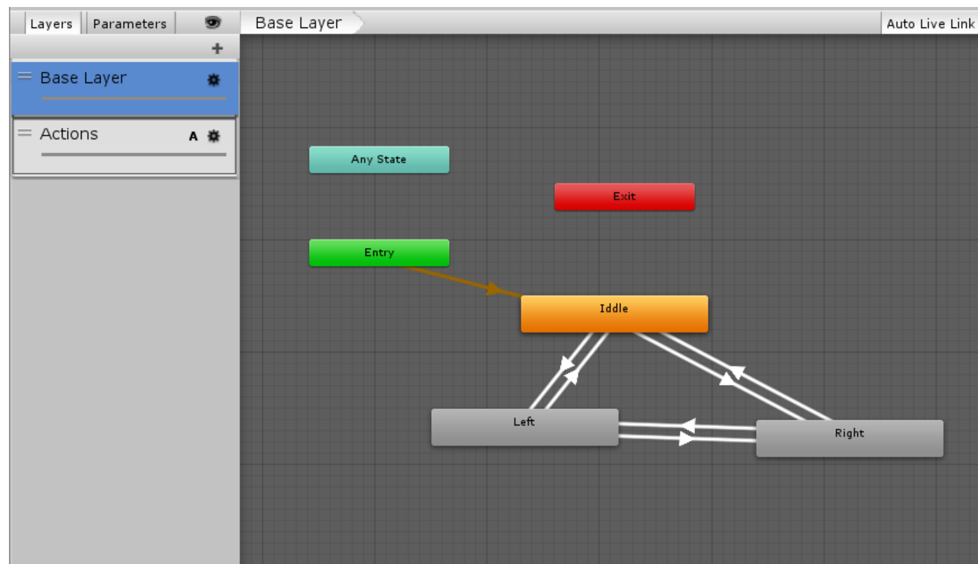


**Figura 5.1:** Capa del Animator que controla la inclinación.

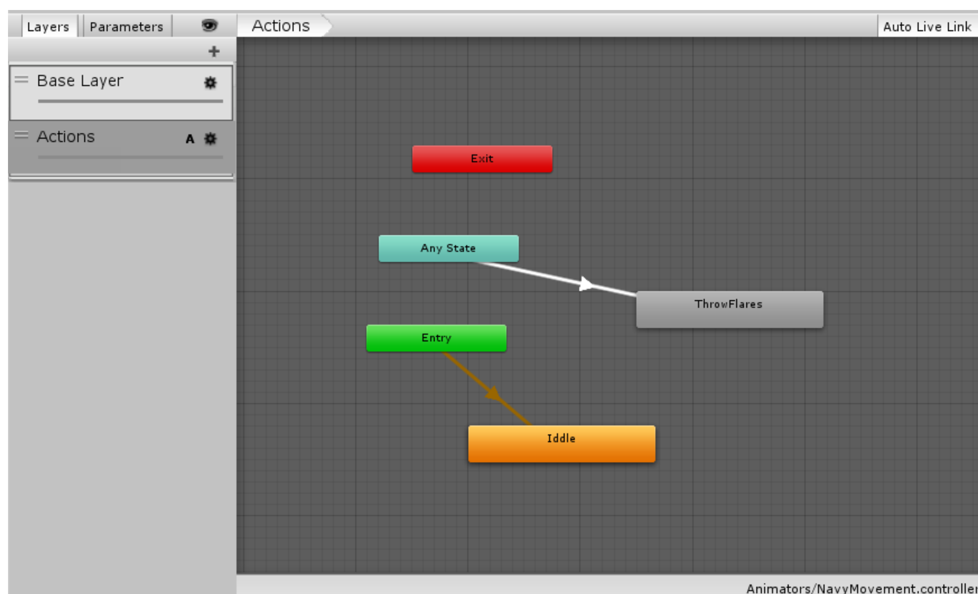


**Figura 5.2:** Capa del Animator que controla la onda.

Las teclas elegidas para controlar la nave en los niveles de combate son: W, A, S, D para el movimiento, botón izquierdo del ratón —se usa el botón virtual Fire1— para disparar y botón derecho del ratón —se usa el botón virtual Fire2— para lanzar la onda.

Al Prefab original descargado de la nave se le han añadido diversos componentes, ver figura 5.3. El componente Audio Source es el que permite reproducir sonidos desde la posición de la nave, la reproducción de estos sonidos es controlada por los *scripts* adjuntos al objeto. En el *script* PlayerCharacter.cs se lleva la cuenta de los puntos de vida del jugador y se detecta cuando recibe un disparo enemigo. Si la vida llega a 0 se ejecuta una función que gestiona la destrucción de la nave. Lo que hace dicha función es reproducir un sonido de explosión, activar un sistema de partículas, el cual crea el efecto de una explosión, desactivar algunos de los GameObjects hijos y lanzar 2 corrutinas. El término corrutina se refiere a una función que es capaz de pausar su ejecución y devolver el control a Unity para luego continuar donde lo dejó en el siguiente fotograma. En la figura 5.4 se puede observar cómo se invoca una corrutina en Unity y como se declara esta en C#.

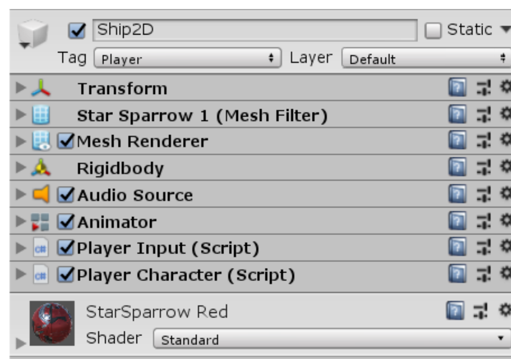


**Figura 5.3:** Componentes que forman el GameObject de la nave del jugador.

Toda función declarada con un tipo de retorno IEnumerator debe contener una instrucción de retorno yield incluida en algún lugar de su cuerpo. La línea con la instrucción de retorno yield es el punto en el cual la ejecución se pausará y se reanudará desde ahí en el siguiente fotograma. Usando la función WaitForSeconds y pasándole como argumento un *float*, que representa segundos, se puede suspender la función hasta que pase dicho número de segundos en vez de suspenderla hasta el siguiente fotograma —ver figura 5.4—. Para lanzar una corrutina en ejecución se usa la función StartCoroutine. Las corrutinas se usarán en repetidas ocasiones durante el desarrollo del videojuego.

```
// Función que inicia la explosión y destruye la nave
private void Die()
{
    _audioSource.PlayOneShot(clipExplosion);
    _rigidbody.velocity = new Vector3 (0,0,0);
    _playerInput.enabled = false;
    _particleExplosion.Play();
    foreach (Transform child in transform)
    {
        if(child.gameObject.name != "BigExplosion")
        {
            child.gameObject.SetActive(false);
        }
    }
    // Inicio de las 2 coroutines
    StartCoroutine(DisableMesh());
    StartCoroutine(Destroy());
}

// Función con tipo de retorno IEnumerator la cual suspende su ejecución
// durante 2 segundos y después destruye la nave y lanza el menú de reiniciar
IEnumerator Destroy()
{
    yield return new WaitForSeconds(2);
    Destroy(this.gameObject);
    restartObj.gameObject.SetActive(true);
}

// Después de 0.2 segundos se deshabilita la geometía de la nave
IEnumerator DisableMesh()
{
    yield return new WaitForSeconds(0.2f);
    _meshRender.enabled = false;
}
```

**Figura 5.4:** Código comentado de la clase PlayerCharacter.

En la figura 5.5 se pueden observar los diferentes GameObjects que tiene como hijos la nave. Los objetos que empiezan por «Afterburner» son sistemas de partículas que simulan los reactores de la nave. ShotSpawn es un objeto vacío que se usa para marcar la posición en la cual aparecerán los disparos cuando se pulse el botón de disparar. NewWa-

ve es el sistema de partículas que se activa cuando se lanza la onda y Wave es un objeto que contiene un componente Sphere Collider, el cual se usa para marcar el rango de la onda y destruir o infligir daño a los objetos que se encuentren dentro de esta esfera. Este Collider está marcado como *trigger* —palabra que se puede traducir como disparador—, lo que significa que no se usará para calcular colisiones con otros objetos sino como una zona de activación, se puede detectar si en la zona que define este objeto entra otro con funciones de MonoBehaviour. BigExplosion es un sistema de partículas que se encarga de simular el efecto de una explosión y el objeto Collider contiene varios objetos que son Box Colliders, los cuales definen la geometría de la nave para el motor de físicas.

Los sistemas de partículas anteriormente mencionados, a excepción del de la onda, han sido obtenidos de la Asset Store y han sido modificados en Unity para amoldarlos a las necesidades del proyecto. El sistema de partículas de la onda ha sido creado de cero, usando el editor de partículas de Unity y GIMP, debido a la imposibilidad de encontrar nada adecuado en la Asset Store.

**Figura 5.5:** Jerarquía de GameObject de la nave del jugador.

## 5.2 Desarrollo del nivel del tutorial

El primer paso ha sido crear un plano, el cual hará la función de fondo, para este nivel se ha elegido que el fondo simule un océano. Esto se ha llevado a cabo utilizando un material configurado adecuadamente —los materiales son un componente de tipo Mesh los cuales definen el aspecto de un objeto—, para conseguir el efecto deseado ha sido necesario añadir varios *normal maps* al material. Los *normal maps* son un tipo especial de textura que permite agregar detalles a las superficies de un modelo tales como golpes, bultos, surcos, rayones.

Tras esto se ha posicionado la cámara a una distancia adecuada —es importante recordar que en este tipo de niveles se ha usado una cámara ortográfica—, y se han definido unos límites en los cuales se puede mover el jugador. El jugador se puede mover en todo el espacio que se ve en cámara siempre que la nave sea del todo visible. Para limitar el espacio en el que puede desplazarse la nave primero se optó por posicionar GameObjects que tuvieran como componentes Colliders, un símil adecuado para este caso sería que se intentaron colocar muros invisibles, pero esta implementación no fue satisfactoria, dado que al colisionar con las paredes la nave se movía más lento de lo que se movería sin estar colisionando con ellas. Entonces se desarrolló otra forma de llevar a cabo esto, la cual consiste en posicionar Colliders configurados como zonas de activación y detectar en el *script* PlayerInput.cs cuando se entra en estas zonas, y cuando esto ocurra prohibir el movimiento hacia la dirección del Collider.

Para lograr el efecto de que la nave se está moviendo hacia delante se ha puesto un Animator al plano, el cual va desplazando las texturas y las *normal maps* hacia atrás indefinidamente. Así parece que la nave esté avanzando hacia delante todo el tiempo. Todo lo anteriormente comentado en este apartado se usa en todos los niveles de combate, pero

en algunos niveles se cambia el material del plano. Concretamente en los niveles 2, 3 y 4 se usa el fondo que representa un océano y en los niveles 6, 7 y 9 se usa un material diferente. Este otro material da el aspecto de desierto al plano.

El nivel del tutorial trata de seguir unas indicaciones que van apareciendo en un recuadro ubicado en la esquina superior derecha, ver figura 5.6. Estas indicaciones dan a conocer al jugador los controles y las mecánicas del juego. Para detectar cuando el jugador ha realizado las acciones que se le piden se ha usado un sistema de mensajería como el expuesto en 4.2.4. Para poder usar este sistema en Unity se ha utilizado una librería externa que implementa un sistema de paso de mensajes como el descrito [10]. Para ello se han creado dos *scripts*: Messenger.cs, el cual contiene el código que implementa el sistema de paso de mensajes y otro llamado GameEvent.cs donde se definen los tipos de eventos que puede generar el juego.

**Figura 5.6:** Captura del nivel del tutorial.

En la clase PlayerInput.cs se ha declarado una variable booleana la cual se controla desde el inspector de Unity —si declaras una variable como pública o si añades «[SerializeField]» delante de su declaración a esta se le pueden dar valores desde la ventana inspector de Unity—. Si esta variable tiene el valor verdadero asignado en el *script* se ejecutará código al detectar que el usuario ha probado los controles que se le están mostrando en pantalla, básicamente lo que hace el código es enviar un mensaje para que la clase que gestiona el tutorial sepa que se ha llevado a cabo la acción y avance a la siguiente fase del tutorial. Después de mostrarle todos los controles al usuario llega la última parte del tutorial que consiste en derrotar a 2 enemigos normales. Todos los niveles tienen un GameObject sin geometría el cual se usa para poner los *scripts* gestores en escena y que de esta forma se puedan ejecutar. Se llama *scripts* gestores a los *scripts* que no definen el comportamiento de un GameObject sino que se usan para gestionar el avance del nivel o alguna parte del juego como puede ser la interfaz gráfica.

## 5.3 Desarrollo de los enemigos

En esta sección se comentará muy brevemente como han sido creados los Prefabs de los enemigos, ya que es bastante similar a como se ha creado la nave que controla el

jugador. Donde se profundizará más es en el algoritmo implementado para que no se atraviesen ni se choquen unos con otros.

### 5.3.1. Implementación de los drones

El enemigo normal y el avanzado tienen los mismos componentes, solo cambia el valor de algunas variables, tales como la variable de los puntos de vida, la que establece la velocidad de movimiento, etc. También cambia el Prefab del que crean una instancia cuando disparan. El disparo del enemigo avanzado no se puede destruir, mientras que el del enemigo normal sí, a la hora de implementarlo esto significa que el GameObject que representa el disparo del enemigo normal tiene que controlar si entra en la zona de activación de la onda enemiga o del disparo enemigo, mientras que el disparo del enemigo avanzado no.

El enemigo normal y el avanzado aparecen en una posición fuera de la pantalla y avanzan hacia otra posición dentro de la zona de juego, esta posición se les asigna al crearlos, en los niveles normales se elige de antemano mientras que en el nivel infinito se elige aleatoriamente. También se les puede asignar una segunda posición al crearlos, en este caso el comportamiento que tendrán será distinto, irán desplazándose de un punto al otro mientras no sean destruidos. Además, todos los drones, menos el láser, siempre están mirando al jugador, esto significa que cuando el jugador se mueve estos enemigos rotan. Esto se consigue con la función LookAt que proporciona Unity, la cual se invoca en el Update de los enemigos. Para controlar el disparo se usa una variable a la que se le asigna un valor que representa segundos, y en cada fotograma se le resta el tiempo pasado desde el último fotograma, lo que equivale a una cuenta atrás. Cuando llega a 0 el enemigo dispara hacia la posición donde se encuentra el jugador y empieza la cuenta de nuevo.

El enemigo explosivo también aparece fuera del rango de la cámara y desde ahí se dirige a la posición del jugador, en la función Update de su *script* se actualiza la posición del jugador, lo que significa que este enemigo necesita una referencia a la nave aliada. Tras probar la implementación de este se ha notado que no queda muy claro dónde está el límite de su explosión y ,por consiguiente, tampoco se sabe el rango en el cual puede hacer daño. A causa de esto, se ha añadido una circunferencia dibujada alrededor del dron la cual marca el rango en el cual no se debe entrar, ver figura 5.7.

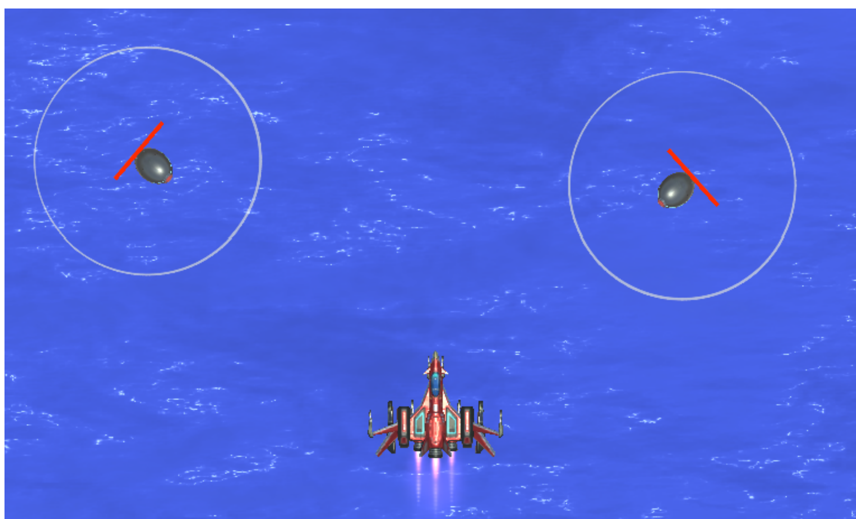

**Figura 5.7:** Drones explosivos.

Por último, se ha desarrollado el dron láser, en este han hecho falta más GameObject hijos y más componentes que en los demás, ya que tiene varios sistemas de partículas y una capsula que representa el rayo que lanza. Este enemigo tiene un tiempo establecido tras el cual lanza el rayo durante unos segundos, después de eso lo apaga y empieza a contar de nuevo hasta cumplir el tiempo de espera establecido. Estos tiempos se controlan mediante variables las cuales por comodidad se han declarado para que puedan ser editadas desde el inspector. Se hace uso de un sistema de partículas para avisar al jugador que el enemigo va a lanzar el rayo en breves, y se usa otro sistema de partículas junto a la capsula anteriormente mencionada para dar el aspecto al rayo, ver figura 5.8.

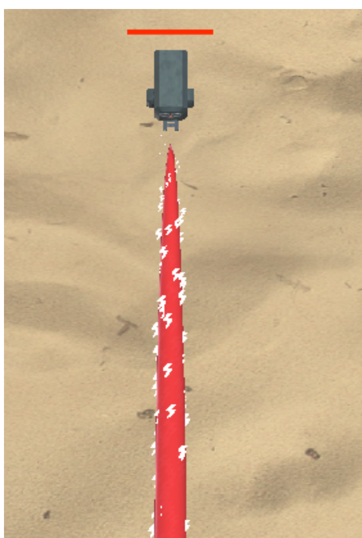

**Figura 5.8:** Dron que lanza rayos.

Todos los drones tienen un Canvas —que es un GameObject con un componente Canvas en él, del que todos los elementos UI deben ser hijos— que tiene como hijo una barra de vida, esta se puede ver en las figuras 5.8 y 5.7. Cuando impacta un disparo del jugador en los drones esta barra se actualiza. También todos los enemigos tienen un sistema de partículas que es una explosión, la cual se activa cuando la vida del enemigo llega a 0. El enemigo explosivo tiene una explosión diferente a los demás porque realiza daño con ella y se ha considerado adecuado que fuese distinta para diferenciarla fácilmente. Al activarse la explosión se reproduce un efecto de sonido que es el sonido de una explosión.

### 5.3.2. Algoritmo de desplazamiento

En la primera implementación de los enemigos, para llegar de un punto a otro, estos simplemente usaban la siguiente función de Unity:

public static Vector3 MoveTowards(Vector3 current, Vector3 target, float maxDistanceDelta);

Esta función calcula un punto entre la posición current y target a una distancia maxDistanceDelta de current. Esto permite mover un objeto de la posición actual a otra posición destino a una velocidad determinada, para que la velocidad del objeto sea independiente de los fotogramas por segundo es necesario multiplicar la velocidad por Time.deltaTime, que son los segundos transcurridos desde el último fotograma. Por ejemplo, si se da el valor 20 * Time.deltaTime a la variable maxDistanceDelta esto hace que la velocidad sea de 20 unidades por segundo.

Al principio se usó únicamente esta función para mover a los drones enemigos, pero cuando dos drones coincidían en el mismo punto había 2 posibilidades, o hacer que no ocurriese nada y se atravesasen o que se detectara una colisión entre ellos y chocaran, pero ninguna de las dos opciones funcionaba bien en la práctica. Si se atravesaban el juego transmitía una sensación de baja calidad, que dos objetos se atraviesen no queda bien, y si colisionaban en ocasiones se bloqueaban la ruta uno al otro y no avanzaban. Así que se optó por usar un algoritmo que detectara si tienen otro dron en su ruta, y si es el caso, rodearlo.

Este algoritmo consiste en lanzar dos rayos, uno girado 7 grados respecto al vector que define la dirección de avance y otro girado -7 grados respecto al mismo vector, esto da mejores resultados que trazando únicamente un rayo en la dirección de avance. Pero un rayo tiene un volumen muy pequeño y resulta mucho más fiable crear una esfera alrededor del rayo y comprobar si esa esfera ha impactado con algún dron. En la figura 5.9 se han dibujado mediante *gizmos* —que es una funcionalidad que proporciona Unity para *debugging* visual— los rayos y una esfera al final de estos, en el algoritmo la esfera pasará por toda la longitud del rayo hasta llegar a la posición donde está ubicada en la imagen.

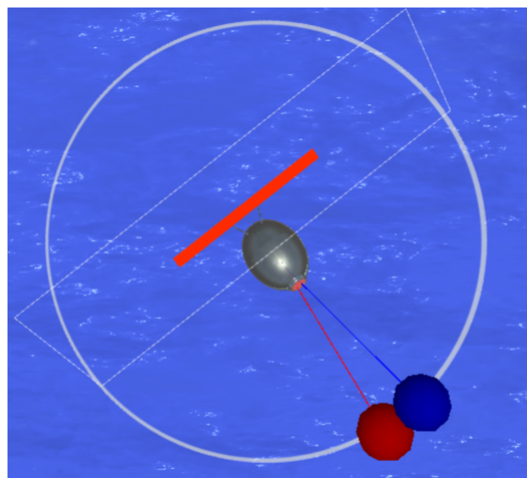

**Figura 5.9:** Dron explosivo con *gizmos*.

Si al hacer lo anterior, se detecta que ha habido un impacto con un dron se llama a la función FindPath para desviar al dron y evitar el choque. Pero las esferas que se generan pueden impactar con cualquier cosa, cuando se detecta un impacto el objeto impactado no necesariamente tiene que ser un dron. Eso se soluciona pasando a la función de crear las esferas un parámetro que es una máscara de capas. En Unity se pueden definir diferentes capas y asignar los GameObjects a estas capas, esto es muy útil en diferentes situaciones. En la figura 5.10 se pueden ver las capas definidas en el proyecto y como se asignan a un GameObject, el dron avanzado en este caso. Todos los enemigos tienen que estar asignados a la capa Enemies para que el algoritmo de desplazamiento funcione.

La función que se usa para efectuar el desplazamiento de los drones se llama MoveToPosition, y en la figura 5.11 está escrita en pseudocódigo una simplificación de la función desarrollada que permite entender el método usado para desplazar a un dron. En la práctica esta función es más compleja, dado que hay que tener en cuenta algunos detalles más.

La función FindPath se encarga de buscar un punto de destino nuevo para el dron en caso de que en su camino haya un obstáculo. Esta función hace lo siguiente: primero se elige hacia qué dirección girar el vector que representa la dirección de avance, si el rayo que ha golpeado es el que está girado hacia la derecha respecto a la dirección de avance

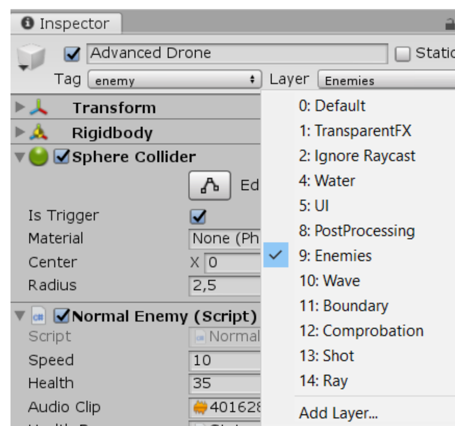


**Figura 5.10:** Selección de capa en un GameObject.

```
MoveToPosition()
        IF (Distancia(miPosición, destinoActual) < margen && destinoActual != destinoFinal)
                destinoActual = destinoFinal;
        IF (Distancia(miPosición, destinoActual) < margen && destinoActual == destinoFinal)
                FIN;
        ELSE
                direcciónRayo = Normalizar(destinoActual - miPosición);
                direcciónRayo1 = Girar(direcciónRayo, 7°);
                direcciónRayo2 = Girar(direcciónRayo, -7°);
                rayo1 = CrearRayo(miPosición, direcciónRayo1);
                rayo2 = CrearRayo(miPosición, direcciónRayo2);
                boolImpacto1 = EmitirEsfera(rayo1, radio, hitInfo, distancia, máscaraDeCapas);
                boolImpacto2 = EmitirEsfera(rayo2, radio, hitInfo, distancia, máscaraDeCapas);
                IF (boolImpacto1 | boolImpacto2)
                        FindPath();
                ELSE
                        miPosicion = MoveTowards(miPosición, destinoActual);
```

**Figura 5.11:** Pseudocódigo de la función usada para mover a un dron.

el dron girará hacia la izquierda, en cambio si el rayo que ha golpeado es el girado a la izquierda el dron girará a la derecha. Si los 2 rayos han golpeado a un objetivo se elige aleatoriamente la dirección de giro. El giro es de 30 grados respecto al vector director que representa la dirección de avance. Para obtener el nuevo destino se multiplica el vector director unitario que representa la dirección de avance actual por un número x y se suma al vector resultado la posición actual, obteniendo así un punto en la dirección del vector girado a una distancia x del dron. Ese punto será el nuevo destino. Al tener un nuevo destino hay que comprobar de nuevo si hay obstáculos cercanos en el trayecto. Esto se hace igual que en la función anteriormente explicada —MoveToPosition—. Si se da el caso de que vuelve a haber un dron en el camino, se vuelve a realizar todo lo anterior con un ángulo de giro diferente. En vez de que el ángulo de giro que se aplica sobre el vector de la dirección original sea de 30 grados será de 45 —en cada iteración se le suman 15 grados al ángulo de giro—. El número de iteraciones está limitado a 8, así si el dron se encuentra rodeado de enemigos no estará indefinidamente buscando un camino que no existe, en este caso se quedará quieto hasta el siguiente fotograma.

## 5.4 Generación de enemigos

En esta sección se explica como se ha implementado la generación de enemigos en los niveles.

### 5.4.1. Matriz para la generación de enemigos

Para ubicar los enemigos en pantalla es necesario trabajar con coordenadas. Hay una estructura en Unity llamada Vector3 que sirve para representar vectores o puntos en 3 dimensiones. Una primera idea fue crear objetos vacíos —solo con el componente Trasnform— y posicionarlos en la escena para usarlos como posibles ubicaciones de los drones. Esta práctica funciona bien si los puntos a marcar son pocos, pero como los enemigos deberían poder ubicarse en muchos puntos distintos esto no sería una buena manera de hacerlo.

La manera de obtener las coordenadas ha sido definir un espacio en el cual se pueden ubicar enemigos y en ese espacio trazar líneas horizontales y verticales. Cada intersección entre una linea horizontal y otra vertical es una posible posición para ubicar un dron. En la figura 5.12 el rectángulo rojo representa la zona definida para ubicar enemigos y con las líneas negras se obtienen los puntos para ubicar enemigos. Tener en cuenta que en la práctica la zona será más amplia y el número de líneas será superior al del ejemplo.

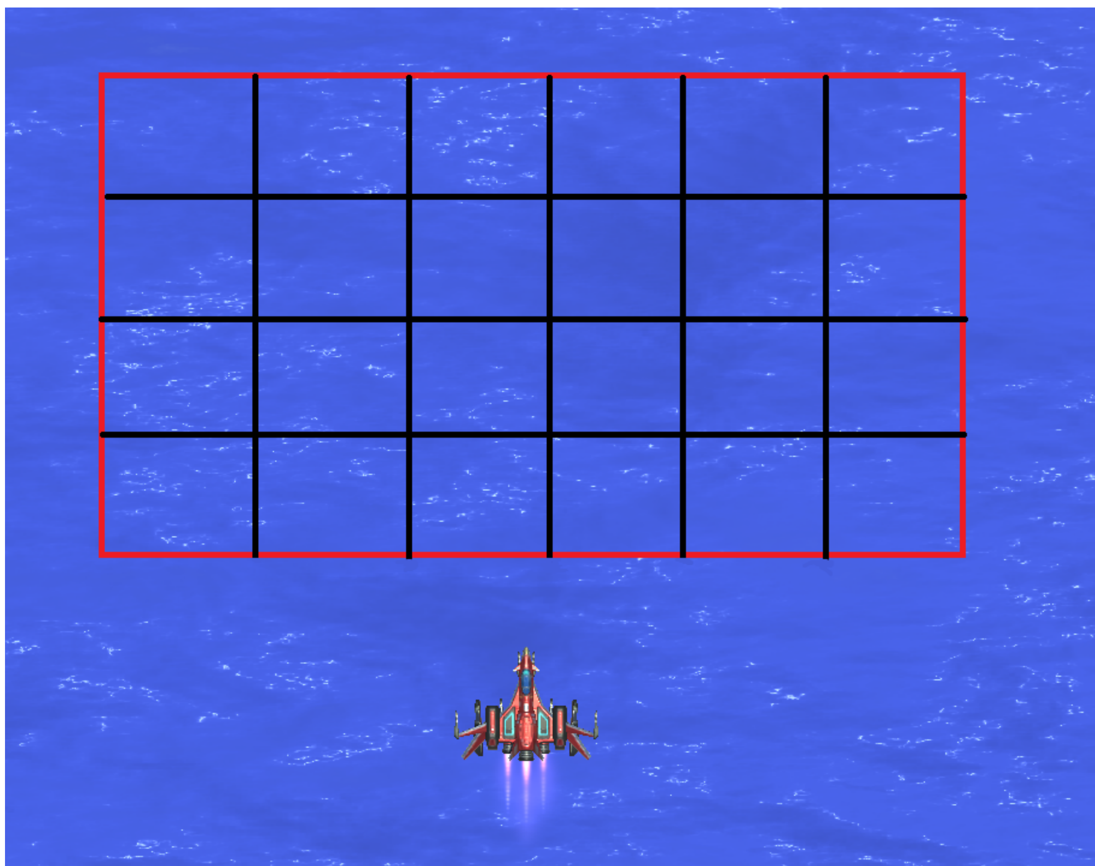

**Figura 5.12:** Ejemplo de como se obtienen las coordenadas para ubicar enemigos.

Para el cálculo de las intersecciones de estas líneas primero se definen los vértices del rectángulo. Dado que en algunos niveles puede ser interesante variar la zona en la que están los enemigos, se permitirá dar valor a las variables que definen la zona desde el inspector de Unity. Hay 2 variables que se usan para establecer el número de líneas

horizontales y verticales, a estas también se les da valor desde el inspector. Al disponer de estos datos se puede calcular la separación que debe haber entre las líneas para formar una rejilla en la cual la distancia entre todas las líneas sea la misma. Se calculan las posiciones en las que se intersecan las líneas y se almacenan en una matriz. Con esto se obtiene una matriz con todas las posiciones disponibles para ubicar enemigos. Pero esto no es suficiente debido a que necesitamos controlar en que posiciones ya hay un objeto. En cada posición se puede ubicar un enemigo o el objeto el cual al ser recogido restablece una cantidad de la vida del jugador. Controlar las coordenadas ocupadas es especialmente importante para el nivel infinito porque en este la generación de enemigos es aleatoria, mientras que en los niveles de la campaña únicamente es aleatoria la ubicación del objeto reparador.

Para controlar las posiciones ocupadas, se ha pensado en disponer de una matriz de booleanos de las mismas dimensiones que la matriz de coordenadas, en la cual el valor de una posición sea verdadero si está ocupada la posición y falso si no lo está. Pero esto no es muy eficiente y no es adecuado para la generación aleatoria debido a que si elegimos una posición aleatoriamente y resulta que está ocupada, tenemos que volver a generar números aleatorios y repetir el proceso. Y en el caso de que tengamos la matriz casi llena, este proceso tiene un coste computacional muy alto. El método que se ha elegido para controlar las posiciones ocupadas es usar una lista de cadenas —List<string>freePositions—, la cual contiene todas las posiciones de la matriz que contienen coordenadas libres. Las posiciones libres en esta lista se representan como una cadena formada por el número de la fila de la matriz, una coma —que se usa para separar— y el número de la columna. Por ejemplo para representar la posición de la matriz fila 2 y columna 3 se usaría la cadena «2,3».

Para elegir las coordenadas aleatoriamente se calcula un número aleatorio que esté entre 0 y la longitud de la lista. Después se extrae —coste $O(1)$— y elimina —coste $O(n)$— el elemento que ocupa esa posición en la lista. Al objeto a crear se le pasan las coordenadas que están almacenadas en la posición de la matriz que indica el elemento extraído de la lista. Cuando el objeto es destruido envía un mensaje con las posiciones de la matriz en las que están sus coordenadas al gestor. Así este puede añadir esa posición a la lista para que esté disponible de nuevo. Esto requiere que todo objeto almacene la posición de la matriz que ocupan sus coordenadas para que al ser destruido pueda informar al *script* gestor.

Los enemigos que se mueven entre dos posiciones suponen una dificultad, se pueden marcar como ocupadas las dos posiciones entre las que se desplazan, pero en su camino habría posiciones que se supone que están libres pero no deberían estarlo, ya que por ahí pasa el dron. Este problema es muy fácil de evitar en los niveles de la campaña porque los drones se ubican manualmente y basta con no ubicar drones en el trayecto, y si por casualidad toca ubicarlos en una posición en la cual hay un objeto reparador este se destruye —se ha programado que si un enemigo colisiona con un objeto reparador este último se destruya—. Pero en el nivel infinito la situación es totalmente distinta, ya que todo es aleatorio. Debido a ello se ha establecido la restricción de que los drones se muevan solo entre posiciones contiguas de la matriz. Cuando se elige la primera posición la segunda se restringe a las celdas de la matriz contiguas, y así se evita que un dron se interponga en el camino de otro.

### 5.4.2. Funciones para la generación de enemigos

Se han creado varias funciones que permiten crear instancias de los Prefabs de los drones en la clase EnemyGeneration. Para el enemigo normal y avanzado se usan las mismas funciones, ya que estos usan la misma clase y las variables que se necesita que se les pase

valor al crearlos son las mismas. Se han programado dos funciones para la creación de estos drones debido a que tienen dos modos de funcionamiento y en cada caso necesitan ser configurados de diferente manera. Ambas funciones se llaman igual, pero tienen un número de parámetros diferente, se hace uso de la sobrecarga de funciones. La función que crea un dron que no se mueve recibe una coordenada x y otra coordenada z —los drones y el jugador se mueven solo por el plano xz, la altura nunca varía— que son las coordenadas a las que llega y se queda inmóvil disparando. Mientras que la función que crea drones que se mueven de un punto a otro constantemente recibe dos coordenadas x y dos coordenadas z.

En cuanto al dron explosivo y al dron láser, cada uno tiene una función distinta que resuelve todas las dependencias que se generan al crear la instancia de estos. Estas funciones permiten seleccionar desde qué lado va a entrar el enemigo al campo de batalla. El enemigo normal y el avanzado siempre aparecen por arriba. En la figura 5.13 se puede ver la declaración de las funciones de creación de enemigos.

```
protected void CreateEnemy(GameObject navyModel, int row, int col)...

protected void CreateEnemy(GameObject navyModel, int row, int col, int rowDest, int colDest)...

protected void CreateExpEnemy(float pos, string side)...

protected void CreateRayEnemy(int row, int col, int rowDest, int colDest, string side)...
```

**Figura 5.13:** Cabecera de las funciones de creación de drones.

## 5.5 Nivel infinito

La principal diferencia de este nivel con los niveles de combate de la campaña se encuentra en el *script* que gestiona el nivel. Nada más empieza la partida se pausa el juego y aparece en pantalla una ventana la cual indica cuantos puntos se consiguen por derrotar cada tipo de enemigo, ver figura 5.14. A continuación, se crea una oleada de enemigos normales y se establece en una variable —timeUntilNextEnemy— un número que representa segundos, a esta variable se le irá restando el tiempo transcurrido desde el último fotograma, y cuando llegue a 0 se creará una instancia de un enemigo. Para elegir qué tipo de enemigo se crea se genera un número aleatorio entre 0 y 1, y con varias instrucciones *if* y *else* se comprueba en qué intervalo está ese número. Dependiendo en qué intervalo se encuentre se genera el tipo de enemigo establecido para ese intervalo. Por ejemplo, si el número generado aleatoriamente está entre 0 y 0,05 creamos un enemigo explosivo, lo que quiere decir que la probabilidad de que al crear un enemigo este sea explosivo es del 5 %. En la tabla 5.1 se puede ver la probabilidad de crear cada tipo de enemigo.

| Tipo de enemigo | Probabilidad |
|---|---|
| Enemigo normal inmóvil | 50 % |
| Enemigo normal móvil | 10 % |
| Enemigo avanzado inmóvil | 20 % |
| Enemigo avanzado móvil | 10 % |
| Enemigo explosivo | 5 % |
| Enemigo láser | 5 % |

**Tabla 5.1:** Probabilidades.

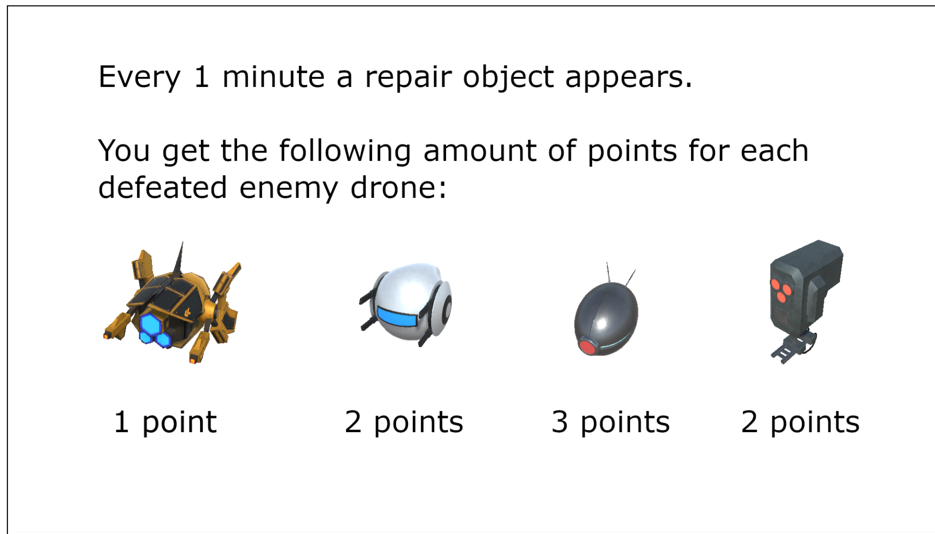


**Figura 5.14:** Ventana del nivel infinito.

Cuando la variable timeUntilNextEnemy, que se usa para la cuenta atrás, llega a 0 se crea un enemigo y se asigna un nuevo valor a esa variable que será el tiempo hasta la creación del siguiente enemigo. Ese tiempo se calcula con la siguiente función:

$$f(x) = 10/ln(x)$$

En la función anterior $x$ es el tiempo transcurrido hasta el momento y $f(x)$ es el tiempo que se esperará hasta crear al siguiente enemigo. Ambos valores representan segundos. Ha sido elegida esta función porque proporciona tiempos adecuados para la generación, en la tabla 5.2 se puede ver el valor de la función para diferentes valores de $x$. Mientras que en la figura 5.15 se puede observar la gráfica de esta función para valores de $x$ mayores que 1. El *script* calcula $f(x)$ por primer vez cuando $x = 5$, la siguiente vez que se calcula $f(x)$ será cuando hayan transcurrido $f(5)$ segundos —en ese instante es cuando se creará un enemigo—, lo que significa que en ese momento $x \simeq 5 + f(5)$.

| x | f(x) |
|---|---|
| 5 | 6.21 |
| 10 | 4.34 |
| 30 | 2.94 |
| 60 | 2.44 |
| 120 | 2.09 |
| 240 | 1.82 |

**Tabla 5.2:** Tabla de valores.

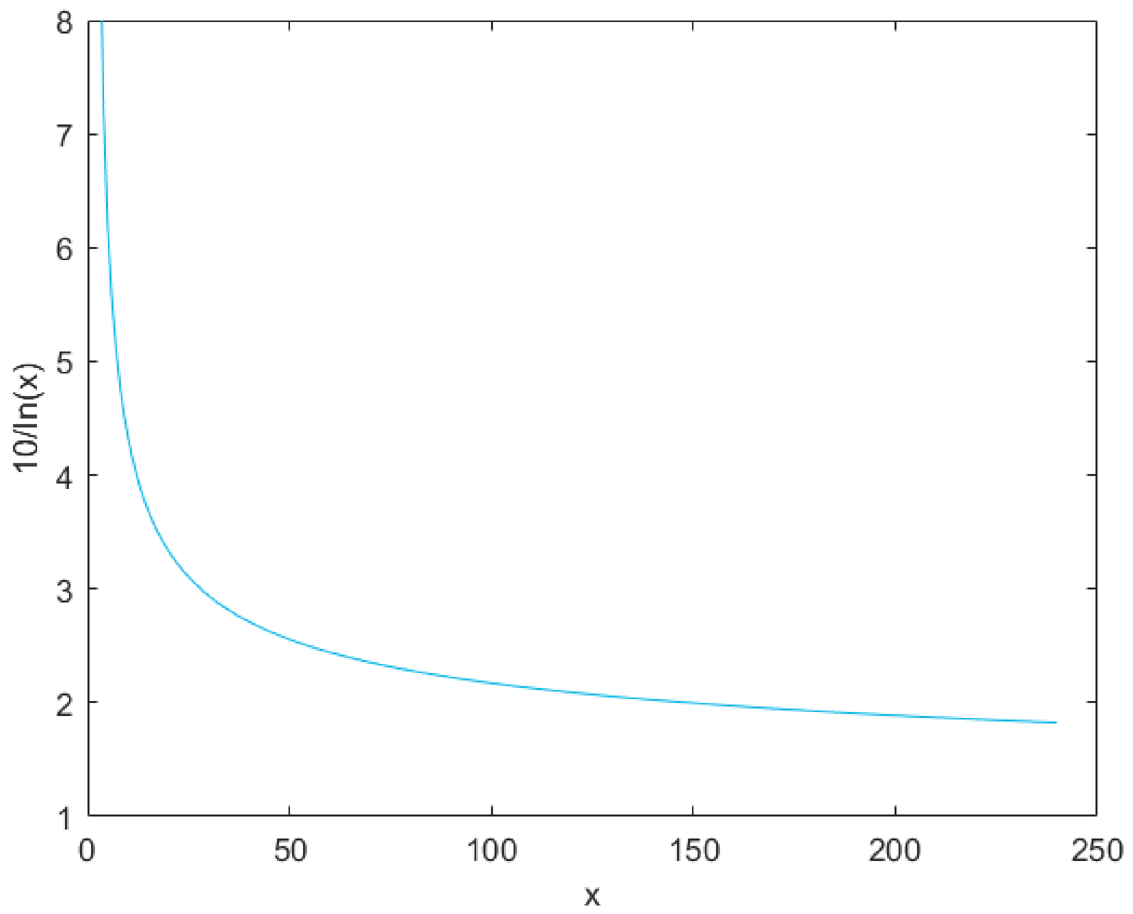


**Figura 5.15:** Gráfica de la función.

## 5.6 Niveles 3D

En estos niveles la nave se adentra en un túnel lleno de obstáculos y el jugador tiene que esquivarlos. La cámara sigue a la nave en todo momento, manteniéndose a la misma distancia atrás de la nave. Esto se implementa con un *script* que ha sido añadido como un componente de la cámara. En el primer nivel de este tipo se le comunica al jugador mediante un recuadro, igual que en el nivel tutorial, que se desplace usando el ratón. El desplazamiento que puede controlar el jugador es el de los ejes x e y, que se corresponde a mover la nave arriba y abajo —eje y— y a derecha e izquierda —eje x— mientras esta avanza hacia delante —eje z— a una velocidad constante.

La nave que controla el jugador es la misma que la de los niveles de combate, lo que se ha cambiado es el *script* que va adjunto a la nave, ya que en este escenario se requieren acciones totalmente diferentes, además, se han tenido que modificar algunos parámetros del Rigidbody para que cuando la nave se estrelle contra un obstáculo sea más realista el movimiento de esta. El movimiento del ratón se detecta con Input.GetAxis("Mouse X") y Input.GetAxis("Mouse Y"), que devuelven un valor entre -1 y 1. Para mover la nave se cambia la velocidad de su Rigidbody. La velocidad se obtiene multiplicando los valores obtenidos con las instrucciones anteriores por una variable que representa la velocidad y por un valor que es la sensibilidad del ratón —esta se puede configurar desde el menú de opciones—. Es necesario limitar el movimiento de la nave dentro de los límites de la pantalla. Para ello lo que se ha hecho es definir 4 variables: xMax, xMin, yMax e yMin y controlar con instrucciones *if* en la función Update que la posición de la nave no sea menor que las mínimas —xMin e yMin— o mayor que las máximas —xMax e yMax—, si se incumple alguna de estas restricciones, no se permite que la nave se mueva hacia la dirección en la que se ha llegado al límite, limitando la velocidad a 0 hacia esa dirección. Aun así, la nave puede salir de los límites ligeramente porque cuando se hace la comprobación es posible que ya esté más allá del límite, en ese caso se cambia mediante código la posición de la nave para devolverla dentro de los límites, esto no se nota a la hora de jugar porque la nave suele salirse muy poco de los límites establecidos.

En cuanto a la escena, se ha construido un túnel utilizando cubos —que se pueden crear y modificar desde Unity—, a estos cubos se les ha asignado un material para darles color o texturas en algunos casos. El techo del túnel tiene asignado un material especial el cual es transparente y da el aspecto de cristal a esta superficie. Los objetos que hay que esquivar son también cubos, están posicionados dentro del túnel y los hay de diferentes tamaños y colores. En la figura 5.16 se puede ver una captura de uno de estos niveles.

Para el fondo de la escena se han usado diversos Skyboxes —que son como envolturas, lo que se ve de fondo—. Estos han sido obtenidos de la Asset Store y se ha intentado que se adapten bien al nivel, por ejemplo, el penúltimo nivel es un nivel de esquivar y se le ha establecido un Skybox más oscuro con tonos rojizos para así crear un ambiente que transmita peligro, ya que el siguiente nivel es el del jefe final. También se han incorporado nuevos sonidos en estos niveles: la música de fondo es distinta a la de los niveles de combate y se ha añadido un nuevo efecto de sonido que es el ruido del motor de la nave.

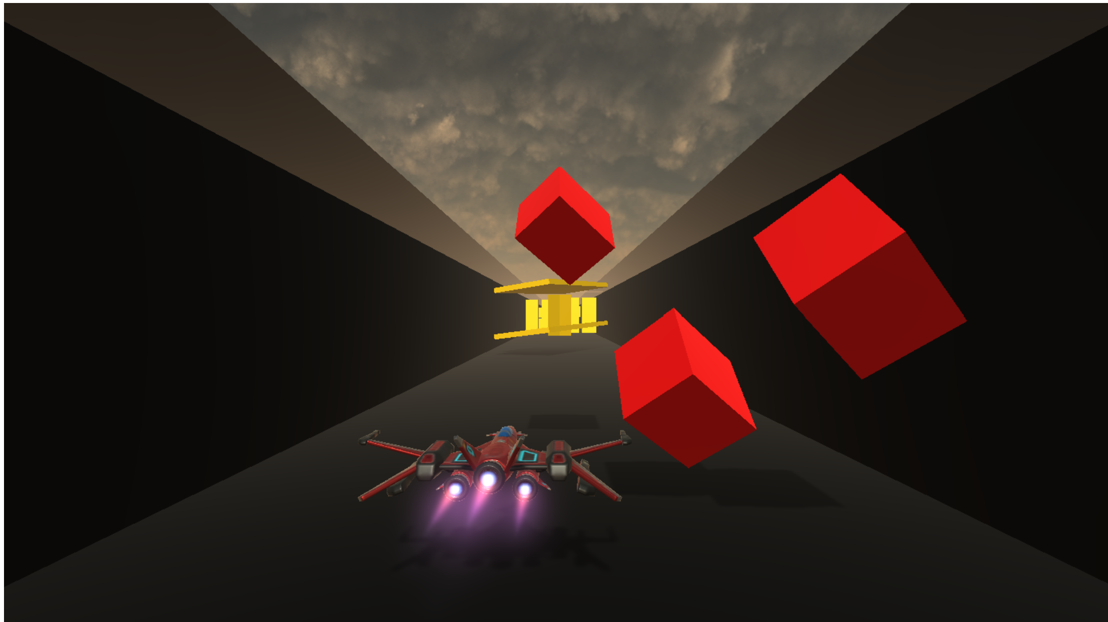

**Figura 5.16:** Captura de pantalla de uno de los niveles 3D.

## 5.7 Jefe Final

En esta sección se explicará cómo ha sido implementado el enemigo final de la campaña. Al empezar el nivel este enemigo está fuera del rango de la cámara y se va desplazando hasta llegar al centro de la pantalla, entonces empiezan sus ataques. El jefe final se puede encontrar en 4 estados diferentes que definen su comportamiento:

- Estado Entry: el enemigo se desplaza desde fuera del campo de batalla hasta una cierta posición dentro de este. Este estado solo se da una vez y es cuando empieza el nivel, no hay forma de volver a este desde otro estado. Cuando el jefe final llega a la posición establecida se pasa a otro estado.
- Estado MoveAndShootCircles: en este estado el enemigo se mueve aleatoriamente dentro de una zona de la pantalla preestablecida y dispara esferas de color naranja —las mismas que el enemigo normal— que forman una circunferencia.
- Estado MoveAndShootBrust: cuando el jefe final se encuentra en este estado se mueve dentro de una zona del escenario de juego, esta zona se ubica más arriba que la zona del estado anterior porque es más adecuado para el tipo de disparo. El enemigo dispara a la vez tres esferas naranjas que avanzan hacia la posición del jugador, esto se repite cada 0,5 segundos.
- Estado DeployDrones: el jefe final se queda quieto en la posición en la que se encuentra al entra en este estado durante unos segundos, y cerca de él aparecen dos drones explosivos.

Estos estados han sido añadidos a una máquina de estados, usando la ventana Animator. En la figura 5.17 se puede observar esta y las posibles transiciones entre los estados.

Para controlar cuando se pasa de un estado a otro se usan variables de tipo *trigger*, las cuales se activan desde código. Todos los estados, menos Entry, tienen establecida una determinada duración. Los estados MoveAndShootCircles y MoveAndShootBrust se van turnando hasta que se cumple la cuenta atrás del estado DeployDrones. Cuando

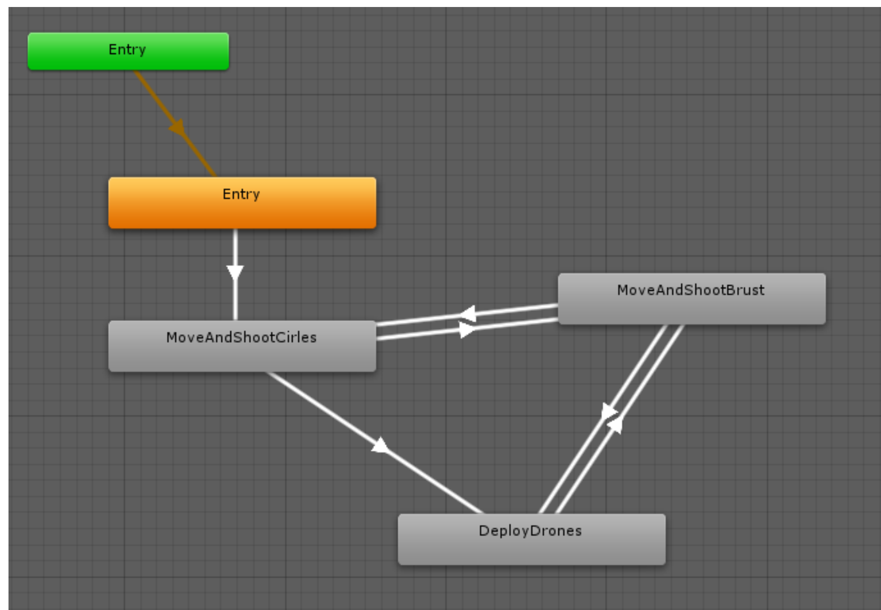


**Figura 5.17:** Máquina de estados del jefe final.

esta cuenta atrás llega a 0 y haya acabado la duración del estado actual el próximo estado al que se transitará será DeployDrones. Como se puede observar en la máquina de estados de DeployDrones siempre se pasa a MoveAndShootBrust, y esto es debido a que si la nave empezara a disparar esferas que forman una circunferencia y hay dos drones explosivos persiguiendo al jugador sería muy difícil sobrevivir.

Un aspecto importante de esta implementación es el lugar donde se encuentra el código que controla el comportamiento de la nave. Cada estado tiene un *script* asociado que hereda de la clase StateMachineBehaviour, esta proporciona varias funciones muy útiles como: OnStateEnter, que se ejecuta cuando se entra en el estado, OnStateUpdate, que se ejecuta en cada fotograma en el que el estado está activo menos en el primer y en el último fotograma, y OnStateExit, que se ejecuta en el último fotograma del estado. Haciendo uso de estas funciones se controla el comportamiento de la nave en cada estado. Desde estos *scripts* se tiene fácil acceso a las componentes de la nave, así que tenemos una forma fácil de controlar la nave desde la máquina de estados.

Es interesante comentar como se crea la circunferencia de esferas, para ello se hace uso de algunos conceptos de trigonometría. Si tenemos una circunferencia unitaria como la de la figura 5.18 un punto de la circunferencia se puede calcular fácilmente: la coordenada x es el coseno de alfa y la coordenada z es el seno de alfa. Sabiendo esto solo queda adaptarlo a la situación que se presenta en el juego.

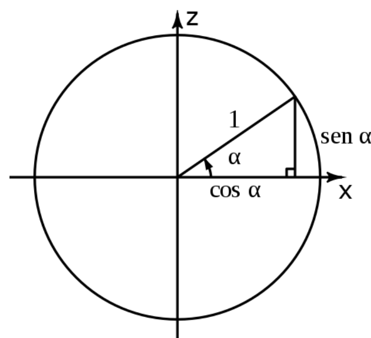


**Figura 5.18:** Circunferencia unitaria.

Primero hay que tener en cuenta que la circunferencia que creará el jefe no será de radio 1, por consiguiente, habrá que multiplicar por la hipotenusa obteniendo:

$$x = h \cdot \cos \alpha$$

$$z = h \cdot \sin \alpha$$

Tampoco el centro de la circunferencia que creará el enemigo es el origen de coordenadas, así que habrá que sumar a las coordenadas anteriores el desplazamiento correspondiente en cada eje. Obteniendo así:

$$x = h \cdot \cos \alpha + x_e$$

$$z = h \cdot \sin \alpha + z_e$$

Donde $x_e$ y $z_e$ son las coordenadas x y z del enemigo.

Las dos fórmulas anteriores se ubican dentro de un bucle, el cual en cada iteración varía el valor de $\alpha$ obteniendo varios puntos de la circunferencia. En el juego se ha elegido construir la circunferencia con 15 esferas, así que se calculan 15 puntos del espacio y se crean esferas en ellos. El bucle empieza con $\alpha = 0°$ y termina con $\alpha = 336°$ —en cada iteración se incrementa $\alpha$ en 24° dado que 360/15 = 24—.

Las esferas se irán distanciando del centro de la circunferencia con el paso del tiempo, aumentando así el radio de esta. Para implementar esto, al crear las esferas en la función Start del *sccript* que tienen asignado las esferas se obtiene un vector director unitario que define la dirección en la que debe desplazarse cada esfera. Este vector se obtiene restando las coordenadas del origen de la circunferencia —que es la posición de la nave del jefe final— las coordenadas de la posición de la esfera y normalizando el vector resultado. Después en el Update del *script* de las esferas se programa que se muevan en la dirección calculada una cierta distancia que viene definida por una variable que es la velocidad de movimiento. El resultado se puede observar en la figura 5.19.

**Figura 5.19:** El jefe final disparando esferas en forma de circunferencia.

La creación periódica de los disparos, en los 2 modos de disparo, se lleva a cabo mediante corrutinas. En estas corrutinas hay un bucle dentro del cual se crean instancias de los disparos y al final del bucle se encuentra la instrucción «yield return new WaitForSeconds(0.5f);» —es importante recordar que las corrutinas son funciones con tipo de

retorno IEnumerator, lo que quiere decir que son iteradores—. Esto lo que hace es ejecutar todo el cuerpo del bucle y cuando llega al *yield* se suspende la ejecución durante 0,5 segundos. Y de esta manera se consigue crear disparos periódicamente eficientemente.

## 5.8 UI

El menú principal está formado por una imagen de fondo representativa del juego y 4 botones: Continue, New Campaign, Infinity Battle y Options. No hay un botón para salir dado que se está desarrollando para WebGL y para salir basta con cerrar la pestaña del navegador. En la figura 5.20 se puede ver el aspecto del menú y en la figura 5.21 la ventana de opciones. En la ventana de opciones se encuentran 3 *slidders*, con el primero se controla el volumen de la música de fondo y con el segundo el volumen de los efectos de sonido, ambos están predefinidos para que estén al máximo. El último *slidder* permite controlar la sensibilidad del ratón, muy útil para los niveles 3D, ya que en estos la velocidad de movimiento depende de la sensibilidad que tenga el ratón de cada usuario.

Cuando se produce un cambio de escena, ya sea desde el menú principal a un modo de juego o al avanzar de nivel, se reproduce una animación la cual es un efecto de transición. Este consiste en que se superpone una imagen con cierta transparencia a lo que hay en pantalla y se va disminuyendo la transparencia de la imagen de transición hasta que es solo visible esta, y a partir de ese momento se vuelve a subir la transparencia de la imagen hasta que desaparece y en pantalla queda visible la nueva escena. Este efecto tienen una duración de 2 segundos.

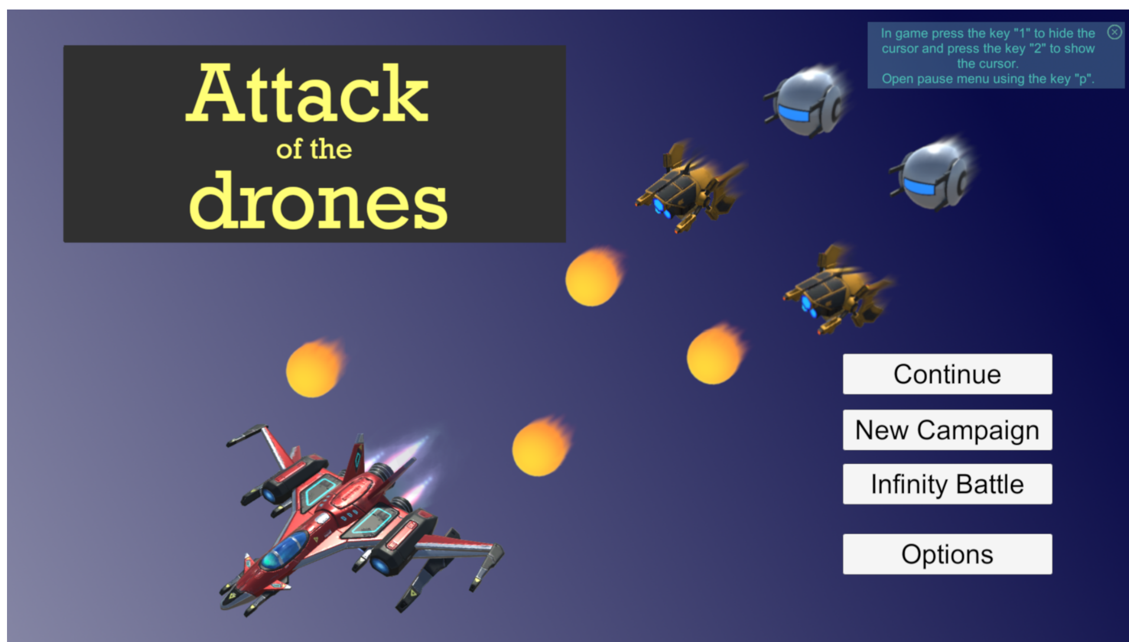


**Figura 5.20:** Menú principal.

En los niveles de combate la barra de vida del jugador se encuentra en la esquina superior izquierda y abajo de esta está el indicador de la onda. Además, cada enemigo tiene una barra de vida, esto se puede ver en la imagen 5.19. Durante cualquier nivel si se pulsa la tecla p se activa un GameObject del Canvas de la escena el cual implementa el menú de pausa —se ha creado un Prefab del Canvas para poder reutilizarlo fácilmente en los distintos niveles—. Lo más adecuado sería que la tecla de pausa fuese Escape, como suele ser en los juegos de ordenador, pero dado que el navegador usa esa tecla para salir del modo pantalla completa no es buena opción. Es posible activar y desactivar un

**Figura 5.21:** Ventana de opciones en el menú principal.

GameObject de la escena desde código con la función SetActive(bool) que proporciona la clase GameObject. Al abrir el menú de pausa se activa una imagen. Esta imagen es como un filtro que oscurece la zona de juego, en la figura 5.22 se puede observar el menú de pausa abierto. Todo lo relacionado con la interfaz gráfica en los niveles se controla mediante un *script* llamado UIController.cs, el cual no se explicará ya que es bastante sencillo y no aporta nada nuevo respecto a lo ya visto en *scripts* anteriores.

**Figura 5.22:** Ventana de opciones en el menú principal.

## 5.9 Persistencia

Conforme el proyecto iba avanzando se han detectado diversas situaciones en las que hay que almacenar datos para que cierta información se mantenga en la siguiente sesión de juego. Es necesario almacenar información en disco que represente en que parte de la campaña se ha quedado el jugador, para así no tener que volver a empezar del principio. También es conveniente que el récord del nivel infinito se mantenga a pesar de cerrar la aplicación. Y por último están las opciones, lo más adecuado es que se guarde la configuración que ha establecido el usuario en las siguientes sesiones, porque sino tendría que volver a configurar el juego.

Unity proporciona una herramienta para almacenar datos en disco, se trata de una clase llamada PlayerPrefs, la cual permite guardar datos de tipo *string*, *int* y *float*. Las funciones que proporciona para guardar datos toman 2 argumentos, el primero es un nombre con el cual se identificará la información almacenada, y el segundo argumento es el valor a almacenar.

El guardado del progreso de la campaña se ha resuelto dentro de los *scripts* que gestionan los niveles, cuando se detecta que se ha completado el nivel se guarda en disco un entero que tiene como valor el número del nivel siguiente. Por ejemplo, al completar el nivel 2 se guardará en disco un 3, y en el menú principal al pulsar el botón Continue se lee ese valor y se lanza el nivel 3. En cuanto al récord, cuando el jugador pierde si su puntuación supera al récord previo, se actualiza el valor en disco. Hay que tener en cuenta que todos los datos que se quieren almacenar en disco deben tener nombres identificativos diferentes. Cuando se abre la escena del nivel infinito se lee siempre el valor de la puntuación más alta obtenida hasta el momento. Con la configuración de sonido y sensibilidad del ratón se hace algo parecido, cada vez que se modifica el valor de un *slidder* se actualiza el valor en disco que almacena el valor del *slidder*, y al iniciar el programa se leen estos valores para establecer la configuración de la sesión anterior.

# CAPÍTULO 6
# Pruebas

En este capítulo se expondrán una serie de pruebas que se han llevado a cabo en el videojuego para comprobar que todo funciona adecuadamente y que la experiencia que proporciona este sea la buscada.

## 6.1 Rendimiento con WebGL

Esta primera prueba consiste en generar los archivos finales del videojuego y probarlo en ordenadores de diferente potencia para observar si el rendimiento que presenta es adecuado. Se ha añadido un *script* en todas las escenas el cual mostrará los fotogramas por segundo en todo momento. Las especificaciones de los ordenadores de prueba son:

**Ordenador 1:**
SO: Windows 10
CPU: AMD Ryzen 5 2600
RAM: 16GB 2933 Mhz
GPU: NVIDIA GeForce GTX 1060 6GB

**Ordenador 2:**
SO: Windows 10
CPU: Intel Core i5 8250U
RAM: 8GB 2400 Mhz
GPU: NVIDIA GeForce GTX 1050 2GB

**Ordenador 3:**
SO: Windows 10
CPU: Intel Core i3 4030U
RAM: 4GB 1600 Mhz
GPU: Intel HD Graphics 4400

El primer ordenador es sobremesa y tiene buen rendimiento en juegos, pero los otros 2 son portátiles con procesadores de bajo consumo. El ordenador 2 es bastante actual y con tarjeta gráfica dedicada, mientras que el ordenador 3 es un portátil más antiguo y sin tarjeta gráfica dedicada. El objetivo es que el videojuego sea jugable en las 3 maquinas. Lo adecuado sería que los primeros 2 ordenadores lo pudiesen ejecutar a 60 fotogramas por segundo, mientras que en el tercero se deberían obtener al menos 30 fotogramas por segundo. Hay que tener en cuenta que si tuviésemos un ejecutable para Windows el rendimiento sería muy superior al que se obtiene ejecutando en el navegador.

## 6.2 Dificultad y controles

Es necesario probar si la dificultad es adecuada, que el juego no sea fácil ni muy difícil. La dificultad que se busca que tenga el videojuego es media-alta, para verificar si es así no basta con que el creador lo juegue, ya que conoce muy bien el juego y tiene mucha experiencia en este a causa de las pruebas que se van haciendo durante el desarrollo. Por eso, el juego será probado por personas que no están implicadas en el desarrollo, las cuales después darán su opinión sobre la dificultad. También una persona con experiencia en videojuegos jugará a todos los niveles y rellenará una tabla con el número de veces que ha muerto en cada nivel.

En cuanto a los controles, se probará si el movimiento de la nave responde bien y como cambiaría la experiencia al modificar diferentes parámetros de los dispositivos de entrada en la ventana Project Settings.

## 6.3 Desplazamiento de los enemigos

El algoritmo que se usa para que los enemigos no se choquen tiene diferentes parámetros, los más importantes son los siguientes: ángulo de giro de los rayos respecto a la dirección de avance, tamaño de la esfera que se lanza alrededor del rayo, longitud del rayo y la distancia al nuevo destino desde la posición actual. Se probarán diferentes combinaciones de valores en estos parámetros y se observará como funciona en la práctica el algoritmo con ellos. Se elegirán los valores que proporcionen el mejor resultado. Entendiendo como mejor resultado el caso en el que los enemigos no se atraviesan y en el que tampoco cambian su ruta cuando no es necesario.

CAPÍTULO 7

# Resultados

## 7.1 Rendimiento y optimización

En el ordenador 1 se ha probado el videojuego a una resolución de 2560 × 1440 y este se ha ejecutado a 60 fotogramas por segundo sin problemas. Únicamente al iniciar el juego, durante los primeros segundos, los fotogramas por segundo han bajado de los 60, esto probablemente se asocie a que el juego aún no había cargado todos los archivos necesarios. En el ordenador 2 se ha ejecuta el juego a una resolución de 1920 × 1080 y el resultado ha sido el mismo que con el ordenador 1. En cambio, con el ordenador 3, en el cual se ha ejecutado a una resolución de 1366 × 768, el resultado de las pruebas no fue satisfactorio. El juego en el nivel infinito, que es el más exigente debido a la gran cantidad de enemigos en pantalla y a los *scripts* para generar enemigos aleatoriamente, se mantenía entre los 20 y 30 fotogramas por segundo y el control de la nave se notaba ralentizado, al pulsar una tecla el resultado tardaba en verse en pantalla.

Tras el anterior resultado en el ordenador 3 se procedió a optimizar el juego para que al menos se pueda jugar en ese ordenador a 30 fotogramas por segundo y con una respuesta más rápida de los controles. En los videojuegos, el número mínimo de fotogramas por segundo que se recomienda son 30, bajando de 30 la experiencia de juego que se obtiene no es buena. Se ha considerado que si el juego es capaz de ejecutarse a ese número de fotogramas en el ordenador 3 será jugable en la gran mayoría de ordenadores que están en uso hoy en día, dado que ese portátil tiene un procesador de bajo consumo, de gama baja y lanzado en 2014.

En el proceso de optimización se ha usado la ventana de Unity llamada Profiler Window, la cual, entre otras cosas, muestra durante la ejecución cuantos recursos consume cada aspecto de la escena —algunos aspectos de la escena que aparecen son las físicas, el renderizado de los GameObjects y la ejecución de los *scripts*—. Haciendo uso de dicha ventana se ha observado que lo que más procesador consume es el renderizado de la escena. Para disminuir la carga que supone el renderizado se han llevado a cabo diferentes acciones. Primero se ha reducido el tamaño de las texturas que se utilizan, se ha buscado un equilibrio entre tamaño y calidad. Ha sido posible reducir el tamaño drásticamente sin observar mucho impacto en la calidad dentro del juego, debido a que los objetos están ubicados a bastante distancia de la cámara y desde esa distancia no es posible apreciar los detalles de las texturas. Otra optimización llevada a cabo es marcar objetos de la escena que no se van a mover como estáticos, lo que permite a Unity realizar optimizaciones en ellos calculando algunas cosas al exportar el juego en vez de hacerlo en tiempo real.

Al seleccionar un material en Unity, en la ventana inspector aparecen diferentes opciones y hay una interesante para la optimización llamada *GPU instancing*. Esta opción es conveniente activarla en materiales que van a aparecer en objetos repetidos en la escena,

así se dibujarán a la vez y se consumirán menos recursos. Esta opción ha sido activada en los materiales de los disparos enemigos, en el enemigo normal y en el enemigo avanzado. En la ventana Project Settings, en la pestaña de físicas, hay una matriz en la cual se puede seleccionar qué capas deben colisionar entre sí. Se han desactivado las colisiones innecesarias para evitar algunas llamadas a las funciones que detectan las colisiones y los campos de activación.

Llevando a cabo todo lo anterior se ha conseguido que el videojuego se ejecute entre 30 y 40 fotogramas por segundo en el equipo 3. También los controles responden mucho mejor en este equipo tras los cambios.

## 7.2 Dificultad y controles

La opinión de los jugadores que han probado el juego es que no es excesivamente complicado, pero sí que les ha costado superar algunos niveles. El usuario que ha rellenado la tabla ha muerto al menos 1 vez en cada nivel, menos en el del tutorial. Se ha tomado la decisión de reducir la velocidad de disparo del enemigo normal y del enemigo avanzado en 0,5 segundos. Después del cambio el enemigo normal disparará una bola cada 2,5 segundos, mientras que el enemigo avanzado lo hará cada 3,5 segundos.

En cuanto a los controles, antes de enviar la versión de prueba se ha configurado que la respuesta a las teclas sea prácticamente inmediata. Unity por defecto añade algo más de retraso, quitando ese pequeño retraso hace menos realista el control, dado que una nave no debería poder frenar y cambiar de dirección inmediatamente. Pero de esta forma la jugabilidad es mejor. Las personas que lo han probado han comentado que el control está bien y que es muy preciso.

Al probar el juego en el navegador, se ha notado que al no ponerlo en pantalla completa puede ocurrir que se pulse con el ratón alguna cosa fuera del juego. Si el usuario va con cuidado esto no debería suceder, pero es conveniente poder ocultar el cursor. Esto se puede hacer cambiando el valor de la variable lockState de la clase Cursor. Con «Cursor.lockState = CursorLockMode.Locked» se consigue que el cursor esté en el centro de la ventana bloqueado e invisible. Pero los navegadores web por seguridad no permiten bloquear el cursor a menos que sea en respuesta a un evento iniciado por el usuario. Por eso, se ha hecho que el cursor se bloquee con la tecla 1, y que con la tecla 2 se desbloquee, además, al abrir el menú de pausa siempre se desbloquea para poder usar los botones. Cómo bloquear y desbloquear el cursor se le explica al usuario en el menú principal en una pequeña ventana.

## 7.3 Desplazamiento de los enemigos

Después de hacer varias pruebas el mejor resultado ha sido obtenido con los siguientes parámetros:

- Ángulo de giro de los rayos respecto a la dirección de avance = 7°
- Radio de la esfera que se lanza alrededor del rayo = 1,5 unidad [1]
- Longitud del rayo = 10 unidades
- Distancia al nuevo destino desde la posición actual = 10 unidades

[1] La escala predeterminada de unidades de Unity es 1 unidad = 1 metro.

## 7.4 Videojuego

Después de corregir los aspectos deficientes detectados en la fase de pruebas, este proyecto queda finalizado. El resultado final de este trabajo es un videojuego de niveles variados y que tiene un rendimiento óptimo en la Web. Hay disponibles 2 modos de juego: la campaña y el modo infinito. La campaña alterna entre niveles de combate 2.5D y niveles de esquivar obstáculos que son 3D, y, por último, está el nivel en el que hay que derrotar a un jefe final. Todos estos niveles tienen una dificultad adecuada: hacen el juego desafiante y al mismo tiempo no llegan a frustrar al jugador. Si bien quedan algunos detalles que se podrían pulir o ideas que se podrían implementar, que se verán en el último capítulo de este trabajo, el resultado es un videojuego bastante completo. En las imágenes 7.1 y 7.2 se pueden ver los niveles de combate, mientras que en la imágen 7.3 se puede observar una captura de uno de los niveles 3D.

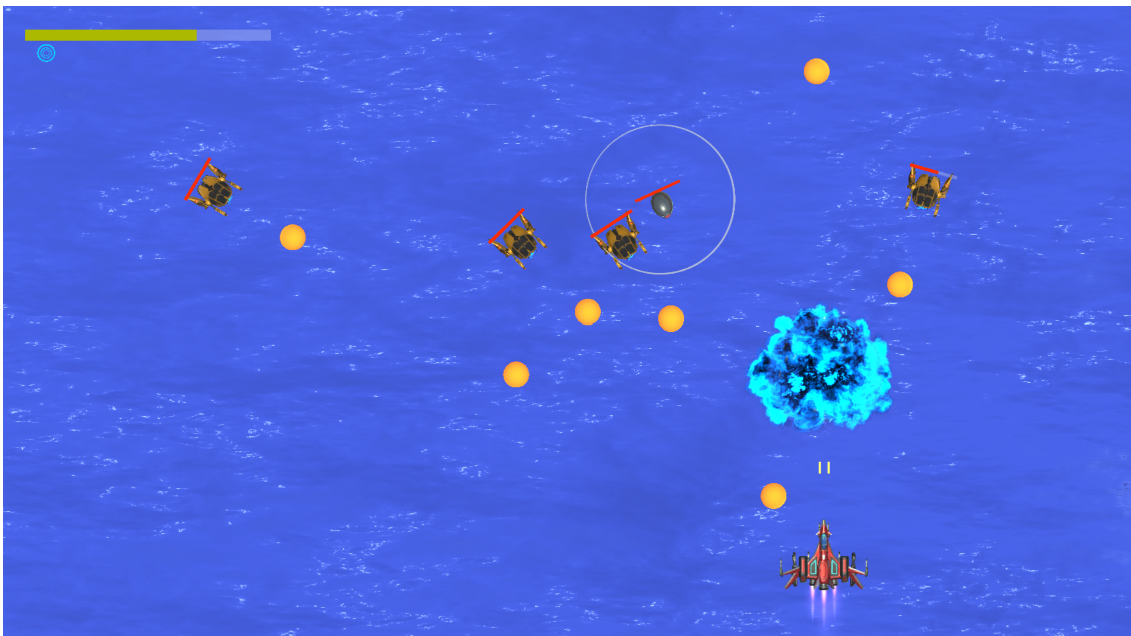

**Figura 7.1:** Captura de pantalla del nivel 4.

**Figura 7.2:** Captura de pantalla del nivel 5.

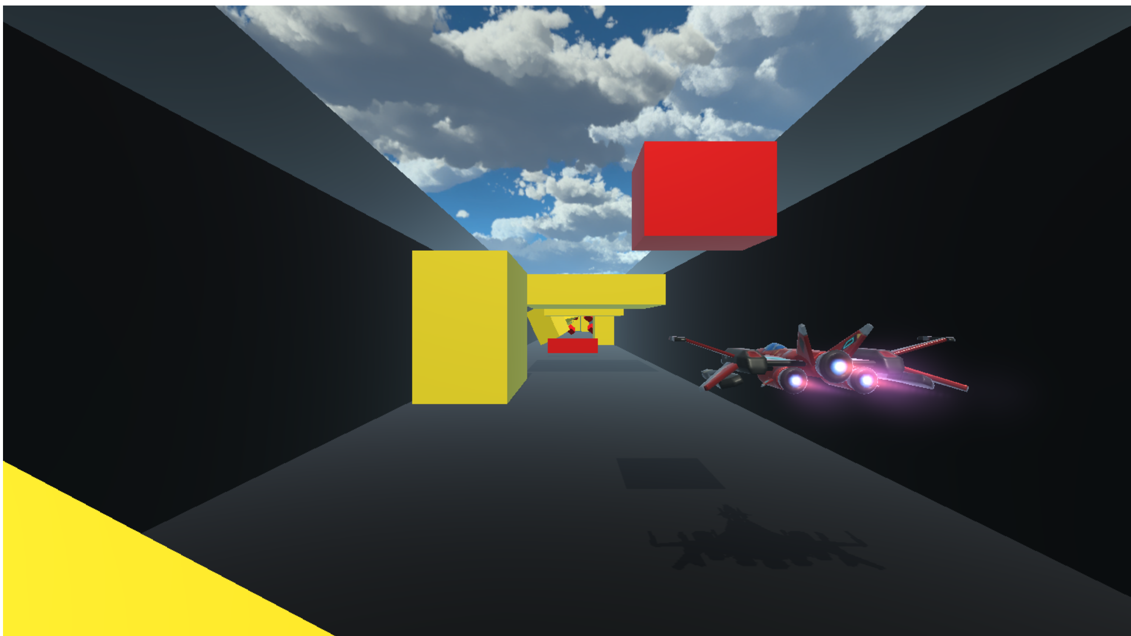

**Figura 7.3:** Captura de pantalla del nivel 7.

CAPÍTULO 8
# Conclusiones

Al inicio de este trabajo se han planteado diversos objetivos, ahora, tras haberlo realizado, se analizará si han sido alcanzados y después se relacionará el trabajo con lo estudiado en el grado.

El primer objetivo que fue propuesto es desarrollar un videojuego sencillo pero a la vez completo. Esto ha sido llevado a cabo con éxito ya que el videojuego que se ha desarrollado parte de una idea sencilla que se ha implementado correctamente, proporcionando una buena jugabilidad acompañada de música y sonidos que se adecuan a cada situación y mejoran la experiencia. También la interfaz de usuario es intuitiva, sigue las pautas de la industria de hoy en día y proporciona varias configuraciones del juego que son muy útiles. El siguiente objetivo que se estableció es desarrollar un juego que sea posible de ejecutar en la Web, como se ha visto en el capítulo de resultados, el juego se puede ejecutar en máquinas poco potentes desde el navegador, por consiguiente, este objetivo queda cumplido también.

El tercer objetivo, el cual trataba de crear un videojuego entretenido también ha sido cumplido, hay diferentes modos de juego y cada nivel de la campaña añade algún elemento nuevo. También la dificultad ayuda a que el juego no sea aburrido ya que plantea un reto al jugador. El último objetivo del trabajo es familiarizarse con Unity y con el uso de C# en este motor. Llevando a cabo este trabajo he aprendido muchas cosas nuevas sobre cómo se trabaja con Unity, también he descubierto que ofrece muchas herramientas y posibilidades, que no han sido exploradas todas dado que no han sido del todo necesarias para el videojuego, así que quedarían más cosas por aprender sobre Unity. Pero tras lo aprendido en este proyecto sería capaz de afrontar proyectos de mayor complejidad con este motor de videojuegos.

En este trabajo han sido utilizados muchos conocimientos adquirido en asignaturas del grado. Empezando por la parte de programación, en múltiples asignaturas de los primeros 2 cursos se ha usado Java, que es un lenguaje orientado a objetos muy similar a C#, además, en tercero, en la asignatura de «Ingeniería del software» las prácticas fueron realizadas en C#. Por otra parte, en la asignatura «Introducción a los sistemas gráficos interactivos» se han visto muchos conceptos de gráficos por ordenador que aparecen en cualquier motor de videojuegos. También en cuarto he cursado varias optativas sobre desarrollo de videojuegos, las cuales me han proporcionado una base para el desarrollo de este trabajo.

# CAPÍTULO 9
# Trabajo futuro

Dadas las limitaciones de tiempo de un trabajo de esta índole han quedado algunos aspectos sin desarrollar. En futuras versiones del juego se podrían añadir algunas de las siguientes ideas o mejoras:

- Implementar en el modo infinito un ranking en línea, así los jugadores podrían competir contra otros usuarios.
- Desarrollar algunos niveles más para la campaña.
- Trabajar más la historia que se expone al inicio de la campaña.
- Sustituir las esferas que representan los disparos enemigos por sistemas de partículas para darles un aspecto mejor, habría que controlar que los sistemas de partículas no fuesen muy complejos para que el rendimiento con WebGL fuese bueno.

# Bibliografía


[1] El videojuego en el mundo.[En linea] *Asociación española de videojuegos* [fecha de consulta: 31 mayo 2019]. Disponible en http://www.aevi.org.es/la-industria-del-videojuego/en-el-mundo/.

[2] Simone Belli y Cristian López Reventós. Breve historia de los videojuegos. *Athenea Digital* [en línea]. otoño 2008, núm. 14: 159-179 [fecha de consulta: 5 junio 2019], ISSN: 1578894. Disponible en https://www.raco.cat/index.php/Athenea/article/view/120290/164303.

[3] Paul, Partha & Goon, Surajit & Bhattacharya, Abhishek. History and comparative study of modern game engines. *International Journal of Advanced Computer and Mathematical Sciences* [en línea]. 2012, Vol 3, 245-249 [fecha de consulta: 6 junio 2019]. ISSN: 2230-9624. Disponible en https://bipublication.com/files/IJCMS-V3I2-2012-07.pdf.

[4] Henry Lowood. Game Engines and Game History. *Kinephanos* [en línea]. Enero 2014. [fecha de consulta: 6 junio 2019]. ISSN: 1916-985X. Disponible en https://www.kinephanos.ca/Revue_files/2014-Lowood.pdf.

[5] Colaboradores de Wikipedia. Shoot 'em up [en línea]. Wikipedia, La enciclopedia libre [fecha de consulta: 1 agosto 2019]. Disponible en https://es.wikipedia.org/wiki/Shoot_%27em_up.

[6] Manual de usuario de Unity 2018.3 [en línea]. [fecha de consulta: 15 mayo 2019]. Disponible en https://docs.unity3d.com/es/current/Manual/UnityManual.html.

[7] Hocking, Joseph. *Unity in action: multiplatform game development in C#*. 2nd ed. Shelter Island, NY: Manning, cop. 2018. ISBN 9781617294969.

[8] Referencia de Lenguaje de Unity 2018.3 [en línea]. [fecha de consulta: 30 mayo 2019]. Disponible en https://docs.unity3d.com/es/current/ScriptReference/index.html.

[9] Explorador de API de .NET [en línea]. [fecha de consulta: 30 mayo 2019]. Disponible en https://docs.microsoft.com/es-es/dotnet/api/?view=netframework-4.8.

[10] CSharpMessenger Extended [en línea]. [fecha de consulta: 1 mayo 2019]. Disponible en http://wiki.unity3d.com/index.php/CSharpMessenger_Extended.

# ANEXO A
# Game Design Document

## Attack of the Drones

**Desarrollador:** Radosvet Desislavov Georgiev
**Plataforma:** Web
**Género:** Shoot 'em up
**Ambientación:** Futurista
**Motor:** Unity

## A.1 Descripción

Juego de disparos en el cual se toma el control de una nave y hay que ir acabando con diversos enemigos. La cámara está ubicada encima del campo de batalla y la nave se puede mover en 2 dimensiones. Se compone de dos modos de juego, una campaña y un nivel de combate sin fin en el cual hay que obtener la mayor puntuación posible. En la campaña tras varios niveles de combate se pasará a un nivel de esquivar obstáculos con la nave, el cual aporta una jugabilidad totalmente nueva para después regresar con los niveles de combate. La campaña finalizará con un combate contra un jefe final.

## A.2 Historia

Es el año 2030 y los drones de combate están siendo muy utilizados en todo el mundo con fines militares. Pero un día sucede algo y los gobiernos pierden el control sobre los drones y estos empiezan a atacar a la humanidad. Entonces les corresponde a las personas salir a combatirlos. El protagonista es el piloto de un caza que debe destruir a los drones que están fuera de control.

## A.3 Personajes

### A.3.1. Nave que controla el jugador

Es el caza con el cual se juega, dispara proyectiles y también puede lanzar una onda. En la figura A.1 se puede observar este personaje disparando un proyectil.

**Figura A.1:** Nave aliada.

### A.3.2. Dron normal

Este es el enemigo más básico que se puede encontrar, se puede desplazar entre dos puntos dados del campo de batalla o puede estar quieto. Dispara esferas de color naranja. El aspecto de este y el proyectil que dispara se pueden ver en la figura A.2.

### A.3.3. Dron avanzado

Enemigo con más vida que el normal y con disparos de color negro, también puede estar quieto en un punto o desplazarse entre dos puntos. El aspecto de este y el proyectil que dispara se pueden ver en la figura A.3.

### A.3.4. Dron explosivo

Este enemigo persigue en todo momento a la nave aliada y cuando está cerca se autodestruye provocando una explosión enorme que daña al jugador. Es el enemigo que tiene más vida y provoca más daño, pero su velocidad de movimiento es baja. El aspecto del dron se puede ver en la figura A.4.

### A.3.5. Dron que lanza rayos

Enemigo que se desplaza siempre entre dos puntos del campo de batalla y cada cierto tiempo lanza un rayo, el cual daña a la nave del jugador mientras esta colisione con el rayo. En la figura A.5 se puede ver este dron lanzando el rayo.

**Figura A.2:** Dron normal.

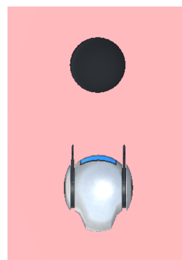

**Figura A.3:** Dron avanzado.

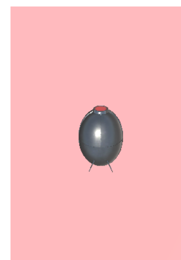

**Figura A.4:** Dron explosivo.

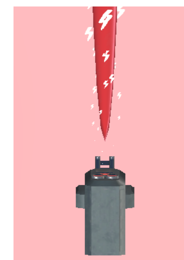

**Figura A.5:** Dron que lanza rayos.

### A.3.6. Jefe final

Este enemigo se encuentra únicamente en el último nivel de la campaña y tiene una gran cantidad de puntos de vida. Tiene 3 posibles ataques: disparar esferas naranjas que forman una circunferencia, disparar esferas naranjas de 3 en 3 y crear drones explosivos. En la figura A.6 se puede ver su aspecto.

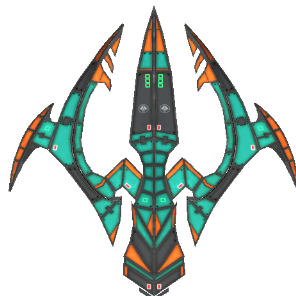

**Figura A.6:** Jefe final.

## A.4 Jugabilidad

En este apartado hay que diferenciar entre 2 casos: los niveles de combate, en los cuales se usa una cámara ortográfica, y los niveles de esquivar, en los cuales la cámara es perspectiva.

### A.4.1. Niveles de combate

En estos niveles el personaje puede desplazarse a cualquier punto del plano en el que ocurre la acción. A parte de moverse por el plano tiene dos posibles acciones más que son ataques, un disparo láser que va en línea recta hacia arriba y una onda que se expande alrededor de la nave. Si alguno de los dos ataques llega a impactar con un enemigo este recibe daño y si su vida llega a 0 se destruye. La onda puede destruir los proyectiles enemigos de color naranja que se encuentren en su radio pero hay que esperar 5 segundos para poder volver a utilizarla. Cada cierto tiempo aparecerá una llave inglesa en pantalla y si la nave aliada pasa encima de ella, se reparará.

### A.4.2. Niveles de esquivar

En los niveles de esquivar obstáculos la cámara está ubicada atrás de la nave. La nave avanza constantemente hacia delante y el jugador puede mover la nave con el ratón a cualquier punto dentro de los límites de la pantalla para evitar chocar. En estos niveles no hay enemigos y, por consiguiente, no se puede disparar.

## A.5 Niveles

En el videojuego habrá dos modos de juego, la campaña y el que será llamado nivel infinito.

### A.5.1. Campaña

La campaña se compone de 9 niveles. En el nivel 1 se cuenta la historia del juego y el nivel 2 es un tutorial en el que se enseña al jugador como afrontar los niveles de combate. Los niveles posteriores, el 3 y el 4, son niveles de combate en los que el jugador tiene que destruir todos los drones enemigos para superarlos. El nivel 5 es el primer nivel de esquivar obstáculos que va seguido de otros 2 niveles de combate —el nivel 6 y el 7—, estos últimos son bastante más difíciles que los previos. El nivel 8 es otro nivel de esquivar tras el cual llega el turno del jefe final, que será el nivel 9 en el cual solo habrá un enemigo a derrotar pero bastante fuerte.

### A.5.2. Nivel infinito

Este es como un nivel de combate de la campaña con la diferencia de que el nivel acaba cuando el jugador muere, ya que se generan enemigos infinitamente. Cada enemigo derrotado proporciona una cantidad de puntos que se irá sumando a la puntuación del jugador. La puntuación actual y la mayor puntuación obtenida hasta el momento se mostrará en todo momento en pantalla durante este nivel.

## A.6 Mecánicas

Se pueden diferenciar tres tipos de niveles en los que las mecánicas son distintas.

### A.6.1. Niveles de combate de la campaña

El jugador puede desplazarse por toda la pantalla y debe derrotar con los dos ataques de los que dispone a las oleadas de enemigos que se presentan para superar este tipo de niveles. Si la vida del jugador llega a cero se deberá reiniciar el nivel. De vez en cuando aparecen objetos reparadores que el jugador puede recoger para recuperar puntos de vida. Los proyectiles enemigos de color naranja se pueden destruir disparándoles o activando la onda, en cambio, los proyectiles de color negro deberán ser esquivados.

### A.6.2. Niveles de esquivar obstáculos

En estos niveles la nave sigue en todo momento el movimiento del ratón, y así se tienen que esquivar todos los obstáculos. Si la nave se choca con un obstáculo esta se destruye y la partida acaba y hay que reiniciar el nivel.

### A.6.3. Nivel infinito

Las mecánicas es este nivel son muy parecidas a las de los niveles de combate de la campaña con la diferencia de que este nivel no tiene un final programado, el objetivo no es acabar con todas las oleadas. El objetivo es obtener la mayor puntuación posible y para ello el jugador debe evitar recibir daños y destruir el mayor número de enemigos posible.

## A.7 Habilidades del jugador

Este videojuego requiere ciertas habilidades del jugador. La primera de ellas son reflejos rápidos, dado que suele haber gran número de proyectiles enemigos en pantalla en

los niveles de combate, y que en los niveles de esquivar hay que evitar obstáculos que se acercan a gran velocidad, los reflejos rápidos son fundamentales para avanzar en el juego. También es necesaria capacidad de predicción para suponer como avanzarán los disparos enemigos y no quedarse atrapado en una zona de la pantalla en la cual no se podrá esquivarlos. Otra habilidad importante es la percepción de la distancia. Esta habilidad hace falta para saber cuándo es oportuno usar la onda y para controlar que los enemigos explosivos no entren en rango de explotarse.

## A.8 Sonido

Los efectos de sonido son los siguientes:

- Sonido de explosión normal: se reproduce cuando cualquier nave o dron es destruido, a excepción del dron explosivo.
- Sonido de explosión fuerte: se reproduce cuando el dron explosivo explota.
- Sonido de disparo láser: cada vez que la nave aliada dispara este sonido es reproducido.
- Sonido de reactor de caza: este efecto solo está presente en los niveles de esquivar obstáculos y está activado siempre, ya que la nave nunca está quieta.

La música para los niveles ha sido obtenida de la página OpenGameArt [1] y es la siguiente:

- Cyberpunk Moonlight Sonata: se usa como música de fondo en los niveles de combate.
- Through Space: es la música de fondo de los niveles de esquivar obstáculos.
- Tragic Ambient: esta pista se reproduce durante el nivel en el cual se cuenta la historia.

[1] https://opengameart.org/