Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Tank Defender! Desarrollo de un videojuego de acción en tercera persona mediante Unity3D

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Josep Vicent Campos Puig

Tutor: Javier Lluch Crespo

2018-2019

Resumen

En este proyecto se crea un videojuego en 3D en el que controlamos un tanque y debemos defendernos de hordas de robots. El objetivo del videojuego es aguantar cuantas más rondas y no morir en el intento. Para ello existe una tienda en la que se pueden comprar tanto nuevas armas como escudos y mejoras. Los enemigos, al morir, dan unos puntos de experiencia que puede usar el jugador para comprar en dicha tienda. Además, los enemigos aparecen en oleadas de dificultad ascendente y cada 5 rondas existe la posibilidad de que aparezcan enemigos más difíciles pero que dan mejores recompensas.

Palabras clave: tanque, videojuego, 3D, unity, robots, rondas.

Abstract

In this project is a creation of a third person action videogame where the player controls a tank and must defend itself against a horde of robots. The main goal is to survive the maximum number of rounds without getting killed. For that, the player can buy weapons, armor and upgrades at the store. When the enemies die, they give the player experience points that he or she uses to buy these upgrades. Furthermore, the enemies come in rounds which have ascending difficulty and every five rounds there is a chance that a more challenging kind of enemy shows up.

Keywords: tank, videogame, 3D, unity, robots, rounds.

Resum

Es aquest projecte es crea un videojoc en 3D en el que controlem un tanque i hem de defensar-nos de una horda de robots. El objectiu principal del joc es aguantar viu a més rondes millor I no morir en el camí. Per a questa tasca el jugador pot comprar armes, defenses i millores en una tenda. Els enemics, al morir, donen al jugador punts d'experiencia que pot gastar per a comprar les millores en la tenda. Amés, els enemics apareixen en rondes de dificultad ascendent i cada cinc rondes cap la posibilitat de que apareguen uns enemics més difícils que donen millors recompenses.

Paraules clau: tanque, videojoc, 3D, unity, robots, rondes.

Tabla de contenidos

Índice de imágenes

1. Introducción

En este primer capítulo, se van a presentar los motivos que nos han llevado a realizar este proyecto y los objetivos que deseamos alcanzar.

1.1 Motivación

En el año 2018, el valor de la industria del videojuego en el mercado global alcanzó los 137.900 millones de dólares estadounidenses (un 17% más que en 2017) según *Newzoo,* una empresa que se dedica a realizar estudios de mercado. Esto sugiere dos cosas, que los videojuegos son producto cada vez más demandado y que, por ello se va a invertir más y más en este mercado.

Pero no solamente las grandes compañías como *EA,* o *Ubisoft* pueden desarrollar videojuegos, también existen grupos pequeños de desarrolladores que, sin tener el respaldo de una gran empresa, se dedican a la creación de videojuegos. Esto cada vez es más común gracias a plataformas como *Steam,* que aunque no apoyen económicamente a estos pequeños grupos, les permiten alcanzar una mayor visibilidad que no podrían alcanzar por si solos.

La motivación personal que ha llevado a la creación de este proyecto es el querer trabajar en el desarrollo de videojuegos en un futuro. Mediante este proyecto se busca afianzar los conocimientos adquiridos en el diseño y la creación de un videojuego.

También cabe destacar que este videojuego entraba dentro de un concurso para minijuegos.com, en el que diez estudiantes para su proyecto de final de grado crean un videojuego y los dos mejores son seleccionados para poder subir dicho videojuego a la web de minijuegos.

1.2 Objetivos

Los objetivos de este proyecto son crear un videojuego en tercera persona en el que el jugador controla un tanque y debe defenderse de una horda de robots. Estos robots serán de diferentes tipos, una especie de arañas que se acercan al jugador y realizan ataques cuerpo a cuerpo, y unos drones que sobrevuelan el entorno y disparan a distancia. El propio jugador dispondrá de diferentes tipos de torretas para defenderse, como un cañón; que tiene un daño moderado y una cadencia moderada, una ametralladora; que hace menos daño pero dispara más rápido y un misil dirigido que hace daño en área. Además existirá una tienda en la que se podrán comprar mejoras como una torreta que nos sigue y dispara automáticamente a los enemigos, blindajes poder resistir más golpes de los enemigos, munición para las armas y salud.

Para exponer estos objetivos de una manera concreta vamos a separarlos en diferentes puntos:

- Tanque controlado por el jugador:
  - Tres torretas con diferentes armas
- Enemigos:
  - Enemigo cuerpo a cuerpo que persigue al jugador
  - Enemigo volador que sobrevuela el mapa y dispara
- Mejoras:
  - Tienda donde comprarlas
  - Torreta que sigue al jugador y dispara a los enemigos cercanos
  - Mejoras en los blindajes
  - Mejoras en las armas

Todo esto se creará utilizando diferentes herramientas como: *Unity3d*, entorno de desarrollo de videojuegos, *Blender*, un programa de modelado en 3D en el que crearemos algunos modelos, y *GIMP*, un software de edición de imágenes.

1.3 Estructura de la memoria

En este apartado, vamos a presentar cómo se estructura este documento.

Esta memoria se divide en diferentes capítulos, en el primero, la introducción, ya hemos presentado los motivos que nos han llevado a realizar este proyecto y los objetivos que queremos conseguir. En el siguiente capítulo, el estado del arte, introducimos el panorama en el que se desarrolla y exponemos las diferentes tecnologías que existen actualmente relacionadas con la creación de videojuegos. Más adelante, presentamos el diseño, cómo queremos que se estructure este y cómo queremos implementarlo. En el siguiente capítulo, indicamos cómo se ha implementado finalmente explicando las diferentes clases y scripts creados. Para acabar, en los últimos apartados se exponen las pruebas realizadas, las conclusiones extraídas del proyecto y modificaciones que se pueden realizar en un futuro.

2. Estado del arte

En este capítulo, presentamos el contexto en el que se desarrolla el proyecto y las diferentes tecnologías que existen en el mercado actualmente relacionadas con la creación de videojuegos.

2.1 Entornos de desarrollo de videojuegos

En este apartado veremos algunos de los programas de diseño de videojuegos más utilizados en los últimos años.

CryEngine

El 2 de mayo de 2002 la empresa *CryTek* lanzaba el motor *CryEngine*, que inicialmente fue creado para demostraciones de *Nvidia*. Este motor siempre ha gozado de gran calidad visual en sus gráficos tal y como se puede observar en muchos de los videojuegos creados con esta plataforma, véase *Crysis, FarCry* o *Kingdom Come: Deliverance.* Desde hace unos años, este motor tiene el modelo de negocio "*pay what you want*", es decir, que su código fuente está al alcance de todos sin tener que pagar nada.

Figura 1: Imagen del videojuego Crysis 3 desarrollado en CryEngine

Unreal Engine

Unreal Engine es otro entorno de desarrollo lanzado por *EpicGames* en 1998. Siempre ha gozado de buena fama ya que desde sus inicios contaba con buenas herramientas de iluminación y de renderizado. Además, gracias a su lenguaje de scripting permitía a los usuarios crear alteraciones de los juegos hechos con este motor lo que gustó en gran medida a la comunidad de *modders*.

En el año 2015 *Unreal Engine* pasó a ser de código abierto con lo que en estos momentos también se trata de una herramienta gratuita y accesible para todo el mundo.

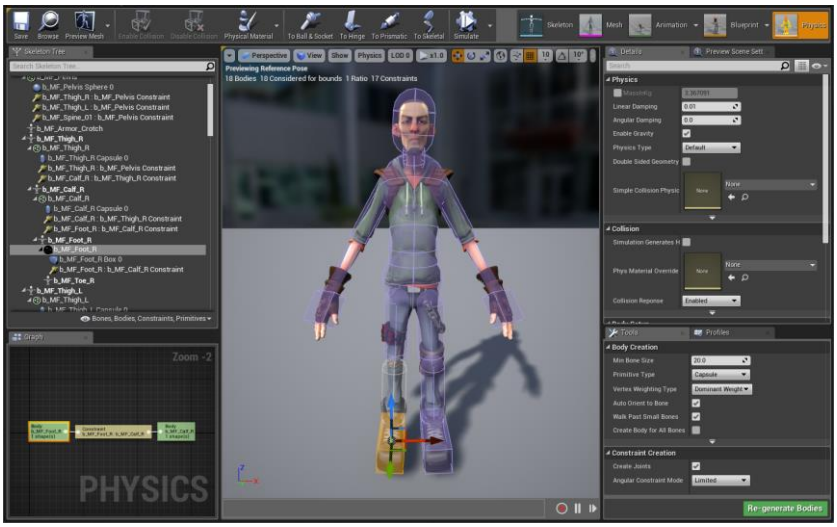


Figura 2: Interfaz de UnrealEngine

También cabe destacar que este motor no se usa solamente para el desarrollo de videojuegos sino que también se utiliza en el cine puesto que permite renderizar escenas con una gran calidad visual.

Unity3D

Unity es un software de creación y diseño de videojuegos, fue lanzado en 2005 por la empresa *Unity Technologies*, y desde entonces ha estado ganando popularidad debido a su potencia y sencillez.

Una de sus mejores ventajas es la capacidad para compilar el mismo proyecto para diferentes plataformas de una manera sencilla, por ejemplo, se puede

compilar para *Mac,* Linux y *Windows,* y desde una opción en la configuración del proyecto, es posible pasar a *WebGL* sin tener que realizar ninguna configuración más, el propio motor se encarga de realizar todos los cambios. Además, unity en su versión gratuita tiene todas las funcionalidades que ofrece en su licencia de pago. Por estos motivos, *Unity3D* es uno de los motores más utilizados por los desarrolladores en los últimos años.

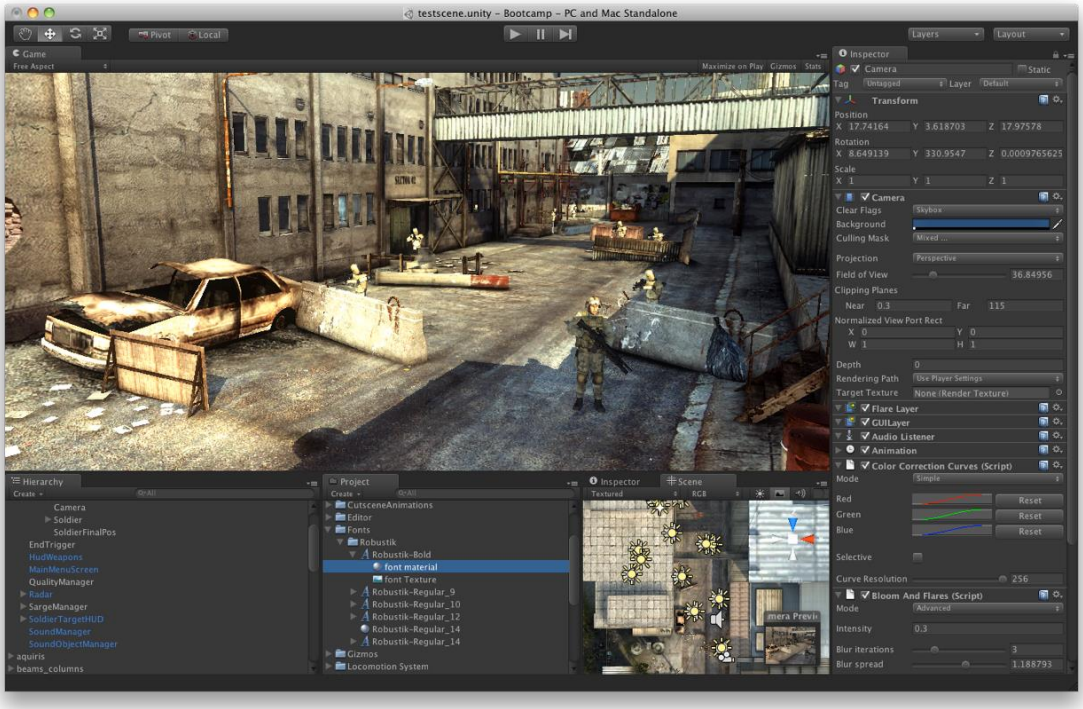


Figura 3: Interfaz del UnityEditor

También cabe destacar que este motor no ha sido el mejor en cuanto a la calidad gráfica de sus videojuegos, pero en los últimos años el equipo de desarrollo de *Unity* ha estado mejorando el motor en este aspecto.

Características de Unity3D

Para conocer mejor el entorno de *Unity*, vamos a presentar algunas de sus características básicas de su funcionamiento.

Los proyectos en *Unity* se distribuyen en escenas llamadas scenes. En estas escenas es donde se crea el entorno donde el jugador interactuará con los objetos, enemigos, terrenos, fuentes de luz, sonidos e interfaces de nuestro juego. Estas escenas se pueden entender como las pantallas o niveles de nuestro juego.

Los elementos de estas escenas son llamados *GameObjects*, son objetos que contienen los llamados componentes que dotan de funcionalidad a estos objetos. *Unity* nos ofrece una gran cantidad de componentes que agregar a nuestros *GameObjects* como:

- *Transform*: La mayoría de objetos tienen este componente por defecto. Define la posición, rotación y escalado en el espacio. También se usa para establecer una relación de jerarquía entre diferentes objetos.

- *Collider*: Este componente agrega una malla al objeto en cuestión para que este pueda interactuar con otros objetos, colisionando con ellos o detectando lo cerca que están. Existen diferentes tipos básicos que tienen como malla objetos primitivos, *SphereCollider, BoxCollider, CapsuleCollider,* pero también se puede seleccionar un modelo en 3D para que se convierta a la malla con la que se puede colisionar con el componente *MeshCollider.*

- *Rigidbody:* Define cómo interactúa el objeto con el motor de físicas de *Unity*, se pueden ajustar diferentes parámetros dentro de este componente como su masa, fricción o si s afectado por la gravedad.

- *MeshRenderer*: Renderiza una malla en tres dimensiones en la posición del objeto. Al igual que con el *Collider,* existen diferentes primitivas de modelos con figuras en 3D básicas pero también se pueden seleccionar modelos en 3D importados.

- *Light*: Este componente crea una luz en la posición que marque el transform del objeto. Existen diferentes tipos de luz y cada tipo tiene un comportamiento distinto. Por ejemplo, las *PointLight* son luces que iluminan desde un punto en una esfera a su alrededor, luego las *SpotLight* que necesitan un punto y una dirección en la que proyectan un cono de luz y por último las *DirectionalLights* que sólo necesitan una dirección y crean una luz ambiente en todos los puntos de la escena. Cada tipo de luz es completamente personalizable pudiendo cambiar el color, la intensidad o el radio según sea necesario.

- *Prefabs:* Los prefabs no son componentes como tal, son *GameObjects* que ya sea porque queremos copiar el objeto repetidas veces o quremos crearlo en otra escena podemos guardarlos para generarlos más tarde mediante código. Estos *prefabs* deben de estar completos, es decir, sólo pueden hacer referencia a objetos del propio prefab o a otros prefabs.

En *Unity,* se utilizan scripts escritos en el lenguaje *C#* para dar funcionalidad a los objetos de nuestro videojuego. Existe una clase que viene dada por *Unity* llamada *MonoBehaviour,* que si un script hereda de ella, se convierte en un componente que podremos añadir con facilidad a nuestros *GameObjects,* en ese momento se pueden acceder a otros componentes del mismo objeto desde dicho script. Para cada ciclo de ejecución del juego, *Unity* envía una serie de mensajes predefinidos a cada uno de estos scripts que heredan de *MonoBehaviour*, de esta manera, se puede programar métodos que se ejecuten en momentos exactos de este ciclo, veamos algunos de ellos:

- **Start:** Se llama al inicio del primer *frame* en el que el objeto está activo.

- **Update:** Este método se ejecuta una vez por ciclo.

- **OnDestroy:** Se llama cuando el objeto al que está asignado va a ser destruido, ya sea mediante código (con la llamada a *Destroy(GaameObject))*, a un cambio de escena o al cierre del propio juego.

Además, existen otros mensajes que envían los *Colliders* anexados a los propios objetos.

- **OnCollisionEnter, OnCollisionStay y OnCollisionExit:** Estos métodos se ejecutan cuando, respectivamente, un *collider* entra en contacto con el *collider* del objeto, cuando se mantiene en contacto y cuando deja de estar en contacto.

Existen muchos otros mensajes y componentes con los que trabajar pero con los mencionados anteriormente es suficiente para el desarrollo de nuestro videojuego.

Herramientas utilizadas

Ahora presentaremos las herramientas que hemos utilizado, tanto las que proporciona *Unity* como otras externas:

- **Unity Editor:** El propio entorno de trabajo que ofrece *Unity* donde se crean las escenas del juego. Incorpora un compilador para el código que hayamos creado y su interfaz es completamente moldeable. Cada una de sus ventanas ofrece una funcionalidad distinta. Ahora vamos a ver algunas de ellas:

    - Scene: Desde esta ventana se pueden controlar los objetos que tengamos en la escena activa.

    - Game: Nos ofrece una vista de cómo se vería el juego al ejecutarlo, es posible cambiar el ratio de esta ventana para adaptarlo a diferentes tipos de patallas.

    - Inspector: Nos permite ver los componentes adheridos al *GameObject* que tengamos seleccionado y así controlar sus parámetros.

    - Heriarchy: Muestra todos los objetos de la escena activa, desde esta ventana podemos controlar las relaciones de parentesco entre los *GameObjects*.

    - Project: Contiene todos los archivos y carpetas que hay en nuestro proyecto, a estos archivos se les llama *assets* y son todos los materiales que podemos utilizar en el proyecto como texturas, modelos, scripts, etc.

    - Navigation: Aquí se puede manejar la malla que utilizarán los agentes controlados por inteligencia artificial que *Unity* pone a nuestra disposición. Esta malla define las zonas por las que pueden moverse estos agentes.

- Animator: Esta es la herramienta que proporciona *Unity* para el manejo de animaciones de los objetos en las escenas. Mediante esta herramienta hemos creado las animaciones de los enemigos y el misil dirigido.

- **ShaderGraph:** En la creación de videojuegos, normalmente son necesarios diferentes tipos de materiales con los que dar texturas a nuestros objetos o hacer que interactúen con la luz de una manera específica.

  Dentro de unity, desde la actualización 2018.2.1f1, se añadió una herramienta para la creación de *shaders* llamada ShaderGraph. Esta herramienta permite crear *shaders* muy complejos de una manera sencilla mediante diagramas de flujo sin tener que programar mediante código.

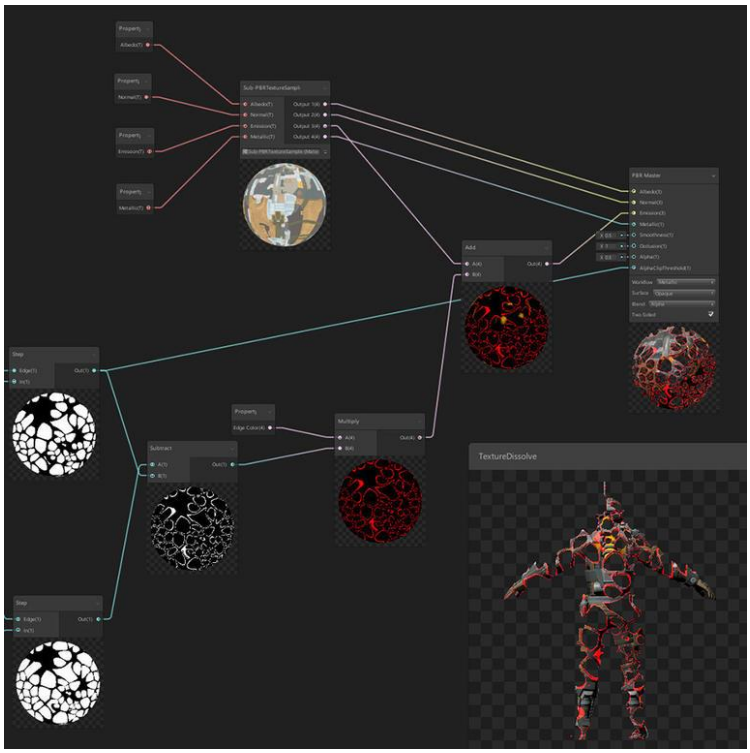


Figura 4: Interfaz ShaderGraph

- **Unity Colaborate:** Este control de versiones está incluido en el propio editor de unity y permite subir a la nube nuestro proyecto para poder controlar los cambios realizados en el proyecto de una manera muy simple y rápida.

- **Visual Studio:** Entorno de desarrollo que puede estar integrado con *Unity* para la creación de scripts. Este entorno permite la depuración del código para solucionar los errores de una manera más sencilla y rápida. Como hemos comentado anteriormente, se utiliza el lenguaje *C#,* un lenguaje orientado a objetos desarrollado por *Microsoft*.

- **Gimp (GNU Image Manipulation Program):** Es un software gratuito de manipulación de imágenes. Se ha utilizado para la creación de texturas e iconos del juego.

2.2 Herramientas de modelado

Ahora vamos a presentar algunas de las herramientas de modelado en 3D que presenta el mercado actual.

Maya

Autodesk Maya, mayormente conocido como *Maya* simplemente, es un software de creación de gráficos en 3D disponible para Linux, Mac OX y Windows. *Maya* trabaja con diferentes tipos de superficies como *NURBS,* polígonos y subdivisión de superficies.

Este software es ampliamente utilizado no sólo en el ámbito de los videojuegos sino también en el cine para la creación de efectos especiales o CGI (Computer Generated Image) en una gran cantidad de películas.

Blender

Blender es otro programa de edición, modelado y renderizado de gráficos en 3D. Este programa al contrario que Maya, es un software multiplataforma, de código abierto y ha sido siempre gratuito por lo que es mucho más accesible para estudios pequeños o estudiantes desde un punto de vista económico. Dicho esto

cabe destacar que no es un software muy intuitivo a la hora de aprender a usarlo. *Blender* está pensado para representar escenas 3D generando una imagen final en 2D, esto lo consigue mediante el uso de diferentes motores gráficos que trae por defecto.

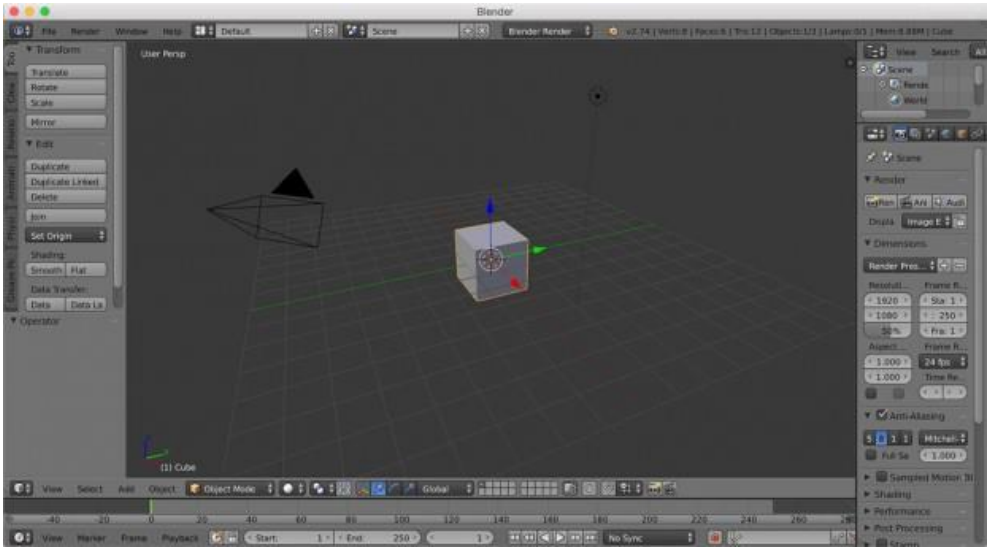


Figura 5: Interfaz de Blender

Recientemente, *Blender* ha sido actualizado a la versión 2.8, que trae numerosas mejoras, tanto en modos de renderizado como en accesibilidad a nuevos usuarios que era de lo que *Blender* pecaba mayormente.

3. Diseño del videojuego

Para documentar el diseño de *Tank Defender!*, vamos a exponer cómo se ideó y cómo se implementó hablando sobre las herramientas utilizadas.

3.1 Interfaces

El videojuego empieza con la pantalla principal con varias opciones, el botón Jugar, el botón Controles y el botón de salir.

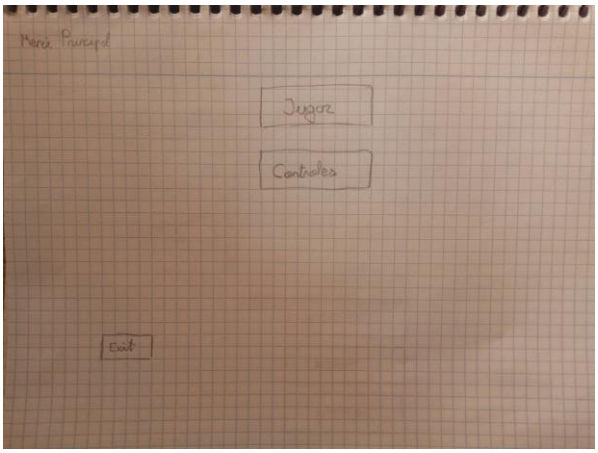


Figura 6: Boceto del menú principal

El botón Controles nos muestra las teclas del teclado y el ratón que controlan el comportamiento del tanque.

El botón de Jugar nos lleva a la escena principal donde ocurre la acción del juego. En esta escena existen diferentes interfaces:

- Pausa: Este menú se activa al pulsar la tecla Escape, se congela el tiempo en el juego y se muestra un panel donde hay 4 opciones, reanudar, para volver al juego, reiniciar para reiniciar el nivel desde la primera ronda, opciones para ajustar la dificultad y un botón para salir al menú principal.

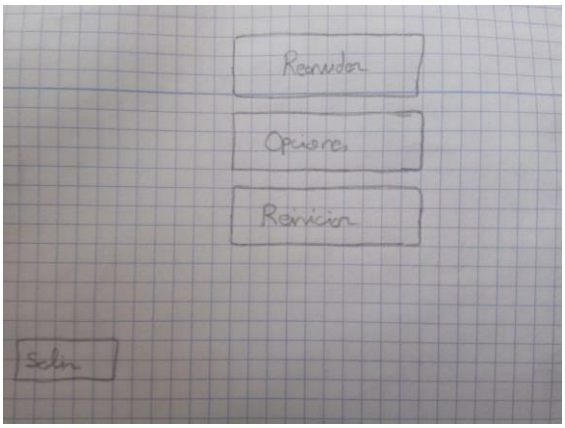


Figura 7: Boceto del menú de pausa

- HUD (Head-Up Display): El HUD tiene diferentes elementos que lo componen: la ronda en la que nos encontramos, la barra de salud del jugador, la munición del arma actual y la cantidad de puntos de compra que disponemos.

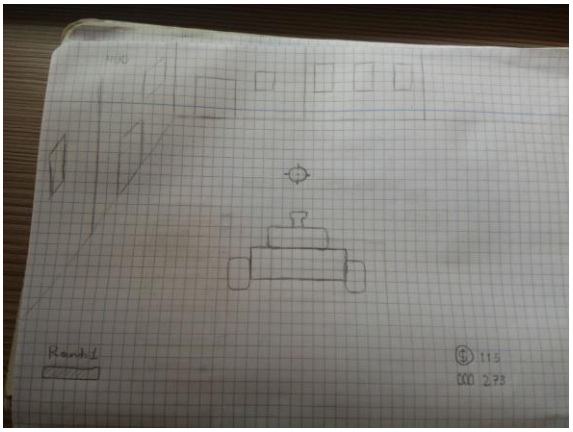


Figura 8: Boceto del HUD

- Enemigos: Los enemigos tienen una interfaz sobre sus cuerpos que mediante una barra de vida indica la salud que les queda a estos.

- Muerte: Cuando el jugador tenga 0 puntos de vida se activa esta interfaz donde se muestra la ronda a la que ha llegado, los enemigos muertos y un botón para reiniciar la partida.

- Tienda: Muestra la interfaz de la tienda donde comprar diferentes elementos cómo las mejoras de las armas y defensas. Cada apartado se

muestra mediante un Dropdown y cada elemento tiene un pequeño texto explicativo además del valor de cada elemento.

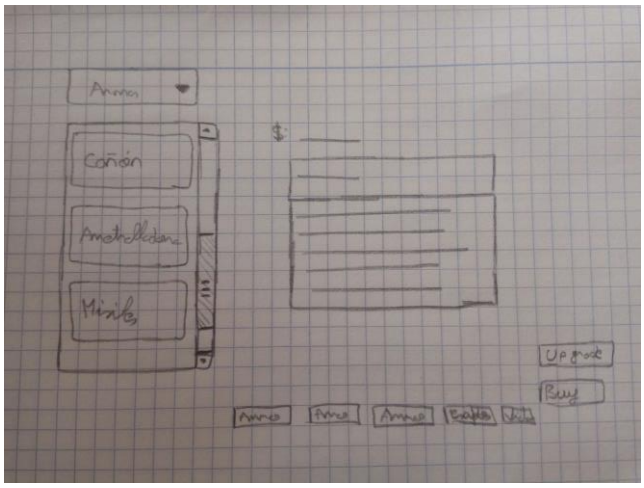


Figura 9: Boceto de la interfaz de la tienda

3.2 Jugador

El jugador controlará un tanque, mediante las teclas *W* y *S* se desplazará hacia delante y atrás, con *A* y *D*, todo el tanque rotará sobre su eje vertical y mediante el movimiento del ratón se controlará la torreta. Con el *botón izquierdo del ratón* y con el botón *Control* la torreta que tengamos equipada disparará su proyectil asignado.

Figura 10: Modelo del tanque desde Unity

Además, al pasar cerca de la tienda, si pulsamos el botón *E,* se activará la interfaz de dicha tienda.

3.3 Enemigos

Existen dos tipos de enemigos:

- Arañas: Son el enemigo básico, se tratan de unas arañas que aparecen por rondas y persiguen al jugador. Atacan cuerpo a cuerpo, hacen 15 puntos de daño y tienen 200 puntos de vida.

- Drones: Estos enemigos sobrevuelan el entorno y en vez de perseguir al jugador, le apuntan y disparan a distancia, tienen una cadencia de disparo alta pero sus proyectiles producen un daño menor haciendo solamente 5 puntos de daño por disparo y tienen 200 puntos de salud.

3.4 Cambios en la implementación

El desarrollo de videojuegos es un proceso iterativo, con lo que en numerosas ocasiones el producto final no es lo que se tenía en mente desde un principio, ya sea por desechar ideas, adoptar algunas nuevas o por la propia implementación de estas. En este caso no ha sido diferente y algunos elementos no son exactamente lo que se había planteado.

Por ejemplo, en un primer momento se tenía pensado implementar un tercer tipo de enemigo, uno pequeño y rápido que explotaba al acercarse al jugador, pero al testear el juego se observó que no casaba bien con el dinamismo de los demás elementos con lo que se decidió desechar esta idea. También cabe destacar que en un principio la tienda solamente estaría activa entre rondas pero finalmente se decidió ponerla en un espacio en la escena para darle una mayor capacidad de decisión al jugador.

4. Implementación

En este apartado vamos a hablar de cómo se ha implementado el videojuego. Para este proyecto se ha utilizado el lenguaje *C#,* que pese a su leve peso en la grado, es similar a otros lenguajes orientados a objetos si estudiados en este.
Como entorno de programación se ha utilizado Visual Studio ya que puede ser integrado en *Unity.* Para el editor de *Unity,* se ha utilizado la versión 2018.3.0f2 ya que es la primera que incluye la herramienta *ShaderGraph* tal y como comentábamos con anterioridad.
La implementación se realizó en diferentes fases, primero se creó el tanque que controla el jugador, luego los enemigos y el control de la escena y más tarde la tienda. Finalmente, se añadieron modelos y texturas a los diferentes objetos de la escena.

4.1 Tank_controller.cs

Este fue el primer script implementado. Controla el movimiento del tanque, tanto su desplazamiento como la rotación de la torreta además de la vida, los escudos y la munición de este. El movimiento frontal se ha implementado mediante la llamada a la función Transform.Translate (float x, float y, float z) en el método *Update()* que permite mover un *GameObject* por la escena.

```
Mover el objeto hacia delante una cantidad de espacio = forwardSpeed * Tiempo transcurrido desde
el último frame * (+1 si se pulsa W ó -1 si se pulsa S)

Rotar el objeto en el eje Y una cantidad de grados = rotationSpeed * Tiempo transcurrido desde
el último frame * (+1 si se pulsa A ó -1 si se pulsa D)
```

ForwardSpeed es una variable float que indicamos con anterioridad para controlar la velocidad del tanque, Input.GetAxis() es la función que proporciona *Unity* para detectar el input del usuario, en este caso tanto las teclas W,S como las flechas Arriba, Abajo. Además, se ha utilizado Time.deltaTime para que el movimiento no dependa de la fluidez del *framerate* del equipo del usuario.

Para la rotación horizontal de la torreta se ha utilizado un método similar sólo que usando la función Transform.Rotate() que rota un GameObject en la escena.

```
155 - mousePos = (0, movimiento del ratón en eje X, 0)
156 - rotamos la torreta en el eje y una cantidad de grados = mousePos.y * Tiempo transcurrido desde el
      último frame
157 - Si la torreta tiene eje X entonces
158 -   mientras MousePos.x > 180 entonces mousePos.x -= 360
159 -   mousePos.x -= movimiento del ratón en Y * sensibilidad * tiempo transcurrido desde el último frame
160 -   Limitamos mousePos.x entre 30 y -30
161 -   Asignamos mousePos a la rotación de la torreta
```

Hemos separado la rotación horizontal de la vertical puesto que horizontalmente se permite un rotación de 360º, pero para la rotación vertical necesitamos algún tipo de límite ya que no queremos que la torreta atraviese el cuerpo del propio tanque. Es por ello que se ha utilizado el método Clamp() de la librería *Mathf*, este método permite limitar un valor entre unos valores máximos y mínimos.

En este script también se controlan diferentes aspectos del tanque como la vida restante, los escudos o la munición de las armas. Estas variables se controlan mediante floats en el caso de la vida y los escudos y mediante un array de enteros para el caso de la munición donde cada índice representa un tipo distinto de munición, una para cada torreta.

También cabe destacar que al tener un rigidbody, el propio tanque está afectado por el motor de físicas de Unity, con lo que si se encontraba en una superficie elevada y se desplazaba hacia su límite, el tanque caía de lado y no era capaz de llegar a una posición plana que es como debería de estar. Es por esto que se han añadido unas líneas de código para mantener el tanque estable mientras baja de una posición elevada.

```
119 - Si existe algún objeto debajo del tanque
120 -   Rotamos en tanque poco a poco hasta que su vector Y sea igual a la normal de la superficie
        detectada
```

Tal y como se muestra en la imagen, a cada frame se lanza un rayo hacia abajo desde el origen del tanque para detectar si existe una superficie debajo de este, si existe la rotación del tanque se rota hasta ser la misma que la normal en el punto impactado para así estar siempre perpendicular al plano donde se apoya, esto sumado a la opción de los rigidbody, que permite “congelar” un eje de rotación para que no sea afectado por las físicas, hace posible que el tanque no vuelque al caer. También existe el método Die() que es llamado cuando se recibe daño y la salud del jugador es menor que cero. En este método se activa la pantalla de muerte y se muestran las rondas conseguidas y los enemigos abatidos.

4.2 Turret.cs

Para el control de los disparos del jugador se ha creado un script llamado *Turret.cs*, en este script se guarda información de dónde instanciar los proyectiles de las torretas, que proyectiles instanciar y la tasa de disparos de la torreta en cuestión.

Puesto que es el script anterior el que controla la munición, se necesita una referencia a un *GameObject* que tenga dicho script para así cada vez que disparamos actualizar la munición restante.

Como cada torreta tiene un comportamiento ligeramente distinto se ha decidido crear para cada una de ellas un script que herede de este, con lo que para el control de los disparos tendremos *Turret_01.cs*, *Turret_02.cs* y *Turret_03.cs* que controlan cada una de las torretas a nuestra disposición. En el método *Update()* de cada uno de estos scripts comprobamos que ha pasado suficiente tiempo desde el último disparo y si tenemos pulsado el botón izquierdo del ratón o el botón Ctrl entonces instanciamos el objeto asignado al proyectil y este ya se encargará de desplazarse hasta que entre en contacto con algún objeto, además, utilizamos el método de Tank_controler.cs Recharge() para recargar -1 balas y así indicar que hemos utilizado una unidad de munición.

```
14 - Si se ha pulsado el botón de disparo y queda munición del arma actual y ha pasado el tiempo suficiente
     entonces
15 -     Reseteamos el tiempo transcurrido
16 -     Instanciamos el proyectil
17 -     recargamos(-1) balas
```

Al instanciar el proyectil, se le pasa como parámetro la rotación con la que aparece con lo que ya sale propulsado en la dirección del cañón hacia donde estemos apuntando. Esto es diferente en la torreta lanzamisiles puesto que los proyectiles de esta funcionan de manera diferente como veremos a continuación.

4.3 Bullet_script.cs

Este es un pequeño script que tienen los proyectiles para que se desplacen hacia delante simulando así que han sido disparados. En el método Update(), usan la función translate() para desplazarse al igual que con el tanque pero esta vez de manera continua. Además, en tienen el método OnTriggerEnter(Collider collision), este

método detecta cuando algún collider colisiona con el collider del propio proyectil y así dependiendo de contra que haya chocado realiza una función u otra.

```
24 - Cuando se colisione con algún Collider
25 - Si es un enemigo o el propio jugador entonces
26 -    se le quita vida correspondiente al daño del proyectil
27 - Si es otra cosa que no sea una bala entonces
         se destruye este objeto
```

Como tanto el jugador como algunos enemigos pueden disparar proyectiles si alguna de las balas impacta contra algún enemigo o el propio tanque este llamará a la función take_dmg() que le quitará la vida correspondiente al daño del propio proyectil. En cambio, si la bala golpea cualquier otra cosa que no sea otra bala esta simplemente se destruye para ahorrar espacio.

4.4 Missile_Script.cs

En el caso de la torreta lanza-misiles, los proyectiles no salen directamente hacia delante sino que suben hacia arriba describiendo un arco hacia el objetivo marcado como un círculo en el suelo que sigue la rotación de la torreta. Para a creación de este efecto, se ha utilizado el componente Animator, que permite crear animaciones para los objetos de la escena. En este caso se ha creado una animación que simplemente sube el misil hacia arriba una vez se ha instanciado, una vez allí se gira hacia el objetivo y avanza hacia este.

```
20 - A cada frame
21 -    Si se ha acabado la animación entonces
22 -        Rotamos el objeto hacia la posición del objetivo
23 -        Movemos el objeto hacia la posición del objetivo
```

Una vez el misil ha entrado en contacto con algún objeto, detecta todos los objetos a su alrededor mediante la función Physics.OverlapSphere(transform.position,radius). Esta función crea un esfera que detecta todos los colliders a su alrededor en un radio dado, si alguno de estos colliders tiene asociado un objeto del tipo “enemigo” entonces le hace la cantidad de daño asociado al misil y una vez hecho esto, el misil se destruye a sí mismo.

```
32 - Cuando el misil choca contra un collider
33 -    Se detectan todos los collider cercanos
34 -    Para cada uno de estos collider
35 -        Si es un enemigo entonces
36 -            Se le quita vida
37 -    Se destruye el objeto del misil
```

También cabe destacar que el objetivo del misil es un objeto hijo de la propia torreta, con lo que al desplazarla y girarla este se desplaza con ella. Esto puede ocasionar que en algún momento el objetivo se encuentre dentro de alguna pared u otro objeto, cosa que no queremos que ocurra por lo que se ha creado también

un pequeño script *Mark_Script.cs* que mediante un *Raycast* desde arriba de la posición del objetivo choca con cualquier collider que encuentre y transforma la posición del objetivo a esta nueva posición. Así si chocha con el suelo el objetivo se mantiene en su sitio pero si colisiona con un objeto que esté más alto el objetivo se posiciona sobre este.

4.5 Enemy_Controller.cs

Este script es el que controla el comportamiento de los enemigos básicos. Para que su indicador de salud restante esté siempre visible, en el método Update() se gira el canvas asociado para que mire siempre hacia donde está la cámara.

```
39 - A cada frame
40 -    Se va rotando el canvas poco a poco hacia la posición de la cámara
```

Por otra parte, para el control del movimiento se ha utilizado el paquete de componentes *NavMesh*. Mediante este paquete de componentes podemos definir un espacio por dónde queremos que se desplacen nuestros objetos, y añadiendo el componente *NavMeshAgent* a estos objetos podemos indicar una posición y estos se desplazan automáticamente hacia este objetivo evitando las zonas marcadas como no traspasables. Lo bueno de este componente es que puede realizar esta tarea de marcar zonas automáticamente dados los Colliders presentes en la escena.

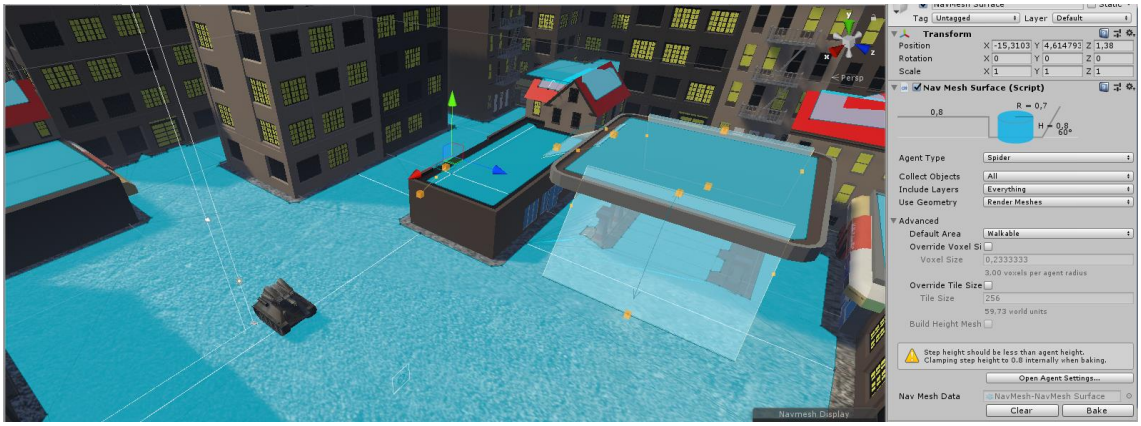
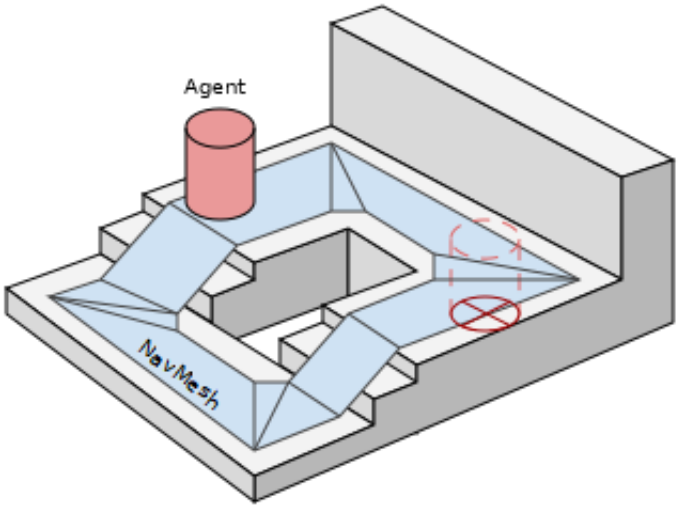


Figura 11: Superficies caminables creadas por NavMeshSurface

Como podemos ver, dados los parámetros del agente seleccionado el componente NavMeshSurface automáticamente genera las superficies por donde los propios agentes pueden desplazarse.



En el paquete NavMesh, los agentes son descritos como cilindros, y las áreas que se crean son todas aquellas zonas en las que este cilindro puede desplazarse sin colisionar con nada. Estas áreas se guardan como polígonos convexos, y al guardarse de esta forma se sabe con seguridad que entre dos puntos del mismo polígono es seguro desplazarse puesto que forman parte del área traspasable. Además se guarda qué polígonos son vecinos entre sí, para que dados dos puntos, el inicio y el fin, se vayan recorriendo los polígonos vecinos hasta encontrar el punto objetivo. Para encontrar el camino de un punto al otro Unity utiliza el algoritmo A*, algoritmo ampliamente usado en el campo del desarrollo de videojuegos por su eficiencia y precisión. Este algoritmo forma parte de los algoritmos de búsqueda en grafos de tipo heurístico. Usa una función de evaluación f(n) = g(n) + h'(n), donde g(n) representa el coste hasta el nodo n y h'(n) representa el valor heurístico desde el nodo n hasta el nodo objetivo.

Para el control del agente hemos creado una subrutina. Una subrutina no es más que una función como otra pero que puede detener su ejecución y continuar en el frame siguiente para así no tener que ejecutar todo su código en un mismo ciclo y poder ejecutar código a intervalos de tiempo estables o realizar cambios de manera gradual. Por ejemplo en la subrutina `Reposition()` tenemos un bucle que se ejecuta cada segundo e indica al agente del enemigo si desplazarse hasta la posición del tanque mediante `agent.SetDestination(tank.transform.position);` o si ya está lo suficientemente cerca, atacarle.

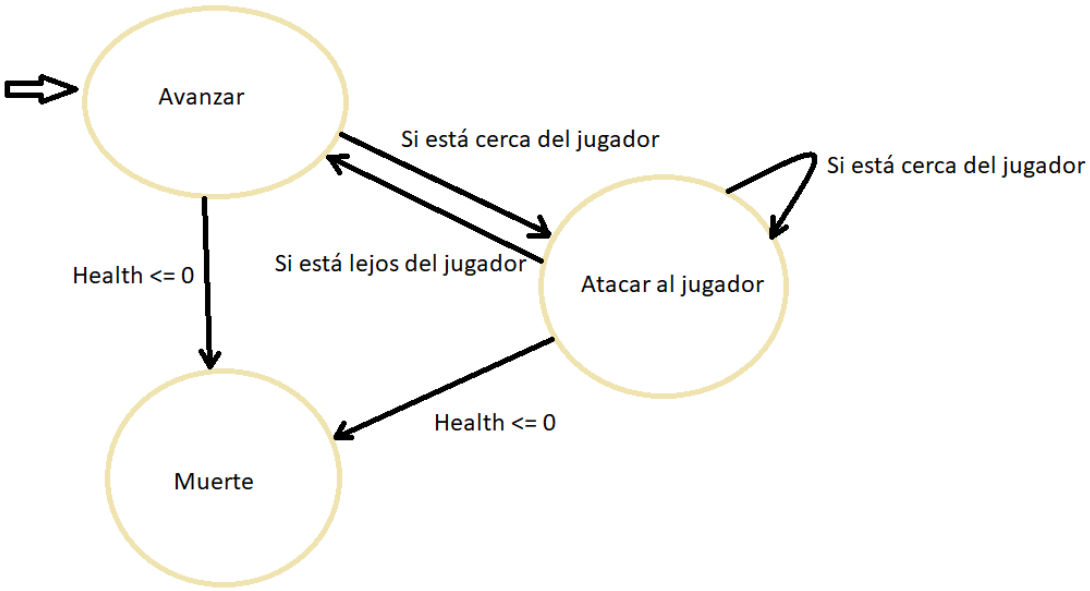


Figura 12: Diagrama de flujo de la lógica de los enemigos

También, al relizar las acciones de avanzar y atacar se interactúa con algunos valores del componente *Animator*, esto es para activar las transiciones entre los diferentes estados de la máquina de estados que controla este componente para que el enemigo realice las animaciones correspondientes.

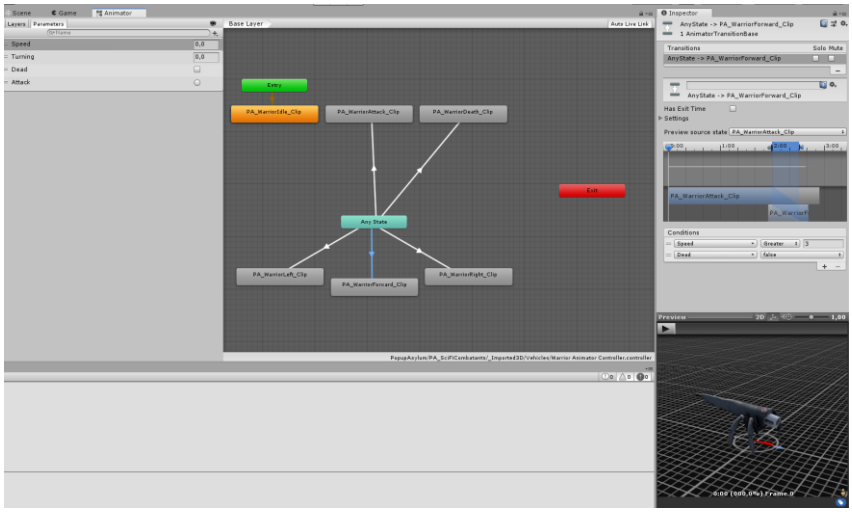


Figura 13: Diagrama de estados del componente Animator del enemigo

Para el paso a la animación de moverse hacia delante, se puede realizar desde cualquier otro estado y es necesario que se cumplan ciertas condiciones.

También cabe destacar que mediante shadergraph hemos creado un material especial que ajustando el parámetro de alpha junto con una función de ruido conseguimos la ilusión de que el enemigo aparece de la nada como si de un holograma se tratase.

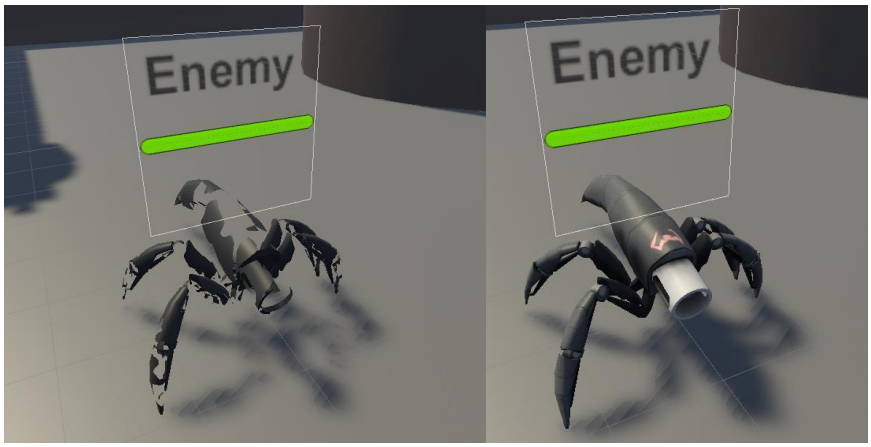


Figura 14: Transición entre los diferentes materiales de los enemigos

En ShaderGraph se pueden definir propiedades que pueden ser modificadas mediante código, como el canal alpha que define la transparencia del objeto o el color del propio material. Esto usado conjuntamente con algunos elementos que ya proporciona la propia herramienta como la textura generada mediante ruido perlin nos permite crear la ilusión de que el enemigo aparece de la nada de una manera fluida.

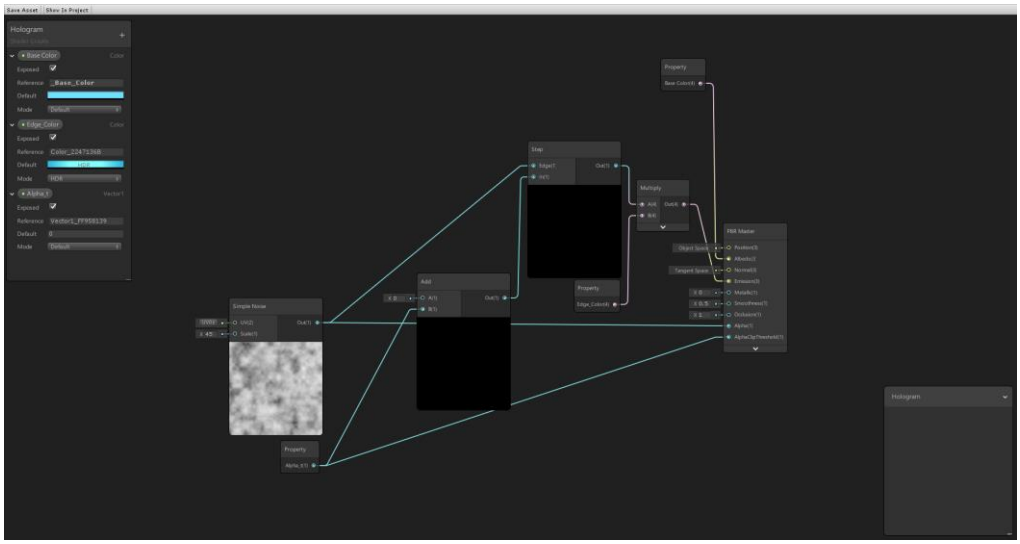


Figura 15: Diagrama del shader usado en los enemigos en ShderGraprh

4.6 Drone_Script.cs

Este script controla los enemigos del tipo Dron. Al igual que con los enemigos básicos, estos también tienen una barra de vida sobre sus cuerpos indicando su salud restante que apunta a la cámara en todo momento. Por otra parte, los drones, en vez de seguir al jugador realizan el mismo patrón sobrevolando el entorno, esto se ha conseguido mediante animaciones creadas con el componente animation.

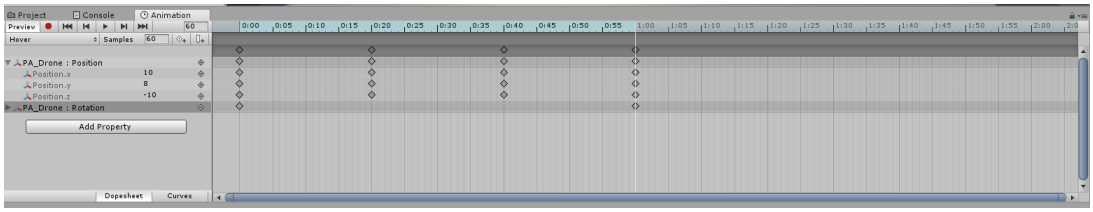


Figura 16: Componente Animation de los drones

Este enemigo además, apunta en todo momento al jugador, esto se consigue mediante la función Quaternion.LookRotation(tank.transform.position - this.transform.position, new Vector3(0, 1, 0)); que transforma la dirección entre el dron y el jugador a un Quaternion que es como unity guarda las rotaciones de los objetos. Además de esto, también disparará un proyectil en la dirección que esté apuntando, es decir hacia el jugador. Este disparo instancia un prefab creado con anterioridad que usa el script Bullet_Script.cs.

4.7 Turret_follower.cs

Para la implementación de una pequeña torreta que siguiera al jugador hemos creado este script que en su Update() utiliza la función Vector3.Slerp() para interpolar entre dos vectores, su posición actual y la posición del jugador con un pequeño offset. De esta manera, el objeto al que esté incorporado se desplazará de una manera suave hasta un punto cerca del jugador. Luego realiza un Physics.OverlapSphere() desde la posición del jugador, con esto se recogen todos los colliders dentro de un radio desde una posición dada. Si alguno de estos colliders es un enemigo la torreta apuntará a este objeto cambiando su rotación y le hará daño llamando a la función take_dmg() de ese enemigo. Para mostrar visualmente

este efecto, se ha creado un shader que junto a un LineRenderer hace el efecto de un láser.

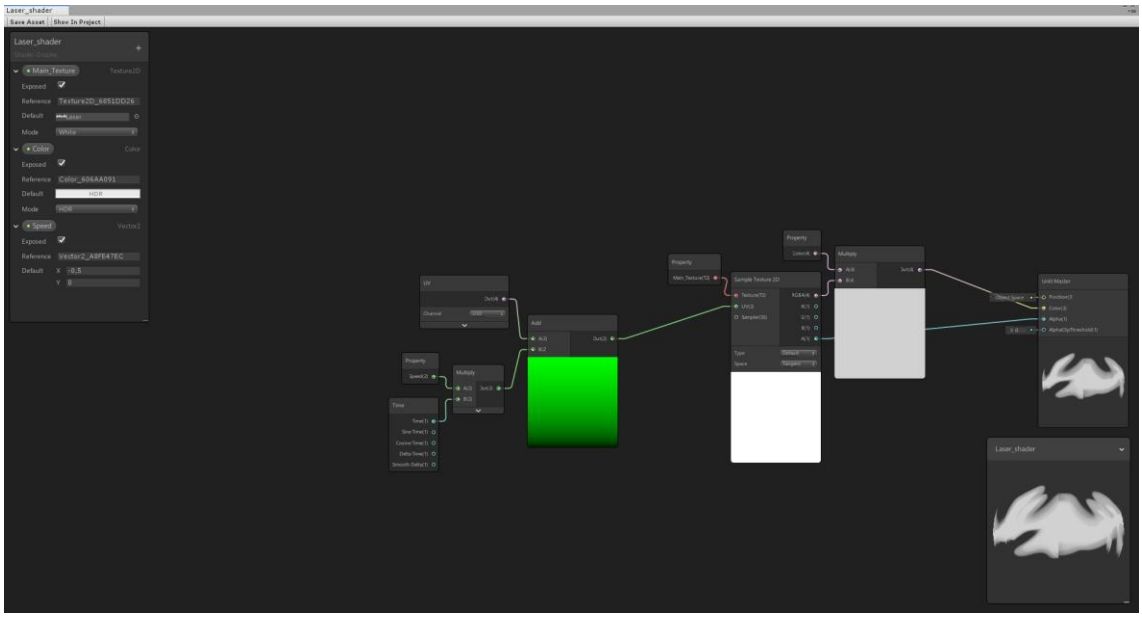


Figura 17: Diagrama del shader del láser en ShaderGraph

Este shader utiliza una textura que hemos creado mediante GIMP, y la desplaza en un eje a una velocidad dada por el parámetro speed para dar la ilusión de que este láser está en movimiento.

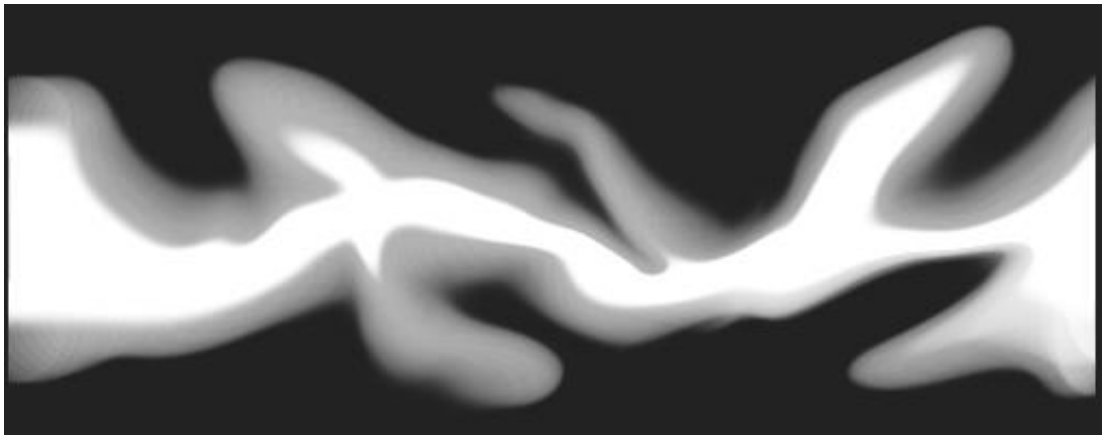


Figura 18: Imagen creada en GIMP para la textura del láser

Luego se le aplica este shader al material del LineRenderer, que simplemente renderiza una textura en 2D desde un punto A a un punto B con una anchura dada.

4.8 Scene_Controller.cs

Mediante este script se maneja la lógica de las oleadas de enemigos además de los puntos conseguidos, los enemigos muertos hasta el momento y la dificultad de estos. Para esto, en la función Start() llamamos a la subrutina Enemy_Spawn() que mediante un bucle while() hasta que el jugador haya muerto, calcula la cantidad de enemigos que deben de aparecer en función de la ronda en la que nos encontremos y los va instanciando cada segundo en un punto de aparición aleatorio del mapeado. Luego mientras hayan enemigos aún vivos la subrutina espera para empezar la siguiente ronda. Una vez han muerto todos los enemigos, deja un margen de 10 segundos para que el jugador pueda reposicionarse o ir a la tienda. En este script también se guarda la dificultad actual seleccionada por el jugador, todos los enemigos que aparezcan en adelante tendrán su cantidad de vida máxima y daño aumentados según el nivel escogido.

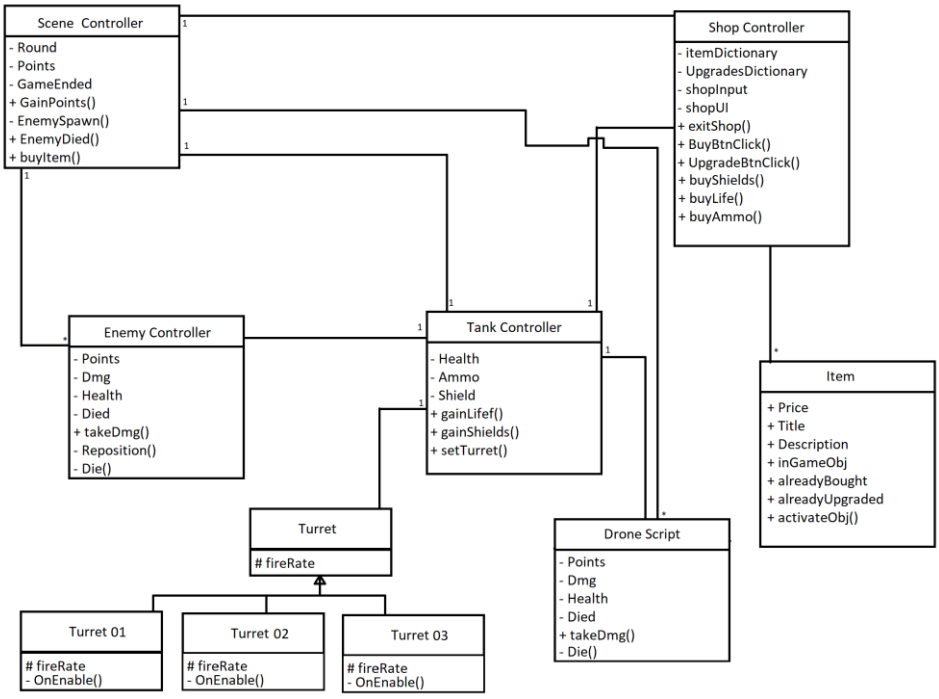


Figura 19: Diagrama UML de las clases relacionadas con el control de la escena

4.9 Pause_script.cs

Con este script manejamos el menú de pausa. Para ello en el Update() comprobamos que si se ha pulsado la tecla escape, si se ha pulsado y el menú no

estaba activado, lo activa, en caso contrario, es decir si estaba activo entonces el juego vuelve a la pantalla principal. Se decidió hacerse de esta manera para que el jugador mediante unas pocas pulsaciones de teclado pudiese salir rápidamente del juego. Además, al activar el menú se pausa el tiempo del juego mediante la instrucción `Time.timeScale = 0.0f;` esto modifica la velocidad a la que se reproducen los frames, con lo que si la ponemos a 0 paramos el tiempo. Esto no impide que la interfaz se bloquee o que se registren las entradas de teclado, solamente congela el tiempo para la física, el desplazamiento de objetos y las animaciones. Con esto el jugador puede controlar las opciones tranquilamente sin tener que estar pendiente de los enemigos. También cabe destacar que en este script tenemos diferentes métodos que se ejecutan al pulsar los botones que contiene este menú, como el botón Reiniciar, que pone el timescale a 1 y vuelve a cargar la escena con lo que se reinicia el juego desde cero, el botón de opciones que abre el panel de opciones y el botón de salir que cierra el juego mediante la instrucción `Application.Quit(0);`.

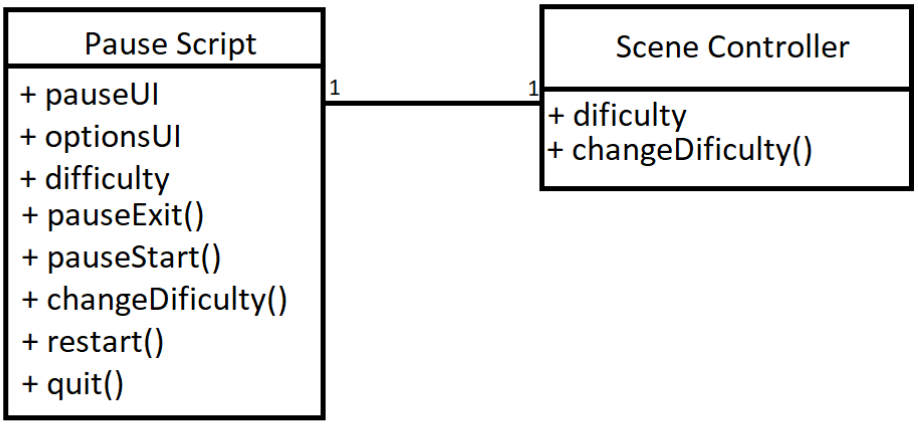


Figura 20: Diagrama UML control del menú de pausa

4.10 Shop_controller.cs

Mediante esta clase se manejan los objetos de la tienda, cuáles de ellos hemos adquirido y el precio de estos. Para ello hemos creado otra clase llamada Item que guarda los valores de los elementos de la tienda como el nombre, el precio, la

descripción y los objetos que activa al ser comprado. En la propia clase Shop_controller tenemos un array de Item en el que guardamos cada objeto que queremos mostrar. Para mostrarlo, en la interfaz de la tienda tenemos un dropdown que nos muestra diferentes secciones, armas, defensas y mejoras. Cada una de estas secciones dispone de diferentes cuadros de texto y de diferentes botones, cada botón al pulsarlo, llama a la función onClickBtn() de Shop_controller y le pasa un identificador i que carga la posición i-ésima del array de ítems y muestra en los cuadros de texto mencionados los valores del ítem. Luego, en el botón de comprar, se quitan los puntos correspondientes al ítem comprado y se activan los objetos que tenga asignado. Además, el GameObject al que esté asignado esta clase, tiene un Collider del tipo trigger para que cuando el jugador pase cerca, este pueda pulsar la tecla E y activar la interfaz de la tienda, con esto también se pausa el tiempo al igual que con el menú de pausa.

Al comprar un ítem, se activa el objeto de la escena que referencia además de activar la compra de mejoras de dicho ítem. Estas mejoras dependen del objeto seleccionado, si se trata de una torreta aumentan su daño y si es un escudo o la coraza aumentan la capacidad máxima.

Además de las armas y escudos también es posible recuperar salud, recargar los escudos y recargar munición.

4.11 AutoTransparent.cs

Al crear el mapeado final, se advirtió que la cámara, al ser espacios más cerrados, podía quedar metida dentro de los edificios y bloquear la vista al jugador. En ese momento se crearon diferentes scripts para solucionar este problema. Uno de ellos, AutoTransparent, que cambia gradualmente el color de cada uno de los MeshRenderer asociados a los objetos hijos del objeto actual a un color transparente para permitir ver a través de estos y si ya son transparentes los devuelve a su color original. Una vez cambiado el color, se elimina dicho componente para ahorrar coste computacional.

Por otro lado, el script ClearSight.cs, mediante un SphereCast() detecta todos los objetos entre la cámara y el propio tanque y para cada uno de estos objetos, si tiene un MeshRenderer, se le añade el componente AutoTransparent para que cambie su color a uno transparente.

4.12 Gráficos

Por la parte gráfica, debido a la falta de experiencia en el modelado en 3D ya la falta de presupuesto para el proyecto, se han descargado diferentes modelos y texturas de internet para dar un acabado un poco más profesional al juego.

Para el propio tanque, se han usado dos modelos para el cuerpo y 3 para las distintas torretas.

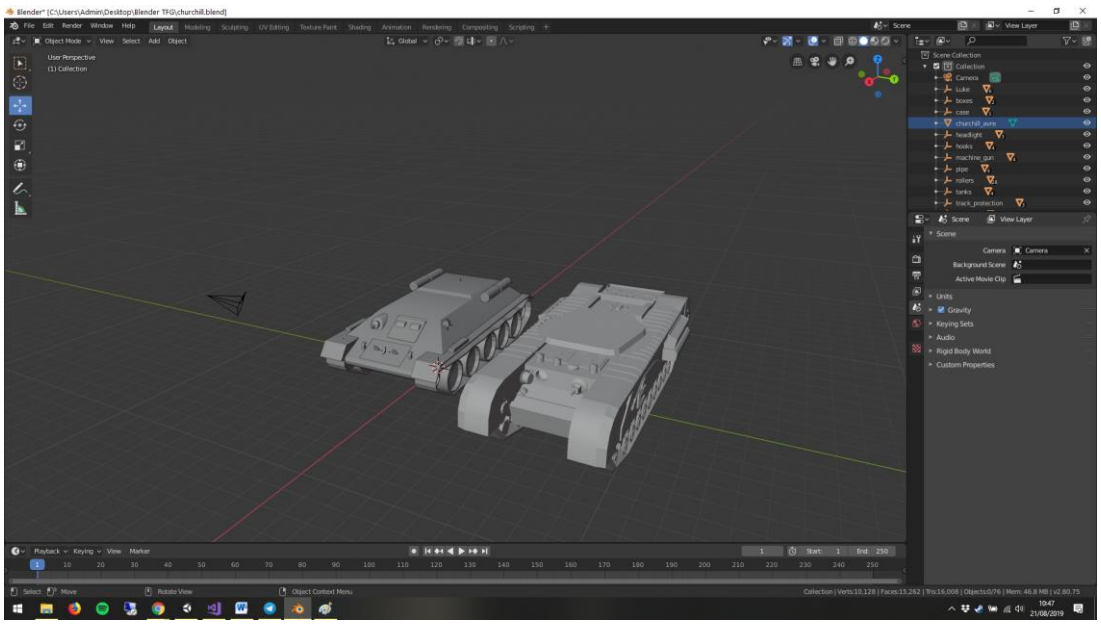


Figura 21: Modelos de los cuerpos del tanque en Blender

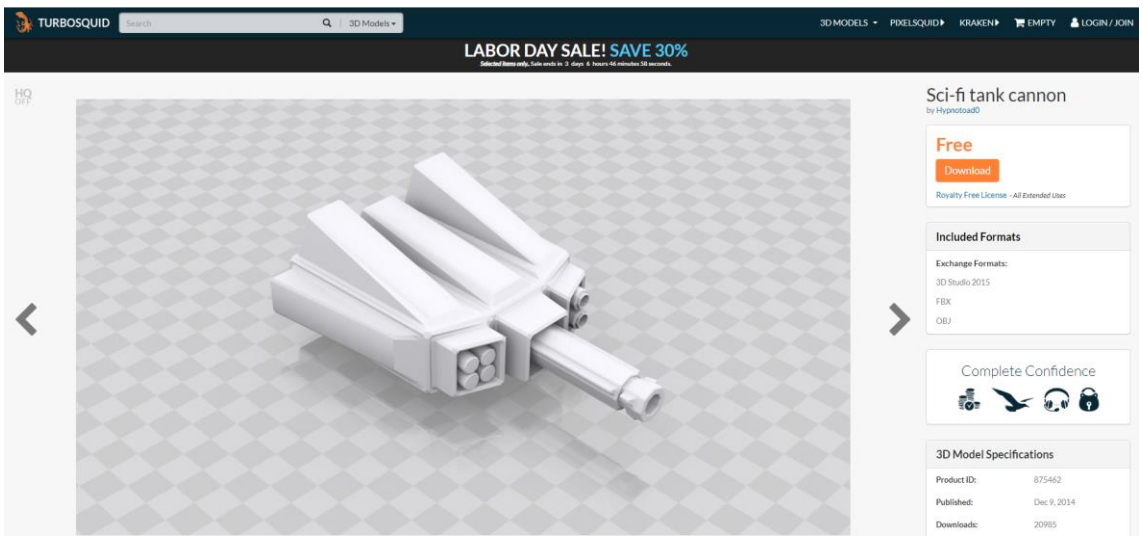


Figura 22: Modelo del cañón del tanque

Los modelos de los tanques y de las torretas se pueden encontrar en www.cadnav.com y www.turbosquid.com.

Tanto para los enemigos como para los edificios, se han utilizado assets disponibles en la propia tienda de unity.

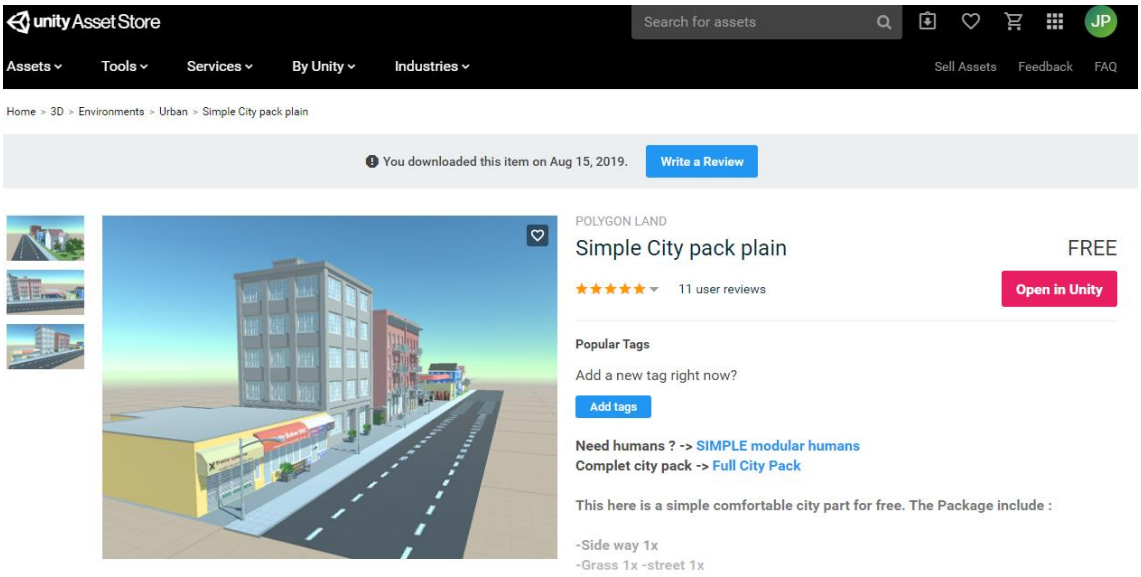


Figura 23: Asset usado para los edificios

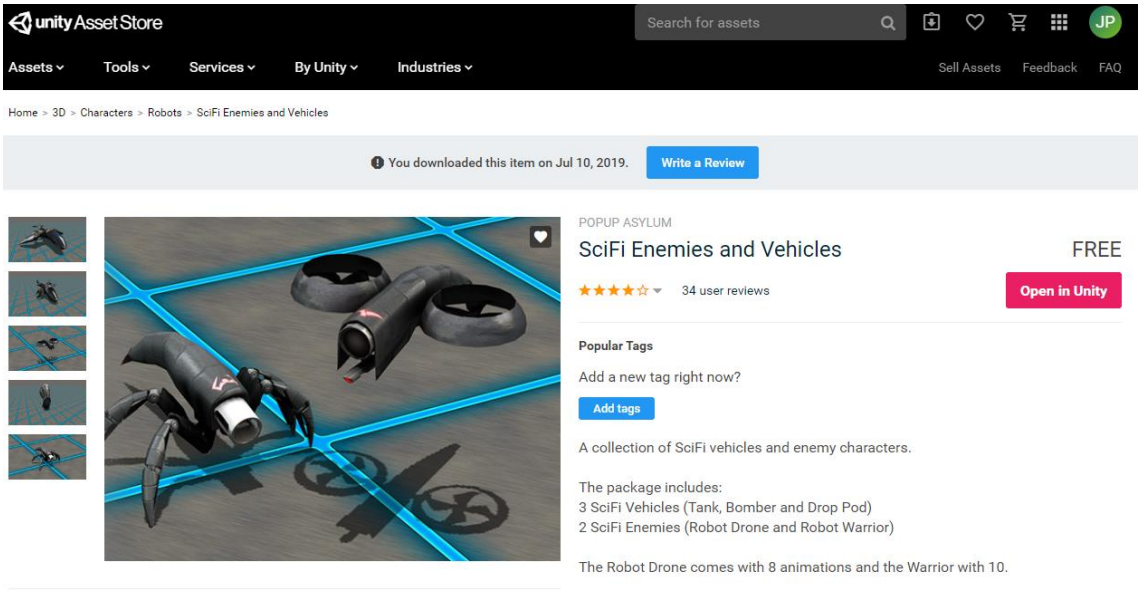


Figura 24: Asset usado para los enemigos

5. Pruebas

Para comprobar que todos los elementos funcionaban como era debido, se realizaron distintas pruebas.

Durante estas pruebas, se observó un fallo de diseño al no tener en cuenta los proyectiles que fallaban y al no golpear a ningún objetivo, estos continuaban existiendo hasta que se acabase la partida. Esto pese a no observarse ningún impacto significativo en el rendimiento, se decidió solucionar dando un tiempo de vida máximo a los proyectiles. También se solucionó un error que al cambiar el tamaño del cuerpo del tanque al comprar el segundo escudo, el tamaño del collider también aumenta con lo que había ocasiones en las que los propios disparos del jugador dañaban a este. Simplemente de alejó un poco la distancia a la que aparecen dichos disparos y se consiguió subsanar este error. A parte, se observó que en ocasiones era un poco difícil apuntar a los enemigos. Es por ello que se añadió una cruceta mediante una imagen hija de la propia torreta con lo que sigue el movimiento de esta. Incluso con este añadido aún resulta un poco difícil apuntar a los enemigos muy lejanos pero como el mapeado no es lo suficientemente grande como para que esto pase no se ha intentado mejorar esta situación, quizá si en un futuro se implementan nuevos escenarios, sí que se debería atacar este ámbito. Además, al crear lo que sería el mapeado final se encontró que la cámara en algunas ocasiones atravesaba los edificios y obstaculizaba la vista. Para solucionar esto se creó un script que mediante Raycasts, detectaba los objetos entre la cámara y el jugador y les daba un material de color transparente para poder ver a través de estos.

Los enemigos no han mostrado ningún comportamiento inesperado, tanto las arañas como los drones. También cabe destacar que, los enemigos, al aparecer cada pocos segundos en diferentes puntos del mapa, acaban normalmente por rodear al jugador. Esto es un comportamiento que no está programado como tal pero que se observa al tener un número suficiente de enemigos. A parte de esto, no mostraron ningún tipo de comportamiento erróneo.

La tienda funciona correctamente permitiendo comprar solamente si se tienen los puntos necesarios.

El juego fue probado por distintos amigos y funcionó correctamente en todos los dispositivos probados.

6. Resultados

En este apartado se muestra el resultado final del videojuego Tank Defender!

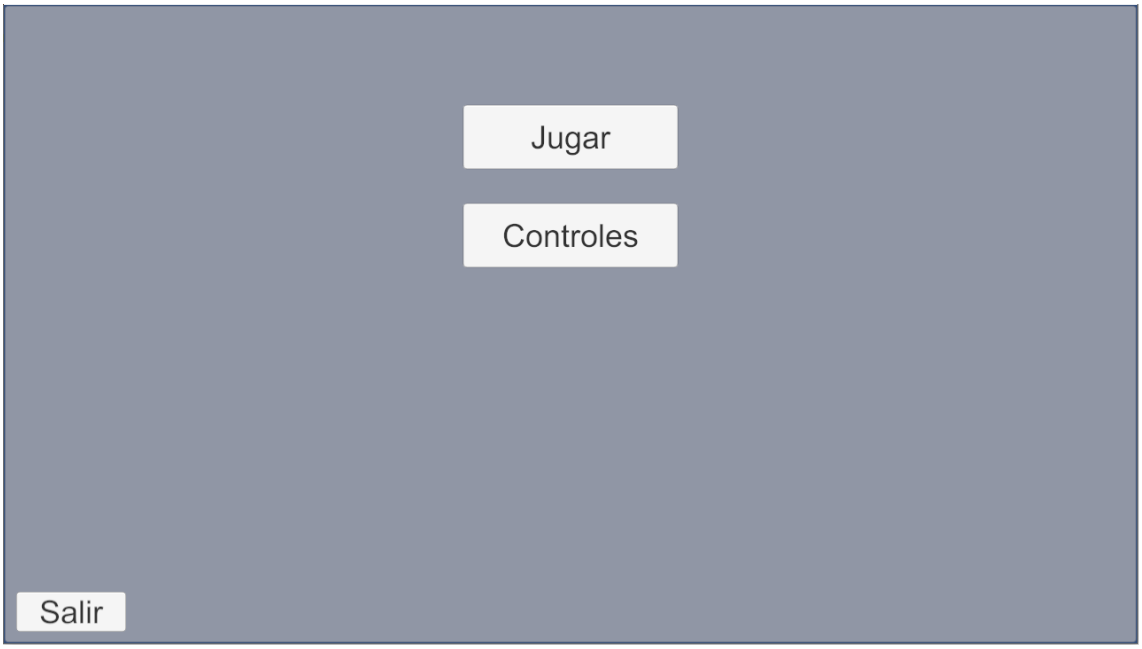


Figura 25: Menú prinicpal

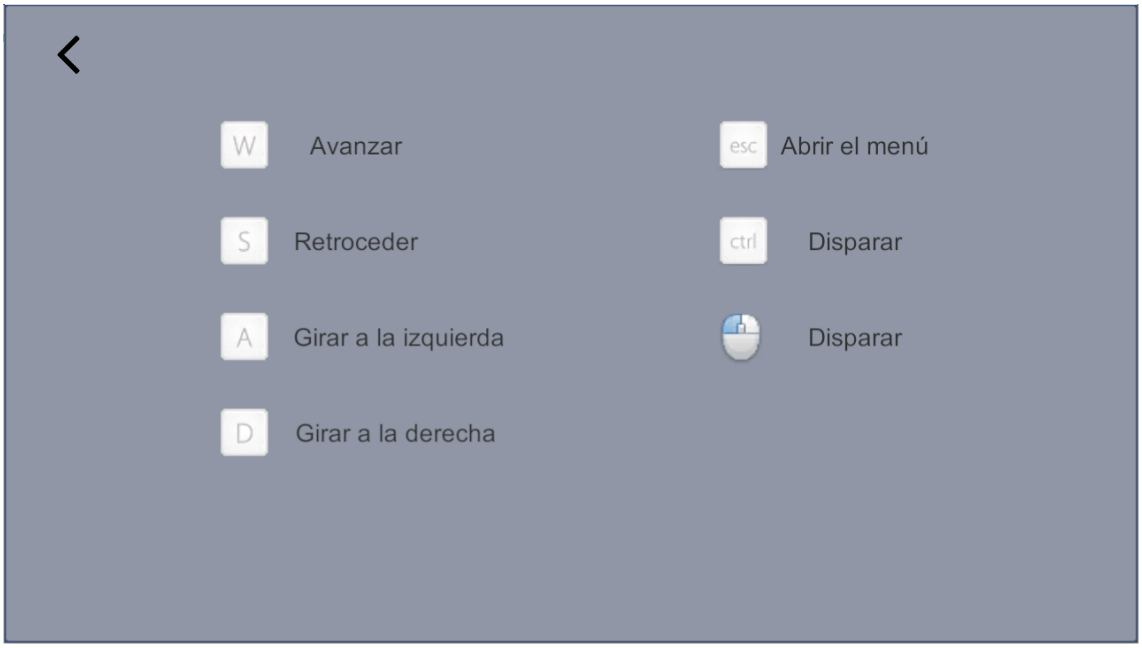


Figura 26: Controles

Estas dos imágenes presentan el menú principal con los controles del juego.

Figura 27: Ametralladora disparando

[Figura 21] Muestra la ametralladora disparando, es un arma de disparo rápido pero de poco daño. Es bastante buena contra los enemigos aéreos.

Figura 28: Tanque con el segundo cuerpo ya comprado

[Figura 22] Muestra el tanque con el segundo cuerpo y la ametralladora. Este cuerpo es más grande que el primero y por tanto más fácil de golpear para los enemigos pero también nos aporta un escudo mucho mayor.

Figura 29: Tanque con lanzamisiles apuntando a un grupo de enemigos

[Figura 23] Los misiles son lentos pero hacen un gran daño en área. Son efectivos contra grandes grupos de enemigos pero inútiles contra los drones.

Figura 30: Dron disparando al jugador

[Figura 24] Los drones aparecen cada 5 rondas y sobrevuelan el mapa disparando al jugador. Su movimiento sigue un patrón predefinido en forma de cuña que varía según a donde esté apuntando.

7. Conclusiones

El desarrollo de un videojuego es una tarea ardua que pone a prueba muchos de los conocimientos del desarrollador, desde la inteligencia artificial hasta las estructuras de datos pasando por redes solamente en el campo de la informática sin entrar en otros ámbitos como el diseño gráfico o la psicología.

Durante el desarrollo de este proyecto se han asentado muchos conocimientos a la hora de desarrollar videojuegos, tanto adquiridos durante la carrera como otros nuevos y de ahora en adelante será mucho más fácil afrontar nuevos proyectos.

El resultado final ha sido suficiente puesto que se han llevado a cabo los objetivos que se tenían en mente para el proyecto puesto que el videojuego dispone de:

- Un tanque controlado por el jugador
- Diferentes torretas con distintos tipos de jugabilidad
- Dos tipos de enemigos
- Una dificultad adecuada

Cabe destacar que incluso durante la fase de pruebas, jugar resultaba satisfactorio y desafiante con lo que cumple el cometido de todo videojuego de entretener al jugador.

8. Trabajos futuros

Aún queda bastante margen a la hora de mejorar el juego, algunos de los elementos que añadir en un futuro podrían ser los siguientes:

- Más enemigos, niveles y armas. Aunque se trate de un juego de escaso tamaño, se puede ampliar en diferentes frentes, tanto añadiendo nuevos escenarios más grandes y con mayor variedad como implementando otro tipo de enemigos.
- Sonidos: Debido a la falta de tiempo no se ha podido añadir sonidos que mejorasen la experiencia de juego. Los sonidos y la música suelen pasar desapercibidos para muchos jugadores pero que se echan en falta cuando no están presentes.
- Gráficos: De realizar una ampliación a mayor escala del proyecto, sería interesante contar con expertos en el campo del diseño para mejorar visualmente el juego y darle una estética más profesional.

9. Bibliografía

[1] Wikipedia – Videojuego.

https://es.wikipedia.org/wiki/Videojuego

[2] CryEngine.

https://www.cryengine.com/

[3] UnrealEngine.

https://www.unrealengine.com

[4] Unity.

https://unity.com/es

[5] Matt, S (2015). Unity 5.x Cookbook.

https://www.amazon.es/Unity-5-x-Cookbook-Matt-Smith/dp/1784391360

[6] Blender.

https://www.blender.org/

[7] Maya.

https://www.autodesk.es/products/maya/overview

[8] Marc, L (2017). BLENDER. Curso práctico.

https://www.amazon.es/BLENDER-Curso-pr%C3%A1ctico-MARC-LIDON/dp/849964712X

[9] Wikipedia – Shader

https://es.wikipedia.org/wiki/Sombreador

[10] Wikipedia – Ray Casting

https://es.wikipedia.org/wiki/Ray_casting

[11] GIMP

http://www.gimp.org.es/

[12] Wikipedia – Quaternion

https://es.wikipedia.org/wiki/Cuaterni%C3%B3n

[13] Youtube – Unity Shader Graph – Laser Beam Tutorial

https://www.youtube.com/watch?v=mGd3nYXj1Oc&feature=youtu.be

[14] Youtube – HOLOGRAM using Unity Shader Graph

https://www.youtube.com/watch?v=KGGB5LFEejg

[15] Tom, W. (2018). Newzoo. Mobile Revenues Account for More Than 50% of the Global Games Market as It Reaches $137.9 Billion in 2018

https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/

[16] Unity Documentation - Inner workings of the Navigation System

https://docs.unity3d.com/Manual/nav-InnerWorkings.html

[17] Wikipedia – Algoritmo A*

https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*