Universidad Pontificia Comillas

ICAI

iMAT Zombies

Paradigmas y técnicas de programación

Lydia Ruiz Martínez, Jorge Vančo Sampedro

PROYECTO FINAL

Curso 2024-2025

Índice

1. Propuesta de videojuego

1.1. Nombre

iMAT Zombies

1.2. Género

Survival/ Shooter

1.3. Descripción

Este juego de supervivenvia sitúa al jugador en un ambiente desafiante, donde hay zombies que acechan por todos lados. El jugador ha de enfrentarse a ellos, zombies que aparecen en grupos y se mueven hacia el jugador, que necesita frenar su avance y destruirlos usando para ello distintas estrategias y armas.

A medida que el jugador va eliminando zombies, consigue puntos y dinero. Puede usar el dinero en un cofre para comprar un arma, granadas y munición. Además, en el suelo hay balas para que pueda recuperar la munición usada. Por un lado, el jugador emplea las armar para aniquilar a los zombies. Por otra parte, los zombies intentan acabar con la vida del jugador.

El código del proyecto se puede encontrar en el siguiente repositorio de GitHub: Repositorio del proyecto

1.4. Objetivo

El principal objetivo es sobrevivir durante el mayor número de oleadas posibles de zombies, a los cuales hay que eliminar y así ir acumulando puntos que pueden usarse para adquirir armas y munición mediante el cofre. Hay que gestionar las armas adecuadas y otros recursos de forma estratégica para cada situación.

2. Diseño de arquitectura

2.1. Diagrama UML

El enlace al diagrama UML en el siguiente: Diagrama UML

La arquitectura propuesta se ha dividido en tres capas principales: **Manager Layer**, **Scene Manager** y **Agent Manager**. Esta organización permite una clara separación de responsabilidades y mejora tanto la escalabilidad como el mantenimiento del sistema.

- **Manager Layer:**
  - En esta capa se encuentra el **GameManager**, que actúa como el núcleo de control del sistema, supervisando y coordinando los diferentes componentes del juego.
  - El *GameManager* tiene una relación de **agregación** con los siguientes elementos:
    - **DataManager**: Encargado de gestionar los datos persistentes del juego.
    - **RoundManager**: Controla la lógica de las rondas en el juego.
    - **MenuManager**: Gestiona las interfaces de menú. Este componente, mediante una relación de **composición**, contiene tanto el menú principal como el menú de compra de items.
  - **SceneManager**: Vinculado por una relación de composición con las distintas escenas del juego, como:
    - **GameScene**: La escena principal donde ocurre la acción del juego.
    - **MainMenuScene**: El menú inicial donde se encuentran las opciones de inicio y configuración del juego como la dificultad.
    - **BuyingMenu**: La interfaz de compra de objetos y otras mejoras para el jugador.
    - **GameOverScene**: La escena que aparece cuando el usuario ha perdido el juego y que da paso a la escena del menú principal.
  - **UIManager**: Maneja los elementos de la interfaz de usuario y mantiene una relación de **composición** con componentes como la barra de vida, el contador de rondas y el inventario.

  Los *managers* se implementarán como **singletons**, permitiendo acceder a sus datos y métodos desde cualquier parte del sistema sin la necesidad de instanciar cada *manager* en otras clases.

- **Scene Layer:**
  - En esta capa se encuentran componentes como el **Inventory**, que gestiona los objetos y mejoras disponibles para el personaje.

- La **Interfaz IUsable** es una interfaz que se utiliza para definir los objetos que pueden ser utilizados dentro del juego. Esta interfaz mantiene una relación de **agregación** con **Inventory** y **BuyingChest**. El **Inventory** contiene los objetos que el jugador puede usar, mientras que el **BuyingChest** se encarga de la gestión de compra de nuevos objetos durante el juego.
- La interfaz **IUsable** es implementada por tres tipos de **Weapons**: **Mele**, **FireWeapon** y **Grenade**. Cada tipo de arma tiene características únicas: las **Mele** tienen un alcance, las **FireWeapon** tienen una capacidad, y las **Grenade** tienen un radio de daño determinado que afecta a los enemigos en un área.
- Los spawners que se encargan de hacer que aparezcan zombies y munición a lo largo del mapa.
- También se ha creado la interfaz IThrowable, que implementa la granada, pues esta se puede lanzar.

- **Agent Layer:**
  - En esta capa se encuentra el **Character**, el cual tiene una relación de **asociación** con los eventos del juego. El **Character** también tiene una relación de **composición** con el **Inventory**, ya que el personaje mantiene su propio inventario de objetos.
  - Los **Zombies** se dividen en tres tipos diferentes: **FastZombie**, **NormalZombie** y **SlowZombie**. Cada tipo de zombie posee características únicas, tales como velocidad, daño y alcance, lo que determina su comportamiento y dificultad en el juego. Para la creación de estos zombies, se ha implementado un patrón de diseño **Factory**, lo que permite generar los distintos tipos de zombis de manera eficiente sin la necesidad de instanciarlos manualmente en cada lugar del código. Cabe destacar que, aunque inicialmente se consideró el uso del patrón de diseño **Flyweight** para la creación de los zombies, finalmente se optó por la implementación de la **Factory**. El patrón **Flyweight** es más comúnmente utilizado en sistemas que gestionan un gran número de objetos similares, como partículas (miles), mientras que en nuestro caso solo existen tres tipos de zombis con características bien definidas.

2.2. Historias de usuario

- Como jugador, quiero matar a los zombies con las armas que tengo.
  - **Objetivo:** El jugador necesita usar sus armas para defenderse de los zombies.
  - **Requisitos:**

    - Implementar una mecánica para que las balas sean recogidas al pasar cerca de ellas.
    - Añadir una función de ataque personalizada para cada arma (disparar, explotar o apuñalar).
    - Asignar una cantidad de daño a cada arma.
  - **Tareas:**
    - Crear el sistema de interacción con objetos en el suelo.
    - Desarrollar las mecánicas de combate de las armas.

- Como jugador, quiero recuperar vida al acabar cada ronda para tener más probabilidad de sobrevivir en cada ola.
  - **Objetivo:** Tras acabar la ronda, se le suma algo de vida al jugador para que tenga más posibilidades de sobrevivir la siguiente ola.
  - **Requisitos:**
    - Implementar una mecánica para que se sume algo de vida tras cada ronda.
    - Mostrar la vida del jugador en la pantalla.
  - **Tareas:**
    - Hacer el LifeBar de la interfaz.
    - Desarrollar las mecánicas de la vida del jugador.

- Como juego, quiero tener un cofre con armas para que el jugador pueda comprar arsenal.
  - **Objetivo:** El jugador puede comprar armas en un cofre especial.
  - **Requisitos:**
    - El cofre contiene distintas armas que pueden ser compradas por el jugador.
    - Cada arma debe tener un precio.
  - **Tareas:**
    - Crear cofre de armas.
    - Asignar a cada arma un precio.
    - Desarrollar mecánica de compra de armas.

- Como jugador, quiero tener dinero para comprar arsenal en el cofre.
  - **Objetivo:** El jugador debe poder conseguir dinero al eliminar zombies o de otra forma similar para luego poder comprar mejores armas y equipamiento.
  - **Requisitos:**

    - El jugador debe poder conseguir dinero al eliminar zombies.
    - El jugador puede usar ese dinero para comprar equipamiento.
  - **Tareas:**
    - Implementar la mecánica de conseguir dinero.
    - Permitir al jugador gastarse ese dinero en el cofre.

- Como jugador, quiero tener puntos para marcar mi progreso.
  - **Objetivo:** El jugador necesita implemetar un sistema de puntación para mostrar su progreso a lo largo del juego.
  - **Requisitos:**
    - Los puntos deben acumularse en función de las acciones realizadas en el juego, como eliminar zombies o completar rondas.
    - Los puntos deben ser visibles al usuario, permitiendo al jugador saber cuántos puntos ha acumulado en todo momento.
  - **Tareas:**
    - Visualizar el total de puntos acumulados en la interfaz de usuario, asegurándose de que el jugador pueda ver la cantidad de puntos disponibles en todo momento.

- Como zombie, quiero perseguir al jugador para atacarle y eliminarlo.
  - **Objetivo:** El zombie debe perseguir al jugador de manera autónoma, usando un sistema que permita detectar dónde está el jugador para poder atacarlo y hacer que pierda vidas.
  - **Requisitos:**
    - Los zombies deben detectar la posición del jugador y moverse hacia él de forma eficiente, utilizando IA para determinar su trayectoria
    - Los zombies deben tener diferentes comportamientos según su tipo (por ejemplo, velocidad de movimiento, daño y vida).
    - Los zombies deben atacar al jugador cuando estén lo suficientemente cerca, infligiendo daño al personaje del jugador.
    - Los zombies deben tener un sistema de vida para ser eliminados si reciben suficiente daño por parte del jugador.
  - **Tareas:**
    - Crear el sistema de interacción con objetos en el suelo.
    - Desarrollar las mecánicas de combate de las armas.

- Como partida, quiero tener un contador de rondas para contabilizar las rondas jugadas, de modo que el progreso de la partida se pueda medir.
  - **Objetivo:** Proporcionar una forma de seguir el progreso de la partida, ajustando la dinámica de los enemigos a medida que avanzan las rondas.

- **Requisitos:**
  - El contador de rondas debe actualizarse automáticamente al final de cada ronda.
  - El juego debe aumentar su dificultad a medida que se completen las rondas (aumentando la velocidad o el número de zombies por ejemplo).
- **Tareas:**
  - Implementar el **RoundManager** para gestionar el avance de las rondas.
  - implementar el RoundCounter para que el **UIManager** muestre en la escenYa la ronda actual.

- Como arma de fuego, quiero tener balas para poder disparar.
  - **Objetivo:** Las armas de fuego tienen que tener balas que puedan ser disparadas.
  - **Requisitos:**
    - Cada arma de fuego debe poder disparar balas.
  - **Tareas:**
    - Implementar un BulletPool para las balas.
    - Permitir que el arma dispare las balas.

3. Implementación

El desarrollo del juego comenzó con la implementación del jugador y de los zombies. En ambos casos surgieron problemas con la gravedad, a pesar de que tenían un collider y un rigidbody. Es por esto que se tuvo que implementar una gravedad propia que fuese aumentando la velocidad negativa en el eje Y para que cayesen. Además, se les tuvo que añadir un GroundCheck para que cuando tocasen el terreno se dejase de añadir esa velocidad y se quedasen en el juego. Esto tampoco llegaba a funcionar del todo bien, pues de vez en cuando algún zombie atravesaba el suelo y caía indefinidamente, lo que impedía seguir jugando y avanzar en las rondas. Sin embargo, al implementar el lanzamiendo de las granadas, estas no tuvieron ningún tipo de problema con la gravedad. Tras hacer las granadas, se volvió a intentar hacer lo mismo con los zombies, y se descubrió que el problema era la opción Apply Root Motion que tienen las animaciones del zombie.

Se implementó un spawner para hacer que los zombies apareciesen de forma aleatoria por todo el mapa. Sin embargo, al ser un mapa irregular, se tuvo que usar raycasting para lanzar un rayo desde una altura determinada hasta que tocase el terreno, y hacer que el zombie aparezca a esa altura. También se tuvo que hacer lo mismo para hacer que aparezcan balas por el mapa para recoger, por lo que se creó la clase abstracta Spawner, de la que heredan BulletSpawner y ZombieSpawner.

Se han implementado interfaces como IUsable e IThrowable para favorecer la generalidad del código y la facilidad de expansión. Por ejemplo, la interfaz IUsable permite que en un futuro se puedan añadir vendas para que el jugador se pueda curar, o distintos tipos de armas.

La gran mayoría de los problemas enfrentados resultan del conocimiento limitado que se tiene de Unity, pues es el primer juego que hacemos, pero con ayuda de internet y tutoriales se han podido implementar la gran cantidad de ideas que teníamos en un principio. Por ejemplo, a la hora de implementar el arma de fuego, resultaba imposible hacer que las balas fueran consistentemente hacia donde miraba el jugador. Esto era porque en un principio se le añadia una velocidad a la bola, pero no se hacía que se moviera en el Update de la bola (cosa que no sabíamos todavía), por lo que se cambió a un impulso, pero la bola caía muy rápido, por lo que se subió la fuerza del impulso. Esto sí que hacía que el arma disparase, pero había veces que las balas se disparaban a diferentes sitios. Por lo que se implemento raycasting, como el usado para el spawner, para hacer que se disparase en la dirección del centro de la pantalla. Pero el problema seguía estando. Al final, se descubrió la parte de hacer que se mueva la bala en el Update, por lo que se volvió al modelo del principio, pero se dejó el raycasting porque facilitaba apuntar, porque era justo al centro de la pantalla, en vez de un poco al lateral, de donde salen las balas de la punta del arma.

Otro gran problema, ha sido implementar la funcionalidad de los menús. Hacer que aparezcan correctamente, no interaccionen con los elementos de las escenas cuando se están usando, pero que sí puedan modificar el estado del juego ha sido un suplicio.

Sin embargo, el mayor reto ha sido mantener un código limpio y estructurado, haciendo uso de patrones y los principios SOLID, con el trabajo en parejas. En muchas ocasiones se hacía todo lo posible por juntar implementaciones de ambos y que funcionasen correctamente, olvidándonos de usar patrones, lo que llevaba a repetición de código y malas prácticas. Sobre todo porque al tener conocimientos limitados de Unity se priorizaba conseguir que funcionase a toda costa. Es por esto que se ha decidido no implementar algunas ideas que teníamos al principio para poder centrarse en limpiar el código.

3.1. Patrones usados

Los principales patrones usados han sido el Singleton para el GameManager y el BulletPool. Haciendo esto permitía acceder al GameManager siempre, y el arma podía acceder al BulletPool, asegurándose de que no se instanciaban más balas de las que se han creado.

Además, se ha implementado una factoría de zombies para poder crear los distintos tipos de zombies.

También se ha hecho uso del patrón observer. Sobre todo para la implementación de la lógica de la UI, para ejecutar ciertas funciones al empezar y acabar las rondas y al eliminar los zombies.

4. El juego

Cuando empieza la partida, el jugador solo cuenta con un cuchillo que puede usar para eliminar a los zombies cuando están cerca de él. Al elminiarlos, consigue puntos y dinero, el cual puede usar para comprar un arma de fuego, balas y granadas. Las granadas se pueden lanzar y explotan tras un tiempo, inflingiendo daño a todos los elementos que están a su alrededor a una cierta distancia, incluido el propio jugador, por lo que tiene que tener mucho cuidado. Cuando elimina a todos los zombies, se pasa a la siguiente ronda tras 10 segundos, en los que el jugador puede aprovechar para recargar o comprar armas. Además, se le suma algo de vida al jugador para que pueda sobrevivir a la siguiente, se empieza curando 5 puntos de vida y en cada ronda se aumenta este valor en otros 5 puntos hasta llegar a 50, el máximo de este valor.

En cada ronda aparecen más zombies, por lo que es cada vez más complicado, pero también puede conseguir más dinero y aparecen también más balas, para que sea una experiencia equilibrada.

Conforme las rondas van pasando, se aumenta el dinero y los puntos que se dan al eliminar a los zombies.

5. Guía de uso

El jugador se puede mover con las teclas WASD y saltar con el espacio. Al darle click izquierdo puede usar las armas. Puede acceder al cofre cuando está junto a él y presiona la tecla E. Las balas que aparecen en el suelo se recogen al pasar por encima de ellas. Una vez se han comprado granadas o el arma, se puede cambiar presionando la tecla C.

Cuando se usa el arma de fuego, se puede presionar la R para recargar. También se recarga automáticamente si no se tienen balas en el cargador y se intenta disparar.