UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Surge Survivor
Un videojuego tipo shoot them up 3D, casual de supervivencia creado con el framework Unity DOTS

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Ruiz Guirao, Antonio

Tutor/a: Muñoz García, Adolfo

CURSO ACADÉMICO: 2023/2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València
Escuela Técnica Superior de Ingeniería Informática

Surge Survivor

Un videojuego tipo shoot 'em up en 3D, casual de supervivencia creado con Unity DOTS

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Ruiz Guirao, Antonio

Tutor: Muñoz García, Adolfo

Curso 2023-2024

Resumen

Este TFG presenta el desarrollo de un videojuego estilo 'shoot 'em up' en 3D, utilizando el framework Unity DOTS. Para mejorar el rendimiento y la eficiencia en comparación con el sistema tradicional de GameObjects. El objetivo es optimizar y aumentar el rendimiento en videojuegos con muchas entidades simultáneas.

En Surge Survivor, el jugador debe sobrevivir el mayor tiempo posible. Para lograr esto, se ha implementado un sistema de oleadas sucesivas de enemigos, donde el jugador debe resistir y obtener la mayor puntuación posible. La dificultad se ajusta dinámicamente según las oleadas, y el jugador adquiere mejoras al subir de nivel.

El trabajo busca demostrar la implementación de sistemas y Jobs para gestionar entidades con Unity DOTS, además de crear una experiencia de juego compleja y desafiante. Asimismo, pretende proporcionar una base sólida para futuros proyectos que utilicen este framework.

Palabras clave: DOTS, Burst Compiler, ECS, GameObject, Jobs, Rendimiento, Shoot 'em up, Unity, Entitad, FPS

Resum

Este TFG presenta el desenvolupament d'un videojoc estil '*shoot '*em *up' en 3D, utilitzant el framework Unity DOTS, per a millorar el rendiment i l'eficiència en comparació amb el sistema tradicional de GameObjects. L'objectiu és optimitzar i augmentar el rendiment en videojocs amb moltes entitats simultànies.

En Surge Survivor, el jugador ha de sobreviure el major temps possible. Per a aconseguir això, s'ha implementat un sistema d'onades successives d'enemics, on el jugador ha de resistir i obtindre la major puntuació possible. La dificultat s'ajusta dinàmicament segons les onades, i el jugador adquirix millores en pujar de nivell.

El treball busca demostrar la implementació de sistemes i Jobs per a gestionar entitats amb Unity DOTS, a més de crear una experiència de joc complexa i desafiador. Així mateix, pretén proporcionar una base sòlida per a futurs projectes que utilitzen este framework.

Paraules clau: DOTS, Burst Compiler, ECS, GameObject, Jobs, Rendiment, Shoot 'em up, Unity, Entitat, FPS

Abstract

This TFG presents the development of a 3D "shoot 'em up" style videogame, using the Unity DOTS framework, to improve performance and efficiency compared to the traditional GameObjects system. The goal is to optimize and increase performance in games with many simultaneous entities.

In Surge Survivor, the player must survive as long as possible. To achieve this, a system of successive waves of enemies has been implemented, where the player must resist and obtain the highest possible score. The difficulty is dynamically adjusted according to the waves, and the player acquires upgrades as he levels up.

The work seeks to demonstrate the implementation of systems and Jobs to manage entities with Unity DOTS, in addition to creating a complex and challenging game experience. It also aims to provide a solid foundation for future projects using this framework.

Key words: DOTS, Burst Compiler, ECS, GameObject, Jobs, Performance, Shoot 'em up, Unity, Entity, FPS

CAPÍTULO 1
Introducción

1.1 Motivación

La motivación de este proyecto surge, desde mi infancia donde los videojuegos fueron una parte importante en mi día a día. Este interés se mantuvo a lo largo de los años, influyendo en mis decisiones académicas. Cuando empecé mis estudios de Ingeniería Informática, tenía como objetivo participar en el desarrollo de Videojuegos.

Gracias a la asignatura Desarrollo de Videojuegos 3D, pude adquirir un base sólida respecto a la creación de videojuegos. En este contexto descubrí el framework Unity DOTS (Data-Oriented Technology Stack) para Unity [3]. Debido a esto, decidí desarrollar un videojuego que emplease este framework por sus características de alto rendimiento.

Para ello mi proyecto consiste en un videojuego de tipo "shoot 'em up", en el que el jugador se enfrenta a oleadas de enemigos. Donde deberá sobrevivir sin que lo derroten. Esta elección no solo responde a mis gustos personales, sino también a una manera de mostrar cómo DOTS puede mejorar la eficiencia y rendimiento en el desarrollo de videojuegos.

1.2 Objetivos

Este proyecto no solo busca crear un videojuego entretenido y desafiante, sino también explorar y comparar diferentes enfoques de programación dentro de Unity. Específicamente el framework Unity DOTS (Data-Oriented Technology Stack), comparado al sistema tradicional de GameObjects [8]. A continuación se detallan los objetivos de este TFG.

- Desarrollar un videojuego shoot 'em up utilizando Unity
  - Crear un juego de oleadas de enemigos, donde el jugador se enfrente a diversos tipos de enemigos.
  - Diseñar mecánicas de juego que incrementen la dificultad de manera progresiva.
  - Implementar diferentes tipos de enemigos con comportamientos únicos.
- Implementar mejoras progresivas para el personaje del jugador
  - Desarrollar un sistema de puntos de experiencia y niveles.
  - Implementar mejoras que aumenten la velocidad, el poder de disparo y la salud del personaje.

- Evaluar el rendimiento utilizando Unity DOTS
  - Implementar el videojuego utilizando el framework DOTS (Data-Oriented Technology Stack) de Unity.
  - Optimizar el manejo de entidades y componentes para maximizar el rendimiento.
  - Analizar el uso de memoria y la eficiencia del procesamiento de datos.
- Comparar el rendimiento con el uso de GameObjects por defecto en Unity
  - Realizar comparativas de rendimiento entre el uso de Unity DOTS y el sistema por defecto basado en GameObjects.
  - Medir los FPS, uso de memoria y tiempos de carga en ambas implementaciones.
  - Documentar las diferencias en la facilidad de implementación y mantenimiento del código.
- Documentar el proceso de desarrollo y los resultados
  - Crear una documentación detallada del proceso de desarrollo del videojuego.
  - Escribir un diario de desarrollo que incluya las decisiones de diseño y las iteraciones del proyecto.
  - Redactar un análisis crítico de los resultados obtenidos en las pruebas de rendimiento.

1.3 Estructura del Trabajo

En este documento, se estructuran varios capítulos. El primer capítulo introduce el proyecto, explicando su contexto y objetivos. A continuación de este apartado, se revisa el estado del arte de los videojuegos 'shoot 'em up' y las tecnologías de desarrollo relevantes, como Unity DOTS y Unreal Engine.

En el tercer capítulo se describe el framework Unity DOTS, detallando sus componentes y características. Asimismo, se especifica la solución propuesta del trabajo, junto al diagrama de Gantt para describir las fases del proyecto y el plan a seguir del proyecto.

Siguiendo con el cuarto capítulo, se aborda el diseño del videojuego. Cubriendo la arquitectura de los sistemas implicados, así como el bucle jugable del videojuego, además del escenario donde ocurre la acción dentro del juego. Igualmente se describe el diseño de los menús de navegación del juego, y como están diseñados. El quinto capítulo se dedica a describir las herramientas empleadas en el proyecto, y su funcionalidad en este. Así como detallar el proceso de desarrollo.

Continuando con el sexto capítulo, se aclara la implementación técnica llevada a cabo en el videojuego, detallando los diferentes scripts y sistemas que llevan a cabo la lógica del videojuego. De igual manera, se razonan las decisiones tomadas a la hora de desarrollar el proyecto, y la experiencia de juego que ofrece el videojuego. Acabando con optimizaciones en Unity, [4] que se han realizado para mejorar el rendimiento. Durante el capítulo séptimo realizamos pruebas para verificar el rendimiento de la solución obtenida, y explicamos los diferentes valores obtenidos.

Por último, contamos con el octavo capítulo que concluye el TFG, resumiendo los hallazgos y resultados obtenidos. Al final del documento se pueden encontrar el anexo 'Códigos del Videojuego' donde se muestran información adicional, en fragmentos del

código fuente del proyecto, como en A.4. Finalmente, acabamos con el noveno capítulo donde se cubren las mejoras pendientes del desarrollo, así como los caminos a evitar en el desarrollo, junto a adiciones en las mecánicas y sistemas del videojuego.

1.4 Vídeo demostración

A continuación se muestra el enlace a un vídeo de demostración, donde se puede observar el resultado final del videojuego en tiempo de ejecución. En este se puede ver toda la interfaz gráfica asi como los diferentes enemigos y el jugador disparando contra estos.

Figura 1.1: Videojuego Surge Survival (haz clic en la imagen para ver el video)

Tras la introducción al trabajo realizado, se pasará a describir el estado del arte, así como la crítica de los videojuegos.

CAPÍTULO 2
Estado del arte

Un videojuego es un software donde uno o más jugadores interactúan a través de un dispositivo controlador, visualizado en una «plataforma» electrónica como una computadora, una máquina arcade, una consola, un dispositivo portátil o un teléfono móvil. Los jugadores mediante el controlador, manejan a un personaje y sus acciones, las cuáles varían según el videojuego, convirtiéndolo en un medio interactivo.

Los videojuegos tienen su origen en los años 70, y han ido evolucionado significativamente, tanto en complejidad como en accesibilidad, así como la cantidad de recursos que necesitan. Los primeros videojuegos, como 'Pong' y 'Space Invaders', eran sencillos y se jugaban en máquinas arcade. Gracias al avance de las tecnologías, se han desarrollado consolas y computadoras personales capaces de ejecutar videojuegos más complejos, y conseguir gráficos más avanzados.

En los últimos años, los videojuegos han tenido un crecimiento muy grande, y un gran impacto en el sector del entretenimiento a nivel internacional. Actualmente la industria del videojuego genera más dinero que la del cine y música juntos. En 2023 generó 180 mil millones de dolares estadounidenses [5]. En gran parte este crecimiento se debe a pandemia del COVID-19, donde varios servicios de entretenimiento tuvieron su auge, junto a los dispositivos móviles, que poco a poco convirtiéndose en la principal generador de ingresos de la industria. Los videojuegos no solo tienen un impacto económico significativo, sino que también influyen culturalmente, ofreciendo nuevas formas de narración y experiencias interactivas.

El desarrollo de videojuegos ha avanzado gracias a diversas tecnologías y motores de juego, los cuáles agilizan el proceso de desarrollo, como 'Unity', 'Unreal Engine' y 'Godot Engine'. Entre ellos Unity es uno de los motores gratis más populares actualmente, utilizado por desarrolladores de todo el mundo, debido a su flexibilidad y capacidad para crear juegos tanto en 2D como en 3D.

Una de las tecnologías más recientes y avanzadas introducidas por Unity es el Entity Component System (ECS) [6]. ECS es un framework de trabajo orientado a datos, que permite a los desarrolladores tener un mayor control y un alto grado de determinismo en sus juegos.

El estado actual de la industria de los videojuegos es una mezcla de tradición e innovación. Los avances tecnológicos, como motores de juego y Unity DOTS están empujando los límites de lo que es posible en el desarrollo de videojuegos, permitiendo a los desarrolladores crear experiencias más complejas y ricas. Este proyecto se enfoca en explorar estas tecnologías emergentes, comparando su rendimiento y efectividad con las metodologías tradicionales, para asi contribuir al conocimiento y práctica en el campo del desarrollo de videojuegos.

2.1 Propuesta de Trabajo

El presente trabajo se centra en el desarrollo de un videojuego de tipo "shoot 'em up" utilizando Unity, con un enfoque en la implementación del nuevo framework 'Data Oriented Technology Stack' (DOTS). Este proyecto llena un espacio significativo en el campo del desarrollo de videojuegos y el rendimiento de estos videojuegos.

2.2 Revisión de videojuegos shoot 'em up existentes

En el género Matamarcianos (shoot 'em up), ha sido uno de mis favoritos desde que tengo memoria. En este el jugador controla un personaje u objeto solitario, que dispara contra hordas de enemigos que van apareciendo en pantalla, con el objetivo de completar un nivel, o sobrevivir a los enemigos.

En la actualidad, los juegos de este género han evolucionado considerablemente, aprovechando las mejoras tecnológicas. Desde clásicos como 'Space Invaders' y 'Spacewar!' que sentaron las bases del género, hasta títulos modernos como 'Vampire Survivors', 'Enter the Gungeon' y 'Cuphead' que han revisionado el género de una manera creativa y diferente.

2.2.1. Referentes

A continuación voy enumerar los videojuegos que se tuvieron en cuenta como referentes para la creación del videojuego, junto a lo razón por las cuáles me inspiraron.

- **Space Invaders** aunque no fuese el primer shoot 'em up, fue el que más popularizó el género en 1979. Desarrollado por Tomohiro Nishikado. Introdujo mecánicas que se marcarían un estándar para el futuro género, como la idea de enfrentarse a oleadas de enemigos y progresión de la dificultad. Este juego tuvo un importante impacto en la cultura y industria del videojuego de la época.

Figura 2.1: Videojuego Space Invaders

- **Vampire Survivors** es el videojuego más inspiro con la idea del videojuego. Desarrollado por Poncle. En Vampire survivors el jugador debe sobrevivir durante 30 minutos a oleadas de enemigos más complicadas sin cesar. Para ayudar al jugador este va adquiriendo gemas de experiencia, con las cuáles sube de nivel volviéndose más poderoso. La jugabilidad del juego reside en el movimiento del jugador mediante el dispositivo controlador, y seleccionar opciones en los menús, esto es debido a que los proyectiles del jugador hacia los enemigos, son automáticos. Debido a la naturaleza de Vampire Survivors, surgió la idea de realizar un videojuego donde aparezcan oleadas de enemigos, y asi aprovechar el framework de Unity DOTS. En la Figura 2.2 se puede observar el videojuego.

Figura 2.2: Videojuego Vampire Survivors

- **Enter the Gungeon** es un videojuego que pertenece a los géneros matamarcianos, 'bullet hell' y 'roguelike'. Desarrollado por Dodge Roll. En este el jugador debe ir avanzando por mazmorras generadas aleatoriamente en diferentes niveles de pisos, y en cada sala derrotar enemigos, en salas especiales puede obtener equipo nuevo para su arsenal (armas, munición, objetos pasivos). Así hasta entrar en la sala principal del piso el jefe, donde se accede al siguiente nivel. Los controles del jugador le permiten, disparar, una voltereta para esquivar balas y saltar obstáculos, usar el activo y balas de fogueó, y botones del menú. Enter the gungeon sirvió de inspiración para el sistema de disparos, y el aspecto de los enemigos. En la Figura 2.3 se puede observar mejor las características de este.

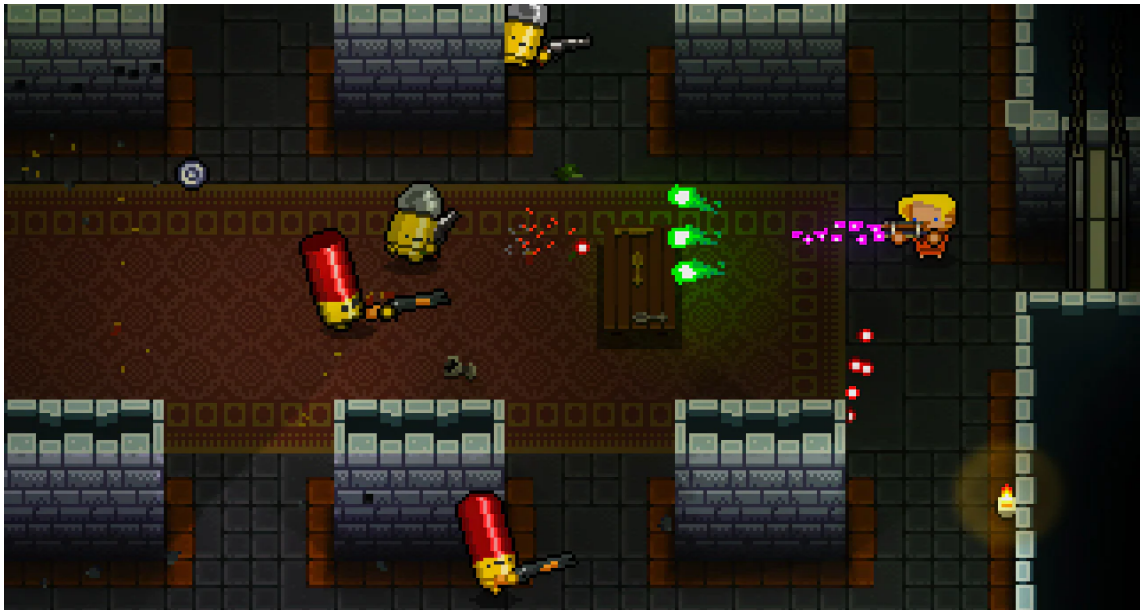

Figura 2.3: Videojuego Enter the Gungeon

2.2.2. Conclusión sobre el estado del género shoot 'em up

Para el contexto de este trabajo, considero que el género shoot 'em up es ideal para explorar y demostrar la utilidad de usar el framework ECS (Entity Component System) de Unity. Ya que la naturaleza del género, con sus oleadas de enemigos y múltiples disparos en pantalla, cumple con el objetivo de Unity DOTS, para aprovechar su rendimiento.

Este trabajo explora las capacidades de Unity DOTS en el desarrollo de un videojuego 'shoot 'em up', un género que se beneficiaría enormemente de la gestión eficiente de múltiples entidades y operaciones en tiempo real.

2.3 Contribución a desarrolladores

Al documentar el proceso de desarrollo, desde la idea, el código, pruebas y resultados, este trabajo sirve como documentación para otros desarrolladores interesados en Unity y DOTS. Las cuáles pueden ser de gran utilidad para la comunidad de desarrolladores de videojuegos, facilitando la comprensión y utilización de esta nueva tecnología. De igual modo, el código fuente del proyecto se pueden encontrar en GitHub fomentando su utilización y uso.

2.4 Tecnologías similares a Unity DOTS

Unity DOTS (Data-Oriented Technology Stack) es un conjunto de tecnologías orientadas a datos que incluye a ECS, el Job System y el Burst Compiler. Estas tecnologías trabajan en conjunto para maximizar el rendimiento del hardware moderno.

- **ECS (Entity Component System):** Permite una gestión más eficiente de los datos y las entidades del juego. Separando la lógica de la presentación y optimizando el uso de las memorias.

- **Job System:** Facilita la creación de tareas paralelas, aprovechando mejor los procesadores multinúcleo para realizar múltiples operaciones simultáneamente, asi como repartir el trabajo en diferentes hilos de ejecución.
- **Burst Compiler:** Compila el código C Sharp a un nivel de máquina altamente optimizado, mejorando significativamente la velocidad en tiempo de ejecución.

En mi experiencia, Unity DOTS me resulta más atractivo, al estar familiarizado con el entorno de desarrollo de Unity.

Por otro lado en Unreal Engine 4 y 5 se están incorporando nuevas tecnologías, que pueden revolucionar la creación de videojuegos con sus altas prestaciones.

- **'Mass Entity framework'**: Un sistema basado en la arquitectura de ECS mediante entidades, en proceso de desarrollo. Que busca ofrecer una gestión de entidades similar a Unity DOTS. Este sistema se enfoca en simular de forma masiva agentes y entidades, así como el uso de múltiples hilos de ejecución para realizar trabajos en paralelo.
- **'Chaos Physics and Destruction System'**: El sistema de física y destrucción en Unreal Engine. Que permite simular físicas complejas de manera eficiente. Pudiendo gestionar una gran cantidad de objetos y colisiones, simulando grandes cantidades de físicas.

Además, NVIDIA integrando inteligencia artificial para mejorar el rendimiento gráfico y la calidad visual en los videojuegos. Lo más destacable es:

- **DLSS (Deep Learning Super Sampling)**: Que emplea redes neuronales, para escalar imágenes de menor resolución a una resolución más alta en tiempo real, mejorando asi la calidad visual y aumentando los FPS.
- **'RTX Ray Tracing'**: Implementa trazado de rayos en tiempo real, proporcionando reflejos, refracciones e iluminación global precisos y realistas.

Actualmente hay varias tecnologías que se centran en mejorar el rendimiento entre el software y el hardware, las cuáles permiten la creación de videojuegos más complejos y llamativos.

Tras esto, concluye el apartado de estado y crítica del arte, para dar paso a la descripción detallada sobre Unity DOTS.

CAPÍTULO 3
Unity DOTS

3.1 Descripción de Unity DOTS

Primero de todo, se va describir como funciona Unity Engine, para comprender mejor su funcionamiento, para entender las mejoras y diferencias que aporta DOTS.

3.1.1. Unity Engine

Unity es un motor donde el 'scripting' (código) deriva de Mono (Monobehaviour), la implementación de código abierto de '.NET Framework'. Aunque los programadores utilizan 'UnityScript' (Un lenguaje personalizado basado en la sintaxis ECMAScript), que esta inspirado en C Sharp o Boo.

En Unity, se emplea el 'parenting' (Padre-Hijo), para organizar y manejar los GameObjects dentro de una escena. Esta estructura permite agrupar objetos de manera lógica y eficiente, facilitando su manipulación conjunta.

Antes de profundizar en Unity Engine, y sus características, es importante entender el concepto de 'GameObject' y 'Prefab' en Unity.

GameObjects

Un GameObject es un bloque de construcción fundamental en Unity. Representa los componentes y datos de un objeto en una escena que definen su comportamiento y apariencia. Los GameObjects, pueden ser desde un personaje, una luz, un terreno, un punto vacío, un cubo, etc. Esto otorga una gran flexibilidad al desarrollador.

Algunos GameObjects funcionan como contenedores, para componentes, los cuáles definen sus propiedades. Dentro de la variedad de componentes, los más comunes son:

- **Transform**: Gestiona la posición, rotación y escala del GameObject, es un mundo 2D o 3D, en este caso en 3D, por lo que empleara las coordenadas (x, y, z).
- **Renderer**: Permite visualizar la malla del objeto 3D en la pantalla.
- **Collider**: Define el área donde el objeto interactúa con otros 'colliders', y las físicas.
- **Scripts**: Permiten añadir lógica, mediante código a los GameObjects, los scripts son un elemento muy importante, para añadir funcionalidades nuevas a los GameObjects.

Prefabs

Los Prefabs en Unity, son plantillas reutilizables de GameObjects, donde se pueden crear múltiples instancias del mismo objeto en el videojuego. Esto es muy práctico para elementos que se repiten a lo largo del videojuego, como enemigos, elementos del entorno, objetos, etc.

Estos Prefabs se pueden instanciar por código mediante los scripts, permitiendo añadir a una escena en ejecución objetos complejos, sin mucha dificultad.

Cuando un Prefab recibe un cambio, este se refleja en los objetos dentro de una escena, es decir se actualizan todas sus instancias, facilitando la gestión y organización de los objetos.

Parenting en Unity

Tras comprender el concepto de GameObject y Prefab a continuación, se explica cómo funcionan estas jerarquías y el concepto de parenting en Unity:

En la ventana de jerarquía en Unity, se muestras todos los GameObjects presentes dentro de una escena, dentro de esta se pueden organizar todos sus elementos. Se pueden tanto añadir como eliminar elementos, y mover elementos. Cada elemento tiene un GameObject padre, y el padre de todos es la propia Escena.

Dentro de la ventana de jerarquía en Unity, se muestran todos los GameObjects presentes en una escena. En esta, se pueden ver y organizar modelos, cámaras, prefabs, y otros elementos. Cada vez que añades o eliminas un GameObject en la vista de escena, estos cambios se reflejan automáticamente en la ventana de jerarquía.

Además, la ventana de jerarquía puede contener múltiples escenas, cada una con sus propios GameObjects. Esto permite gestionar proyectos más complejos que requieren múltiples escenarios o niveles.

- Parent-Child Relationships (Relaciones Padre-Hijo):
  - **GameObject Padre**: Es el objeto de nivel superior en una jerarquía. Sirve para contener otros GameObjects denominados hijos.
  - **GameObject Hijo**: Es un objeto que hereda propiedades y transformaciones del Padre. Esto implica que si el padre se mueve, rota o escala (su componente Transform) todos sus hijos también lo hacen en consecuencia. Pero el Hijo no hereda scripts, u otras componentes que tenga el Padre, además es capaz de tener más hijos, volviendo asi al Hijo en padre de otros GameObjects.

Estas relaciones-padre hijo se pueden crear desde el menú, o arrastrando y moviendo GameObjects a la ventana de jerarquía. En la Figura 3.1, se puede observar la ventana de jerarquía, la escena y en inspector con informacion del GameObject seleccionado.

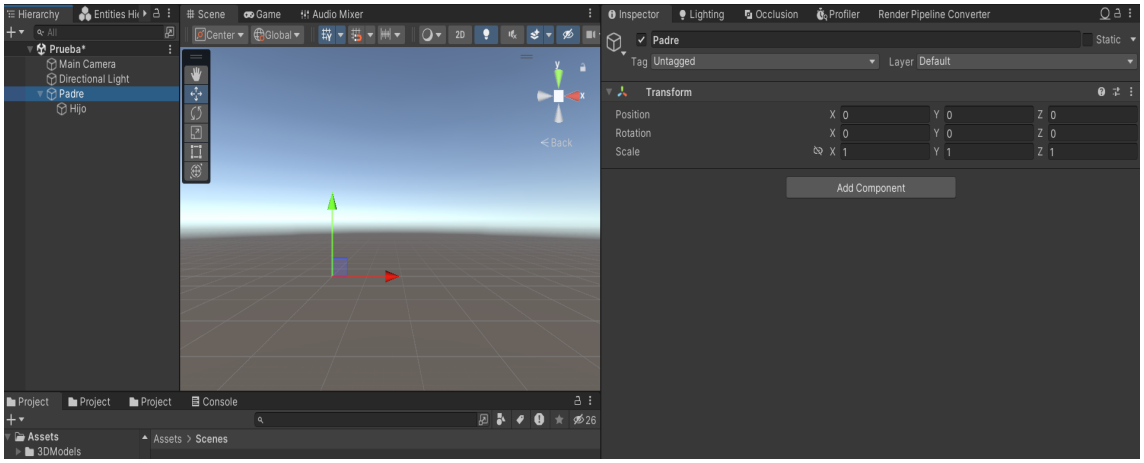

Figura 3.1: Ventana de Jerarquía, Escena, e Inspector

Pasando a las características de Unity DOTS, estas se explicarán a continuación.

3.1.2. Unity DOTS

A diferencia del enfoque de Unity tradicional, Unity DOTS (Data-Oriented Technology Stack) emplea un conjunto de tecnologías avanzadas que incluyen a ECS (Entity Component System), Job System y el Burst Compiler. En oposición a Unity, para usar DOTS hay que crear una sub-escena dentro de la escena principal, que contendrá los sistemas de DOTS.

A continuación se explicarán más las tecnologías que aporta Unity DOTS.

3.1.3. Entity Component System (ECS)

ECS permite estructurar el código de manera que los datos y la lógica se ejecuten de forma separada. Esto es posible mediante la creación de Entidades, las cuáles son GameObjects, que añaden un identificador único (32-bits 'integers'), y datos que se asocian a las Entidades. Esto permite compactar y alinear los datos en bloques contiguos de memoria, lo cual hace que las escrituras y lecturas, sean muchos más eficientes que la lectura de memoria que emplean las 'RAM' tradicionales.

Para lograr un alto rendimiento y eficacia, se debe estructurar la lógica del juego mediante sistemas que operen con estas Entidades y datos. De esta manera se podrá dividir en tareas los trabajos, que se podrán ejecutar de forma paralela en múltiples núcleos de CPU. Asimismo, ECS ofrece determinismo con los datos, para que las ejecuciones sean reproducibles y predecibles.

Para visualizar la jerarquía de Entidades en Unity, una vez creada la 'sub escena', en la ventana de jerarquía de entidades se pueden estructurar y acceder a sus datos y componentes. En la Figura 3.2 se puede observar como esta seleccionada la Entidad 'PlayerEntity', y a la derecha de la imagen se aprecian sus componentes.

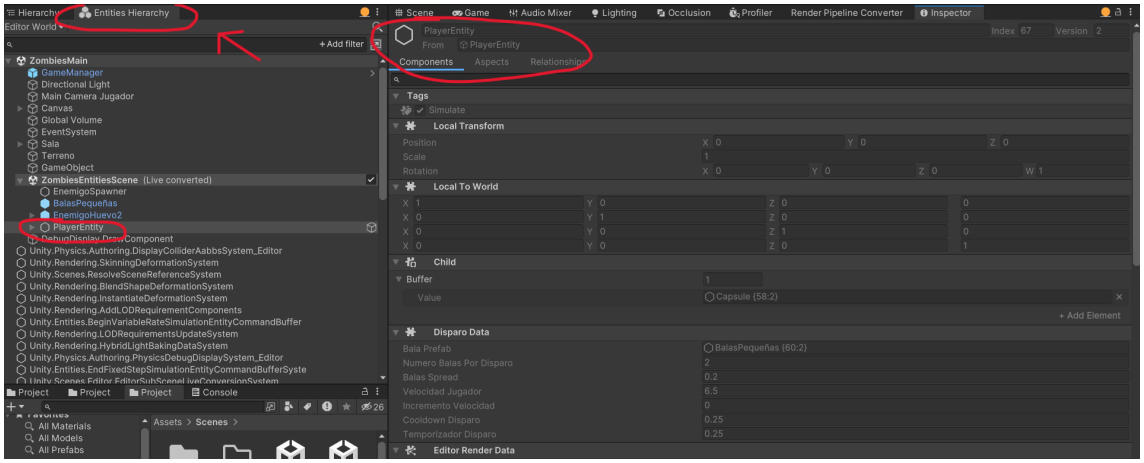

Figura 3.2: Jerarquía de Entidades en Unity

Además, dentro de Unity, hay 3 modos de visualizar los datos de un Entidad.

Modos de Inspector

1. **Authoring mode**: Contiene los datos de la Entidad, cuando no esta en ejecución, desde este modo podemos asignar componentes a la Entidad, como scripts que deriven de MonoBehaivour y asignar valores a sus variables.

2. **Runtime mode**: Muestra los datos de las componentes de la Entidad, cuándo se están ejecutando, y podemos observar como varían en el modo Juego.

3. **Mixed mode**: Una mezcla entre Authoring y Runtime, donde podemos observar los datos tanto en ejecución, como en no ejecución.

En la Figura 3.3 se puede observar una imagen del botón que permite variar de modo.

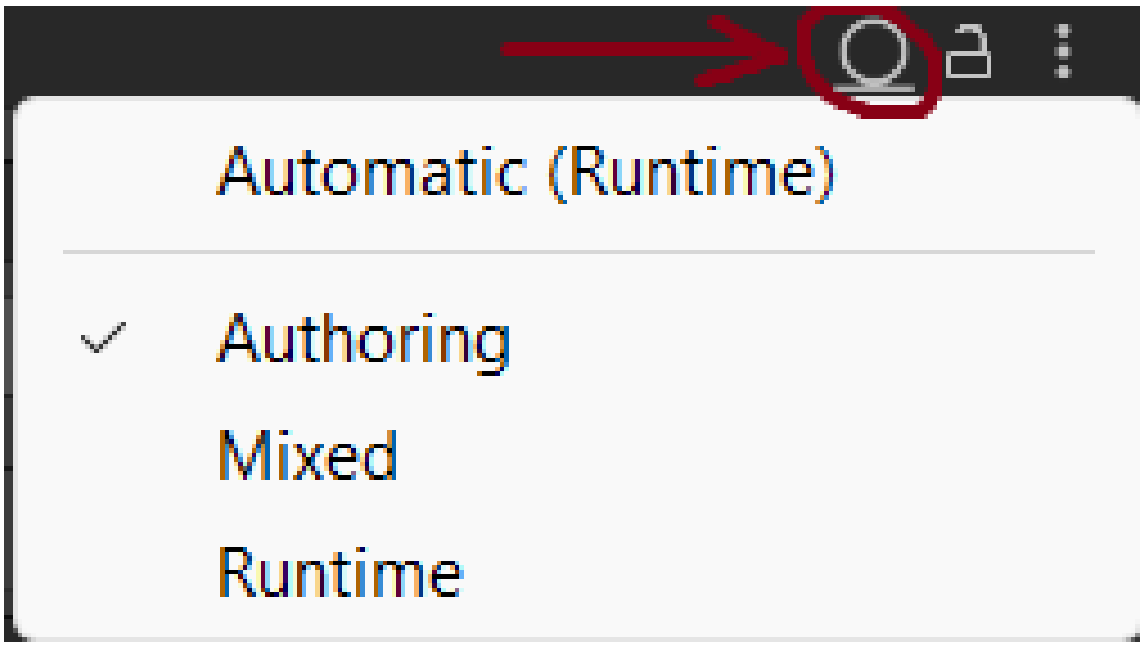

Figura 3.3: Modo de Inspector

3.1.4. Job System

Para poder escribir código que se ejecute paralelamente se emplea el Job System, el cual permite utilizar varios núcleos de CPU para realizar operaciones paralelas. Maximizando así las capacidades del hardware en sistemas multinúcleo actuales.

Para la utilización del Job System, este nos aporta interfaces de las cuáles nuestras clases pueden heredar:

- **IJobEntit**: Para crear trabajos que se ejecuten de forma asíncrona.
- **ParallelFor Jobs**: Permite dividir tareas en múltiples hilos que se ejecutan en paralelo.
- **IJobParallelFor**: Permite crear 'IJobs' de forma paralela para dividir la carga del trabajo, en partes más pequeñas.
- **Native Containers**: Contenedores de datos nativos como 'NativeArray', 'NativeList' y 'NativeQueue', que son seguros para realizar operaciones concurrentes.

3.1.5. Burst Compiler

El Burst Compiler es una herramienta que convierte el código de alto nivel escrito en C Sharp que emplea Unity, a un lenguaje máquina muy optimizado. Este compilador se encarga de integrar las Entidades y datos de ECS con los trabajos paralelos, mejorando así las prestaciones del juego.

De igual modo utiliza técnicas de vectorización para procesar datos simultáneamente. Mejorando así el tiempo de respuesta del juego y reduciendo los tiempos de procesamiento de la CPU.

Esta técnica empleada es 'Just-In-Time' (JIT) vs 'Ahead-Of-Time' (AOT) Compilation, que significa que Burst compilará el código en el momento que se tenga que usar, esto por defecto se realiza de forma asíncrona, por lo que el código se ejecutara en el JIT de Mono, hasta que la compilación de Burst acabe. Una vez completada pasará a gestionarlo Burst.

Sin embargo, una vez construida la build del proyecto como un 'Standalone Player' (Archivo ejecutable del videojuego), se compilará de forma AOT, de forma que estará optimizado para la versión final del juego.

También resulta interesante comentar el funcionamiento de la memoria aliasing del compilador Burst.

Sin memoria Aliasing

La memoria aliasing es un concepto que puede llevar a optimizaciones significativas para un compilador, que es consciente de cómo se utiliza la información en el código.

Si los dos arrays, Vector1 y Vector2 no se superponen ligeramente (sus ubicaciones de memoria no se solapan), el compilador, si es consciente de que no hay aliasing, optimizará el bucle escalar mediante la vectorización.

Esto significa que reescribirá el bucle para procesar elementos en pequeños lotes (por ejemplo, 5x5 elementos). Estos vectores se pueden ver en la Figura 3.4, donde crea 2 bloques, el 1º con los valores (a, b, c) y el 2º con (d, e).

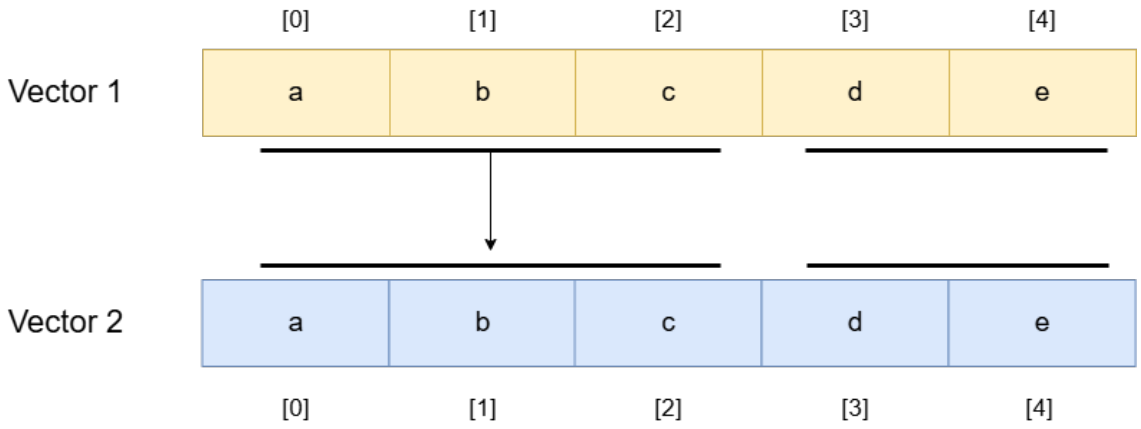

Figura 3.4: No Memoria Alisasing

Con Memoria Aliasing

Si por alguna razón (que no es fácil de introducir directamente con el JobSystem), el array Vector2 se superpone con el array Vector1 en un elemento (por ejemplo, Vector2[0] apunta realmente a Vector[1]), implica que hay aliasing de memoria, obteniendo así un resultado diferente al ejecutar un trabajo (suponiendo que el autovectorizador no esté activado), en este caso, el código generado sería incorrecto y podría provocar errores graves si no son identificados por el compilador.

Esto se observa mejor en la Figura 3.5

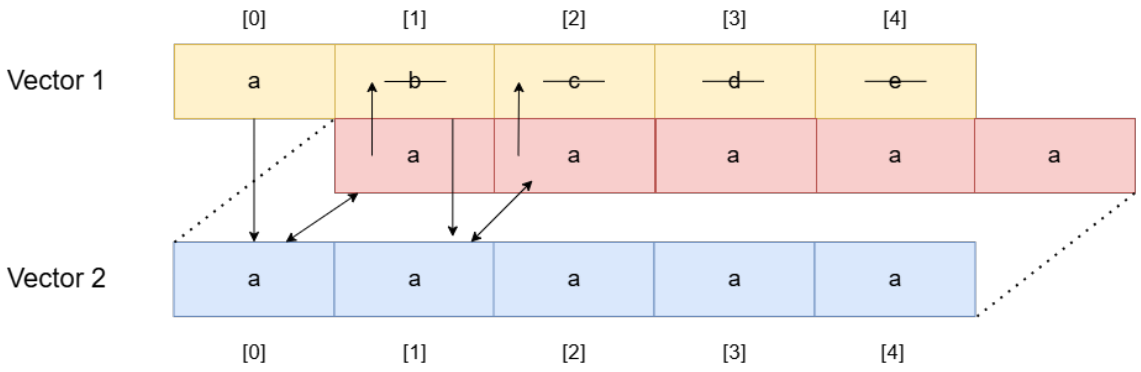

Figura 3.5: Memoria Alisasing

Finalmente, para utilizar el Burst Compiler en nuestro código, es necesario usar la directiva [BurstCompile] que indica al compilador que queremos emplear el optimizador de BurstCompiler en ese bloque de código. De igual manera, hay que emplear tipos de datos que el BurstCompiler pueda aprovechar, como 'NativeArray', 'float3', 'uint', entre otros.

Por último, es fundamental emplear clases como el 'EntityCommandBuffer' y el 'EntityManager' para enlazar los 3 sistemas (ECS, Job System y Burst Compiler) para que funcionen al unísono, logrando así aprovechar su gran ventaja.

Entity Command Buffer

El Entity Command Buffer (ECB) en Unity, es una herramienta que permite programar cambios estructurales en entidades de forma segura y eficiente. Un ECB mantiene una cola comandos seguros para hilos de ejecución, de esta forma se pueden añadir comandos desde diferentes hilos y ejecutarlos paralelamente tras el hilo principal.

Entity Manager

El encargado de controlar la creación y destrucción de entidades, así como su modificación y consultas de datos, es el Entity Manager. Este proporciona herramientas para interactuar con las entidades del juego de manera eficiente y estructurada.

Para servirse del Entity Manager se puede lograr mediante dos métodos:

- **World**: Sirviendo del mundo de entidades donde existe.
- **Referencia**: Pasando como referencia el 'state' (estado) de 'SystemState' para obtener el Entity Manager. En el anexo A.6 se puede encontrar el código del sistema 'Enemigo System' donde se realiza esta acción, y un fragmento de 'PlayerInterfaz.cs' donde se obtiene del propio mundo de entidades.

A continuación se hablará sobre la solución que se propone en el proyecto.

3.2 Solución propuesta

La solución propuesta en este trabajo, propone ampliar la práctica y desarrollo de videojuegos. Para ello la propuesta se centra en evaluar y comparación el rendimiento de un videojuego realizado Unity DOTS frente a los métodos tradicionales.

A diferencia de trabajos anteriores, este se centra en la teoría y practica desarrollando un videojuego con DOTS. Puesto que gracias a esta tecnología se obtendrá un rendimiento mayor en términos de FPS(Frames Per Seconds) y uso de memoria. Así como demostrar como emplear estas herramientas para futuros desarrollos.

3.3 Plan de Trabajo

3.3.1. Diagrama de Gantt

El Diagrama de Gantt es una herramienta muy útil para gestionar y planificar el proyecto. Debido a que este aporta una visión sobre las diversas fases del proyecto, y ayuda a establecer una idea general del proceso a seguir.

- **Planificación**: Abarca actividades como la búsqueda de información, el estudio de esta, instalación del entorno y herramientas para llevar a cabo el proyecto, y la propia creación del diagrama Gantt. Esta fase ocupa un sexto del cronograma.
- **Arte**: En esta fase que comienza tras acabar la planificación, corresponde con la fase donde se buscarán recursos, como efectos de sonidos, objetos 3D, imágenes. Así como modelar objetos 3D, y crear imágenes para usar dentro del videojuego.

- **Desarrollo**: El desarrollo se puede paralelizar con el arte, aunque algunas tareas si dependen de elementos del arte, como tener un modelo de un enemigo, para poder visualizar su movimiento. Asimismo, esta fase incluye todas las tareas de desarrollo y ejecución del proyecto, como el movimiento del jugador, el sistema de oleadas de enemigos, e integraciones del arte previamente creado o importado al proyecto.
- **Pruebas**: Contiene las pruebas realizadas, una vez terminado el desarrollo e integrado el arte, estas pruebas recaudan información sobre el rendimiento del videojuego, en memoria, CPU, FPS. Del mismo modo también sirvieron para ajustar la dificultad del videojuego (vida enemigo, daño jugador, velocidad, cantidad enemigos).Y mejorar aspectos del juego, como la posición de la cámara, y diversos retoques para mejorar la experiencia de juego.

Como se puede observar en la Figura 3.6 se muestra visualmente el diagrama Gantt descrito anteriormente. Donde el proceso de desarrollo empieza en Mayo y concluye en Junio. Varias de las tareas observadas en el diagrama, se pueden paralelizar como obtener efectos de sonido y música. Así como las pruebas de jugabilidad, que han sido testeadas a los largo del desarrollo, así se comprobaba que el juego tenia el funcionamiento esperado.

Este diagrama de Gantt fue diseñado mediante la herramienta web, 'Monday.com', que permite realizar espacios de trabajo donde organizar el proyecto. Y posteriormente convertir las tablas creadas en el espacio de trabajo en el diagrama de Gantt final.

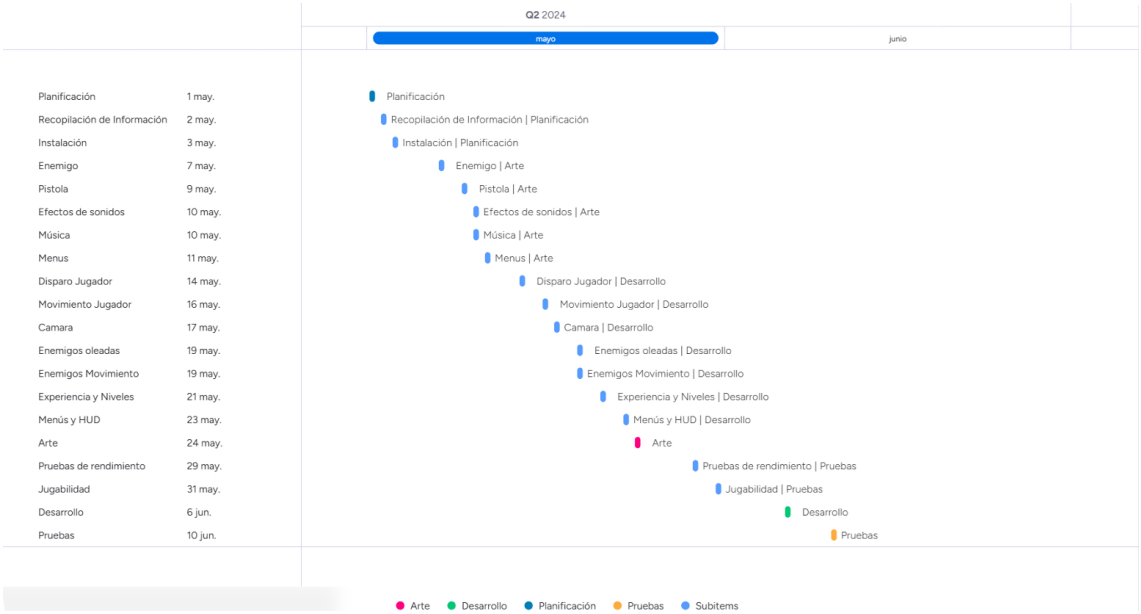

Figura 3.6: Diagrama de Gantt del proyecto

Despues de entender mejor Unity DOTS, y el estado del arte. Se detallará en el siguiente cpítulo el diseño y arquitectura que emplea el videojuego desarrollado.

CAPÍTULO 4
Diseño del videojuego

4.1 Arquitectura del Sistema

En este capítulo se comentara la arquitectura de los sistemas, además de los diferentes elementos del videojuego.

4.1.1. Descripción de la Arquitectura

La arquitectura del videojuego, se ha diseñado siguiendo el paradigma orientado a datos que utiliza Unity DOTS (Data-Oriented Technology Stack). Esta arquitectura permite una separación clara entre los datos y la lógica del juego, mejorando la eficiencia y el rendimiento.

Dentro de los sistemas, se encuentran varios subsistemas que trabajan para proporcionar una experiencia de juego eficiente y rápida. A continuación, se detallan subsistemas de la solución propuesta.

- **Entidades (Entities)**: Representan todos los objetos del juego, como el jugador, los enemigos, los proyectiles, etc. Cada entidad tiene componentes diferentes, como scripts (Archivo C Sharp), como su identificador de entidad, su posición (x, y, z) trasladada al mundo físico del videojuego.
- **Componentes**: Cada componente es un contenedor de datos asociado a una entidad. Por ejemplo, DisparoData, PlayerDañoData, EnemigosPropiedades. Los componentes no contienen lógica, solo datos, que se usaran luego con su respectivo archivo para convertirlos en authoring data (datos de autoría), que permiten asignar valores a estos datos en tiempo de ejecución.

  Este proceso se conoce como Baking, donde se transforma un GameObject en authoring data. Un ejemplo de esta conversión se puede observar en el apéndice A.4. De esta manera se pueden crear convertir GameObjects (Prefabs) a Entidades en tiempo de ejecución.
- **Sistemas**: Los sistemas contienen la lógica del juego y operan sobre las entidades y sus componentes. Algunos ejemplos son DisparoYMovimientoSystem, BalasYNiveles, ColisionesEnemigoPlayerSystem. Cada sistema, ejecuta su lógica en todas las entidades que tengan los componentes necesarios añadidos mediante un script mono. De esta manera podemos identificar a una entidad mediante sus datos y componentes. Con esto podemos buscar a un enemigo en un sistema y poder modificar sus valores. Un enemigo se identifica por el componente añadido de 'EnemigosPropiedades' en el propio sistema.

4.2 Bucle del Juego

El bucle del juego, o 'Game loop' [2], es la estructura central que define el flujo del videojuego y las interacciones del jugador con el entorno y los enemigos. En este videojuego, el jugador se enfrenta a oleadas sin fin de enemigos con el objetivo de sobrevivir el mayor tiempo posible, y obtener una puntuación alta al derrotar enemigos. Para ayudar al jugador a sobrevivir, este irá adquiriendo experiencia para subir de nivel y mejorar sus habilidades a medida que derrota enemigos.

Descripción del Game-loop

- Al comienzo de cada oleada, se generan enemigos en puntos de 'spawn' alrededor del jugador. Estos puntos se distribuyen a lo largo de un radio alrededor del jugador, la cantidad de enemigos y la dificultad aumentan con cada oleada. Convirtiendo el juego en un constante desafió.
- El jugador debe derrotar a los enemigos que aparecen en oleadas. Utilizando su arma, para disparar a los enemigos, mientras el jugador debe sobrevivir de los enemigos evitando ser derrotado, moviéndose por el mapa.
- Cuando se derrota a un enemigo, el jugador obtiene puntos de experiencia. Si el jugador acumula suficientes puntos de experiencia, sube de nivel. Cada vez que el jugador sube de nivel, se le presenta un menú de mejoras al jugador. El menú dispone de tres opciones aleatorias. Estas opciones se generan aleatoriamente de una lista de mejoras, siendo estas mejoras del personaje, como su vida, daño, velocidad.
- Cada oleada contiene más enemigos que derrotar que la anterior, hasta un máximo de 150 enemigos simultáneos. Asimismo, cada vez que el jugador sube de nivel, se incrementan las características de los enemigos aumentando el desafío.
- El objetivo del jugador es sobrevivir el mayor tiempo derrotando enemigos a la vez. El juego termina cuando el jugador es derrotado, registrándose su puntuación y progreso.

En la Figura 4.1 se puede observar un diagrama del game-loop del videojuego, que empieza en el inicio

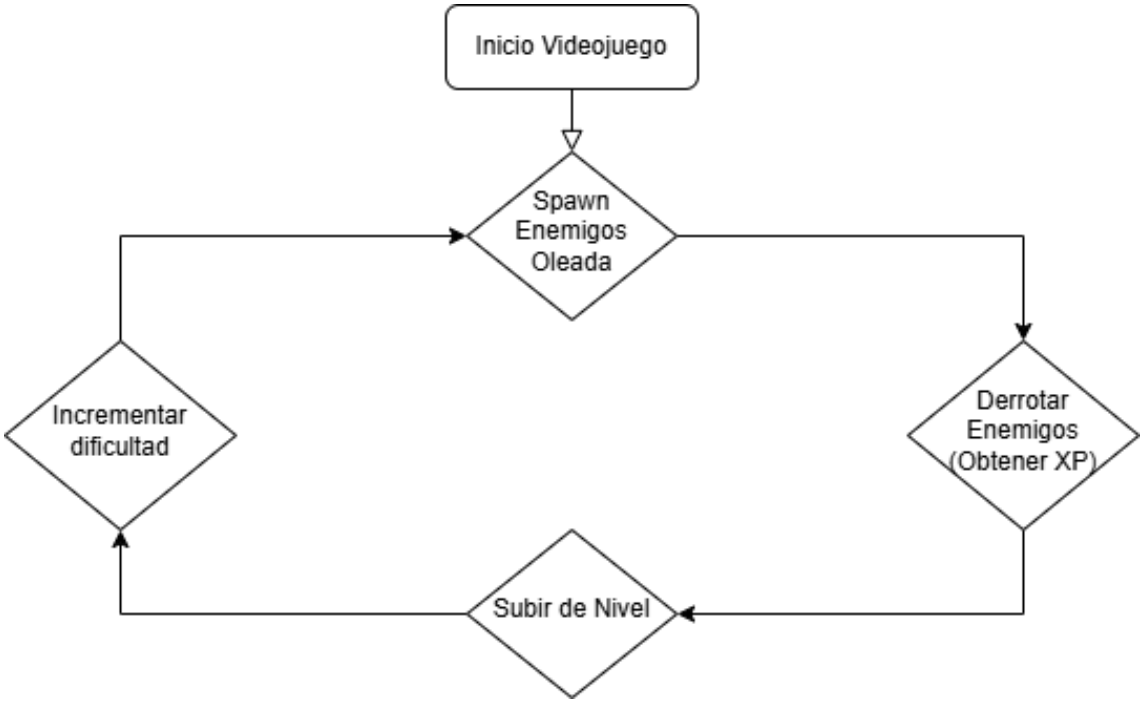

Figura 4.1: Diagrama del Game-loop jugable

4.3 Escenario

El escenario del videojuego proporciona el contexto visual y ambiental en el que se desarrolla la acción. Un entorno bien diseñado no solo mejora la inmersión del jugador, sino que también puede influir en la jugabilidad y las estrategias utilizadas.

Utilizando la herramienta de creación de terrenos de Unity, se diseñó un paisaje con montañas y hierba que rodean el área central de acción. Este terreno brinda una sensación de vastedad y realismo, haciendo que el jugador se sienta inmerso en un mundo amplio y natural.

La herramienta de creación de terrenos permite esculpir el terreno, aplicar texturas y añadir detalles como hierba y piedras, creando un entorno visualmente atractivo y variado.

Ajuste del Skybox y la Niebla

Un 'Skybox' es una técnica empleada en los gráficos de videojuegos, para crear un entorno visualmente atractivos que rodea al jugador, simulando un fondo tridimensional con un horizonte distante. El Skybox consiste en una serie de texturas que se aplican a las caras internas de un cubo, envolviendo la escena del juego, creando el efecto del fondo tridimensional.

Esta técnica permite ahorrar la necesidad de crear y renderizar objetos distantes, los cuáles serían muy costosos computacional-mente. Estos fondos pueden variar entre un cielo, un paisaje montañoso o un espacio interestelar, según la atmósfera deseada en el juego.

Para mejorar la atmósfera y el realismo del escenario, se modificó el Skybox dentro del proyecto, que es la cúpula que representa el cielo y el horizonte. Se eligió un Skybox que simula un cielo oscuro y peligroso, para conseguir una sensación de peligro constante du-

rante el juego, complementando con un terreno montañoso. Se añadió niebla al escenario para simular un mundo real y ocultar el borde del nivel. La niebla crea una sensación de profundidad y distancia, haciendo que el entorno se sienta más vasto y continuo.

En la Figura 4.2 se ilustra el escenario del juego, donde se encuentran el Skybox y el terreno montañoso creado para videojuego.

Figura 4.2: Skybox y terreno del escenario

Detalles Adicionales

Se colocaron elementos decorativos como hierba y piedras alrededor del terreno. Estos detalles no solo embellecen el escenario, sino que también sirven para mejorar el ambiente y la atmósfera general del escenario.

Para mejorar el aspecto gráfico del juego, el nivel de detalle de algunos elementos gráficos, como la hierba se renderizan solo si se encuentran a una distancia especificada. De esta manera asignando una distancia corta, si el jugador se acerca podrá apreciar los detalles y si halla lejos no los observará debido a la distancia entre ellos, asi evitamos renderizar objetos lejanos.

4.4 Menús

En esta sección, se van a describir los diferentes menús del videojuego, los cuáles son fundamentales para la navegación y la interacción del usuario. Cada menú tiene su propio propósito y funcionalidad, facilitando una experiencia de usuario fluida y organizada. Los menús implementados en el videojuego son los siguientes.

Menú Principal

El menú principal es la primera interfaz que ve el jugador al iniciar el juego. Este menú contiene las siguientes opciones:

- **Jugar**: Inicia el juego, llevando al jugador directamente al nivel del juego.
- **Salir**: Cierra la aplicación del juego y vuelve al escritorio.

El diseño del menú principal es simple y directo, asegurando que los jugadores puedan comenzar a jugar rápidamente o salir del juego con facilidad. En la Figura 4.3 se puede ver el menú principal del juego.

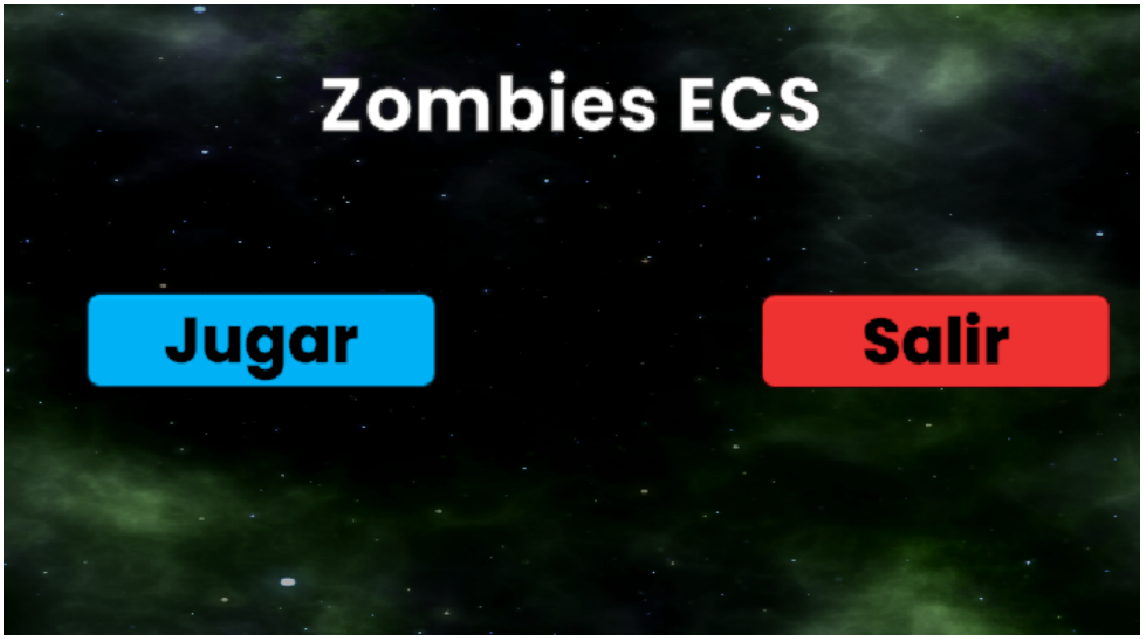

Figura 4.3: Menú principal del videojuego

Menú de Pausa

El menú de pausa se activa cuando el jugador presiona la tecla 'Escape' en tiempo de ejecución. Este menú permite al jugador, pausar el juego y continuar sin tener consecuencias en el juego. Las opciones de este menú son las siguientes:

- **Continuar**: Retoma el juego desde el punto en que se pausó.
- **Volver al Menú Principal**: Lleva al jugador de vuelta al menú principal, abandonando la partida actual.
- Salir: Cierra la aplicación del juego, y lleva al usuario al escritorio.

  En cuanto al diseño del menú de pausa, se empleo el diseño estándar que suelen incorporar este tipo de menús, en la Figura 4.4 se muestra el Menú de pausa.

Figura 4.4: Menú de Pausa activado

Menú subir de Nivel

Este menú aparece cuando el jugador alcanza un nuevo nivel durante la partida. Este nivel permite al jugador escoger entre tres mejoras, así como pausar el juego, para permitir al jugador elegir correctamente. Las opciones de este menú son:

- **Mejora aleatoria 1**: Una mejora seleccionada aleatoriamente.
- **Mejora aleatoria 2**: Otra opción de mejora seleccionada aleatoriamente.
- **Mejora aleatoria 3**: Una tercera opción de mejora seleccionada aleatoriamente.

Un ejemplo de las diferentes opciones aleatorias que pueden generarse y el diseño del propio menú de subir de nivel se muestra en la Figura 4.5, en esta imagen se muestran las mejoras de 'velocidad', 'reducir daño recibido' y 'daño'. Estas mejoras se explicarán más detalladamente en futuros apartados.

Figura 4.5: Menú Subir de Nivel con tres opciones

Menú Game over

Finalmente el menú de muerte (Game Over) aparece cuando al jugador se queda sin puntos de vida. Este menú muestra la puntuación del jugador y ofrece opciones para reiniciar el juego o salir al escritorio, estas opciones son:

- **Reiniciar**: Comienza una nueva sesión de juego desde el inicio.
- **Menú principal**: Vuelve al menú principal del juego, donde el jugador puede decidir salir de la aplicación o volver a jugar.

El diseño del menú de muerte, emplea el uso del color amarillo, para destacar las funciones principales, como ver la puntuación del jugador y el botón de reiniciar, asimismo el fondo tiene un color negro con opacidad, para poder observar el juego, pero no ser el foco principal. En la Figura 4.6 se observa el menú de muerte una vez que el jugador es derrotado.

Figura 4.6: Pantalla de Game over

Tras esto el jugador, puede volver a jugar el nivel. O volver al menú principal, y desde este cerrar la aplicación.

Con esto concluyen los diferentes menús del videojuego. En el próximo capítulo se hablará del proceso de desarrollo del trabajo.

CAPÍTULO 5
Desarrollo

5.1 Entorno de Trabajo

En este proyecto, he utilizado una combinación de herramientas y frameworks para la implementación y el manejo del desarrollo del videojuego. A continuación, se detallan las herramientas utilizadas para llevar a cabo el proyecto. Que se detallan a continuación:

5.1.1. Herramientas y Tecnologías Utilizadas:

1. **Unity 2022.3.3f1 LTS**: Es el motor de desarrollo de videojuegos utilizado para desarrollar el videojuego en 3D. La versión elegida es la LTS (Long-Term Support) ya que proporciona estabilidad y soporte extendido por Unity, además de ser las únicas versiones donde el desarrollo de videojuegos con DOTS es posible. Unity DOTS también requiere usar un proyecto URP (Universal Render Pipeline) que ofrece las posibilidades de cambiar el renderizador pre-construido.

   Dentro de Unity mediante el Unity Package Manager, y la Asset Store de Unity se han instalado paquetes, y recursos adicionales para agilizar el proceso de desarrollo, así como esenciales para llevarlo a cabo. A continuación se muestran los paquetes utilizados.

   - **Unity Entities 1.0.16**: Maneja la creación y gestión de entidades y sus componentes.
   - **Entities Graphics 1.0.16**: Facilita la representación gráfica de entidades dentro de Unity.
   - **Unity Physics 1.0.16**: Proporciona simulaciones físicas para las entidades.
   - **Burst 1.8.8**: Compila el código C Sharp y lo construye a código máquina optimizado, mejorando significativamente el rendimiento.
   - **Collections 2.1.4**: Ofrece estructuras de datos optimizadas para usar con DOTS.
   - **Input System 1.6.3**: Paquete que permite leer la entrada de diferentes tipos de dispositivos controladores.

5.1.2. Assets utilizados

Los assets son recursos creados por otras personas, que abarcan desde, herramientas, modelos 3D, música, imágenes, efectos especiales, etc. En el proyecto se han utilizado diversos assets, a continuación se detallan cuáles se han empleado:

- **Lowpoly Environment - Nature Free**: Creado por Polytope Studio, que proporciona modelos y texturas para poder crear un terreno realista en Unity.
- **Fantasy Skybox FRE**: Creado por Render Knight, el cual aporta varias skybox (fondo del cielo). Con el que se obtiene un cielo ya creado, para dar una mayor atmósfera al videojuego, y hacer al jugador la experiencia mas realista.
- **Sci-Fi Gun Model**: Creado por MASH Virtual. Es un modelo de una pistola, el cual fue usada en el juego como arma principal.
- **Fuente de texto**: Para el texto en los menús se ha empleado la fuente de Poppins, obtenida de Google Fonts.
- **Dragon Castle**: Creada por Makai Symphony, se puede encontrar en Soundcloud, es una canción fue utilizada para la música de fondo dentro del juego, es una pieza de acción épica, la cual acompaña con la jugabilidad y temática del juego.
- **Space Debris**: Creada por y VOiD1 Gaming, en su paquete de música de menús gratis (Free Game Menu Music Pack), la cuál fue empleada para la música del menú principal.

2. **Visual Studio** es utilizado como editor de código por defecto en Unity. Su uso en el proyecto es crear y editar los diferentes archivos C Sharp, que contienen toda la programación del proyecto. En la Figura 5.1 se puede observar la clase 'DisparoMono.cs'

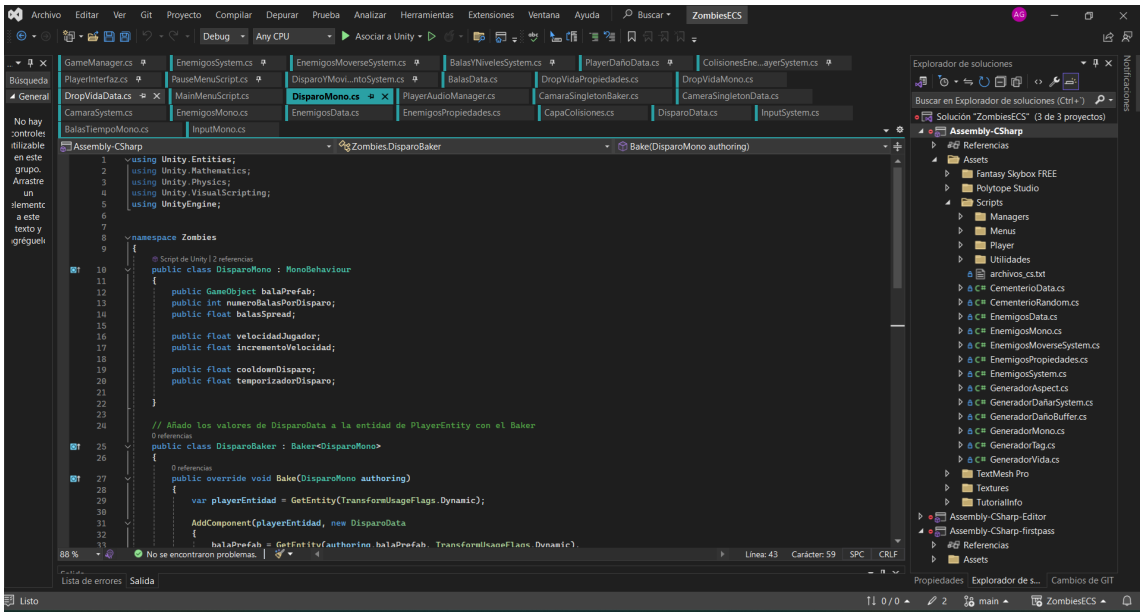

Figura 5.1: DisparoMono.cs en Visual Studio

3. **Blender** es una herramienta de modelado de objetos 3D gratuita y de código abierto, animaciones, efectos visuales y más. En el Proyecto ha sido utilizado para modelar algunos objetos 3D que se integran en el videojuego, y así obtener poder diferenciarse de otros videojuegos.

   En la siguiente Figura 5.2 se puede ver el enemigo principal del videojuego, que fue modelado en Blender.

Figura 5.2: Enemigo principal del videojuego en Blender

4. **Git** es un software de control de versiones, diseñado para la con-fiabilidad y compatibilidad para mantener el código, y los archivos dentro de un proyecto. Así como mantener un historial, permitiendo revertir cambios a versiones anteriores.

   Junto a Git, se empleo GitHub Desktop, una aplicación gráfica de código abierto que permite trabajar con el código que se hostea en GitHub, además de ayudar a realizar comandos típicos de Git de manera sencilla, como push, pull, revert, etc.

   En el proyecto se ha dedicado a realizar un seguimiento de cambios en los archivos, y mantener un historial de cambios, así como revertir cambios, cuando se producían errores fatales en el proyecto. En el enlace siguiente se puede visitar el Repositorio GitHub asociado.

   Repositorio GitHub del proyecto

5. **Trello** es una herramienta de gestión de proyectos basada en tableros, que permite organizar tareas y colaborar de manera efectiva. En el proyecto se ha empleado para dividir tareas y organizar el seguimiento de estas, para comprobar el estado del proyecto. En la Figura 5.3 se puede apreciar el tablero creado en Trello, para ir gestionando las tareas. Estas se agrupan en: no empezado, en proceso, revisar y acabado.

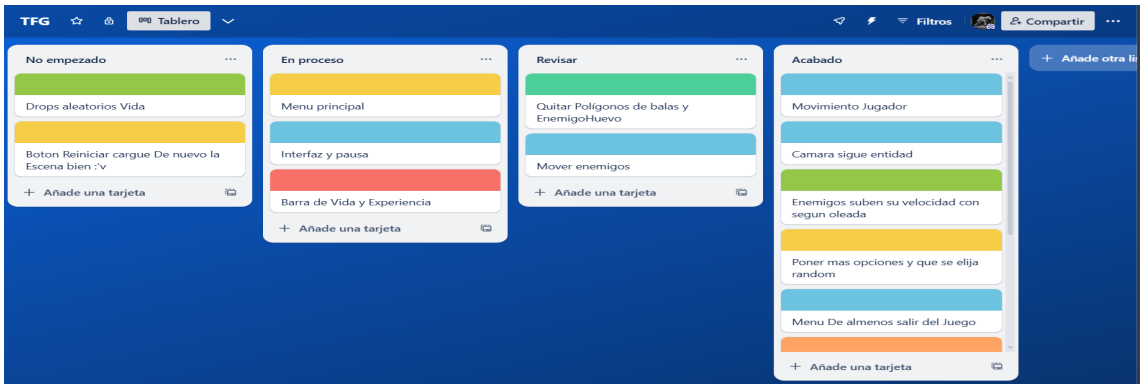

Figura 5.3: Tablero de Trello

5.2 Proceso de Desarrollo

El proceso de desarrollar un videojuego puede resultar ser complejo, para ello generalmente se redacta un GDD (Game Design Document) [1] que es un documento que se va actualizando según se redacta. En este el contenido abarca todos los aspectos de un videojuego, tales como la jugabilidad, mecánicas, historia, personajes y más. Así pues se logra documentar y estructurar el avance del proyecto.

En este proyecto, de manera similar el proceso de desarrollo se ha dividido en varias etapas clave. A continuación, se describen estas etapas:

- **Objetivo**: Crear un videojuego 'shoot 'em up' utilizando Unity DOTS, optimizando el rendimiento a través de ECS, el Job System y el Burst Compiler.
- **Alcance**: Determinar las características principales del juego, como el número de niveles, tipos de enemigos, mecánicas de juego, y mejoras del personaje.
- **Estudio del Estado del Arte**: Investigar otros videojuegos del género 'shoot 'em up' y las tecnologías similares a Unity DOTS para comprender las tendencias actuales y las mejores prácticas actuales.
- **Diagramas y Tableros**: Crear diagramas como el de Gantt y el tablero de Trello, para representar la estructura del videojuego y la interacción entre los distintos componentes, así como el avance y progreso del proyecto durante el desarrollo.
- **Instalación de Herramientas**: Una vez realizada toda la preparación e investigación, se procede a configurar e instalar Unity 2022.3.3f1 LTS, Visual Studio, Git, Blender y otras herramientas necesarias.
- **Integración de Paquetes**: Importar los paquetes de Unity DOTS, como Unity Entities, Entities Graphics, Unity Physics, Burst, Collections, y el Input System, así como paquete obtenidos del Asset Store de Unity, o otros recursos de arte como efectos de sonido y música.
- **Historia y Ambientación**: El juego tendrá una historia casi inexistente ya que no tendrá relevancia durante el juego. Por otra parte la ambientación del nivel, si aportará a la experiencia de juego. El objetivo es crear un ambiente oscuro y aterrador, para dar a entender que el jugador se encuentra en una situación de peligro.
- **Bucle de Juego**: El bucle del juego compone las mecánicas centrales del juego, como el movimiento del personaje, el sistema de disparos, las oleadas de enemigos, así como crear los enemigos, mover los enemigos hacia el jugador. y subir de nivel derrotando enemigos.
- **Arquitectura del Juego**: Estructurar el juego utilizando el framework DOTS, dividiendo el juego en sistemas y componentes para facilitar la gestión y el desarrollo.

Finalizando, en este apartado se ha descrito el desarrollo del proyecto. Tras esto, en el siguiente capítulo se explicará la implementación detallada de Unity DOTS.

CAPÍTULO 6
Implementación técnica

6.1 Implementación de Unity DOTS

En esta sección se explicarán los detalles del código sobre el framework Unity DOTS, en el desarrollo del videojuego 'Surge Survivor'. Así como servir de documentación para futuros proyectos. Se abordarán conceptos sobre el Entity Component System (ECS), y se incluyen ejemplos que ilustran la implementación de estas tecnologías.

Los ejemplos de código serán representados en el anexo A.1

Explicación del Código

Primero de todo, se explica como crear un sistema sencillo a partir de los datos asignados mediante Mono, y mediante el proceso de Baking a una entidad de la escena. En este sistema, se crea una entidad a partir de un prefab y se mueve en el eje X cada frame.

A continuación se explicarán los detalles del código, y como crear datos y asignarlos a una entidad durante la ejecución del juego.

6.1.1. Sistema Sencillo

1. **Importación de librerías**: Se importan las librerías necesarias para trabajar con Unity ECS, transformaciones, matemáticas, y Burst Compiler.

2. **Definición de EnemigosData**: EnemigosData es una estructura que implementa IComponentData y contiene la información necesaria para crear y mover enemigos. Como el prefab del enemigo y su velocidad. Se recomienda crear tanto el Data como el Mono en clases C Sharp diferentes.

3. **Clase EnemigosMono**: Esta clase es un MonoBehaviour que actúa como un script adjunto a un GameObject en la escena. Contiene una clase interna EnemigosBaker, que se encarga de convertir los datos del MonoBehaviour a componentes ECS.

4. **Clase EnemigosBaker**: Esta clase hereda de Baker<T>, y se encarga de convertir el prefab de enemigo y su velocidad a componentes ECS, que podrán ser utilizados por el SistemaSencillo más tarde.

5. **Definición del SistemaSencillo**: SistemaSencillo es una estructura que implementa un ISystem, y se compila mediante Burst Compiler para optimizar su rendimiento. Gestiona la creación y actualización de entidades en cada frame.

6. **Método OnCreate**: Este método se llama cuando el sistema se crea, y puede ser utilizado para inicializar cualquier estado necesario. Útil para obtener referencias de variables y clases al empezar el juego.

7. **Instanciación de Entidad** : Se crea una entidad a partir del Prefab obtenido anteriormente.

8. **Burst Compiler**: Se le indica al compilador en qué partes del código utilizar el Burst Compiler, en este caso lo usaremos en el cuello de botella. Es decir el bucle OnUpdate debido a su alto coste computacional, al ejecutarse cada frame del juego, de esta forma será optimizado.

9. **Método OnUpdate**: Este método se llama cada frame y contiene la lógica principal del SistemaSencillo.

10. **Movimiento de la Entidad**: Se actualiza la posición de la entidad en el eje X, mediante la variable de velocidad del enemigo.

11. **Creación de una entidad a partir del prefab**: Se crea la entidad del enemigo, a partir del Prefab creado anteriormente.

12. **Ejecución de Comandos**: Ejecución de los diversos comandos almacenados en el EntityCommandBuffer:

13. **EntityManager**: Obtención del EntityManager para gestionar entidades.

14. **EntityCommandBuffer**: Creación de un EntityCommandBuffer para realizar cambios en las entidades.

15. **Método OnUpdate**: Finalmente la liberación de los recursos utilizados por el EntityCommandBuffer.

Como se puede observar en el listing A.1 este código de ejemplo, puede actuar como una base sólida para crear sistemas y optimizarlos dentro de un juego utilizando Unity DOTS.

6.1.2. Sistema Balas

Seguidamente se explicara la creación de otro sistema así como los datos de sus componentes que creará y destruirá, además de como gestionar una entidad en tiempo de ejecución. Este ejemplo se puede encontrar en el listing A.2

A continuación, se muestra cómo buscar y destruir una entidad cuando su tiempo de vida ha terminado.

1. **Importación de librerías**: Como el caso anterior se importan las librerías necesarias para trabajar con Unity ECS, transformaciones, matemáticas, y Burst Compiler.

2. **Definición de BalaData**: BalaData es una estructura que implementa IComponentData y contiene la información necesaria para crear y mover balas, como la posición de la bala, su velocidad y su tiempo de vida.

3. **Clase BalaMono**: Esta clase es un MonoBehaviour, que actúa como un script adjunto a un GameObject en la escena. Contiene una clase interna BalaBaker que se encarga de convertir los datos del MonoBehaviour a componentes ECS.

4. **Clase BalaBaker**: Esta clase hereda de Baker<T>, y se encarga de convertir el prefab de la bala y su velocidad a componentes ECS, para que puedan ser utilizados por el SistemaBalas más tarde.
5. **Definición del SistemaBalas**: SistemaBalas es una estructura que implementa un ISystem y se compila con Burst para optimizar su rendimiento. Gestiona la creación y actualización de entidades en cada frame.
6. **Método OnCreate**: Este método se llama cuando el sistema se crea y puede ser utilizado para inicializar cualquier estado necesario, útil para obtener referencias de variables y clases al empezar el juego.
7. **Instanciación de Entidad**: Se crea una entidad a partir del prefab obtenido anteriormente de BalaData.
8. **Burst Compiler**: Indicación al compilador para compilar con Burst Compiler esta parte de código.
9. **Método OnUpdate**: Este método se llama cada frame y contiene la lógica principal del SistemaBalas.
10. **Movimiento de la Entidad**: Se actualiza la posición de la entidad en el eje Z mediante la variable de velocidad de la bala.
11. **Creación de una entidad a partir del prefab**: Se define el arquetipo de la bala y se crea la entidad.
12. **Ejecución de Comandos**: Ejecución de los comandos almacenados en el EntityCommandBuffer para aplicar los cambios realizados a las entidades.
13. **EntityManager**: Obtención del EntityManager para gestionar entidades.
14. **EntityCommandBuffer**: Creación de un EntityCommandBuffer para realizar cambios en las entidades.
15. **Método OnUpdate**: Finalmente ocurre la liberación de los recursos utilizados por el EntityCommandBuffer.

Este sistema aporta otra manera de usar un sistema en Unity, el cual puede crear y gestionar entidades en tiempo de ejecución mediante el código.

6.1.3. Trabajo Sencillo con IJobEntity

Siguiendo con los ejemplos, el siguiente proporciona una forma de crear un trabajo sencillo y ejecutarlo de forma paralela en diferentes hilos. El ejemplo se puede observar en el listing A.3

1. **Importación de librerías**: Igualmente se importan las librerías necesarias para trabajar con Unity ECS, transformaciones, matemáticas, y Burst Compiler.
2. **Definición del TrabajoSencillo**: TrabajoSencillo es una estructura que implementa un ISystem y se compila con Burst, para optimizar su rendimiento. Que gestionará el movimiento de una entidad mediante un Trabajo (Job) que se ejecutará paralelamente.
3. **Método OnCreate**: Este método se llama cuando el sistema se crea, y define una posición fija (posicionNueva) a la que los enemigos se moverán.

4. **Método OnUpdate**: Este método se llama cada frame y contiene la lógica principal del sistema. Se configura y programa el trabajo MoverEnemigosJob para que se ejecute en paralelo. Sus variables 'DeltaTime' y 'PosicionNueva' se pasan al trabajo, es de importancia el uso de mayúsculas en los nombres de las variables.
5. **Estructura MoverEnemigosJob**: MoverEnemigosJob es una estructura que implementa un IJobEntity y se compila con Burst, para optimizar su rendimiento. El método Execute será ejecutado por el Trabajo, y en el se define la lógica de movimiento de los enemigos. En el ejemplo, los enemigos se mueven hacia 'PosicionNueva' a una velocidad definida en EnemigosData.
6. **Movimiento de la Entidad**: Se actualiza la posición de la entidad en el eje Z, mediante la variable de velocidad del enemigo.
7. **EntityManager**: Obtención del EntityManager para gestionar entidades.
8. **EntityCommandBuffer**: Creación de un EntityCommandBuffer para realizar cambios en las entidades.
9. **Ejecución de Comandos**: Ejecución de los comandos almacenados en el EntityCommandBuffer.
10. **Liberación de Recursos**: Finalmente, la liberación de los recursos utilizados por el EntityCommandBuffer.

Con este último finaliza la explicación detallada de los diferentes ejemplos en Unity DOTS. Seguidamente se describirán los diferentes componentes y sistemas del proyecto.

6.2 Componentes y Sistemas

En cuanto a la implementación técnica del juego los componentes y sistemas son el núcleo del juego, controlando los aspectos principales de este. A continuación se presenta un esquema donde se explica su función y como los sistemas interactúan entre si.

Disparo Y Movimiento System

- **Función**: Este sistema controla el movimiento del jugador y permite disparar las balas. Emplea los inputs del jugador para actualizar la posición y rotación del mismo en función del botón presionado, así como permitir al jugador disparar para poder derrotar a los enemigos.
- **Interacción**: Recibe datos del Input System y actualiza los componentes de 'Transform' e interactúa con el Balas y Niveles system para gestionar ambos los proyectiles creados.

Balas Y Niveles System

- **Función**: Encargado de gestionar los diferentes datos y componentes relacionadas con el jugador y las balas creadas, paralelamente gestiona las colisiones de balas con enemigos y la lógica para dañarlas.
- **Interacción**: Controla las balas disparadas por el Disparo Y Movimiento System, además calcula el daño infligido a los enemigos colisionados, incrementa la experiencia del jugador y destruye las balas sobrantes en el cálculo de daño infligido.

Enemigo System

- **Función**: Se encarga de calcular la puntos de spawn de los enemigos y spawnearlos (instanciados en la escena) en sus posiciones. Además, incrementa la dificultad con cada oleada de enemigos.
- **Interacción**: Genera los enemigos, que luego Enemigos Moverse System, controla para posicionarlos y rotarlos hacia el jugador.

Input System

- **Función**: Un sistema escuchando los inputs realizados por el usuario, y los pone a disposición de otros sistemas que los necesiten.
- **Interacción**: Provee datos de input al Disparo Y Movimiento System, para poder realizar el movimiento del jugador, que es una Entidad.

Colisiones Enemigo Player System

- **Función**: Encargado de la lógica para colisiones entre el jugador y los enemigos. El sistema esta a la espera de una colisión mediante el método de distancia, una vez colisionados, destruye al enemigo para prevenir más daño al jugador, y decrementa los puntos de vida del jugador.
- **Interacción**: Detecta colisiones entre los enemigos spawneados por el Enemigos System, junto a los controlados por el Enemigos Moverse System, para que colisionen con el jugador.

Enemigos Moverse System

- **Función**: Es un sistema con un Job paralelo (IJobEntity) que mueve a los enemigos hacia la posición del jugador, así como rotarlos hacia este para que miren en la dirección del jugador, asimismo crea una animación para ir balanceando a los enemigos con una función sinusoidal.
- **Interacción**: Actualiza continuamente las posiciones de los enemigos para que se muevan hacia el jugador, utilizando las transformaciones del mismo.

Cámara System

- **Función**: Conecta la cámara de la escena con la entidad del jugador, permitiendo visualizar el movimiento del jugador desde una perspectiva en tercera persona. También crea un efecto de bobbing (en movimiento) a la cámara del jugador, para que al moverse el jugador el arma se mueva balanceándose de lado a lado, creando un efecto de movimiento más realista.
- **Interacción**: Sincroniza la posición y rotación de la cámara con los valores del movimiento de Disparo Y Movimiento System para seguir al jugador por escenario. Emplea el script 'Camara Singleton', en la cámara principal, para asignarle valores a la altura y distancia detrás del jugador, para así no colocarla dentro del jugador.

Estos sistemas se pueden observar en la Figura 6.1 en tiempo de ejecución, donde se ejecutan en el bucle Update, y podemos observar el número de entidades que controlan los sistemas en la columna subrayada en rojo.

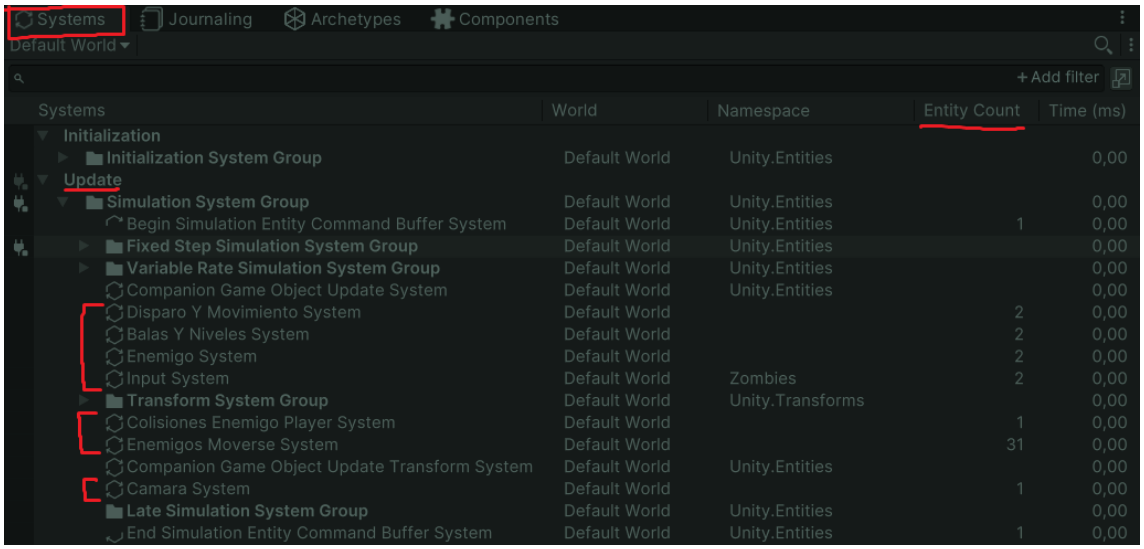

Figura 6.1: Ventana de Sistemas de ECS

6.3 Lógica del videojuego

Con la lógica del videojuego, nos referimos a cómo los diferentes componentes y sistemas interactúan para crear una experiencia de juego coherente y divertida. Seguidamente se explicara el flujo de acciones que ocurren.

6.3.1. GameManager

Fuera de los sistemas y componentes, se encuentra el GameManager, este es un componente esencial para gestionar el estado del juego. En Unity el GameManager es un objeto Singleton (una única instancia) que garantiza que los datos y sus funciones sean accesibles en cualquier parte del juego, asegurando una gestión unificada.

En el proyecto, el GameManager es un Prefab que se encuentra en la escena del menú principal y durante el juego, en ambos maneja la música, sonidos, estado de menús, así como coordinar el cambio de escenas. En la Figura 6.2 se pueden apreciar los componentes del GameManager que consisten en:

- **Transform**: en este caso la posición no importa.
- **Dos 'AudioSource'**: componentes que permiten reproducir audio y música.
- **Script**: Encargado de coordinar el cambio de escenas, guardar referencias a otros scripts y GameObjects y controlar el estado del menú de pausa.

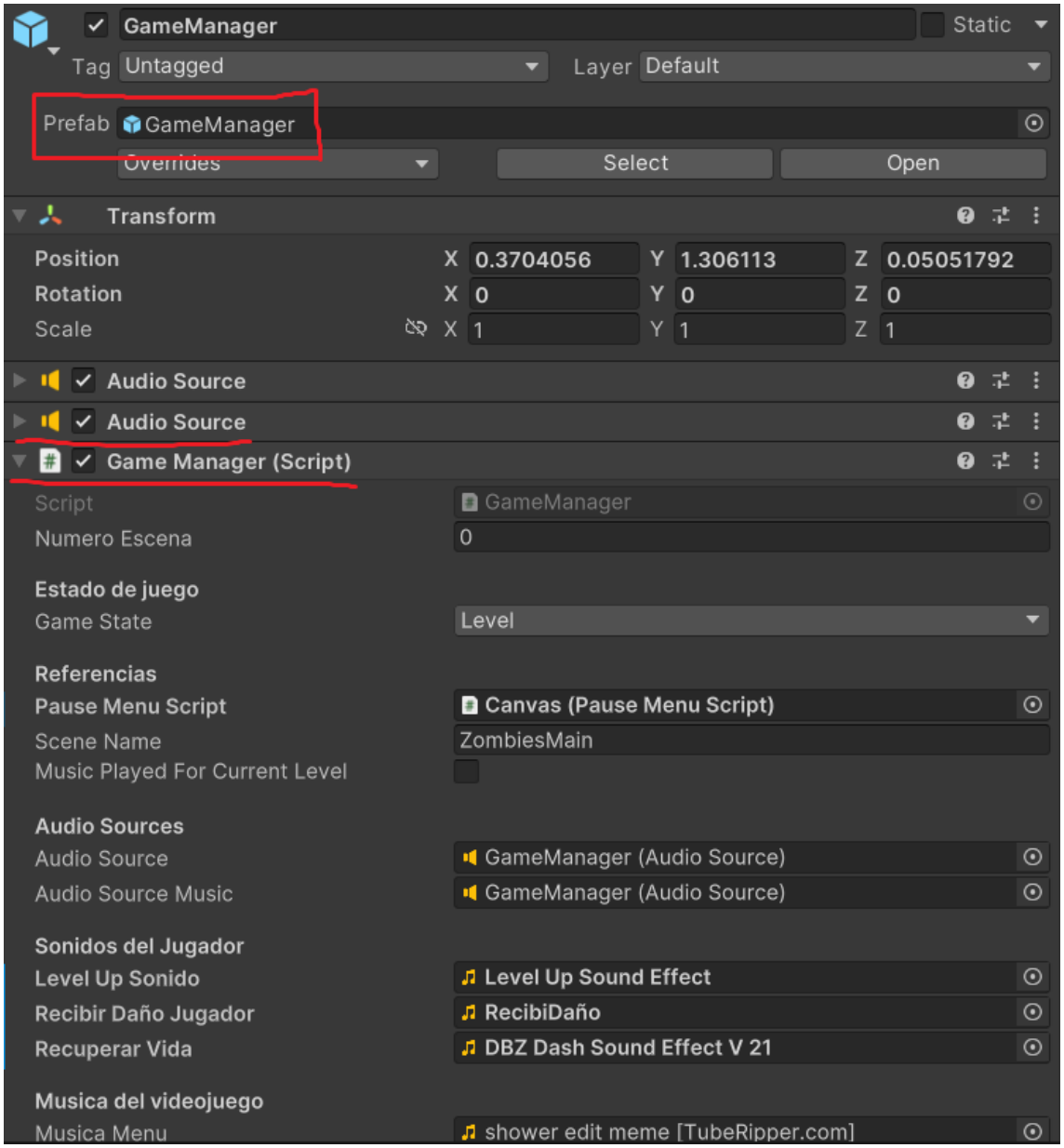

Figura 6.2: Prefab GameManager y sus componentes

Una vez descritos los diferentes sistemas del videojuego, la fase de inicialización, consiste en lo siguiente:

1. **Inicio del juego**: El juego comienza con la inicialización de los componentes y sistemas. Los enemigos son generados en sus posiciones iniciales mediante el Enemigo System, al jugador entidad se le asignan los componentes y datos necesarios, y el juego empieza.

2. **Captura de Input**: El Input System detecta las acciones del jugador, como movimientos y disparos. Esta información se pasa al Disparo Y Movimiento System.

3. **Actualización de la Cámara**: El Cámara System sincroniza la posición y rotación de la cámara con la del jugador, proporcionando una perspectiva en primera persona adecuada durante el juego.

4. **Spawning de Enemigos**: El Enemigo System calcula las posiciones de spawn de nuevos enemigos y los genera en el mapa. Con cada nueva oleada, se incrementa la dificultad de los enemigos.

Tras la fase de inicialización, los siguientes elementos del videojuego, ya estarían en funcionamiento:

- **Movimiento del Jugador**: Basado en los inputs, el Disparo Y Movimiento System actualiza la posición y rotación del jugador. También se encarga de gestionar los disparos, creando nuevas entidades de balas.
- **Gestión de Balas**: Los sistemas encargados a las balas, encargados de manejar la trayectoria y el daño de las balas. Cuando una bala impacta a un enemigo, se calcula el daño y se actualizan las estadísticas del jugador.
- **Movimiento de Enemigos**: Los enemigos se mueven continuamente hacia el jugador, gestionado por el Enemigos Moverse System, que utiliza un IJobEntity para ejecutar esta lógica en paralelo, mejorando el rendimiento.
- **Colisiones**: El sistema Colisiones Enemigo Player System detecta colisiones entre el jugador y los enemigos, reduciendo la vida del jugador y eliminando los enemigos impactados.
- **Progreso y Dificultad**: A medida que el jugador avanza y derrota enemigos, gana experiencia y niveles. El Balas Y Niveles System gestiona este progreso, mientras que el Enemigo System ajusta la dificultad de los enemigos para mantener el desafío.
- **Interfaz del juego (GUI)**: Gestionado por un script, actualiza los valores de la interfaz en tiempo de ejecución. Estos valores consisten en información útil para el jugador, como la vida y experiencia.

6.4 Movimiento del Jugador

Para obtener el input de la entidad del jugador se ha empleado un Sistema de Unity ECS. Este sistema deriva de 'SystemBase' y se encarga de crear una Entidad con el script InputMono, sino existe ya, y luego se asigna al 'InputPlayerECS', el Input del Input System dentro del paquete de Unity. En el bucle del Sistema de Input, se revisa si el jugador presiona la tecla de disparo y se cambia el valor de la variable.

Estos controles corresponden al movimiento y disparo del jugador, la configuración del Input System se puede ver en la Figura 6.3

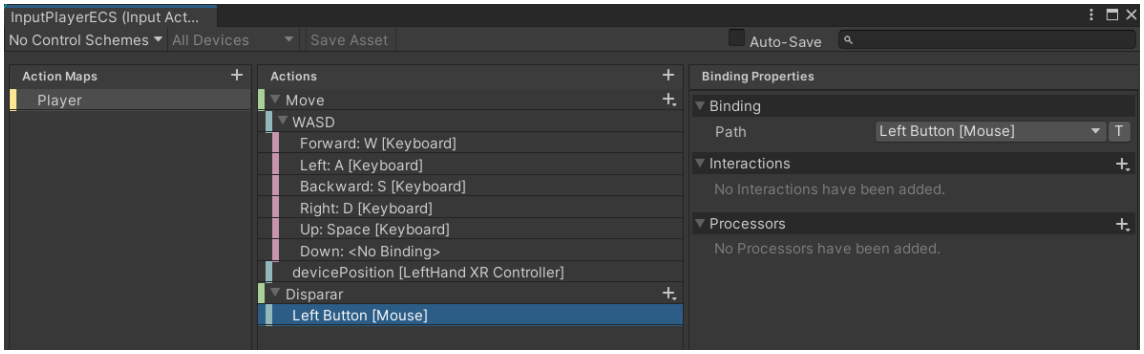

Figura 6.3: Configuración del Input System

Para el movimiento del jugador se han mapeado los siguientes botones:

Control	Acción
Click Izquierdo	Disparar y seleccionar mejoras
Movimiento Ratón	Mover la cámara horizontalmente, y el cursor
WADS	Movimiento del jugador
Escape	Menú de Pausa
Left Shift	Esprintar

Tabla 6.1: Controles del juego

De esta forma el jugador interactúa con el videojuego, de forma sencilla, al seleccionar botones comúnmente utilizados en la industria del videojuego.

6.5 Colisiones

Para detectar las colisiones dentro del videojuego, debido a la naturaleza de las Entidades y los GameObjects, estos no pueden interactuar entre si. Por lo que he empleado dos métodos para acercar el comportamiento de colisionar.

6.5.1. Método de capas

Primero el método de capas (layers) , en el script 'CapasColisiones.cs' se compone de dos partes principales. Una enumeración CapaColisiones que define las diferentes capas de colisión, y una clase DevolverCapa que contiene el método ObtenerCapaTrasColision, para determinar las capas involucradas en una colisión, y de esta forma obtener las 2 capas que colisionan.

Este método de colisiones ha sido empleado para detectar las colisiones entre las balas, paredes y los enemigos. La lógica que encarga de esto se realiza mediante el script BalasYNivelesSystem, que es un 'ISystem', mediante un 'NativeList<ColliderCastHit>' para almacenar todas las colisiones ocurridas y el método ObtenerCapaTrasColision, de esta forma podemos detectar colisiones entre estas Entidades. En la Figura 6.4 se pueden apreciar las capa con el valor Enemigo (capa), asignada a los Enemigos.

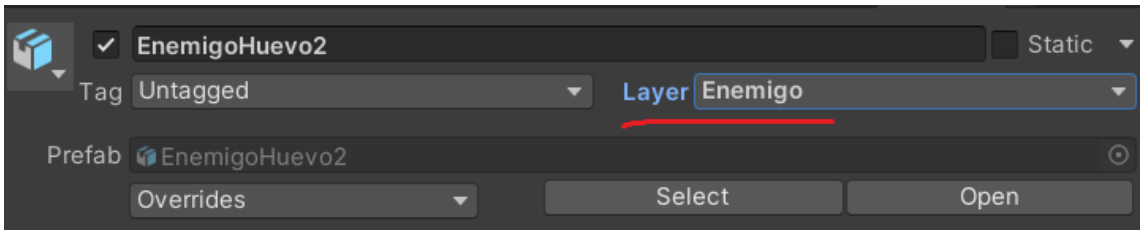

Figura 6.4: Capa enemigo

6.5.2. Método distancia

En cuánto al método de distancia. En la clase 'ColisionesEnemigoPlayerSystem.cs' detectamos si el enemigo esta lo suficientemente cerca del jugador, como para ser considerada una colisión y emular las físicas. Para lo cual mediante la función 'math.distance' que devuelve un 'float3' con la distancia entre dos vectores, siendo estos dos. El 'LocalTransform' (Vector de 3 coordenadas [x, y, z]) del enemigo y el jugador, y así comprobar si es menor a una distancia dada (1.0f), podemos detectar si han colisionado.

De esta dos maneras podemos simular que dos Entidades han colisionado, utilizando sus vectores de posición.

6.6 Disparos

El sistema de disparo (Disparo Y Movimiento) se activa cuando el jugador presiona el botón de disparo y el temporizador de disparo ha llegado a cero. En ese momento, se crean varias entidades de balas utilizando un Prefab predefinido. Las balas reciben su componente de 'DisparoData' que definen su velocidad, daño, tiempo entre disparos, así como un componente de tiempo que determina cuánto tiempo permanecen activas antes de desaparecer.

Cada bala se posiciona inicialmente en el punto de disparo del jugador y se le aplica una pequeña dispersión aleatoria, en forma de radio para simular la dispersión del tiro. Luego, las balas son instanciadas y sus componentes son actualizados con la nueva posición y rotación.

El otro sistema encargado de gestionar el movimiento de las balas, es 'Balas Y Niveles System', en este se extraen los componentes DisparoData y PlayerDañoData del jugador, que contienen información sobre el daño que las balas pueden infligir y la experiencia y puntuación del jugador.

Todas las balas disparadas se mueven hacia adelante en función de su velocidad y la dirección actual. Si una bala alcanza su tiempo de vida máximo sin colisionar con nada, se destruye para liberar recursos. Para detectar colisiones se emplea el método de capas mencionado en la sección de colisiones.

Cuando una bala colisiona con un enemigo, el sistema calcula el daño infligido por cada bala que impacte, si el daño es suficiente para derrotar al enemigo, este es eliminado, y el resto de balas continúan su trayectoria. Si el daño no es suficiente para destruir al enemigo, se reduce la vida del enemigo. Una vez un enemigo es destruido el jugador obtiene experiencia y puntuación. Si el jugador supera el umbral requerido para subir de nivel, el jugador sube de nivel, y aumenta el umbral de experiencia para el próximo nivel.

6.7 Enemigos

En el desarrollo de videojuego los enemigos componen una parte crucial de la dificultad del videojuego, así como complicar al jugador sus objetivos. En este proyecto, se han implementado tres tipos de enemigos: Normal, Fuerte y Rápido. Cada tipo tiene diferentes atributos, que crean comportamientos diferentes con el jugador.

Además, se ha implementado un sistema de creación y posicionamiento de los enemigos en el nivel, así como su movimiento hacia el jugador.

6.7.1. Tipos Enemigos

Se han creado tres tipos de enemigos para tener variedad, y ofrecer un problema diferente al jugador con cada uno. Para ello cada uno tiene diferentes atributos de 'EnemigosPropiedades'. Una descripción de estos enemigos se puede ver a continuación:

- **Enemigo Normal**: Es el enemigo normal, tiene vida y velocidad normales, y es el 1º en aparecer en la primera oleada, para familiarizar al jugador con los enemigos.

- **Enemigo Fuerte**: Este enemigo, posee más vida que el normal, haciendo más difícil acabar con el, para ello su velocidad se ha reducido para que sea el más lento.
- **Enemigo Rápido**: Se caracteriza por ser muy rápido, pero a cambio posee menos vida, haciendo más fácil su eliminación, el jugador es capaz de ser más rápido que el si mantiene presionada el botón de correr.

En la Figura 6.5 se pueden observar el diseño de los tres enemigos, desde la izquierda a la derecha, primero tenemos al enemigo Fuerte, Normal, y Rápido.

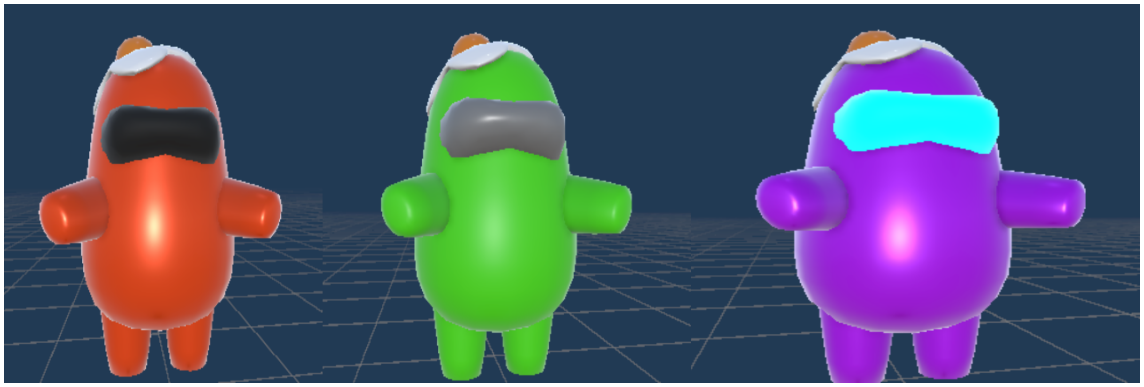

Figura 6.5: Diferentes enemigos

6.7.2. Escalado Enemigos

Para hacer más complicada la supervivencia del jugador, todos los enemigos van escalando sus características (vida y velocidad) según el número de oleada actual, de esta manera se mantiene la dificultad constante durante toda la partida, de esta forma tanto al principio como al final los enemigos supondrán un reto que el jugador deberá lidiar.

Esta decisión de escalado, se ha llevado a cabo ya que a niveles altos el jugador se vuelve muy poderoso, y rápidamente se pierde el interés en el juego. De igual modo, plantea al jugador elegir sus mejoras teniendo en cuenta esto, ya que por cada nivel que obtenga, el juego será más difícil y complejo.

6.8 Niveles y Experiencia

Para realizar más fácilmente la tarea de derrotar enemigos y sobrevivir, se ha implementado un sistema de niveles y experiencia que permiten al jugador volverse más fuerte. Cuándo el jugador derrota a un enemigo este incrementa la experiencia actual del jugador, al llegar a cierto número de experiencia el jugador sube de nivel (este se incrementa según una fórmula), pudiendo asi elegir una mejora de sus características, que le permitirá sobrevivir más tiempo y derrotar enemigos de manera más sencilla.

Para obtener estas mejoras el jugador deberá seleccionar una de entre tres mejoras posibles al subir de nivel, seleccionando del menú la que mas le convenga. Cada mejora se almacena en una lista, cada una con probabilidad (sobre 100) de ser seleccionada. Estas serán elegidas al azar, de esta manera el jugador deberá elegir la que sea más útil en el momento de subir de nivel.

En cuanto a las mejoras que puede obtener el jugador se encuentran en la tabla siguiente Tabla 6.2

Mejora	Descripción	Probabilidad
Velocidad	El jugador se mueve más rápido	20
Daño	Aumenta el daño realizado a enemigos	20
Vida	Aumenta la vida máxima del jugador	20
Área de balas	Aumenta el radio de dispersión de balas	10
Número de balas	Aumenta el nº de balas disparadas	10
Reducir daño recibido	Reduce el daño que recibe el jugador	10
Cadencia de disparo	Reduce el tiempo entre disparos	10

Tabla 6.2: Probabilidad de las mejoras

6.9 Interfaz del Juego

Dentro del videojuego la interfaz es fundamental para proporcionar al jugador información clave del estado del videojuego durante la partida. En este juego, se ha desarrollado un script llamado 'PlayerInterfaz.cs' encargado de gestionar la interfaz y actualizarla en tiempo de ejecución.

Debido a la naturaleza de las Entidades y los GameObjects, hay dos formas de realizar la interfaz de juego con Unity ECS, la primera basada en eventos, y la segunda consiste en leer datos de entidades desde un script que derive de MonoBehaviour. En este caso empleamos la segunda ya que PlayerInterfaz es un script mono.

Este acercamiento tiene una desventaja y es perder la implementación solo a base de Unity DOTS, aun así por sencillez se eligió esta opción. Leer datos de entidades es posible desde un script Mono a un script con ECS, de forma inversa no es posible realizar esta acción, es decir no se puede interactuar desde un script basado en ECS, con un script que derive de MonoBehaviour.

El script se ocupa de actualizar los valores de los puntos de experiencia del jugador, el nivel, la vida actual y los FPS (frames per second). Para ello gracias a la ayuda el 'EntityManager' podemos acceder a las componentes de las entidades desde un MonoBehaviour, y asi obtener el valor de los valores en tiempo de ejecución, así como poder cambiarlos mediante la función 'SetComponentData'. Asimismo, el script calcula las tres mejoras a escoger en el menú de subir de nivel, cuando el jugador sube de nivel. Paralelamente se incrementan los valores de los atributos del jugador cuando selecciona una mejora, para reflejar las acciones del jugador.

6.9.1. Tabla mejoras

En la tabla 6.3 se pueden apreciar las diferentes mejoras y los valores que incrementan o disminuyen. Algunos valores como la cadencia de disparo o la reducción de daño al jugador tienen un límite de valores, para prevenir interacciones extrañas. Como el jugador no recibiendo daño debido a varias mejoras de reducción de daño. O un tiempo entre disparos de 0 segundos, creando demasiadas balas.

Los valores asignados a estas mejoras, se han ajustado múltiples veces, durante el desarrollo del videojuego. Para crear una experiencia balanceada del juego. Para ajustar los valores iterado varias veces sobre los valores y la experiencia de juego, con estos ajustes el juego al principio se siente difícil y abrumador, pero a medida que el jugador obtiene más mejoras se vuelve más poderoso.

Mejora	Valor
Velocidad	+1.5f
Daño	+2.5f
Vida	+20
Área de Balas	+0.15f
Número de Balas	+2
Reducir daño recibido	-2.5f
Cadencia de disparo	-0.03f

Tabla 6.3: Mejoras del jugador

6.9.2. Jerarquía submenús

Dentro de Unity, los menús dentro del nivel principal están estructurados y organizados en submenús, a continuación se describirá brevemente sus características:

- **PanelTextoBarrasCrosshair**: Este panel se encuentra siempre activo, ya que muestra información esencial para el jugador en todo momento. Dentro se encuentran el punto de mira (crosshair), para ayudar al jugador dirigir las balas y conocer su trayectoria. Además, incluye las barras de vida y la barra de experiencia. La Figura 6.6 ilustra el contenido de la interfaz del juego.

Figura 6.6: Elementos Interfaz GUI

- **PanelMenuPausa**: Este submenú se activa cuándo el jugador presiona la tecla de escape y pausa el juego, de esta manera se puede pausar el juego y resumir en cualquier momento.
- **PanelSubirNivel**: En este submenú el jugador, deberá elegir una de las tres opciones disponibles, para ir fortalenciendo a su personaje. Para activarse el jugador debe subir de nivel.
- **MenuGameOver**: Muestra la puntuación final del jugador, una vez es derrotado. Esta puntuación incrementa al eliminar a un enemigo, de esta manera el jugador querrá superar su récord anterior, pudiendo medir su eficacia con la puntuación.

En la Figura 6.7 se observan los cuatro submenús, y como están organizados en la ventana de jerarquía de Unity.

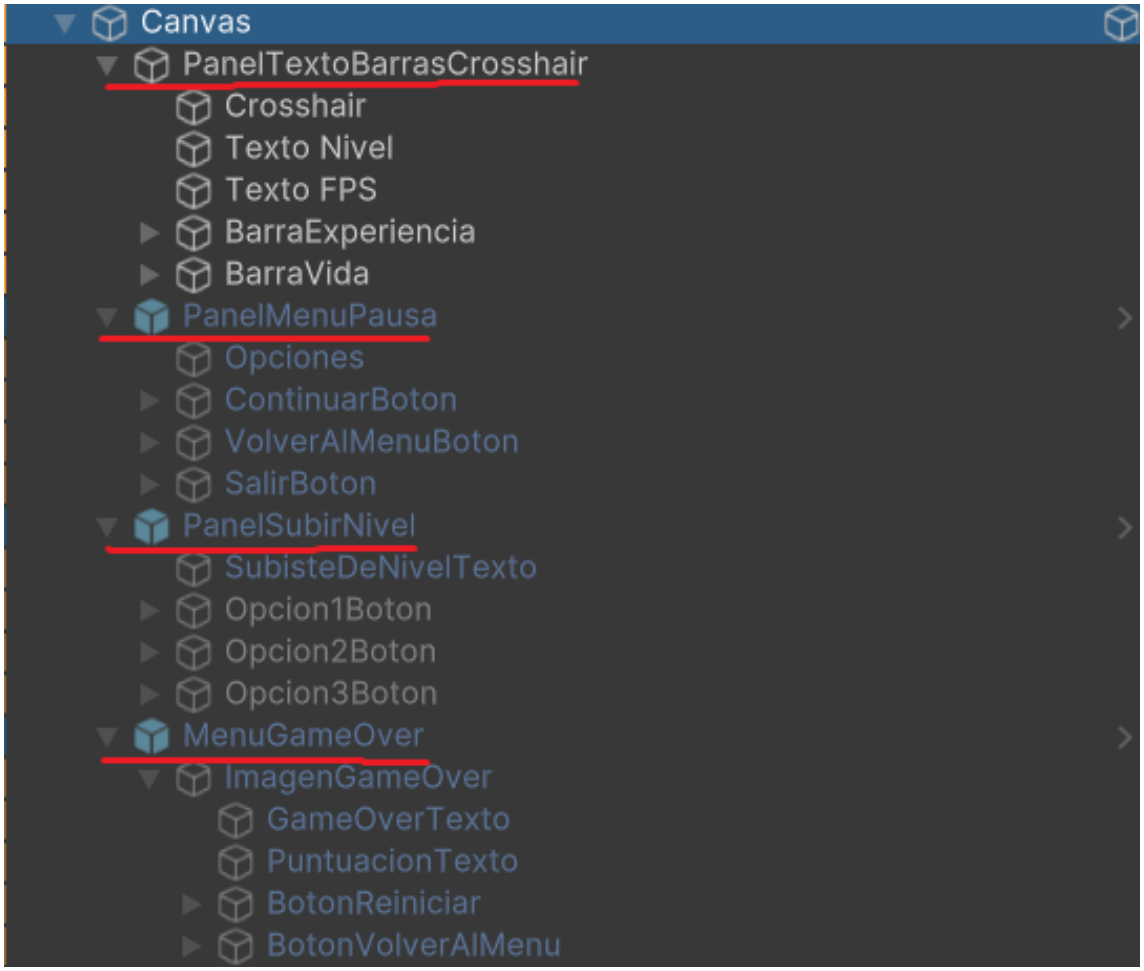

Figura 6.7: Menús dentro del videojuego

6.10 Optimización Rendimiento

Una medida tomada para optimizar el número de triángulos en pantalla, fue reducir el nº de polígonos de los modelos 3D de los objetos. Este proceso se llevó a cabo en Blender, mediante el modificador 'Decimate'. El modificador 'Decimate' es una herramienta que reduce el número de vértices, aristas y caras en un modelo 3D. Esto se logra simplificando la geometría del modelo, lo que resulta en una malla menos densa. El objetivo principal es disminuir el conteo de polígonos sin comprometer significativamente la calidad visual del modelo.

Otra medida realizada para optimizar el rendimiento del juego es utilizar el 'Oclusion Culling'. Esta técnica permite ahorrar tiempo de CPU y GPU evitando renderizaciones innecesarias.

En Unity el Oclusion Culling, se elabora mediante el proceso de 'baking', durante este proceso Unity divide la escena en celdas y genera datos que describen la geometría dentro de esas celdas y sus celdas adyacentes. Así en tiempo de ejecución se utilizan estos datos obtenidos para determinar lo que la cámara puede ver. De esta forma se evita el 'overdraw' (renderizar mismo elemento varias veces) y solo se renderiza aquello que se ve con la cámara disminuyendo el tiempo de CPU y GPU.

Con esto finaliza, el capítulo de la implementación técnica, y se pasa a explicar las pruebas realizadas.

CAPÍTULO 7
Pruebas

En este capítulo, se detallarán las pruebas realizadas, y los datos implicados en estas, así como los resultados y la descripción de los resultados.

7.1 Pruebas realizadas

En este apartado se presentan las pruebas de rendimiento que se han realizado para verificar que la solución funciona correctamente, para comprobar que el sistema cumple con las expectativas, y las especificaciones técnicas. También se incluyen pruebas de carga para evaluar la eficiencia y el consumo de recursos del videojuego.

Para medir el rendimiento del juego, se utilizaron los FPS (frames por segundo) como indicador principal de la fluidez del juego. Además, se empleó la herramienta Unity Profiler, para obtener datos detallados sobre el rendimiento del juego en diferentes escenarios.

Cáculo de los FPS

En cualquier videojuego es importante conseguir un número de FPS alto y constante, ya que determinan una experiencia fluida y realista. Para obtener el número de FPS dentro de Unity se calcula el tiempo entre frames. Primero se obtiene el tiempo del primer frame renderizado, y posteriormente el tiempo del segundo frame, el tiempo transcurrido entre el primer y segundo frame son los frames por segundo.

Unity proporciona un valor de estos tiempos llamado 'Time.unscaledDeltaTime', que proporciona el delta time sin tener en cuenta la escala del tiempo, es decir, no es afectado por interrupciones o cambios en el juego. Con este valor podemos calcular un promedio ponderado para suavizar las variaciones de tiempo entre frames, adquiriendo un número estable de FPS.

El Unity Profiler permite analizar varios aspectos del rendimiento, como el uso de la CPU, la GPU, la memoria y la red. A continuación, se describen los diferentes datos que se pueden obtener con el Unity Profiler y los escenarios de prueba utilizados. En la Figura 7.1 se ilustra el funcionamiento del Unity Profiler. Donde podemos observar diferentes apartados como el uso actual de CPU, el coste de renderizar objetos en pantalla, y el uso de la memoria.

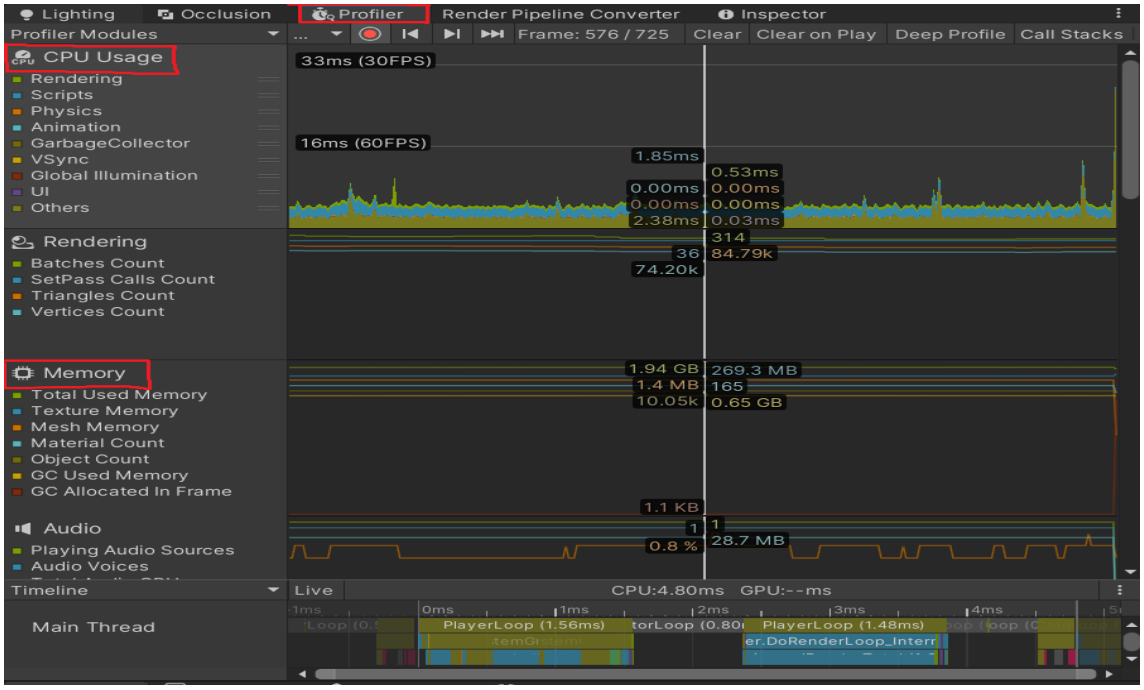

Figura 7.1: Datos de Unity profiler

7.2 Resultados de las pruebas realizadas

Sin emplear Burst Compiler

Para desactivar el BurstCompiler, es tan sencillo como dirigirse a la barra de menús, en Jobs, desplegar Burst, y desactivar el checkbox de 'Enable Compilation'. De esta forma podemos activarlo y desactivarlo para realizar pruebas. En la Figura 7.2 se ilustra este proceso. Desactivando el BurstCompiler obtenemos una forma de evaluar el rendimiento, similar a la forma tradicional que se realiza en Unity. Así se obtiene una comparación cercana a los GameObject que emplea Unity.

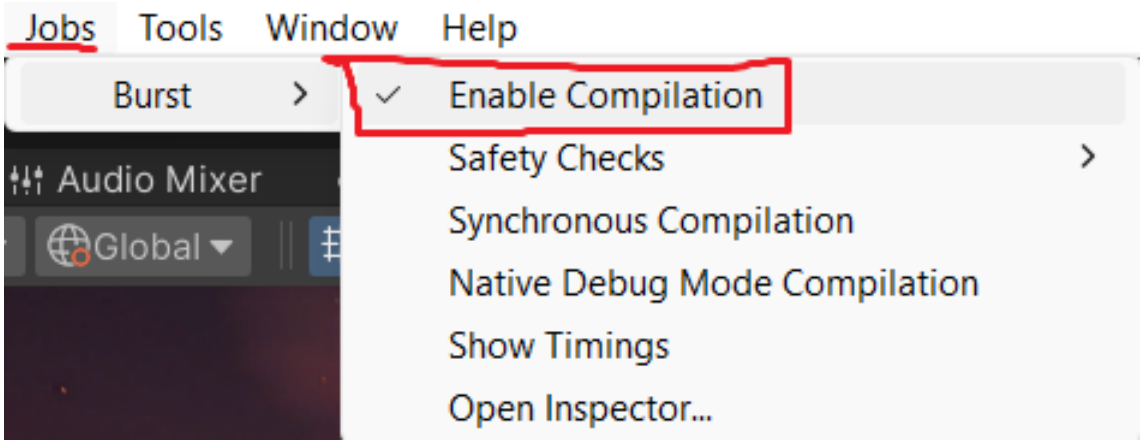

Figura 7.2: Proceso de desactivar Burst Compiler

Para realizar las pruebas de rendimiento se han empleado dos ordenadores, un ordenador portátil (Gaming) y uno de sobremesa (media-alta gama) con las siguientes especificaciones hardware:

Ordenador Portátil

- **Procesador**: 12th Gen Intel(R) Core(TM) i5-12500H, 2500 Mhz, 12 procesadores principales, 16 procesadores lógicos (16 hilos)
- **Memoria física instalada (RAM)**: 8,00 GB
- **Memoria virtual total**: 22,2 GB
- **Gráfica**: NVIDIA GeForce GTX 4050 Laptop GPU
- **Memoria total aprox grafica**: 9877 MB
- **Sistema Operativo**: Windows 11 Home

Ordenador de Sobremesa

- **Procesador**: 5th Gen Intel(R) Core(TM) i5-6500H, 3200 Mhz, 4 procesadores principales, 4 procesadores lógicos (4 hilos)
- **Memoria física instalada (RAM)**: 16,00 GB
- **Memoria virtual total**: 18,3 GB
- **Gráfica**: NVIDIA GeForce GTX 950
- **Memoria total aprox grafica**: 10139 MB
- **Sistema Operativo**: Windows 10 Home

7.2.1. Sin emplear Burst Compiler

En esta primera prueba, se empleo el ordenador portátil y el ordenador de sobremesa, donde el juego se ejecutó sin utilizar el Burst Compiler. Los resultados mostraron un rendimiento básico con valores de FPS que variaban dependiendo de la cantidad de enemigos y balas en pantalla. El Unity Profiler reveló que la CPU estaba realizando una gran cantidad de trabajo en cada frame, con tiempos de procesamiento elevados para las tareas de actualización y renderizado.

El análisis del Unity Profiler obtenido en el ordenador portátil se puede observar en la Figura 7.3 que muestra el uso de la CPU, carga de los objetos renderizados, y el uso de la memoria de forma general.

Figura 7.3: Primera prueba del Unity Profiler visión general

En cambio en la Figura 7.4 se puede observar el uso de la visión general y el Uso de la CPU en el ordenador de sobremesa.

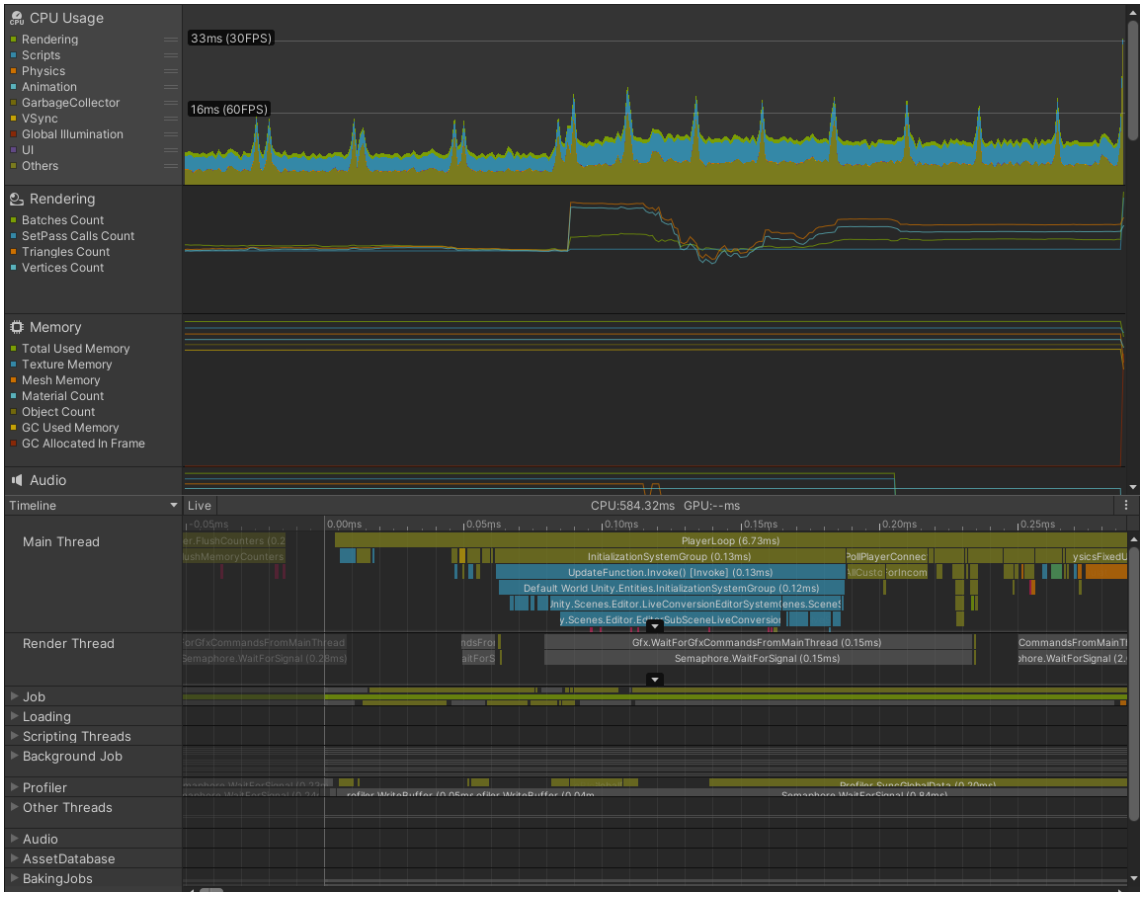

Figura 7.4: Primera prueba del Unity Profiler visión general, en el ordenador de sobremesa

A continuación se muestran los valores de FPS, CPU y Memoria conseguidos desde el Unity Profiler:

- **FPS portátil**: Promedio de 80-100 FPS
- **FPS PC sobremesa**: Promedio de 100-110 FPS.
- **Uso de CPU**: Elevado, con picos significativos durante las oleadas de enemigos.
- **Uso de Memoria**: Moderado, con algunas variaciones durante el juego.

En cambio en la Figura 7.5 se muestra el apartado de CPU de forma completa, donde podemos observar las funciones ejecutándose en el hilo principal (Main Thread), el hilo encargado del renderizado (Render Thread), los trabajos que se están ejecutando, así como otros trabajos como el audio, el proceso de baking, y el propio Unity Profiler.

En la parte superior de la imagen se puede observar un tiempo de CPU de 339,70 ms (milisegundos) para el portátil, mientras que el PC de sobremesa obtuvo un tiempo de CPU de 584.32ms, este valor nos señala el tiempo total que la CPU emplea procesando todas las tareas en un fotograma. Este valor obtenido de tiempo, es importante para entender el rendimiento del juego, ya que un tiempo de CPU elevado puede ser indicativo de que el juego está sobrecargando la CPU. Obtener un número bajo de FPS (Frames Per Second) implica afectar a la fluidez y jugabilidad del juego.

En este caso, la mayoría del tiempo que la CPU esta ocupada se distribuye entre los hilos principales, realizando trabajos paralelos y renderizando los objetos en pantalla, sugiriendo que el juego está realizando operaciones costosas que podrían ser beneficiadas, por una optimización adicional.

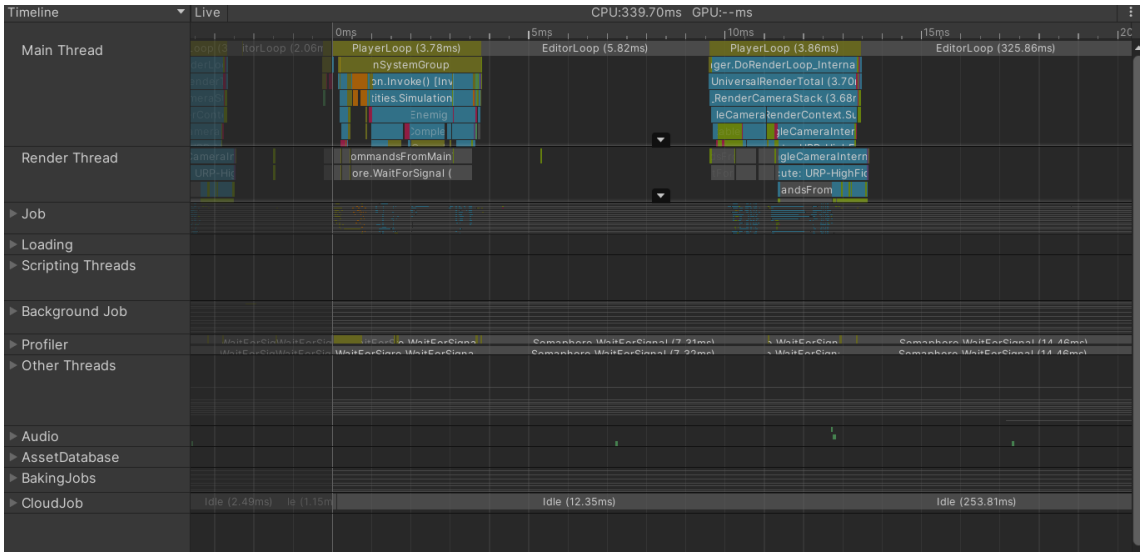

Figura 7.5: Primera prueba del Unity Profiler CPU usage en el portátil

En el apartado Rendering, podemos observar el número de triángulos renderizadonse en pantalla en el frame actual. A la hora representar objetos 3D, se utilizan triángulos al ser la figura geométrica más sencilla de representar, y estructurando triángulos se pueden representar todos los modelos 3D. En la Figura 7.6 se puede apreciar un número de triángulos de 218,2k, junto con el espacio que ocupan en memoria mantener sus texturas cargadas y el tamaño de los buffers. En cambio en la Figura 7.7 se observan un menor número de triángulos (165,3k), en cambio se realizaron más llamadas a la función de dibujo, así como un uso mayor de la memoria.

En pantalla estos triángulos corresponden, a los enemigos, el arma del jugador, proyectiles disparados, terreno del escenario, indicando así el número de objetos en pantalla simultáneamente.

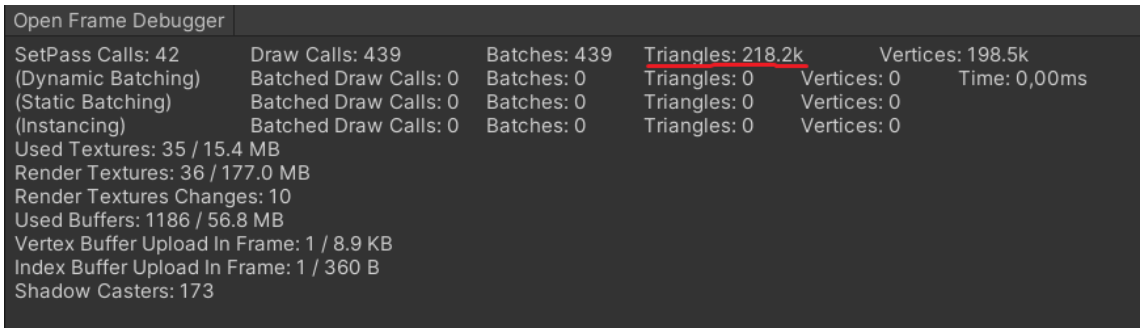

Figura 7.6: Primera prueba del Unity Profiler rendering en el portátil

Figura 7.7: Unity Profiler memoria obtenido en el PC sobremesa

En la Figura 7.8 se muestran los datos de memoria obtenidos en el portátil, sin emplear el Burst Compiler. A continuación se detallaran los aspectos y datos relevantes obtenidos en el portátil:

- **Memoria Total Comprometida**: 1,65 GB (Giga Bytes). Representa la memoria actual que utiliza el juego actualmente en ejecución.
- **Memoria Total Reservada**: 2,17 GB (Normalizada). Indica la cantidad de memoria que se ha reservado en el sistema. De esta forma se dispone de memoria en futuros casos.
- **Montón Administrado**: 329,8 MB (Mega Bytes). Esta es la memoria gestionada por el sistema de gestión de memoria de Unity, consiste en objetos como scripts y estructuras de datos
- **Gráficos y Otros**: 317,2 MB. Memoria utilizada para texturas y mallas de modelos así como otros recursos gráficos.
- **Profiler**: 450 MB, memoria utilizada por el propio Unity Profiler.
- **Memoria Sistema**: 49,5 MB. Manifiesta la memoria usada por el sistema operativo para otros procesos.

Estadísticas de Objetos:

- Texturas: 994

- Mallas de objetos: 92
- Materiales : 156
- Objetos en escena: 660

Por otro lado en la Figura 7.9 se aprecian los datos conseguidos en el PC de sobremesa, que se describirán a continuación:

- **Memoria Total Comprometida**: 1,61 GB (Giga Bytes).
- **Memoria Total Reservada**: 2,32 GB (Normalizada).
- **Montón Administrado**: 434,3 MB.
- **Gráficos y Otros**: 268,3,4 MB.
- **Profiler**: 316,4 MB,
- **Memoria Sistema**: desconocida.

Estadísticas de Objetos:

- Texturas: 919
- Mallas de objetos: 218
- Materiales : 145
- Objetos en escena: 738

El análisis de estos datos de memoria, muestra que el juego está utilizando una cantidad significativa de recursos gráficos, y de código ejecutándose. La diferencia notable entre la memoria total comprometida y la reservada, implica que hay espacio para optimizar en cuanto al manejo de la memoria en el juego.

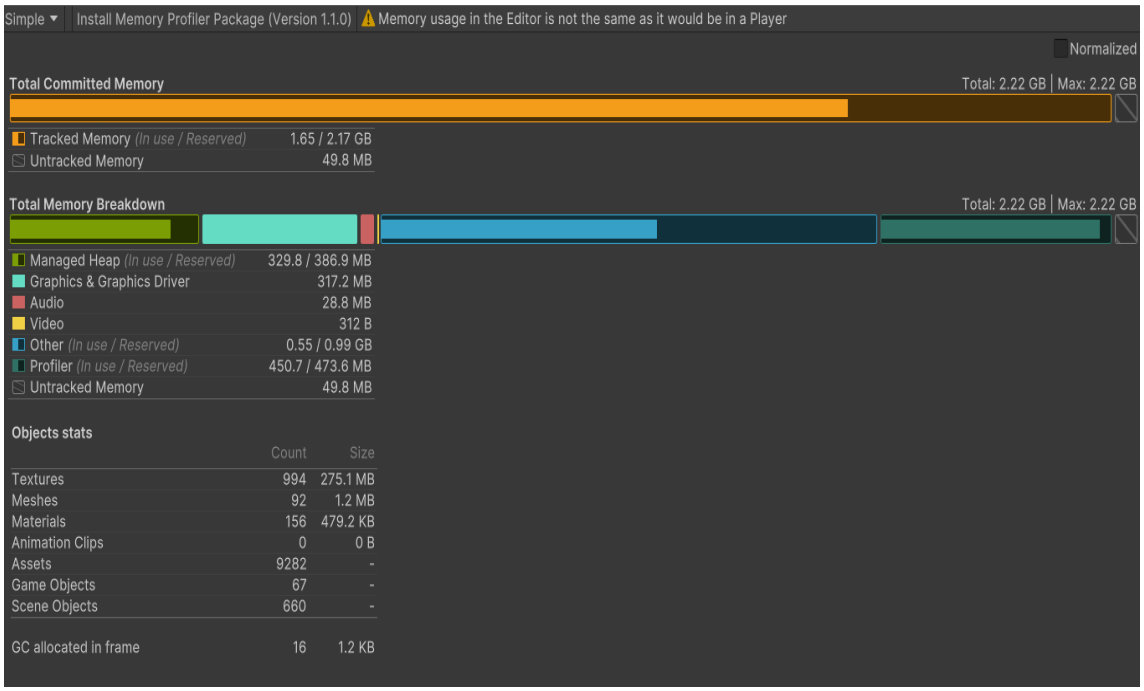

Figura 7.8: Primera prueba del Unity Profiler Memoria

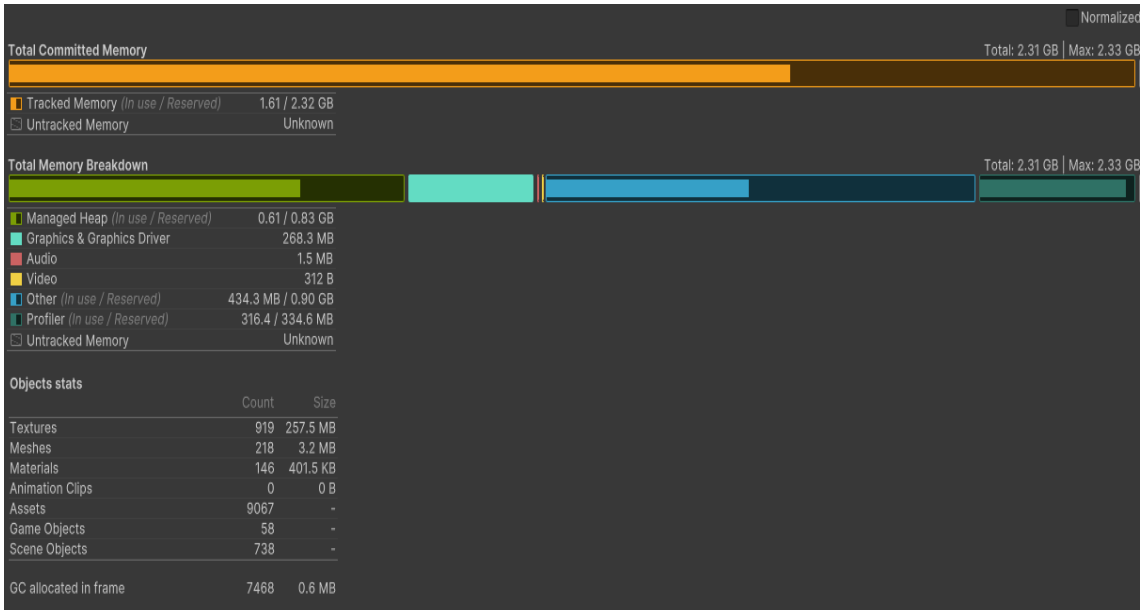

Figura 7.9: Unity Profiler memoria obtenido en el PC sobremesa

7.2.2. Empleando Burst Compiler

En esta segunda prueba, se utilizó el Burst Compiler para optimizar el código. Debido a que Burst Compiler compila el código C Sharp en código máquina altamente optimizado, este mejorará significativamente el rendimiento. Antes de realizar las pruebas, hay que volver a activar Burst Compiler. Tras esto los resultados conseguidos mostraron una mejora notable en los FPS y una reducción en el uso de la CPU, junto a un incremento del uso de Memoria.

- **FPS**: Promedio de 120-130 FPS.
- **FPS**: PC de sobremesa: Promedio de 140-150 FPS.
- **Uso de CPU**: Elevado, con picos significativos durante las oleadas de enemigos.
- **Uso de Memoria**: Moderado, con algunas fluctuaciones durante el juego.

En la Figura 7.10 se ilustran los datos obtenidos en el Unity Profiler usando el portátil, los cuáles nos indican un incremento en el número de FPS, así como un menor uso de CPU. Por otra parte en al Figura 7.11 tenemos los datos obtenidos utilizando el ordenador de sobremesa, donde obtenemos un ligero incremento de eficiencia.

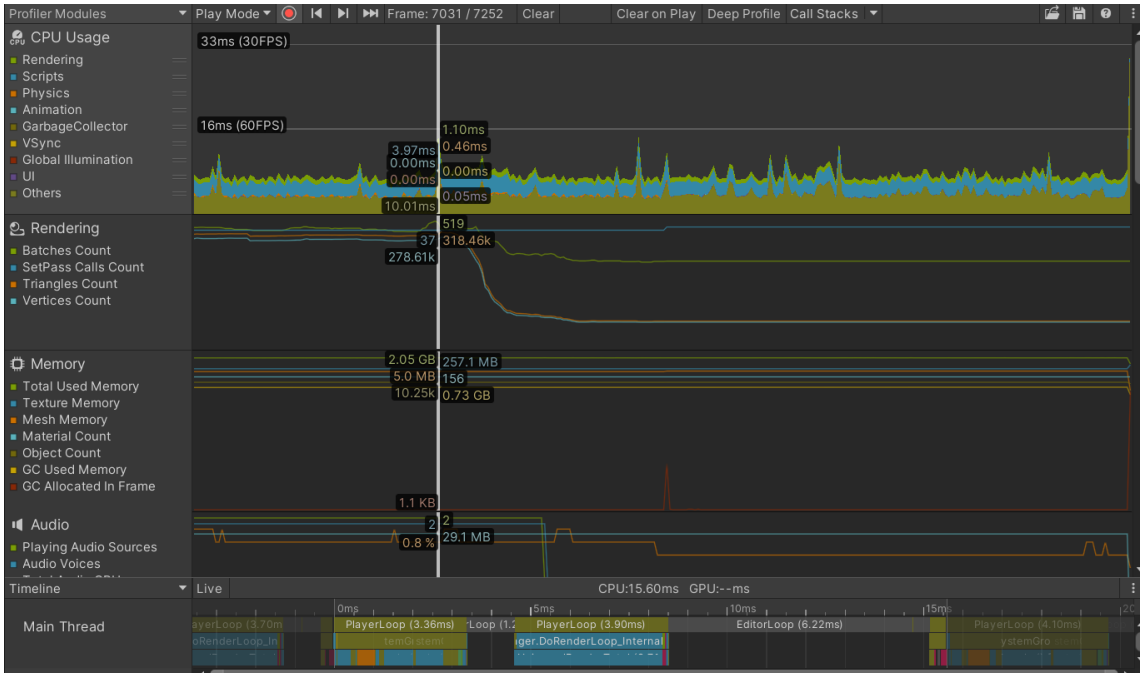

Figura 7.10: Segunda prueba del Unity Profiler visión general

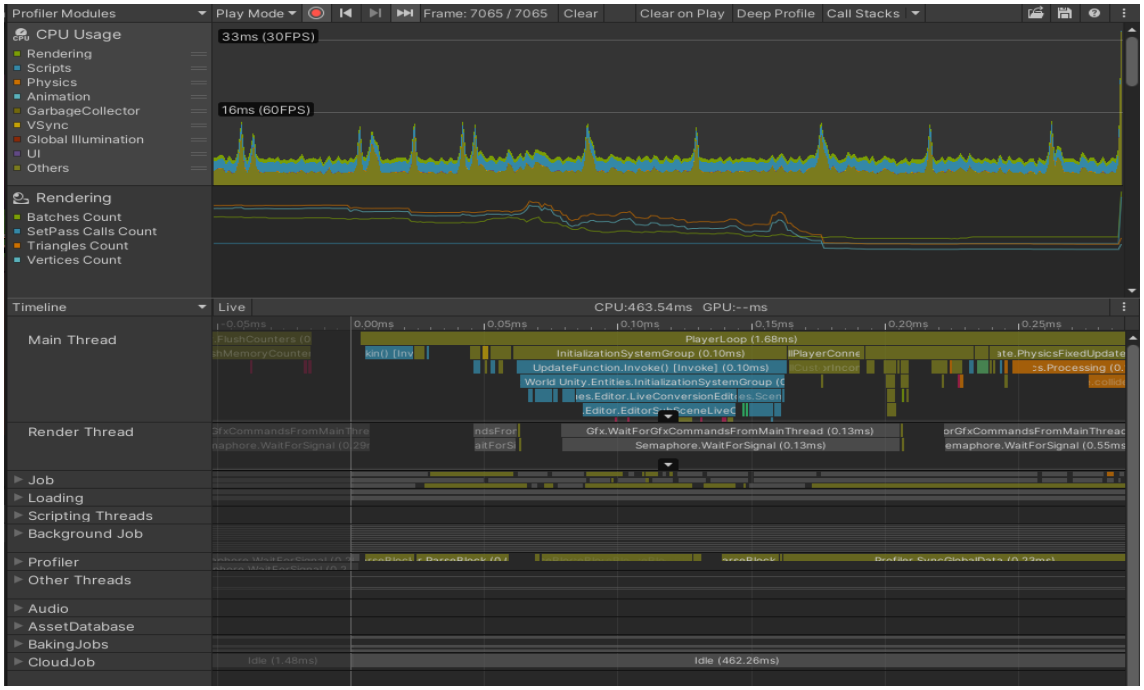

Figura 7.11: Visión general del Unity Profiler obtenido en el PC sobremesa

Como era de esperar en la Figura 7.12 se observan los datos obtenidos sobre la CPU del portátil, en este caso se obtuvo un tiempo de CPU significativamente reducido de 15,10 ms. En cambio en el PC de sobremesa se obtuvo un tiempo de CPU de 463,64ms. Esta mejora se debe a la optimización que Burst Compiler proporciona al convertir el código de alto nivel, en código de máquina altamente optimizado, aprovechando las características de la arquitectura de la CPU de forma eficiente.

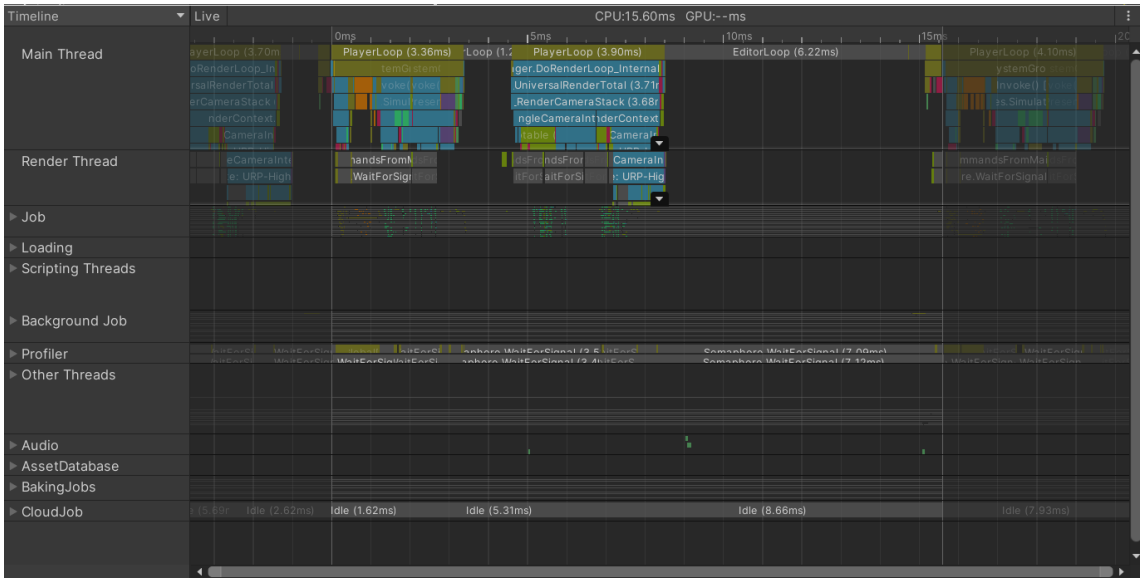

Figura 7.12: Segunda prueba del Unity Profiler CPU usage

En el frame donde se obtuvo el menor uso de CPU, también se puede observar un mayor número de triángulos en pantalla a renderizar, en este caso cien mil triángulos más. Esto contribuye a la mejora del rendimiento de la CPU obtenido con Unity DOTS, donde en un videojuego tradicional creado con GameObjects a mayor número triángulos en pantalla menor número de FPS se obtendrán a causa del mayor tiempo renderizando en pantalla, pero gracias a Burst obtenemos más FPS.

El estado del apartado rendering obtenido en el portátil se puede apreciar en la Figura 7.13, asimismo en la Figura 7.14 se puede observar el conseguido con el PC de sobremesa. Donde igual que en la prueba sin Burst Compiler, se obtiene un número alto de llamadas a la función de dibujar, mientras se tiene menos triángulos renderizandose en pantalla por parte del sobremesa.

Figura 7.13: Segunda prueba del Unity Profiler Rendering

Figura 7.14: Apartado rendering obtenido en el PC de sobremesa

Por otro lado con Burst Compiler activado, la memoria total comprometida (2,05 GB) aumentó, y asimismo la memoria total reservada incremento (2,76GB). Este incremento se debe a la mayor cantidad de objetos en escena, sin embargo este aumentó no indica un peor rendimiento de la memoria, de hecho, refleja una mejor gestión y optimización de los recursos por parte de Burst Compiler. En la Figura 7.15 se observan los nuevos datos obtenidos durante la segunda prueba en el portátil:

- **Memoria Total Comprometida**: 2,05 GB (Giga Bytes), en la prueba anterior 1,61 GB.
- **Memoria Total Reservada**: 2,76 GB (Normalizada), anteriormente 2,32 GB.
- **Montón Administrado**: 0.61 GB, que comparte el valor anterior.
- **Gráficos y Otros**: Disminuyo ligeramente de 317,2 MB a 284,5 MB, lo que sugiere una pequeña optimización gráfica.
- **Profiler**: 316 MB, coincide con el valor anterior.
- **Memoria Sistema**: 68,1 MB, esta vez si se obtuvo la memoria del Sistema.

Estadísticas de Objetos:

- Texturas: 927
- Mallas de objetos: 292
- Materiales : 148
- Objetos en escena: 825

Por otra parte en la Figura 7.16 se muestran los valores obtenidos por el PC de sobremesa así como los valores obtenidos a continuación:

- **Memoria Total Comprometida**: 1,65 GB (Giga Bytes), en la prueba anterior 0,65 GB.
- **Memoria Total Reservada**: 2,45 GB (Normalizada), anteriormente 2,17 GB.
- **Montón Administrado**: 0.73 GB. El cambio más significativo se refleja aquí, donde incremento de 329 MB a 0,73GB. A primeras puede parecer un aumentó considerable del uso de memoria, es importante comprender que Burst Compiler optimiza el código nativo, resultando en un mayor uso inicial de memoria, a cambio de un mayor rendimiento.
- **Gráficos y Otros**: Disminuyo ligeramente de 317,2 MB a 301,7 MB, lo que sugiere una pequeña optimización en el apartado de recursos gráficos.

- **Profiler**: 452,2 MB, tuvo un ligero incremento respecto al anterior.
- **Memoria Sistema**: 203,5 MB, la memoria del sistema aumentó, reflejando así un mayor uso de recursos del sistema para gestionar eficientemente el juego.

Estadísticas de Objetos:

- Texturas: 1012
- Mallas de objetos: 371
- Materiales : 156
- Objetos en escena: 948

El crecimiento ligero del número de texturas, mallas y objetos indica que Burst Compiler, permitió gestionar y optimizar estos recursos, para obtener un mayor número de recursos en pantalla.

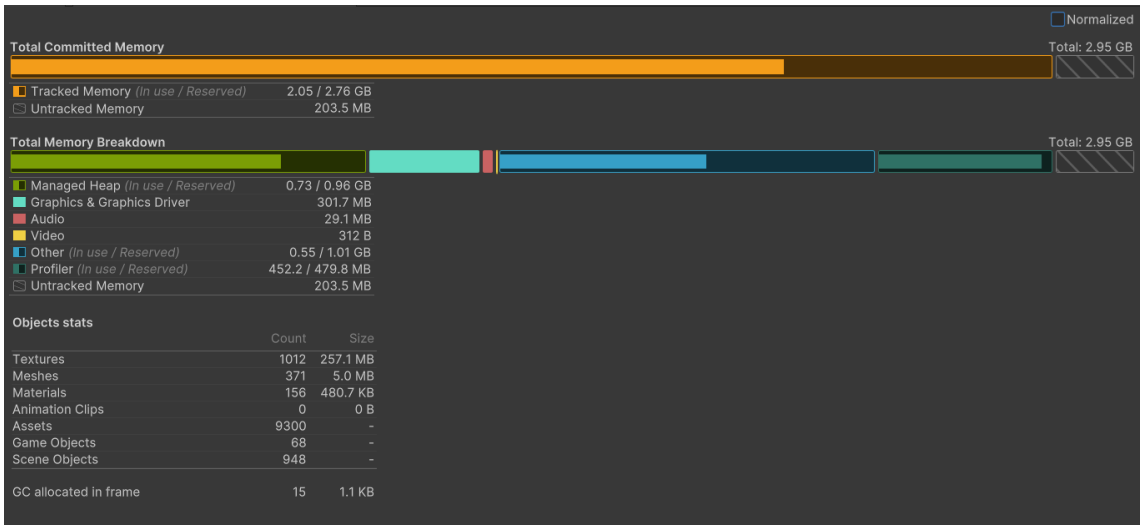

Figura 7.15: Segunda prueba del Unity Profiler Memoria

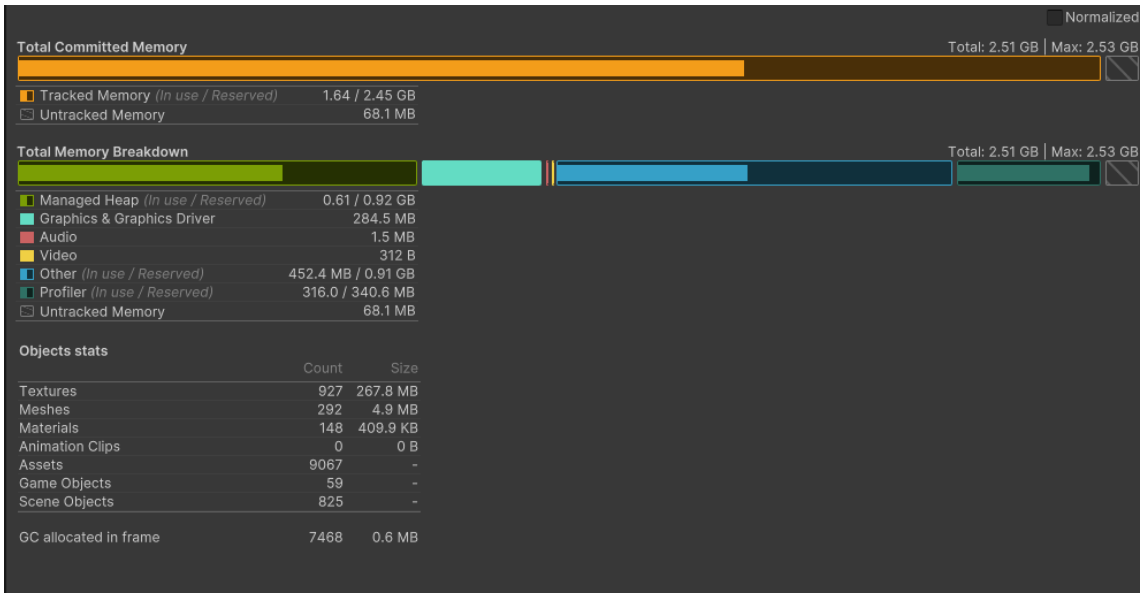

Figura 7.16: PC Sobremesa valores obtenidos con el Unity Profiler en Memoria

Con esto concluyen las dos pruebas realizadas dentro del editor de Unity.

7.2.3. Ejecutable del juego

En la tercera prueba, se evaluó el rendimiento del juego en su formato ejecutable, es decir, el juego compilado y ejecutado fuera del entorno de desarrollo de Unity, para realizar este ejecutable según lo explicado por Thompson [7], hay que realizar varios pasos antes de compilar el ejecutable, como cerrar la 'sub escena', revisar como se obtienen las variables 'Singletones', comprobar que el tipo de las variables sean nativas, si se emplea Burst Compiler, y finalmente borrar la caché de entidades. De esta forma se pudo compilar exitosamente el ejecutable del juego. Los resultados obtenidos mostraron la mejor optimización, con un rendimiento superior debido a la eliminación de la sobrecarga del entorno de desarrollo.

Al principio, en ambos ordenadores se obtuvo un valor de FPS menor al esperado, esto se debía al tener activada por defecto la sincronización virtual en el ejecutable del juego y el monitor del ordenador de sobremesa tener 60hz (hercios), y en el portátil de 144hz, lo cual limita la tasa de FPS. El 'VSync' (Vertical Sync) se encarga dentro del juego sincronizando la tasa de cuadros por segundo con la frecuencia de actualización de la pantalla para evitar el 'tearing' de la pantalla (divisiones en la pantalla), activándolo se obtiene una visualización perfecta de la pantalla sin los defectos del 'tearing'. En este proyecto debido al interés del rendimiento y con el fin de realizar las pruebas se desactivará esta opción para comprobar la potencia del hardware y el framework Unity DOTS.

Para quitar este límite se cambio en los ajustes de Unity, dentro de 'Quality' (Calidad gráfica) cambiar el valor de 'VSync Count' a 'Don't Sync', de esta manera los FPS no estarán limitado a 60FPS o 144FPS y se podrán mostrar tantos FPS como el hardware pueda generar.

- **FPS ordenador portátil**: Dentro del juego, se obtuvieron unos FPS que rondaban los 140-150 FPS. Tras desactivar el VSync, se obtuvo un gran incremento de entre 350-400 FPS, el cuál indica que se obtuvo un excelente rendimiento del juego, esto implica un incremento del 158,62 por ciento, respecto a emplear VSync.
- **FPS ordenador sobremesa**: Se consiguieron unos FPS que rondaban los 60 FPS en la primera prueba. Después de desactivar el VSync, se consiguieron unos 190-200 FPS. Obteniendo así un incremento del 225 por ciento.

Finalmente en la versión ejecutable del videojuego, debido a que no se puede emplear el Unity Profiler, se opto por medirlo mediante los FPS. En la Figura 7.17 se puede observar el número de FPS ubicados en la esquina superior derecha, obtenidos con el portátil con el VSync activado, y en la Figura 7.18 el número de FPS con este desactivado.

Confirmando que el código fue optimizado por el Burst Compiler tras ejecutar el ejecutable, de esta forma podemos observar que efectivamente tras ser compilado el código y mejorado para ser ejecutado mejora el rendimiento del juego de forma notable, obteniendo resultados de rendimiento altos. El mejor rendimiento obtenido, puede deberse a la diferencia entre los núcleos lógicos entre ambos portátiles, o debido a la gráfica, que en los videojuegos es la que más cálculos realiza al renderizar los gráficos.

Figura 7.17: Resultados de FPS obtenidos durante el ejecutable, con el VSync activado y en el portátil

Figura 7.18: FPS obtenidos durante el ejecutable, sin el VSync activado y en el portátil

Asimismo, en el ordenador de sobremesa también incremento el número de FPS de forma notable tras desactivar el VSync. En la Figura 7.19 se aprecian los FPS sin el VSync, y en la Figura 7.20 se muestran los valores obtenidos tras desactivar esta opción. Como se puede ver el incremento también es apreciable.

Figura 7.19: Resultados obtenidos en el ordenador de sobremesa durante el ejecutable, con el VSync activado

Figura 7.20: Resultados obtenidos en el ordenador de sobremesa durante la ejecución del juego, con el VSync desactivado

7.2.4. Conclusión pruebas

El uso de Burst Compiler en Unity DOTS resultó en una mejora notable en los aspectos de optimización y rendimiento del juego. A pesar de un aumentó en la memoria total comprometida y reservada, se obtuvo una gestión eficiente de los recursos, que permitió incrementar el número de objetos y detalles en la escena, mejorando la calidad visual y la experiencia de juego, así como un mayor rendimiento. En la Figura 7.21 se puede apreciar un gráfico que corresponde a los valores obtenidos del uso de memoria con y sin Burst Compiler activado, donde se aprecia la variación visualmente de estos valores. En color

azul se muestran los valores que obtiene el videojuego sin Burst Compiler y en color ojo se observan los valores con Burst Compiler activado durante las pruebas.

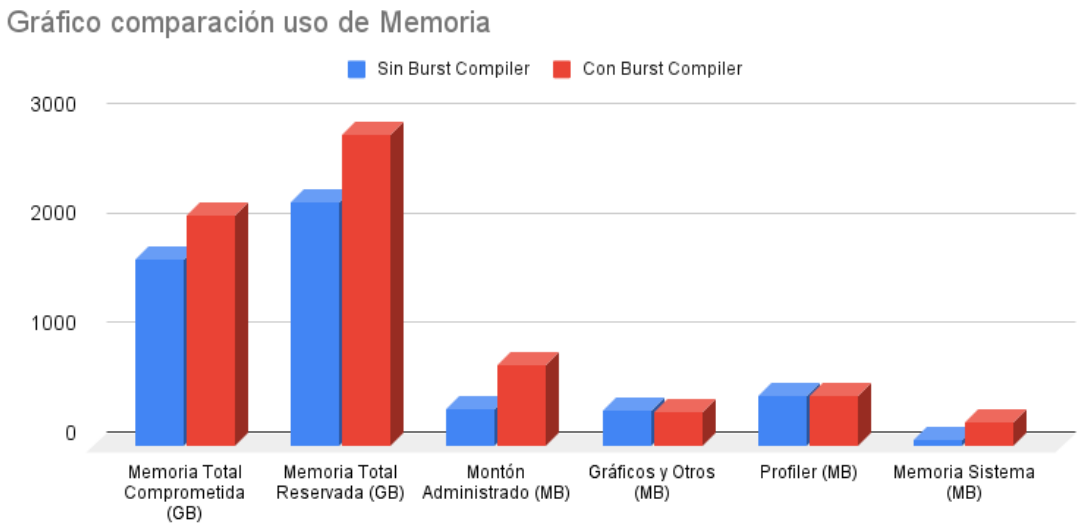

Figura 7.21: Gráfico comparación uso de Memoria

La optimización del código nativo optimizada por Burst Compiler redujo el tiempo de CPU necesario, lo cual permitió obtener unos FPS más altos y resultando en una jugabilidad más fluida.

En resumen, la activación de Burst Compiler ha demostrado ser beneficiosa, permitiendo una mayor complejidad en el juego con un rendimiento optimizado. Igualmente se obtuvo mejor rendimiento por parte del portátil en el ejecutable del juego, frente al PC de sobremesa, esto es debido a Burst Compiler y al hardware del portátil al ser más potente en trabajos paralelos gracias al mayor número de núcleos que posee así como procesador y gráfica.

Asimismo, en el gráfico de la Figura 7.22 se muestran los valores obtenidos sin Burst Compiler, con Burst Compiler y dentro del ejecutable del juego. Como se puede observar en el ejecutable se obtiene una media superior a los obtenidos dentro de Unity, esto es conseguido mediante la compilación del código de Burst Compiler y a Unity DOTS.

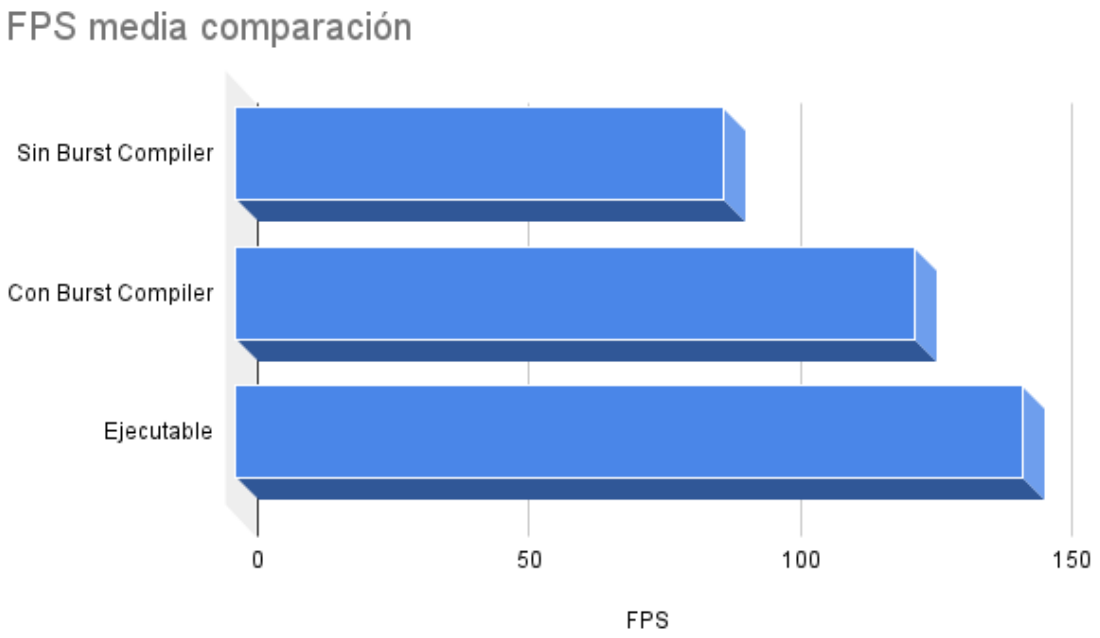

Figura 7.22: FPS media obtenidos comparación

Con esto finaliza el análisis de los resultados obtenidos en las pruebas. Donde se comprueba la eficacia de implementar Unity DOTS de forma correcta. Dando paso a las conclusiones del trabajo realizado.

CAPÍTULO 8
Conclusiones y resultados

Durante este capítulo, se describirán las conclusiones obtenidas, tras observar los resultados y finalizado el desarrollo del proyecto.

8.1 Resumen de Conclusiones

En este Trabajo de Fin de Grado, se han planteado y alcanzado los objetivos establecidos en la introducción. Como principal objetivo se ha conseguido desarrollar un videojuego estilo 'shoot 'em up' 3D empleando el framework Unity DOTS, y así comparar el rendimiento con el sistema tradicional. Mediante las pruebas realizadas y el desarrollo del videojuego, se ha demostrado que el uso de Unity DOTS mejora significativamente el rendimiento del juego, permitiendo gestionar un gran número de entidades y recursos del sistema.

Al lo largo del desarrollo, se han encontrado varios desafíos, como aprender y aplicar la tecnología de Unity DOTS, al ser una herramienta avanzada no cubierta en la carrera. Así como la implementación de sistemas y bakers en el proyecto, igualmente supuso un reto la programación concurrente con Jobs y entidades durante el desarrollo del videojuego. Sin embargo, estos retos fueron superados mediante el aprendizaje autodidacta y la aplicación de nuevas técnicas. Esto ha permitido no solo completar los objetivos, sino establecer una base sólida para futuros proyectos de videojuegos, especialmente que utilicen el framework Unity DOTS.

8.2 Relación del trabajo desarrollado con los estudios cursados

Como estudiante de Ingeniería Informática en la ETSINF (Escuela Técnica Superior de Ingeniería Informática), gracias a la rama de Tecnologías de la Información, he podido adquirir una amplia gama de conocimientos y habilidades que he empleado a lo largo del Trabajo de Fin de Grado (TFG). A continuación, se detalla cómo los estudios se han relacionado directamente con el trabajo desarrollado.

8.2.1. Aplicación de Conocimientos Adquiridos

A lo largo de la carrera, he estudiado diversas técnicas y lenguajes de programación, incluyendo C Sharp, que se ha utilizado para desarrollar el videojuego. Durante los cursos de programación, estructuras de algoritmos y desarrollo de software proporcionaron una base sólida para escribir el código necesario en el proyecto, implementar sistemas

de juego complejos y resolver problemas de manera eficiente. La estructura del código del Game Manager es un ejemplo claro de la aplicación de principios de programación orientada a objetos, además de la gestión de estados aprendidos en la carrera, así como los IJobs relacionados con la programación concurrente de hilos.

La gestión de múltiples hilos y la programación concurrente, temas abordados durante la carrera, fueron esenciales para entender y utilizar el EntityCommandBuffer (ECB) y EntityManager en Unity DOTS. Estos conocimientos permitieron optimizar el rendimiento del juego, al manejar eficientemente las entidades y sus componentes, así como programar cambios estructurales de manera segura y efectiva desde múltiples hilos.

Desarrollo de Videojuegos

Las asignaturas de Desarrollo de Videojuegos en 3D, y Animación y Diseño de Videojuegos, fueron fundamentales para realizar el diseño del videojuego junto con la implementación de las mecánicas del juego. Además, la experiencia adquirida en el uso de herramientas como Unity fue crucial para el desarrollo y la programación del juego.

Herramientas de Desarrollo

El uso de Visual Studio para la codificación, Git para el control de versiones, 'Draw.io' para realizar diagramas y esquemas visuales, Trello para organizar tareas de forma visual, y el diagrama de Gantt son ejemplos de herramientas empleadas durante el desarrollo y planificación del proyecto.

8.3 Agradecimientos

Me gustaría agradecer a mis padres y mi hermana, por haberme apoyado en todo momento, gracias de verdad, sin vosotros esto no habría podido seguir avanzando en la carrera y convertirme en quien soy a día de hoy. A mi tutor Adolfo por hacer posible la realización de este trabajo. A mis amigos por hacerme reír, y estar ahí siempre para conmigo.

Muchas Gracias a todos.

Tras las conclusiones, se describirán, posibles mejoras para un trabajo futuro o ampliación.

CAPÍTULO 9
Trabajos Futuros

En este apartado final se presentan las posibles ampliaciones, mejoras y nuevas líneas de desarrollo que podrían aplicarse al proyecto 'Surge Survivor' en el futuro. También se destacan algunas áreas que no se recomienda explorar y se explican las razones para evitar seguir esos caminos.

9.1 Mejoras pendientes

A lo largo del desarrollo de este TFG, hubo ciertos aspectos que me hubiera gustado abordar pero que no fue posible debido a las limitaciones de tiempo. Estos incluyen:

- **Implementación de un Sistema de Niveles Más Complejo**: Se podría ampliar el sistema de niveles básico del juego, y agregar opciones más complejas a la hora de subir de nivel el jugador, más que simples modificaciones de valores numéricos, como incluir habilidades nuevas, niveles con varias mejoras, así como más mejoras básicas del jugador para hacer más divertida y entretenida la jugabilidad.
- **Integración de IA para Enemigos**: Es posible mejorar la inteligencia artificial de los enemigos, para perseguir al jugador de otras maneras, como agrupar enemigos, que algunos carguen hacia el jugador en una dirección fija cuando son creados, crear diferentes formas de calcular los puntos de spawn para crear situaciones donde el jugador tiene que alejarse de una zona de enemigos y crear situaciones mas interesantes para el jugado. Asimismo, añadir más tipos de enemigos para ampliar la variedad de las oleadas.
- **Ampliación de herramientas del jugador**: Implementar diferentes armas que el jugador pudiese elegir con diferentes patrones de disparo, así como diferentes atributos y mejoras para incrementar la variedad y rejugabilidad del videojuego, los cuáles son elementos importantes en un videojuego shoot 'em up para mantener al jugador interesado en el juego, así como ampliar la complejidad y profundidad de cada partida.
- **Desarrollo de Herramientas de Desarrollo**: Crear herramientas y bibliotecas adicionales para facilitar el uso de Unity DOTS en futuros proyectos, así como mejorar la documentación sobre el framework Unity DOTS, proporcionando así ayuda a futuros desarrolladores interesados en el tema.
- **Optimización Avanzada de Memoria**: Se podrían explorar técnicas más avanzadas para la gestión de memoria y la optimización del uso de recursos, especialmente para dispositivos de gama baja.

9.2 Ampliaciones y Mejoras

Para mejorar tanto la eficiencia como las funcionalidades del trabajo realizado, se podrían considerar las siguientes ampliaciones y mejoras:

- **Mejora de Eficiencia**: Continuar explorando técnicas de optimización para reducir el uso de CPU y memoria, así como mejorar la tasa de frames por segundo (FPS) para mejorar la fluidez del juego y experiencia del jugador.
- **Funcionalidades Adicionales**: Añadir nuevas características al juego, como opciones de personalización para el jugador, y una mayor variedad de armas y enemigos, implementar habilidades nuevas al jugador, como un salto, habilidades especiales vinculadas a un input nuevo, para dar más opciones al jugador a la hora de derrotar enemigos.
- **Emplear técnicas de optimización en videojuegos**: Como los Levels of Detail (LODs), que son una técnica de optimización que consiste en crear múltiples versiones de un modelo 3D con diferentes niveles de detalle. La versión más detallada se usa cuando se encuentra cerca de la cámara del jugador, y las demás según cuanto alejadas se encuentren del jugador, de esta forma se reduce la carga de procesamiento cuando los objetos se encuentran alejados, sin sacrificar demasiada fidelidad visual. En el proyecto se podrían crear diferentes LODs de enemigos, debido a que son el principal objeto a renderizar en el juego.

9.3 Caminos a evitar

Durante el desarrollo del videojuego empleando Unity DOTS, se adquirió experiencia y se identificaron algunos caminos que es mejor evitar y las razones para ello:

- **Uso Exclusivo de GameObjects para Juegos Complejos**: Utilizando Unity a medida que la complejidad y el número de objetos en el juego aumentan, el uso exclusivo de GameObjects puede llevar a problemas de rendimiento significativos. Para evitar esto, es recomendable utilizar Unity DOTS para gestionar grandes cantidades de entidades de manera más eficiente así como aumentar el rendimiento.

  Debido a la complejidad de Unity DOTS, es recomendable analizar los puntos críticos del juego, para así de antemano planear como solucionarlos. Como utilizar sistemas para generar enemigos, mover enemigos y más interacciones. Así como crear el resto del juego con GameObjects y de esta forma juntar ambos mundos y aprovechar lo mejor de ambas arquitecturas.
- **Optimización Prematura:** A la hora de desarrollar un videojuego, optimizar todas las partes del juego al principio puede resultar en una pérdida de tiempo. Para evitar esto, es mejor identificar y enfocarse en los cuellos de botella críticos del juego. Mejor que realizarlo una vez este completada gran parte del desarrollo, así se pueden optimizar varios puntos críticos al principio.

Bibliografía

[1] Schell, J. *The Art of Game Design: A Book of Lenses, Third Edition*. CRC Press, 2019.

[2] Nystrom, R. *Game Programming Patterns*. Genever Benning, 2014.

[3] Hocking, J. *Unity in Action, Third Edition*. Manning Publications, 2018.

[4] Dickinson, C. *Unity Game Optimization*. Packt Publishing, 2019.

[5] Statista. Datos de videojuegos actuales. Recuperado de: https://es.statista.com/temas/9150/industria-mundial-del-videojuego/#topicOverview, 2024.

[6] Unity Technologies. Unity ECS guía y API. Recuperado de: https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html, 2024.

[7] Thompson, J. Unity ECS Zombie Tutorial Update Recuperado de: https://www.tmg.dev/tuts/zombieupdate/, 2024.

[8] Unity Technologies. Unity Manual y API. Recuperado de: https://docs.unity3d.com/Manual/index.html, 2024.

APÉNDICE A
Códigos del Videojuego

A.1 Ejemplos de ECS

Código de los ejemplos de ECS A.1

```
1
2  // Importamos las librerias necesarias para utilizar ECS, Unity, Burst Compiler
       etc.
3  using Unity.Entities;
4  using Unity.Transforms;
5  using Unity.Mathematics;
6  using UnityEngine;
7  using Unity.Burst;
8
9  [BurstCompile]
10
11 // SistemaSencillo es una estructura que deriva de un ISystem
12 public struct SistemaSencillo : ISystem
13 {
14
15     // Creamos una variable de un EntityManager para gestionar entidades
16     private EntityManager entityManager;
17
18     // Entidad para instanciar mediante el sistema
19     private Entity entidadParaInstanciar;
20
21     // Variable de la clase EnemigosData la cual contiene el prefab y datos de
           la Entidad
22     private EnemigosData enemigosData;
23
24     public void OnCreate(ref SystemState state)
25     {
26         // Inicializar cualquier estado necesario cuando el sistema es creado
27         // Mediante el estado actual 'state'
28     }
29
30     [BurstCompile]
31     public void OnUpdate(ref SystemState state)
32     {
33         // Obtener el EntityManager para gestionar entidades
34         EntityManager entityManager = state.EntityManager;
35
36         // Crear un nuevo EntityCommandBuffer para realizar cambios en las
               entidades
37         // Y le asignamos el valor de Allocator.Temp para maneras valores
               Nativos y de forma temporal
38         EntityCommandBuffer entityCommandBuffer = new EntityCommandBuffer(Unity
               .Collections.Allocator.Temp);
```

```
39
40          // Obtener el Prefab convertido a Entidad
41          Entity enemigoPrefab = enemigosData.enemigoPrefab;
42
43          // Creamos enemigo en el juego instanciandolo con el entityManager
44          Entity enemigoEntidad = entityManager.Instantiate(enemigoPrefab)
45
46          // Obtenemos el componente de LocalTransform de la entidad, que
                corresponde el Transform en un GameObject
47          LocalTransform enemigoTransform = entityManager.GetComponentData<
                LocalTransform>(enemigoEntidad);
48
49          // Al estar en un bucle movememos en cada momento de tiempo a la
                Entidad, para ello incrementamos su posicion x
50          enemigoTransform.Position = new float3(enemigoTransform.Position.x +
                1.0f, enemigoTransform.Position.y, enemigoTransform.Position.z);
51
52          // Despues de realizar el calculo y cambiar los valores asignamos a la
                entidad que hemos creado su nuevo LocalTransform
53          // De esta forma se reflejan los cambios en el juego y en la Entidad
54          entityCommandBuffer.SetComponent(enemigoEntidad, enemigoTransform);
55
56          // Ejecutar los comandos almacenados en el EntityCommandBuffer
57          entityCommandBuffer.Playback(entityManager);
58
59          // Liberar los recursos del EntityCommandBuffer
60          entityCommandBuffer.Dispose();
61      }
62  }
```

Listing A.1: Ejemplos de código sobre funcionalidades de ECS

```
1
2   // Importamos las librerias necesarias para utilizar ECS, Unity, Burst Compiler
        , etc.
3   using Unity.Entities;
4   using Unity.Mathematics;
5   using UnityEngine;
6   using Unity.Burst;
7
8   // Estructura de datos para asignar un Prefab
9   public struct BalaData : IComponentData
10  {
11      public Entity balaPrefab;
12      public float3 position;
13      public float velocidadBala;
14      public float tiempoDeVida;
15  }
16
17  // Clase MonoBehaviour, que sera asignada como Script a la entidad que se le
        asignara el Prefab mas tarde
18  public class BalaMono : MonoBehaviour
19  {
20      public GameObject balaPrefab;
21      public float velocidadBala;
22      public float tiempoDeVida;
23
24      public class BalaBaker : Baker<BalaMono>
25      {
26          // Matodo para 'bakear' una clase MonoBehaviour
27          public override void Bake(BalaMono authoring)
28          {
29              var balaSpawner = GetEntity(TransformUsageFlags.Dynamic);
30
```

```
31              AddComponent(balaSpawner, new BalaData
32              {
33                  balaPrefab = GetEntity(authoring.balaPrefab,
                        TransformUsageFlags.Dynamic),
34                  position = float3.zero,
35                  velocidadBala = authoring.velocidadBala,
36                  tiempoDeVida = authoring.tiempoDeVida
37              });
38          }
39      }
40  }
41
42  [BurstCompile]
43  public struct SistemaBalas : ISystem
44  {
45      private EntityManager entityManager;
46
47      public void OnCreate(ref SystemState state)
48      {
49          // Inicializar cualquier estado necesario cuando el sistema es creado
50      }
51
52      [BurstCompile]
53      public void OnUpdate(ref SystemState state)
54      {
55          // Obtener el EntityManager para gestionar entidades
56          entityManager = state.EntityManager;
57
58          // Crear un nuevo EntityCommandBuffer para realizar cambios en las
                entidades
59          EntityCommandBuffer entityCommandBuffer = new EntityCommandBuffer(Unity
                .Collections.Allocator.Temp);
60
61          // Obtener el Prefab convertido a Entidad
62          Entity balaPrefab = balaData.balaPrefab;
63
64          // Crear una entidad bala
65          Entity balaEntity = entityManager.CreateEntity(balaPrefab);
66
67          // Asignar componentes iniciales a la entidad, de esta forma se pueden
                inicializar en tiempo de ejecucion a entidades que se creen por
                codigo
68          entityManager.SetComponentData(balaEntity, new BalaData
69          {
70              position = float3.zero,
71              velocidadBala = 10f,
72              tiempoDeVida = 5f
73          });
74
75          // Obtener todas las entidades de balas
76          NativeArray<Entity> entidadesBalas = entityManager.GetAllEntities();
77
78          foreach (Entity bala in entidadesBalas)
79          {
80              if (entityManager.HasComponent<BalaData>(bala))
81              {
82                  // Obtener los datos de la bala
83                  BalaData balaData = entityManager.GetComponentData<BalaData>(
                        bala);
84
85                  // Obtener el LocalTransform para la posicion de la bala
86                  LocalTransform balaTransform = entityManager.GetComponentData<
                        LocalTransform>(bala);
87
```

```
                // Mover la bala hacia adelante, con su velocidad y el tiempo del frame actual
                balaTransform.Position += new float3(0, 0, balaData.velocidadBala * SystemAPI.Time.DeltaTime);

                // Disminuir el tiempo de vida de la bala, y asignamos los datos de la bala a la entidad de la bala
                balaData.tiempoDeVida -= Time.DeltaTime;
                entityManager.SetComponentData(bala, balaData);

                // Destruir la bala si su tiempo de vida ha terminado
                if (balaData.tiempoDeVida <= 0f)
                {
                    entityCommandBuffer.DestroyEntity(bala);
                }
            }
        }

        // Ejecutar los comandos almacenados en el EntityCommandBuffer
        entityCommandBuffer.Playback(entityManager);

        // Liberar los recursos del EntityCommandBuffer
        entityCommandBuffer.Dispose();

        // Liberar el array de entidades
        entidadesBalas.Dispose();
    }
}
```

Listing A.2: Ejemplos de código sobre funcionalidades de ECS

```

[BurstCompile]
public partial struct TrabajoSencillo : ISystem
{
    private float3 posicionNueva;

    public void OnCreate(ref SystemState state)
    {
        // Definir la posicion fija del objetivo
        posicionNueva = new float3(0, 0, 0);
    }

    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        // Creamos un trabajo llamado MoverEnemigosJob que utiliza el tiempo actual y la Posicion nueva
        var job = new MoverEnemigosJob
        {
            DeltaTime = SystemAPI.Time.DeltaTime,
            PosicionNueva = posicionNueva
        };

        Dependency = job.ScheduleParallel(Dependency);
    }

    // Trabajo (Job) que deriva de IJobEntity
    [BurstCompile]
    public partial struct MoverEnemigosJob : IJobEntity
    {
        public float DeltaTime;
        public float3 PosicionNueva;

```

```
33          // Metodo que ejecuta el codigo del Trabajo
34          private void Execute(ref LocalTransform enemigoTransform, ref
                EnemigosData enemigoData)
35          {
36              // Mover enemigos hacia una posicion fija
37              float3 direccionAlObjetivo = math.normalize(PosicionNueva -
                    enemigoTransform.Position);
38
39              enemigoTransform.Position += enemigoData.velocidadEnemigo *
                    DeltaTime * direccionAlObjetivo;
40          }
41      }
42  }
```

Listing A.3: Ejemplos de código sobre funcionalidades de ECS

A.2 Baking

Código del ejemplo de baking A.4

```
1  DisparoData.cs
2
3  // Clase DisparoData que contiene la Entidad, que luego sera convertida a
       GameObject
4
5  public struct DisparoData : IComponentData
6  {
7      public Entity balaPrefab;
8  }
9
10 *DisparoMono.cs*
11 // En DisparoMono, esta deriva de MonoBehaviour por lo que puede usar
       GameObjects
12
13 public class DisparoMono : MonoBehaviour
14 {
15     public GameObject balaPrefab;
16 }
17
18 // Con el Baker de esta manera convertimos un GameObject a Entidad, pudiendo
       crear asi Entidades en ejecucion
19 public class DisparoBaker : Baker<DisparoMono>
20 {
21     public override void Bake(DisparoMono authoring)
22     {
23         var playerEntidad = GetEntity(TransformUsageFlags.Dynamic);
24
25         AddComponent(playerEntidad, new DisparoData
26         {
27             balaPrefab = GetEntity(authoring.balaPrefab, TransformUsageFlags.
                    Dynamic),
28         }
29     }
30 }
```

Listing A.4: codigo fuente de CapasColisiones.cs y fragmento de BalasYNiveles.cs

A.3 Colisiones

Código de los ejemplos de colisiones A.5

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public enum CapaColisiones
{
    Default = 1 << 0,
    Wall = 1 << 6,
    Enemigo = 1 << 8,
    Player = 1 << 9,
    DropVida = 1 << 10,
}


public class DevolverCapa
{

    // Obtener las capas que colisiono
    public static uint ObtenerCapaTrasColision(CapaColisiones capa1,
        CapaColisiones capa2)
    {
        var capaRes = (uint)capa1 | (uint)capa2;

        return capaRes;
    }
}

*Fragmento de BalasYNivelesStstem.cs*

// Singleton para el mundo de fisica
PhysicsWorldSingleton physicsWorldSingleton = SystemAPI.GetSingleton<
    PhysicsWorldSingleton>();


(...)

// Detectar colisiones
NativeList<ColliderCastHit> colliderCastHits = new NativeList<ColliderCastHit>(
    Allocator.Temp);

float3 punto1 = new float3(balaTransform.Position - balaTransform.Right() *
    0.15f);
float3 punto2 = new float3(balaTransform.Position + balaTransform.Right() *
    0.15f);
float radio = balaData.tamanoBala / 2;
float3 direccion = float3.zero;
float distanciaMaxima = 1f;

uint capasColision = DevolverCapa.ObtenerCapaTrasColision(CapaColisiones.Wall,
    CapaColisiones.Enemigo);
physicsWorldSingleton.CapsuleCastAll(punto1, punto2, radio, direccion,
    distanciaMaxima, ref colliderCastHits, new CollisionFilter
{
    BelongsTo = (uint)CapaColisiones.Default,
    CollidesWith = capasColision,
});

(...)
```

Listing A.5: codigo fuente de CapasColisiones.cs y fragmento de BalasYNiveles.cs

A.4 Colisiones

Código de los ejemplos para obtener el EntityManager A.6

```
[BurstCompile]
public partial struct EnemigoSystem : ISystem {

    private EntityManager entityManager;

    (...)

    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        // Referencias
        entityManager = state.EntityManager;
        enemigoSpawner = SystemAPI.GetSingletonEntity<EnemigosData>();
        enemigosData = entityManager.GetComponentData<EnemigosData>(
            enemigoSpawner);

        playerEntity = SystemAPI.GetSingletonEntity<DisparoData>();

        SpawnearOleadaEnemigos(ref state);

    }

    (...)
}

public class PlayerInterfaz : MonoBehaviour {
    private EntityManager entityManager;

    private IEnumerator InitializeAfterDelay()
    {
        yield return new WaitForSeconds(0.2f);

        entityManager = World.DefaultGameObjectInjectionWorld.EntityManager;

    (...)

    }

    (...)
}
```

Listing A.6: Fragmento de codigo donde se incializa el Entity manager con el state del SystemState

APÉNDICE B
Objetivos de Desarrollo Sostenible

Mi Trabajo de Fin de Grado 'Surge Survivor' se relaciona con los Objetivos de Desarrollo Sostenible (ODS) en algunas áreas. En la Figura B.1 se pueden observar los diferentes ODS, su relación con el proyecto, los objetivos relacionados se muestran marcados con una equis (X).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**			X	
ODS 4. **Educación de calidad.**		X		
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**		X		
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**				X
ODS 12. **Producción y consumo responsables.**			X	
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Figura B.1: Objetivos de Desarrollo Sostenible y su relación con el trabajo

A continuación, se detalla como mi proyecto contribuye en los siguientes objetivos:

- **Salud y bienestar (ODS 3)**: Aunque el objetivo principal de mi TFG (Trabajo de Fin de Grado) no sea la salud, los videojuegos pueden tener un impacto positivo en el bienestar mental. Pudiendo ofrecer entretenimiento y una forma de escape saludable, contribuyendo al bienestar mental de los jugadores. Además, los videojuegos, cuando se juegan de forma moderada y controlada, pueden reducir el estrés, mejorar el estado de ánimo y proporcionar un sentimiento de logro. En los videojuegos multijugador se fomenta la socialización en línea y el trabajo en equipo, lo que ayuda a la salud mental en su conjunto.

- **Educación de calidad (ODS 4):** Mi proyecto puede servir como recurso adicional para estudiantes y desarrolladores interesados en la tecnología de Unity DOTS (Data-Oriented Technology Stack) y la creación de videojuegos. Al documentar el proceso de desarrollo y compartir los resultados, contribuye al acceso de información técnica de calidad y fomenta el aprendizaje en el campo del desarrollo de videojuegos. Además, de proporcionar un ejemplo práctico de cómo aplicar teorías y conceptos de informática en un proyecto real, ayudando en el desarrollo de futuros proyectos.

- **Trabajo decente y crecimiento económico (ODS 8):** El desarrollo de videojuegos es una industria que actualmente está en crecimiento, con posibilidades de generar empleo y oportunidades económicas. Mi proyecto al mostrar las ventajas de rendimiento de Unity DOTS puede ayudar a desarrolladores a mejorar la eficiencia en videojuegos y productividad en la industria. La optimización del rendimiento permite a las empresas y desarrolladores reducir el coste de desarrollo y mantenimiento, haciendo el desarrollo de proyectos más viables económicamente.

- **Industria, innovación e infraestructuras (ODS 9):** El objetivo principal de mi TFG se centra en explorar y demostrar como la tecnología Unity DOTS puede mejorar significativamente el rendimiento en videojuegos. Contribuyendo a la adopción de nuevas tecnologías en la industria del videojuego, así como la mejora de infraestructura tecnológica apoyando la industria del entretenimiento para avanzar en un dirección más eficiente.

- **Producción y consumo responsables (ODS 12):** Aunque de manera indirecta, la optimización del rendimiento en los videojuegos puede llevar a un uso más eficiente de los recursos computacionales. Esto puede traducirse en una menor necesidad de hardware costoso y de alta potencia, promoviendo así un consumo más responsable de recursos computacionales, disminuyendo la huella de carbono asociada con la producción del hardware, y contribuyendo a la sostenibilidad ambiental.

En conclusión, aunque mi TFG 'Surge Survivor' no abarca la mayoría de los ODS, contribuye significativamente en área relacionadas con la educación, el bienestar mental, el crecimiento económico e innovación tecnológica en el sector de los videojuegos e informática, así como el consumo responsable. Estos aportes son necesarios para fomentar un desarrollo sostenible en la industria del videojuego y la tecnología en general. Al integrar estos objetivos en mi proyecto, contribuye a un futuro más sostenible y profesional.