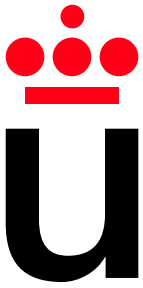


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Diseño y Desarrollo de Videojuegos

Curso 2021-2022

Trabajo Fin de Grado

DISEÑO Y DESARROLLO DE UN MOTOR DE VIDEOJUEGOS BASADO EN LA ARQUITECTURA ECS

Autor: Antonio Arian Silaghi
Tutor: Julio Guillén García


BY NC SA

Agradecimientos

A mis amigos, familia, Micky, Cookie y Pepe.

Resumen

En este trabajo se plantea el diseño e implementación de un motor de videojuegos 2D desarrollado en C++ usando OpenGL como API gráfica y que contenga un modelo de objetos/entidades basado en diseños ECS.

Se ha realizado un estudio de distintos motores e implementaciones y se ha diseñado e implementado el motor en cuestión siguiendo las conclusiones que a las que se han llegado después de los análisis. Se han usado también principios de diseño orientado a datos de distintas maneras para la implementación.

Por último, se han creado distintos proyectos usando el motor con al finalidad de validad sus funcionalidades y comprobar que tiene las suficientes funcionalidades para crear juegos pequeños o mini-juegos.

Palabras clave:

- Videojuegos
- Diseño Orientado a Datos
- Motores de Videojuegos
- OpenGL
- Entity Component Systems (ECS)

Abstract

This project is based upon the design and implementation of a **2D game engine deveoloped using C++ and OpenGL**. One of its key characteristics is that it implements a gameobject/entities model based on the **ECS architecture**.

Multiple studies of existing game engines and systems implementations have been made and the insights from these studies were used to develop and implement the engine. There has also been a study of principles of **Data-Oriented Design** and these principles were taken into acount in the development process of the engine.

Multiple sample projects have been created to validate and showcase the functionalities of the engine.

Key Words:

- Videogames
- Data-Oriented Design
- Game Engine
- OpenGL
- Entity Component Systems (ECS)

Índice de contenidos

Índice de figuras

Glosario

API Refiriendose a Application-Programming-Interface, es un tipo de interfaz que permite la comunicación entre distintas piezas de software. 1, 59, V

asset Cualquier elemento como texturas, modelos, sonidos, etc.... 6

batch rendering Método de renderizado que consiste combinar múltiples drawcalls con el fin de minimizar el número de estas mismas y aumentar el rendimiento. 31

collider Elemento que delimita el area o volumen de un objeto dentro del mundo digital generalmente usado para calcular físicas.. 1

drawcall Llamada a la API gráfica para dibujar un elemento. 31

ECS Arquitectura de software basada en el diseño orientado a datos. 1

handle Identificador o referencia a un elemento. 37

heap Dentro del contexto de C++, espacio de memoria en el que las reservas y liberaciones de memoria estan completamente controladas por el desarrollador. 32

ID Elemento identificador de un objeto. 1

metadata Hace referencia a datos que contienen información sobre otros datos. 58

Non Photorealistic Rendering Técnicas de renderizado que se centran en habilitar la creación de distintos estilos de arte.. 1

pipeline Secuencia de elementos que procesamiento organizados de tal manera que la salida de uno es la entrada del siguiente. 36

pixel art Diseño artístico que se basa en la creación de imágenes a traves de píxeles colocados intencionalmente. 8

polling En computación es un tipo de operación en la que se comprueba constantemente un elemento. 59

profiling Técnicas que analizan la complejidad de tiempo y espacio de un programa en tiempo de ejecución.. 1

quad Polígono de 4 vértices conectados por 4 aristas. 31

ray tracing Método de renderizado que modela el camino de la luz con la finalidad de crear imágenes digitales. 6

singleton Patrón de diseño que límita la instanciación de una clase a una sola instancia. 45

sprite Término que hace referencia a una imagen, principalmente usado en videojuegos 2D.. 8

spritesheet Conjunto de sprites que se encuentran guardados en una misma imagen. 1

stack Dentro del contexto de C++, espacio de memoria contigua en el que la reserva y liberación esta controlada "por el lenguaje". 32

thread Elemento usado para ejecutar instrucciones de código de manera simultanea. 86

Unity Motor de videojuegos 2D y 3D popular en el espacio indie y de móviles creado por Unity Technologies. 2

wrapper Abstracción de una API con la finalidad de no depender directamente de esta misma. 31

Capítulo 1

Introducción

1.1. Análisis Histórico

En la actualidad la gran mayoría de juegos se desarrollan usando un motor de videojuegos ya sea comercial o propietario[1].

Esto no fue siempre de esta manera. En los inicios se tenían que diseñar y desarrollar los juegos prácticamente desde cero. Creando un diseño específico y especializado para la plataforma en la que se fuera a ejecutar el juego. Esto se debe a las ganancias de rendimiento que se obtenían y a que no había la estandarización de hardware que hay hoy en día.

Esta situación hacia que no tuviera sentido crear lo que se consideraría un motor porque había pocas funcionalidades que fueran reusables.

Aunque este termino surgió en los años 90, existen sistemas que se considerarían motores que aparecieron en los años 80 como por ejemplo SCUMM (Script Creation Utility for Maniac Mansion) desarrollado por LucasArts. Este motor/lenguaje de scripting permitía a los diseñadores crear contenido para el juego sin tener que escribir código en el lenguaje final en el que acabara el juego.[2]

SCUMM fue desarrollado con la finalidad de facilitar el desarrollo de Maniac Mansion pero se reutilizo para otros juegos.

Junto a este, también se desarrollaron lo que se llamarían *construction*

Figura 1.1: Gameplay de Ultimate Doom

kits. Estos kits ofrecían funcionalidades para crear de una manera más sencilla distintos tipos de juegos. Ejemplos de estos kits serian: Pinball Construction, Thunder Force Construction, Adventure Construction, Wargame Construction, etc...

A finales de los años 90, juegos como **Quake III Arena y Unreal** estaban diseñados teniendo en cuenta la reusabilidad de distintos módulos. Estos motores ya eran bastante personalizables mediante lenguajes de scripting.

Doom tenia una arquitectura en la que había una separación bastante clara entre lo que hoy llamaríamos el motor, que contenía todos los sistemas como el sistema de renderizado, detección de colisiones, audio... y el propio código que especificaba las mecánicas.[3]

La capacidad de computo de los dispositivos fue aumentado drásticamente cada año, lo que a la larga permitió la creación de motores que intentan acomodar de manera general muchos tipos de juegos o aplicaciones interactivas como por ejemplo Unity.

Aun así. Todos los motores creados están diseñados para un uso más o

Figura 1.2: Primera versión de UnrealEd

menos concreto y dentro de un contexto específico. Generalmente no es buena coger un motor diseñado crear juegos shooter en primera persona e intentar hacer un juego de carreras. Es algo que se puede hacer y conseguir en cierta manera. pero el resultado final sera uno mucho peor y con peor rendimiento que el que se hubiera conseguido si se hubieran hecho sistemas que tengan en cuenta el contexto en el que se desarrolla un juego de carreras generalmente.

En los últimos 15 años se ha estado haciendo cada vez más popular un tipo de arquitectura llamada **ECS**. De manera notable este tipo de arquitectura apareció en juegos como **Operation Flashpoint: Dragon Rising** en 2007. Esta arquitectura permitía un uso más efectivo del hardware lo que aumentaba en ciertos casos considerablemente el rendimiento.

Actualmente este tipo de diseño esta siendo muy popularizado por el motor de videojuegos Unity. Juegos como de relativamente gran escala como **Frostpunk**[4] y especialmente **Overwatch**[5] usan esta arquitectura.

1.2. Mercado actual

Actualmente existen múltiples motores comerciales u open-source que están disponibles al público. En este apartado se va a hablar de los más conocidos/usados como **Unity, Unreal, Godot, Game Maker y RPG**

Maker

1.2.1. Unity

Es uno de los motores más populares especialmente dentro del desarrollo de aplicaciones para android e IOS[6]. Es multiplataforma y puede ser usado para hacer juegos tanto 2D como 3D[7].

Es un motor gratuito inicialmente para personas individuales o empresas pequeñas. Solo se tiene que pagar por la licencia si se superan ciertos umbrales de ganancias con los productos creados con el motor

Este motor es especialmente popular en el espacio indie debido a la gran cantidad de recursos que existen para aprender cuestiones del motor junto con la facilidades que ofrece para crear juegos pequeños.

1.2.2. Unreal Engine 5

Unreal engine es un motor también muy popular el cual destaca por ofrecer muchas funcionalidades para poder crear gráficos de gran calidad como por ejemplo el sistema de **Lumen**[8] para crear iluminación realista o **Nanite**[9] para poder renderizar escenarios de muy alta calidad con un buen rendimiento.

Es gratis si se usa para realizar proyectos con fines educativos o proyectos internos. Junto a esto, tiene un modelo parecido al de Unity en el cual solo se tienen que pagar royalties si se supera un umbral de ganancias con el proyecto.

Ofrece un sistema de **Blueprints** en el que el usuario puede programar lógica del juego usando bloques visuales. Junto a esto también permite programar funcionalidades directamente en C++.

Unreal ofrece el código fuente completo del motor con fines educativos.

1.2.3. Godot

Godot es un motor gratuito open source con capacidades para crear juegos 2D y 3D para PC, movil o web.[10]

Tiene su propio lenguaje de scripting junto con un lenguaje visual y da soporte a otros como C# y C++. Junto a esto ofrece las herramientas para poder implementar otros lenguajes como lenguaje de scripting. Gracias a esto la comunidad ha implementado otros lenguajes como Rust y Javascript.[11][12]

Usa OpenGL ES 3.0 como base para el motor de renderizado 3D aunque se esta desarrollando soporte para Vulkan. El motor de renderizado 2D es un motor independiente del 3D que ofrece funcionalidades especificas para 2D como luces 2D, tilesets, parallax, etc..[11]

1.2.4. GameMaker Studio 2

Gamemaker Studio 2 es un motor diseñado para la creación de juegos 2D. Este motor tiene una versión gratuita que permite la creación y exportación de juegos a la plataforma GX.games. Si se quieren crear versiones para desktop o para otras plataformas se tiene que pagar una licencia mensual.[13]

Usa un lenguaje de scripting propio para crear la lógica del juego. Junto a esto tiene un lenguaje visual para la creación de lógica.

Este motor tiene la ventaja de ser muy sencillo de usar sobretodo si se tienen conocimientos limitados sobre la programación o la creación de videojuegos.

1.2.5. RPG Maker

RPG Maker es una serie de motores diseñados para crear juegos 2D Top-Down que tengan un gameplay parecido a los de juegos RPG japoneses. Puede ser usado también para crear juegos narrativos y juegos point and click.[14]

Es un motor diseñado especializado en la creación de este tipo específico

de juegos y ofrece muchas ayudas que permite crear videojuegos sin conocimientos de programación. Junto a esto ofrece múltiples sets de assets gratis y de pago. También ofrece distintas herramientas para crear personajes y efectos especiales.

Tiene una versión de prueba gratis de 30 días y una versión de pago único que contiene todas las funcionalidades.

1.3. Estado del arte

El estado del arte de motores de videojuegos es un tema muy amplio debido debido a que se puede considerar que un motor es el conjunto de una alta cantidad de sistemas distintos que pueden llegara ser muy complejos individualmente como por ejemplo el sistema de renderizado.

1.3.1. Renderizado

En este campo de renderizado la tecnología más popular en la actualidad disponible al público es el sistema llamado **Nanite** [9] incluido en **Unreal Engine 5**, este sistema virtualiza la geometría de los modelos 3D, en términos generales genera y ajusta dinámicamente la geometría de los modelos dependiendo de la vista de la cámara, esto permite usar modelos con una alta cantidad de polígonos, como por ejemplo modelos escaneados, manteniendo un buen rendimiento.

En los últimos años se están usando cada vez más tecnologías que permiten tener iluminación global dinámica como **Ray tracing**, existen implementaciones que usan hardware para poder conseguir esto en tiempo real (RT Cores de tarjetas gráficas Nvidia RTX) [15] pero también existen implementaciones que no necesitan esta aceleración por hardware como el sistema **Lumen** [8] que viene integrado en **Unreal Engine 5**

Existen también métodos de renderizado alternativos como las SDFs neuronales, estos métodos anteriormente eran demasiado costosos computacionalmente pero hay avances que permiten el uso de estas técnicas de renderizado en tiempo real.[16]

1.3.2. Modelos de Objetos/Entidades

En cuanto a los modelos de objetos del juego o entidades. No existen muchos avances considerables. Los modelos más "modernos"son los llamados ECS que aunque tengan múltiples implementaciones distintas. Todos siguen de manera general la misma idea.

El modelo más interesante que se pudo encontrar es un modelo hibrido implementado en **Kruger Engine** en el que se intentan fusionar las mejores cualidades que ofrecen los modelos basados en actores o gameobjects y los modelos ECS[17].

1.3.3. Herramientas

Aunque principalmente en este trabajo se trate la parte del motor que se encarga de ejecutar el propio juego o *runtime*. Las herramientas son una parte muy importante que se podrían considerar parte del motor. Juegos Witcher 3 tienen cientos de assets y un mundo de gran tamaño el cual no se podria haber construido de manera razonable sin la creación de herramientas que ayudaran con esta tarea.

Una de estas herramientas usaba tecnologías de bases de datos SQL para guardar toda la información de lo assets del mundo junto con metadatos de estos assets. Esto les permitía hacer queries sobre todos los objetos del mundo de manera sencilla para encontrar distintos tipos de objetos. Esta herramienta ayudo enormemente en la creación del mundo y ayudo a encontrar fallos con estos assets de una manera más rápida.[18]

1.3.4. Animaciones

La manera tradicional de animar modelos 3D es crear un rig para este modelo hecho de huesos y animar dichos huesos haciendo que los vértices del modelo los sigan de distintas maneras predefinidas. Para facilitar el proceso se suelen usar distintas técnicas como constrains o IKs. También existen animaciones que se realizan mediante técnicas de *motion capture* que se basan en capturan el movimiento real de actores.

Cada vez más existen distintas técnicas que se basan en el uso de distintos tipos de redes neuronales para animar humanoides o cuadrúpedos[19]. Junto a esto también existen técnicas como el uso de IK Rigs lo que permite el reuso de rigs y animaciones para modelos que necesiten tener distintos parámetros [20].

1.3.5. Pixel art

Aunque este tipo de estilo artístico haya existido durante muchos años y no parece que sea algo en lo que se pueda avanzar tecnológicamente. Si que existen técnicas modernas interesantes para la creación de videojuegos con este estilo como por ejemplo el sistema de sprites implementado en un juego en desarrollo llamado **Astortion**[21].

Este sistema se basa en definir los sprites de un objeto o personaje como si fuera un modelo 3D. Se crea una textura de UV's la cual se usa en conjunto con la textura de color del personaje para producir el sprite final. De esta manera, si se quieren crear animaciones o poses para el personaje pixel art. Lo que se define es la textura UV para indicar como sera el sprite en esa pose y se usan las coordenadas UV de esta textura para conseguir el color final de cada pixel de la textura de color.

El tener una separación entre lo que seria la textura del personaje y su pose de la misma manera que se tendría en un entorno 3D permite que, por ejemplo, se pueda modificar el personaje para cambiarle el color de la ropa o para añadirle accesorios sencillos sin la necesidad de tener sprites con las animaciones de todos estos casos.

Capítulo 2

Objetivos

2.1. Descripción del problema

Se plantea la creación de un motor de videojuegos 2D usando C++ como lenguaje y OpenGL como API gráfica. El motor se creara y diseñara siguiendo un diseño orientado a datos. Al final se estudiara la facilidad y efectividad de este tipo de diseños.

Se usaran librerías como GLAD para cargar las funciones de OpenGL, IMGUI para mostrar ventanas con información de debug dentro del motor, Optick para hacer profiling, stb image para cargar las imagenes de los sprites y GLFW para crear la ventana de Windows y el contexto de OpenGL

El cliente/juego podrá crear entidades con sprites y añadirles lógica usando el sistema ECS del motor. todo esto mediante código.

2.2. Motivación

Los motores más populares como Unity, están diseñados para que sean lo más generales posibles, esto es una ventaja para ciertos tipos de proyectos que requieren esta flexibilidad, pero este diseño tiene sus desventajas ya que al intentar ser tan general, se pierde el contexto en el que se esta trabajando,

estos motores crean sistemas abstractos que tienen que soportar y tener en cuenta múltiples tipos de usos distintos, la gran mayoría de las funcionalidades que ofrecen no se acaban usando porque no son relevantes para el tipo de juego que se quiere crear y en el peor de los casos acaban entorpeciendo.

Al usar un motor como Unity, es posible que el desarrollador tenga que construir encima de los sistemas que ya existen para poder resolver los problemas que tiene el tipo de juego que se quiere crear, esto puede crear complejidad innecesaria lo cual puede hacer que sea más fácil introducir bugs en el sistema, sea más complejo de mantener y de entender.

La creación de un motor propio para un tipo de juego especifico permite al desarrollador tener control absoluto sobre todo el funcionamiento interno de su juego y hace que este pueda diseñar y crear sistemas o funcionalidades que solucionen muy bien los problemas que se le plantean dentro del contexto del juego.

Junto a esto. Una gran motivación es el estudiar e intentar aplicar en el diseño conceptos de diseño orientado a datos.

2.3. Objetivos

- Analizar las características generales de los motores de videojuegos más relevantes.
- Definir la especificación técnica de un motor de videojuegos basado en ECS.
- Implementar un prototipo del motor de videojuegos basado en ECS.
- Desarrollar una demo en el prototipo.
- Validar y evaluar las funcionalidades del proyecto.

2.4. Metodología empleada

En el proyecto se ha usado una metodología de desarrollo iterativo. Inicialmente se definieron las funcionalidades y módulos a un alto nivel que se preveían que tendría el motor final y se realizaron múltiples **iteraciones** simples para implementar los distintos elementos.

Inicialmente, se realizo una lista de funcionalidades extensa la cual se redujo a lo que se considero que seria lo mínimo para que el motor fuera funcional. Dejando el resto de funcionalidades para iteraciones futuras si se tuviera los recursos

Cada iteración consta de los siguientes pasos:

1. Definición de las funcionalidades especificas que se quieren conseguir.
2. Investigación de diseños e implementaciones ya existentes.
3. Diseño e implementación de las funcionalidades.
4. Análisis del resultado y retroalimentación.

Generalmente en esta metodología se definen tiempos fijos de iteración pero debido a las circunstancias del desarrollo y a la incertidumbre del tiempo esto no se llevo a cabo. Las iteraciones han llevado desde 2-3 días hasta 2 semanas.

Una gran ventaja de este proceso es que estas iteraciones permitieron que se fuera revisando constantemente los sistemas ya existentes con la finalidad de mejorarlos con lo aprendido.

2.5. Planificación

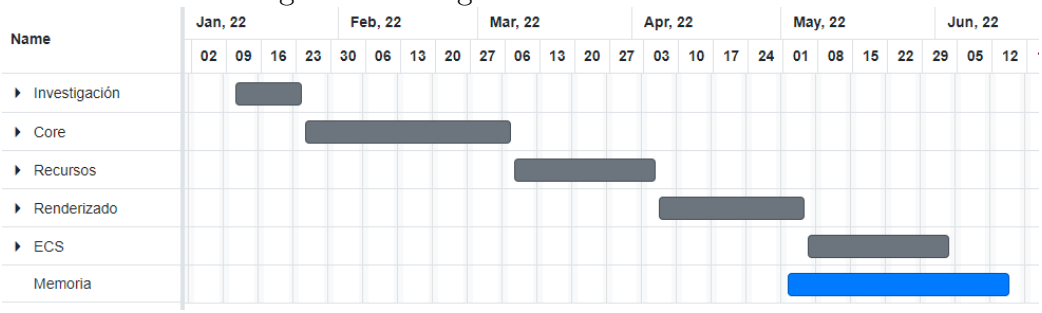
El proyecto se planteo divido en 6 partes distintas, **Investigación, Implementación del Core del motor, Sistema de Recusos, Rendering, Sistema ECS y Desarrollo de la Memoria**, en cada una de estas partes

se estimo lo que se tardaría en implementar cada funcionalidad y el orden en el que se realizarían.

En la planificación no se introdujo el tiempo de testeo y refinamiento, ya que esto se realizaría constantemente a lo largo de todo el proyecto.

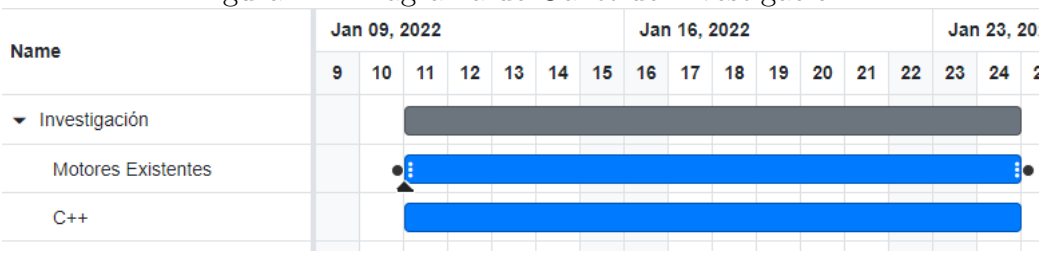
Cabe destacar que esto fue una primera planificación inicial y a la hora de implementar se tuvo tuvo que cambiar por desgracia casi por completo debido a distintos sucesos que hicieron que se tuviera casi un 70 % menos de tiempo para el desarrollo del proyecto.

Figura 2.1: Diagrama de Gantt General



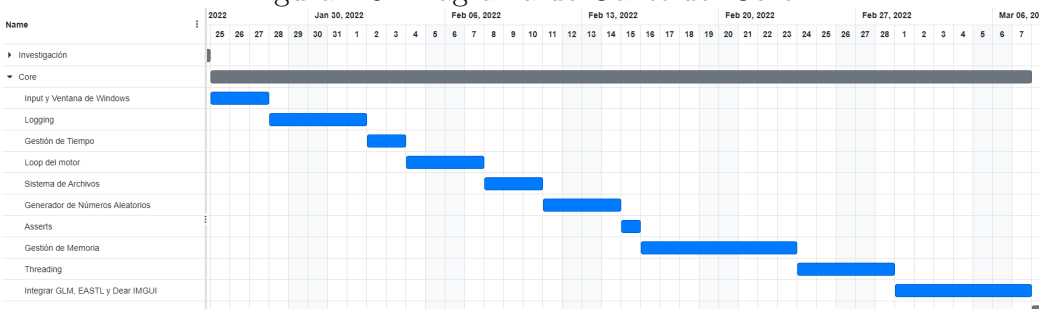
En la figura 2.1 se puede ver una vista general de todas las partes en las que se dividió el proyecto, una observación interesante es que inicialmente se estimó que el diseño y desarrollo de la parte **Core** del motor es una de las partes que más tiempo llevaría junto con la creación de la memoria del proyecto. Esto fue así con la intención de crear una base muy robusta sobre la que desarrollar el resto de los sistemas del motor.

Figura 2.2: Diagrama de Gantt de Investigación



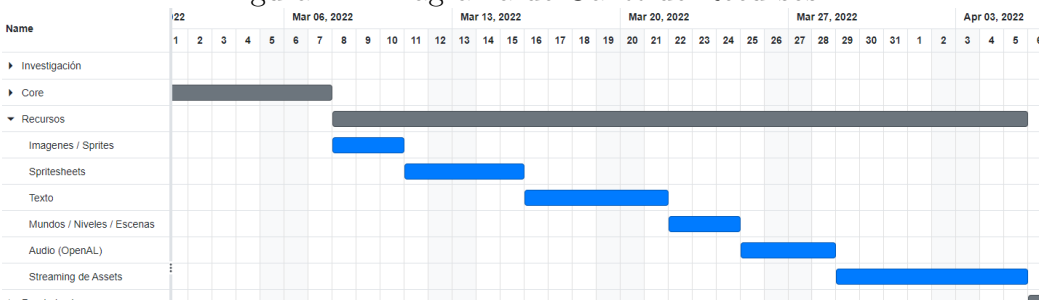
Como se puede ver en la figura 2.2, en la parte inicial de investigación, se analizarían las implementaciónes de motores ya existentes como **Blah, Hazel, Kruger, Halley y Bevy**, se añadió también la tarea de investigar y estudiar sobre el propio lenguaje de programación **C++** y sobre arquitecturas de software. ya que se asumió que el autor inicialmente no tenia la suficiente soltura con el lenguaje para realizar el proyecto.

Figura 2.3: Diagrama de Gantt del Core



Como se puede observar en la figura 2.3, en este periodo, que seria el primero de implementación, se desarrollarían todas las funcionalidades de crear la ventana del motor, gestionar input y crear todos los sistemas base que todo el resto del motor usarían.

Figura 2.4: Diagrama de Gantt de Recursos



En cuanto a la gestión de recursos del motor, inicialmente se planeo que soportaría imágenes, texto, spritesheets, escenas, sonido y streaming sencillo de assets, debido a cuestiones de tiempo muchas de estas funcionalidades no se acabaron implementando.

Figura 2.5: Diagrama de Gantt de Rendering

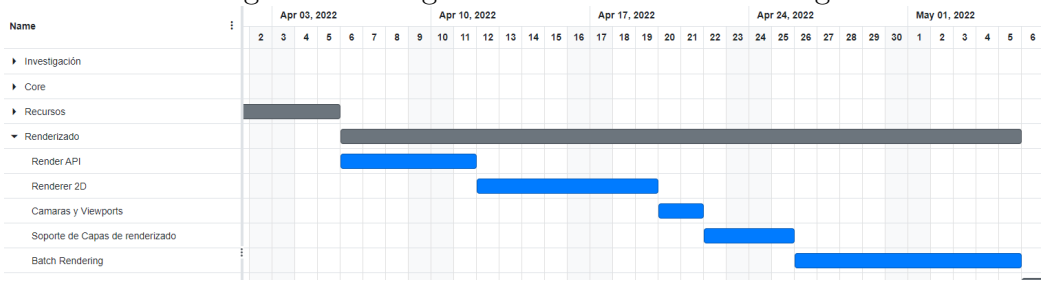
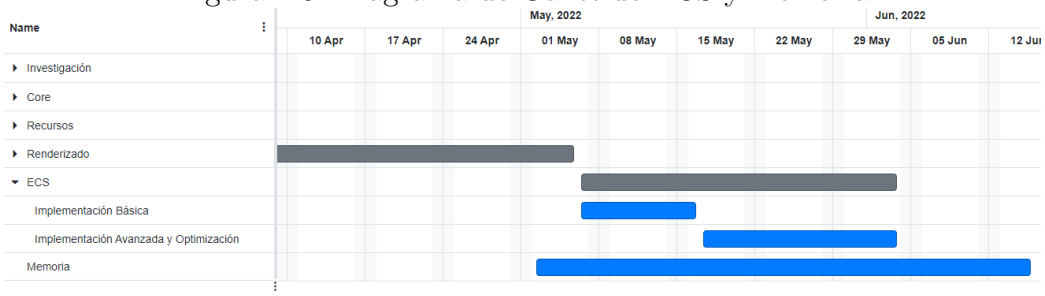


Figura 2.6: Diagrama de Gantt de ECS y memória



2.6. Estructura de la memoria

La memoria está estructurada en los siguientes 6 capítulos:

1. **Introducción:** Este capitulo contiene un análisis histórico de los motores de videojuegos, el mercado actual de estos mismos y el estado del arte de distintos aspectos de estos motores.

2. **Objetivos:** Describe el problema, la motivación y los objetivos de este proyecto. Junto a esto, muestra la metodología empleada y la planificación inicial que se realizo.

3. **Marco Teórico:** Expone todos los casos de estudio de distintos motores e implementaciones. Muestra un análisis del diseño orientado a datos y por último explica cuales fueron las alternativas que se consideraron.

4. **Diseño e Implementación:** Expone todos los requisitos de la implementación del motor, junto con todas las librerías usadas, el proceso de las primeras implementaciones y una descripción de la implementación final a la que se llego.

5. **Validación:** Analiza el resultado final del motor y muestra casos de uso de este mismo.

6. **Conclusiones:** Realiza un análisis de los logros que se han conseguido en este proyecto, las lecciones aprendidas y expone distintas lineas futuras para el proyecto.

Capítulo 3

Marco teórico

3.1. Diseño Orientado a Datos

3.1.1. Introducción

Este tipo de diseño ha existido durante muchos años pero empezó a ser llamado diseño orientado a datos o data oriented design en artículos de 2009[22]. Desgraciadamente en la actualidad al menos de manera popular, el diseño orientado a datos se menciona casi exclusivamente en relación con organizar los datos que el ordenador procesa de tal manera que utilizar mejor la cache de procesador. A su vez esto provoca que el diseño orientado a datos generalmente se vea como algo que solo tiene sentido aplicar en situaciones en las que se necesiten procesar múltiples elementos en una CPU con un alto rendimiento.

Aunque el diseño orientado a datos encamine al desarrollador a realizar este tipo de software que puede aprovechar la cache de la CPU y tener un mayor rendimiento por ello. Esta manera de programar abarca una gran cantidad de principios sobre como diseñar software que sea **mantenible, fácil de cambiar y optimizable**.

Este término fue aumentando en popularidad a lo largo de los años en la industria de los videojuegos debido a las necesidades que tenia esta industria

de producir software con ciertos requisitos de rendimiento.

3.1.2. ECS

Las arquitecturas ECS que se han estado tratando en este proyecto están basadas fundamentalmente en principios del diseño orientado a datos. Pero esta arquitectura es solo una manera de implementar algunos de los principios del diseño orientado a datos.

Estas arquitecturas contienen de manera general tres unidades lógicas

- Componentes: Mayormente contenedores puros de datos.
- Sistemas: Funciones de operan y mutan el estado de los componentes o del programa
- Entidades: Contenedores de componentes*

Una distinción importante es que aunque una entidad se pueda considerar un contenedor de componentes, a nivel de implementación una entidad en muchos casos suele ser solo un ID.

Los componentes suelen estar guardados en arrays en memoria contigua y para acceder a un componente de una entidad se usa su ID.

El que los componentes estén en memoria contigua ayuda a minimizar los fallos de cache y a aumentar el rendimiento debido a esto **pero esto no tiene porque ser siempre así.**

Es importante tener en cuenta que simplemente guardar los datos en componentes en arrays contiguos no significa necesariamente que se vallan a minimizar los fallos de cache. Especialmente si se accede a múltiples arrays de componentes distintos.

3.1.3. Orientar todo a objetos

Para poder definir mejor los principios del diseño orientado a datos. Se empezara hablando de alguno de los principios del diseño orientado a objetos

y la programación orientada a objetos debido a que este es el paradigma más "popular"hoy en día.

Este paradigma se define de manera general por, como indica su nombre, definir la estructura de un programa separándolo en distintos objetos que se comunican entre ellos mediante mensajes. El estado del programa se vería encapsulado dentro de estos objetos.

Antes de continuar, cabe destacar una cosa muy importante, desgraciadamente la definición de programación y diseño orientado a objetos varia considerablemente. Hay definiciones en las que Smalltalk se considera un lenguaje orientado a objetos pero lenguajes como C# o Java no. La situación se complica más cuando dependiendo de las definiciones. Se pueden considerar ciertos diseños de programas orientados a objetos y funcionales o imperativos al mismo tiempo[23][24].

Es importante también tener en cuenta que existen definiciones de OOP que no tienen algunos de los problemas que se van a tratar en este apartado. Por ejemplo, existen definiciones en las que se da mucha más importancia a los mensajes entre los objetos que a los propios objetos.

También es importante tener en cuenta que existen importantes diferencias entre definiciones de programación orientada a objetos y diseño o modelado orientado a objetos.

Dicho esto, a continuación se va a hablar de la programación orientada a objetos más "popular"que esta basada en principios como los definidos por el acrónimo SOLID, los principios de encapsulación, abstracción, herencia y polimorfismo y que esta implementada en lenguajes como Java y C#.

Este tipo de diseño orientado a objetos incita a un diseño en el que el desarrollador intenta crear objetos que encapsulan los datos y ofrecen distintos métodos para modificar estos mismos. Estos objetos suelen representar elementos que existen en el modelo mental del problema a solucionar.

Es decir, si se quiere crear un juego y se tiene una silla dentro del mundo de juego, Se crearía un objeto que representaría esta silla. Se encapsularían los datos de la silla y se crearían métodos para manipular el estado de esta silla. Esto hace que se apliquen cuestiones del dominio del problema al propio diseño del código.

Si el resto de objetos del juego se intentan diseñar de la misma manera. Es muy posible que aparezcan problemas de diseño si se quiere crear una silla con propiedades físicas, otra silla que se use solo para cuestiones de gameplay, etc...

En cuanto se tengan los suficientes tipos de objetos este diseño empieza a causar problemas de mantenibilidad y rendimiento. Si el diseño cambia de tal manera que se quiera añadir una silla que funcione como un objeto de físicas y otra silla que funcione solo para cuestiones de gameplay, generalmente se tiene que rediseñar en gran medida el diseño completo de los objetos que representan sillas.

El modelo entero de que existe un objeto del tipo silla con sus datos encapsulados y métodos deja de tener sentido. Se tendrían que crear dos tipos de sillas y compartir los datos comunes de distintas maneras. Ya sea mediante herencia, lo cual no es recomendado por distintas razones, o teniendo un objeto que sea una composición de objetos.

Cabe destacar que este tipo de diseño es uno de los diseños que se usaba inicialmente en la industria de los videojuegos y se dejo de usar por los problemas mencionados.

El diseño orientado a objetos es relativamente bueno soportando cambios que se han previsto por el desarrollador. Pero los tipos de cambios que modifican el modelo mental del problema son los que generan muchos problemas[22]. La transición entre estos tres casos presentados sobre el diseño de una silla suele ser muy compleja si se ha seguido un diseño orientado debido a las grandes diferencias entre diseños y las repercusiones que tendrá en el resto del programa que use este diseño.

Otro de los problemas que tiene el intentar orientar todo a objetos es que estos objetos generalmente tienen una gran cantidad de datos agrupados, esto se ve amplificado más aun si un objeto tiene referencias públicas a otros objetos (Que según ciertas definiciones de OOP, esto no estaría permitido). Esto no es ideal pero generalmente es lo que acaba sucediendo. Todo esto provoca que si se quiere acceder a un dato individual, se tienen que acceder al objeto entero.

Una crítica famosa sobre la programación orientada objetos (parafraseada y traducida) por parte de **Joe Armstrong**, uno de los creadores del lenguaje

de programación **Erlang**, es que el problema con este tipo de programación y los lenguajes que la implementan, es que tienen todo un entorno implícito de datos que transportan de un lado a otro. Una metáfora que explica esto seria que **se intenta coger un plátano, pero se acaba cogiendo la jungla entera con un gorila que sostiene el plátano.** [25]

3.1.4. Unidades demasiado pequeñas

La programación orientada a objetos también incita a la **creación de unidades lo más pequeñas posibles y que realicen un solo trabajo**[26][27]. Esto en teoría es bueno ya que individualmente estas unidades parecen estar muy bien organizadas y parece que son fáciles de cambiar.

En la práctica esto lo único que consigue es mover la complejidad del sistema a otro sitio. Aunque las unidades individuales sean simples. La gran cantidad que suele haber y la necesidad de comunicación entre ellas se convierte en un problema muy grande y es una fuente considerable de fallos y perdidas. Esto es uno de los problemas que tienen por ejemplo los tipos de arquitecturas como la arquitectura de micro-servicios.

3.1.5. Optimización

Una de las ventajas que tiene el diseño orientado a datos es que incita a diseños que generalmente son más sencillos de optimizar.

Esto se debe a que este tipo de diseño incita al análisis de los datos, su procesamiento y el contexto en el que trata con estos datos. El análisis del contexto especialmente es lo que permite realizar distintos tipos de optimizaciones, siendo las más notables las optimizaciones que reducen los fallos de cache.

Si se ignora este contexto y se realiza un único diseño que no lo tenga en cuenta o que intente ofrecer todas las posibles funcionalidades cuando estas no sean necesarias, el diseño tenderá ser más complejo y a tener un menor rendimiento por no tener en cuenta el uso real del software o por intentar acomodar dichas funcionalidades.[28]

Por ejemplo, el problema de tener que actualizar 100 entidades distintas en un juego es muy distinto al problema de tener que actualizar 10.000.000 entidades. Las soluciones a estos dos problemas a su vez también son generalmente distintas.[29][28]

Otro ejemplo seria usar una librería diseñada para leer modificar y guardar archivos JSON con la finalidad de solo leer y parsear archivos JSON relativamente pequeños. La librería tendrá que tener en cuenta múltiples tipos de uso distintos lo cual hará que tenga un rendimiento menor y una complejidad mayor.

Es importante tener esto en cuenta porque por ejemplo se puede dar el caso de que estas librería se usen para cargar datos del juego en el motor por ejemplo. Los compromisos que ha hecho la librería para ofrecer tantas funcionalidades pueden provocar que, por ejemplo, se carguen los datos del juego en 3 segundos cuando se podrían cargar en 0.001 segundos. Si este proceso se realiza varias veces a lo largo del día de varios desarrolladores se empiezan a tener perdidas considerables de tiempo a lo largo de múltiples semanas y meses.[30]

"La optimización prematura es el origen de todos los males" es un dicho popular en el mundo del desarrollo de software. Por desgracia en la mayoría de los casos es usado de manera incorrecta. Provocando que el desarrollador se olvide completamente del rendimiento del software que esta desarrollando hasta que esto sea un problema. Creando múltiples capas de abstracciones las cuales acaban haciendo que todo sea más lento uniformemente. En muchos casos una vez esto se convierte en un problema no suele existir una manera sencilla de solucionarlo.[22]

Esto no significa que se tenga que optimizar todo al máximo desde el primer momento. Pero si se debería hacer un análisis de en que contexto se va a usar el software que se esta desarrollando, cuales son las restricciones de rendimiento que tiene que tener un sistema, cual es el orden de magnitud de la cantidad de datos que van a procesar dichos sistemas y **siempre medir el rendimiento que se esta teniendo y comprender el porque antes de intentar realizar cualquier optimización**.

Cualquier proyecto tiene estas restricciones aunque no se definan. Puede que sean más o menos restrictivas pero siempre existen implícitamente.[29]

3.1.6. Especulación de como va a cambiar el código

La programación orientada a objetos incita a generalizar el código de manera especulativa y hacer que pueda usarse aunque existan cambios en el futuro o se quiera extender de distintas maneras.

El desarrollador asume como se va a extender este código y como se va a modificar en este futuro. Mientras tenga razón, el diseño seguramente se sostenga. Pero es muy posible que existan cambios que no se tuvieron en cuenta, debido a que no se puede predecir el futuro.

Estos cambios pueden romper en gran parte el diseño y suelen llevar a un gran periodo de refactorización debido a la segregación de datos y lógica que se hizo en el momento y que con el nuevo modelo mental no tiene ningún sentido. [31] [32] [24]

3.1.7. Los Datos como pilar principal

Todo programa que ha existido hasta el momento tiene una única finalidad. Coger datos de entrada, procesarlos y generar datos de salida. Este esquema se aplica a cualquier pieza de software. Es la manera en la que funcionan las propias CPUs. Cualquier cuestión que directamente no trate este proceso es algo extra añadido por distintos motivos.

El diseño orientado a objetos acaba creando demasiadas capas de abstracción lo que complica exponencialmente los problemas que se quieran solucionar, El querer añadir una funcionalidad nueva sencilla puede provocar el tener que cambiar una gran cantidad de objetos y conexiones.

El diseño orientado a datos se centra en estos mismos y en las transformaciones que tienen que sufrir para conseguir el resultado que se busca. Ya sea aumentar la vida actual de un personaje cada segundo o convertir números que representan posiciones de vértices de un quad en píxeles que se muestran por pantalla.

Este tipo de diseños incita a que se definan estos datos y todas sus transformaciones. No incita a crear un sistema que pueda solucionar el problema a tratar pero también todos los problemas que estadísticamente hay una baja

probabilidad de que aparezcan en el futuro. Se encarga de resolver el problema actual que se ha definido anteriormente dentro de los parámetros y restricciones que se han analizado. Junto a esto, este diseño incita al análisis de cuales son las probabilidades de que se tenga que modificar el sistema y de que manera. Habiendo realizado este análisis se puede diseñar el sistema de tal manera que sea más sencillo implementar los cambios que estadísticamente tienen una probabilidad suficientemente alta de ser necesarios.

Si se necesitan realizar cambios en la funcionalidad del sistema. Se analizan y modifican los datos y se modifica la lógica que los procesa de la manera más apropiada. No existe el problema de tener que rehacer todo el modelo de objetos y su comunicación para que se adapte al nuevo modelo mental del problema que tiene el desarrollador el cual puede ser completamente subjetivo. Esto se debe a que no existen múltiples capas de abstracción, simplemente existen datos y operaciones que se realizan sobre estos.

El dejar que los datos simplemente existan y no intentar juntarlos con las operaciones que se realizan sobre ellos genera un diseño sobre el cual es mucho más fácil razonar. Siendo una de las razones que es mucho más simple. **El diseño orientado a datos no incita a esconder todas las transformaciones dentro de objetos y dentro de cadenas gigantes de llamadas de funciones entre múltiples objetos.** No incita a crear un modelo simbólico de .ºbjetos"basándose en el modelo mental subjetivo del desarrollador en un momento dado, el cual puede cambiar drásticamente.

El centrarse en representar los datos que se tienen y como se procesan, tiene la gran ventaja de que facilita la creación de diseños que aprovechen la gran cantidad de núcleos que tienen las CPUs actualmente. Al representarse todo mediante estructuras de datos principalmente planas. Si se tienen por ejemplo 10.000 datos que se quieren procesar. Estos datos se definir de tal manera que se puedan dividir en 4 grupos distintos y puedan ser procesados en 4 hilos distintos de ejecución.

Aunque un análisis extenso de este tipo de diseño sale del alcance de este proyecto y existe poca literatura que trate estos temas. El libro Data-Oriented Design de Richard Fabian[22] explica en mucha más profundidad los principios del diseño orientado a datos. Junto a esto existen charlas de personas como Mike Acton[29] [33] [28] que hablan sobre el tema centrándolo en el desarrollo de motores y la industria de los videojuegos. Stoyan Niko-

lov mostrando las aplicaciones de este diseño en el diseño de sistemas de navegadores web[34] y Tony Albrecht [35].

3.1.8. Conclusiones

Existen momentos en los que tiene sentido diseñar un elemento como si fuera un objeto con estado interno encapsulado y con métodos. Pero en otros muchos casos se realiza esta decisión de crear un objeto para identificar todos los elementos que componen un programa sin ninguna razón, excepto la inercia de tener práctica programando en este paradigma.

En varias situaciones es útil tener en cuenta que todos los programas siguen el esquema de tener datos de entrada, procesarlos y generar una salida o una modificación en el sistema. Estos datos vienen dados en distintos formatos pero fundamentalmente son números enteros, números de coma flotante, caracteres etc... Finalmente estos datos son procesados mediante distintas operaciones y controlando el flujo de ejecución del programa mediante distintas sentencias como if's, for, while, etc...

Elemento como las clases, métodos virtuales, la instanciación de objetos, herencia o distintas implementaciones de polimorfismo. Son elementos extra que pueden o no ser ofrecidos por el lenguaje de programación con el que se trabaje y que pueden o no ser útiles dependiendo del problema que se quiere solucionar y el contexto en el que se soluciona.

La existencia de estos elementos junto con la existencia de distintas prácticas no significa que se tengan que usar solo porque es una costumbre común en la cultura de la programación o porque produzcan código que subjetivamente se considere "limpio". Todos los paradigmas como los procedimentales, funcionales, orientado a objetos, imperativos... Intentan solucionar problemas que se fueron encontrando al desarrollar software a lo largo de las últimas décadas[23]. **Es importante entender cuales son estos problemas, de que maneras se han intentado solucionar y cuales son las ventajas y desventajas que tiene cada una de estas soluciones. Analizando todas estas cuestiones de la manera más objetiva y científica posible.**[35][36]

3.2. Casos de estudio

Para este proyecto, se han analizado distintos motores completos e implementaciones y diseños ECS. A continuación se van a mostrar estos análisis, las cuestiones más interesantes que se observaron y las conclusiones a las que se llegaron.

3.2.1. Hazel Engine

Hazel[37] es un motor en C++ que esta actualmente en desarrollo, consta de una versión 3D y una versión 2D la cual en el momento en el que se escribe esta memoria es la principal.

Este motor fue un caso de estudio interesante ya que, aunque existan muchas contribuciones de múltiples personas en la plataforma de Github donde esta subido el código del motor, gran parte del diseño base fue realizado por una sola persona. Esto hizo que fuera una buena referencia para estimar el alcance de lo que podría desarrollar una sola persona en un periodo de tiempo determinado.

Fue de gran ayuda al principio del desarrollo gracias a que el creador grabo múltiples vídeos educativos en los que enseñaba su razonamiento y el proceso a nivel general que siguió para crear el motor desde cero.

En cuanto al diseño general, analizando las distintas partes que componían el motor se observo que seguía un diseño orientado a objetos.

Entry Point

El primer trozo de código que se ejecuta en el motor se encuentra en lo que se denomina el punto de entrada o **entry point** de la aplicación. Este motor junto con otros como **Halley**, implementa la función *main* dentro del propio código del motor, en este entry point se ejecuta la función que el juego/cliente tiene que definir para crear el objeto que representa la aplicación.

Definir el punto de entrada dentro del propio motor hace que el código del juego que no forma parte del motor no tenga que preocuparse de definir

el punto de entrada correcto para la aplicación.

Application

Clase que representa la propia aplicación, contiene parte la gestión de eventos de input. Esta gestión de eventos se realiza mediante el propio sistema de eventos del motor del que se hablara en los siguientes apartados.

La aplicación esta diseñada por lo que llamarían capas, una capa en general es objeto que tiene funciónes que se ejecutan por el motor cuando se activa o desactiva dicha capa, y cuando se actualiza el motor. También ofrece funciones para responder a eventos.

Este diseño por capas se usa para poder por ejemplo, hacer que una capa contenga toda la información y controles de debug y esta este por encima de la capa del propio juego. De esta manera si se quiere deshabilitar toda la vista de debug se puede deshabilitar la propia capa entera.

También tiene utilidad a la hora de responder a eventos de input, si por ejemplo el usuario hace click en un punto de la aplicación, este evento de click se propagara desde la capa más alta hasta la más baja, si por ejemplo se hace click en un elemento de la interfaz de debug, este elemento puede consumir el evento y hacer que no se propague al resto de capas del juego.

Eventos e Input

Este motor usa GLFW para la creación de la ventana y gestión de eventos de input, pero implementa un sistema de eventos propio encima de los eventos de GLFW. Esto le permite por ejemplo dejar de usar GLFW o usar otra librería y tener que hacer cambios mínimos para que el resto del motor que use los eventos siga funcionando.

El motor define sus propios códigos para cada una de las teclas de input, no define todos estos códigos desde cero porque no tendría sentido en si, copia los códigos de implementaciones existentes como GLFW. Aun así, esto puede ser útil porque aunque los códigos sean iguales en este caso, se crea una capa de abstracción entre los códigos de GLFW y el propio motor que

usa sus propios códigos.

Asserts, Defines y Plataforma

El motor define distintos tipos de asserts internos solo para el motor y públicos para el propio juego, estos asserts están definidos para poder ser quitados en las builds que no sean de desarrollo. existen también defines para los distintos tipos de logs que se pueden crear.

Se definen aliases para tipos de datos como por ejemplo los punteros inteligentes.

También implementa código para detectar automáticamente la plataforma para la que se esta intentando compilar el código basándose en macros predefinidas para dichas plataformas.

Modelo de objetos de gameplay

Este motor usa la librería **Entt**[38] para crear su modelo de objetos de gameplay, esta librería ofrece un sistema ECS con bastante rendimiento.

Herramientas y Editor

El motor consta de un editor escrito en C++ el cual usa la librería de **IMGUI**[39]. Cabe destacar que la versión subida a github es una versión antigua y no tiene casi ninguna funcionalidad, la versión actual es de pago.

3.2.2. Blah

Este motor esta siendo desarrollado principalmente por Noel Berry[40], uno de los desarrolladores del juego **Celeste**[41].

A continuación se va a hablar de todo lo que se observo en el estudio de este motor.

Estilo General

Una de las cuestiones interesantes que se observaron a lo largo del análisis es la gran mayoría del código esta altamente comentado. Esto siempre es una buena práctica si es necesario para entender el funcionamiento, lo interesante que se observo es que la gran mayoría de los comentarios a primera vista parecían redundantes pero aun así aclaraban o confirmaban suposiciones que sin los comentarios no estarían completamente claras.

Junto a los comentarios, todos los *headers* del motor están muy organizados para que sean lo más legibles posible.

Application

Para ejecutar el motor lo primero que tiene que hacer el juego es crear una estructura de datos con la configuración, esta estructura se pasara como parámetro a la función que ejecutara el motor. En esta configuración se pueden asignar tanto datos primitivos como funciones.

La aplicación no es una clase o un objeto en si, si no una colección de funciones dentro del *namespace* de *app*. Se asume que esto es debido a que no tiene sentido crear una clase y un objeto para la aplicación si se sabe que solo va a existir una sola. Este diseño se repite a lo largo de todo el motor.

Time

Estructura de datos que contiene todo el estado relacionado con el tiempo de ejecución del motor, junto a esto contiene funciones que facilitan ciertas operaciones relacionadas con el tiempo.

Input

Define sus propios codigos de teclas (Keycodes) a partir de unos ya existentes. Una cuestión interesante en cuanto a la implementación es el uso de distintas macros para facilitar la definición de estos keycodes.

Ofrece soporte a mandos y un sistema de bindings para que el juego/jugador pueda reasignar los controles del juego.

Una cuestión que aunque pequeña fue bastante interesante fue el análisis de la implementación de como se calculaba el estado de una tecla en un momento dado.

Todo el estado de input esta guardado en estructuras de datos. Por cada tecla se guarda si ha sido presionada este fotograma, si se ha soltado, y si se esta manteniendo. Cada mando/control tiene su propia estructura de datos en la que guarda los mismos datos de cada botón.

Para comprobar si se ha presionado el botón de un mando en especifico se tienen que realizar varias comprobaciones como por ejemplo, el ID del mando y del botón tienen que estar dentro del rango correcto y después se tiene que comprobar que efectivamente se ha presionado. Esto parece que se calcula evitando el uso de *branches*. Si este es el caso, esto podría ayudar ligeramente minimizando los fallos en predicciones de branches. Cabe destacar que esto no se sabe si es intencional o es simplemente una cuestión estética.

File System

El sistema de archivos ofrece funcionalidades para abrir, leer, escribir o destruir archivos binarios. Contiene también funciones de utilidad relacionadas con directorios y con la dirección/path de archivos.

Asserts

Define un assert el cual se puede desactivar en las builds que no sea de desarrollo, una cuestión interesante es el hecho de que rodea todo el código del assert dentro de la sentencia *do ... while(0)*. Tras investigar se llego a la conclusión de que esto se hace para evitar fallos en el caso en el que se usen sentencias *if() ... else ...* sin el uso de *brackets*.[42]

Es interesante también el uso de macros como *___FILE___, ___LINE___* en el mensaje de los asserts, esto ayuda a que se muestre el archivo y la linea de código en la que no se cumplió un assert.

Junto a esto, también se usa la función *abort()* de la libreria *cstdlib*. Si existe un debugger activo, esto provocara un breakpoint como lo haría la implementación de *__debugbreak* de **MSVC**, si no, entonces se terminaría el programa.

Common

El motor define aliases para todos los tipos de datos primitivos y también para los punteros inteligentes con la finalidad de poder modificarlos fácilmente sin modificar el propio código del motor.

Logging

En cuanto a la implementación del logging, usa un sistema simple pero efectivo basado en el uso de *va_lists*. Los mensajes se ven formateados de la misma manera que la función *printf* los formatearía. Esto se consigue usando la función *vsnprintf*.

Operaciones Matemáticas

Crea wrappers para las funciones matemáticas básicas como *seno, coseno, tangente, valor absoluto, etc...* que ofrece la librería *corecrt_math.h*. De esta manera el motor no depende directamente de esta. También define constantes matemáticas como π, τ y algunos ángulos en radianes.

En el análisis una cosa que destaco es que estas funciones que son unicamente wrappers no se definieron como *inline*.

Color

El motor ofrece una implementación de una estructura de datos simple que representa un color de 32 bits.

Easings

El motor ofrece una librería con lo que parecen ser todos funciones para los tipos más comunes de easings. Este tipo de funciones son especialmente útiles a la hora de programar cuestiones de gameplay, interfaces o animaciones.

Vectores y Matrices

El motor usa su propia implementación de vectores y matrices usando templates. También define tipos de datos como rectángulos, quads, lineas y círculos.

Un apunte interesante en cuanto a la implementación es que define aliases de las templates con los tipos de datos más usados, es decir, define un alias para los vectores de dos coordenadas que sean floats, otro alias si son ints, etc...

Imágenes y Texturas

Representa una imagen en la CPU como un objeto. En esencia es un buffer que contiene los datos de la imagen. Ofrece métodos interesantes para conseguir la información de la imagen dentro de una sección dada dentro de esa imagen. Ofrece también la posibilidad de sobre-escribir los datos de la imagen. Estos se usan específicamente a la hora de crear atlases que combinen múltiples imágenes para poder renderizar todas esas imagenes en una sola *drawcall* (batch rendering).

Las imágenes están representadas con el valor del alpha pre-multiplicado. Esto se debe a que en renderizado por ordenador produce mejores resultados a la hora de componer múltiples capas de imágenes[43].

Una textura es la representación en GPU de una imagen. En esencia es un wrapper de las funciones de la API gráfica relacionadas con texturas.

Fonts

Representación de una tipografía, contiene todos los datos de dicha tipografía como los glifos, valores de interletraje y propiedades generales. A partir de esta información se puede crear una representación en un bitmap.

Packer

Este modulo tiene la funcionalidad de combinar múltiples imágenes en un gran atlas para poder renderizarlas de una manera más eficiente. Esto es especialmente útil a la hora de renderizar texto, ya que se puede crear un bitmap con todos los caracteres que se quieran usar.

Vector, Stack Vector y String

El motor tiene su propia implementación de un vector sencillo de datos, esto suele ser bastante común para tener mayor control sobre el uso de memoria y conseguir mayor rendimiento.

Stackvector este tipo de contenedor de datos que es una mezcla entre un array un vector pero que existe en el stack. Tiene una capacidad máxima fija definida al compilar.

Existen otras implementaciones que permiten indicar la capacidad en el stack al compilar pero si se excede este límite, se reserva memoria en el heap para los elementos nuevos. No es el caso en la implementación de stack vector.

Flujo de Ejecución

El motor sigue el siguiente flujo de ejecución:

1. El juego configura y ejecuta el motor.
2. Inicialización de la plataforma y la ventana.
3. Inicialización de los gráficos.

4. Inicialización del input.
5. Se actualiza el input para resetear los datos.
6. La plataforma actualiza los datos de input.
7. Se ejecuta el game-loop.
    a) Se procesa el game-loop (Tiempo fijo).
    b) Se actualiza el input.
    c) Se actualiza el modulo de gráficos (Este paso no hace nada ya que en la versión analizada los gráficos estaban dirigidos por una función llamada *on_render* la cual el juego definía).
    d) Se ejecuta el callback de update definido por el juego.
    e) Se ejecuta el callback de render definido por el juego.
    f) Se presenta la imagen.
8. Se liberan los recursos de los sistemas.
9. Se termina el ciclo de la aplicación.

Gestión de memoria

La gestión de memoria de recursos como las imágenes se realiza mediante *shared pointers*. El propio cliente/juego recibe un shared pointer al recurso creado el cual se liberara cuando este shared pointer deje de estar referenciado en cualquier punto del programa.

Modelo de Objetos/Entidades

Este motor no define ningún tipo de modelo de objetos o entidades. Ofrece un control de relativamente bajo nivel que permite crear y mostrar imágenes en pantalla de manera directa.

3.2.3. Game Engine Architecture

Game Engine Architecture de Jason Gregory es un libro que destaca por ser una guía de la arquitectura y practicas usadas en la creación de motores de videojuegos.

Es considerado uno de los mejores libros sobre el tema de motores de videojuegos.

Organiza las funcionalidades de un motor en distintas capas, las capas superiores usan las funcionalidades de las capas inferiores pero no viceversa en general al menos.

Estas serian las capas más generales:

Sistema operativo, Drivers y Hardware

Esta capa es la de más bajo nivel posible, contiene el propio sistema operativo sobre el que se crea el motor, los drivers que se usan, y cuestiones específicas del la arquitectura del hardware.

Third Party SDK

Esta capa contiene todas las librerías externas que se vayan a usar en el motor como por ejemplo las API's gráficas, los motores de físicas, las librerías de contenedores de datos y algoritmos, librerías para la Animación 3D, etc...

Platform Independece Layer

Esta capa se encarga de abstraer distintas implementaciones concretas de la plataforma en la que se trabaje. Esta capa es especialmente necesaria para juegos multi-plataforma.

En esencia esta capa es un wrapper de todas las funcionalidades de bajo nivel de la plataforma. Tiene la finalidad de ofrecer una API consistente al resto del motor.

Abstrae distintas funcionalidades como:

- Tipos de datos primitivos como int32, int16, float32, float64...
- Acceso al sistema de archivos de bajo nivel
- Networking de bajo nivel
- Gestión del tiempo y cronómetros de alta-precisión
- Threading
- Wrappers de la API gráfica

Esta capa también es responsable de detectar la plataforma en la que se encuentra y configurar el motor para dicha plataforma.

Cabe destacar que es relativamente sencillo, aunque tedioso, hacer un wrapper para una API como por ejemplo para una API gráfica. La dificultad que existe es intentar abstraer APIs que ofrecen un distinto nivel de control como por ejemplo Vulkan y OpenGL ya que se puede acabar perdiendo rendimiento.

Como estas dos APIs tienen que acabar pasando por esta abstracción. Es muy fácil hacer que esta abstracción tenga solo las partes comunes entre las dos APIs. Pero eso haría que el resto del motor no pueda usar las funcionalidades especificas que hacen que Vulkan pueda tener un mayor rendimiento.

Al contrario si se exponen funciones especificas de Vulkan en esta abstracción, aparece el problema de decidir que es lo que ocurrirá con esas funciones cuando se este usando OpenGL.

El caso de usar la misma abstracción para Vulkan y OpenGL es un caso extremo debido a las grandes diferencias entre las dos APIs. Aun así, si por el contexto del motor que se vaya a crear es absolutamente necesario usar múltiples APIs gráficas, hay que tener en cuenta el coste de mantenimiento y rendimiento que estas conllevan.

Core Systems

Esta capa contiene la gran mayoría de utilidades que van a ser usadas o expandidas por las capas superiores:

- Assertions, las cuales son eliminadas de las build finales
- Logging y Printing de texto
- Profilling e Instrumentación
- Gestión de memoria (Custom Allocators, Hierarquías de memoria...)
- Matemáticas
- Contenedores, estructuras de datos y algoritmos (Aunque existen librerías que ofrecen este tipo de estructuras, en muchas ocasiones se implementan versiones personalizadas por cuestiones de memoria y rendimiento.)
- Strings & Hashed String IDs (Los strings son usados ampliamente en videojuegos y es bastante común tener implementaciones personalizadas por razones de memoria y rendimiento. También se usan como IDs legibles para assets o objetos, estos IDs son posteriormente transformados en versiones optimizadas para el hardware como números enteros)
- Localización (Generalmente es una buena idea implementar la localización lo más pronto posible si va a ser necesario en el motor)
- Parsers de JSON, CSV, etc... (Suelen ser útiles para la gran mayoría de sistemas del motor de una manera u otra, se pueden usar para parsear configuraciones de sistemas e incluso los propios datos del juego ya que estos datos suelen ser generados por un pipeline de herramientas y suelen estar en un formato legible para los personas como JSON, el cual posteriormente se convierte en un formato binario optimizado)
- Generadores de Números aleatorios (Muchos sistemas sobretodo sistemas de gameplay acaban usando números aleatorios de una manera u otra)

- IDs únicos y Handles de objetos (Los IDs o Handles son muy usados ampliamente assets)
- Curvas y superficies (Bezier, B-Splines... Son útiles a la hora de describir muchas propiedades que componen un juego)
- Runtime Type Introspection (RTTI) y Serialización (Estas dos funcionalidades se presentan juntas porque sistemas RTTI permiten escribir código genérico para serializar distintos tipos de datos. Estos dos sistemas son muy útiles a la hora de crear un editor para el juego)[44]
- Operaciones de I/O de archivos asíncronas

Gestión de Recursos

Esta capa ofrece una API para manipular todos los tipos de assets que usa el motor como por ejemplo modelos, texturas, escenas, archivos de descripción de elementos de gameplay...

Algunos motores ofrecen una API en la que se diferencian distintos tipos de assets mientras que otros tratan todos los archivos iguales de manera genérica.

Motor de Renderizado

El motor de renderizado, especialmente si es un motor 3D, puede llegar a ser un sistema muy amplio y complejo. El propio renderizado esta dividido en subcapas/módulos.

Low-Level Renderer o renderer de bajo nivel es la primera capa en el motor de renderizado.

Esta capa ofrece una API para todas las funcionalidades de renderizado de bajo nivel. Esta capa se encarga de preparar y emitir drawcalls de todas las primitivas que se le envíen sin realizar ninguna comprobación de que sean visibles o no. Se encarga también de inicializar todo el contexto gráfico.

Esta capa generalmente abstrae los siguientes elementos:

- Materiales y Shaders (Sistema de materiales)
- Iluminación estática y dinámica
- Camaras y Viewports
- Texto y Tipografías
- Texturas
- Primitivas de Debug

Scene Graph, Subdivisión espacial y Culling es la siguiente capa del motor de renderizado. Esta capa limita y filtra todas las llamadas que se envían al renderer de bajo. Los métodos para hacer esto varían ampliamente dependiendo del tipo de juego. Algunos juegos pueden solo implementar simplemente un **frustum culling** mientras que otros motores necesitan implementar estructuras de subdivisión espacial.

A continuación esta la capa de **efectos visuales**. Esta capa se encarga de gestionar partículas, decals, lightmapping, postprocesado, etc...

La capa de **front end** se encarga de gestionar la interfaz generalmente 2D del juego. Los menús, cinemáticas dentro del juego, renderizado de vídeos, etc...

Por ultimo generalmente existe una capa de **renderizado especifico del juego**. Esta capa se encarga de cuestiones como el renderizado de terrenos, simulaciones y renderizado de agua...

Audio

Esta capa generalmente se encarga de proporcional todas las funcionalidades de audio que pueda necesitar el motor, el tamaño y complejidad de este modulo puede varia ampliamente dependiendo de los requisitos que se tengan. Estas son algunas de las funcionalidades que puede ofrecer esta capa[45]:

- Un modelo de audio 3D

- Efectos DSP
- Soporte para localización
- Sincronización de sonidos
- Streaming de audio
- Obstrucción y oclusión de sonidos
- Sonidos Multi-canal
- Reverberación
- Soporte de Música y Música dinámica
- Soporte para sonidos de ambiente.
- Requisitos específicos de la plataforma

Gestión de Input y HID

Esta capa se encarga de la gestión de HID (Human Interface Device) como por ejemplo teclados, ratones, mandos, joysticks, etc...

Networking

Aunque se empiece a hablar de la capa de networking en este punto. Esta capa tiene una gran influencia en muchos aspectos del motor. Si tenerla es un requisito indispensable. Es una buena idea diseñar desde el primer día todos los sistemas teniendo en cuenta networking.

Gameplay Foundations

Esta capa contiene todas las herramientas y funcionalidades para crear el comportamiento específico del juego.

Estas funcionalidades pueden ser sistemas de scripting ya sean por texto o visual y generalmente implementaciones de maquinas de estado y flow general

del juego. Este tipo de implementaciones suelen ser increíblemente útiles ya que gran parte de lógica de un juego se puede expresar de una manera sencilla utilizando estos diseños. [46] [47] [48]

3.2.4. Halley Engine

Halley es un motor ligero multiplataforma con us sistema ECS escrito en C++ usado para crear un juego de estrategia por turnos llamado **Wargroove**.

Este motor se estudio principalmente por su arquitectura general. También se analizo parte de la implementación. Una de las cuestiones más interesantes que se observo es el gran uso de *unique pointers*,*shared pointers* y las transferencias de *ownership* que hay entre sistemas de estos *unique pointers*

El motor esta dividido en los siguientes proyectos:

- Engine
  - Core
  - Audio
  - Entity
  - Utils
  - Networking
  - UI
- Plugins
  - Asio
  - DX11 / OpenGL / Metal
  - SDL / WinRT
- Tools
  - Editor
  - CMD

- Runner
- Tools

El proyecto Engine implementa todo el core del motor junto con los sistemas de audio y de entidades. No implementa cuestiones especificas como la API gráfica de bajo nivel o cuestiones especificas de la plataforma. Estas partes están implementadas en el proyecto de Plugins

3.2.5. Kruger Engine

Kruger es un prototipo de motor que esta en desarrollo. Esta siendo creado principalmente por una sola persona llamada Bobby Anguelov. Implementa una gran cantidad de sistemas pero no implementa un renderer. Contiene todos los sistemas bases de un motor que gestionan la serialización, logging, matemáticas, etc... Junto a esto implementa:

- Modelo hibrido de objetos/entidades
- Pipeline de recursos con GUI
- Sistema de animación mediante grafos
- Editor simple
- Editor de ragdolls

La parte que principalmente se analizo fue el modelo híbrido de objetos/entidades que implementa el motor. Este modelo intenta mezclar las cualidades que tienen los sistema basados en actores con las ventajas que tienen los sistemas ECS.

De manera general, mantiene el concepto de entidades y componentes que mayormente solo son contenedores de datos (aunque las entidades pasan a ser un objeto en lugar de ser solo un ID). La gran diferencia es que define dos tipos de sistemas, unos locales que se ejecutan individualmente por entidad y otros globales que serian los sistemas ECS mas clásicos.

Los sistemas locales se añaden a las entidades individualmente y se encargan de modificar el estado interno de la entidad. Los sistemas solo tienen acceso al estado de la propia entidad.

Cuando una entidad de actualiza, lo único que se hace es actualizar todos sus sistemas locales. Esto permite ejecutar las actualizaciones de las entidades en paralelo en distintos *work threads* sin que existan problemas de acceso/modificación de datos.

Los sistemas globales se pueden considerar sistemas más clásicos de ECS, estos sistemas operan sobre todos los componentes de tipos específicos que existan en el mundo. Se encargan procesar la lógica que afecta al estado global del mundo y también de transferir datos entre las entidades

3.2.6. Bevy Engine

Bevy es un motor open-source en desarrollo implementado en Rust. Usa un modelo ECS para representar el mundo del juego e implementa un renderer 2D y 3D.

Este motor se estudió principalmente con el objetivo de encontrar distintas maneras de crear una API pública en código para el motor. Para crear una aplicación usando Bevy, el juego tiene que crear la propia aplicación usando una función predefinida por el motor. Esta aplicación contiene funciones para acceder y configurar la gran mayoría de los parámetros del motor. Utiliza lo que comúnmente se llama *builder pattern* para crear esta API.

3.2.7. Austin Morlan ECS

A continuación se va a analizar una implementación sencilla de ECS creada por Austin Morlan[49]. Se van a analizar todos los componentes que forman esta implementación y describir su funcionalidad.

Entity

Aunque existan implementaciones que crean una estructura con información extra para representar una entidad. Esta implementación trata las entidades como simplemente un ID.

Component

Un componente es simplemente un contenedor de datos, cada tipo de componente tiene un ID asociado que lo identifica.

Signature

Para identificar rápidamente el set de componentes que tiene una entidad se utiliza el concepto de *signature*. Una estructura de datos que principalmente funciona como una mascara de bits. Cada una de las posiciones de esta máscara esta asociada con cada tipo de componente.

De esta manera se puede calcular comparando mascaras de bits si una entidad tiene los componentes necesarios para ser procesada.

Component Array

Una estructura de datos que tiene la finalidad de mantener los datos de los componentes alineados consecutivamente en memoria.

Para conseguir esto, se crea un array que contenga los componentes contiguos y un array que, dado el ID de la entidad, devuelva el índice en el que esta su componente dentro del array de componentes contiguos. También implementa un array que, dado el indice del componente dentro del array, devuelve el ID de la entidad.

Esta implementación se asemeja a un tipo de datos normalmente llamado **sparse set**.

System

Clase base que representa un sistema en el modelo ECS, este sistema contiene una **signature** que indica el tipo de componentes que necesitar tener una entidad para ser procesada por este sistema y un array en el que se tienen referencias a todos los IDs de las entidades que tienen los componentes de la signature del sistema.

Entity Manager

Uno de los tres managers de la implementación. Este manager se encarga de crear y destruir las entidades, proporcionar IDs libres que identifiquen estas entidades y modificar su **signature**.

Component Manager

Se encarga de crear y modificar todos los arrays de componentes. Por simplicidad en esta implementación todos los componentes tienen que ser registrados previamente por el usuario para ser usados. Existen implementaciones como la de *EnTT* que no tienen necesitan este paso.

Systems Manager

Se encarga de registrar sistemas nuevos, actualizar las listas de entidades que mantienen los sistemas y de actualizarlos.

Coordinator

Su única funcionalidad es coordinar las funcionalidades de los tres managers que se han tratado en los apartados anteriores. También sirve como punto de entrada para el usuario al sistema ECS.

3.2.8. Overwatch ECS

Overwatch es uno de los pocos juegos triple A relativamente modernos que usan un sistema ECS y han enseñado al público parte de su diseño[5]. Estas son las cuestiones más interesantes que se han analizado:

Componentes Singleton: Overwatch implementa el concepto de componentes singleton, estos son un tipo de componentes globales que solo tienen una instancia en todo el mundo del juego. Se usaron para guardar datos como por ejemplo el input del jugador. Son extremadamente útiles para guardar cualquier tipo de información que no vaya ligado a una entidad. Hasta un 40 % de los componentes que implementa overwatch son componentes singleton.

Aplazamiento de operaciones: Overwatch implementa sistemas que modifican el estado del juego en distintas maneras, algunos sistemas simplemente leen el estado y hacen modificaciones pequeñas. El problema es que existen otros sistemas que cogen esas modificaciones pequeñas realizadas anteriormente y basándose en ellas realizan grandes cambios en el mundo del juego.

Una técnica relativamente sencilla que usaron para controlar cuando se ejecutaban este tipo de modificaciones al mundo del juego se basa en aplazar dichos cambios hasta un momento futuro dentro del propio fotograma.

No forzar todos los sistemas dentro del paradigma ECS: Una de las cuestiones importantes que se observaron en el desarrollo de overwatch es que es mejor no intentar forzar ciertas funcionalidades del juego dentro del propio modelo ECS si no tiene sentido hacerlo. Esto se puede ver en la implementación de las habilidades de los personajes. Todas las habilidades están creadas mediante un sistema llamado *StateScript*[48].

Este sistema es un propio lenguaje de scripting que usa grafos en el que se definieron todas las habilidades y lógica de estas mismas. El sistema ECS lo único que contiene es un componente con datos que usa este lenguaje de scripting y un sistema que lo ejecuta.

Este mismo modelo se puede observar en la implementación de otros sistemas como el sistema de físicas. Este sistema esta casi en su totalidad implementado fuera del sistema ECS. Existe un sistema de físicas en la parte

ECS pero tiene la única funcionalidad de transmitir la información entre el mundo del juego y el propio mundo de físicas.

3.2.9. Unity ECS

Unity tiene actualmente en desarrollo un modelo ECS que forma parte de lo que llaman DOTS o Data Oriented Tech Stack. Una cuestión interesante de esta implementación es el como se guardan los componentes en memoria.[50]

Este modelo implementa lo que se llaman *arquetipos*. Estos son una combinación especifica de componentes que tiene un tipo de entidades. A la hora de guardar los componentes, se pueden guardar de manera que todos los componentes de un arquetipo están juntos en memoria, en lugar de tener cada componente separados por tipo. Es decir:

Si se tienen los componentes A, B y C. Estos componentes en la implementación de Austin Morlan de ECS por ejemplo, se guardarían en arrays separados formando el siguiente patrón.

- **Array A: A A A A A A A**
- **Array B: B B B B B B B**
- **Array C: C C C C C C C**

Esto hace que si se recorre solo un array de componentes de manera lineal, habrá pocos fallos de cache. Pero si se recorren múltiples arrays al mismo tiempo se producen fallos de cache porque se accede a posiciones de memorias que no están próximas entre si.

Unity lo que permite es guardar estos tres componentes de manera consecutiva para no tener que cambiar de array a la hora de iterar sobre ellos. Los componentes quedarían guardados de la siguiente manera en memoria.

- **Array ABC: A B C A B C A B C**

Esto aumenta el rendimiento si se itera sobre los tres al mismo tiempo pero lo reduce si se itera sobre solo uno.

3.2.10. Conclusiones

Tras el análisis de todos los casos tratados se llegado a las siguientes conclusiones.

Todos los motores tienen variaciones en la arquitectura pero de manera general todos o casi todos están divididos en ciertos módulos a alto nivel. Los módulos que existen en todos los motores estudiados son los que tratan cuestiones de la **plataforma, renderizado, input del usuario y gestión de recursos**. Junto a esto dependiendo de la complejidad del motor se añaden también módulos para representar objetos en el mundo del juego, para gestionar networking, etc...

Algunos de los motores estudiados como Hazel y Halley tienen un diseño mayormente orientado a objetos mientras que motores como Blah y Bevy tienen un diseño mixto entre orientado a objetos y procedimental.

En cuanto al modelo de objetos del juego, dentro de la arquitectura general ECS existen múltiples variaciones. Kruger implementa el modelo de objetos más interesante pero al mismo tiempo es el más complejo. El diseño de Austin Morlan es el más simple de ECS que se ha tratado pero a su vez no ofrece muchas de las funcionalidades necesarias para poder crear un juego completo.

El diseño de Overwatch a un alto nivel sigue siendo relativamente simple pero tiene funcionalidades extra como los componentes *singleton* que ayudan en gran medida.

Se decidió que se crearía un motor que inicialmente tuviera los módulos básicos que todos los motores estudiados tienen como el modulo de **plataforma, input, renderizado y gestión de recursos**. Siguiendo la arquitectura definida en el libro **Game Engine Architecture** y siguiendo los principios de diseño de Blah y Bevy.

Para el modulo ECS se decidió implementar una versión basada en el diseño de Austin Morlan que implementara también funcionalidades del modelo ECS de overwatch.

3.3. Estudio de alternativas

El motor que se ha realizado esta diseñado en C++ con OpenGL como API gráfica y con distintas librerías como GLFW para facilitar ciertos aspectos.

Aunque este sea el caso, se estudio la posibilidad de realizarlo en otros lenguajes de programación o con otras API's gráficas. En concreto una de las primeras decisiones que se tuvo que tomar fue si hacer el motor en **C++** o hacerlo en el lenguaje de programación llamado **Rust**

Rust es un lenguaje de programación de bajo nivel como C++. Se considero usar Rust debido a que es un lenguaje relativamente más moderno que C++, aunque este se vaya actualizando cada pocos años, y por esta razón tiene algunas funcionalidades que podían ser útiles en el desarrollo.

Se considero Rust especialmente debido a su amplia documentación moderna actualizada, las facilidades sintácticas que ofrece sin dejar de ser un lenguaje cuyo objetivo es genera código eficiente, concurrente y especialmente el hecho de que el diseño orientado a datos encaja perfectamente con el propio diseño del lenguaje y este mismo ofrece una gran cantidad de ayudas si se diseñan los programas de esa manera.

Uno de los sistemas más interesantes que ofrece es el llamado **borrow checker**, este sistema, de manera muy general, se encarga de asegurarse de que cada dato del programa tiene un propietario el cual en principio es el único que puede modificar estos datos a no ser de que se transfiera la propiedad de estos mismos.

Aunque se hubiera preferido usar rust para el proyecto, debido a la gran falta de experiencia del autor en el lenguaje y a que ya de por si el proyecto tendría bastante complejidad en distintos campos. Se decidió usar C++ ya que al menos en ese lenguaje se tenia algo de experiencia mínima y tenia la ventaja de ser el lenguaje más usado en el desarrollo de motores.

En cuanto a API gráfica, se eligió OpenGL por la experiencia que se tenia usando esa API. Aunque OpenGL sea relativamente fácil de usar y multiplataforma, en el desarrollo de juegos triple AAA a gran escala no es una API muy popular, sufre de limitaciones de rendimiento y concurrencia

que tiene por su diseño en comparación con otras opciones.

APIs como DX11-12, Vulkan o Metal ofrecen mucho más control lo que, en buenas manos, permite conseguir un mayor rendimiento. Cabe destacar que el usar correctamente estas APIs para conseguir un alto rendimiento no es una tarea trivial. Estas APIs ofrecen control de muy bajo nivel al desarrollador pero si no se usan de la manera más ideal, pueden llegar a producir un resultado bastante malo.

Dicho esto. Dependiendo del contexto en el que se vaya a usar el motor. Usar OpenGL puede ser una buena decisión.A menos de que tenga en requisito de tener que usar una API especifica por la plataforma o se necesiten renderizar escenas 3D con una alta fidelidad gráfica de ultima generación, OpenGL seguramente sea suficiente y la sencillez que ofrece en su uso seguramente ayude al desarrollo.

En el contexto de este motor OpenGL es la API que más sentido tenia usar por su sencillez, la familiaridad del autor con ella y porque en cuanto a rendimiento, al ser un motor principalmente 2D con un renderizado sencillo, se asumió que no se tendría ningún problema.

Capítulo 4

Diseño e Implementación

4.1. Análisis

A continuación se presentaran los requisitos funcionales y no funcionales que definen las funcionalidades que tiene el motor.

4.1.1. Requisitos Funcionales

RF1 El motor sera una librería estática de C++.

RF2 Se podrá configurar parámetros de la ventana mediante código.

RF3 Se podrán cargar imágenes png para ser usadas posteriormente en el motor.

RF4 Se podrán crear sprites con las imágenes previamente cargadas en el motor por el usuario.

RF5 Se podrá indicar la relación entre pixeles del sprite y unidades de distancia del mundo del juego.

RF6 Se podrán crear entidades en el mundo del juego y añadirles componentes.

RF7 Se podrán crear tipos de componentes nuevos que podrán tener las entidades.

RF8 Se podrán crear nuevos tipos de componentes globales.

RF9 Se podrán crear sistemas nuevos que sean ejecutados por el sistema ECS del motor.

RF10 Se podrán hacer queries dentro de los sistemas que devuelvan entidades que tengan un set de componentes específicos.

RF11 Se podrán hacer queries dentro de los sistemas que devuelvan componentes globales.

4.1.2. Requisitos No-Funcionales

RNF1 El motor tendrá una API simple.

4.2. Librerías

Para el motor se usaron las siguientes librerías externas:

GLFW: Librería multiplataforma para el desarrollo de aplicaciones que usen OpenGL, OpenGL ES o Vulkan, se ha usado esta librería para gestionar la creación de la ventana, el contexto de OpenGL y el input del usuario.[51]

Glad: Librería que se encarga de cargar los punteros a las funciones de OpenGL.[52]

GLM: Librería de matemáticas que contiene todo lo necesario para realizar cálculos con vectores y matrices.[53]

STB Image: Librería que proporciona funcionalidades para cargar y decodificar imágenes.[54]

Optick: Profiler sencillo diseñado para videojuegos que se usa principalmente para analizar y arreglar problemas de rendimiento en el motor.[55]

IMGUI: Librería para crear una interfaz de usuario de modo inmediato, usada para crear ventanas con información de debug.[39]

Box2D: Librería de físicas 2D simple.[56]

En el diseño inicial se planeo usar también las siguientes librerías pero por cuestiones de tiempo no se acabaron integrando.

EASTL: Implementación de los contenedores/algoritmos que contiene la librería STL pero con énfasis en gestión de memoria y alto rendimiento.[57]

4.3. Diseños Iniciales

El diseño inicial motor estaba dividido mayormente en capas/módulos de tal manera que las funcionalidades de las capas inferiores generalmente son usadas por las capas superiores pero no viceversa.

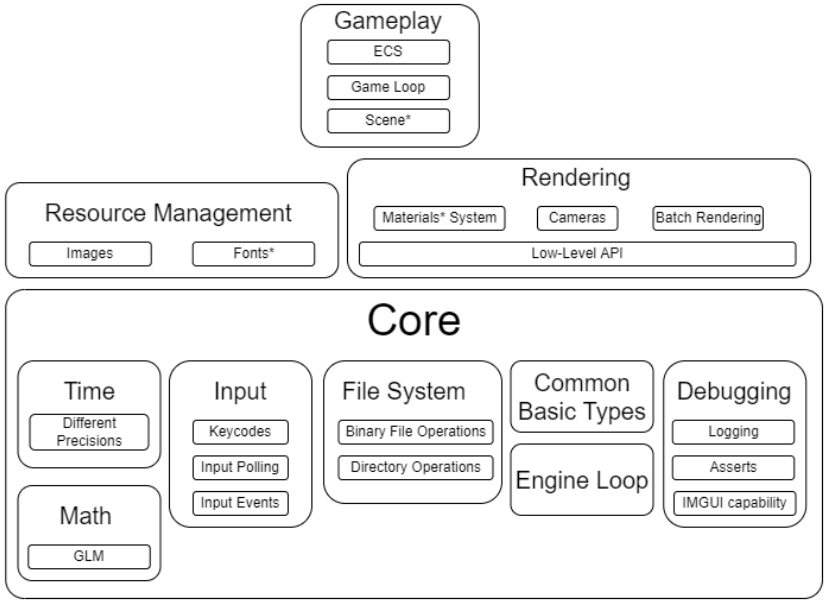
- Platform Independence Layer
  - SDL2 or GLFW (Window, Audio, Input)
  - Glad (OpenGL Loader)
- Core
  - Window Input & Events (Keyboard, Mouse, Joystick?)
  - File System
  - Hi-Res Timer
  - Logging
  - Random number generator
  - Asserts
  - Profiling & Instrumentation
  - Memory Allocators (Stack/Frame/Pool Allocators)
  - Fast String
  - Engine/Game Loop

    - Threading
- 3rd Party
    - GLM (Math Library)
    - EASTL (Data Structures & Algorithms)
    - Dear IMGUI (GUI Debugging Tools)
- Resource Management
    - Images/Sprites
    - SpriteSheets
    - Fonts
    - World/Level
    - Audio (OpenAL?, SDL2)
    - Asset Streaming
- Rendering
    - Rendering API Wrapper
    - 2D Renderer
    - Cameras & Viewports
    - Basic Primitive/Sprite Quad Submision
    - Render Layers
    - Sprite Atlases
    - Batch Renderer

Tras iterar sobre el diseño, se redujo a las funcionalidades básica para ajustarlo mejor a las horas previstas de trabajo y a la incertidumbre que había sobre dichas horas

Finalmente tras otra iteración se llego a un diseño más sencillo 4.1 que se podía empezar a implementar y el cual se podría expandir fácilmente con las funcionalidades más avanzadas de los diseños anteriores.

Figura 4.1: Diseño inicial del prototipo del motor



4.4. Sistema de Builds

Para el proceso de compilación, debido a que se uso **Visual Studio 2019** como IDE, se uso **MSBuild**[58]. A lo largo del desarrollo se empezó a usar **CMake**[59] para crear los archivos de build para **MSVC**[60]. Junto a esto se empezó también a usar **Visual Studio Code** debido a que el proceso de configuración con CMake resultó más sencillo.

CMake es uno de los programas más populares para la creación de archivos de build. Es multiplataforma y permite crear archivos para una gran cantidad de compiladores y plataformas.

Una gran desventaja que tiene es la curva de aprendizaje y la escasez de documentación. Aun así se acabo usando por su popularidad.

Debido a distintas desventajas sobre el workflow de visual code con cmake, principalmente problemas que se tuvieron que provocaban que herramientas como IntelliSense dejaran de funcionar constantemente. Se acabo cambiando el sistema de generación de archivos de builds a **Premake**[61] y se volvió a usar Visual Studio 2019. Se decidió usar este programa por la facilidad de

uso y porque por defecto generaba un proyecto más organizado para Visual Studio 2019.

Cabe destacar que inicialmente se estuvo analizando la opción de crear el motor como una DLL que el juego cargara. El usar un DLL podría causar perdidas de rendimiento debido a restricciones que no permiten realizar ciertas optimizaciones[62]. Finalmente se decidió crear el motor como una librería estática por simplicidad **RF1**.

En el proceso de análisis se barajo la posibilidad de crear el motor como ejecutable y crear el código del juego como un DLL. Este diseño permitiría hacer modificaciones al código del juego en tiempo de ejecución.[63]

4.5. Proceso Primera Implementación

4.5.1. Core

Se empezó implementando la capa de más bajo nivel llamada Core, esta capa se encargaría de crear la ventana con un contexto de OpenGL y de responder y procesar eventos de input de windows. uno de las primeros módulos de bajo nivel que se implementaron fue el modulo de Logging para facilitar el hacer debugging en el proceso de desarrollo.

Esta capa también se encarga de abstraer y definir los tipos de datos primitivos, definir contenedores de datos y crear macros para poder usar asserts y loggear solo en las builds de desarrollo.

4.5.2. Renderizado

Una vez se realizo la primera pasada de sistemas de bajo nivel generales, se empezó a desarrollar el sistema de renderizado, el proceso de implementación fue altamente iterativo.

Primero se implemento el renderizado de un simple triangulo de color solido, tras esto se avanzo a un Quad y posteriormente se le añadió una textura. Un quad con textura es la manera más básica de renderizar un

sprite que se usara en el motor.

Posteriormente paso a paso, se empezó a abstraer la API de OpenGL ya que permitiría usar otras APIs gráficas haciendo que implementen la misma interfaz compartida, cabe destacar que esto no entra en el alcance del proyecto y es un problema complejo de resolver correctamente por distintas razones que se explican en la sección del diseño final.

En este punto del desarrollo la implementación del renderizado seguía siendo muy básica solo pudiendo renderizar un solo quad con una sola textura individualmente. Esto se mejoro para que soportara múltiples sprites al mismo tiempo pero aun así estos sprites estaban predefinidos internamente en el motor, estaban siendo renderizados individualmente y el cliente no tenia una API para poder crearlos y usarlos.

Aun no existe el concepto de entidad u objeto que se pueda renderizar, pero con la finalidad de testear como ser haría esto en la practica se intento falsear todo ese sistema de entidades. Esto por suerte fue relativamente sencillo ya que el sistema de renderizado a grandes rasgos sigue los siguientes pasos:

1. Coger datos sobre los sprites que existen el mundo del juego como su posición y el ID del recurso que representa el sprite.
2. Procesar los datos de estos sprites y preparar llamadas para renderizarlos.
3. Hacer las llamadas a la API de renderizado.

Esto significa que para falsear la existencia de un sistema de entidades se crearon unos datos de entrada predefinidos que aunque no tuvieran el mismo formato exacto al que tendrían con el sistema **ECS** (debido a que aun no se conoce del todo como se implementara este sistema), se parecerían lo suficiente.

4.5.3. Entidades

A continuación, se implemento el sistema base de gestión de entidades y se modifico el sistema de renderizado para que iterara sobre las entidades

que fueran renderizables, junto a esto también se crearon un sistema que añadieran lógica a las entidades para testear las funcionalidades del sistema de entidades.

En este punto aun se sigue pudiendo solo renderizar un quad de tamaño fijo con texturas predefinidas dentro del código interno del motor y las entidades solo pueden ser creadas internamente.

El siguiente paso fue definir e implementar una API para que el cliente-juego pueda crear entidades. Para esto se creo un modelo base de como se querría que fuera la API a nivel general y se implemento usando este modelo de guía.

4.5.4. Recursos

El ultimo sistema por implementarse fue el sistema de gestión de recursos, con este sistema el usuario ya podría cargar las imágenes que considere y crear sprites de estas imágenes para usarlos en las entidades.

4.5.5. Proceso de Iteración

Hay que tener en cuenta que a lo largo del desarrollo de cada una de las partes, estas no se implementaron de manera isolada, Al implementarse cada una de las partes se realizaron modificaciones en otras para que se conectaran mejor entre si.

También cabe destacar que no se implementaron exactamente en ese orden, se creo un prototipo en el que se pudiera iniciar el motor y mostrar sprites por pantalla de la manera más básica posible y posteriormente se fueron segregando y organizando las funcionalidades en dichas partes.

Después de esta primera pasada de los sistemas generales del motor, se analizo el rendimiento del motor y se intento crear pequeños juegos/simulaciones, a la hora de intentar usar el propio y en concreto el sistema de entidades se encontraron múltiples problemas e inconvenientes que se fueron solucionando y de los que se hablara posteriormente en el diseño final.

4.6. Diseño Final

4.7. Core

Esta capa contiene todas las funcionalidades de bajo nivel del motor las cuales son usadas por capas superiores. Esta capa también se encarga de abstraer las funcionalidades especificas de cada plataforma, como por ejemplo el leer archivos o el mostrar texto por una consola, los tipos de datos primitivos, etc...

4.7.1. File System

El sistema de archivos se encarga de proporcionar una API sencilla al resto del motor para leer y escribir archivos, principalmente es un wrapper para la API especifica de la plataforma en la que se use el motor, en el caso de este proyecto esta implementada usando la librería estándar de C++.

En el motor este sistema esta implementado de la manera más simple posible, solo ofreciendo las funcionalidades de:

- Leer y escribir archivos binarios.
- Leer archivos de texto.
- Wrappers para los streams de archivos.

Esto se debe a que esas funcionalidades fueron las mínimas necesarias que acabo usando el resto del motor pero en un motor más grande, este sistema también se encargaría de:

- Manipular directorios
- Manipular rutas de archivos/directorios
- Ver y modificar metadata sobre los archivos (Es de solo lectura?, ultima fecha de modificación)

4.7.2. Input System

Sistema encargado de procesar los eventos de input de la plataforma, la versión final de este sistema esta implementada como un propio sistema dentro del modelo ECS.

Este sistema procesa los eventos de input y escribe los resultados sobre el estado de los botones en un componente global de input para que el resto de sistemas puedan leerlo.

Si se quiere hacer que por ejemplo en un juego el jugador dispare un un arma cuando se presione un botón, existen principalmente dos maneras de procesar input para conseguir esto:

- Preguntando cada fotograma al sistema de input si se ha presionado dicho botón en ese fotograma, esta técnica tiene el nombre de polling.
- Suscribiéndose a un evento que se dispare cuando el botón sea presionado, indicando que es lo que se quiere que ocurra cuando esto suceda. Este método generalmente se considera de más alto nivel

Aunque inicialmente se implemento un prototipo sencillo por eventos, finalmente se decidió por simplicidad solo ofrecer un sistema basado en polling.

4.7.3. Logging

Sistema encargado de ofrecer al motor una API que sea agnóstica en cuanto a la plataforma para loggear mensajes. Ofrece las siguientes funcionalidades:

- Tipos de prioridad/verbosidad para los mensajes (Commentario, Info, Aviso, Error, Critico)
- Canales de mensajes (Core, Gameplay, etc...)
- Filtración de mensajes por prioridad/verbosidad y canal

- Referencia al archivo/función/linea en la que se hizo el log

Aunque el sistema de logging ofrezca distintas funcionalidades como los canales y la verbosidad, en la practica debido al alcance del motor la gran mayoría no se han usado y algunas funcionalidades muy básicas como el poder ver el tiempo en el que apareció el mensaje no fueron implementadas debido a que no se consideraba que, al menos para este proyecto, fueran lo suficientemente necesarias.

4.7.4. Matemáticas

Para todas las necesidades de elementos y operaciones matemáticas del motor se uso la librería **GLM**, por esa razón, el modulo **Math** del motor se usa solo para definir un alias a los tipos de datos más usados por el motor como por ejemplo vectores de floats, matrices y quaternions.

4.7.5. Platform

Este modulo se encarga de configurar y crear la ventana del motor junto con el contexto OpenGL, principalmente esta capa es un wrapper de GLFW.

4.7.6. Profiling

Wrapper que se encarga de definir las funciones usadas para hacer profilling del motor

4.7.7. Random

Contiene utilidades para generar números aleatorios como floats o enteros para el motor.

4.7.8. Time

Contiene el sistema que se encarga de avanzar el tiempo del motor y mantener información sobre cuanto tiempo y fotogramas han pasado desde la ejecución del motor.

4.7.9. Types

Este modulo contiene wrappers para todos los tipos de datos primitivos como ints o floats de distintos tamaños, contenedores de datos como vectores o mapas, strings y punteros inteligentes.

4.7.10. Application y API

La ultima parte de la capa **Core** seria la parte de aplicación, este modulo sirve como punto de entrada y configuración del motor.

Se encarga de inicializar todos los sistemas necesarios en orden, entrar en el bucle del motor y liberar todos los recursos antes de cerrar la aplicación.

Este modulo también define los tipos de datos que el juego/aplicación que use el motor podrá/tendrá que definir como por ejemplo datos sobre la ventana y referencias a funciones para la creación del mundo del juego.

4.8. Render

Este modulo contiene todo los elementos y funcionalidades necesarias para el renderizado de sprites, esta dividido en dos submódulos, **Render API** y **Rendering System**.

4.8.1. Render API

Render API es una abstración/wrapper de las funcionalidades de bajo nivel de renderizado. Este submódulo abstrae la API especifica de renderizado que se use en el motor, en este caso OpenGL.

A nivel práctico en el contexto de este motor la existencia de esta abstracción, como la del resto de abstracciones de la capa core, no ofrece ningún beneficio ya que debido al alcance del motor no se iba a dar soporte completo a distintas APIs gráficas.

4.8.2. Rendering System

Rendering System es el sistema dentro del modelo ECS que se encarga de renderizar todas las entidades que sean renderizables, este realiza el siguiente proceso dividido en dos partes en cada fotograma:

1. Agrupar todas las entidades por el ID de sprite que estén usando
2. Por cada uno de los grupos creados, configurar un batch y renderizarlo

De esta manera se crea y configura un batch por cada tipo de sprite y se reducen drásticamente el numero de **drawcalls** lo cual da un rendimiento mucho mayor.

Este diseño no es el más eficiente debido a que se están creando los grupos de entidades por id de sprite cada fotograma cuando estos grupos casi nunca o nunca cambian.

Analizando que es lo que puede producir cambios en los grupos, se ve que estos solo cambian si se crean o destruyen entidades.

Sabiendo esto, podríamos actualizar solo los grupos cuando se creen o destruyan entidades lo cual seria en teoría mucho más eficiente.

Esto no se implemento en el motor final debido a que el actualizar los grupos no era un gran cuello de botella en comparación con otras operaciones realizadas por el motor.

Capas

Para implementar que los sprites estén en distintas capas se acabo usando la coordenada Z del vector de posición y dejando que OpenGL organice los sprites por profundidad.

Otra implementación que se considero se basaba en renderizar manualmente los sprites por capas, empezando por la capa más "lejana.ª la cámara. Esto haría que las capas sucesivas se superponieran sobre las anteriores.

Esto significaría que las entidades estarían agrupadas primero por capa y después por id de sprite. A la hora de renderizar por cada una de las capas seleccionaríamos cada uno de los grupos y crearíamos **Batches**.

Finalmente se decidió usar la implementación usando la coordenada Z debido por su simplicidad.

Debugging del sistema de rendering

Durante el desarrollo del sistema de renderizado, varias de las primeras implementaciones no funcionaban por distintos motivos, al principio era suficiente usar el debugger incluido con **Visual Studio 2019** para encontrar los fallos pero llego un punto en el que esto no era suficiente.

OpenGL ofrece un callback con el que se pueden mostrar por consola distintos mensajes sobre el estado de OpenGL, esto fue útil para ciertas cuestiones pero pero aun asi no era información suficiente y el proceso de debugging con estos mensajes era lento.

Por estas razones, al final se acabo usando una herramienta llamada **RenderDoc**, esta herramienta permite capturar fotogramas de la aplicación y ver el estado del contexto de OpenGL en estos mismos.

Esta herramienta fue extremadamente útil a la hora de desarrollar el **Batch Renderer** ya que permitió encontrar varios fallos que hacían que el renderer no mostrara ninguno de los objetos que tenia que renderizar.

Rendering de información de debug

Este sistema también se encarga de crear una ventana de debug usando IMGUI en la que se muestra información sobre la duración del fotograma o en numero de batches en este mismo.

4.9. Resources

El sistema de gestión de recursos se encarga de cargar y preparar todos los recursos que use el motor, que en el contexto de este motor serian Imágenes y Sprites, cumpliendo los requisitos **RF3** y **RF4**.

Imágenes: Son archivos de imagen (png, jpg, etc...) que el motor usa para generar sprites.

Sprites: Recurso generado por el motor que contiene parte de la información necesaria para renderizar una imagen. **RF4**,**RF5**

El diseño general del sistema de gestión de recursos se basa en el uso de **IDs** llamados **Handles**.

Cuando otro sistema o el juego quiere cargar un recurso, este recibe un handle del recurso en cuestión y puede usar este handle para liberar el recurso posteriormente o para acceder a él justo en el momento en el que sea necesario. El sistema externo no tiene una referencia directa al recurso que ha cargado.

El propio sistema de recursos es el propietario de todos los datos de los recursos del motor y se encarga administrar el ciclo de vida de estos mismos.

Un diseño alternativo que se planteo se basaba en crear objetos que representaran cada recurso. De esta manera, cuando se quisiera cargar una recurso se crearía en memoria un nuevo objeto del tipo del recurso y se accedería a él mediante su puntero.

Este diseño tiene la desventaja de que el sistema que cargue el recurso tiene que mantener el puntero a este mismo. Esto provoca problemas si el recurso se comparte entre varios sistemas y que no hay un propietario claro.

No se puede saber cual de los sistemas es el encargado de liberar el recurso. Esto se puede arreglar usando punteros inteligentes. Esto conlleva su coste y existe el problema de que un recurso se puede borrar en cualquier punto de la ejecución.

Por simplicidad se acabo eligiendo el diseño basado en IDs/Handles.

Al iniciarse el sistema de recursos. Se reserva en memoria un array de punteros con tamaño igual al numero máximo de recursos de ese tipo que van a existir en el juego y se generan los posibles handles. Estos handles son los índices que se usarían en el array del tipo de recurso para acceder al recurso en concreto.

4.10. Physics

Este sistema mayormente es un wrapper que integra la librería **Box2D** con el sistema de entidades del motor, se encarga de convertir las posiciones del mundo de físicas a posiciones del mundo del juego.

4.11. Entities

El sistema de entidades es uno de los sistemas principales del motor. Este sistema está basado en diseños ECS. Las partes principales de este sistema son:

- Entidades
- Componentes
- Componentes Singleton/Globales
- Sistemas
- Vistas de Entidades

A continuación se va a explicar cada una de estas partes y el diseño global del sistema de entidades.

4.11.1. Entidades o EntityID

Una entidad en este sistema es simplemente un ID que se usa para acceder a los componentes de dicha entidad, existen implementaciones que añaden información extra a una entidad pero en el contexto de este motor eso no se vio necesario.

Se conoce que un problema que puede provocar la implementación actual es el siguiente, si se guardan referencias a una entidad en algún componente, por ejemplo, si se crea un componente global para guardar el ID de la entidad que representa al jugador principal, un sistema puede borrar esa entidad y volver a crear una nueva al instante usando ese mismo ID y no habría una manera sencilla de detectar que esa entidad ya no es el jugador principal que pensábamos que era aunque esté usando el mismo ID ahora.

Esto tiene un arreglo sencillo que es añadir a ese ID un contador que subiría cada vez que se crea una entidad nueva con ese ID, de esta manera para detectar que una entidad es la misma tendríamos que comprobar su ID y contador. Este método generalmente es conocido por el nombre de **Índices Generacionales**.

Este arreglo no se implementó ya que se consideró que no era de alta prioridad teniendo en cuenta los casos de uso del motor y su alcance.

Cabe destacar, que aunque exista la opción de crear y borrar entidades en mitad del fotograma, la implementación de esta funcionalidad es extremadamente simple y tiene muchos problemas relativamente graves.

Los problemas principalmente emanan de la simpleza de las de entidades. El crear o destruir entidades o componentes incluso en distintos puntos del fotograma provoca que las vistas de las entidades cambien. Si un sistema hace esto múltiples veces mientras está iterando sobre la propia vista, provocará fallos en el sistema.

Una solución que se planteó fue diseñar las vistas de tal manera que administren esta creación y destrucción de entidades y respondan adecuadamente a ello aunque se haga en mitad de iteración.

Otra posible solución más sencilla sería retrasar la creación y destrucción de entidades hasta un punto específico en el fotograma, por ejemplo, crear un

sistema A que indique que quiere crear una entidad, un sistema B encargado de crear todas las entidades que ha indicado el sistema A y un sistema C que configura y usa las entidades ya creadas.

4.11.2. Componentes

Los componentes son exclusivamente contenedores de datos aunque es posible que tengan métodos con la finalidad de hacer que acceder a sus datos sea más sencillo.

Los componentes están asociados a una entidad y están guardados en arrays de componentes, estos arrays están diseñados de tal manera que los componentes están almacenados uno detrás de otro en memoria. generalmente este diseño es conocido con el nombre de **sparse set**.

Para acceder a un componente teniendo el id de la entidad, primero usamos este id como indice en un array llamado **sparse array**, este array nos dará el indice que tendremos que usar en otro array que llamaremos **dense array** para conseguir el componente que queremos. este ultimo es el array que siempre tiene todos los componentes alineados consecutivamente en memoria.

En la implementación final el sparse array se acabo nombrando entityToIndex ya que describe mejor su funcionalidad y se uso también un array que guarda la relación inversa de indexToEntity. Este último se usa unicamente para saber a que entidad pertenece el último elemento del array denso, ya que al eliminar un elemento de este mismo, se reemplaza por el último elemento del array para mantener la continuidad en memoria.

4.11.3. Componentes Singleton o Globales

Son contenedores de datos globales, solo existe un componente global de cada tipo, son extremadamente útiles para muchas cosas como guardar todos los datos de input o guardar los datos de puntuación de la partida.

4.11.4. Sistemas

Son contenedores de lógica, se encargan de definir las operaciones que se harán sobre los datos que existen en distintos componentes, tienen una fase de inicialización, destrucción y una función de actualización que se ejecuta cada fotograma, los sistemas se ejecutan en el orden en el que se registran.

Estos sistemas pueden tanto iterar sobre múltiples entidades con distintos tipos de componentes mediante vistas como solo cambiar un componente global [**RF11**,**RF8**]. En esencia el uso y creación de sistemas ayuda en la separación de lógica del juego, uno podría crear un juego cuya lógica este dentro de un solo sistema, pero en muchos casos seria más conveniente separar ese gran sistema en sistemas más pequeños siempre que tenga sentido.

Se decidió nombrar a estos contenedores de lógica sistemas porque era el termino más usado, pero en realidad de manera más general se pueden considerar pasadas en el pipeline de procesamiento de datos del fotograma.

Existen casos en los que considerarlos y tratarlos como sistemas tradicionales dificulta la situación. Supongamos que tenemos un sistema A y un sistema B que lógicamente tiene sentido que estén separados pero existen puntos en los que estos sistemas tienen dependencias de datos entre ellos. En un punto el sistema A necesita datos que hayan sido procesados por el sistema B y viceversa. La manera de solucionar este problema sin fusionar los dos sistemas seria dividirlos en más sistemas de modo que:

1. El sistema A.1 se ejecuta hasta necesitar datos que el sistema B.1 aun no ha procesado.
2. El sistema B.1 procesa los datos hasta necesitar datos de A.2
3. El sistema A.2 se ejecuta con los nuevos datos que B ha procesado y genera los datos para B.2
4. El sistema B.2 se ejecuta con los datos que A.2 ha procesado

Esta solo es una posible combinación dependiendo del orden de dependencias que tengan estos sistemas. En el caso mostrado, el concepto de sistema deja de tener sentido. si se supone que el sistema A es el sistema de sonido

de un juego, no tiene sentido el decir que existen dos sistemas de sonido, seguramente muchas funciones auxiliares del sistema de sonido 1 se usen en el sistema de sonido 2. pero si que todo encaja mejor en el modelo si se considera que son dos pasadas del mismo sistema y se diseña el modelo de entidades teniendo eso en cuenta.

En la implementación final, esta última parte no se desarrollo debido al alcance del motor. Se implemento la versión básica de la que se hablo al principio y se intento dejar bastante libertad sobre lo que se puede hacer con los sistemas.

4.11.5. Vistas de Entidades

En el diseño inicial, los sistemas tenían que indicar los componentes que tenían que tener las entidades sobre las que quisieran iterar y el propio sistema mantendría una lista con las entidades que tienen esos componentes. Esto tiene la limitación muy grande de que cada sistema solo podía iterar sobre un tipo de entidades. Si un sistema quería iterar sobre entidades con un componente A primero, y después necesitara iterar sobre entidades con el componente B, no tenia ninguna manera de hacerlo.

Para solucionar este problema se implemento el concepto de **vistas**, el cual es bastante común en implementaciones ECS.

Una vista tiene la misma funcionalidad de ofrecer la capacidad de iterar sobre entidades con unos componentes específicos. Al separar esta funcionalidad/capacidad fuera de los propios sistemas, se tiene la ventaja de poder crear múltiples vistas que se usen en un solo sistema.

En el diseño inicial de las vistas, si dos sistemas querían iterar sobre los mismos tipos de componentes, los dos tenían que mantener una lista de entidades y actualizarla en todo momento. Con la implementación final, las vistas se pueden compartir entre sistemas. Cada vista tiene un ID el cual se le asigna dependiendo de los tipos de componentes que se usaron para crear esa vista. De esa manera cuando un sistema necesita acceder a una vista, primero se comprueba si esta ya existe y si es así se devuelve y no se crea una nueva.

4.12. API para el Juego

El motor ofrece una API al juego con operaciones de alto nivel para facilitar el proceso de desarrollo. A continuación se van a mostrar todas las partes de esta API y como se usan para crear un proyecto de ejemplo:

4.12.1. Diseño General

Toda la API esta diseñada de tal manera que el juego para configurar cuestiones del motor, tiene que crear una estructura que contendrá dicha configuración llamada **GameInitData** y pasársela al motor al iniciarlo.

Aunque este fue el diseño final, se plantearon dos diseños mas que podrían haber sido perfectamente válidos también.

Uno de los diseños estaba basado en la creación de una clase abstracta de la cual el juego tendría que heredar y sobrescribir distintos métodos para cargar recursos, crear entidades, configurar la ventana etc.

Otro diseño que se estudio estaba basado en la creación de un objeto que contendría toda la API como métodos y cada método devolvería el propio objeto, de esta manera para crear el juego se cargarían recursos y registrarían sistemas haciendo llamadas continuas a este objeto. Este diseño generalmente se conoce por el nombre de **patrón builder**.

Se acabo implementando el diseño en el que el juego proporcionaba dicha estructura de datos por simplicidad y con la finalidad de delimitar de una manera más clara cuales son los datos que el juego puede modificar.

4.12.2. Configuración de Ventana

El juego puede ajustar parámetros de la ventana modificando la estructura llamada **WindowDescription**. **RF2**

4.12.3. Carga de Recursos

La carga de los recursos necesarios para el motor se realiza al iniciarse este mismo, para indicar que recursos se quieren cargar, el juego tiene que definir una función con los parámetros que se le indiquen, dentro de esta función el juego podrá hacer llamadas al sistema de recursos.

Posteriormente el juego tendrá que pasar dicha función como un dato dentro de la estructura que se le proporcionara al motor.

4.12.4. Componentes y Sistemas

El juego puede crear nuevos tipos de componentes [**RF7**,**RF9**] simplemente creando estructuras de datos y para crear sistemas [**RF10**] lo único que se tiene que hacer es heredar de la clase abstracta **system** que se ofrece.

Para registrar nuevos tipos de componentes y sistemas, el juego tiene que seguir los mismos pasos que se hicieron para la carga de recursos, pero esta vez la definición de la función tiene un parámetro que es una referencia al sistema de entidades del juego, teniendo esta referencia el usuario puede registrar todos los componentes y sistemas previamente creados.

4.12.5. Creación del Mundo

Para crear el mundo se usa el mismo proceso que para la carga de recursos, componentes y sistemas, pero esta vez en la función que se definirá se harán llamadas que crearan entidades y les añadirán componentes.

4.13. Optimización

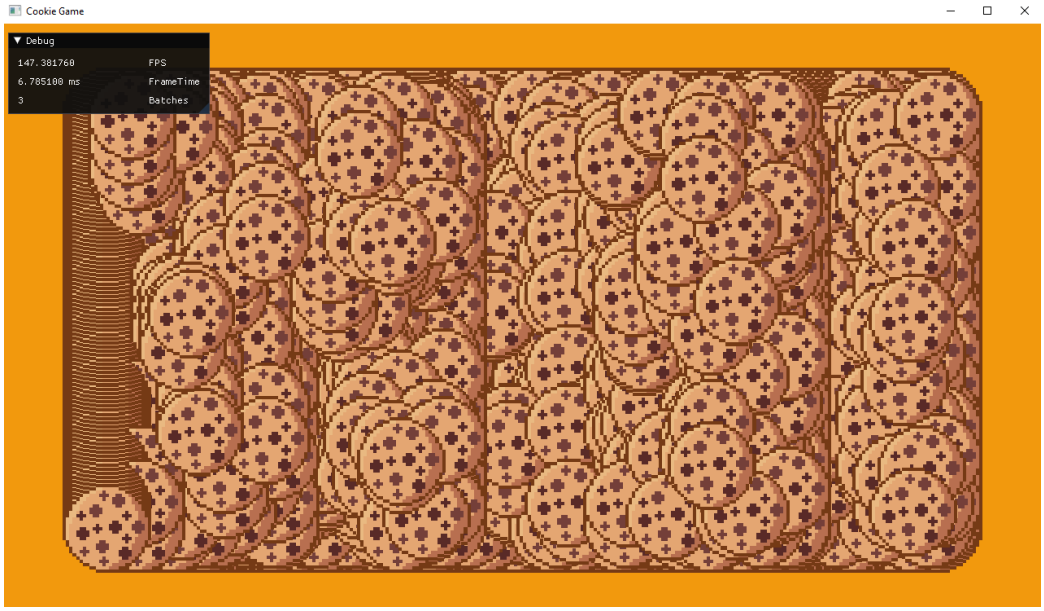
Una vez se implementaron mayormente las versiones finales de cada modulo. Se empezó a hacer pruebas para medir el rendimiento.

Estos tests fueron realizados en un ordenador con las siguientes especificaciones:

- CPU: Ryzen 7 2700x
- GPU: RTX 2070 1710 MHz 8 GB GDDR6
- RAM: 2x8 GB 2934 MT/s
- HDD: 2 TB 5400 RPM

Para esto se usó el proyecto Cookie Game, Todas las pruebas se realizaron con 10.000 entidades. Estas entidades estaban influenciadas por un sistema que permitía que el usuario las moviera con las teclas WASD y un sistema que modificaba el parámetro de rotación del componente transform de las entidades.

Figura 4.2: Proyecto Cookie Game mostrando 10.000 entidades

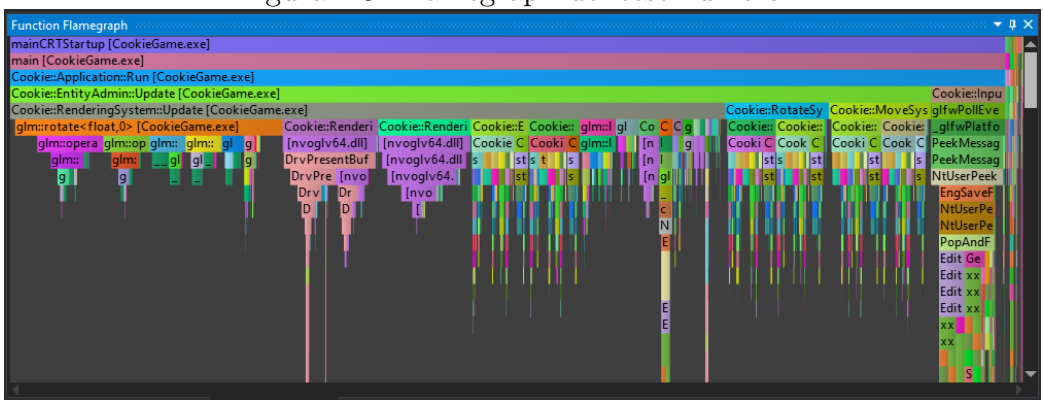


Cabe destacar que este parámetro, aunque se modificara, se dejo de usar porque se decidió que no seria necesario para el motor añadir la posibilidad de rotar sprites y el soportarlo tendría un coste de rendimiento aunque este no se use.

Todos estos tests se hicieron sin optimizaciones y con información de debug completa.

4.13.1. Test 1

Figura 4.3: Flamegraph del test número 1



El primer test que se hizo mostró lo que se esperaba que era que lo más lento de todo el sistema era el renderizado, esto se debe a que en este punto aun no se implemento el batch renderer y se estaba renderizando individualmente cada Quad.

En este Test de media cada fotograma tenia una duración de **180 ms** de los cuales el renderizado ocupaba un 70 %.

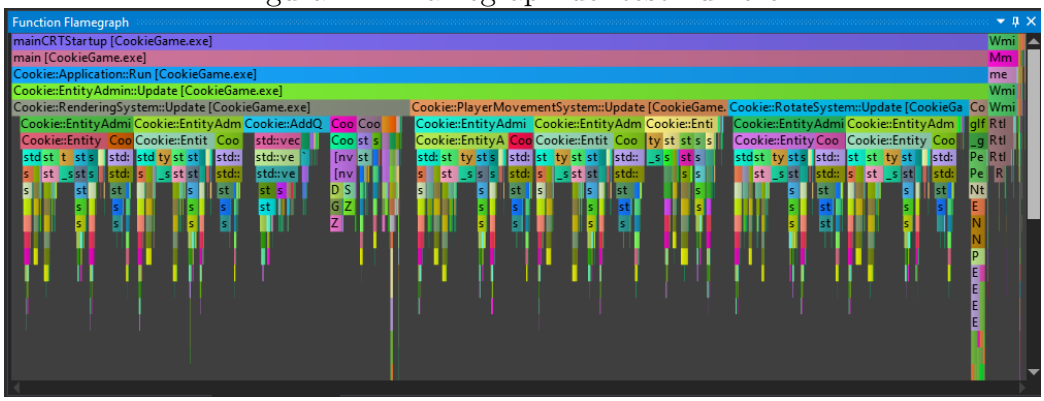
4.13.2. Test 2

El segundo test se hizo ya con una de las primeras implementaciones del batch renderer y hubo una mejora considerable, el tiempo por fotograma bajo a **90 ms** con el renderizado ocupando el 39 % y los dos sistemas implementados por el usuario 55 %.

En este test se observo que la gran mayoría del tiempo (hasta un 95 %) que gasta un sistema se utiliza solo para acceder a los propios componentes y no para procesarlos.

Esto se debe a que la implementación ECS que había en aquel momento solo ofrecía funciones para acceder al componente especifico de una entidad. Esto era un problema ya que para poder hacer esto la función tenia que:

Figura 4.4: Flamegraph del test número 2



1. Sacar el tipo del componente.
2. Calcular el hashcode de este tipo.
3. Usar el hashcode para sacar el array de componentes genérico del *unordered_map*.
4. Usar dynamic_cast para castear el array al tipo del componente concreto
5. Finalmente acceder al array y devolver el componente

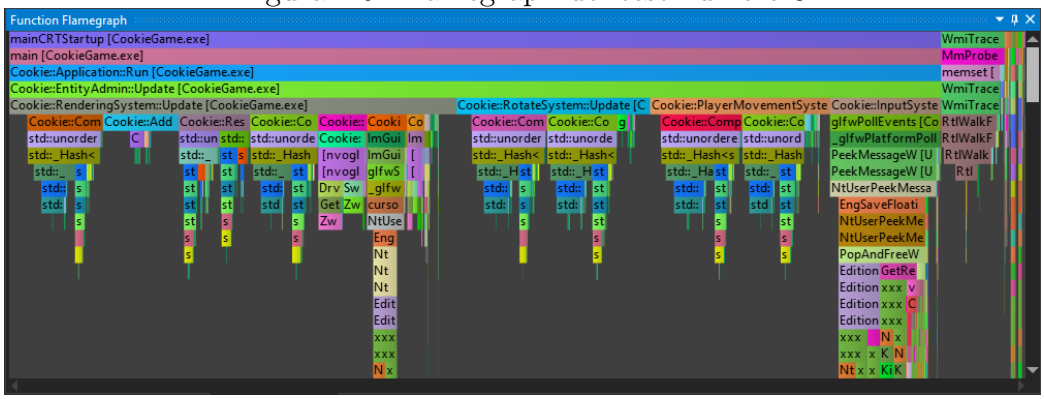
Este proceso se realizaba cada vez que se quería acceder a cualquier componente de cualquier entidad. Lo cual provoca grandes perdidas de rendimiento en realizar operaciones que no son necesarias.

Por suerte el arreglo es sencillo, lo que se hizo fue dar acceso directo a que se pudieran conseguir el array directamente. De esta manera los primeros 4 pasos solo se realizarían una vez por cada array de componentes al que se acceda en cada sistema.

4.13.3. Test 3

El test número 3 se realizo con los cambios previamente discutidos los cuales hicieron que el tiempo por fotograma bajara a **25 ms**.

Figura 4.5: Flamegraph del test número 3



En este test aun se estaba viendo que la gran mayoría del tiempo los sistemas estaban calculando cuestiones internas de los *unorderer maps* que estaba usando la implementación de *component array*. Estos *unordered maps* se estaban usando para mantener una relación entre una entidad y el indice dentro del array de componentes donde estaría su componente.

Se llego a la conclusión de que no era necesario tener estos *unordered maps* para mantener dicha relación entre entidades e índices.

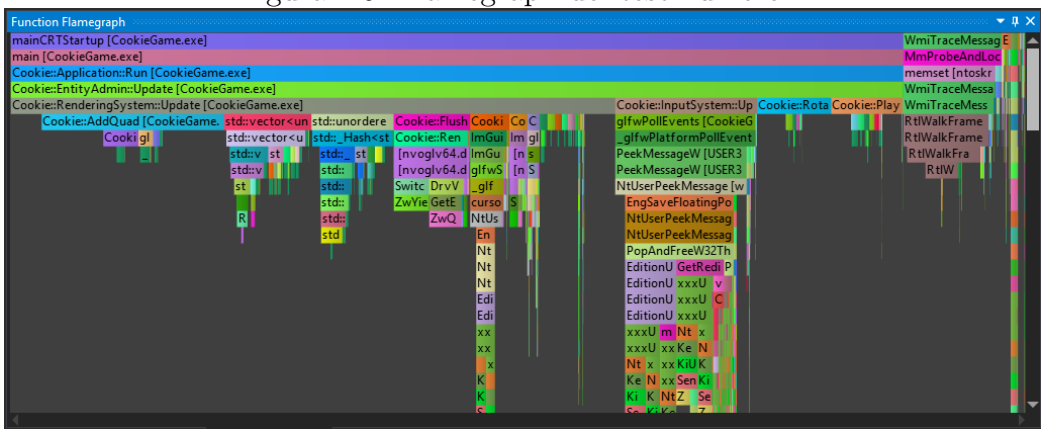
Se decidió usar un simple array el cual tendría un tamaño igual al número máximo de entidades. Lo que anteriormente seria la *key* del unordered map ahora se convertiría en el indice usado para acceder al array, y el valor se convertiría en el propio valor que tendría el array en esa posición.

Esto hace que los arrays de componentes ocupen la máxima cantidad de memoria desde el principio. Aun así en este caso es preferible el aumento de rendimiento que se gana con los accesos.

4.13.4. Test 4

En el 4 test habiendo aplicado los últimos cambios se bajo a un tiempo por fotograma de **9 ms**. Cabe destacar que para este tests se arreglaron ciertos fallos con el batch rendering que hacia que no funcionara correctamente con múltiples tipos de sprites lo que aumento çonsiderablemente.[el] tiempo de

Figura 4.6: Flamegraph del test número 4



ejecución del sistema.

El batch renderer se podría optimizar más aun ya que en la implementación final actualiza los grupos de sprites en cada fotograma cuando esto solo es necesario en si cuando se crea o destruye alguna entidad.

Una cuestión a tener en cuenta es que aunque los arrays de componentes tengan estos componentes alineados en memoria uno de tras de otro. Los accesos a estos componentes no son nada consecutivos. Esto se debe a que se accede al array usando el ID de las entidades. Esto hace que haya saltos de memoria en los accesos.

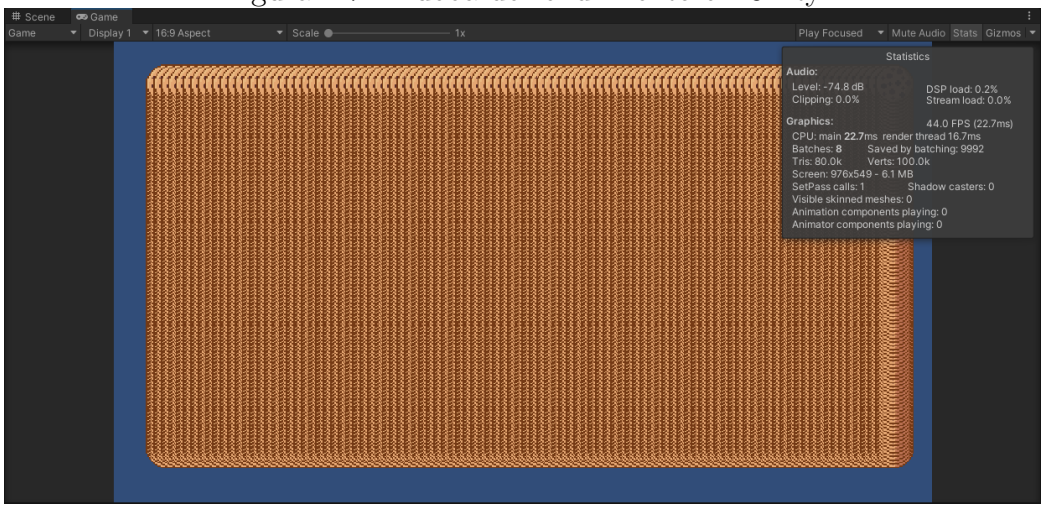
4.13.5. Comparativa con Unity

Se realizo una prueba en la que se implementaba el mismo test usando la versión base de Unity (Sin ECS).

Cabe destacar que la implementación del test se realizo de la manera más directa y sencilla que permite Unity. Existen muchas maneras de optimizar este test teniendo en cuenta las características y el funcionamiento de Unity.

En el test se registro un tiempo por fotograma de 21 ms de media, bajando hasta 17-18 ms cuando las entidades no estaban siendo movidas por el input

Figura 4.7: Prueba de rendimiento en Unity



del usuario.

Capítulo 5

Validación

5.1. Resultado Final

El motor final es una librería estática que tiene que ser añadida, junto con todas sus dependencias, al proceso de compilación del juego que se desarrolle.

El motor ofrece las funcionalidades para crear una ventana con distintos parametros, cargar imágenes, crear entidades en el juego con distintos atributos como componentes y añadirles lógica mediante sistemas.

5.2. Ejemplos de uso

A continuación se mostraran distintos proyectos que se crearon para testear las funcionalidades del motor.

5.2.1. Cookie Boids

Este proyecto es una implementación de una simulación de boids. Se decidió crear este proyecto de prueba debido a que se teorizo que seria un caso de prueba que mostraría los problemas de rendimiento que pudieran existir.

Se define un componente personalizado llamado *BoidComponent* que contiene la velocidad del boid.

Junto a esto se crean dos sistemas. Un sistema llamado *CameraSystem* el cual se encarga de controlar el movimiento de la camara y un sistema llamado *BoidsSystem* el cual contiene toda la lógica para la simulación.

El motor consigue mantener aproximadamente 57 fps estables en el sistema en el que se hicieron pruebas con 2500 boids en la simulación. Esto fue sorprendente porque en la implementación de la simulación. Cada uno de los 2500 boids tiene que comprobar su estado con respecto a los 2499 restantes cada fotograma.

5.2.2. Cookie Physics

Este proyecto se creo con la finalidad de comprobar las funcionalidades del sistema de físicas Box2D que se integro. En este proyecto solo se consiguieron crear de manera estable 1000 entidades con colliders circulares.

Aunque no se hizo un análisis exhaustivo sobre el porque de esto. Se sospecha que se debe a que por como esta diseñada la escena. Existen demasiados objetos de físicas dinámicos colisionando al mismo tiempo en un espacio relativamente pequeño

5.2.3. Cookie Survival

Este proyecto es una pequeña aproximación a un juego con vista top-down existente llamado **Vampire Survivors**[64]. Se realizo para confirmar que el motor tenia las suficientes funcionalidades para realizar un juego o un minijuego.

El proyecto se basa en la existencia de un jugador representado por una galleta grande. Este jugador se puede mover usando las teclas WASD.

Existen lo que se denominan enemigos que son representados por sprites de galletas pequeñas, cubos de azucar blanco y mantequilla. Estos enemigos se mueven constantemente hacia el jugador.

Figura 5.1: Proyecto Cookie Survival con 100.000 entidades

El jugador periódicamente realiza ataques de manera automática. Existen tres tipos de ataques los cuales se van ciclando constantemente.

- Ataque Horizontal: Afecta a los enemigos que se encuentran en un rectángulo orientado horizontalmente.
- Ataque Vertical: Afecta a los enemigos que se encuentran en un rectángulo vertical cerca del jugador
- Ataque Donut: Afecta a los enemigos que se encuentran en un área con una forma semejante a un donut donde el centro es la posición del jugador.

Cuando los enemigos son atacados. Estos "mueren", se aumenta la puntuación y vuelven a aparecer más enemigos.

El proyecto se ejecuta a aproximadamente 70 fps con 100000 enemigos activos en el ordenador de desarrollo.

Este proyecto consta de los siguientes componentes y sistemas:

Componentes

- PlayerCharacterComponent: Contiene los datos necesarios para el movimiento del personaje que controla el jugador.
- EnemyComponent: Contiene los datos de los enemigos.
- AttackComponent: Contiene información sobre los ataques

Componentes Globales/Singleton

- SinglMainPlayerComponent: Este componente contiene una copia de la posición del jugador.
- ScoreSinglComponent: Contiene la información de la puntuación de la partida.

Sistemas

- PlayerMovementSystem: Contiene la lógica para el movimiento del personaje.
- CameraSystem: Encargado de controlar la cámara.
- AttackSystem: Contiene la lógica de los ataques del jugador.
- EnemySystem: Contiene la lógica de los enemigos como su IA.

Flujo de ejecución

De manera general. El flujo de ejecución de los sistemas del juego es el siguiente:

- El jugador se mueve dependiendo del input recibido.
- La cámara sigue al jugador.
- Se ejecuta la lógica de los ataques temporizados del jugador.
- Los enemigos se acercan al jugador.

Capítulo 6

Conclusiones

6.1. Logros alcanzados

En este apartado se va a evaluar el grado de cumplimiento de cada uno de los objetivos propuestos. Se analizaran los pasos que se han seguido para cumplir el objetivo y el resultado final.

6.1.1. Analizar las características generales de los motores de videojuegos más relevantes

Se realizaron múltiples análisis de material que exponía conceptos teóricos o de diseño sobre los motores de videojuegos y sobre el diseño de sistemas específicos de motor como sistemas ECS.

Junto a esto, se realizo un análisis de múltiples implementaciones de motores open source e implementaciones de sistemas ECS. En estos análisis se intento especular cuales fueron las razones que llevaron a cada diseño.

En las implementaciones se analizaron ligeramente cuestiones del uso de las funcionalidades del propio lenguaje de C++.

Se llevo a cabo un análisis sobre el diseño orientado a datos, su relación con las arquitecturas ECS y se trataron algunos de los principios que rigen

este tipo de diseño.

Se considera que este objetivo se ha cumplido satisfactoriamente.

6.1.2. Definir la especificación técnica de un motor de videojuegos basado en ECS

Partiendo de los análisis realizados, se definieron las funcionalidades que tendría el motor las cuales se fueron ajustando a lo largo del proyecto por cuestiones de tiempo y recursos.

Junto a esto se definió la arquitectura general del motor y de los distintos módulos que lo compondrían junto con las funcionalidades generales de estos módulos. No se realizaron diseños de cuestiones de bajo nivel específicas debido a la volatilidad que tendrían estos diseños. Estas cuestiones serian tratadas caso por caso a la hora de implementar el motor.

Se considera que el objetivo se ha cumplido en su mayoría dentro del contexto del proyecto.

6.1.3. Implementar un prototipo del motor de videojuegos basado en ECS

Se realizo un prototipo del motor inicial y se fue iterando sobre este con la finalidad de ir mejorando las funcionalidades existentes e implementar las funcionalidades restantes que se definieron.

Por desgracia por cuestiones de tiempo se tuvieron que quitar múltiples sistemas del motor pero aun así se consiguió implementar una versión base de los sistemas más importantes como el sistema de input, renderizado y el sistema ECS.

Se considera que el objetivo está cumplido dentro del alcance de este proyecto aunque existen una gran cantidad de mejoras posibles.

6.1.4. Desarrollar una demo en el prototipo

Se realizaron múltiples demos en la parte final del desarrollo que exponían distintas funcionalidades de este mismo como por ejemplo:

- Cookie Boids: Implementación de un algoritmo de boids dentro del motor de videojuegos.
- Cookie Physics: Demo del motor de físicas Box2D integrado en el motor.
- Cookie Survival: Demo de un pequeño videojuego en el que se implementan distintos sistemas como un sistema de movimiento, puntuación y ataques.

Se considera que este objetivo se ha cumplido.

6.1.5. Validar y evaluar las funcionalidades del proyecto

Se realizaron múltiples demos para validar que el motor tenia las suficientes funcionalidades para crear como mínimo minijuegos y para confirmar que estas funcionaban correctamente.

Junto a esto, usando las demos implementadas se realizaron análisis de rendimiento y se mejoro de distintas maneras el rendimiento del motor.

Se implemento también una demo en Unity con la finalidad de comparar el rendimiento.

Por todo esto se considera que este objetivo se cumplió satisfactoriamente. Habría sido interesante realizar un estudio de usuarios externos usando el motor para evaluar las funcionalidades.

6.2. Lecciones aprendidas

El diseño orientado a datos ha ayudado bastante en cuanto al diseño y desarrollo del motor. Aunque al principio del desarrollo no se aplicaron tanto como se hubieran tenido que aplicar este tipo de diseño por inercia a crear un diseño estandar más orientado a objetos y encapsular todo de manera estricta. Esto en casi todos los casos acabo produciendo un resultado negativo y casi todos los sistemas desarrollados de esa manera se acabaron cambiando.

Es importante tener en cuenta que cada paradigma de diseño y muchas prácticas de programación tienen sus pros y sus contras aunque en muchas fuentes solo se hable de los pros. Es conveniente analizar y entender cuales son estos pros y contras. Esto puede ser difícil en algunos casos porque los contras de una metodología pueden no ser aparentes debido a distintos factores.

En proyectos futuros se intentará refinar y se seguirá desarrollando software usando estos principios desde el inicio del proyecto.

Una de las principales ventajas que se han notado usando principios de diseño orientado a datos es la facilidad con la que se podían realizar grandes cambios en distintos sistemas como por ejemplo el sistema de renderizado. Principalmente debido a que el diseño se basaba en coger datos del mundo, procesarlos y crear drawcalls.

Junto a esto. Antes del desarrollo del proyecto el autor solo tenía conocimientos teóricos sobre motores de videojuegos. nunca llegando a implementar ninguno. A lo largo de este proyecto se han estado estudiando muchos motores, varias implementaciones distintas y se ha implementado un motor relativamente completo y sencillo.

Se ha aprendido mucho sobre todas las cuestiones que entran dentro de la creación de un motor como por ejemplo la creación de renderers, gestión de recursos, proceso de creación de builds de c++, etc... Sobretodo se han descubierto posiblemente cientos de cuestiones interesantes que se podrían estudiar e implementar en un motor.

6.3. Lineas futuras

Una de las mejoras principales seria refinar la arquitectura de ciertas partes del motor y ofrecer una API al juego que ayude con los posibles fallos que se puedan realizar como por ejemplo que se intente acceder a un componente que no se ha registrado.

Existen varias funcionalidades que no están expuestas directamente al usuario como por ejemplo el poder cambiar ciertos parámetros de la cámara o cambiar el orden de ejecución de los sistemas internos del motor.

Junto a esto, existen muchos sistemas que no se pudieron realizar en este proyecto pero que serian interesantes de estudiar e implementar en proyectos futuros como por ejemplo:

6.3.1. Threading y Concurrencia

Existen arquitecturas de motor en las que constan de un main thread, un render thread y un grupo de threads llamados worker threads que ejecutan las tareas que sean necesarias en ese momento.

6.3.2. Sistemas de Gestión de Recursos

Sistemas de gestión de recursos avanzados como el implementado en el motor Krueger Engine estudiado.

Estos sistemas funcionan como una servidor con una base de datos y se encargan de convertir todos los assets en versiones optimizadas para el motor y ofrecer estos assets a distintas máquinas de desarrollo para ejecutar el juego.

6.3.3. Herramientas

Por desgracia para este proyecto no se pudieron hacer ningún tipo de herramientas o editores pero seria muy interesante crear un editor de mundo

como una aplicación externa que se comunique con el juego mediante TCP por ejemplo.

Junto a esto se podrian crear editores para definir lógica de gameplay. Editores de materiales específicos para el motor. Un editor con un lenguaje de scripting visual propio, etc...

6.3.4. Sistemas de Entidades/GameObject Especializados

En este motor se desarrollo una implementación bastante básica de ECS. Pero en si este tipo de arquitectura es solo una de las posibles maneras de aplicar cuestiones de diseño orientado a datos. Existen diseños distintos que pueden ser interesantes de explorar y desarrollar como el modelo de Actores/Entidades ECS de Krueger Engine.

6.3.5. Renderizado

Para este motor se creo un sistema de renderizado 2D simple usando batch rendering pero aun existen muchísimas mejoras que se podrían hacer. cuestiones como spritesheets, animaciones, soporte para shaders, renderizado de modelos simples como píxel art.

Estas cuestiones que parecen simples a primera vista pero, por ejemplo, la creación de un sistema de animaciones 2D pueden llegar a ser muy interesante dependiendo del contexto.

Por ejemplo, en un motor orientado a pixel art existe la posibilidad de meter todos los sprites de la animación en una misma imagen. Pero si el motor tiene que dar soporte a sprites de 3000x3000 píxeles esto no es una opción y se tienen que buscar otras alternativas de codificar, guardar y procesar estas animaciones.

Cabe destacar que también existe todo el mundo que es el renderizado 3D tanto realista como no-fotorealista o NPR.

6.3.6. Audio

En este motor no se llego a implementar audio pero es otra funcionalidad que puede convertirse en algo extremadamente complejo e interesante.

6.3.7. Gestión de Memoria

En el proyecto no ha existido casi ningún tipo de gestión de memoria quitando la gestión básica de decidir si algo debería estar en el stack o en el heap. Pero existen muchos tipos de técnicas y arquitecturas para gestionar la memoria. Un proyecto interesante se basaría en medir y reservar toda la memoria que va a usar el motor para el juego en especifico y subdividir y jerarquizar este bloque de memoria dependiendo del uso que se le quiera dar a cada sección.

6.3.8. Conclusiones

Existe una gran cantidad de lineas futuras que se podrían seguir. todas ellas siendo muy muy densas. Cabe destacar que seguro que a la hora de empezar cualquiera de estos proyectos. Se empezara con un motor desde 0 con todo lo aprendido.

Bibliografía

[1] *Libro Blanco del Desarrollo Español de Videojuegos 2021*, 2021. [Online]. Available: https://dev.org.es/en/libroblancodev2021

[2] "Game engine wikipedia article." [Online]. Available: https://en.wikipedia.org/wiki/Game_engine

[3] J. Gregory, *Game Engine Architecture*, 2009.

[4] A. Kauch, "Content fueled gameplay programming in frostpunk," 2019. [Online]. Available: https://www.youtube.com/watch?v=9rOtJCUDjtQ&

[5] T. Ford, "Overwatch gameplay architecture and netcode," 2017. [Online]. Available: https://www.youtube.com/watch?v=zrIY0eIyqmI&

[6] "Unity wikipedia article." [Online]. Available: https://en.wikipedia.org/wiki/Unity_(game_engine)

[7] "Unity." [Online]. Available: https://unity.com/

[8] UE5, "Ue5 lumen documentation." [Online]. Available: https://docs.unrealengine.com/5.0/en-US/lumen-global-illumination-and-reflections-in-unreal-engine/

[9] ——, "Ue5 nanite documentation." [Online]. Available: https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/

[10] "Godot." [Online]. Available: https://godotengine.org/

[11] "Godot wikipedia article." [Online]. Available: https://en.wikipedia.org/wiki/Godot_(game_engine)

[12] "Godot features." [Online]. Available: https://godotengine.org/features

[13] "Gamemaker studio." [Online]. Available: https://gamemaker.io/en/gamemaker

[14] "Rpg maker." [Online]. Available: https://www.rpgmakerweb.com/

[15] Nvidia, "Nvidia rtx developer website." [Online]. Available: https://developer.nvidia.com/rtx/ray-tracing

[16] T. Takikawa, J. Litalien, K. Yin, K. Kreis, C. Loop, D. Nowrouzezahrai, A. Jacobson, M. McGuire, and S. Fidler, "Neural geometric level of detail: Real-time rendering with implicit 3d shapes," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 11 358–11 367. [Online]. Available: https://openaccess.thecvf.com/content/CVPR2021/html/Takikawa_Neural_Geometric_Level_of_Detail_Real-Time_Rendering_With_Implicit_3D_CVPR_2021_paper.html

[17] B. Anguelov, "Kruger engine: Game engine entity/object models," 2020. [Online]. Available: https://www.youtube.com/watch?v=jjEsB611kxs&

[18] M. Thorzen, "The witcher 3: Optimizing content pipelines for open-world games," 2015. [Online]. Available: https://www.youtube.com/watch?v=p8CMYD_5gE8

[19] "Siggraph, deep learning for character control compilation." [Online]. Available: https://github.com/sebastianstarke/AI4Animation

[20] A. Bereznyak, "Ik rig, procedural pose animation talk," 2018. [Online]. Available: https://www.youtube.com/watch?v=KLjTU0yKS00&

[21] aarthificial, "Pixel art animation. reinvented - astortion devlog," 2022. [Online]. Available: https://www.youtube.com/watch?v=HsOKwUwL1bE

[22] R. Fabian, *Data-Oriented Design*, Sep. 2018. [Online]. Available: https://www.dataorienteddesign.com/dodbook/

[23] "Programming across paradigms - goto 2017," 2017. [Online]. Available: https://www.youtube.com/watch?v=Pg3UeB-5FdA&

[24] B. Will, "Object-oriented programming is bad," 2019. [Online]. Available: https://www.youtube.com/watch?v=QM1iUe6IofM

[25] "Oop criticism wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Object-oriented_programming#Criticism

[26] "Single responsability principle." [Online]. Available: https://en.wikipedia.org/wiki/Single-responsibility_principle

[27] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, 2003.

[28] M. Acton, "Solving the right problems for engine programmers," Sep. 2020. [Online]. Available: https://www.youtube.com/watch?v=4B00hV3wmMY

[29] ——, "Data-oriented design and c++," Sep. 2014. [Online]. Available: https://www.youtube.com/watch?v=rX0ItVEVjHc

[30] A. Fredriksson, "Context is everything." [Online]. Available: https://vimeo.com/644068002

[31] J. Ward, "Oop is the root of all evil talk," 2012. [Online]. Available: https://www.youtube.com/watch?v=748TEIIlg14

[32] B. Will, "Object-oriented programming is good*," 2019. [Online]. Available: https://www.youtube.com/watch?v=0iyB0_qPvWk

[33] C. M. Mike Acton, "Handmadecon 2015, mike acton," May. 2015. [Online]. Available: https://www.youtube.com/watch?v=qWJpI2adCcs

[34] S. Nikolov, "Oop is dead, long live data-oriented design," Oct. 2018. [Online]. Available: https://www.youtube.com/watch?v=yy8jQgmhbAU

[35] "Pitfalls of object oriented programming, revisited," Sep. 2020. [Online]. Available: https://www.youtube.com/watch?v=VAT9E-M-PoE

[36] M. Godbolt, "Path tracing three ways: A study of c++ style," 2019. [Online]. Available: https://www.youtube.com/watch?v=HG6c4Kwbv4I&

[37] "Hazel engine github." [Online]. Available: https://github.com/TheCherno/Hazel

[38] "Entt: Modern ecs library github." [Online]. Available: https://github.com/skypjack/entt

[39] "Dear imgui github." [Online]. Available: https://github.com/ocornut/imgui

[40] "Noel berry github page." [Online]. Available: https://github.com/NoelFB

[41] "Celeste website." [Online]. Available: http://www.celestegame.com/

[42] "Do ... while(0) macro explanation." [Online]. Available: https://stackoverflow.com/questions/1067226/c-multi-line-macro-do-while0-vs-scope-block

[43] "Premultiplied alpha." [Online]. Available: https://microsoft.github.io/Win2D/WinUI3/html/PremultipliedAlpha.htm

[44] "Uses of rtti for games." [Online]. Available: https://gamedevcoder.wordpress.com/2013/02/16/c-plus-plus-rtti-for-games/

[45] G. Somberg, "Lessons learned from a decade of audio programming," 2016. [Online]. Available: https://www.youtube.com/watch?v=Vjm--AqG04Y&

[46] B. Waszak, "Gameplay programming patterns talk," 2021. [Online]. Available: https://www.youtube.com/watch?v=OPZs6V8EuHI

[47] ——, "Gameplay programming in a rich and immersive open world," 2016. [Online]. Available: https://www.youtube.com/watch?v=kk4dXY5dWsU

[48] D. Reed, "Networking scripted weapons and abilities in overwatch," 2017. [Online]. Available: https://www.youtube.com/watch?v=5jP0z7Atww4&

[49] A. Morlan, "A simple entity component system," 2019. [Online]. Available: https://austinmorlan.com/posts/entity_component_system/

[50] Unity, "Unity entities package." [Online]. Available: https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/index.html

[51] "Glfw website." [Online]. Available: https://www.glfw.org/

[52] "Glad github." [Online]. Available: https://github.com/Dav1dde/glad

[53] "Glm: Opengl mathematics github." [Online]. Available: https://github.com/g-truc/glm

[54] "Stb libraries github." [Online]. Available: https://github.com/nothings/stb

[55] "Optick: Super lightweight c++ profiler for games." [Online]. Available: https://optick.dev/

[56] "Box2d website." [Online]. Available: https://box2d.org/

[57] "Eastl: Ea standard template library." [Online]. Available: https://github.com/electronicarts/EASTL

[58] "Msbuild: Microsoft build engine." [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild

[59] "Cmake website." [Online]. Available: https://cmake.org/

[60] Microsoft, "Msvc: Microsoft visual c++." [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_Visual_C%2B%2B

[61] "Premake website." [Online]. Available: https://premake.github.io/

[62] "Position-independent code." [Online]. Available: https://en.wikipedia.org/wiki/Position-independent_code

[63] C. Muratori, "Handmade hero day 021 - loading game code dynamically," 2014. [Online]. Available: https://www.youtube.com/watch?v=WMSBRk5WG58&

[64] poncle, "Vampire survivors steam page," 2021. [Online]. Available: https://store.steampowered.com/app/1794680/Vampire_Survivors/