Escuela Politécnica Superior de Jaén
Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

PROTOTIPO DE MOTOR PARA EL DESARROLLO DE APLICACIONES GRÁFICAS PARA EL ENTRETENIMIENTO, DOCENCIA E INVESTIGACIÓN

Alumno

Miguel Jesús Collado Moreno

Tutores

Francisco de Asís Conde Rodríguez
(Departamento de Informática)

José Negrillo Cárdenas
(Departamento de Informática)

Septiembre, 2021

Universidad de Jaén

Departamento de Informática

Don Francisco de Asís Conde Rodríguez y Don José Negrillo Cárdenas, tutores del Trabajo Fin de Grado titulado: '**Prototipo de motor para el desarrollo de aplicaciones gráficas para el entretenimiento, docencia e investigación**', que presenta Don Miguel Jesús Collado Moreno, otorgan el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, Septiembre de 2021

El alumno: Los tutores:

Firmado por COLLADO MORENO MIGUEL JESUS - 77366870E el día 10/09/2021 con un certificado emitido por AC FNMT Usuarios

Miguel Jesús Collado Moreno

Francisco de Asís Conde Rodríguez

José Negrillo Cárdenas

Agradecimientos

Gracias a todos los que me han ayudado en este proceso y etapa en mi vida. A mis profesores, mis amigos y sobretodo a mis padres, Miguel y Charo, por apoyarme tanto y animarme en cada momento.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Fecha de asignación	23/03/2021
Descripción corta	Este trabajo de fin de grado es una propuesta para el desarrollo de un prototipo de un motor gráfico. Este tipo de software tiene una gran serie de ventajas, ya que permite abstraer el proceso de creación de una aplicación gráfica y facilitando en gran medida su desarrollo, ya sea mediante bibliotecas o de forma interactiva. Las aplicaciones gráficas a día de hoy se usan para multitud de situaciones: entretenimiento, docencia, medicina, industria, turismo, etc. En resumen, una aplicación gráfica es una herramienta interactiva que ayuda a realizar una tarea o tareas específicas. Sin embargo, el desarrollo de una aplicación gráfica suele ser un proceso tedioso, en el que se requieren múltiples partes para hacer funcionar la aplicación completa. En muchas ocasiones la aplicación gráfica se necesita únicamente para la visualización del resultado de un algoritmo geométrico, procesamiento de datos, etc., lo que implica un gasto de tiempo en su desarrollo. Un caso práctico puede ser para fines docentes, en los que los estudiantes tengan que implementar algoritmos específicos sin tener que usar, o bien, una aplicación propia o un gran motor gráfico. Este trabajo consiste en diseñar e implementar un prototipo de un motor gráfico generalista, que sea fácil de usar, que sea eficiente y sea flexible. Se incluirá una pequeña demostración de una aplicación desarrollada con el mismo para probar su eficacia.

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

Contenido

1 - Motor gráfico: TRON 13

1.1 - Introducción 13
1.1.1 - Estado del arte 13
1.1.2 - Justificación del proyecto 14
1.2 - Objeto del proyecto 15
1.3 - Definiciones y abreviaturas 16
1.4 - Organización y gestión del proyecto 17
1.5 - Especificaciones del sistema 21
1.5.1 - Análisis: Tron Core 21
1.5.1.1 - Modelado de interacción 21
1.5.1.2 - Requisitos 24
1.5.2 - Análisis: Tron Editor 25
1.5.2.1 - Modelado de interacción 25
1.5.2.2 - Requisitos 27
1.5.2.3 - Diseño de la interfaz 28
1.5.3 - Diseño del Sistema 29
1.6 - Presupuesto 31

2 - Desarrollo del proyecto 34

2.1 - Tecnologías utilizadas 34
2.1.1 - Software 34
2.1.1.1 - Visual Studio 34
2.1.1.2 - GitKraken 35
2.1.1.3 - Photoshop 35
2.1.1.4 - Google Chrome (Profiler) 36
2.1.2 - Tecnologías 37
2.1.2.1 - C++ 17 37
2.1.2.2 - ImGui 37
2.1.2.3 - Entt 37
2.1.2.4 - Git 38
2.2 - Primera Iteración: Tron Core 38
2.2.1 - Entry Point de la aplicación. 38
2.2.2 - Logging básico 41
2.2.3 - Implementación GLFW 42
2.2.4 - Layers 43
2.2.5 - Eventos 45
2.2.6 - ImGui 47
2.2.7 - Shaders 48
2.2.8 - Contexto de renderizado y Renderer 50
2.2.9 - Pruebas para esta iteración 53
2.3 - Segunda iteración: Tron Editor 54
2.3.1 - Visual Profiling 55
2.3.2 - Implementación ECS con EnTT 56
2.3.3 - Desarrollo de UI/UX 58
2.3.4 - Pruebas para esta iteración 58

3 - Conclusiones y trabajos futuros 59

4 - Apéndices 60

4.1 - Instalación y configuración del sistema 60

5 - Bibliografía 61

Índice de ilustraciones

Ilustración 1.1 Tecnologías usadas para desarrollar videojuegos por año 14
Ilustración 1.2 Ejemplo de gestión de tareas en Git (GitKraken) 18
Ilustración 1.3 Prototipo de interfaz del editor 28
Ilustración 2.1 Software Visual Studio 34
Ilustración 2.2 Software GitKraken 35
Ilustración 2.3 Software Photoshop 36
Ilustración 2.4 Profiler de Google Chrome 37
Ilustración 2.5 Entry point de la aplicación 39
Ilustración 2.6 Diagrama de clases de Application 40
Ilustración 2.7 Implementación de una aplicación Tron 40
Ilustración 2.8 Ejemplo de Log usando spdlog 41
Ilustración 2.9 Diagrama de clases de GraphicsContext 42
Ilustración 2.10 Ejemplo de uso de Layers en Tron 44
Ilustración 2.11 Diagrama de flujo del Ciclo de vida 45
Ilustración 2.12 Diagrama de flujo de eventos 47
Ilustración 2.13 Implementación básica de ImGui 48
Ilustración 2.14 Diagrama de clases de la arquitectura Shader 49
Ilustración 2.15 Shader en un único fichero glsl 50
Ilustración 2.16 Diagrama de flujo de inicialización del Renderer 51
Ilustración 2.17 Ejemplo de uso de RenderCommand 52
Ilustración 2.18 Batería de pruebas para la primera iteración 54
Ilustración 2.19 Visualización del profiling de una ejecución de un frame del Renderer 55
Ilustración 2.20 Estadísticas concretas de un método en el proceso de Rendering 56
Ilustración 2.21 Diagrama de clases de una entidad 57
Ilustración 2.22 Ejemplo de componente tipo Tag 57
Ilustración 2.23 Interfaz final de Tron Editor 58

Índice de tablas

Tabla 1.1 Tabla de costes asociada al proyecto *33*

1 - MOTOR GRÁFICO: TRON

1.1 - Introducción

A día de hoy, cada vez más la industria del desarrollo gráfico va creciendo más y más con la llegada de nuevas tecnologías al mercado. Tecnologías gráficas tan avanzadas como el Ray Tracing en tiempo real, proporcionado por la empresa NVIDIA, fabricante y creadoras de tarjetas gráficas del mismo nombre; llegando hasta los dispositivos móviles actuales, capaces de ejecutar aplicaciones gráficas muy potentes, que hace cuestión de 20 años ni se soñaba de su existencia.

Todo este desarrollo viene influenciado tanto por el hardware, como por el software, y concretamente, en el uso eficiente y adecuado de los recursos de cada dispositivo. Es por esto, que un buen desarrollo software es la clave para conseguir avanzar hacia adelante con este desarrollo tecnológico. Y de esta idea es de dónde partí para iniciar un viaje bastante duro y difícil de aprendizaje con este proyecto.

La idea de este trabajo de fin de grado (TFG) es la de desarrollar un prototipo de motor gráfico, con una API robusta y fácil de usar para la creación de diversos softwares gráficos (orientado tanto a investigación, docencia o incluso videojuegos), con el objetivo de entender y comprender cómo funciona un motor gráfico desde su arquitectura más básica, hasta su funcionamiento conjunto como una herramienta de desarrollo. En este trabajo, se ha creado desde 0, partiendo de los conocimientos adquiridos en las asignaturas de desarrollo gráfico del Grado de Ingeniería Informática, y profundizando en aspectos de arquitectura y desarrollo de software, claves para poder realizar este trabajo.

1.1.1 - Estado del arte

Actualmente, ya existen diversas soluciones profesionales al reto de crear aplicaciones gráficas de todo tipo. En el mercado existen diversos motores gráficos muy potentes, como pueden ser Unreal Engine, Unity, CryEngine o Source 2 entre muchos otros.

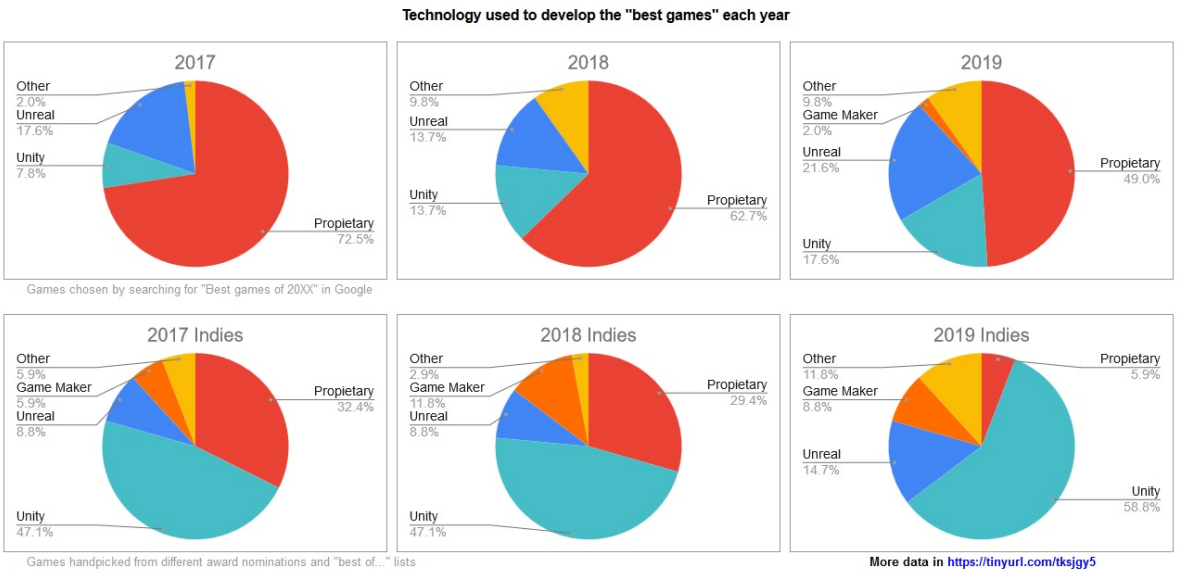

Ilustración 1.1 Tecnologías usadas para desarrollar videojuegos por año

Como podemos observar en la ilustración 1.1, desde el año 2017 en adelante, vemos un descenso de uso de motores desarrollados por la propia empresa, y un mayor uso de soluciones comerciales como estos motores anteriormente citados, siendo los más usados Unity y Unreal Engine. También ocurre algo similar en los juegos indies, siendo todavía más predominante el uso de Unity respecto al uso de un motor propietario. Este viene dado a que cada vez más programadores prefieren invertir su tiempo en crear el propio contenido de la aplicación, a desarrollar todo el esqueleto alrededor: al final, el tiempo es muy valioso, y estas herramientas favorecen a un uso más optimo del tiempo.

Aunque se vea claramente el descenso en uso a favor de crear tu propio motor gráfico, porcentualmente sigue siendo el más predominante.

1.1.2 - Justificación del proyecto

Desde la concepción del proyecto, la idea de este proyecto no es la de crear el “sustituto” perfecto para un motor gráfico de terceros, ni tampoco la de crear el motor gráfico definitivo, lleno de funcionalidades avanzadas. Al final, cuando un desarrollador opta por usar su propio motor gráfico, intercambia una serie de ventajas por otras, y dependiendo el problema a resolver, te puede beneficiar más o menos.

Las principales ventajas que obtienes al usar tu propio motor gráfico son las de tener una arquitectura personalizada al problema, en la que es el propio desarrollador es el que limita hasta donde desarrollar ciertas características, o qué características desechar; o una mayor eficiencia, al tener solo los sistemas que se ajusten al problema.

Pero, realmente mi objetivo en este proyecto no era resolver un único problema. Con este proyecto quiero tratar tres principales puntos a nivel personal:

- Crear una API que ayude tanto a desarrolladores de videojuegos, como a investigadores que se dediquen a aspectos gráficos.
- Crear un motor gráfico “generalista” con una curva de aprendizaje muy baja, o lo más baja posible. Uno de los grandes problemas de los motores gráficos creados exclusivamente para resolver un problema en específico es que no son nada reutilizables.
- Y por último, y no menos importante, formarme mucho más sobre arquitectura de software, arquitectura de motores gráficos y sobre programación en general.

Es este último punto el que me hizo seguir adelante con el proyecto, ya que para mi una de las cosas más importantes en la vida es la de seguir sumando conocimiento, y necesitaba de alguna forma entrar en el mundo del desarrollo gráfico y seguir formándome en la materia.

1.2 - Objeto del proyecto

En cualquier proyecto se deben establecer una serie de objetivos para poder así garantizar un desarrollo ideal del trabajo. Los objetivos principales para este proyecto son:

- Adquirir conocimiento sobre el diseño de aplicaciones y motores gráficos.
- Elegir y diseñar una arquitectura que permita desarrollar de forma eficiente el motor gráfico.

- Desarrollar un prototipo que permita crear una aplicación gráfica.
- Añadir herramientas para el análisis del rendimiento de las aplicaciones desarrolladas con el propio motor.

1.3 - Definiciones y abreviaturas

- **ECS**: Entity Component System. Es una arquitectura de software para mantener, de forma eficiente y separada un conjunto de objetos (llamados entidades), sus características (componentes) y su lógica de negocio (system). Tiene numerosas ventajas, pero la más importante de todas es la de mantener muy desacoplado tanto la definición del modelo de datos, de la propia lógica, haciendo que la instanciación de objetos sea mínima (ya que solo sería necesario guardar en memoria una pequeña estructura con metadatos de cada entidad).
- **GIT**: Git es un software de versionado de código, (con siglas VCS en inglés) que ayuda al desarrollador de software una manera fácil y segura de mantener su código versionado, y además permite compartir su base de código con los demás integrantes del equipo, de forma que cada uno puede trabajar paralelamente sobre la misma base de código.
- **Issue (Git):** Tarea básica mínima. Suelen ser objetivos a corto plazo
- **Feature (Git):** Rama en **Git** creada a partir de las especificaciones de una issue.
- **Release (Git):** Rama en Git creada para iniciar y finalizar una subida de versión del proyecto.
- **Hotfix (Git):** Rama en Git creada sobre master para realizar un cambio rápido que concluye en error.
- **Milestone (Git):** Tarea de larga duración. Son objetivos conjuntos orientados a largo plazo

1.4 - Organización y gestión del proyecto

La metodología usada para este proyecto ha sido una metodología incremental estándar, ya que, pese a proyectar desde un inicio las funcionalidades básicas que necesitará el proyecto, es difícil prever todos los requisitos que se necesitaran en el proyecto.

Respecto a la organización y gestión del proyecto, se ha usado **Git** tanto para mantener un versionado de código, como para gestionar todas las tareas y **milestones** que ha requerido el proyecto. Para tratar justamente este problema, se ha usado el sistema integrado que incorpora GitHub, que es una webApp que gestiona sus propios repositorios **Git**, y además incluye herramientas como la de Issue Boards, GitHub Actions, y muchas más.

Para poder llevar a cabo este proyecto, han sido necesarias de unas pautas de organización y de gestión, y que se han llevado a complimiento estricto. Dichas normas serían las siguientes.

- Usar dos ramas principales, **master** y **develop**. En **master** se subirán exclusivamente las versiones estables del proyecto, asignando un tag por versión. En **develop**, se irán mergeando las diferentes **features** que sean necesarias, probando cada funcionalidad después de cada subida para mantener una rama estable.

- Se usará **GitFlow**, una arquitectura de proyectos Git que nos ayuda a diferenciar mediante tres tipos de ramas los diferentes procesos del proyecto. Se usarán las ramas **feature**, **release** y **hotfix.**

  Cada **feature** se incorporará a develop, y nunca podrá subirse a master directamente.

  Cada **release** partirá del último commit de develop y se creará un tag con la versión del proyecto (por ejemplo, partiendo de 0.0.1) y mergeará tanto a master como a develop, para mantener la consistencia entre ramas.

- Cada **issue** será tratada como una **feature**, por lo que la relación entre las issues del repositorio y cada rama será directa. Así podremos hacer back-tracking de cualquier problema si lo hubiera.

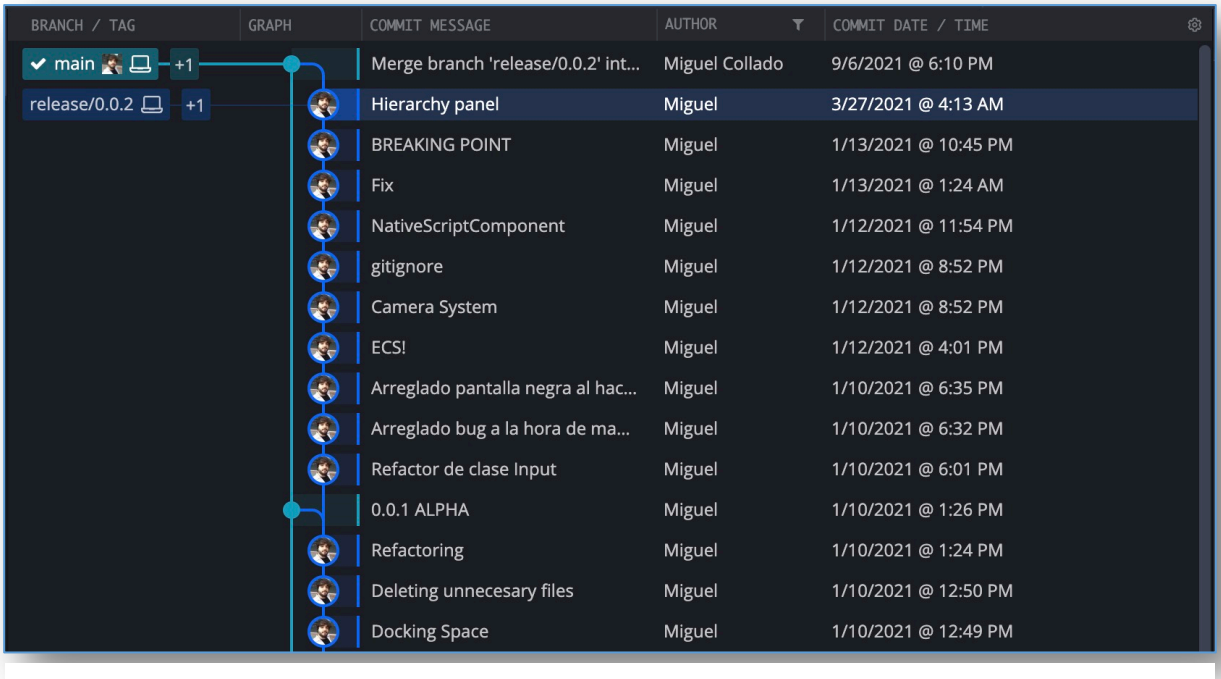

Ilustración 1.2 Ejemplo de gestión de tareas en Git (GitKraken)

Aquí (ilustración 1.2) proporciono un ejemplo de mi gestión de tareas sobre el proyecto. Inicialmente, se plantearon tres grandes milestones: **Arquitectura Base**, **Renderer** y **ECS**.

El tiempo asignado a cada milestone ha sido de aproximadamente **1 mes**. Lo ideal es dejar un plazo fijo para cada milestone, pero debido a las características de tiempo y del tipo de proyecto, se ha planteado mejor un tiempo flexible.

Además, cada issue se ha planteado para ser realizada como máximo en dos días (48h). Las issues de este proyecto han sido las siguientes, ordenadas por milestone:

1. **MILESTONE: Arquitectura Base**

   (Estimación: 2 semanas, Duración Real: 1 semana)

   - **1# Entry Point**: Creación del punto de entrada de la aplicación. Duración real: 1 hora.
   - **2# Logging**: Creación de un sistema de Log básico para el Core. Duración real: 1 hora.

- **3# Premake:** Implementación de un fichero premake para fafcilitar la instalación del sistema. Duración real: 1 hora.
- **4# Event System:** Creación de un esqueleto para el manejo de eventos en la aplicación. Duración real: 2 horas.
- **5# GLFW:** Incorporación del framework GLFW en el proyecto. Duración real: 3 horas.
- **6# Layers:** Creación del concepto de Capas (Layers) en el proyecto. Duración real: 2 horas.
- **7# ImGui:** Incorporación al proyecto de la biblioteca de intefaces ImGui. Duración real: 5 horas.

2. **MILESTONE: Renderer**

   (Estimación: 1 mes, Duración Real: 19 días)

   - **8# Rendering Context:** Creación de una aquitectura de Renderer multiplataforma. Duración real: 2 días.
   - **9# Shaders:** Abstracción e implementación de una arquitectura de Shaders. Duración real: 4 días.
   - **10# Buffers:** Abstracción e implementación de una aquitectura de Buffers. Duración real: 1 día.
   - **11# Renderer:** Finalización de la creación del Renderer. Duración real: 1 día.
   - **12# Shader Library:** Creación de una arquitectura de componentes Shaders en fichero. Duración real: 5 días.
   - **13# Profiling:** Implementación básica de profiling para todo el ciclo de rendering de la aplicación. Duración real: 2 días.
   - **14# Batch Rendering:** Implementación básica de un algoritmo de Batch Rendering. Duración real: 4 días.

3. **MILESTONE: ECS**

   (Estimación: 1 mes, Duración Real: 15 días)

- **15# Implementación base ECS:** Creación de el esqueleto base para crear un ECS. Duración real: 3 días.
- **16# Camera System:** Creación del sistema de cámaras. Duración real: 2 día.
- **17# Basic Component:** Creación de componentes básicos para ECS. Duración real: 10 días.

En este proyecto serán necesarias varias iteraciones para alcanzar un proyecto completo. Para ello, y basándome en los milestone creados, se realizaron dos iteraciones finales, con su correspondiente tag creado en Git.

1. **Tron Core**. Funcionalidad básica y esqueleto inicial. Corresponde con el milestone **Arquitectura Base y Renderer**
2. **Tron Editor**. Programa con editor UI visual, usando ECS. Corresponde con el milestone **ECS**.

Gracias a plantear el proyecto con estas dos grandes iteraciones, podremos probar partes diferenciadas del prototipo, como son el propio Core de la aplicación, como el Editor, y todos los sistemas acoplados a ellos. En el capítulo *Desarrollo del proyecto* se extenderá todo el funcionamiento y decisiones tomadas al respecto.

A continuación, se explicará de forma general cada iteración:

1. *Primera iteración: Tron Core.*

La idea principal tras esta iteración es comprobar que los sistemas principales de la aplicación son funcionales y no contienen fallos. Para ello, se creó un sub-proyecto de este, llamado Sandbox (se incluye en los archivos de código fuente del proyecto) en el que se testean todos los sistemas que verifican el funcionamiento correcto del prototipo.

2. *Segunda iteración: Tron Editor.*

En esta iteración, se comprueba tanto que el ECS está funcionando correctamente, como que la nueva funcionalidad del Editor está correctamente implementada. Para ello, al igual que en la primera iteración, se creó un sub-proyecto del Core y se llamó Tron Editor.

Hay que destacar que, tras acabar cada issue, la nueva funcionalidad es probada en un proyecto Tron llamado **Sandbox**, en el que se realizan todas las pruebas pertinentes para asegurar que la nueva funcionalidad no presenta errores críticos, y que está preparada para subirse al repositorio.

1.5 - Especificaciones del sistema

En este apartado se definirán todos los requisitos funcionales y no funcionales del sistema, además de los casos de uso que existen en él.

Como se ha comentado anteriormente, en este proyecto existen dos grandes bloques que habría que definir. Por un lado, está Tron Core, que representa el motór gráfico y la API de desarrollo, y por otro lado Tron Editor, que representa la implementación de una aplicación tipo Tron. Por ello, será necesario definir ambas estructuras.

1.5.1 - Análisis: Tron Core

1.5.1.1 - Modelado de interacción

Para modelar la interacción que el actor (en este caso, el programador) va a realizar con el sistema, se especificarán una serie de casos de uso para la API de Tron Core. Los casos de uso para este proyecto son los siguientes:

- **Renderer**. Es uno de los casos de uso principales. Representa la posibilidad de manejar mediante una API el Renderer principal de la aplicación.
- **Manejo de eventos**. Representa la posibilidad de responder a eventos ejecutados por el actor principal.

- **Manejo de escenas**. Representa la capacidad de poder crear escenas constituidas por entidades (ECS) y aplicar configuraciones estancas entre ellas.
- **ECS**. Es otro de los casos de uso principales. Representa la capacidad de registrar, editar y eliminar entidades en el sistema, añadir componentes a entidades y proporcionar lógica a esas entidades.
- **Manejo de recursos**. Representa la capacidad de poder manejar imágenes, fuentes, ficheros, etc. Actualmente, podemos manejar Texturas y Shaders en el sistema.

1.5.1.1.1 - Renderer

- **Actores:** Programador
- **Precondición:** Haber creado un entry point implementando un *Application.*
- **Escenario principal:**
  1. El programador Inicializar y crear una escena en *OnAttach*().
  2. El programador, en el método *OnUpdate*(), inicia una escena mediante *Renderer2D*::*BeginScene*()
  3. El programador crea un objeto mediante un *RenderCommand*, como por ejemplo un *DrawQuad*()
  4. El programador acaba la escena con *Renderer2D::EndScene();*
- **Postcondición:** Se visualiza en pantalla la escena recién creada.

1.5.1.1.2 - Manejo de eventos

- **Actores:** Programador
- **Precondición:** Haber creado un entry point implementando un *Application.*
- **Escenario principal:**
  1. El programador recibe los eventos en el método *OnEvent()* de *Application.*
  2. El programador, en función del tipo de evento, despacha el evento al correspondiente gestor de eventos.

- **Postcondición:** El evento es procesado y eliminado del sistema.

1.5.1.1.3 - Manejo de escenas

- **Actores:** Programador
- **Precondición:** Haber creado un entry point implementando un *Application.*
- **Escenario principal:**
    1. El programador crea una escena en OnAttach().
    2. El programador añade las entidades que necesite
    3. El programador actualiza la escena en *OnUpdate()*.
    4. El sistema automáticamente recoge cada entidad en la escena y hace render de cada una.
- **Postcondición:** Por cada frame, la escena se renderiza teniendo en cuenta cada entidad en ella.

1.5.1.1.4 - ECS

- **Actores:** Programador
- **Precondición:** Haber creado una escena en la *Layer* actual.
- **Escenario principal:**
    1. El programador crea una entidad usando *Scene->CreateEntity()*
    2. El programador añade un componente tipo *SpriteRendererComponent* para generar un cuadrado y añadir un color por defecto a la entidad.
- **Postcondición:** Se visualiza la entidad correspondiente en la escena

1.5.1.1.5 - Manejo de recursos

- **Actores:** Programador
- **Precondición:** Haber creado un entry point implementando un *Application.*
- **Escenario principal:**

1. El programador crea una instancia de Texture2D en OnAttach() de la capa actual.
2. El sistema busca y carga la textura, devolviendo un puntero a al objeto.

- **Postcondición:** Se ha cargado el recurso en el sistema y pueden ser usados por el resto de la aplicación.

1.5.1.2 - Requisitos

Ahora, habiendo definido los casos de uso más generales, se definirán los requisitos que la aplicación deberá cumplir. En esta sección se incluirán tanto los requisitos funcionales como los no funcionales.

- *Requisitos funcionales*

  Los requisitos funcionales representan las funcionales o tareas que la aplicación debe ser capaz de realizar. En este caso, estos requisitos están orientados a crear una API de rendering.

  1. **Crear una interfaz para poder manejar el Renderer**. Debe de ser totalmente transparente con la API gráfica usada, para garantizar la propiedad multi-api del proyecto.
  2. **Crear una interfaz para poder manejar eventos**. Estos eventos pueden ser desde eventos por interacción humana, como eventos generados por el sistema. Es necesario proporcionar una API en la que poder suscribirse desde el código cliente, para que sea el programador el responsable de la lógica de cada evento.
  3. **Crear un manejador de escenas**. Será necesario que el proceso de recolección de entidades sea invisible para el usuario.
  4. **Crear un sistema ECS** (Entity-Component-System). Funcionará en tándem con el requisito anterior de escenas, ya que están fuertemente ligados. Se deberá implementar un registro de entidades persistente en memoria que interactuará en todo momento con la escena activa.

5. **Crear un sistema de manejo de recursos.** Será necesario implementarlo de forma transparente al sistema operativo en el que nos encontremos, y deberá ofrecer una API lo más completa posible.

- *Requisitos no funcionales*

  Los requisitos no funcionales sin embargo, representan características que no definen una funcionalidad específica de la aplicación. En este caso, se definirán teniendo en cuenta características de facilidad de uso, nomenclatura simple y sencilla, y sobretodo eficiencia.

  1. La nomenclatura de métodos y atributos debe ser consistente. Se usará PascalCase para Métodos y CamelCase para atributos de clase.
  2. La API debe ser lo más desacoplada posible. Se debe delegar funcionalidades de forma que cada una de ellas esté contenida en clases diferenciadas.
  3. Se debe mantener una estructura de código limpia y concisa.
  4. Se debe desarrollar la API teniendo en cuenta la capacidad del sistema de poder cambiar de API gráfica y de plataforma en cualquier momento.

1.5.2 - Análisis: Tron Editor

1.5.2.1 - Modelado de interacción

Para especificar la interacción entre la aplicación y el usuario que lo usa, se recogerán los distintos casos de uso. En este caso, el actor definido será el programador o usuario técnico. Al ser un prototipo, contamos con pocos casos de uso, que irán creciendo a medida que sean necesarias más funcionalidades. Los principales casos de uso para esta aplicación serían los siguientes:

- **Manejo de entidades**. Caso de uso principal. Representa tanto la creación como la destrucción de entidades en el editor. Una entidad es un objeto virtual al que se le acoplan funcionalidades o *componentes.*
- **Configuración de componentes**. Representa la capacidad del usuario para añadir o quitar funcionalidad a las diferentes entidades.

1.5.2.1.1 - Manejo de entidades

- **Actores**: El programador o técnico
- **Precondición**: Tener creada una escena inicial. (Por defecto)
- **Escenario principal:**
  1. El programador hace click en Crear entidad.
  2. El editor crea una nueva entidad y la registra.
  3. El programador accede al menú de configuración.
  4. El programador añade un nuevo componente a la entidad.
- **Postcondición**: La nueva entidad y su componente quedan registrados en el sistema y preparados para render.

1.5.2.1.2 - Configuración de componentes

- **Actores**: El programador o técnico
- **Precondición:** Haber creado una entidad y haber añadido un componente
- **Escenario principal:**
  1. El programador selecciona la entidad que desea modificar.
  2. El editor genera una vista de configuración con los componentes acoplados
  3. El programador editar la configuración específica de un componente.
  4. El editor maneja los eventos de modificación de esos valores y los guarda.
- **Postcondición**: La entidad afectada cuenta con una nueva configuración y está preparada para render.

1.5.2.2 - Requisitos

Ahora, habiendo definido los casos de uso más generales, se definirán los requisitos que la aplicación deberá cumplir. En esta sección se incluirán tanto los requisitos funcionales como los no funcionales.

- *Requisitos funcionales*

  Los requisitos funcionales representan las funcionales o tareas que la aplicación debe ser capaz de realizar. En este caso, estos requisitos sobretodo están orientados a la experiencia de usuario (UX) y a la interfaz de usuario (UI).

  1. Crear un sistema de gestión de entidades (ECS). Se podrá añadir, eliminar o editar cada entidad en el sistema.
  2. Crear un panel en la interfaz que contenga las entidades registradas.
  3. Crear un panel en la interfaz en la que poder ver los componentes añadidos a una entidad.
  4. Crear un panel en la interfaz en la que poder modificar las configuraciones de los componentes específicos de cada entidad.
  5. Implementar un panel de visualización del viewport.

- *Requisitos no funcionales*

  Los requisitos no funcionales sin embargo, representan características que no definen una funcionalidad específica de la aplicación. En este caso, se definirán teniendo en cuenta características de usabilidad e interfaz, entre otras.

  1. La interfaz gráfica debe mantener una disposición limpia y sencilla, para inducir al menor error posible durante la interacción.
  2. El uso del editor deberá ser intuitivo para que no sea necesaria una especialización para usar el software.
  3. La experiencia de uso deberá ser lo más fluida y de respuesta lo más rápida posible.

4. La aplicación deberá ser eficiente. Es decir, se deberá diseñar de tal forma que consuma solamente los recursos verdaderamente esenciales que este necesite. Esto favorece la fluidez de la aplicación.

5. Se deberá usar una paleta de colores que favorezca su visualización para cualquier persona. Es decir, personas con visibilidad reducida, personas con daltonismo etc…

1.5.2.3 - Diseño de la interfaz

Para realizar el diseño de la interfaz, se desarrollaron diferentes prototipos a mano, basándose en la propia interfaz de Unity como referencias. Las claves principales a tener en cuenta es la usabilidad, por lo que solo se mostraran en pantalla funcionalidades útiles para el usuario, ocultando procesos que no sean necesarios de la vista principal.

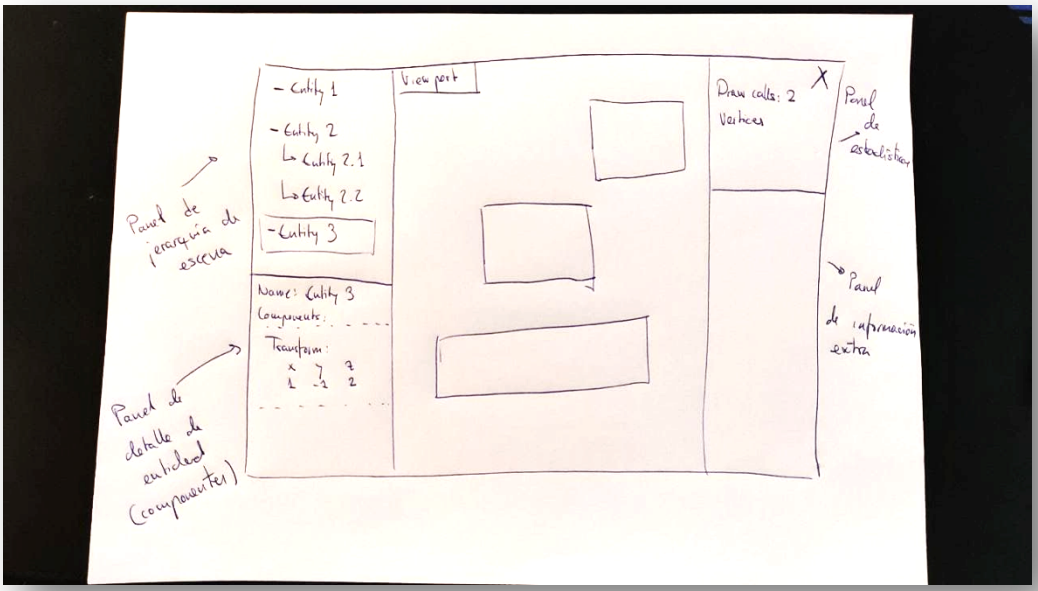

Ilustración 1.3 Prototipo de interfaz del editor

Este es el prototipo inicial de la interfaz del editor. La ventana se dividiría en 4 partes fundamentales:

- **Un panel con el árbol de jerarquía de la escena** (arriba izquierda). En él, se muestran todas las entidades que pertenecen a esta escena. Se muestra

un botón con el tag de nombre de cada entidad, y un desplegable en el caso en el que haya entidades hijas anidadas a ella.

- **Un panel con el detalle de cada entidad** (abajo izquierda). Aparecería al hacer click sobre una de las entidades del panel de jerarquía. Aquí podemos encontrar toda la información de la entidad, y además todos los componentes que se han añadido a la entidad.
- **El viewport** (centro). Es el panel principal. Aquí se visualiza todo el contexto gráfico. En este caso, se muestra la escena activa.
- **Panel con estadísticas** (arriba derecha). Dentro de este, aparecerá información relativa a la aplicación, como número de objetos renderizados, número de vértices, llamadas a la API gráfica etc.

1.5.3 - Diseño del Sistema

Una vez realizado el análisis del proyecto, se desarrollará el diseño arquitectónico del sistema, explicando a grandes rasgos los diferentes componentes del mismo.

Tron cuenta con dos apartados claramente diferenciados. El núcleo, o Core, en el que se encuentran todos los sistemas implementados, y Plataforma, o **Platform**, en el que se encuentran las implementaciones específicas por plataforma y por API gráfica del motor.

Con esta separación garantizamos que el sistema está fuertemente desacoplado de funcionalidades específicas, manteniendo una API limpia y expandible. La ventaja de tratar con un Core externo, es que, a diferencia de otros motores gráficos comerciales como **Unity** o **Unreal Engine**, es totalmente ampliable, y su núcleo está totalmente abierto para su inspección, proporcionando al programador todo el poder de decidir cada implementación específica que requiera su aplicación.

Dentro del núcleo de **Tron,** se han separado todas las funcionalidades en diferentes paquetes. Los sistemas que conforman el núcleo son los siguientes:

- **Core**. Cuenta con todos los sistemas principales del proyecto. Los sistemas principales en el Core son:
    - **Application**
    - **EntryPoint**
    - **Layer**
    - **Log**
    - **Window**
    - **Timestep**
- **Debug**. Sistemas básicos para ayudar al desarrollados con las tareas de implementación. Dentro, nos encontramos con el sistema **Instrumentor.**
- **Events.** Sistemas de recogida y despachado de eventos. Los sistemas principales son:
    - **ApplicationEvent**
    - **KeyEvent**
    - **MouseEvent**
- **ImGui.** Sistema de UI/UX implementado con la biblioteca de interfaces ImGui.
- **Renderer.** Sistema principal de la aplicación. En este paquete nos encontramos todos los pequeños sistemas que en conjunto crear el Renderer de la aplicación. Nos podemos encontrar con:
    - **Renderer2D**
    - **RendererAPI**
    - **Shader**
    - **Texture**
    - **VertexArray**
    - **Buffer**
    - **Framebuffer**

        - **GraphicsContext**
        - **RenderCommand**
    - **Scene.** En este paquete nos encontramos los sistemas encargados de manejar el ECS y de agruparlo todo en escenas. Los sistemas principales son:
        - **Entity**
        - **Components**
        - **Scene**
        - **SceneCamera**
        - **ScriptableEntity**
    - **Scripts.** En este paquete se encuentran implementados una serie de scripts que, añadiéndolos a una entidad, son capaces de ofrecer lógica usando los componentes existentes en esta. Actualmente, sólo contamos con un script llamado CameraController, que es el encargado de manejar la cámara en una escena.

Además, como se ha comentado anteriormente, en **Platform** nos encontraremos las implementaciones de los sistemas específicos, como **Window** o **GraphicsContext**. Este apartado está dividido en diferentes paquetes, un paquete por API gráfica (OpenGL, Vulkan…) y por plataforma (Windows, Linux, Mac).

En el apartado de desarrollo del Proyecto (véase punto 2.) se explicará detenidamente cada uno de los sistemas implicados más importantes.

1.6 - Presupuesto

Respecto al apartado de presupuesto, se han dividido los costes según su naturaleza, es decir, se han diferenciado costes software (como los costes generados

por pagos de software, suscripciones etc…), costes hardware (compra de equipos, componentes etc.) y costes de personal (salarios, dietas…).

En el apartado de software se incluirán todos los programas utilizados en el proyecto. Todo el software usado en el proyecto tiene un coste de suscripción mensual, o son de uso libre.

Adobe Photoshop tiene un coste mensual de 24,19 €/mes por su suscripción anual, lo que hace un total de 290,28€. Este software no se utilizará exclusivamente en este proyecto, ya que lo uso en otros proyectos por lo que se amortizará su coste en función del tiempo empleado en el proyecto

GitKraken tiene un coste mensual de 4.95€/mes por su suscripción anual. Al final que Photoshop, es un programa que uso regularmente en otros proyectos, por lo que se amortizará su precio en función del tiempo usado en este proyecto.

La programación se ha realizado en entornos de desarrollo gratuitos (Visual Studio Community en este caso) por lo que no es necesario añadirlo a la contabilidad.

En el apartado de hardware se incluirá mi ordenador personal, que ha sido con el que se ha trabajado en el proyecto.

En el apartado de personal se incluirá un salario de una persona, que ha sido el desarrollador principal.

- Costes Software
    1. Adobe Photoshop
        - Coste de la licencia: 290,28 €
        - Vida útil: 12 meses.
        - Amortización mensual: $\frac{290{,}28€}{12\ meses} = 24{,}19€/mes$
        - Coste imputable al proyecto: $24{,}19€ * 3\ meses = 72{,}57€$

2. GitKraken
    - Coste de la licencia: 59,40 €
    - Vida útil: 12 meses.
    - Amortización mensual: $\frac{59{,}40€}{12\ meses} = 4{,}95€/mes$
    - Coste imputable al proyecto: $4{,}95€ * 3\ meses = 14{,}85€$

- Costes hardware
    1. Ordenador personal
        - Coste: 2400€
        - Vida útil: 4 años
        - Amortización mensual: $\frac{2400€}{12\ meses} = 200€/mes$
        - Coste imputable al proyecto: $200€ * 3\ meses = 600€$
- Costes de personal
    1. Desarrollador principal del proyecto
        - Hora de trabajo: 20€/hora
        - Horas trabajadas: 300 horas (duración máxima de un TFG)
        - Coste imputable al proyecto: $20€ * 300\ HORAS = 6000€$
- Coste total

Concepto	Coste
Costes software	87,42€
Costes hardware	600€
Costes de personal	6.000€
Subtotal	6.687,42€
Beneficio (15%)	1.003,11€
Total antes de impuestos	7.690,53€
IVA (21%)	1.615,01€
Total	9305,54€

Tabla 1.1 Tabla de costes asociada al proyecto

2 - DESARROLLO DEL PROYECTO

2.1 - Tecnologías utilizadas

Para lograr completar este proyecto, se han usado una serie de tecnologías y software específico para poder desarrollarlo correctamente.

2.1.1 - Software

En este apartado se listarán todos los programas usados durante el desarrollo del TFG.

2.1.1.1 - Visual Studio

Microsoft Visual Studio es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) para Windows y macOS. Es compatible con múltiples lenguajes de (10) programación, tales como C++, C#, Visual Basic .NET, F#, Java, Python, Ruby y PHP, al igual que entornos de desarrollo web, como ASP.NET MVC, Django, etc., a lo cual hay que sumarle las nuevas capacidades en línea bajo Windows Azure en forma del editor Monaco.

Ha sido la herramienta de desarrollo principal de código fuente del proyecto. Usar un IDE te aporta grandes ventajas, ya que te ayuda a crear un código mejor estructurado y más limpio (usando herramientas internas como IntelliSense) y te

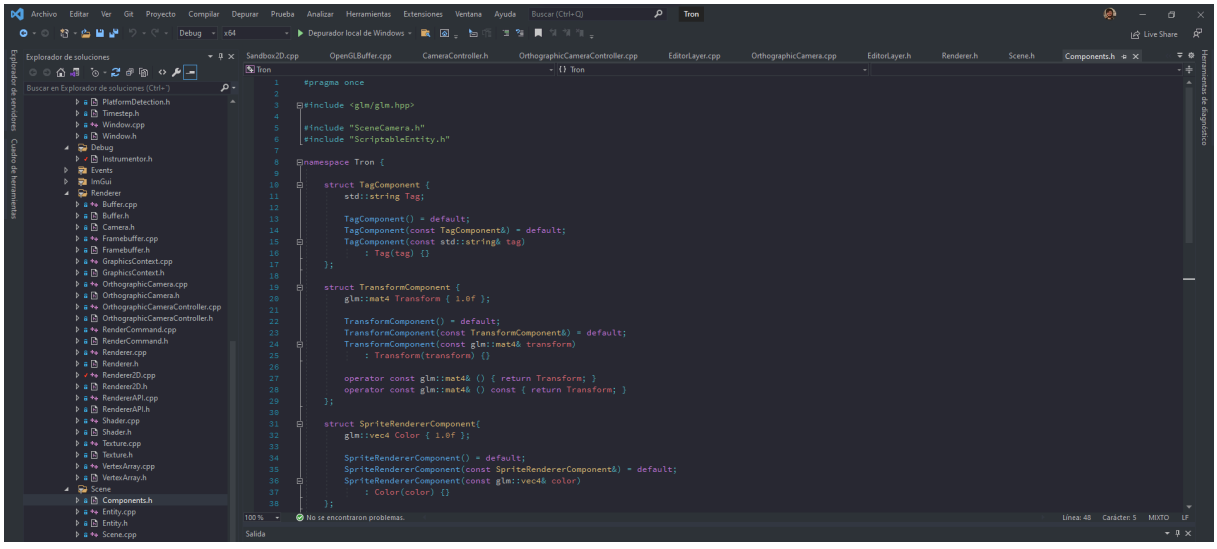

Ilustración 2.1 Software Visual Studio

permite hacer pruebas y lanzar las aplicaciones más fácilmente, sin necesidad de realizar pasos de compilación tediosos por terminal.

2.1.1.2 - GitKraken

GitKraken es un cliente Git para Windows, Mac y Linux. Su principal funcionalidad principal es la de manejar de una forma sencilla repositorios de Git (tecnología de control de versiones para código fuente). Su particularidad es su simpleza a la hora de usarse y su interfaz amigable.

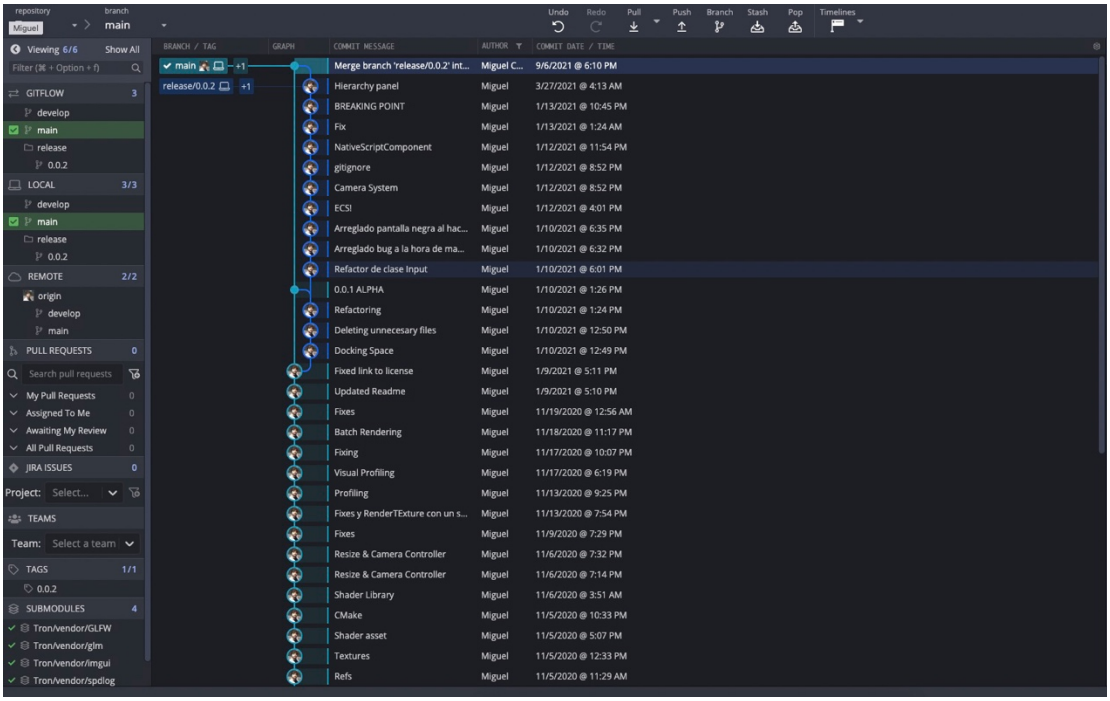

Ilustración 2.2 Software GitKraken

2.1.1.3 - Photoshop

Adobe Photoshop es una herramienta de edición de fotos e imágenes muy potente. Posee una gran cantidad de efectos fotográficos y herramientas de retoque digital que ayudan a la hora de crear todo tipo de elementos gráficos.

Se ha usado principalmente para preparar texturas e imágenes que se usaron en el proyecto.

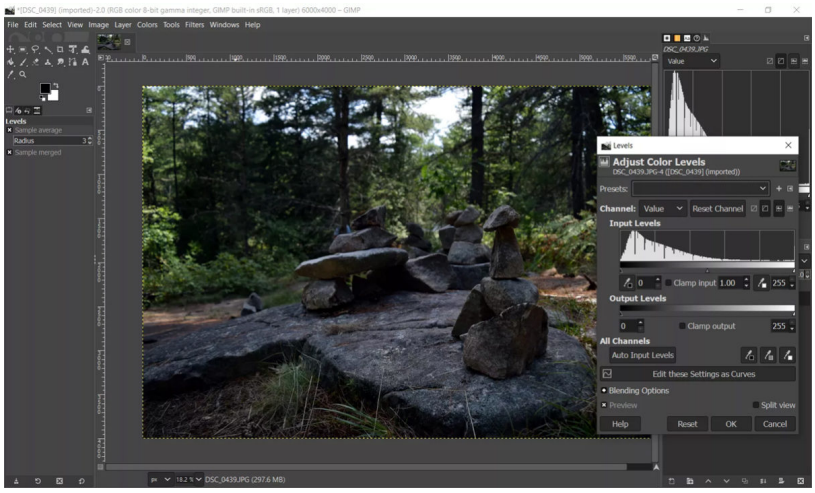

Ilustración 2.3 Software Photoshop

2.1.1.4 - Google Chrome (Profiler)

El Profiler de Google Chrome [] es una pequeña herramienta que incluye el conocido navegador web Google Chrome que sirve originalmente para crear gráficos de rendimiento y respuesta de peticiones web. Tiene una función muy importante, que es la de poder importar un fichero JSON (Javascript Object Notation) con cualquier tipo de información para crear gráficos interesantes para el proyecto.

Se ha usado en el proyecto para visualizar los metadatos temporales de ejecución de la aplicación, pudiendo así comparar y mejorar aspectos del propio desarrollo.

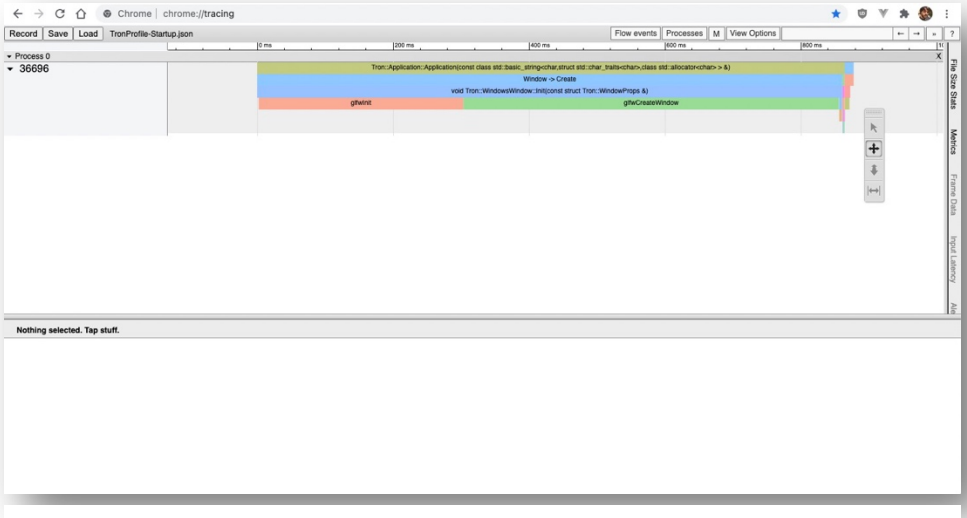

Ilustración 2.4 Profiler de Google Chrome

2.1.2 - Tecnologías

2.1.2.1 - C++ 17

C++17 es la versión más reciente del estándar del lenguaje de programación C++, que a su vez, es un lenguaje que extiende el primigenio lenguaje de programación C. Principalmente se ha incluido porque permite desarrollar de forma más limpia el propio código de la aplicación, haciéndolo más robusto y fácil de mejorar con el tiempo.

2.1.2.2 - ImGui

ImGui es una pequeña biblioteca de creación de interfaces de usuario para C++. Es pequeña, liviana y con una arquitectura fácil para poder realizar interfaces de usuario muy robustas. Se ha usado para la creación de todos los elementos auxiliares del motor, y para crear el editor completamente.

2.1.2.3 - Entt

Entt es una pequeña y liviana biblioteca que gestiona el uso de una arquitectura basada en componentes (Entity-component system, o ECS) para C++. El uso de esta tecnología va ligado con la idea general de hacer una API robusta, y precisamente permite gestionar pequeñas funcionalidades, llamadas componentes, de una forma

totalmente desacoplada de la lógica de la propia aplicación, haciendo que fragmentos de código puedan reusarse a modo de módulos.

2.1.2.4 - Git

Git es un sistema de versionado de código gratuito y open-source, diseñado tanto para proyectos pequeños como grandes. Es una pieza fundamental en cualquier desarrollo de software actual y moderno, ya que permite llevar un registro de cada cambio hecho en tu proyecto, además de solucionar todos los problemas provenientes de trabajar en equipo, como es la sincronización de ficheros remotos, y de cambios.

2.2 - Primera Iteración: Tron Core

Como ya se comentó anteriormente en el apartado de *Organización y Gestión del proyecto,* para realizar correctamente este proyecto se han llevado a cabo dos grandes iteraciones de desarrollo. En la primera iteración, se implementarán los sistemas base de los que posteriormente se nutrirán todas las funcionalidades más abstractas del stack de tecnologías.

A continuación, y siguiendo el orden descrito en las tareas en el apartado de *Organización y gestión del proyecto,* explicaré detenidamente cada tarea realizada.

2.2.1 - Entry Point de la aplicación.

Lo principal a la hora de crear cualquier proyecto software, desde mi punto de vista, es comenzar con una buena base sólida y con pocas fisuras (o al menos tratar de dejar las menos posibles). Por eso, quizás este apartado sea uno de los más complicados de toda la aplicación.

Inicialmente, se planteó que el punto de entrada inicial de la aplicación fuese similar al usado durante el trascurso del grado, en el que existe un fichero “main” el cual es encargado de tanto crear la aplicación, gestionar eventos, mantener el ciclo de eventos y de finalizar la ejecución.

Sin embargo, para este proyecto se planteó de una forma similar pero tratando de simplificar y desacoplar todo lo posible el punto inicial del sistema. Ahora, el archivo "main" simplemente realizará una serie de llamadas a una API genérica, sencilla, y simple ¿Qué ganamos con este cambio?

1. **Limpieza de código**. Mantener una API limpia y robusta es el principal foco de este proyecto. Haciendo que desde la parte de cliente el uso sea sencillo, evitamos confusiones y complejidad a la hora de desarrollar una aplicación usando Tron.
2. **Desacoplamiento**. No se garantiza un desacople total, ya que al final estas usando una API, pero si se encapsula todo el comportamiento dentro del Core. Con esto, podemos generar más cómodamente código multiplataforma y multi-api, que es otro aspecto fuerte de este proyecto.
3. **Facilidad de desarrollo**. Viene dado por la limpieza de código, pero es importante que la API también sea de fácil uso semántico.

```
extern Tron::Application* Tron::CreateApplication();

int main(int argc, char** argv) {

    auto app = Tron::CreateApplication();
    app->Run();
    delete app;
}
```

Ilustración 2.5 Entry point de la aplicación

Como se puede observar (ilustración 2.5), la limpieza y sencillez son notorias, y esto hace que el desarrollo y el testeo sea mucho más ágil. Simplemente, la responsabilidad de la lógica de negocio en este caso reside en **Application,** y es

EntryPoint el que usa su API para crear, mantener el ciclo de vida y destruir la aplicación.

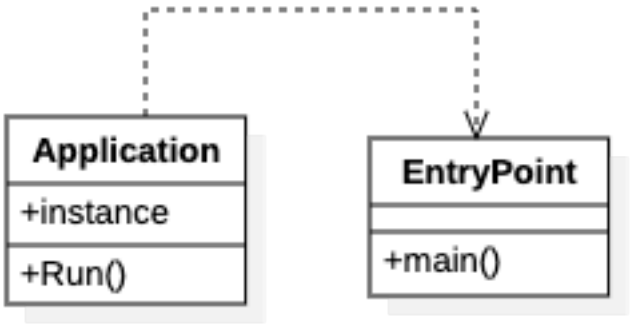

Ilustración 2.6 Diagrama de clases de Application

La clase **Application** es la encargada de manejar el ciclo de vida de la aplicación. Esta clase se ha implementado de tal forma que sirve como interfaz para crear aplicaciones Tron, por lo tanto, abstraemos este proceso mucho.

```
namespace Tron {
    class TronEditor : public Application {
    public:
        TronEditor() {
            PushLayer(new EditorLayer());
        }

        ~TronEditor() override = default;
    };

    Application* CreateApplication() {
        return new TronEditor();
    }
}
```

Ilustración 2.7 Implementación de una aplicación Tron

En este ejemplo (ilustración 2.7), podemos ver cómo crear una aplicación **Tron** desde 0. Creamos una clase principal, llamada **TronEditor** que hereda de **Application**. A partir de aquí, la tarea es muy sencilla. Sólo necesitamos implementar el método *CreateApplication()* (que si observamos en la ilustración 2.6, podemos ver que es el método que se llamará siempre desde nuestro Entry Point) y si lo necesitamos, crearemos tantas **Layers** como necesitemos (se explicará a fondo en el apartado 2.2.4 Layers).

2.2.2 - Logging básico

Otro aspecto fundamental para cualquier desarrollo es contar con algún sistema para obtener en tiempo de ejecución información de nuestra aplicación para probar y determinar fallos.

Para ello, se implementó un sistema básico de logs basado en macros de C++, usando la biblioteca **spdlog.** Spdlog no es más que una pequeña biblioteca de logs por consola, muy sencilla de usar, pero muy potente. Usándola como base, se crearon unas macros que se pueden usar a nivel global en el Core, y que simplemente envuelven la API de **spdlog** y nos muestra, de una forma formateada, información por consola.

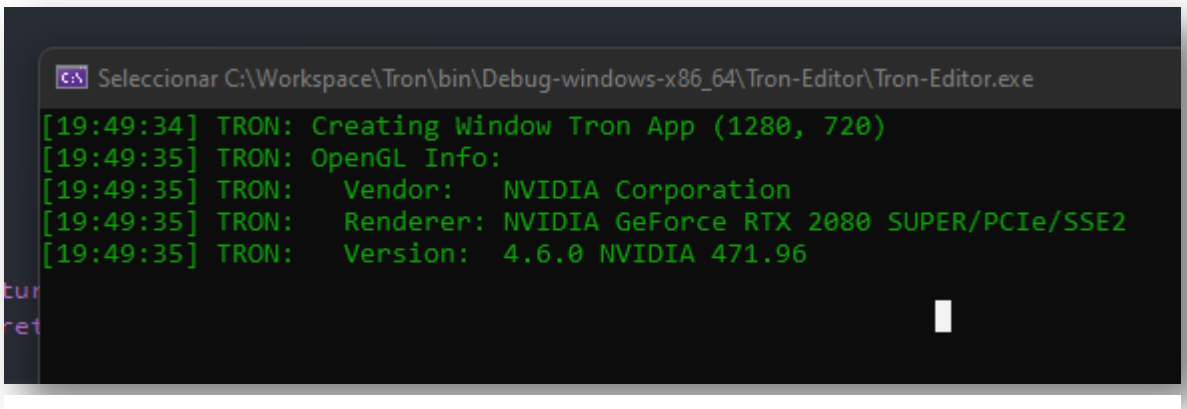

Ilustración 2.8 Ejemplo de Log usando spdlog

En Tron Core, se usa este sistema de logs tanto para mostrar información sobre el sistema al inicio de la ejecución, hasta mostrar errores de compilación de shaders.

2.2.3 - Implementación GLFW

Inicialmente, se planteó este proyecto como multiplataforma (es decir, que permite ser ejecutado tanto Windows, Linux o Mac) como multi-api (o lo que es lo mismo, capaz de ejecutar la misma aplicación usando diferentes APIs gráficas, como son OpenGL, DirectX o Vulkan). Al estar este proyecto planteado como un prototipo, debíamos ajustarnos a unos requisitos alcanzables para llegar a un MVP (Minimum Viable Product, o producto mínimo viable) por lo tanto, se optó por desarrollar este proyecto en Windows con OpenGL. En un futuro, se proyectará implementar diferentes framework de ventanas y de gráficos compatibles para otras plataformas.

Dicho esto, ya que se ha optado por usar la api gráfica OpenGL, se planteó usar GLFW como gestor de ventanas. GLFW es una biblioteca gráfica que nos ayuda a crear con una API muy sencilla, contextos gráficos, ventanas y el manejo de eventos. Se ha utilizado GLFW porque durante el grado, es el framework que principalmente se ha utilizado, y realizar la implementación básica es más rápido que investigar cualquier otro framework.

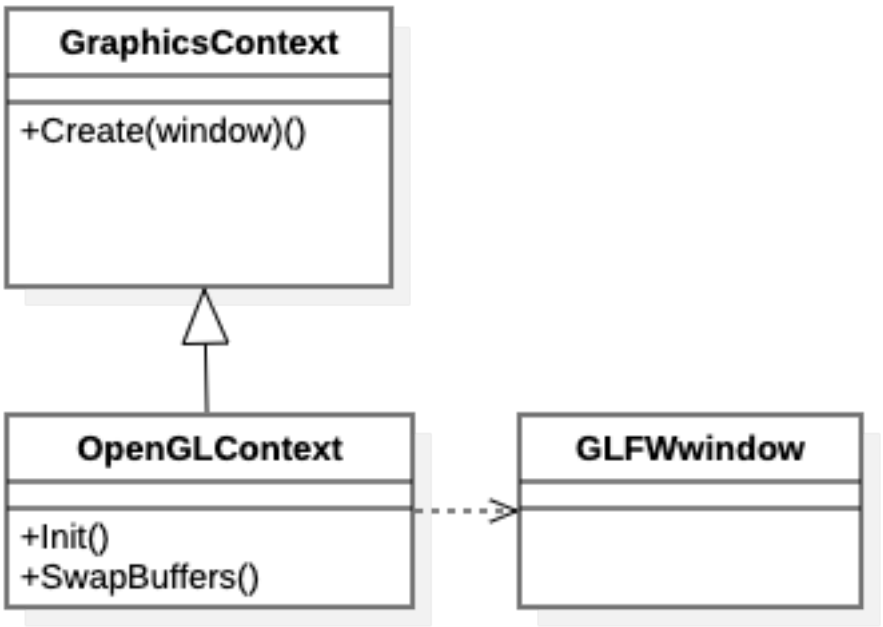

Ilustración 2.9 Diagrama de clases de GraphicsContext

Para poder llevar a cabo todo esto, se ha creado un GraphicsContext, que no es ni más ni menos que una abstracción de las configuraciones específicas de plataforma y de API que necesitan ser implementadas. La clase padre GraphicsContext nos sirve como interfaz para crear todos los contextos gráficos que el Core necesite. En este caso, se ha implementado este prototipo para OpenGL, por lo que se ha implementado un contexto OpenGL que se encargará de:

- Inicializar su contexto y activar todos los flags y configuraciones que necesite
- Activar ciertas funcionalidades abstractas, como la implementación de SwapBuffers, en este caso.

A su vez, cada contexto gráfico podrá acoplarse sin problema con otras bibliotecas, como GLFW en este caso, para llevar a cabo toda su inicialización.

La ventaja de usar esta arquitectura es desacoplar completamente el resto del sistema de cada uno de las implementaciones de plataforma y de API, obteniendo una legibilidad y organización mucho mayor.

2.2.4 - Layers

Otro concepto clave en la arquitectura de la aplicación es el concepto de “capas” de **Layers.** Las **Layer** se pueden definir como un contexto gráfico, paralelas entre ellas, que mantienen un ciclo de vida independiente en la aplicación. Son una forma de abstracción más que incluye el sistema, de forma que se pueden crear de forma modular comportamientos en forma de “plugins”, que pueden activarse y desactivarse a placer.

El concepto detrás de esto es el siguiente. Pongamos la situación que tenemos un equipo de personas trabajando en el motor al mismo tiempo. Imaginemos, que necesitamos 3 sistemas en total en la aplicación. Un sistema de UI, que se encargará de mostrar información en pantalla en forma de texto y símbolos; un runtime que renderizará todos los objetos de una escena; y finalmente un sistema de sonido, que permitirá reproducir en tiempo real música y efectos de sonido.

La forma clásica de realizar esto sería creando una arquitectura para cada uno de los sistemas, creando clases para cada uno de ellos, e instanciándolos en el hilo principal de ejecución de la aplicación. Problemas afrontando este método:

- Cualquier modificación en el código común entre los integrantes del equipo (en este caso, el ciclo de vida principal) puede romper el código de otro compañero.
- Acoplamiento indirecto, ya que necesitas saber cuando y cómo se instancian cada uno de esos sistemas en el ciclo de vida, para no solapar comportamientos
- Ofuscación de código. Resulta más complicado resolver problemas y "bugs" en el código si todos los sistemas están acoplados en el mismo sitio.

Ahora, usando estos mismos sistemas, utilizando el sistema de Layers, la tarea se simplifica mucho. Cada sistema se implementa en una Layer independiente, por lo tanto, no tenemos ningún tipo de acoplamiento en el código de cliente. Solo existe acoplamiento con el código de la propia API Tron, que es totalmente factible. Además, separamos cada sistema en sus propias clases, por lo que el trabajo en equipo se vuelve mucho más seguro, ya que el código común entre las "capas" es Tron, y este no será modificado por ningún compañero. Y por último, soluciona la claridad de

Ilustración 2.10 Ejemplo de uso de Layers en Tron

código, ya que cada capa se añadirá al stack de capas, de forma ordenada, y la funcionalidad está implementada en su propia clase, por lo que resolver bugs se convierte en una tarea mucho más eficiente y fácil de resolver.

En este ejemplo (véase ilustración 2.10) se implementa en pseudocódigo el ejemplo comentado anteriormente. La interfaz *Application* ofrece un método llamado *PushLayer()* que añade cada capa a una pila de capas (denominada *LayerStack*). El orden de inserción de cada capa es importante, ya que determina el orden en el que se renderizará cada una de las capas. El orden de render es FIFO, por lo tanto, el primero en ser insertado será el primero en renderizarse.

Este sistema además permite ampliar la aplicación con nuevas funcionalidades de forma muy fácil y segura, porque tan solo basta con insertar la nueva capa en el stack, y Tron será capaz de realizar todo el ciclo de vida automáticamente.

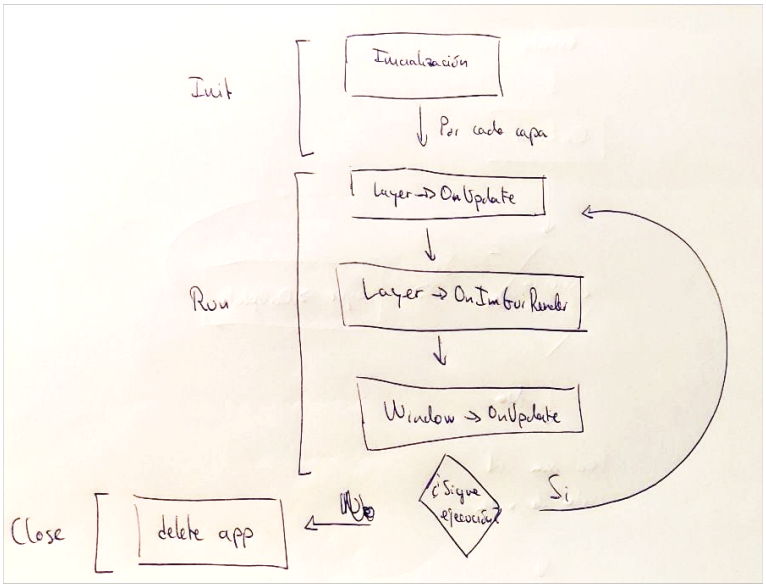

Ilustración 2.11 Diagrama de flujo del Ciclo de vida

2.2.5 - Eventos

En cualquier motor gráfico moderno, el manejo de eventos puede a llegar ser una tarea bastante compleja de implementar. Al fin y al cabo, un motor gráfico tiene que ser capaz de interactuar de forma rápida y eficaz con los eventos de entrada, y

tratar de sacar una respuesta en el menor tiempo posible. En este proyecto se ha optado por una solución intermedia, balanceando facilidad de uso con eficiencia y rapidez.

En otros motores gráficos más avanzados y completos, como pudiera ser Unreal Engine, el sistema de eventos está basado en blueprints, que generalizan la forma de programación, o el de Unity, basado en scripting. Sin embargo, e imitando la forma de trabajar de los navegadores web, se ha implementado un sistema de eventos de suscripción, en el que el sistema recibe los eventos y los "emite" a todos los suscriptores que estén escuchando cada tipo de evento. Estos eventos se propagan de abajo hacia arriba en el stack de capas, haciendo que cada capa pueda recibir el evento y pueda decidir que hacer.

Aquí surge un problema. Si todos los eventos se propagan a cada capa, ¿eso quiere decir que un mismo evento puede despacharse varias veces en distintas capas? En efecto, el principal inconveniente de este sistema es que cada capa es responsable de bloquear o no el evento, para que siga propagándose por el stack de capas. Aunque, las ventajas de hacerlo de esta forma es que tienes el control total sobre el evento, y puedes crear interacciones muy complejas de una forma muy sencilla.

Es decir, imaginemos que llega un evento de "*pulsado de tecla"* del teclado. El sistema propagará el evento a todas las capas que haya incluidas en el *LayerStack* en orden inverso, desde la última capa, hacia la primera. Imaginemos que en nuestro LayerStack tenemos en primer lugar una capa de renderizado principal, y al final una capa de interfaz. Primero, el sistema propagará ese evento a la última capa en el stack, en este caso la UI. Esta capa tratará ese evento con la acción que corresponda (abrir un menú, destruir una entidad etc…) y en ese momento, decidirá si se propaga a la siguiente capa o no. Es el propio código cliente el que decide si esa tecla pulsada necesita ser enviada al runtime o no, por lo que podemos sesgar muy bien qué eventos pasan y cuales no, haciendo que el sistema ni siquiera inicie esos métodos de evento y así, haciendo la aplicación más rápida, comparándolo con la opción clásica, que sería mandar a todos los suscriptores el evento, pasando de ejecutarse esos métodos

un 100% de las veces, a solo la cantidad justa y necesaria. Por así decirlo, se ha creado un sistema de eventos selectivo.

2.2.6 - ImGui

Una parte importante de este motor gráfico quería que fuera la capacidad de mostrar información sobre el sistema en cualquier momento. Ya sea por consola, usando el Logger (véase capítulo 2.2.2) o por el propio viewport. Para ello, era necesario implementar las bases de un sistema de UI embebido en el motor, para ofrecer esa funcionalidad a cualquier usuario que use el motor. Por ello, se decidió usar una biblioteca OpenSource llamada ImGui, que no es más que un pequeña API para crear interfaces gráficas en C++, de forma muy rápida.

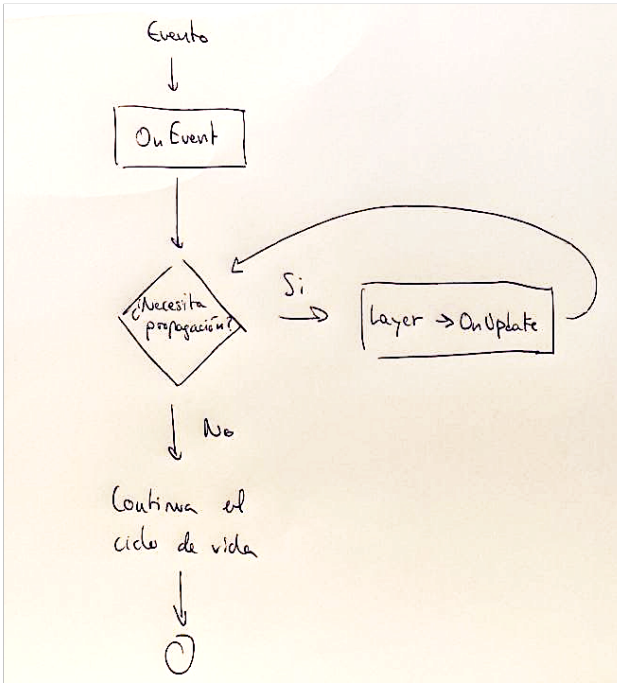

Ilustración 2.12 Diagrama de flujo de eventos

Se optó por ImGui porque, dentro de la comunidad de creación de motores gráficos, es una biblioteca bastante popular entre los programadores por su sencillez de uso, y por su potencia (basado en la actividad del repositorio de GitHub y los stars del proyecto, que suelen denotar actividad por parte de la comunidad y seguimiento de bugs). Se estudiaron otras alternativas como Qt, o MiniGui, pero durante el desarrollo y prueba de estas bibliotecas surgieron varios problemas, como la poca

compatibilidad con el sistema ya creado, o sugerían cambios drásticos en la arquitectura.

En esta primera iteración, se crearon las bases e implementaciones básicas de la biblioteca, generando una pequeña interfaz de prueba para mostrar datos del Renderer.

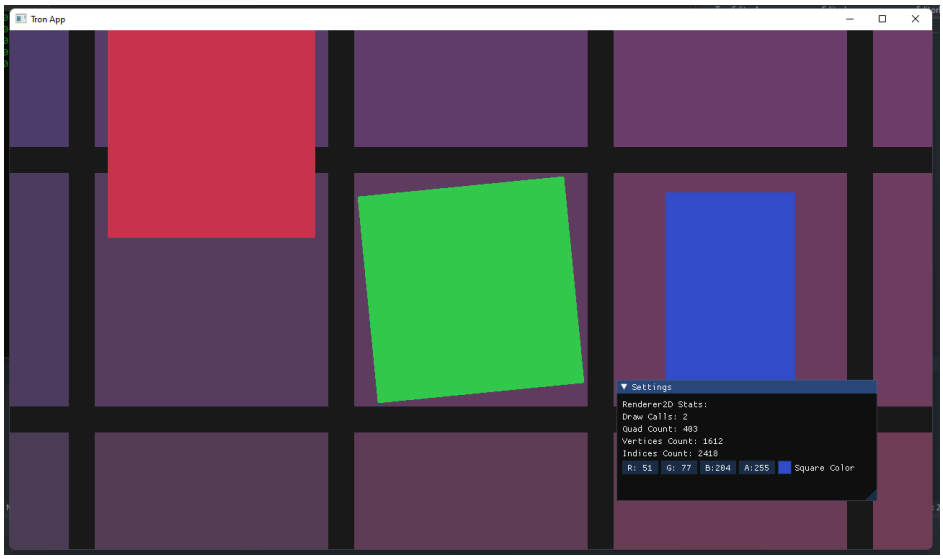

Ilustración 2.13 Implementación básica de ImGui

2.2.7 - Shaders

Otro aspecto fundamental en una aplicación gráfica, y sobretodo en un motor gráfico donde la eficiencia y la rapidez del sistema son críticos, es el manejo de Shaders, o sombreadores.

Un shader no es más que el código fuente que la GPU debe ejecutar durante el proceso de rendering. Este pequeño “programa” es la clave para renderizar cualquier objeto en una escena, ya que es el encargado de pintar, pixel a pixel, cada color (o ausencia de él) en el viewport.

Existen distintos tipos de shaders: desde los *Vertex Shaders* (o sombreadores por vértice) que se ejecutan una vez por vértice de la geometría, los *Geometry Shaders* (sombreadores de geometría) que se ejecutan por cada grupo de vértices, hasta los *Fragment Shaders* (o *Pixel shaders*, dependiendo la API utilizada) que son los encargados de asignar un color a cada "fragmento" del viewport. Actualmente, el motor soporta los shaders mínimos para la ejecución. El *Vertex Shader* y el *Fragment Shader*.

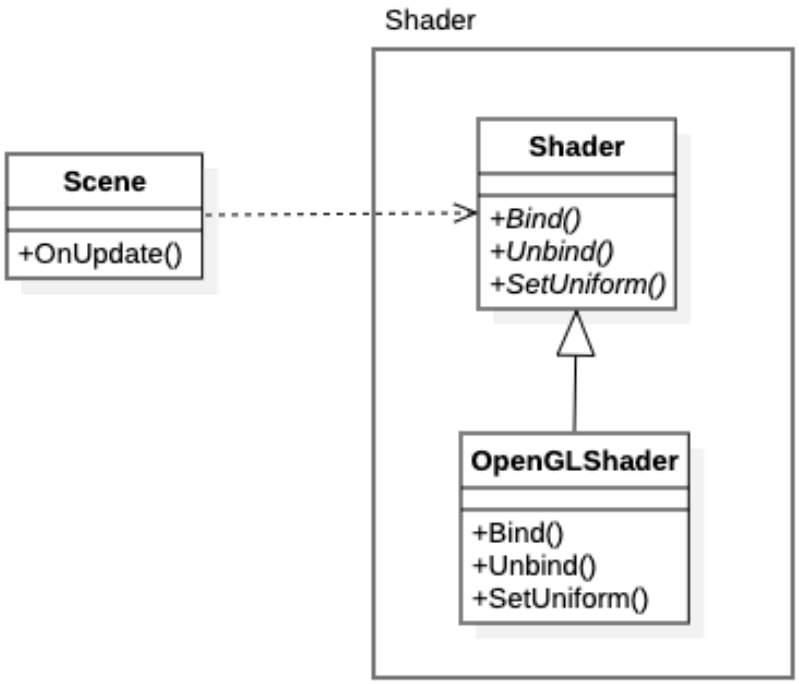

Ilustración 2.14 Diagrama de clases de la arquitectura Shader

A la hora de diseñar la arquitectura de shaders en Tron, se han seguido los mismos requisitos que el resto de la aplicación: Encapsulación, limpieza de código y sencillez de uso. Por ello, y por la naturaleza multi-api del proyecto, un Shader ha de ser implementado por cada API gráfica que soporte el sistema. Actualmente, al sólo trabajar con OpenGL de momento, como ya se comentó, solo han sido necesarias implementar las clases correspondientes a OpenGL.

Hay que destacar que, como se ha implementado un sistema de Escenas, y además se usa un sistema ECS, el manejo de los Shaders se ha ocultado

completamente al cliente, por lo que la sencillez a la hora de programar una aplicación con Tron de este tipo es significativa.

Hay distintas formas de manejar assets de shaders en un motor gráfico. En este proyecto se ha optado por usar un único fichero para contener cada uno de las distintas fases del rendering. Así, reducimos la cantidad de ficheros en un proyecto y, bajo mi punto de vista, hacemos que todo esté más organizado.

```glsl
// Flat Color shader

#type vertex
#version 330 core

layout(location = 0) in vec3 a_Position;

uniform mat4 u_ViewProjection;
uniform mat4 u_Transform;

void main() {
    gl_Position = u_ViewProjection * u_Transform * vec4(a_Position, 1.0);
}

#type fragment
#version 330 core

layout(location = 0) out vec4 color;

uniform vec4 u_Color;

void main() {
    color = u_Color;
}
```

Ilustración 2.15 Shader en un único fichero glsl

2.2.8 - Contexto de renderizado y Renderer

En un motor gráfico, realmente no hay ningún componente que sea más importante que otro. Al final, un motor gráfico es un conjunto de sistemas funcionando entre si. Aunque quizás, el Renderer y todo su contexto gráfico ocupe un lugar muy especial en esta arquitectura, ya que es el componente del que parten el resto de

sistemas. Si un Renderer está mal diseñado, posiblemente el desarrollo de los demás sistemas sea mucho más laborioso.

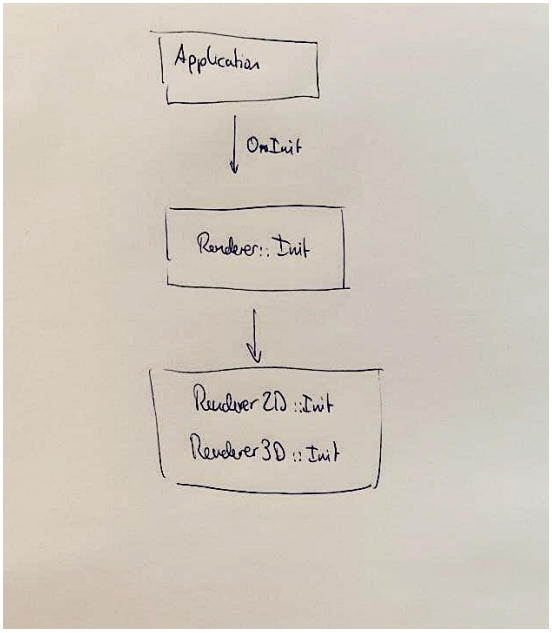

Ilustración 2.16 Diagrama de flujo de inicialización del Renderer

En Tron, el Renderer es el que guía todo el proceso de rendering de la aplicación. Digamos que es el director de cada una de las etapas del motor. Inicialmente, se planteó que este motor gráfico tuviera separados dos tipos de "Renderer" en función del problema a tratar, teniendo así un **Renderer2D**, encargado de realizar el proceso de rendering en aplicaciones con visualización exclusivamente 2D (es decir, usando los ejes X, e Y exclusivamente) y un **Renderer3D**, que se encargaría de realizar el proceso de rendering común en una aplicación gráfica.

Esta idea se planteó así para poder diferenciar claramente unos procesos de otros, reduciendo así la complejidad de código a la hora de ser desarrollado y facilitando su mantenimiento posterior. Al final, ambos renderer usan las implementaciones de cada API, de tal forma que comparten código, pero son transparentes el uno con el otro.

Otro concepto a que se ha introducido en esta etapa es el denominado *RenderCommand*. Un *RenderCommand* es un tipo especial de acción a nivel global, que realiza tareas de alto grado de abstracción, como pudieran ser hacer un *clear* del *viewport* o reiniciar la escena. El *RenderCommand* está disponible de forma global en la aplicación, por lo que pueden ser usados transparentemente independientemente de la API gráfica usada. Esto se consigue implementando una abstracción de la API gráfica llamada RendererAPI, que no es más que una pequeña clase ayudante que nos indica qué api se está usando en cada momento, y el renderer es capaz de, en tiempo real, realizar las funciones específicas de la api activa.

```
RenderCommand::SetClearColor({0.0f, 0.0f, 0.0f, 1});
RenderCommand::Clear();

Renderer::BeginScene();

... Ejecutamos nuestro código de render aquí

Renderer::BeginScene();
```

Ilustración 2.17 Ejemplo de uso de RenderCommand

Veamos un ejemplo (véase ilustración 2.17). Imaginemos que queremos iniciar un proceso de rendering, y queremos dibujar en pantalla un cuadrado verde. Para ello, lo primero que debemos hacer es hacer un Clear() de la pantalla. Aquí entran en juego los RenderCommand. Simplemente llamamos al método estatico RenderCommand::Clear() y sin indicar nada más, el renderer ejecuta el código correspondiente de OpenGL (que recordemos, es la única API activa en el momento) y ejecuta a su vez el Clear() de OpenGL. Esto es similar para cualquier otra API gráfica que tengamos implementada en el motor.

Es muy conveniente esta forma de trabajar, sin tener que parar la ejecución del programa, cambiar de API gráfica y ver los resultados inmediatamente sin ningún tipo de interrupción.

2.2.9 - Pruebas para esta iteración

Inicialmente, y cada vez que una issue era completada, se realizaba una prueba de la funcionalidad completa, para evitar que el repositorio contenga errores en el código.

De esta forma, el repositorio de código siempre mantiene una versión de producción y otra versión staging, ambas libres de errores (o al menos, lo suficientemente probadas para que desde cualquier punto se pudiera hacer rollback a iteraciones anteriores).

Inicialmente, se creó una aplicación de pruebas, implementada con el motor gráfico Tron, llamada Sandbox (dentro del la solución de Visual Studio se pueden encontrar los 3 subproyectos).

Dicho esto, para realizar las pruebas se tuvieron en cuenta estas consideraciones:

- Se debe de probar cada funcionalidad añadida de forma completa, intentando realizar un ejemplo de cada caso de uso particular de esa funcionalidad
- Se debe testear en la medida de lo posible, la eficiencia de la nueva funcionalidad.
- La nueva funcionalidad no puede romper cambios anteriores.

Teniendo en cuenta estas consideraciones, las pruebas que se realizaron fueron las siguientes:

1. Creación de una Application (en este caso llamado Sandbox). Con este concepto probamos el ciclo de vida de la aplicación, su entry point, y su destrucción.

2. Implementación de una *Layer*, llamada *Sandbox2D*. En esta capa, se programarán una serie de objetos para probar la API renderer (tantos como fueran necesarios para probar cada método).
3. Se añadirá la Layer dentro de la aplicación, para comprobar que el LayerStack funciona, y todos los eventos generados por el usuario se propagan correctamente.
4. Se creó un pequeño panel de estadísticas de ejecución, usando ImGui. En la ilustración 2.18, se puede observar en la parte superior derecha el pequeño panel de ImGui, mostrando datos relevantes de rendering, y algún que otro control para manejar el color de uno de los cuadrados.

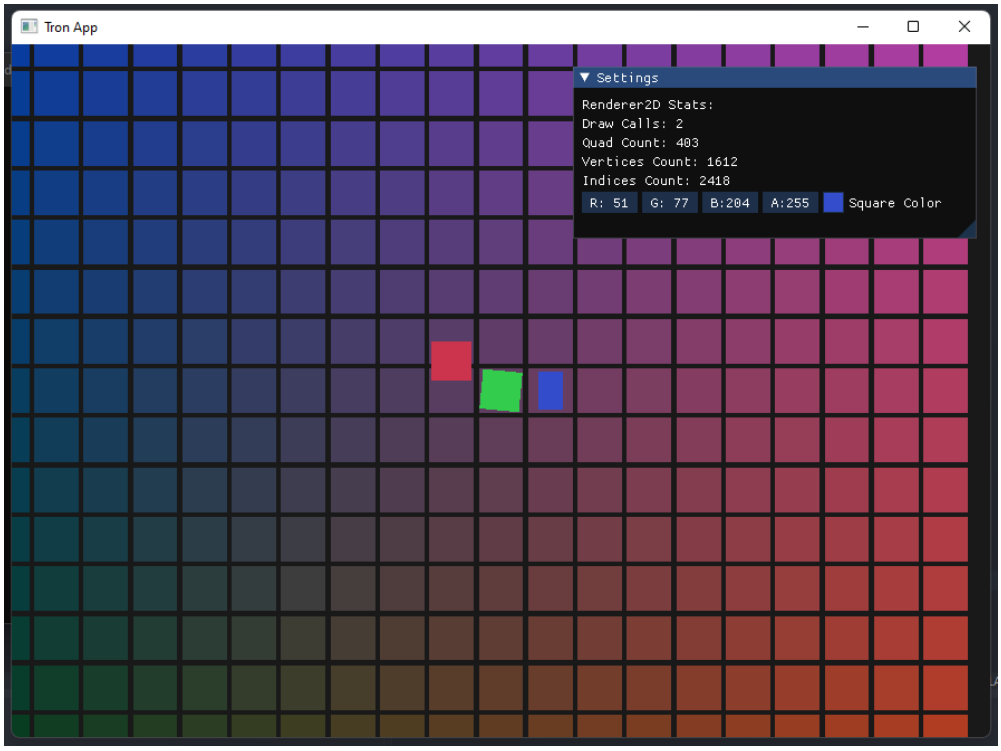

Ilustración 2.18 Batería de pruebas para la primera iteración

2.3 - Segunda iteración: Tron Editor

En esta segunda iteración del proyecto, se trabajará sobre el otro subproyecto del sistema, el concepto de Editor del motor gráfico. Este concepto se asemejaría a cualquier motor gráfico comercial, que todos cuentan con un editor embebido en el

propio motor gráfico, para hacer que el programador sea más eficiente usando la herramienta.

No obstante, también se ha seguido implementado features tan importantes como el Profiling, para tomar datos de efecución y tiempo, para saber qué partes del sistema necesitan más mejora y eficiencia; la implementación del ECS (Entity component System) usando la biblioteca OpenSource EnTT, y otros aspectos como control de cámaras y retoques de UI/UX.

2.3.1 - Visual Profiling

En cualquier software que necesite ser eficiente por naturaleza, como en cualquier aplicación gráfica, es fundamental tener un feedback sobre lo eficiente que es tu sistema.

En este proyecto, se ha implementado una sencilla, pero poderosa herramienta para hacer visual profiling, o medición de rendimiento de una forma totalmente visual e interactiva.

Para ello, se utilizará una característica que incluye Google Chrome, llamada Tracer (se accede desde chrome://tracing). Actualmente, se ha implementado una nueva versión de esta herramienta llamada *Perfetto*, que cumple la misma funcionalidad, pero con una interfaz más amigable.

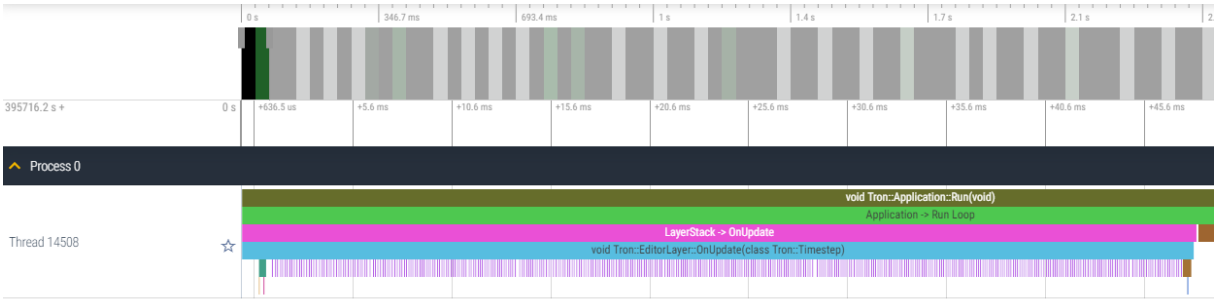

Ilustración 2.19 Visualización del profiling de una ejecución de un frame del Renderer

Para implementar esta funcionalidad, se creó la clase *Instrumentor*, que simplemente se encarga de recoger el tiempo desde que se inicia su contador, hasta que finaliza. Además, guarda en disco los datos en formato JSON (JavaScript Object Notation) compatible con el sistema de tracing de Google Chrome.

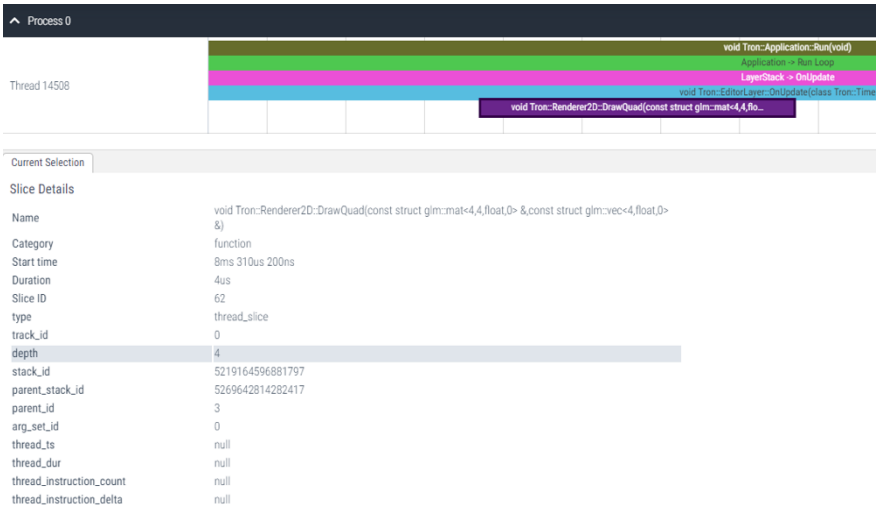

Ilustración 2.20 Estadísticas concretas de un método en el proceso de Rendering

Como se puede observar en la ilustración 2.19, de un solo vistazo, podemos ver todas las funciones que nuestro motor gráfico ha realizado durante un frame. Esto es posible porque por todo el motor, se han incluido estas mediciones de tiempo, indicando un nombre para cada uno de los métodos llamados.

2.3.2 - Implementación ECS con EnTT

En esta feature, se implementó una base solida con la que poder ir iterando e ir ampliando las funcionalidades básicas de un sistema ECS. Para ello, se ha utilizado la biblioteca EnTT que nos proporciona una API muy completa y simple para manejar entidades, componentes y sus sistemas adheridos.

Este proyecto, al ser un prototipo, se han cubierto las funcionalidades básicas y más importantes de un sistema ECS, como son las de añadir una entidad, eliminar una entidad, añadir un componente a una entidad, y eliminar un componenten de una entidad. En futuras iteraciones del proyecto, se irán implementando nuevas características que doten de mayor control al sistema sobre sus entidades.

Una entidad en Tron, es una pequeña clase, que está siempre asociada a una escena (ya que no tendría sentido que existiera una entidad sin escena) y contiene unos métodos auxiliares para ayudarnos a añadir componentes.

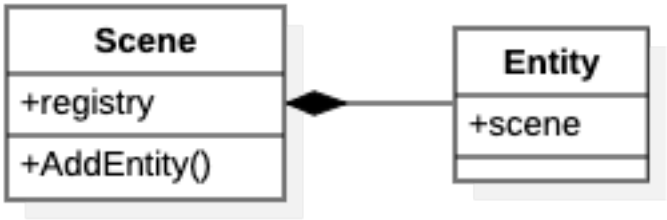

Ilustración 2.21 Diagrama de clases de una entidad

A su vez, una entidad puede tener un número ilimitado de componentes, que son pequeñas clases que le otorgan a la entidad de atributos y funcionalidades modulares. En Tron, todas las entidades por defecto contienen un TagComponent, les otorgan con un nombre para poder ser diferenciados dentro del motor.

```
struct TagComponent {
    std::string Tag;

    TagComponent() = default;
    TagComponent(const TagComponent&) = default;
    TagComponent(const std::string& tag)
        : Tag(tag) {}
};
```

Ilustración 2.22 Ejemplo de componente tipo Tag

Luego, el renderer es el encargado de añadir la funcionalidad (sistema) a cada entidad que contenga unos componentes específicos. Por ejemplo, al ejecutarse el OnUpdate() cada frame, el Renderer buscará todas las entidades con un TransformComponent y le aplicará transformaciones en función de los eventos recibidos.

2.3.3 - Desarrollo de UI/UX

Durante el desarrollo de esta iteración, también se han ido creando los diferentes componentes en ImGui para crear una pequeña interfaz a modo de editor (basándome en la forma de trabajar de motores gráficos comerciales, como Unity) que sea capaz de realizar los diferentes requisitos detallados en el capítulo 1.5.2 del proyecto.

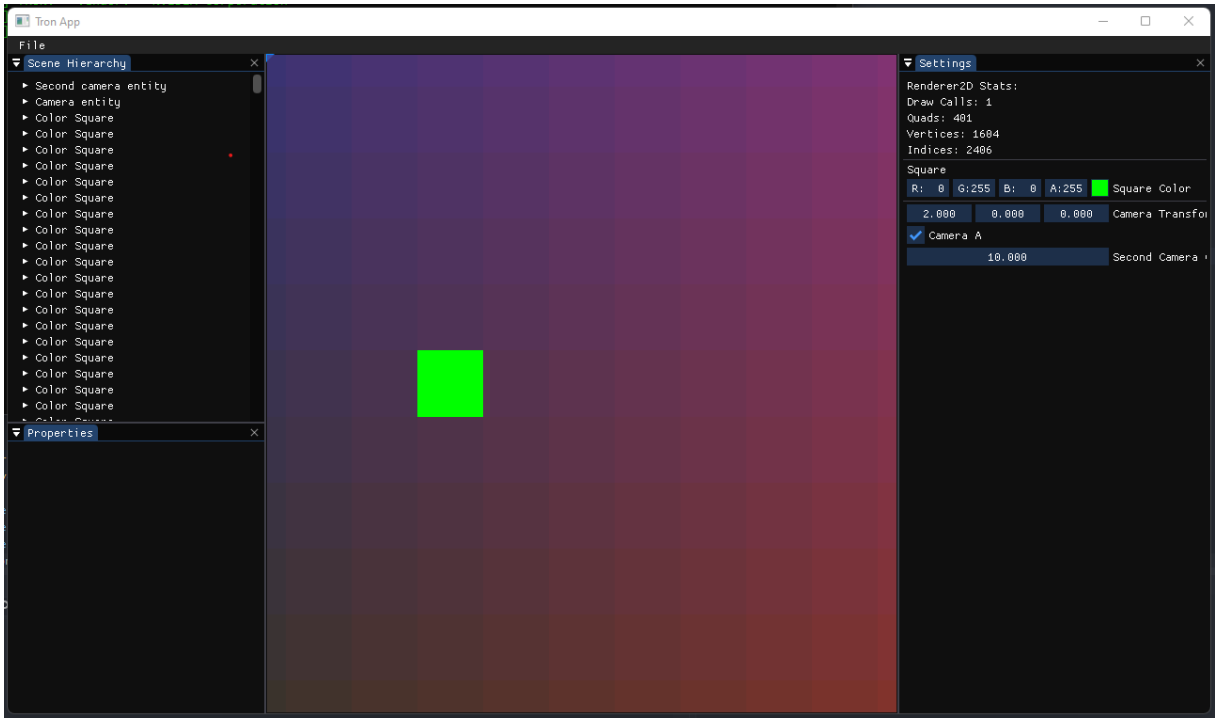

Ilustración 2.23 Interfaz final de Tron Editor

2.3.4 - Pruebas para esta iteración

De forma similar a las pruebas realizadas en el apartado 2.2.9, se han realizado una batería de pruebas de forma que las nuevas características se hayan implementado de forma segura y sin provocar fallos en características antiguas.

Para esta iteración, se volvieron a implementar las pruebas anteriores, para comprobar que todo sigue en orden, añadiendo esta serie de pruebas extra:

1. Creación de una escena. Se añadieron una serie de entidades con sus componentes para renderizar unos cuadrados de ejemplo (véase ilustración 2.23)
2. Creación de texturas de ejemplo.
3. Creación de un script para controlar una entidad con componente de tipo CameraComponent.
4. Se creó un despachador de eventos para detectar el scroll del ratón para aumentar / disminuir el zoom de la cámara principal.

3 - CONCLUSIONES Y TRABAJOS FUTUROS

Como desarrollador del proyecto y de la aplicación, la forma de planificación del proyecto ha sido la adecuada, ya que cada tarea estaba bastante bien diferenciada, con la cantidad justa de trabajo para que supusiera una carga excesiva de trabajo, cumpliendo con los requisitos expuestos en la definición del proyecto.

Además, al trabajar usando una metodología iterativa, en la que por cada tarea, se realiza una prueba de esa funcionalidad, ha hecho que durante la implementación del proyecto, se hayan descubierto todo tipo de pequeño fallos que de otra forma hubieran sido muy tediosos de resolver, hasta tal punto de incluso no poder encontrarlos.

El proyecto, aún siendo un prototipo, tiene bastante potencial en convertirse una herramienta realmente útil para un programador gráfico, haciendo a su vez que el desarrollo de aplicaciones gráficas tenga una curva de aprendizaje menos elevada y atrayendo a nuevos programadores a descubrir este mundo.

Para acabar, algunos de las características que se podrán ir incluyendo en el sistema pueden ser, entre muchas, alguna de las siguientes:

- Renderer3D, para trabajar directamente con problemas en 3D
- Sistema de físicas

- Sistema de audio
- Implementación de otras apis gráficas, como Vulkan o DirectX
- Implementación de la aplicación para otras plataformas como Linux o Mac.

4 - APÉNDICES

4.1 - Instalación y configuración del sistema

Para ejecutar el proyecto, se puede realizar de dos formas: A través de Visual Studio, o desde el ejecutable adjuntado con la memoria.

- **Desde Visual Studio:** Lo primero que debemos hacer es dirigirnos a la carpeta contenedora del código fuente, y ejecutar el archivo scripts > Win-GenProjects.bat. Este fichero creará el proyecto para Visual Studio a partir de la configuración de tu máquina. Una vez ejecutado, abrimos el archivo Tron.sln, y abrirá Visual Studio con el proyecto. Una vez dentro, seleccionamos el perfil que querramos usar (recomendado: Release) y hacemos click en Depurador local de Windows.
- **Desde el archivo compilado:** Dirigete a la carpeta del proyecto, y dentro de este, encontrarás un archivo .exe. Simplemente ejecútalo y aparecerá la aplicación.

5 - BIBLIOGRAFÍA

1. **NVIDIA GPU Gems**. [En línea] https://developer.nvidia.com/gpugems/gpugems2/copyright.
2. **NVIDIA. Ray Tracing**. [En línea] https://www.nvidia.com/es-es/geforce/rtx/.
3. **League, Sebastian. Conceptos sobre programación gráfica.** [En línea] https://www.youtube.com/c/SebastianLague.
4. **LearningOpenGL. Información y aprendizaje sobre OpenGL**. [En línea] https://learnopengl.com/Introduction.
5. **Games, Epic. Unreal Engine**. [En línea] https://www.unrealengine.com/en-US/.
6. **Unity. Unity**. [En línea] https://unity.com.
7. **CryTek. CryEngine**. [En línea] https://www.cryengine.com.
8. **TheCherno. Aprendizaje sobre ECS.** [En línea] https://www.youtube.com/watch?v=Z-CILn2w9K0.
9. **Falconer, Dylan. Implementación básica de ECS**. [En línea] https://www.youtube.com/watch?v=s6TMa33niJo.
10. **Microsoft. Visual Studio.** ***Visual Studio: IDE y Editor de Cógido****.* [En línea] 2021. https://visualstudio.microsoft.com/es/.
11. **Axosoft. GitKraken**. [En línea] 2010. https://www.gitkraken.com.
12. **Conservancy, Software Freedom. Git.** [En línea] 2021. https://git-scm.com.
13. **Google. Google Chrome**. [En línea] 2021. https://www.google.com/chrome/.
14. **orconut. ImGui**. [En línea] 2021. https://github.com/ocornut/imgui.
15. **skypjack. EnTT**. [En línea] 2021. https://github.com/skypjack/entt.
16. **Stroustrup, Bjarne. C++**. [En línea] 1979. https://github.com/skypjack/entt.
17. **Adobe. Adobe Photoshop**. [En línea] 2021. https://www.adobe.com/es/products/photoshop/landpb.html.
18. **TheCherno. Conceptos generales sobre arquitectura de motores gráficos.** *Youtube.* [En línea] 2021. https://www.youtube.com/c/TheChernoProject.
19. **Reddit. Bibliotecas populares de GUI para C++**. [En línea] https://www.reddit.com/r/cpp/comments/babfl5/a_pretty_big_list_of_c_gui_libraries/.
20. **Wikipedia. Shader**. [En línea] https://es.wikipedia.org/wiki/Sombreador.
21. **Lázaro, Mario García**. Porcentaje de uso de motores gráficos. [En línea] https://docs.google.com/spreadsheets/d/1ECZdAY8Ut9Erv0jIrBRLyF1lK0iAfnblynFtcg2ukbY/edit#gid=0.