Editor y motor de juegos 2D para no programadores

TRABAJO DE FIN DE GRADO

Pablo Fernández Álvarez
Yojhan García Peña
Iván Sánchez Míguez

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

Septiembre 2023

Documento maquetado con TEXIS v.1.0+.

Editor y motor de juegos 2D para no programadores

Memoria que se presenta para el Trabajo de Fin de Grado

Pablo Fernández Álvarez, Iván Sánchez Míguez, Yojhan García Peña

Dirigida por el Doctor

Pedro Pablo Gómez Martín

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

Septiembre 2023

Agradecimientos

Queremos expresar nuestro sincero agradecimiento a todos aquellos que han contribuido de manera significativa en nuestro desarrollo como estudiantes. En primer lugar, extendemos nuestro reconocimiento a nuestros respetados profesores, cuya orientación y sabiduría han sido fundamentales para guiarnos a lo largo de este proceso académico. Sus conocimientos compartidos y su apoyo constante nos han permitido crecer y prosperar en este proyecto. Además, deseamos mostrar nuestro agradecimiento a nuestras familias, cuyo inquebrantable respaldo y ánimo han sido una fuente inagotable de motivación. Su apoyo emocional y comprensión han sido esenciales para superar los desafíos y celebrar los logros. Nuestro más sincero agradecimiento a todos aquellos que han estado a nuestro lado en este viaje, ayudándonos a alcanzar este hito académico.

Resumen

El desarrollo de videojuegos es un campo que ha experimentado una evolución significativa a lo largo de los años. Inicialmente, los videojuegos eran programas simples con un comportamiento básico y con gráficos muy reducidos pero con el tiempo, y debido en parte a la evolución de la teconología, han aumentado considerablemente en complejidad y alcance.

Un motor de ejecución es un software que proporciona la teconología necesaria para desarrollar el gameplay de un videojuego. Es un conjunto de librerías (gráficos, audio, físicas...) agrupadas de forma coherente que abstraen la teconología al desarrollador para que pueda centrarse en el desarrollo del videojuego.

Hoy en día, el desarrollo de videojuegos es una tarea compleja. Por un lado, se puede llevar a cabo desde cero, es decir, programando la teconología que requerirá el videojuego y posteriormente desarrollándolo y, por otro lado, usando un motor de ejecución que proporcione esa tecnología independiente a las necesidades concretas del videojuego a desarrollar.

En algunos casos, con el fin de ayudar al desarrollo, se incorporan los editores de videojuegos. Los editores simplifican el desarrollo al unir la creación de elementos de juego, definición de comportamiento, depuración y generación de versiones ejecutables, entre otras cosas, en una sola herramienta visual. Esto evita tener que comunicarse directamente con el motor de ejecución a través de la programación.

Esto supone una gran ventaja a los desarrolladores experimentados pero motores como Unity o Unreal Engine pueden albergar demasiada complejidad para personas sin experiencia en programación, incluso aunque su objetivo sean juegos sencillos en 2D. Además, son sistemas enormes que al pretender servir para hacer cualquier juego tienen mucha funcionalidad variada y crean versiones ejecutables con gran cantidad de datos innecesarios. Si se quieren hacer juegos pequeños y sencillos, el peaje que se paga es muy grande.

Aquí es donde entra en juego nuestro trabajo. Vamos a hacer un motor con su editor para hacer juegos pequeños en los que la experiencia de desarrollo sea equivalente a la de los editores de motores más grandes, pero

que esté centrado en el desarrollo de juegos 2D más pequeños y suponga una carga mucho menor en ejecución y a la hora de generar las versiones ejecutables. Esto abrirá las puertas a nuestro motor a desarrolladores con poca experiencia en programación o incluso perfiles sin experiencia ninguna en desarrollo de videojuegos.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Motivación

El desarrollo de videojuegos ha supuesto un reto técnico y organizativo desde sus inicios. Especialmente a nivel técnico, requiere de la experiencia de ingenieros informáticos capaces de desarrollar la tecnología necesaria que puede demandar un videojuego. Las empresas desarrolladoras pueden escoger entre desarrollar las tecnologías necesarias para desarrollar un videojuego u obtener esas tecnologías ya desarrolladas y comenzar directamente con el desarrollo.

En el primer caso, el incoveniente reside en desarrollar dichas tecnologías de forma dependiente al videojuego para el que se esté desarrollando, es decir, atendiendo a sus características concretas de tal forma que no se pueda reutilizar para el desarrollo de otro tipo de videojuego.

En el segundo caso, el incoveniente reside en la forma de obtener dichas tecnologías, licencias o costes. A la hora de desarrollar esas tecnologías los equipos deben trazar la línea que separa lo que son las tecnologías para desarrollar un videojuego y el propio desarrollo del videojuego. Ésta separación en necesaria para garantizar la reutilización de las tecnologías.

A estas tecnologías se las conoce como motores de ejecución. En general suelen estar constituidas de varias librerías dedicadas a un fin específico como puede ser los gráficos, el audio, las físicas, el input, etc.

En algunos casos, además del motor de ejecución, las empresas desarrolladoras de motores incluyen editores. Los editores son herramientas que simplifican el desarrollo comunicando las acciones del desarrollador al motor. Entre alguna de sus funciones claves destacan la definición de comportamientos, la creación de assets y elementos en el juego, depuración y generación de versiones ejecutables para distintas plataformas.

Existen en el mercado diferentes motores con distintas licencias y ca-

racterísticas. Los más cómodos para el desarrollo son los que tienen editor, pero un editor es muy costoso de desarrollar, por lo que los motores que lo proporcionan son aquellos que tienen una masa crítica de uso muy grande, como para que merezca la pena el esfuerzo de desarrollo del editor. Esto solo ocurre en motores generalistas que permiten realizar juegos de muchos tipos y con muy buena calidad. El precio a pagar por usar esos motores es complejidad, tanto en el uso del editor como en el motor en tiempo de ejecución. Esto no es un problema si el juego desarrollado hace uso de todas las características punteras, pero si se quiere hacer un juego modesto donde se ponga a prueba la jugabilidad y no tanto la tecnología, el peaje a pagar es alto. Esto además supone una brecha para los perfiles con poca experiencia en desarrollo o programción.

La alternativa para hacer juegos más pequeños es hacer uso de motores más simples, pero normalmente no traen editor y eso alarga el proceso de desarrollo. En este punto es donde entra nuestro nuestro trabajo de fin de grado. Vamos a hacer un motor con su editor para hacer juegos pequeños en los que la experiencia de desarrollo sea equivalente a la vivida con editores de motores más grandes, pero que esté centrado en el desarrollo de juegos 2D mucho más pequeños. Esto simplifica el uso del editor y también el tamaño de las versiones ejecutables realizadas.

Las características del motor que queremos hacer son:

- *Ejecución del juego desde el editor*: Esto significa la posibilidad de lanzar el juego desde el editor sin tener que hacer una versión ejecutable o buscar el archivo ejecutable a mano. Además, se podrá imprimir información por consola visible desde el editor ya sea para depurar o consultar errores.

- *Autosuficiencia*: Aporta funcionalidad para manejar recursos, crear escenas y objetos desde el editor y permite la creación de ejecutables finales del juego para su distribución. Con esta característica se busca la mínima dependencia de herramientas externas.

- *Programación visual*: Esta es la parte a destacar de nuestro motor. Debido a la complejidad que presentan algunos motores de la actualidad para desarrolladores principiantes o inexpertos, la programación visual es una herramienta muy útil e intuitiva para crear lógica y comportamiento en el videojuego.

1.2. Objetivos

El objetivo principal es desarrollar un motor de videojuegos 2D autosuficiente con editor integrado y una programación visual basada en nodos.

Además, se podrá pobrar y ver el progreso del videojuego que se esté desarrollando directamente desde el editor y generar versiones ejecutables.

Con esto, los usuarios dispondrán de una herramienta para desarrollar cualquier tipo de videojuego 2D con un nivel de complejidad accesible y una experiencia de usuario agradable e intuitiva.

1.3. Herramientas

Para comenzar, se ha utilizado Git como sistema de control de versiones a través de la aplicación de escritorio GitHub Desktop. Todo el código implementado se ha subido a un repositorio dividiendo el trabajo en ramas.

Enlace al repositorio: `https://github.com/ivasan07/ShyEngine`

El código ha sido desarrollado en el entorno de desarrollo integrado (IDE) Visual Studio 2022 y escrito en C++.

Por último, hemos llevado a cabo la gestión de tareas a través del sistema de gestión de proyectos Trello.

1.4. Plan de trabajo

Nuestro proyecto se dividirá en tres grandes bloques: motor, editor y scripting visual.

A su vez, el trabajo lo dividiremos en cinco fases: investigación y planificación, desarrollo inicial, integración de proyectos y núcleo del desarrollo, mejoras y cierre del desarrollo y pruebas con usuarios.

- *Investigación y planificación*: La primera fase del trabajo consistirá en investigar distintos motores y editores de videojuegos para comprender su funcionamiento y sus distintas arquitecturas. Buscaremos también librerías que se ajusten a las demandas de nuestro proyecto para conseguir un desarrollo cómodo y eficiente. Por último planificaremos la división del trabajo así como la futura integración de cada una de las partes.

- *Desarrollo inicial*: Para esta fase, desarrollaremos el núcleo de cada proyecto. En cuanto al motor, se crearán los primeros proyectos para probar las librerías que se vayan a utilizar y se implementará la arquitectura de juego básica. En cuanto al editor, se creará el proyecto en el que se integrará la librería de interfaces gráfica seleccionada para probar su funcionamiento. Y en cuanto al scripting visual, se comenzará a prototipar el lenguaje.

- *Integración y núcleo del desarrollo*: Una vez desarrollo el núcleo de cada proyecto, los integraremos de manera que tengamos un flujo funcional entre los tres. Es decir, se implementará un flujo de desarrollo sencillo en el que se creará lógica desde el editor y se podrá ejecutar en el motor. Más tarde, se desarrollará todo lo necesario para conseguir un aproximación a la propuesta original. Por lo tanto, ésta es la fase más importante y duradera.

- *Mejoras y cierre del desarrollo*: Con los tres proyectos integrados, continuaremos el desarrollo de las funcionalidades comunes para dejarlo lo más pulido posible antes de las pruebas con usuarios. Esto conllevará las pruebas/desarrollo de algún juego para detectar posibles errores y corregirlos.

- *Pruebas con usuarios*: Se realizarán pruebas con usuarios de diferentes contextos: usuarios con experiencia en programación y usuarios sin experiencia. De esta manera, aprovecharemos su feedback para arreglar posibles errores y pulir detalles que mejoren la experiencia de los usuarios.

Capítulo 2

Estado del arte

Resumen: En este capítulo se proporcionará el contexto necesario para entender en los próximos capítulos cómo ha sido el desarrollo de nuestro trabajo. El capítulo se centra en los videojuegos, cómo se desarrollan, qué herramientas se pueden utilizar para desarrollarlos y ejemplos de las más utilizadas hoy en día. Además, se explicará qué es un motor de videojuegos, qué partes fundamentales lo contienen y las formas que existen para generar comportamiento en los videojuegos.

2.1. Qué es un videojuego

Un videojuego es un juego electrónico en el que uno o más jugadores interactúan mediante un controlador con un dispositivo electrónico que muestra imágenes de video. Este dispositivo, comúnmente conocido como "plataforma", puede ser una computadora, una máquina de arcade, una consola de videojuegos o teléfono móvil, tableta o una consola de videojuegos portátil.

Típicamente, los videojuegos recrean entornos y situaciones virtuales en los que el jugador puede controlar a uno o varios personajes para conseguir un objetivo dentro de unas reglas determinadas. Dependiendo del videojuego, una partida pueden disputarla una o varias personas contra la máquina o bien múltiples jugadores a través de una red LAN o en línea vía Internet. El género de un videojuego se refiere a una categoría o clasificación que se utiliza para describir su estilo, mecánicas de juego, temas y elementos característicos. Algunos de los géneros más representativos son los videojuegos de acción, rol, estrategia, simulación, deportes o aventura.

Un videojuego se ejecuta gracias a un programa de software que es procesado por una máquina que cuenta con dispositivos de entrada y de salida. El programa contiene toda la información, instrucciones, imágenes y audio que componen el videojuego. Va grabado en cartuchos, discos ópticos, dis-

cos magnéticos, tarjetas de memoria especiales para videojuegos, o bien se descarga directamente a través de Internet.

2.2. Cómo se hace un juego

La creación de videojuegos es una actividad llevada a cabo por las empresas desarrolladoras de videojuegos. Estas se encargan de diseñar y programar el videojuego, desde el concepto inicial hasta el videojuego en su versión final. Esta es una actividad multidisciplinaria, que involucra profesionales de la informática, el diseño, el sonido, la actuación, etc. El proceso es similar a la creación de software en general, aunque difiere en la gran cantidad de aportes creativos necesarios. El desarrollo también varía en función de la plataforma objetivo, el género y la forma de visualización (2d, 2.5d y 3d).

El desarrollo ha pasado por muchas fases hasta el estado en el que se ecuentra en la actualidad. En las primeras décadas a partir del nacimiento de los videojuegos, los desarrolladores tenían que diseñar hardware específico para ejecutar los juegos. A partir de la década de 2000 en adelante, la mayoría de los juegos se desarrollan digitalmente, lo que significa que el código y los activos se crean en computadoras en lugar de hardware dedicado.

Normalmente el desarrollo de un videojuego sigue las siguientes fases:

- *Concepto*: En esta fase se tiene una idea a partir de la cual se conformarán los aspectos fundamentales. Se determina el género o géneros del videojuego, cómo será el proceso de juego, y también se constituye un guión gráfico en el que se tratan todo tipo de ideas preconcebidas que pueden ir adaptándose, como por ejemplo el estilo de los personajes, el ambiente, la música, etc. Una vez se sabe qué hacer entonces es el momento de diseñar.

- *Diseño*: Se empieza definiendo los elementos que componen el juego. Se desarrolla la historia, se crean bocetos de guiones para determinar los objetivos, se deciden los personajes principales, el contexto, etc. El objetivo de esta fase es como objetivo generar el Documento de Diseño (GDD) que especificará el arte, las mecánicas y las dinámicas del videojuego.

- *Planificación*: Esta etapa tiene como objetivo identificar las diferentes tareas para desarrollar el videojuego. Se reparte el trabajo entre los distintos componentes del equipo de desarrollo, se fijan plazos de entregas, se planifican reuniones de seguimiento, etc. Normalmente se suele seguir una metodología de desarrollo como Scrum, en cascada (Waterfall), Kanban, desarrollo Lean, etc.

- *Preproducción*: En esta fase se desarrollan prototipos de juego muy

sencillos, personajes, entornos, interfaces, esquemas de control y otros elementos en el juego para validar las ideas y conceptos clave.

- *Producción*: Se llevan por tanto a cabo todas las tareas de la fase de planificación teniendo como guía el documento de diseño: programación, ilustración, desarrollo de interfaces, animación, modelado, desarrollo del sonidos, etc. Si finalmente se logra ensamblar correctamente todas las piezas entonces esta fase culmina. Sin embargo, al igual que en el desarrollo de software tradicional, es muy difícil que todo salga bien a la primera, por lo que se entra en una fase para probar a fondo el videojuego.

- *Pruebas*: En esta etapa se corrigen los errores del proceso de programación y se mejora la jugabilidad a medida que se prueba el juego. Generalmente encontraremos dos tipos: las pruebas alpha, realizadas por un pequeño grupo de personas generalmente involucradas en el desarrollo, y las pruebas beta, realizadas por un equipo externo de jugadores. Las primeras tienen el objetivo de corregir defectos graves y mejorar características fundamentales no contempladas en el documento de diseño, mientras que las segundas se enfocan en detectar fallos menores y perfilar la experiencia de usuario.

- *Distribución/Marketing*: En esta fase se crean las copias del juego ya finalizado y se lleva a las tiendas (ya sean físicas o digitales) para que los jugadores comprarlo.

- *Postproducción/Mantenimiento*: Pese a que el juego esté finalizado y en las manos de los jugadores, su ciclo de vida aún está lejos de terminar. La fase de mantenimiento es el momento de arreglar nuevos errores, mejorarlo, etc. Ésto se hace sacando parches o actualizaciones al mercado.

2.3. Qué es un juego dirigido por datos

2.4. Qué es un motor de videojuegos

El término "motor de juego"surgió a mediados de la década de 1990 en referencia a juegos de disparos en primera persona (FPS) como el increíblemente popular Doom de id Software. Doom fue diseñado con una separación razonablemente definida entre sus componentes de software principales (como el sistema de renderizado gráfico tridimensional, el sistema de detección de colisiones o el sistema de audio) y los activos artísticos, mundos de juego y reglas de juego que componían la experiencia de juego del jugador. El valor de esta separación se hizo evidente cuando los desarrolladores comenzaron

a licenciar juegos y a adaptarlos para crear nuevos productos mediante la creación de nuevos activos artísticos, diseños de mundos, armas, personajes, vehículos y reglas de juego con cambios mínimos en el software del "motor".

Hacia finales de la década de 1990, algunos juegos como Quake III Arena y Unreal fueron diseñados teniendo en cuenta la reutilización y la "modificación". Los motores se hicieron altamente personalizables mediante lenguajes de programación de scripts como el Quake C de idSoftware, y la licencia de motores comenzó a ser una corriente de ingresos secundaria viable para los desarrolladores que los crearon.

La línea entre un juego y su motor a menudo es borrosa. Algunos motores hacen una distinción razonablemente clara, mientras que otros apenas hacen un intento de separar los dos. En un juego, el código de renderizado podría "saber.^{es}pecíficamente cómo dibujar un orco. En otro juego, el motor de renderizado podría proporcionar facilidades de material y sombreado de propósito general, y la .^{o}rquidad"podría definirse completamente en datos. Ningún estudio logra una separación perfectamente clara entre el juego y el motor, lo cual es comprensible considerando que las definiciones de estos dos componentes a menudo cambian a medida que el diseño del juego se solidifica.

Se podría argumentar que una arquitectura orientada a datos es lo que diferencia a un motor de juego de un software que es un juego pero no un motor. Cuando un juego contiene lógica o reglas de juego codificadas en duro, o utiliza código de casos especiales para representar tipos específicos de objetos de juego, se vuelve difícil o imposible reutilizar ese software para crear un juego diferente. Probablemente deberíamos reservar el término "motor de juego"para el software que es extensible y puede utilizarse como base para muchos juegos diferentes sin modificaciones importantes.

Capítulo 3

Contribuciones

Como se comentó en el plan de trabajo, este TFG esta divido en tres partes fundamentales. Debido a que la carga de trabajo de cada parte es similar, hemos asignado una parte a cada integrante del grupo. A pesar de esta división, sobretodo durante la recta final del trabajo, se han dado contribuciones de todos los integrantes en cada una de las tres partes, aunque eso sí, en menor medida.

3.1. Pablo Fernández Álvarez

Yo me he encargado de la parte del motor. Al principio me dediqué a investigar posibles librerías tanto de físicas como de audio, para integrar al motor. En cuanto a librería de gráficos tuvimos claro desde el principio que íbamos a usar SDL, debido a que la hemos usado bastante durante el grado y estamos acostumbrados, además de que no tiene ninguna limitación para el desarrollo de videojuegos 2D.

Una vez escogidas la librerías, Box2D como motor de física y SDLMixer como motor de audio, comencé a montar el proyecto usando Visual Studio 2022. Organicé la solución en varios proyectos, física, audio, input, render, y fui implementando cada uno de ellos. Empecé con el proyecto de Input, el cuál me llevó algo de tiempo ya que implementé soporte para teclado, mando y múltiples mandos. Una vez terminado, fue el turno del proyecto de sonido/audio. Con la ayuda de la documentación de SDLMixer, implementé soporte para la reproducción de efectos de sonido/sonidos cortos y música. Posteriormente comencé con el proyecto de físicas. En este caso tuve que dedicar bastante más tiempo a aprender sobre la librería, leer artículos y documentación, ya que es más compleja. Investigué también la opción de poder visualizar los colisionadores de los cuerpos físicos ya que supondría una gran ayuda tanto para el desarrollo del motor como para el usuario. En la documentación de Box2D encontré algunos ejemplos pero usaban el OpenGL para el dibujado y el motor usa SDL por lo que tuve que implementar esas

funciones de dibujado con SDL.

Luego llegó el momento de implementar el proyecto del ECS (Entity-Component-System), fundamental para realizar pruebas y visualizar las primeras escenas. Para ello además, implementé el bucle principal del motor con un intervalo de tiempo fijo para el mundo físico. En este momento, con los proyectos principales implmentados, comencé a implementar los primeros componentes básicos, como son el Transform, Image, PhyiscBody y SoundEmitter.

Durante un tiempo simplemente me dediqué a ampliar y probar los componentes y la funcionalidad implementada en los proyectos. Por ejemplo, añadí una matriz de colisiones al proyecto de físicas para manejar el filtrado de colisiones, implementé distintos tipos de PhyiscBody (colisionadores con formas especiales), sincronicé los cuerpos físicos con las transformaciones de la entidad, detección de colisiones, conversión de píxeles a unidades físicas, nuevo componente ParticleSystem para sistemas de partículas, implementé un gestor de recursos para evitar cargar recursos duplicados, mejoré los componentes de sonido para añadir paneo horizontal y sonido 2D, entre otros.

Con la parte del editor más avanzada, implementé la lógica necesaria para leer los datos que genera el editor, como escenas o prefabs. Además, implementé la ventana de gestión de proyectos del editor, añadí control de errores en todo el motor, para conseguir una ejecución continua y esperable, imprimiendo los errores por la salida estándar. Añadí también control de errores en el editor, a través de un fichero de log, con la información de los errores durante la ejecución. Creé una estructura de directorios para el editor y motor haciendo más cómodo el ciclo de desarrollo. Implementé una ventana de preferencias donde ajustar parámetros del motor, como gravedad del mundo físico, frecuencia del motor de audio, tamaño de la ventana de juego, entre muchos otros. Implementé el flujo de escenas, es decir, guardar la última escena abierta, mostrar el viewport de una forma especial en caso de que no haya ninguna escena abierta y mostrar en el viewport el nombre de la escena actual junto con su contenido correspondiente.

Por último, cambié el uso que se hacía de SDL para la implementación de mando en el Input, de SDLJoystick a SDLGameController ya que está más preparada para mando y SDLJoystick es una interfaz de más bajo nivel preparada para cualquier tipo de dispositivo. Añadí más parámetros a la ventana de preferencias, sobretodo para Input, bindeo de teclas rápidas. He mejorado también la ventana de gestión de proyectos para poder eliminar proyectos y ordenar los proyectos por orden de última apertura.

3.2. Yojhan García Peña

Mi principales tareas fueron el desarrollo del lenguaje de scripting, incluyendo su implementación tanto en el editor y en el motor, y también la

unión entre el editor y el motor.

Inicialmente tuve que crear un proyecto vacío de Visual Studio para empezar a prototipar el lenguaje. No quería utilizar ninguna biblioteca externa que implementase la lógica para un editor de nodos, pues prefería poder desarrollarlo desde cero. Tampoco he querido mirar en profundidad el funcionamiento del lenguaje de scripting de otros motores como Unity o Unreal porque no queríamos copiar descaradamente su forma de uso y queríamos desarrollar un lenguaje propio que se adaptase a las necesidades concretas de nuestro motor.

Fue un proceso iterativo donde poco a poco se iba añadiendo funcionalidad, y cuando finalmente conseguí una implementación que me parecía robusta fue cuando decidímos juntar los dos primeros proyectos del TFG, siendo el motor y el scripting. Para poder llevar a cabo la integración tuve que hacer la clase Script, que junto con las clases relacionadas con los nodos ya permitían enlazar los nodos con el bucle de juego principal del motor. También fue necesaria la creación de la clase ScriptFunctionality para poder dotar al lenguaje de funcionalidad básica como sumas, restas y operaciones genéricas para la entrada.

Llegados a este punto pospuse un poco el desarrollo del scripting y empezé a centrarme en cómo sería la unión entre el editor y el motor. Tras varios intentos fallidos finalmente terminé de desarrollar la clase ECSReader, con la cual se puede leer el contenido del motor gracias a unas marcas que se pueden ir añadiéndo en el código. Y tras añadir al proyecto la biblioteca de lectura de JSON de nlohmann, toda la información relevantes del motor pasaba a estar serializada en fichero en disco, siendo la idea que el motor pueda leer los valores desde ese fichero.
Aprovechando que el ECSReader ya estaba creado, fue usado para muchos más usos además de generar ficheros JSON. Finalmente terminamos utilizando el ECSReader para generar código en c++ que pueda leer el motor, de esta forma, automatizando bastante muchos aspectos tediosos. El ECSReader genera tres ficheros diferentes: ClassReflection, que permite serializar los atributos de una clase y poder dotarles de valor desde fuera conociendo el nombre de la variable; ComponentFactory, que con el nombre de un componente en formato string crea un componente de ese tipo, siendo una especie de factoría; y FunctionManager, el cual genera una función para cada método de los componentes y luego los agrupa en un mapa.

Con esto, ya estaba hecha la reflexión con la que queríamos dotar al motor, y siendo la persona que más avanzada iba con su parte decidí ayudar a mis compañeros con algunas tareas. Empezando por el motor y gracias a mi experiencia con serialización de JSON, hice serialización de escenas

y entidades, seguido de la implementación del sistema de Overlays y sus componenetes asociados (Image, Button y Text), la Splash Screen y la serialización de información del proyecto, como tamaño de la ventana o el icono usado.

Cuando el editor iba más avanzado me puse a implementar el editor visual de nodos en el editor. Terminando todo lo relacionado con el editor, y la serialización. Con mi parte terminada en gran medida, fue cuando me puse a hacer la lectura de los ficheros generados por el ECSReader en el Editor, por lo que tuve hacer la clase ComponentReader y ComponentManager. Con todo casi terminado, me puse a hacer cambios en el editor, haciendo una refactorización de cómo se dibujan las ventanas para incluir la rama de Dockind de Imgui en el editor, Para terminar, me puse a crear distintas ventanas de utilidades del editor, como el editor de la paleta de colores, la consola creando un PIPE que lee la salida del programa; la clase Game, que permite ejecutar el exe del motor y controlar el proceso pudiendo detenerlo en cualquier momento haciendo uso de la api de windows; el esqueleto de la clase de preferencias, el manejador del docking para poder cambiar la disposición de las ventanas y el renderizado de la ventana tanto para las entidades físicas como para la interfaz

Por último, estuve implementando funcionalidad que se haya quedado sin implementar y corrigiendo errores.

3.3. Iván Sánchez Míguez

Mi principal responsabilidad en el proyecto se enfocó en el desarrollo del editor. En una fase inicial, mi tarea consistió en llevar a cabo una investigación exhaustiva de proyectos similares para comprender las bibliotecas gráficas utilizadas y evaluar si alguna de ellas podría simplificar nuestro trabajo. Después de un análisis minucioso, llegué a la conclusión de que IMGUI era la elección ideal debido a su amplio conjunto de funcionalidades para la interfaz de usuario (UI).

Una vez seleccionada esta biblioteca, procedí a crear el proyecto en Visual Studio 2022. Inicié con la creación de un programa simple inicial para comprender cómo se inicializa y funciona IMGUI, el cual incluía un proyecto con numerosos ejemplos. Mi enfoque inicial se centró en el diseño de las ventanas principales, como la barra de menú, la jerarquía, la escena, los componentes y el explorador de archivos. Esto se hizo de manera rápida y provisional con el único propósito de familiarizarme rápidamente con IMGUI. Durante este proceso, descubrí la rama ímgui dock'de IMGUI, que permitía el anclaje de ventanas entre sí, lo que resultó ser una característica valiosa para el proyecto.

Posteriormente, reestructuré dicho código para lograr una organización más intuitiva de las ventanas y facilitar la adición de nuevas ventanas. Para ello, cree la clase Window, que es la clase padre de cada ventana y se encarga de incluir la funcionalidad básica de una ventana de IMGUI. Avanzando en el proyecto, me concentré en la creación de la escena, un proceso que inicialmente resultó un tanto complicado debido a la complejidad en la programación, especialmente en lo que respecta al manejo de texturas y su renderización con IMGUI. Sin embargo, con el tiempo, logré comprender estos aspectos y refactorizar el código para obtener una forma más sencilla de renderizar la escena y sus entidades.

Con la escena en funcionamiento, me dediqué a trabajar en la creación de entidades y su representación en la textura. Para ello, creé la clase Entidad, encargada de gestionar su textura y almacenar información sobre su Transform, además de asignar un ID único a cada entidad para diferenciarlas entre sí. Luego, me enfoqué en la gestión de estas entidades a través de la ventana de jerarquía, incorporando un listado con textos seleccionables mediante IMGUI, lo que permitió la selección de entidades. Esto también tuvo un impacto en otras ventanas, como la de componentes, que mostraba información sobre la entidad seleccionada. Inicialmente, la ventana de componentes mostraba solo la información del transform con entradas de IMGUI para modificar sus valores. Finalmente, desarrollé la ventana del explorador de archivos, que aunque al principio no la consideré esencial, resultó importante para la visualización de los activos del proyecto y la navegación entre directorios.

Una vez que logramos tener una versión básica del editor, trabajé en su integración con el motor del proyecto. Esto implicó la serialización de escenas, entidades y sus transformaciones en formato JSON. Posteriormente, tuve que adaptar el proceso de serialización para que fuera compatible con el motor, ya que este tenía formas distintas de leer los componentes disponibles, sus atributos y el tipo de sus atributos. Una vez incorporados los componentes del motor en el editor, me centré en el renderizado y la edición de estos, de modo que cada tipo de atributo tuviera su propia representación en la ventana de componentes, lo que se logró de manera eficiente gracias a IMGUI. También permití la adición de dichos componentes a las entidades.

Posteriormente, me dediqué a la implementación de funciones más avanzadas, como el uso de prefabs y la gestión de la jerarquía entre entidades. Esto fue un desafío significativo, ya que implicaba rehacer el sistema de asignación de IDs para que los prefabs tuvieran IDs negativos y estuvieran referenciados en todas sus instancias. Además, fue necesario tener en cuenta este sistema de IDs en múltiples partes del código para evitar conflictos con el uso normal de las entidades. La gestión de la jerarquía también presentó complejidades, ya que cada entidad contenía referencias a su padre y sus hijos, lo que debía

considerarse en numerosas partes del código. La interacción entre prefabs y la jerarquía también fue un aspecto desafiante, ya que un prefab podía tener hijos. Implementé un sistema para que las entidades padre afectaran el transform de sus hijos, lo que facilitó la interacción entre ellos en la escena, como el movimiento conjunto de las entidades al mover el padre. Además, para gestionar los prefabs, creé una ventana llamada PrefabManager desde la cual se podían ver y editar los prefabs.

Finalmente, junto con el equipo, nos esforzamos en mejorar el editor y corregir sus errores. Para poner a prueba nuestro trabajo, desarrollé una versión básica del juego Space Invaders en nuestro propio editor, identificando y resolviendo numerosos errores en el proceso.

Capítulo 4

Conclusiones

El objetivo de este TFG era desarrollar un motor de videojuegos 2D para no programadores. Hemos cumplido con lo propuesto. Nuestro motor le abre las puestas a aquellos desarrolladores con poca experiencia en programación y a la vez cuenta con la suficiente funcionalidad como para desarrollar videojueos 2D competentes.

Como aplicaciones prácticas, nuestro motor se puede usar en el mundo del desarrollo de videojuegos indie o incluso a nivel didáctico.

A nivel técnico, hemos sacado en conclusión una serie de aspectos:

- Con la implementación actual, las ventanas de ImGUI no se puede mover fuera de la ventana principal de SDL, lo que genera incomodidad en algunas situaciones como al implementar un script, donde seguramente sea intersante visualizar la escena o algún parametro del editor para tomar decisiones.
- Hemos tenido que implementar reflexión en C++. Hubiera sido más cómodo escoger un lenguaje con reflexión e incluso nos hubiera dado más flexibilidad.

Como trabajo futuro pensamos en la siguiente funcinalidad:

- Poder lanzar el juego en el propio editor y no en una ventana separada.
- Cambiar la implementación actual del editor para poder mover las ventanas de ImGUI fuera de la ventana principal de SDL.
- Añadir un sistema de animación y dibujado para disminuir la dependenia de herramientas externas.
- Abstraer al usuario de rutas de ficheros y directorios conviertiendo los ficheros en objetos del motor.

- Poder profundizar más en el scripting, añadir funcionalidad más compleja que permita al usuario una mayor expresividad, por ejemplo, añadiendo arrays editables desde el editor, creación de clases, recursión, temporizadores, corutinas, depuración de los nodos en ejecución.

Bibliografía

Y así, del mucho leer y del poco dormir,
se le secó el celebro de manera que vino
a perder el juicio.

Miguel de Cervantes Saavedra

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes

–Buena está – dijo Sancho –; fírmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.

Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes