Editor y motor de juegos 2D para no programadores

2D game editor and engine for non-programmers

TRABAJO DE FIN DE GRADO

Pablo Fernández Álvarez
Yojhan Steven 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

2D game editor and engine for non-programmers

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.

Queremos dar un especial agradecimiento a Mireia Escobar Gracia, Saúl Baltasar Jimenez, Azazel Cabello Gómez, Daniel Hernández Gagliardi, Silvia Paredero Rubio, Jose Luis Bastida Lorenzo, Rodrigo García Suárez, Martín Mancini, Yuri Maximino Navarro y Lara Gómez Geisler por su compromiso, esfuerzo y su ayuda desinteresada a la hora de realizar pruebas relacionadas con el proyecto, cuya experiencia e ideas aportadas han sido cruciales para obtener información útil para poder mejorar la experiencia de uso.

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 tecnología, han aumentado considerablemente en complejidad y alcance.

Un motor de videojuegos es un software que proporciona la tecnología necesaria para desarrollar el gameplay de un videojuego. Es un conjunto de librerías (gráficos, audio, físicas...) agrupadas de forma coherente que abstraen la tecnología 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 tecnología que requerirá el videojuego y posteriormente desarrollándolo y, por otro lado, usando un motor de videojuegos 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 videojuegos 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. La idea es 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.

Palabras clave: Editor de videojuegos, Entorno desarrollo videojuegos, Motor videojuegos, Scripting visual, Programación por nodos

Índice

Índice de figuras

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 inconveniente 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 la creación de otro tipo de videojuego.

En el segundo caso, el inconveniente reside en la forma de obtener dichas tecnologías, licencias o costes. A la hora de crear 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 es necesaria para garantizar la reutilización de las tecnologías.

A estas tecnologías se las conoce como motores de videojuegos. En general suelen estar constituidos 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 videojuegos, las empresas desarrolladoras de motores incluyen editores. Los editores son herramientas que simplifican el desarrollo comunicando las acciones del usuario 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 carac-

terísticas. Los más cómodos 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 crearlo. 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 su 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 sí lo es si se quiere hacer un juego modesto donde se ponga a prueba la jugabilidad y no tanto la tecnología.

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 la creación de juegos 2D mucho más pequeños para perfiles con poca experiencia en programación o en la creación de videojuegos en general. Esto simplifica el uso del editor y también el tamaño de las versiones ejecutables realizadas.

Las características del motor que queremos hacer son:

- *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 posible 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.
- *Ejecución del juego desde el editor*: Esto proporciona 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.

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 probar 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.

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++, y la generación del PDF ha sido llevada a cabo con LaTeX.

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, núcleo del desarrollo, 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 continua de cada una de las partes.

- *Desarrollo inicial*: Para esta fase, desarrollaremos el núcleo de cada proyecto. En cuanto al motor, se llevarán a cabo las primeras pruebas de 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. Además se llevará a cabo un proceso de integración continua a medida que avanza esta fase para evitar posibles incompatibilidades en la siguiente fase. Esto conllevará la definición de

un formato de fichero de intercambio que será el punto de unión entre el editor y el motor. Se definirá el tipo de información, su estructura básica y cómo se van a referenciar los assets.

- *Núcleo del desarrollo*: Durante esta fase, siendo la más duradera, añadiremos nuevas funcionalidades en cada una de las partes, como nuevos componentes en caso del editor, sistema de ventanas y docking para el editor, etc. Ampliaremos la información de intercambio entre motor y editor de manera que continuaremos integrando y se creará el sistema de scripts visual en el editor para generar los primeros scripts y la lógica para almacenarlos e interpretarlos en el motor.

- *Cierre del desarrollo*: Con los proyectos terminados y completamente integrados, continuaremos el desarrollo de las funcionalidades relacionadas con la mejora de la experiencia de usuario, sobre todo en el editor, 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 usuario.

Capítulo 2

Estado del arte

Resumen: Este capítulo se centra en los videojuegos, cómo se desarrollan, qué es un motor de videojuegos, cuáles son sus partes fundamentales, cuáles son los más utilizados hoy en día y sus características, qué es un editor y porque se usan para el desarrollo, qué significa el scripting 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.

Se puede entender como un programa 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, discos magnéticos, tarjetas de memoria especiales para videojuegos, o bien se descarga directamente a través de Internet.

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.

2.1.1. ¿Cómo se hace un videojuego?

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 modo de desarrollo ha pasado por muchas fases hasta el estado en el que se encuentra 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 finales de la década de 1970 en adelante, la mayoría de los juegos se desarrollan de manera programada, lo que significa que el código y los recursos se crean en computadoras en lugar de hardware dedicado.

La idea fundamental de crear videojuegos se divide en programar el comportamiento, reglas y lógica del juego y crear los recursos (imágenes, audio, mapas, etc) que se quiera cargar en el juego.

Sin entrar mucho en detalle, las etapas que sigue un desarrollo profesional son las siguientes: concepto, diseño, planificación, preproducción, producción, pruebas, distribución y mantenimiento. Para ello, se suelen utilizar metodologías ágiles de producción para controlar y mejorar el desarrollo [3].

2.1.2. Juego dirigido por datos

El concepto de juego dirigido por datos se refiere a la reutilización de la parte de programación de un videojuego para hacer otros nuevos a partir de nuevos recursos, también entendidos como datos. Como ejemplos de juegos clásicos dirigidos por datos serían el Quake II y el Half-Life, desarrollados por id Software y Valve Corporation, respectivamente.

En otras palabras, el motor carga los recursos y cambiando los recursos cambia el juego.

En todos ellos, se creaban nuevos datos/recursos para desarrollarlos. En la figura 2.1 se muestran algunos ejemplos.

(a) Quake II (1997)

(b) Half-Life (1998)

Figura 2.1: Ejemplos de juegos dirigidos por datos con juegos clásicos

2.2. ¿Qué es un motor de videojuegos?

Un motor de videojuegos es un entorno de desarrollo que proporciona herramientas para la creación de videojuegos. Su función principal es dotar al videojuego de una bibloteca para renderizar gráficos 2D y 3D, otra para física que simule las leyes de la física y detección de colisiones, y herramientas para poder crear las animaciones, scripts, sonidos, inteligencia artificial, redes, gestión de memoria, y demás sistemas del videojuego.

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 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 recursos 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 recursos 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 II y Half-Life fueron diseñados teniendo en cuenta la reutilización y la "modificación" de recursos. Los motores se hicieron altamente personalizables mediante lenguajes de programación de scripts como el Quake C de id Software, y la licencia de motores comenzó a ser una corriente de ingresos secundaria viable para los desarrolladores que los crearon.

2.2.1. División entre motor y gameplay

El gameplay se define como la experiencia general de jugar a un videojuego. Incluye las mecánicas, las reglas que gobiernan las interacciones entre los elementos en el juego, los objetivos del jugador, los criterios para ganar o perder, las habilidades del personaje, los NPC (Non-Playable-Characters),

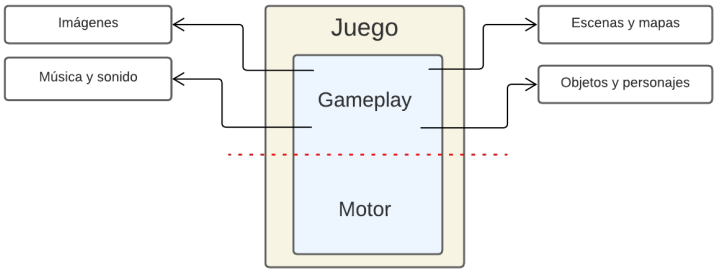

Figura 2.2: División de juego en motor y gameplay.

y el flujo general de la experiencia de juego en su conjunto [4].

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 la figura 2.2 se representa lo que es un juego, la caja más grande, y de lo que está compuesto. Por un lado tenemos el gameplay/datos que son los recursos de imágenes, audio, personajes, modelos, escenas, lógica y comportamiento. Y por otro lado, separado por la línea roja, tenemos el motor, encargado de cargar toda esa información.

En realidad aquí surge un problema. Los recursos obvios son imágenes y audio pero el gameplay también tiene mapas donde se colocan los elementos del juego y, sobre todo, comportamiento que hay que crear y proporcionar al motor. Como veremos más adelante, hay varias formas de conseguir convertir en datos ese comportamiento (que en realidad es programación).

Otro aspecto interesante a comentar es la "altitud" a la que se encuentra la línea divisoria entre motor y gameplay. Si la línea está muy alta, entonces el motor nos aporta mucha funcionalidad y hay que crear poco comportamiento en el gameplay. Eso hace que el motor esté muy focalizado en un tipo de juegos concretos por lo que si el género o tipo del videojuego a desarrollar no es compatible con el tipo de funcionalidad que aporta el motor se va a terminar excluyendo.

De esta forma, es común encontrarse con motores pensados para géneros de videojuegos concretos. Existen motores para desarrollar videojuegos de disparos en primera persona (FPS), videojuegos de plataformas y en tercera persona, videojuegos de lucha, videojuegos de carreras, videojuegos de estrategia en tiempo real o videojuegos multijugador online masivos, conocidos como (MMOG).

Algunos ejemplos de estos motores son Scumm, desarrollado por LucasfilmGames para sus aventuras gráficas, RPG Maker desarrollado por ASCII

Corporation para juegos de rol, las primeras versiones de Frostbite desarrollado por Electronic Arts para juegos de acción y disparos (FPS) o MUGEN desarrollado por Elecbyte para juegos de lucha en dos dimensiones.

En el libro Game Engine Architecture [4] se cita lo siguiente: "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 a mano, 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."

2.2.2. Partes de un motor de videojuegos

Como se ha mencionado anteriormente, un motor proporciona la tecnología necesaria para desarrollar un videojuego. Esta tecnología, en realidad, es un conjunto de varías tecnologías con diferentes propósitos, todos necesarios para el videojuego. Para un motor de carácter general, es decir, sin implementación extra para un tipo de videojuegos concreto, estas tecnologías son las siguientes:

- *Motor de renderizado*: Este componente se encarga de generar los gráficos y la representación visual del juego en la pantalla. Utiliza técnicas de renderizado, como rasterización o trazado de rayos, para crear la imagen final. Los motores modernos a menudo admiten efectos visuales avanzados, como sombras dinámicas, iluminación global y efectos de partículas.

- *Motor de física*: La física es esencial para dar realismo y coherencia al juego. Este motor simula el movimiento de objetos, colisiones, gravedad y otros efectos físicos. Permite que los personajes y objetos del juego interactúen de manera creíble con el entorno y entre sí.

- *Motor de audio*: Gestiona la reproducción de efectos de sonido y música en el juego. Puede incluir capacidades de mezcla de audio y efectos especiales para crear una experiencia auditiva envolvente.

- *Motor de input*: Controla la interacción del usuario con el juego a través de dispositivos de entrada como teclados, ratones, controladores de juegos o pantallas táctiles.

- *Motor de animación*: Esto incluye la animación de personajes, la interpolación de movimientos y la gestión de esqueletos o huesos para modelar la anatomía de los personajes.

- *Motor de red*: Facilita el juego en línea y la comunicación entre jugadores. Administra la sincronización de datos entre los clientes y el servidor, permite el chat en línea, la gestión de partidas y otros aspectos relacionados con la conectividad en línea.

- *Gestor de recursos*: Garantiza que todos los posibles recursos del videojuego se carguen y descarguen de manera eficiente en memoria durante la ejecución del juego.

- *Subsistemas de soporte*: Sistemas para iniciar y cerrar el motor, manejar la asignación de memoria, sistema de ficheros, etc.

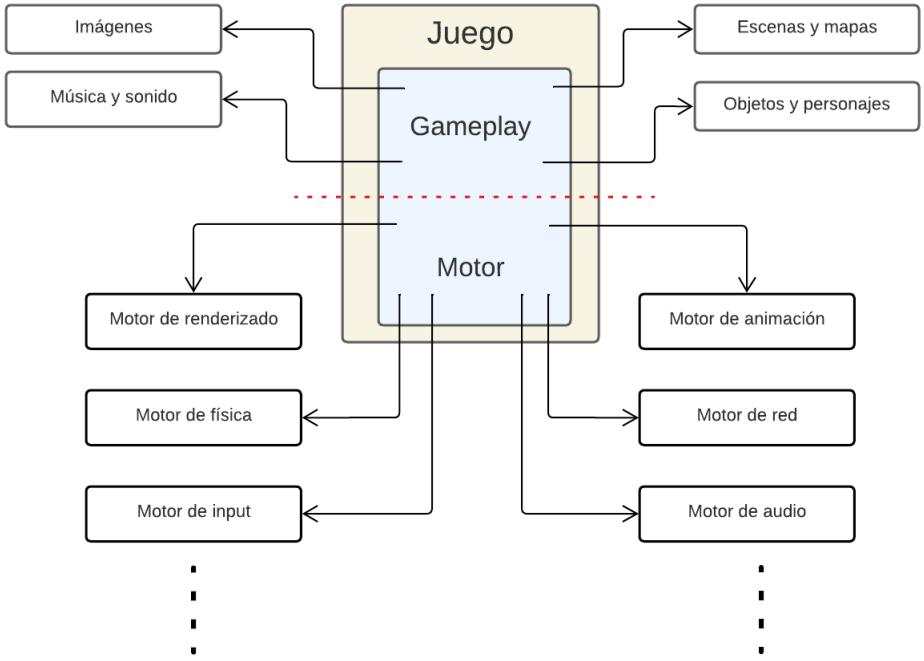

Figura 2.3: Partes fundamentales que componen un motor de videojuegos.

2.3. Arquitectura de videojuegos

Hasta ahora hemos hablado sobre la idea de motor de videojuegos y lo que los separa del gameplay. Pero a la hora de desarrollar un videojuego existen varios patrones o arquitecturas que se pueden seguir para construir sus bases. Estas arquitecturas definen el modo de implementar el gameplay y sirven de conexión con el motor.

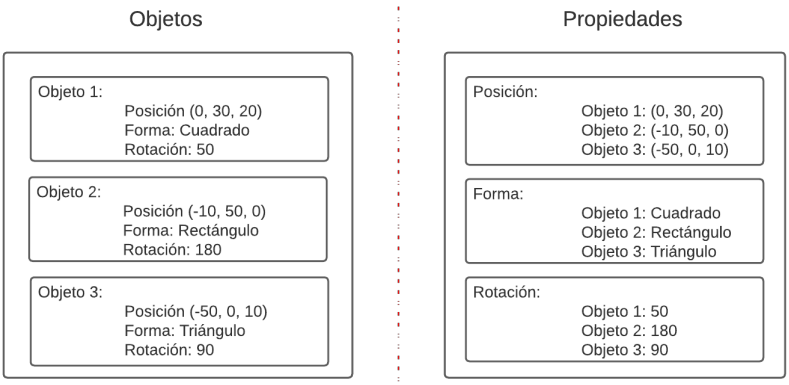

Figura 2.4: Diferencia entre arquitectura centrada en objetos vs propiedades.

Un concepto importante para entender el propósito de las arquitecturas son las entidades. Las entidades representan objetos individuales en el juego. Esto podría ser cualquier cosa, desde un personaje, un enemigo, un proyectil, hasta un objeto inanimado como una piedra. A día de hoy los videojuegos son un conjunto de entidades que interactuán entre sí y la arquitectura usada en el videojuego define como se implementan dichas entidades.

Existen dos tipos de arquitecturas principales [4]:

- *Centrada en objetos*: Cada objeto tiene un conjunto de atributos y comportamientos que están encapsulados dentro de la clase (o clases) de la cual el objeto es una instancia. El mundo del juego es simplemente una colección de objetos del juego.
- *Centrada en propiedades*: Cada objeto tiene un identificador y un conjunto de propiedades asociadas a ese identificador. El comportamiento de un objeto del juego se define de forma implícita por la colección de propiedades de las que está compuesto.

En la figura 2.4 se muestra la diferencia.

2.3.1. Arquitectura basada en herencia

La arquitectura basada en herencia ha sido un enfoque tradicional en el desarrollo de videojuegos y todavía se utiliza en algunos motores y sistemas. Este enfoque se centra en la creación de jerarquías de clases en las que cada clase representa un tipo de objeto en el juego. Por ejemplo, si está creando un juego de rol (RPG), se podría tener una jerarquía de clases que comienza con una clase base “Personaje” y se divide en clases derivadas como “Jugador” y “Enemigo”. Cada clase puede agregar o modificar atributos y métodos según sea necesario.

La arquitectura basada en herencia presenta ciertas ventajas, especialmente en proyectos pequeños o simples. Una de las ventajas clave es su simplicidad conceptual, ya que la herencia permite establecer relaciones claras entre clases a través de una jerarquía.

Sin embargo, esta arquitectura también tiene sus desventajas. Uno de los problemas más significativos es su rigidez. La herencia puede hacer que un sistema sea menos adaptable a cambios o que la creación de nuevas clases sea más complicada, ya que cualquier modificación en la clase base puede afectar a todas las clases derivadas. En proyectos grandes, las jerarquías de clases pueden volverse extremadamente complejas y difíciles de mantener, lo que puede aumentar la complejidad y el tiempo de desarrollo. Además, compartir funcionalidad entre clases no relacionadas en la jerarquía puede ser complicado, lo que dificulta la reutilización de código.

2.3.2. Arquitectura basada en entidades y componentes

La arquitectura basada en entidades y componentes (EC) es un enfoque fundamental en el desarrollo de videojuegos debido a su flexibilidad y eficiencia en el manejo de la complejidad. Esta basado en la idea de arquitectura centrada en propiedades.

En el corazón de la arquitectura EC se encuentran dos conceptos clave:

- *Entidades*: Son contenedores vacíos que no contienen lógica ni información por sí mismos y lo que les define son sus componentes.
- *Componentes*: Los componentes son módulos independientes que contienen datos y lógica. Cada entidad en un juego está compuesta por uno o más componentes que definen sus atributos y comportamiento. Por ejemplo, una entidad de jugador podría tener componentes como "Posición" para representar su ubicación en el mundo, "Imagen" para la apariencia visual o "Movimiento" para el movimiento del personaje, entre otros.

La principal ventaja de esta separación entre entidades y componentes es que permite una gran flexibilidad y reutilización de código. Las entidades pueden ser construidas y modificadas dinámicamente combinando diferentes componentes. Esto facilita la creación de nuevos tipos de objetos en el juego sin necesidad de escribir código específico para cada uno.

2.4. ¿Qué es un editor?

Un editor de videojuegos es una herramienta de software diseñada para facilitar la creación, diseño y modificación de videojuegos. A lo largo de la

historia de los videojuegos, los editores han evolucionado significativamente en términos de funcionalidad y usabilidad [7].

2.4.1. Evolución histórica de los editores

En sus inicios, los editores de videojuegos eran herramientas rudimentarias que requerían un conocimiento técnico profundo y estaban reservados principalmente para desarrolladores de juegos experimentados. Estos primeros editores solían centrarse en la manipulación de código y la creación de niveles o mapas, con una interfaz de usuario limitada.

Con el tiempo, a medida que la industria de los videojuegos creció y se diversificó, los editores se volvieron más accesibles y versátiles. Surgieron soluciones más amigables para el usuario que permitían a diseñadores, artistas y creadores de contenido participar en el proceso de desarrollo de juegos sin necesidad de habilidades de programación avanzadas.

2.4.2. Importancia en el desarrollo de videojuegos

Los editores de videojuegos desempeñan un papel crucial en el proceso de desarrollo de un videojuego. Estas herramientas permiten a los desarrolladores:

1. *Diseñar niveles y escenarios*: Los editores permiten la creación y edición de entornos, niveles y escenarios. Los diseñadores pueden colocar objetos, enemigos, obstáculos y elementos interactivos en el mundo del juego.

2. *Gestionar recursos*: Los editores proporcionan una plataforma para administrar recursos del juego, como imágenes, sonidos, modelos 3D y más. Los recursos se organizan y se pueden vincular a elementos del juego.

3. *Definir comportamientos*: Los editores modernos permiten definir comportamientos de personajes y objetos mediante sistemas visuales de programación o scripts. Esto facilita la creación de interacciones y mecánicas de juego.

4. *Simular y depurar*: Los editores a menudo incluyen herramientas de simulación y depuración que ayudan a los desarrolladores a probar y solucionar problemas en tiempo real.

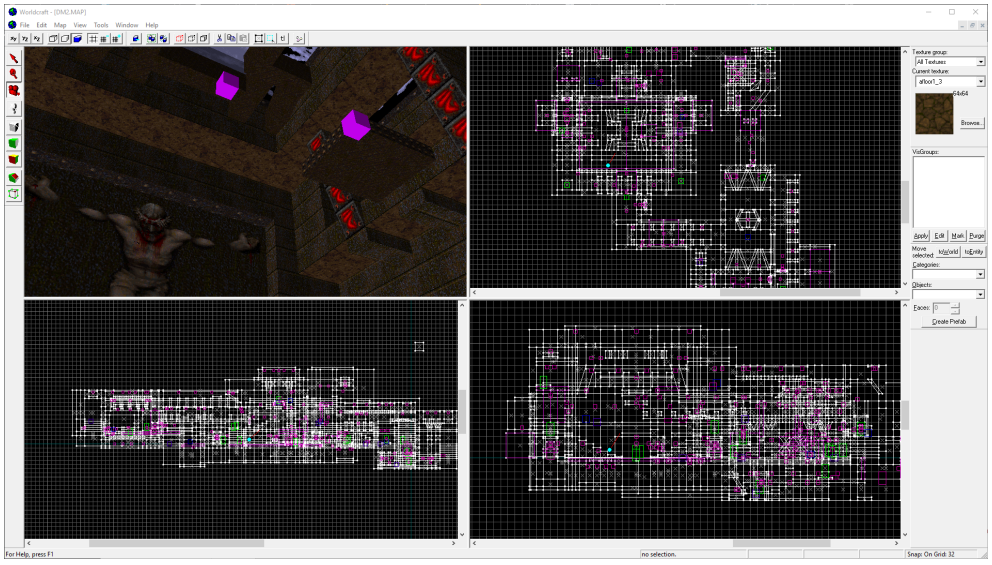

Figura 2.5: Editor de WorldCraft

2.4.3. Enfoque histórico y actual de la integración entre editor y motor

En la actualidad, muchos editores de videojuegos vienen con motores integrados, una evolución significativa con respecto a la forma en que solían operar por separado. Antes, los motores y los editores solían ser herramientas independientes, lo que requería a los desarrolladores crear contenido en el editor y luego importarlo y vincularlo manualmente al motor.

Un ejemplo histórico de esta separación entre motores y editores es el caso de WorldCraft (figura 2.5), una herramienta de diseño de niveles utilizada por Valve para desarrollar juegos como "Half-Life". En ese entorno, los diseñadores creaban mapas y niveles en WorldCraft, pero luego necesitaban compilar esos niveles en un formato que el motor del juego pudiera entender. Este proceso de compilación a menudo llevaba tiempo y podía ser complicado, lo que requería una comprensión profunda de ambas herramientas.

Sin embargo, con la integración de los motores dentro de los editores en la actualidad, este proceso se ha simplificado significativamente. El motor y el editor trabajan en conjunto de manera más fluida, lo que permite a los desarrolladores diseñar niveles, crear contenido y ajustar parámetros directamente en el editor. La comunicación entre el editor y el motor es más directa, lo que significa que los cambios realizados en el editor se reflejan de manera más inmediata en el juego. Por ejemplo, un diseñador puede usar el editor para colocar entidades en el mapa, definir sus atributos y comportamientos, y estos cambios se pueden apreciar de manera inmediata.

Esta integración ha mejorado significativamente la eficiencia y la productividad en el desarrollo de videojuegos, permitiendo a los equipos de desarrollo centrarse más en la creatividad y la iteración rápida.

2.5. Convertir el comportamiento en datos para el motor

Una pregunta esencial en el desarrollo de videojuegos es cómo traducir el comportamiento deseado del juego en información que el motor del juego pueda procesar. La manera en que esto se logra depende en gran medida de cómo esté estructurado el motor y que margen de personalizaciçon se quiere proporcionar a los creadores de gameplay.

Si el motor del juego está diseñado para ser muy específico y limitado en cuanto a la personalización, es posible que no sea necesario programar comportamientos adicionales. Por ejemplo, en las expansiones de Los Sims, gran parte del comportamiento ya está incorporado en el motor del juego, y los datos simplemente se utilizan para ajustar y personalizar ese comportamiento existente. En este caso, el diseñador trabaja dentro de las restricciones del motor existente y utiliza los datos para modificar cómo se comportan las entidades del juego.

En contraste, si el motor no está diseñado para ser tan específico y se requiere agregar comportamientos particulares, entonces es necesario recurrir a la programación. Es aquí donde se pueden dar distintos enfoques:

1. Un enfoque común en esta situación es el uso de Dynamic Link Libraries o bibliotecas de enlace dinámico. Aquí, los programadores escriben código en lenguajes como C++ para definir nuevos comportamientos o características del juego. Estos comportamientos se agrupan en bibliotecas que se integran con el motor del juego.

   La clave del funcionamiento de estas bibliotecas es el enlace dinámico. En lugar de incluir todo el código de la biblioteca en el ejecutable del juego, el programa principal solo contiene referencias a las funciones y datos en esta. Estas referencias se resuelven en tiempo de ejecución cuando el juego se carga. Esto significa que el ".exe" es independiente de la biblioteca, que se carga como datos y se ejecuta siguiendo el convenio acordado por el motor.

   Este enfoque de bibliotecas de enlace dinámico ha sido ampliamente utilizado en motores de juego como Unreal Engine, lo que ha contribuido a su éxito en la industria del desarrollo de videojuegos. Además, también se empleó en juegos icónicos como Quake II y Half-Life.

2. Otra alternativa importante es el uso de "programación no nativa" mediante el empleo de lenguajes de script. Un ejemplo clásico de esto es el uso de Lua. Lua es un lenguaje de script que se interpreta, lo que significa que para el motor del juego es simplemente un archivo de texto que contiene instrucciones.

   Este enfoque tiene varias ventajas. Primero, permite iteraciones rápidas en el desarrollo, ya que no es necesario recompilar todo. Además, no se requiere un entorno de desarrollo costoso como Visual Studio para programar el gameplay, lo que reduce los costos y las barreras de entrada para los desarrolladores ya que se ejecuta en máquina virtual aislada y suelen ser lenguajes mas sencillos [2].

   Sin embargo, este enfoque también presenta desafíos. Se necesita una conexión efectiva entre el mundo nativo (generalmente escrito en lenguajes como C++) y el lenguaje de script para que puedan comunicarse y llamar funciones entre sí, y la ejecución de un lenguaje interpretado es más lenta.

   A cambio, como todo se maneja como datos, la parte de reflexión se vuelve más sencilla para la serialización y la interacción con el editor del juego. Esto significa que los comportamientos y características definidos en lenguajes de script suelen ser más accesibles y configurables en el editor del juego sin necesidad de recompilar el motor.

Por supuesto, también es importante considerar las limitaciones y desafíos que los lenguajes de script pueden presentar. Aunque son poderosos en muchos aspectos, no son lo suficientemente sencillos como para que los diseñadores puedan esbozar comportamientos de manera rápida. Una opción de mayor simplicidad es la programación visual.

Programación visual

La programación visual es una alternativa que busca hacer que la creación de comportamientos y características del juego sea más accesible para los diseñadores. En lugar de escribir líneas de código, los diseñadores pueden utilizar interfaces gráficas y elementos visuales para definir y configurar el comportamiento del juego. Esto hace que el proceso sea más intuitivo y menos dependiente de conocimientos de programación.

En este enfoque, lo que se crea visualmente se traduce aún en datos que el motor del juego puede entender. Estos datos pueden interpretarse de diversas formas. Por ejemplo, se pueden transpilar a código C++, lo que finalmente nos llevaría de nuevo al primer enfoque que mencionamos. También podrían reescribirse en un lenguaje de script como Lua, o simplemente serializarse

en un formato propio y luego ejecutarse mediante un intérprete diseñado específicamente para ese formato.

2.5.1. Comunicación entre motor y scripting

La interacción entre el motor del juego y el sistema de scripting en lo que respecta al gameplay es una consideración central en el proceso de desarrollo de videojuegos en tiempo real. Dado que el motor del juego conoce los eventos que suceden en el juego, como el fin de una partida o colisiones entre entidades, surge la interrogante de cómo las entidades del juego adquieren conocimiento de estos eventos. En este contexto, se emplean principalmente dos enfoques: el sistema de eventos y el método de encuestas:

- *Eventos*: El sistema de eventos podría dividirse en tres partes:

  1. *Generación de eventos*: El motor del juego genera eventos cuando ocurren acciones o situaciones relevantes en el juego. Estos eventos pueden ser muy variados y pueden incluir cosas como colisiones entre objetos, la finalización de una partida, la recolección de un objeto, etc.
  2. *Registro de eventos*: Las entidades del juego pueden registrarse para escuchar eventos específicos. Por ejemplo, un personaje podría registrarse para recibir un evento cuando colisiona con un enemigo.
  3. *Manejadores de eventos*: Cuando se produce un evento, el motor del juego busca a todas las entidades registradas que estén interesadas en ese evento particular y llama a los manejadores de eventos correspondientes en esas entidades. Estos manejadores de eventos son funciones que se ejecutan para reaccionar al evento de una manera determinada.

- *Encuestas*: El método de encuestas implica que las entidades del juego consulten activamente el estado del juego y los eventos relevantes en lugar de esperar a ser notificadas de ellos. En este enfoque, cada entidad del juego, de manera periódica o en respuesta a ciertos eventos de tiempo, realiza una encuesta o consulta al estado actual del juego, y reacciona en base a ello.

  Si bien este método puede ser más simple de implementar en algunos casos, puede resultar en un mayor uso de recursos de CPU ya que las entidades deben realizar encuestas continuas, incluso si no hay eventos relevantes en ese momento.

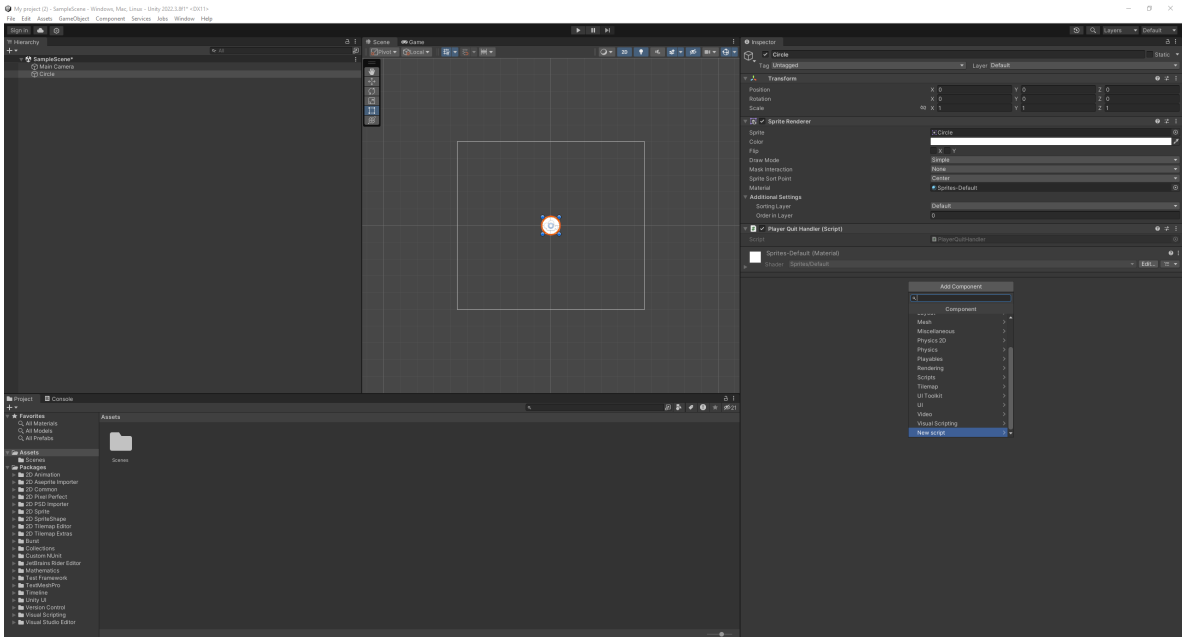

Figura 2.6: Editor de Unity.

2.6. Ejemplos de motores

2.6.1. Unity

Unity es ampliamente reconocido como uno de los motores de videojuegos más populares y versátiles en la industria actual, que fue lanzado en 2005 por Unity Technologies. Lo que distingue a Unity es su separación intermedia entre gameplay y motor, que permite desarrollar una amplia variedad de videojuegos, desde simples juegos móviles 2D hasta experiencias de realidad virtual inmersivas para múltiples plataformas, incluyendo PC, consolas y dispositivos móviles [10].

En la figura 2.6 podemos ver el editor de Unity.

2.6.1.1. Arquitectura

En términos de su arquitectura, Unity adopta una estructura que se alinea con la arquitectura basada en entidades y componentes (EC) [2.3.2]. En este enfoque, los elementos fundamentales son los *GameObjects*, que funcionan como las entidades en el mundo del juego. Cada *GameObject* puede estar compuesto por una combinación de "componentes", que actúan como los módulos responsables de definir su comportamiento y características.

La familiaridad con esta arquitectura es evidente desde el propio editor de Unity. El proceso de desarrollo se basa en la creación de entidades (*GameOb-*

jects) y la asignación de scripts (componentes) para otorgar funcionalidad al juego. Estos scripts, escritos en C#, permiten definir cómo interactúan y responden los *GameObject* en el mundo del juego.

2.6.1.2. Scripting

El sistema de scripting de Unity se basa en C# y hace uso fundamentalmente del componente `MonoBehaviour`. `MonoBehaviour` es una clase de la que todos los scripts en Unity deben heredar para funcionar como componentes adjuntos a objetos en el juego. Significa que cualquier script destinado a ser un componente en un *GameObject* debe ser una subclase de `MonoBehaviour`. Esta herencia permite que Unity comprenda y gestione adecuadamente los scripts como parte integral de la lógica del juego [5].

2.6.2. Unreal Engine

Unreal Engine es un motor de videojuegos desarrollado por Epic Games. Apareció por primera vez en 1998 con el videojuego de disparos en primera persona Unreal y actualmente se encuentra en la versión 5.3. Está escrito en C++ y es multiplataforma. En cuanto a la "altitud" de la línea de separación entre motor y gameplay, Unreal Engine ofrece bastante pero no por ello lo convierte en un motor pensado para un tipo de videojuegos concreto. De hecho, aporta tanta funcionalidad que se considera de propósito general. Es tan potente que además de videojuegos, con Unreal se puede hacer simulación, diseño de automóviles, arquitectura, e incluso vídeos y películas [6].

En cuanto al editor, se muestra en la figura 2.7:

2.6.2.1. Arquitectura

La arquitectura de gameplay en Unreal Engine es una mezcla de componentes y herencia. El motor utiliza una combinación de herencia de clases y la composición de componentes para crear objetos y personajes en el mundo del juego.

En cuanto a la herencia de clases, Unreal Engine utiliza la herencia de clases para crear una jerarquía de objetos y personajes. Por ejemplo, se puede tener una clase base que representa un personaje jugable y luego crear subclases que hereden de la clase base para crear personajes específicos con características adicionales. Esto permite la reutilización de código y la organización de la funcionalidad común.

En Unreal lo correspondiente a los *GameObject* son los *Actor*.

Además de la herencia de clases, Unreal Engine hace un uso extensivo de la composición de componentes. Los componentes son módulos independientes que se adjuntan a los actores (objetos y personajes) para proporcionar

Figura 2.7: Editor de Unreal Engine.

funcionalidad adicional. Por ejemplo, un personaje jugable puede tener componentes para la cámara, el control de movimiento, la colisión y la animación. Los componentes se pueden agregar y eliminar de manera flexible para personalizar el comportamiento de los actores sin necesidad de crear nuevas clases.

Por otra parte, Unreal Engine cuenta con los Blueprints, que permiten definir la lógica y el comportamiento de los actores y componentes de una manera visual y gráfica. Esto significa que se puede definir cómo interactúan los componentes y actores a través de conexiones visuales en lugar de escribir código. Los Blueprints se pueden utilizar tanto para heredar comportamientos como para agregar componentes y personalizarlos.

2.6.2.2. Sistema de scripting

En cuanto al creación de comportamiento, Unreal Engine tiene dos modos distintos pero compatibles:

- *Programación en C++*: Permite a los desarrolladores escribir código en C++ para crear la lógica del juego, personalizar el comportamiento de los actores y desarrollar características específicas. Esta opción es ideal para tareas que requieren un alto rendimiento o una manipulación más detallada de los sistemas del motor. También es cierto que la

programación en C++ puede ser más compleja que el uso de Blueprints, especialmente para quienes no están familiarizados con el lenguaje [11].

- *Blueprints*: Son una herramienta visual que permite a los desarrolladores crear la lógica del juego sin necesidad de programar en C++. Este scripting es de tipo basado en nodos y funciona con la unión de nodos y a través de la creación flujo y lógica. Por ello, se pueden crear prototipos rápidamente y no requieren de conocimientos de programación para utilizarlos [12]. En la figura 2.8 vemos la herramienta Blueprints de Unreal.

En cuanto a la combinación de ambos enfoques, Unreal Engine permite integrar fácilmente código C++ y Blueprints en un mismo proyecto. Esto significa que se puede utilizar C++ para las partes críticas del rendimiento o para implementar sistemas complejos, mientras que se utiliza Blueprints para prototipos rápidos, lógica de juego simple y personalización de componentes y actores. Los Blueprints también pueden comunicarse con código C++ a través de interfaces definidas, lo que permite una colaboración eficaz entre programadores y diseñadores.

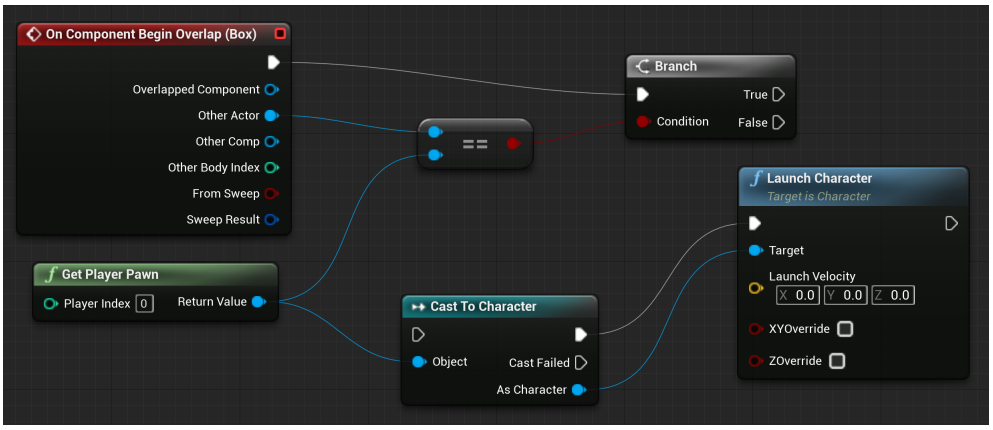

Figura 2.8: Blueprints de Unreal Engine.

Capítulo 3

Visión general del trabajo

En el capítulo anterior, exploramos la naturaleza de un motor de juego y un editor, así como su interacción mutua. Mientras que un motor con un editor puede acelerar significativamente el proceso de desarrollo de juegos, los editores en sí mismos representan una inversión considerable en términos de recursos y tiempo. Los editores con motor integrado suelen ser extensos y ofrecen una amplia gama de funcionalidades para atender diversas necesidades del desarrollo de juegos, lo que puede ser un inconveniente si se busca crear un juego pequeño y simple, ya que obliga a los desarrolladores a aprender a utilizar características innecesarias y puede resultar en un producto final con un exceso de funcionalidades no utilizadas.

Por lo tanto, hemos optado por desarrollar un motor dirigido por datos específicamente diseñado para juegos 2D simples y eficientes, complementado por un editor que permita la creación de estos juegos. En nuestro enfoque, los datos desempeñan un papel central al servir como el medio de comunicación entre el editor y el motor. El editor genera los datos sobre niveles, entidades, componentes, etc y el motor los puede leer e interpretar.

Una decisión crítica en nuestro proyecto fue mantener el motor y el editor separados en lugar de estar integrados. Esto supone tener que diseñar una forma de comunicarlos, pero a su vez nos aporta flexibilidad ya que nos permitiría poder utilizar el motor con un editor diferente y viceversa. Por último, esta separación fue importante para poder acelerar el desarrollo del proyecto, pudiendo paralelizar el trabajo.

Nuestro modelo, es similar al de Unity [2.6.1]. El elemento fundamental es la escena, que contiene un conjunto de entidades. Estas entidades están equipadas con componentes que, a su vez, almacenan atributos que representan datos específicos, al igual que hace Unity. Estos atributos pueden variar desde tipos de datos simples hasta referencias a otras entidades a partir de su identificador o recursos como imágenes a partir de su ruta. La serialización de estos datos se lleva a cabo en formato JSON, lo que permite una

representación legible de los objetos del juego.

Tanto el editor como el motor cuentan además con soporte para prefabs. Un prefab es una entidad predefinida que se puede instanciar en la escena. Contiene una configuración específica y puede reutilizarse en diferentes partes del juego para acelerar y simplificar el proceso de diseño y desarrollo, pudiendo instanciar copias de la entidad tanto desde el editor (pudiendo además modificar el prefab original para modificar simultáneamente todas sus instancias), como desde el motor en tiempo de ejecución.

Como se mencionó en el apartado de motivación [1.1], el objetivo es desarrollar un motor con editor que pueda ser utilizado por gente que no sea programadora. Por ello, se ha decidido implementar un sistema de programación visual. De esta forma, el usuario no tendrá que estar previamente familiarizado con ningún lenguaje de programación, pudiendo centrarse en el desarrollo de videojuegos.

Hemos decidido que los scripts funcionen como datos en vez de como código compilado, ahorrando tiempos de compilación y evitando depender de una programa externo para compilar el código, aunque esta forma de implementación implica que la ejecución del juego no estará tan optimizada, pues en vez de poder ejecutar código directamente, éste tiene que ser interpretado por el motor. Además esta solución también conlleva tener que almacenar la información para los scripts en ficheros externos.

El lenguaje permite controlar el orden de ejecución de cada nodo, otorgando legibilidad y previsibilidad a su ejecución. El lenguaje cuenta con distintos tipos de nodos, cada tipo con una función concreta, como ejecutar una función del motor, almacenar el valor de un dato especificado por el usuario, o modificar el orden de ejecución de los nodos en tiempo real.

Para desarrollar el scripting, el editor de nodos (ubicado en el editor), permitirá al usuario crear distintos tipos de nodos y enlazarlos entre ellos mediante el uso de flechas para crear el comportamiento deseado. Con este comportamiento se puede diseñar funcionalidad que interactúe con el funcionamiento del motor, pudiendo leer y modificar sus valores para de esta forma poder, por ejemplo, mover objetos, cambiar el texto de la interfaz o añadir fuerzas a los objetos físicos de la escena. Esta funcionalidad se obetendrá mediante un flujo de datos conseguidos a partir del motor. Cuando el usuario haya terminado con el proceso de creación, la información relacionada con el script será serializada en un fichero usando un formato JSON. Este fichero formará parte de los recursos del juego, y será leído por el motor durante la ejecución del juego. En el proceso de lectura, el motor leerá y procesará esta información, almacenándolo en una estructura de datos que internamente representa el script y que puede ser accedida más adelante. Esta estructura está preparada para ejecutar el script, funcionando como un

intérprete capaz de ejecutar funciones dentro del motor con la información proporcionada. Esto es gracias a que el motor cuenta con funcionalidad que, conociendo únicamente el nombre de la función que se quiera ejecutar, pueda identificar de qué función se trata y proceder a ejecutarla. Haciendo que el intérprete no dependa del motor y funcione conociendo únicamente el nombre nos permite añadir funcionalidad nueva en el motor sin tener que cambiar el funcionamiento del intérprete, aunque esto podría hacer que cambiando la funcionalidad del motor algún script creado previamente pase a ser obsolete haciendo referencia a funcionalidad que haya sido descartada.

El motor da control al script mediante el uso de una serie de eventos. Estos eventos van desde los más básicos como `Update`, que se lanza en cada fotograma, a algunos eventos más específicos de uso como ejecutarse al detectar una colisión de la entidad, o al posicionar el ratón sobre un elemento de la interfaz.En el editor de nodos, estos eventos son representados como un nodo adicional, haciendo que se mantenga cohesionado con resto del editor de nodos, pudiendo utilizar en el mismo script todos los eventos que se consideren oportunos.

Hemos mencionado que el editor usará información proporcionada por el motor pero no hemos mencionado cómo se consigue. Esta información es generada de manera automática por una herramienta externa al motor desarrollada por nosotros y se almacena en un fichero en formato JSON. La información almacenada contiene toda la información relacionada con los componentes, como el nombre, sus métodos y sus atributos. Esta herramienta se encarga de leer, procesar y filtrar toda la información relacionada con la estructura de componentes del motor, generando los ficheros correspondientes. Si bien esta herramienta es esencial para la comunicación del motor con el editor, también se utiliza para automatizar muchos aspectos tediosos en el desarrollo del motor, como la generación automática de factorías para cada componente, así como la generación automática de estructuras de datos dentro del motor.

En la figura 3.1 vemos un esquema general y simplificado del proyecto.

En los próximos capítulos analizaremos los diversos elementos de este esquema. En primer lugar, se abordará el motor, se expondrá su arquitectura y su funcionamiento. En segundo lugar, se examinará el editor, se describirá su proceso de lectura de los componentes nativos del motor y su capacidad para generar las escenas que posteriormente serán procesadas por el motor, entre otros aspectos.

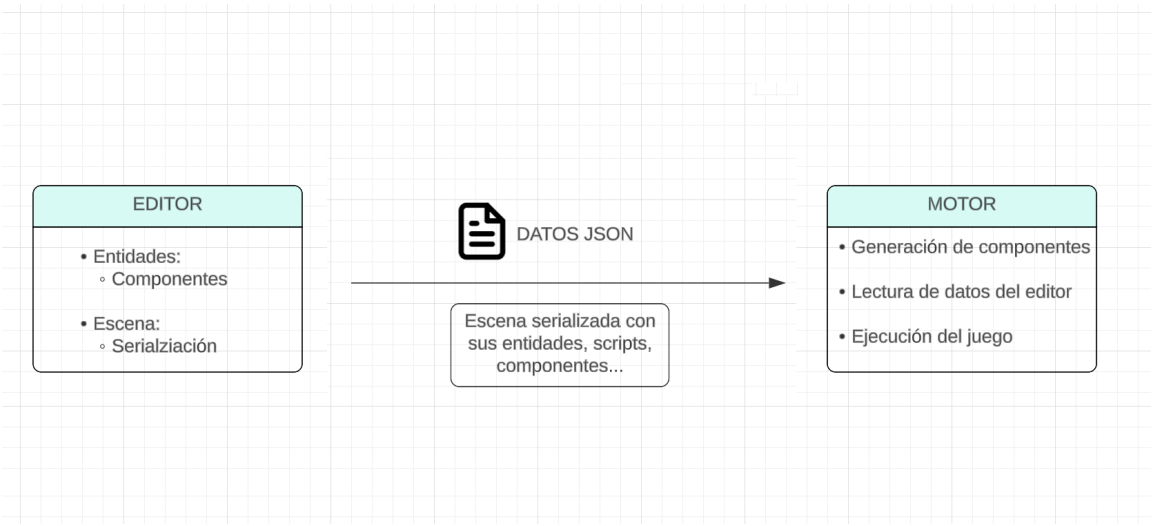

Figura 3.1: Esquema general y simplificado del proyecto.

Capítulo 4

Motor

4.1. Funcionamiento y partes del motor

El motor esta dividido en diez partes fundamentales. A continuación se entrará en detalle sobre la función y detalles de implementación de cada una y de las librerías utilizadas.

Como introducción, en la figura 4.1 se muestra una diagrama resumido sobre la arquitectura del motor.

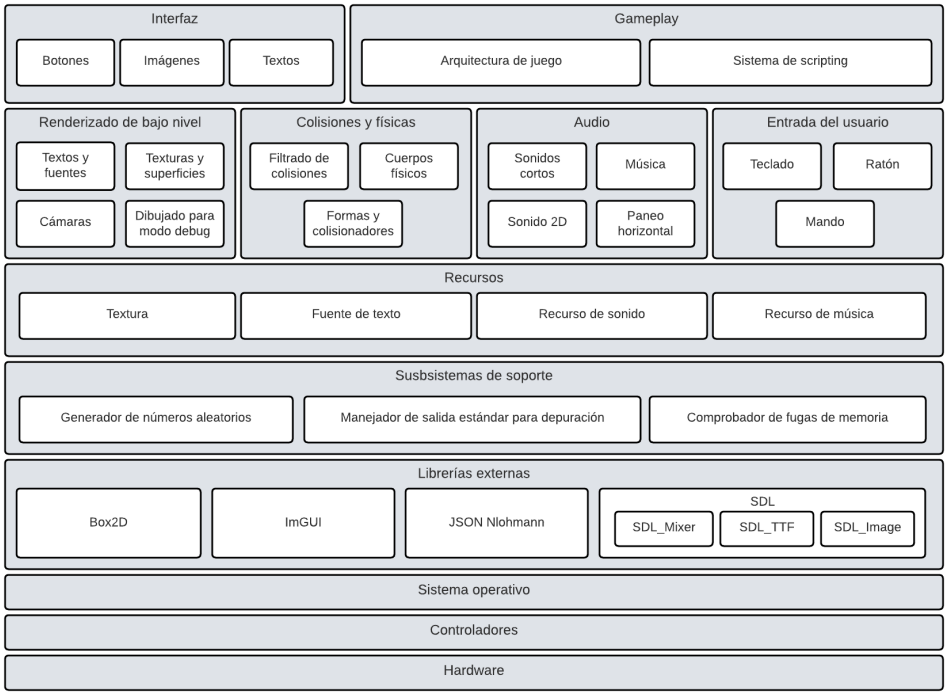

Figura 4.1: Arquitectura del motor.

4.1.1. Consola

Este proyecto contiene una sola clase Output con métodos estáticos que implementan funcionalidad relacionada con el mostrado de la salida estándar por la consola. Tiene métodos para imprimir texto estándar, advertencias o errores. Estos textos se diferencian por su color y sirven principalmente como ayuda para el desarrollo de un videojuego.

El motor, llamado ShyEngine, esta dividido en diez partes y cada una cumple una función específica. A continuación se entrará en detalle sobre la función y detalles de implementación de cada proyecto y de las librerías asociadas al mismo, si las tiene.

4.1.2. Utilidades

El objetivo de este proyecto es implementar código común que pueden necesitar el resto de proyectos evitando así la duplicación de código innecesaria. Contiene clases tanto orientadas a guardar información como a implementar lógica y funcionalidad.

Unos ejemplos de estas clases serían:

- `Vector2D`: Representa un vector bidimensional, contiene información de dos componentes e implementa muchas de sus operaciones básicas.
- `Random`: Contiene métodos estáticos útiles para calcular aleatoriedad entre números enteros, números reales, ángulos, y colores.
- `Color`: Representa un color de tres canales (Red, Green, Blue) además de métodos con algo de funcionalidad como `Lerp`, que calcula un color intermedio entre otros dos dados.
- `Singleton`: Una plantilla para crear instancias estáticas a través de herencia. Es decir, en caso de querer convertir una clase en un `Singleton`, muy útiles para managers, simplemente hay que heredar de esta clase para conseguirlo.

4.1.3. Consola

Este proyecto contiene una sola clase Output con métodos estáticos que implementan funcionalidad relacionada con el mostrado de la salida estándar por la consola. Tiene métodos para imprimir texto estándar, advertencias o errores. Estos textos se diferencian por su color y sirven principalmente como ayuda para el desarrollo de un videojuego.

4.1.4. Recursos

El objetivo de este proyecto es proporcionar un contenedor de recursos en el que se van a guardar todos los recursos del videojuego. En concreto, el tipo de recursos que se pueden guardar son fuentes de texto, imágenes, efectos de sonido y música.

Uno de los objetivos del manager de recursos es reutilizar los recursos creados para solo tener cargada una copia de cada recurso en memoria. Por ello, a la hora de añadir un nuevo recurso al manager, primero comprueba si ya lo contiene y en ese caso, lo devuelve; en caso contrario, lo crea.

4.1.5. Sonido

El objetivo de este proyecto es construir un envoltorio sobre la librería de audio SDL_Mixer para poder implementar posteriormente los componentes de emisión de música y de sonidos.

Para un mejor entendimiento de la implementación es necesario saber que SDL_Mixer diferencia entre efectos de sonido o sonidos cortos en general (WAV, MP3) y música de fondo (WAV, MP3, OGG).

Para la música, la librería solo cuenta con un canal de reproducción por lo que es algo limitado pero simple a la vez ya que no hay que lidiar con el número de canales, al contrario que con los efectos de sonido.

Este proyecto cuenta con tres clases para representar y manejar sonidos:

- `SoundEffect`: Representa un efecto de sonido y contiene un identificador para diferenciado de otro sonidos.
- `MusicEffect`: Representa un sonido de música de fondo. Al igual que `SoundEffect`, contiene un identificador para diferenciarse.
- `SoundManager`: Manager `Singleton` encargado de implementar las funciones principales para reproducir, parar, y detener sonidos, entre otros. También métodos destinados al usuario para modificar el volumen general y cambiar el número de canales disponibles para la reproducción de efectos de sonidos.

4.1.6. Input

Este proyecto tiene como objetivo implementar un manager, también `Singleton`, que contendrá la información del estado de las teclas/botones de los dispositivos de entrada. En concreto, cuenta con soporte para teclado, ratón y mando.

En el manager, las teclas/botones pueden pasar por diferentes estados los cuales se establecen al recibir determinados eventos.

Estos estados se dividen en:

- *Down*: Una tecla esta siendo pulsada.
- *Up*: Una tecla esta soltada.
- *Pressed*: Una tecla acaba de ser pulsada.
- *Released*: Una tecla acaba de ser soltada.

Algo a tener en cuenta es que debido a la posibilidad de tener varios mandos conectados, el manager diferencia entre métodos con identificador y métodos sin identificador. Los métodos con identificador reciben el identificador del mando del que se quiere consultar el estado y los métodos sin identificador devuelven la información del estado del último mando que registró input. De esa manera, si se quiere desarrollar un juego monojugador, el usuario no tendrá que preocuparse por la posibilidad de múltiples mandos teniendo que indicar que identificador tiene su mando.

Además, el manager tiene soporte para conexiones y desconexiones durante la ejecución, e implementa métodos con lógica usada frecuentemente como lo es el movimiento horizontal para facilitar su implementación al usuario.

4.1.7. Físicas

Este proyecto tiene como objetivo implementar un envoltorio sobre la librería de físicas Box2D [9] para proporcionar una API sencilla para el usuario y para desarrollar los componentes de colisión y movimiento físico necesarios [1].

La física esta representada en un "mundo físico" donde se pueden crear cuerpos afectados por la física. La simulación de este mundo tiene la peculiaridad de que se actualiza por intervalos de tiempo fijos, lo que se conoce como "paso físico". En estos "pasos físicos" se realizan cálculos de colisiones, se resuelven restricciones y se actualizan las posiciones y velocidades de los cuerpos. Además, para evitar que todos los cuerpos colisiones con todos, existen las capas de colisión. Cada cuerpo se encuentra en una capa y se puede configurar que capas colisionan con que otras.

Las clases que tiene este proyecto son las siguientes:

- `PhysicsManager`: Clase, `Singleton`, que contiene la funcionalidad necesariara para manejar el filtrado de colisiones e información sobre gravedad del mundo físico.

- `DebugDraw`: Clase que contiene la funcionalidad para dibujar los cuerpos físicos de Box2D. En concreto, puede dibujar polígonos, círculos, segmentos y puntos.

4.1.8. Renderer

Este proyecto es el que se encarga de mostrar los sprites y fuentes en pantalla, así como limpiar la pantalla o guardar información sobre ella. Hace uso de las librería de SDL, SDL_Image y SDL_TTF [8].

Las clases que se encargan de estas funciones son:

- *RendererManager*: Clase, `Singleton`, encargada de inicializar y cerrar la librería de SDL, SDL_Image y SDL_TTF. Contiene información y funcionalidad relacionada con la ventana como su tamaño, borde, icono, cursor, nombre y modo pantalla completa. Además, proporciona los métodos para renderizar y para limpiar la pantalla.
- *Font*: Representa una fuente de texto y tiene la funcionalidad de crear una a partir de una fichero con *'.ttf'* como extensión. Tiene también la funcionalidad de crear un texto o un texto ajustado mediante la creación de una textura.
- *Texture*: Representa una textura y tiene la funcionalidad de crear una a partir de un fichero con una extensión de imagen como *'.png'* o *'.jpg'*.

4.1.9. Arquitectura de gameplay

Como arquitectura de gameplay se ha implementado de tipo entidades y componentes (EC) descrito en 2.3.2.

Este es el proyecto implementa la arquitectura, componentes fundamentales para el usuario y una serie de managers como el de escenas, prefabs, referencias y overlay.

La partes fundamentales de esta arquitectura son las siguientes:

- *Component*: Clase que representa a un componente. Desde un componente se puede acceder a la entidad y escena que lo contiene y establecer su estado, es decir, activarlo o desactivarlo y eliminarlo. Además, contiene una serie de métodos virtuales preparados para ser implementados por los componentes que hereden de esta clase.
- *Entity*: clase que representa una entidad. Contiene una referencia a la escena en la que se encuentra, una lista de componentes y otra de scripts asociados a esta entidad. Tiene información sobre el nombre de la entidad, su estado, un identificador y su orden de renderizado.

- *Scene*: La última pieza que compone esta arquitectura son las escenas. Una escena es un conjunto de entidades. Es un concepto importante en los videojuegos ya que normalmente se quiere dividir el juego en estados como menús, gameplay, inventario, pantallas de carga, mapa, etc. Contiene información sobre su nombre y bastantes métodos comunes a las entidades y componentes.

Como hemos dicho anteriormente, las entidades funcionan como contenedores y los que realmente implementan la funcionalidad son los componentes. Las entidades tienen métodos para añadir componentes, consultar si contienen un componente y eliminarlos.

4.1.9.1. Managers y componentes

Por otro lado, en este proyecto se implementan también los managers necesarios para el funcionamiento del motor y los componentes fundamentales que el motor va a proporcionar al usuario.

En cuanto a los componentes:

- *Transform*: Contiene la información sobre la posición, rotación y escala de la entidad. Además implementa algunos métodos para rotar, escalar y mover la entidad.
- *Overlay*: Componente encargado de representar los elementos de la interfaz de usuario como textos, imágenes y botones.
- *Image*: Componente encargado de cargar una imagen y renderizarla en pantalla en la posición indicada por el Transform de la entidad. Para cargar la imagen hace uso del manager de recursos para reutilizar la imagen en caso de estar ya creada por otra entidad.
- *PhysicBody*: Componente encargado de crear un cuerpo físico de Box2D. Implementa la funcionalidad de sincronizar posición, rotación y escala del Transform de la entidad al cuerpo físico. Contiene la información sobre bastantes propiedades físicas como el tipo de cuerpo (estático, cinemático, dinámico), el rozamiento o la escala de la gravedad.
- *SoundEmitter*: Componente encargado de cargar un sonido e implementar métodos para reproducirlo, detenerlo, pausarlo, etc. Como se comentó anteriomente, SDL_Mixer dispone de un conjunto de canales para reproducir sonidos pero este componente es abstrae la necesidad de canales desde la perspectiva del usuario.
- *MusicEmitter*: Componente encargado de cargar música e implementar métodos para reproducirla, detenerla, pausarla, rebobinarla, etc.

- *ParticleSystem*: Componente encargado de implementar un sistema de partículas configurable. Tiene soporte para usar texturas para las partículas y moverlas con el motor de físicas Box2D.

- *Animation*: Componente encargado de implementar la lógica de reproducción de animaciones.

- *TopDownController*: Componente encargado de implementar un movimiento tipo Top-Down.

- *PlatformController*: Componente encargado de implmentar un movimiento de tipo plataformas.

  Estos dos útlimos componentes no son fundamentales ya que son de gameplay, pero aportan comodidad porque evitan al usuario tener que implementarlos usando el sistema de scripting.

  Mencionar también que es obligatorio que las entidades tengan al menos un componente, Transform u Overlay. Esto es así para poder posicionarlas en pantalla, aunque cambie la manera de dibujarlas.

En cuanto a los managers:

- *SceneManager*: encargado de manejar las escenas. Para ello, cuenta con una pila en la que va almacenando las escenas que se crean. La escena que se va actualizar en el juego es la que se encuentra en el top de la pila.

  Hay 5 operaciones que se pueden realizar:

  1. *Operación PUSH*: carga la escena y la añade al top de la pila.
  2. *Operación POP*: elimina la escena en el top de la pila y avisa, a la escena por debajo del top, si la hay, que va a empezar a actualizarse.
  3. *Operación POPANDPUSH*: realiza una operación `POP` y posteriormente una operación `PUSH`.
  4. *Operación CLEARANDPUSH*: vacia la pila de escenas y añade una nueva al top de la pila que va a empezar a actualizarse.
  5. *Operación CLEAR*: vacia la pila de escenas.

- *SceneLoader*: Clase encargado de leer la información de la escenas creadas en el editor. A la hora de construir la entidades diferencia entre entidades con `Transform` y entidades con `Overlay`.

- *PrefabsManager*: Encargado de cargar la información de los prefabs creados en el editor e implementar métodos para instanciar entidades a partir de la información de esos prefabs. Se diferencia entre prefabs con `Transform` y prefabs con `Overlay`.

- *RenderManager*: Encargado de renderizar por orden las entidades de la escena. A la hora de desarrollar el juego es deseable poder elegir el orden en el que se renderizan las entidades. Esto también se conoce como profundidad o z-order.

- *ReferencesManager*: Encargado de manejar una relación entre las entidades y sus identificadores. Necesario para evitar problemas al acceder a la referencia de una entidad, por ejemplo en caso de que la referencia sea errónea o dicha entidad haya sido eliminada.

4.1.10. Bucle principal

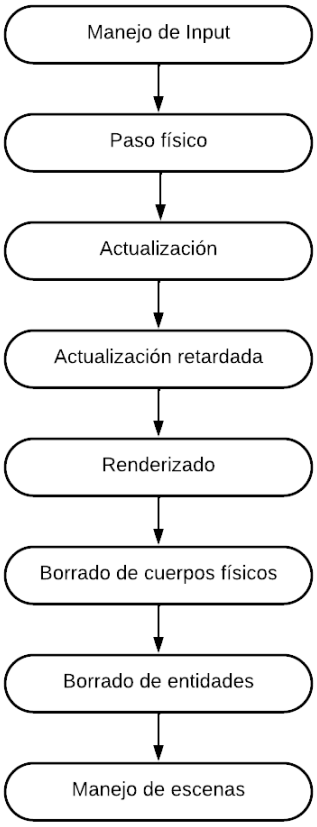

Figura 4.2: Estructura del bucle principal del motor.

Este proyecto implementa la clase `Engine`, encargada de inicializar el motor, ejecutar su bucle principal y cerrarlo una vez terminado.

La figura 4.2 muestra un diagrama con estructura del bucle principal de ejecución del motor.

Además de esos métodos, se realizan cálculos de tiempo para proporcionar al usuario el `DeltaTime`, el tiempo transcurrido desde el inicio de la ejecución del programa o el número de frames/actualizaciones hasta el momento. El `DeltaTime` es una medida de tiempo, generalmente en milisegundos, que informa sobre el tiempo transcurrido entre la iteración anterior y la actual.

Algo a comentar es la diferencia entre el "paso físico" y la "actualización". La librería de físicas Box2D requiere que la actualización del mundo físico se realice en intervalos de tiempo fijo, principalmente por motivos de estabilidad. Por ello, es necesario hacer cálculos adicionales para saber en que momentos se debe ejecutar el "paso físico" ya que no se puede llamar en cada frame, a diferencia de la "actualización".

La potencia del hardware de la computadora y la carga de trabajo afectan directamente al número de actualizaciones por segundo que se producen en el bucle principal de un videojuego. Por lo tanto, la llamada al método "actualización" se puede dar con mucha irregularidad. Sin embargo, el motor de física necesita intervalos de tiempo fijo.

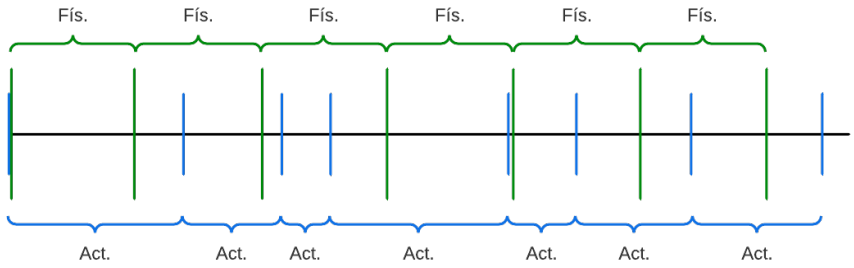

Figura 4.3: Diferencia entre la Actualización y el Paso físico.

Esto se explica mejor con el diagrama que se muestra en la figura 4.3:

Como se puede apreciar, el `Paso físico`, marcado en verde, siempre se ejecuta en el mismo intervalo de tiempo. Ese intervalo de tiempo fijo es un valor que se puede modificar en el motor en base a las necesidades del videojuego.

4.2. Cómo se serializa la información del motor

La lógica del juego funciona mediante la lectura de datos, los cuales tienen como función representar escenas y scripts. Analizando la forma en la que se carga una escena, lo que ocurre es que se crearán todas las entidades que la habiten, y a cada entidad se le añadirán sus componentes asignados. Por lo que en función de lo que se lea del fichero, el motor tiene que disponer de funcionalidad para identificar el tipo del componente para poder crearlo. Por ejemplo, a partir de una cadena de texto Ïmageël motor tiene que identificar que se trata del componente *Image* para poder crearlo. Además, las escenas cuentan con información relacionada con los atributos de los componentes de cada entidad, como puede ser la posición de la entidad para el componente *Transform*. Esta información está representada mediante una pareja de valores, ambos mediante cadenas de texto, usados para almacenar tanto el nombre del atributo como el valor que se le quiere dar. En el caso de la lectura de los scripts ocurre algo similar, pero en vez de tener que acceder a los atributos de los componentes, el motor tiene que ser capaz de invocar un método del componente dado conociendo únicamente su nombre.

Para solucionar estas situaciones, una posible solución sería implementar en el motor estructuras de datos mediante mapas. La idea es que estos mpasa nos permitan tanto crear componentes usando su nombre como clave, como acceder a punteros de los atributos de los componentes para poder modificarlos y también nos permita almacenar los punteros a los métodos de los componentes. Esta implementación mediante mapas tiene dos principales inconvenientes, el primero es la necesidad de tener que abstraer la información almacenada, de forma que en la misma estructura se puedan almacenar datos de diferente tipo. En el caso de la creación de componentes se puede solucionar mediante el uso de herencia y polimorfismo, pero en el caso de los atributos y métodos no resulta tan sencillo pues por ejemplo C++ no permite almacenar en la misma estructura punteros a métodos si pertencen a distintas clases (aunque ambas hereden de una clase común). El otro inconveniente sería el enorme trabajo que supone no solo el tener que programar todas la estructuras sino también el mantenerlas actualizadas cuando según se vaya añadiendo funcionalidad al motor.

En lenguajes de programación de alto nivel este problema tiene una fácil solución, y es mediante el uso de reflexión. La reflexión es la capacidad de dotar a un lenguaje con introspección, pudiendo conocer la estructura o incluso modificarla en tiempo de ejecución. En cualquier caso, esta solución no es aplicable a nuestro motor pues ha sido desarrollado C++, lenguaje el cual no tiene soporte para reflexión.

Para solventar esta carencia del lenguaje, el motor cuenta con un proyecto adicional llamado `ECSReader`. Este proyecto tiene la función de entender

la estructura de componentes para más adelante poder serializar esta información. Se trata de una herramienta interna del motor usada para acelerar nuestro desarrollo, la cual no tiene utilidad durante la ejecución del mismo, y debe ser ejecutada al modificar la funcionalidad pública del motor.

Podemos dividir el ciclo de vida de la ejecución de este programa en dos fases: la lectura de los datos del motor y en la generación de nuevos recursos.

4.2.1. Lectura del motor

El programa comienza leyendo el código fuente del motor, leyendo cada fichero y obteniendo de éste la información que considere necesaria. No es necesario almacenar toda la información de todas las clases que se encuentren, por ello antes de guardar los datos pasan por un proceso de filtrado, descrito a continuación.

- Almacena en una lista el nombre de todas las clases que hereden de componente, o que por el contrario hereden de uno como ocurren en el caso los componentes físicos.
- Para qué atributos de un componente deben guardarse se utiliza una etiqueta especial, `reflect`. Esta marca debe aparecer delante de cada atributo que se quiera poder serializar. De cada atributo se guarda tanto el nombre como el tipo del mismo.

```
publish:
    int getTextureWidth();
    int getTextureHeight();
private:
```

Figura 4.4: Ejemplo de uso de la etiqueta publish.

- Para el proceso de almacenar los métodos de un componente se utiliza otra etiqueta nueva, `publish`. Esta marca funciona como un nuevo modificador de acceso, de forma que todos los métodos que se encuentren entre dicha marca y la siguiente pasarán a ser almacenados. De los métodos se almacena tanto el nombre, como un vector con la entrada (almacenando tanto tipo como nombre) y el valor de salida.

```
reflect Vector2D size;
```

Figura 4.5: Ejemplo de uso de la etiqueta reflect.

Este uso de las etiquetas nos proporciona una gran comodidad y control en el desarrollo del motor, permitiéndonos controlar qué partes del motor queremos que sean expuestas. El motivo de la diferencia de uso de ambas etiquetas es para favorecer la encapsulación, pues en muchos componentes no queremos que haya atributos marcados como públicos pero sí queremos que se puedan modificar desde el editor, cosa que no es cierta en el caso de los métodos, pues toda esa funcionalidad debe ser pública para que el correcto funcionamiento del motor.

4.2.2. Recursos generados

Una vez todos los ficheros dados son leídos, comienza el proceso de generación de recursos. La función de estos recursos es de vital importancia para la comunicación del motor con el editor. A continuación se menciona cada fichero generado junto con su utilidad.

1. *FunctionManager*: Este fichero genera una función asociada a cada método que se haya leído, con la diferencia de que en el caso de que se trate de un método de un componente (en lugar de una método de un manager), se añade un parámetro adicional para acceder al puntero del componente. Esta nueva función tiene como salida una estructura abstracta, la cual puede tomar cualquier valor que queramos que sea soportado por el scripting. La función toma como entrada un vector con esta misma estructura abstracta. En la ejecución de la función se comprueba que la entrada dada sea correcta, y en ese caso hace una llamada al método asociado. Además, este fichero genera un mapa que contiene todas las funciones generadas, con su nombre como clave.

   Con este fichero podemos invocar a la funcionalidad del motor conociendo únicamente el nombre de cada método, lo cual es útil para el intérprete del lenguaje de scripting. Por otro lado, el hecho de haber convertido cada método en funciones, y todas con la misma estructura nos permite poder agrupar todas las funciones en un único mapa.

2. *ClassReflection*:Esta clase se encarga de manipular los atributos de cada componente, simulando el uso de reflexión. Con esta clase se puede modificar cada atributo conociendo únicamente su nombre, lo cual es importante durante el proceso de deserialización de una escena.

3. *ComponentFactory*: Esta clase tiene la función de generar un componente de un tipo determinado dado el nombre del tipo, funcionando a modo de factoría, siendo de vital importancia para el proceso de deserialización de una escena.

4. *Ficheros para el editor*: Se generan dos ficheros, uno para componentes y otro para managers, ambos en formato JSON. Estos contienen toda la información relacionada con cada componente y manager, y se trata de los ficheros utilizados para la lectura de los datos del motor en el editor.

Si bien estos ficheros podrían ser generados a mano, esta automatización supone un enorme ahorro de tiempo para el desarrollador.

4.3. Funcionamiento del lenguaje de scripting

Este lenguaje se trata de un sistema basado en nodos pensando principalmente para ser usado de forma visual, inspirado en el sistema de Blueprints de Unreal. Para entender el lenguaje conviene primero entender el concepto de nodo. Un nodo es una agrupación lógica con una función determinada, pudiendo utilizarse para represetar un dato, ejecutar operaciones aritméticas o llamar a funcionalidad dentro del motor. Estos nodos pueden conectarse entre sí, de forma que un nodo que represente una función se le pueda pasar como parámetros de entrada otros nodos. En los casos de nodos que representen funciones, dependiendo de su funcionalidad puede tener también un valor de salida, pudiendo enlazar la salida de una función con la entrada de otra, de esta forma creando comportamientos más complejos.

La ejecución del lenguaje funciona siguiendo un flujo, es decir, poder establecer el orden en el que se ejecutan los nodos, garantizando que el script se comporte siempre de la misma forma, lo cual es importante para casos donde tenemos varias funciones que queremos que se ejecuten de forma secuencial, sin una dependencia directa entre sus valores de entrada y salida. Este lenguaje de scripting está pensado para funcionar mendiante eventos, como vemos en el apartado 2.5.1, los cuales son señales enviadas por el motor tras cumplirse una condición, como puede ser en la colisión de dos entidades o la inicialización de la entidad. El lenguaje da soporte para marcar qué nodo queremos ejecutar en respuesta al evento recibido. Estos eventos vienen representados por nodos y funcionan como punto de partida inicial para la ejecución del script.

La ejecución de un script sigue la siguiente estructura. Primero se ejecuta el nodo inicial marcado por el evento, y antes de procesar dicho nodo se ejecutan cada una de sus entradas. A su vez, estas entradas siguen el mismo proceso ejecutando primero sus nodos de entrada correspondientes hasta dar con un nodo que no tenga entrada. Cuando todas las entradas y sub-entradas han sido procesadas es entonces cuando se ejecuta el propio nodo, el cual pasará el flujo al nodo siguiente. Este proceso se repite hasta dar con un nodo que no tenga ningún nodo siguiente. Durante este proceso un mismo

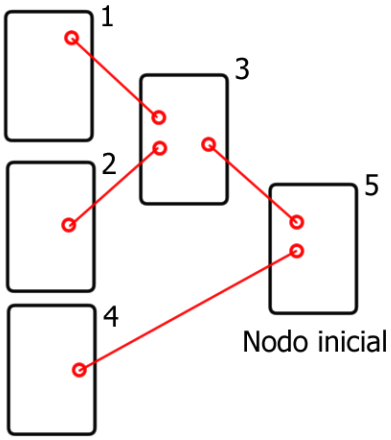

Figura 4.6: Orden de ejecución de nodos de un script. El número situado junto a cada nodo es el orden en el que se ejecutaría.

nodo no puede ser ejecutado dos veces (con la excepción de bucles), y el valor de salida de cada nodo debe ser guardado hasta que termine todo el proceso de ejecución. De esta forma se consigue optimizar la velocidad del lenguaje pues se reduce el número de nodos que son ejecutados, además de mejorar su legibilidad y previsibilidad (por ejemplo en un caso donde un nodo genere un número aleatorio, poder usar el mismo número con cada acceso al nodo), al precio de no permitir hacer programas que funcionen de forma recursiva.

Puede parecer poco intuitivo que el nodo que queremos ejecutar al principio sea el último en ejecutarse, por lo que para facilitar su entendimiento haremos uso de un ejemplo. Supongamos que queremos hacer un programa que escriba por consola el resultado de una suma en cada iteración. Para ello usaremos una estructura similar a la figura 4.6 descartando el cuarto nodo. Nuestro nodo inicial sería el nodo encargado de escribir por consola, y aunque sea el nodo directamente enlazado al evento, no se puede ejecutar inmediatamente pues para ello aún hay que procesar el resultado de la suma, la cual a su vez, primero tiene que procesar los dos valores que se quieran sumar. No se puede procesar un nodo sin haber procesado su entrada anteriormente, por ello,

Existen tres tipos de nodos principales:

1. *Nodo de función*: Este nodo representa una función en el lenguaje, y puede tener entrada y un valor de salida, al igual que puede recibir y pasar el flujo del script. Este nodo puede ser serializado para ser llamado desde otro script, creando así algo semejante a un método en un lenguaje convencional.

2. *Nodo de entrada*: Proporciona un valor constante que puede ser usado como entrada para una función. Este valor puede ser serializado para modificarse desde fuera permitiendo reutilizar scripts con datos diferentes. Por ejemplo, un script que represente la vida de un enemigo, y queramos serializar la vida, de forma que con el uso del mismo script, podamos representar enemigos con número de vidas diferentes. Este tipo de nodo no puede recibir el flujo.

3. *Nodo de bifurcación*: Este tipo de nodo permite la modificación del flujo del script, pudiendo hacerlo mediante lógica condicional y bucles, su comportamiento dependiendo de su tipo interno. Recibe un valor de entrada a modo de condición y no tiene un valor de salida. En el caso de los bucles, los nodos que se ejecuten dentro del bucle sí que pueden ser ejecutados varias veces en la misma iteración del evento.

Todo esto permite al usuario crear comportamientos complejos de una manera visual sin tener que escribir ninguna línea de código.

4.4. Integración scripting en el motor

Para la integración del lenguaje en el motor aparece `Script`, un nuevo tipo de componente, junto con un nuevo manager, `ScriptManager`. Entre estas dos clases se establece el puente entre el motor y el funcionamiento del scripting. `ScriptManager` funciona tanto como un manejador de recursos de tipo script, de forma que no se generen copias innecesarias del recurso cuando el script sea reutilizado por otra entidad, al igual que se encarga de su correcta deserialización del formato JSON. Además, también aporta funcionalidad básica para el correcto funcionamiento del scripting. Por otro lado, la clase `Script` proporciona los eventos necesarios al lenguaje de scripting, redirigiendo los eventos recibidos de la clase componente, además de guardar información adicional para almacenar los distintos valores serializados. La clase `Script` no maneja los nodos, sino que almacena punteros al nodo inicial de cada evento, en caso de que exista.

Para la implementación de la lógica del scripting, se ha creado una clase por cada tipo de nodo, siendo responsable cada uno de manejar su propia información, y entre todos creando una jerarquía de clases para poder manejar el script mediante herencia y polimorfismo. Estas clases sirven para representar internamente un script.

Para representar los valores dentro del motor, manejamos la abstracción de una variable la cual puede tomar cualquier valor, y todo se procesa dentro de una unión (siendo una unión un tipo especial de clase en C++ que solo puede almacenar el valor de uno de sus miembros). El lenguaje está preparado para soportar los tipos: float, char, string, bool, un vector bidimensional

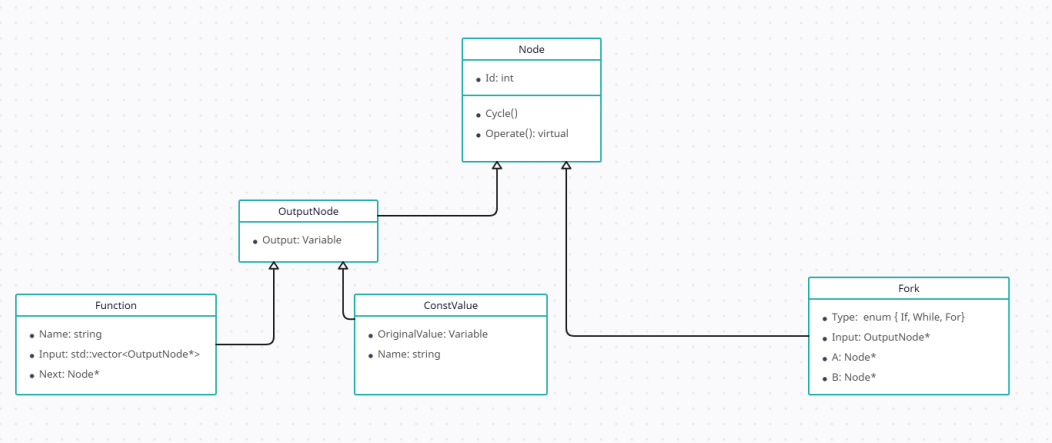

Figura 4.7: Diagrama UML con la estructura de clases de los nodos del scriting

de float, una estructura para representar un color y un puntero a una entidad. Con el fin de simplificar el uso, hemos decidido no distinguir entre tipos númericos (como pueden ser el int o el float) sino manejar una agrupación de ellos, algo parecido al funcionamiento de Javascript[1].

Con el mismo fin de simplificar las cosas, también hemos decidido descartar la capacidad de almacenar un variable de tipo `Component`. Por ello, en las llamadas a funciones donde hiciera falta un parámetro de tipo `Componente`, se pasará de tipo entidad, desde la cual se accederá al componente necesario.

Para el tipo entidad hay dos casos de uso. El primer caso ocurre cuando el puntero se consigue en tiempo de ejecución (por ejemplo al acceder al padre de la entidad actual o acceder a la entidad con la que se ha colisionado), en cuyo caso no se requeriría de ningún paso adicional para su uso, pero en el otro caso, la referencia a esta entidad se consigue desde el proceso de deserialización de la escena, donde al no poder guardar directamente un puntero se guarda el identificador de la entidad. Haciendo uso de este identificador y de *ReferencesManager* (sección 4.1.9: Managers y componentes) se puede acceder al puntero correspondiente.

Para la ejecución de un nodo función, éste almacena el nombre de la función a ejecutar, con lo que se puede ejecutar la función asociada haciendo uso del *FunctionManager*[1].

Por último, para completar el lenguaje se ha creado la clase *ScriptFunctionality*. Esta clase forma parte del motor y se encarga de proporcionar funcionalidad básica que no es aportada por ningún componente, como operaciones aritméticas o escribir por consola.

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/`

Global_Objects/Number

Capítulo 5

Editor

5.1. Funcionamiento y arquitectura

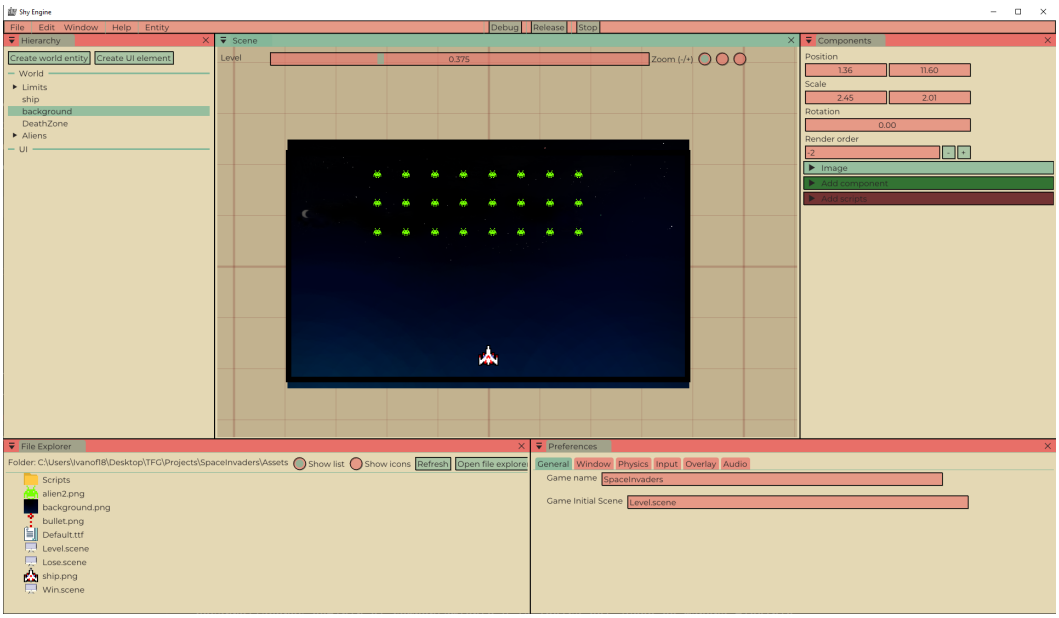

Figura 5.1: Editor de ShyEngine.

Debido a la elección de utilizar ImGui (ImGui es el nombre correcto de la biblioteca), una biblioteca de interfaz de usuario (UI) diseñada específicamente para C++, nuestro editor, ShyEditor, está desarrollado en C++ a pesar de no tener el motor integrado. Esta elección nos brindó acceso a una amplia gama de funcionalidades esenciales para la creación del editor, como la creación de ventanas, dropdowns o checkboxes, entre otras.

En la figura 5.1 podemos ver una imagen de la vista principal de nuestro editor.

En cuanto a su funcionamiento, el editor se compone de tres secciones principales:

1. *Gestor de proyectos*: Esta sección es la primera en iniciarse. En ella el usuario puede crear un proyecto o cargar uno ya creado.
2. *Editor principal*: El núcleo del editor es esta sección, donde se lleva a cabo la mayoría de la creación y edición de contenido. Aquí se manejan las entidades, la jerarquía, los componentes y los assets del juego.
3. *Edición de scripts*: Sección donde se maneja la creación y edición de scripts. Permite a los desarrolladores definir el comportamiento y la lógica del juego.

El flujo de trabajo del editor comienza con la ventana del gestor de proyectos. Después se traslada a la ventana de el editor principal desde donde puede abrirse la de edición de scripts.

En la figura 5.2 podemos ver dicho flujo.

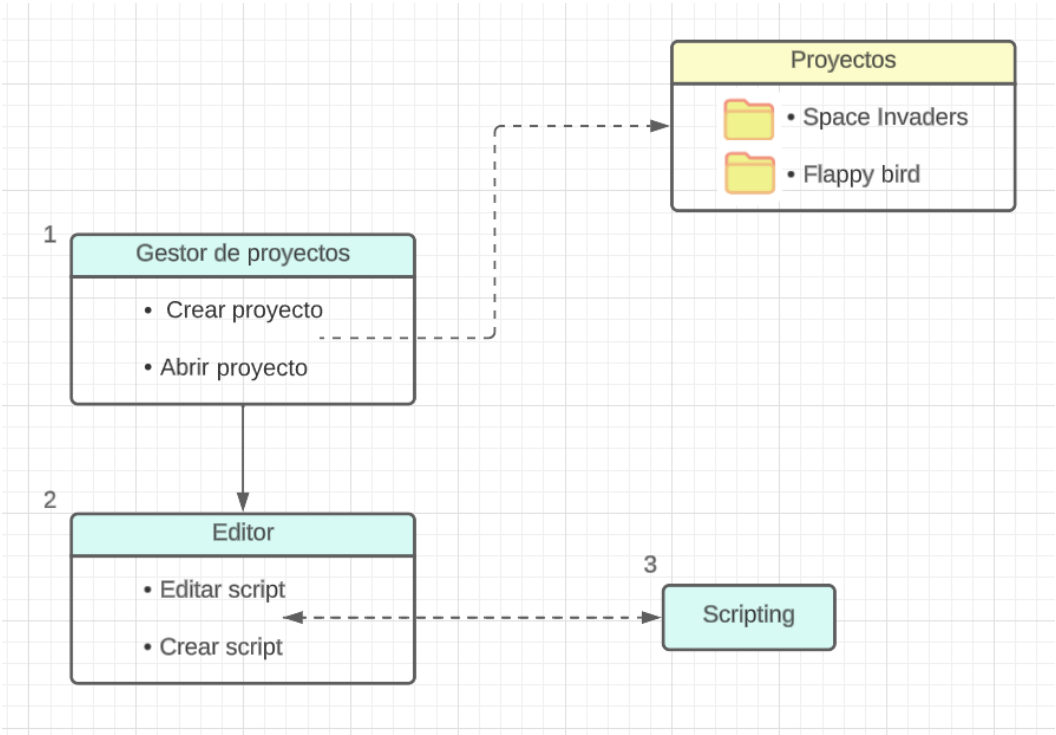

Figura 5.2: Flujo de funcionamiento del editor.

En cuanto al funcionamiento de las ventanas, cada ventana del editor hereda de una clase padre que proporciona el funcionamiento básico necesario

en ImGUI y que sirve como base para definir el comportamiento específico de cada ventana de la escena.

La sección del editor principal cuenta con varias ventanas:

- *Scene*: Esta ventana es en la que se previsualizan todas las entidades de la escena.
- *Hierarchy*: En esta ventana se puede ver un listado de todas las entidades que contiene la escena, así como permite gestionar la jerarquía entre ellas.
- *Components*: Desde esta ventana se puede visualizar, modificar, y añadir componentes y scripts a las entidades.
- *File Explorer*: Esta ventana permite navegar por los directorios de nuestro proyecto para ver y manejar nuestros assets, y realizar acciones como cargar una escena guardada.

Aunque también dispone de otras ventanas más secundarias como la de preferencias, la de gestión de prefabs, la de edición de la paleta de colores, o la consola.

En la figura 5.3 observamos esta arquitectura de ventanas.

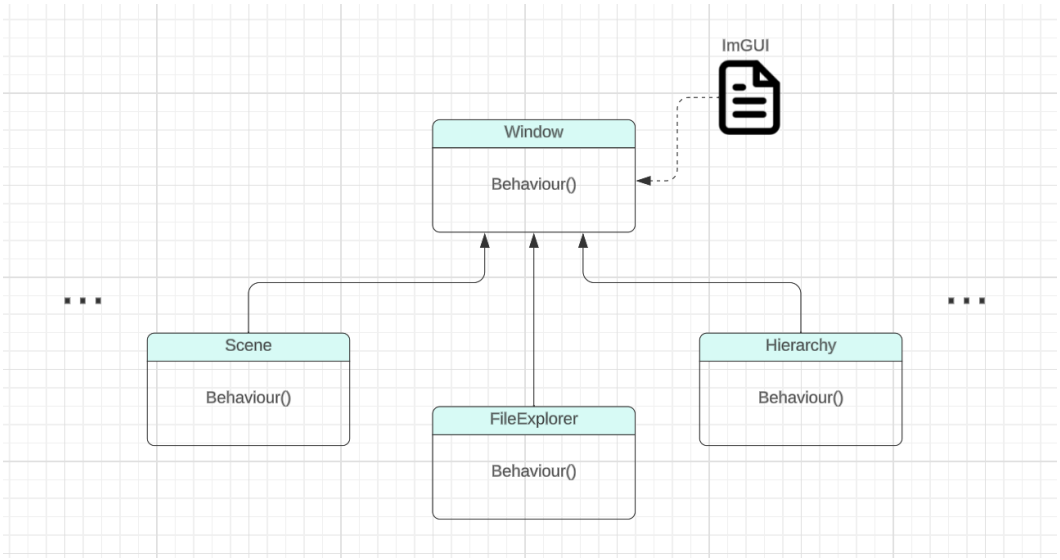

Figura 5.3: Arquitectura de las ventanas.

Cabe destacar que el editor también soporta la modificación del layout de ventanas y personalización de la paleta de colores, como se ve en la figura 5.4.

En relación a la gestión de entidades, cada una de ellas se identifica mediante un ID único. Este ID desempeña un papel crucial al permitir la distinción entre las distintas entidades y se utiliza, por ejemplo, para referenciarlas en scripts. Además, cada entidad puede contener una referencia a su entidad padre o a sus entidades hijas, en caso de que existan relaciones jerárquicas.

Un aspecto importante del sistema de IDs es que también se utiliza para diferenciar las entidades normales de los prefabs. Los prefabs se distinguen por tener un ID negativo, lo que les permite ser identificados de manera única.

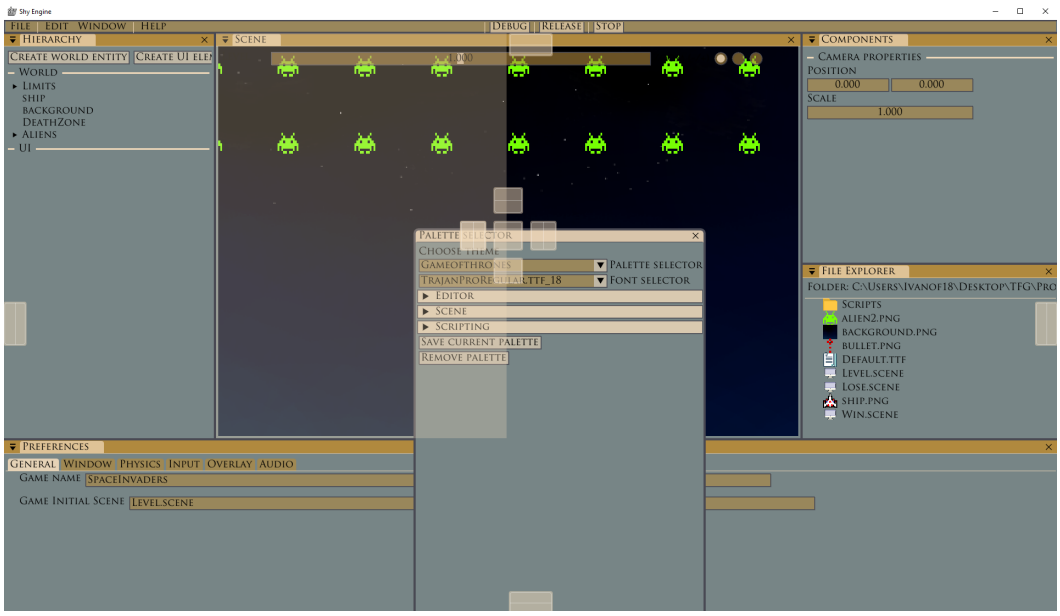

Figura 5.4: Personalización del editor.

5.2. Como se leen los componentes a partir de los datos del motor

Como se mencionó anteriormente en la sección 4.2.2, el motor del cuenta con una herramienta que serializa en formato JSON los nombres de los atributos correspondientes a sus componentes nativos.

Estos datos en formato JSON se interpretan y se utilizan para ser editados desde la ventana de *Components*.

En la imagen 5.5 se puede observar el funcionamiento de la lectura de los atributos de los componentes del motor.

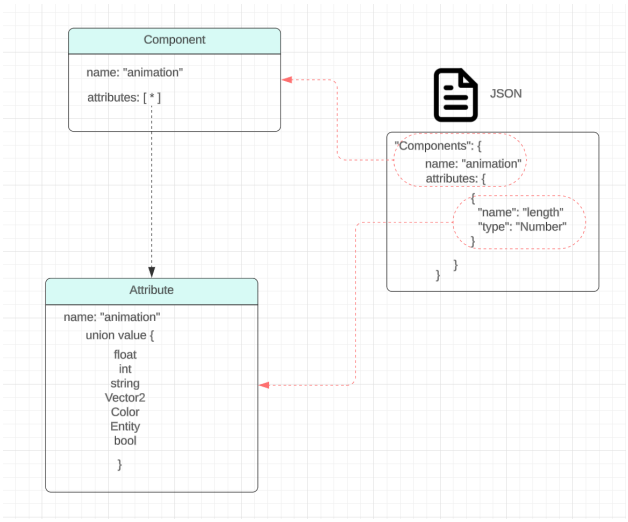

Figura 5.5: Lectura de componentes del motor en el editor.

5.3. Como se trasladan los datos del editor al motor

Una vez creado el juego, llega la parte de pasarle toda la información al motor. Para ello, se serializa toda la información relativa a la escena, como su nombre y sus entidades, en un formato JSON que luego el motor se encargará de interpretar, como se menciona en 4.1.9.

Cabe destacar que hay información como las preferencias o las entidades que son prefabs se serializan en el archivo ".shyproject" (de igual manera que con la escena), ya que son elementos del proyecto y no de una escena en concreto.

La figura 5.6 representa como se serializa la información que será trasladada al motor.

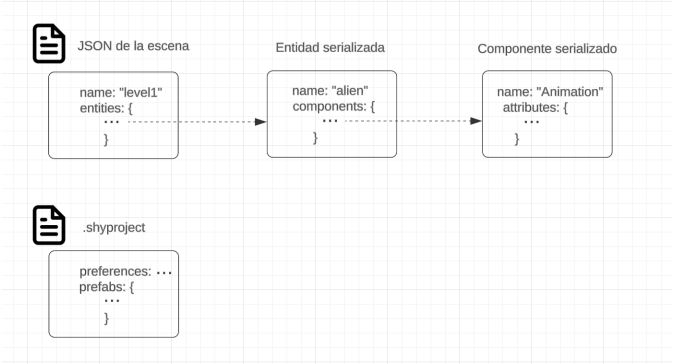

Figura 5.6: Serialización de la información del juego.

5.4. Scripting en el editor

Como hemos mencionado anteriormente, el editor contiene una sección completamente dedicada a la creación y modificación de scripts.

En el editor, los scripts se crean utilizando un sistema de programación visual (sección 2.5) mediante nodos y conexiones. Los nodos representan acciones o eventos (sección 2.5.1) específicos que ocurren en el juego, como el inicio de una escena, la colisión entre objetos o la activación de una trampa. Estos nodos se organizan en el espacio de trabajo del editor y se conectan entre sí para definir la secuencia y lógica del comportamiento.

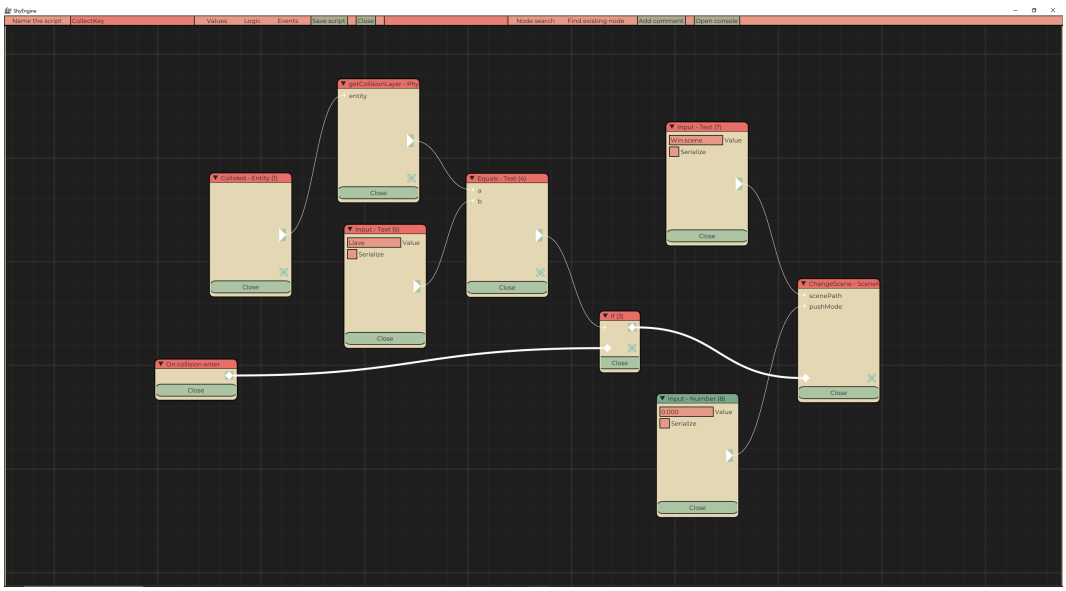

Figura 5.7: Ejemplo de scripting en el editor.

5.4.1. Serialización y uso en el motor

El flujo de nodos y conexiones en el editor es una representación visual de un script, pero esta información debe serializarse en un formato que el motor del juego pueda entender y ejecutar. Para lograr esto, el editor guarda el script en un archivo en formato JSON. Este archivo contiene toda la información sobre los nodos, conexiones y sus propiedades asociadas. Mencionar también que en la serialización de las entidades, estás contienen información sobre que scripts tienen asignado.

5.4.2. Edición de scripts

Además de crear nuevos scripts, el editor también permite la modificación de scripts existentes. Para ello, realiza una lectura e interpretación de los JSON mencionados anteriormente de manera similar a como hacen los componentes.

5.5. Ejecución del juego y generación de una build

Una funcionalidad a destacar del editor es la posibilidad de ejecutar el juego desde este en otra ventana. Esto se realiza ejecutando el ".exe" del motor mediante manejadores de procesos de Windows, para poder pararlo desde el editor y evitar varios procesos del juego simultáneamente, así como redirigir la salida del juego a la consola del editor. Además tambien se puede generar una build ejecutable del juego mediante el uso de un hilo. Para ello, se agrupan todos los assets del juego, junto con el ".exe" (con su correspondiente nombre e icono aportados por el usuario) y los ficheros de configuración del juego generados por el editor. Esto permite a los desarrolladores crear una versión independiente del juego que puede ser distribuida y ejecutada por los usuarios finales.

Capítulo 6

Uso del entorno de desarrollo

Es difícil comprobar la calidad de un entorno de desarrollo por sí mismo sin tener algún videojuego de ejemplo donde poder ver sus capacidades. No solo esto, sino que hacer uso del entorno de desarrollo nos permite identificar errores para poder solucionarlos más adelante, además de pulir detalles que no se tuvieron en cuenta durante el proceso de desarrollo para poder ofrecer una mejor experiencia de usuario.

6.1. Videojuegos de ejemplo

Haciendo uso del entorno de desarrollo hemos desarrollado varios prototipos de videojuegos a modo de ejemplo para demostrar las capacidades tanto del editor como del motor. Estos proyectos nos han permitido poner a prueba el potencial y la versatilidad de nuestro motor, así como demostrar su capacidad para crear una variedad de experiencias de juego. A continuación, destacamos algunos de los videojuegos que hemos desarrollado utilizando nuestras propias herramientas:

6.1.1. Space Invaders

Como primer juego hemos desarrollado una simplificación del clásico videojuego de máquinas recreativas Space Invader diseñado por Tomohiro Nishikado, donde el objetivo es destruir las naves enemigas antes de que estas lleguen al suelo.

Con este prototipo por un lado se puede apreciar la capacidad del editor de representar entidades en la escena, agrupar entidades haciendo uso de una jerarquía, y poder crear y modificar prefabs; y por otro lado se puede apreciar la capacidad del motor de instanciar entidades mediante prefabs en tiempo de ejecución, un manejo de entrada por teclado básico, uso de físicas simples y la capacidad de cambiar de escena. En la figura 6.1 observamos una captura

del juego y en `https://www.youtube.com/watch?v=DOdN1IPDAzQ` un video de ejemplo.

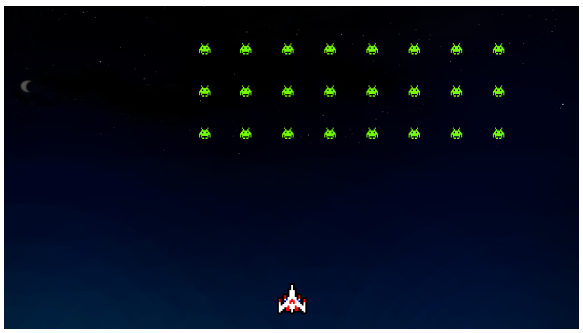

Figura 6.1: Space Invaders desarrollado con ShyEngine.

6.1.2. Plataformas genérico

El siguiente juego desarrollado se trata de un prototipo de videojuego de plataformas en 2D. El género de las plataformas consiste en tomar el control de un personaje y avanzar sorteando obstáculos haciendo uso de plataformas flotantes.

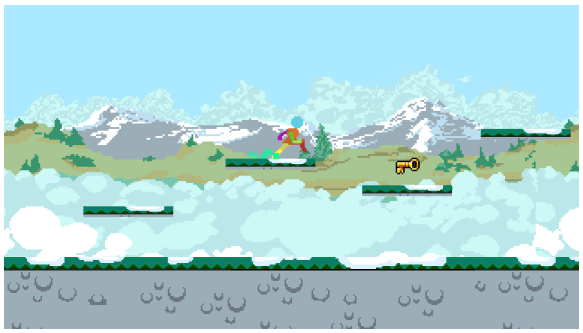

Figura 6.2: Juego de plataformas genérico desarrollado con ShyEngine.

Con este prototipo se pueden apreciar mejor las físicas, haciendo uso de conceptos más avanzados como el uso de capas físicas y el uso de eventos físicos. En este juego, el personaje avanza sin rumbo en un mapa lleno de movimientos con la capacidad de moverse tanto hacia la izquierda como a la derecha, saltar e incluso poder realizar un doble salto.

6.2. Prueba con usuarios

Debido a nuestro amplio conocimiento en profundidad tanto del editor como del motor, no podemos dar una opinión imparcial sobre su usabilidad o

comodidad. Del mismo modo, tampoco podemos decidir si el uso del entorno del trabajo es intuitivo y fácil de manejar para un usuario nuevo, sin experiencia previa en el desarrollo de videojuegos. Por ello, es importante hacer pruebas de uso con usuarios reales para poder rectificar y mejorar distintos elementos del entorno de desarrollo.

6.2.1. Objetivos

Nuestro objetivo de la prueba con usuarios es ver si el entorno de desarrollo se ajusta a las necesidades de un usuario nuevo, de forma que con unos conocimientos básicos sea capaz de desarrollar por su cuenta un prototipo de videojuego sencillo, haciendo uso de las distintas herramientas que le son proporcionadas.

6.2.2. Selección de usuarios

Nuestro entorno de desarrollo está pensado para que pueda ser usado alguien sin previa experiencia en el desarrollo de videojuegos, aunque sí que esperamos que se tenga un mínimo de conocimientos previos de informática y manejo de ordenadores, y conocimientos básicos sobre videojuegos. Este será nuestro principal público objetivo, aunque también es interesante hacer pruebas con usuarios más experimentados, tanto con experiencia programando como desarrollando videojuegos, y ver cómo se desenvuelven con las herramientas dadas, y viendo si su conocimiento previo supone una gran diferencia a la hora de hacer juegos.

En total hemos conseguido reunir 10 personas, principalmente familiares y amigos, para participar en la prueba de usuarios. Este grupo pese a ser bastante heterogéneo en cuanto a sus conocimientos relacionados con la informática y la programación, lo podemos dividir en tres grupos principales: jugadores sin experiencia en la programación o desarrollo (4 usuarios), programadores sin experiencia en el desarrollo de videojuegos (4 usuarios) y programadores con experiencia desarrollando videojuegos en otros entornos (2 usuarios).

6.2.3. Proceso de pruebas

Las pruebas se realizan de manera individual, teniendo una duración aproximada de una hora y media en las que el usuario está acompañado por un supervisor. La prueba inicia leyendo por encima un guión preparado con anterioridad, de forma que todos los usuarios cuenten con la misma información inicial. En este guión se explica el objetivo de la prueba, así como distintos conceptos esenciales en el desarrollo de un videojuego, como qué es una entidad o qué es un componente. También se dará un breve repaso por la interfaz de usuario del editor. Una vez terminado el guión, se entrega

al usuario dos ficheros diferentes, el primero conteniendo el ejecutable del editor y otro con un paquete de recursos gratuitos para disponer de material inicial y no perder tiempo en búsqueda de assets.

El objetivo del investigador es tomar nota de las acciones significativas del usuario, así como ayudarle en caso de encontrar algún comportamiento inesperado o error que pueda dañar su experiencia. Quitando esas situaciones excepcionales, el investigador no deberá aportar más ayuda al usuario a no ser que éste se bloquee durante un gran periodo de tiempo, pues el objetivo no es frustrarle, además de que si el usuario se bloquea no podremos conseguir información útil. Idealmente, la prueba tendrá lugar de forma presencial, aunque en caso de no poder acordar una fecha, se hará por comodidad de forma telemática. Una vez finalizada la prueba, se realizará al usuario una breve encuesta donde podrá valorar distintos aspectos tanto del editor, motor y scripting, finalizando con una valoración abierta donde el usuario podrá aportar sus quejas, ideas y opinión general.

6.2.4. Resultados

Las pruebas se realizaron a lo largo de varios días, todas hechas por el mismo investigador. En cuanto a los juegos desarrollados, la mayor parte de los usuarios decidió hacer un videojuego de plataformas, usando los assets proporcionados. El grupo de los usuarios sin experiencia fue el que más problemas tuvo, sobre todo al principio, teniendo el investigador que acabar ayudándoles en varias ocasiones. El problema más frecuente fue el poco entendimiento del funcionamiento de los componentes. Una vez pasado este primer obstáculo, el resto de la prueba fue más fluida, utilizando principalmente los componentes predefinidos por el motor en vez de aventurarse a crear su propio script, aunque cabe destacar un usuario que se atrevió a desarrollar script más complejos con relativo éxito.

El segundo grupo, formado por usuarios con experiencia programando pero no desarrollando videojuegos, fue bastante mejor, con menos bloqueos que el primero, con más gente atreviéndose a diseñar scripts más complejos, aunque hubo varios que tras varios fracasos se frustraron y terminaron dejándolo con el fin de no perder más tiempo de la prueba. En este grupo hubo bloqueos más variados, aunque ninguno tan fuerte como los bloqueos del primer grupo.

El grupo con experiencia en el desarrollo de videojuegos fue el que más avances hizo con diferencia, asociando rápidamente conceptos vistos en otros motores al nuestro. El único bloqueo que hubo fue por parte de usuario que no terminaba de entender el funcionamiento de los prefabs en nuestro entorno de desarrollo, aunque después de ayudarle con eso no terminó usándolos.

En cuanto al feedback recibido, hemos recibido bastantes quejas y suge-

rencias principalmente sobre el editor. La mayor parte de los usuarios confiesa haberse perdido por la interfaz en algún momento, y otro considerable porcentaje considera los scripts una herramienta bastante confusa la primera vez que interactúan con ella y la mayoría considera que para realizar la prueba sería necesario aportar más información y tiempo.

Por último, hay que mencionar que gracias a realizar las pruebas hemos encontrado bugs críticos tanto en el motor como en el editor, algunos gracias a realizar las pruebas en el propio ordenador del usuario, permitiendo ver el comportamiento del editor con diferentes resoluciones y programas instalados, y a usos del usuario del editor que no habíamos contemplado en un principio.

6.2.5. Conclusiones

Lo primero a mencionar es que nuestra cantidad de usuarios no es lo suficientemente grande como para sacar resultados concluyentes, pero viendo la tendencia general de todos los usuarios podemos estimar algunas conclusiones.

Consideramos que las pruebas con usuarios han sido realmente útiles, permitiéndonos ver que conceptos que dábamos por hecho que serían intuitivos realmente no lo son, como el concepto de transformación y nos han servido para tomar nota de muchos posibles cambios y mejoras que habría que implementar para mejorar la calidad de la experiencia de usuario, como anotaciones en partes complejas del editor que pueden no llegar a entenderse.

Creemos que el principal problema con nuestro entorno de desarrollo es la falta de información que se proporciona al usuario, lo cual es especialmente negativo para los usuarios nuevos. Por ello hemos pensado en implementar varias posibles correcciones, añadiendo más información en forma de marcadores de ayuda por cada elemento del editor que consideremos oportuno en base a nuestras observaciones. Además, creemos importante el implementar una documentación donde recordar los elementos principales del editor, así como poder profundizar sobre el funcionamiento del scripting. Por último, creemos de vital importancia hacer un tutorial (ya bien en formato web o vídeo) donde crear un videojuego sencillo desde cero, que pueda ir siguiendo el usuario para que así pueda ver en acción el funcionamiento de todas las partes del editor. Con estos cambios es probable que no se solucione por completo el problema, pero sería una gran ayuda para sobrepasar la brecha de conocimiento inicial necesaria para usar el entorno de desarrollo.

Capítulo 7

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.

7.1. 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 para leer los datos que genera el editor, como 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.

7.2. Yojhan Steven 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. Tuve que buscar información de otros motores como Unity o Unreal y cómo funciona su scripting para implementarlo en nuestro motor, aunque tampoco queríamos copiar su forma de implementar el scripting o el diseño de su estructura de nodos y queríamos desarrollar un lenguaje propio que se adaptase a las necesidades concretas de nuestro motor. En esta fase inicial se tuvieron que tomar las decisiones clave sobre cómo implementar el scripting, como por ejemplo hacer que el lenguaje de scripting fuese compilado a código en C++ , o si debería ser interpretado.

Fue un proceso iterativo donde poco a poco se iba añadiendo funcionalidad y haciendo una variedad de prototipos, usando Unity como base para la parte gráfica. En estos prototipos fue cuando probamos a hacer que un script fuera compilado, de forma que se mejorase el rendimiento del motor. Para ello creé lógica para su traducción a código en C++, con éxito en scripts pequeños que fueron probados. El siguiente paso sería compilarlo, y después de investigar como llevar a cabo el proceso de compilación para generar una DLL la cual sería enlazada con el motor, no vimos ninguna solución que nos terminase de convencer, pues la mayoría requería de instalación de programas de compilación adicionales o no tenían soporte para la generación de DLLs, por lo que al final tras discutirlo los miembros del equipo decidimos que no se compilase el script sino que fuese interpretado por el motor.

Cuando dimos con una implementación suficientemente robusta fue cuando decidímos juntar los dos primeros proyectos del TFG, siendo el motor y el scripting. No bastaba solo con incluir los ficheros que tenía (adaptando algunos de C# a C++ y añadiendo la biblioteca de JSON de nlohmann) [**?**], sino que para poder llevar a cabo la integración tuve que crear una serie de clases del motor para que funcionasen como puente para el scripting. Esto incluye clases como Script y ScriptManager imprescindibles para poder llevar a cabo la unión. Con el scripting enlazado con el bucle de juego del motor fue momento de empezar a hacer pruebas y ver que todo funcionaba como se esperaba. En este punto fue necesaria la creación de la clase ScriptFunctionality para poder dotar al lenguaje de una funcionalidad básica operaciones aritméticas o salida por consola para poder llevar a cabo las pruebas.

El scripting se encontró con un bloqueo pues no se podía acceder a la funcionalidad del motor, por lo que pospuse el desarrollo del scripting y empezé

a centrarme en cómo sería la unión entre el editor y el motor. Después de investigar posibles soluciones y ver la carencias del lenguaje C++, decidimos solventar esas carencias a mano, por lo que surgió el proyecto de ECSReader, con el cual se puede leer el contenido del motor gracias a unas marcas que se pueden ir añadiéndo en el código. Estas marcas permiten al desarrollador decidir que partes se quieren hacer públicas para el editor. Con esta clase terminada, toda la información relevantes del motor pasaba a estar serializada en un fichero en formato JSON, siendo la idea que el editor pueda leer los valores desde este fichero. Además, se usó el ECSReader ya no solo para generar ficheros JSON, sino para automatizar distintas partes del motor, ahorrando mucho tiempo de haber hecho eso a mano, como crear una factoría de componentes automática, permitir hacer reflexión sobre los atributos de un componente, y una parte fundamental para continuar el desarrollo del scripting, que es poder tener un mapa con todas las funciones del motor para así poder utilizar su funcionalidad desde un script. Con esto hecho, continué haciendo pruebas con los scripts, ahora manejando funcionalidad del motor y corrigiendo todos los errores que iba encontrando.

La mejor forma de comprobar que el scripting funcionaba era utilizándolo de forma exhaustiva, por lo que comencé el desarrollo de varios juegos pequeños para ver si encontraba alguna limitación. Estos juegos eran un pequeño flappy bird y otro un reloj que daba la hora en tiempo real (realmente no se podría decirse que es un juego sino más bien una prueba). El problema del desarrollo de scripts es que estaba enlazado con el motor, pero aún no con el editor, lo cual significaba que cada script lo tenía que hacer a mano, modificando el propio fichero desde un editor de texto, lo cual era bastante tedioso de hacer y se complicaba mucho cuando el script crecía de tamaño. Por ello, mi idea era implementar ya la ventana de edición de scripts en el editor, pero debido a que el editor aún no estaba lo suficientemente avanzado decidí ayudar a mi compañero en el desarrollo del motor.

Del motor ayudé a mi compañero implementando funcionalidad con la que se encontraba a mitad de desarrollo. Lo primero que añadí, debido a mi previa experiencia usando la biblioteca de JSON de nlohmann, fue la serialización de escenas y entidades. Con eso hecho me puse a implementar un sistema de Overlays, siendo los overlays los distintos elementos con los que cuenta el motor para representar la interfaz de usuario, junto con sus componentes correspondientes (OverlayImage, OverlayText y OverlayButton). Además, también hice la estructura para la deserialización de valores que carga el motor, con información como el nombre del juego o el tamaño de la ventana, dejando margen para poder ser expandido con más valores en un futuro. Implementé funcionalidad para otorgar al motor de una jerarquía de entidades. También implementé lógica para dotar al motor de una splashScreen con el logo del motor.

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. Implementé la lógica para mover la cámara por la escena, al igual que para poder cambiar su escala. Hice el renderManager dentro del ECS para poder controlar el orden de renderizado de las entidades. Por último, hice también un ConsoleManager para poder mostrar elementos por consola con un formato unificado, mostrando que componente o script lo ha escrito, al igual que el momento en el tiempo en el que se escribió el mensaje. Todo esto con su respectivo manejo de errores.

Cuando el editor ya se encontraba en un estado más avanzado fue cuando empecé la implementación del editor visual de nodos en el editor. En el momento en el que todo se encontraba más o menos listo, pudiendo crear scripts y poder serializarlos y aproveché para hacer hacer una refactorización general del sistema de ventanas, para permitir al usuario poder moverlas y agruparlas como mejor le pareciese utilizando la rama experimental de docking de ImGui. Implementé la lectura de los datos que genera el motor, con la clase ComponentReader y ComponentManager. Procedí a cambiar la ventana de la escena y la forma en la que se renderizaban las entidades además de añadiendo la opción de renderizar elementos de la interfaz. Con todo esto listo, me dediqué a añadir funcionalidad adicional en el editor, pudiendo ejecutar el juego en una ventana nueva, con la funcionalidad adicional de poder detener su ejecución desde el editor en cualquier momento. También añadí funcionalidad para poder crear una build, reuniendo los assets necesarios junto con el ejecutable del motor en una única carpeta, modificando el nombre e icono del fichero ejecutable. Otras ventanas que añadí fue el esqueleto básico de la ventana de preferencias, el manejador de docking para cambiar la distribución de la ventana y la ventana de modificación de paletas de colores.

Por últimos, estuvimos desarrollando juegos con el editor y el motor, corrigiendo los errores que nos íbamos encontrando. Además, fui quien preparó y llevó a cabo las pruebas con usuarios.

7.3. 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 Docking 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 assets 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. Después, 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 8

Conclusiones

En el desarrollo de nuestro motor de videojuegos, a pesar de ser un proyecto más simple en comparación con los motores líderes de la industria y contar con un equipo de solo tres miembros, hemos logrado crear una herramienta con un acabado bastante funcional y con elementos similares a la de otros motores de renombre en la industria. Además, aunque con margen de mejora, hemos acercado un paso más la programación de videojuegos a personas sin experiencia previa en el campo, lo que amplía las posibilidades de desarrollo de juegos para una audiencia más amplia.

Sin embargo, nuestro motor presenta ciertas limitaciones, como la incapacidad de mover las ventanas de ImGUI fuera de la ventana principal de SDL, lo que puede generar incomodidad en ciertas situaciones, como al implementar scripts. Además, hemos tenido que implementar reflexión en C++, lo que, en retrospectiva, podría haberse simplificado eligiendo un lenguaje reflexivo directamente. Un lenguaje de alto nivel además habría reducido la necesidad de gestionar la memoria manualmente en el proyecto del editor y haciendo más cómodo el desarrollo del mismo.

Mirando hacia el futuro, tenemos varias áreas de mejora y desarrollo potencial para nuestro motor:

- *Abstracción de rutas de ficheros*: Podemos trabajar en la abstracción de rutas de ficheros y directorios, convirtiendo los ficheros en objetos del motor. Esto simplificaría la gestión de recursos y proporcionaría más comodidad del usuario.
- *Añadir textos explicativo*: Incluir cuadros con texto explicativo en zonas que puedan resultar confusas para los usuarios, mejorando así la experiencia de usuario y facilitando la comprensión de la herramienta.
- *Documentación y tutoriales*: Como complemento a la adición de textos explicativos, también se podría desarrollar un apartado con documentación y enlaces a tutoriales que facilitan la curva de aprendizaje.

- *Mejorar el scripting*: Ampliar las capacidades del scripting permitiría a los usuarios expresar sus ideas de manera más completa. Esto podría incluir la adición de funcionalidades más complejas, como la gestión de arrays editables desde el editor, la creación de clases, la recursión, temporizadores, corutinas y la depuración de nodos en ejecución.
- *Estructuras de datos personalizadas*: Proporcionar a los usuarios la capacidad de crear estructuras de datos personalizadas y agrupar elementos dentro del sistema de scripting, lo que aumentaría la flexibilidad y las opciones de diseño.
- *Ejecución en el editor*: Permitir a los usuarios ejecutar sus juegos directamente en el propio editor en lugar de en una ventana separada, lo que agilizaría el proceso de desarrollo y prueba.
- *Mejora de la usabilidad de las entidades en la ventana de escena*: Trabajar en la mejora de la usabilidad de las entidades en la ventana de escena para que los usuarios puedan interactuar y gestionar sus elementos de manera más eficiente.

En resumen, aunque hemos logrado un motor de videojuegos funcional y accesible, reconocemos que siempre hay espacio para la mejora y la expansión. Nuestras metas para el futuro incluyen hacer que la herramienta sea aún más amigable para los usuarios y agregar características adicionales que permitan un desarrollo de juegos más completo y versátil.

Apéndice A

Abstract

The development of video games is a field that has undergone significant evolution over the years. Initially, video games were simple programs with basic behavior and very limited graphics. However, over time, and partly due to the evolution of technology, they have increased considerably in complexity and scope.

A game engine is software that provides the necessary technology to develop the gameplay of a video game. It's a coherent set of libraries (graphics, audio, physics, etc.) that abstracts technology for the developer, allowing them to focus on the game's development.

Nowadays, game development is a complex task. On one hand, it can be done from scratch, meaning programming the technology the game will require and then developing it. On the other hand, you can use a game engine that provides that technology independently of the specific needs of the game to be developed.

In some cases, in order to help the development, game editors are incorporated. The editors simplify development by linking the creation of game elements, behavior definition, behavioral game elements, behavior definition, debugging and generation of executable versions, among other things, in a single visual tool. This avoids having to communicate directly with the videogame engine through programming.

This presents a significant advantage for experienced developers, but engines like Unity or Unreal Engine can be overly complex for individuals with no programming experience, even if their goal is to create simple 2D games. Additionally, these are extensive systems designed to handle a wide range of game types, resulting in a lot of varied functionality and creating executable versions with a large amount of unnecessary data. If you want to create small and simple games, the overhead can be quite substantial.

This is where our work comes into play. The idea is to create an engine

with its editor for developing small games, where the development experience is equivalent to that of larger engine editors. However, it will be focused on smaller 2D game development and will have a much lighter load in terms of execution and generating executable versions. This will open the doors to our engine for developers with little programming experience or even those with no experience at all in game development.

Keywords: Video game editor, Video game development environment, Video game engine, Visual scripting, Node programming

Apéndice B

Introduction

B.1. Motivation

The development of video games has been a technical and organizational challenge since its inception. Especially at the technical level, it requires the expertise of computer computer engineers capable of developing the necessary technology that a video game may require. Development companies have a choice between developing the technologies needed to develop a video game or obtaining those technologies already developed and starting directly with the development.

In the first case, the disadvantage lies in developing these technologies in a way that is dependent on the videogame for which it is being developed, i.e., according to the specific characteristics of the videogame, in such a way that it cannot be reused for the development of another type of videogame.

In the second case, the disadvantage lies in the way of obtaining such technologies, licenses or costs. When developing these technologies, teams must draw the line teams must draw the line between the technologies to develop a videogame and the development of the videogame itself. This separation is necessary to ensure the reusability of the technologies.

These technologies are known as videogame engines. In general, they usually consist of several libraries dedicated to a specific purpose such as graphics, audio, physics, input, etc.

In some cases, in addition to the videogame engine, engine developers include editors. Editors are tools that simplify development by communicating the developer's actions to the engine. Some of their key functions include the definition of behaviors, the creation of assets and in-game elements, the creation of assets and elements in the game, debugging and generation of executable versions for different platforms.

There are different engines on the market with different licenses and

features. The most convenient for development are those with an editor, but an editor is very costly to develop, so the engines that provide it are those that have a very large critical mass of use, so that it is worth the development effort. This only happens in generalist engines that allow to realize games of many types and with very good quality. The price to pay for using these engines is their complexity, both in the use of the editor and in the runtime engine. This is not a problem if the developed game makes use of all the state-of-the-art features, but it is a problem if you want to make a modest game where the gameplay is put to the test and not so much the technology.

The alternative to make smaller games is to make use of simpler engines, but usually they don't come with an editor and that lengthens the development process. This is where our end of degree work comes in. We are going to make an engine with its editor to make small games, aiming to provide a development experience equivalent to that of larger game engine editors. However, our focus will be on creating much smaller 2D games, catering to individuals with limited experience in programming or game development in general. This simplifies the use of the editor and also reduces the size of the executable versions produced.

The features of the engine we want to make are:

- *Self-sufficiency*: It provides functionality to manage resources, create scenes and objects from the editor and allows the creation of final executables of the game for its distribution. With this feature, the minimum dependence on external tools is sought.

- *Visual programming*: This is the most important part of our engine. Due to the complexity that some engines today pose for beginner or inexperienced developers, visual programming is a very useful and intuitive tool for creating logic and behavior in the video game.

- *Execute the game from the editor*: This provides the possibility to launch the game from the editor without having to create an executable version or manually search for the executable file. Additionally, you will be able to print information to the console that is visible from the editor, either for debugging or error checking purposes.

B.2. Goals

The main objective is to develop a self-sufficient 2D videogame engine with integrated editor and node-based visual programming. In addition, it will be possible to populate and view the progress of the game being developed directly from the editor and generate executable versions.

With this, users will have a tool to develop any type of 2D videogame with an accessible level of complexity and a pleasant and intuitive user experience.

B.3. Tools

To begin with, Git has been used as a version control system through the GitHub Desktop application. All the code implemented has been uploaded to a repository.

Link to the repository: `https://github.com/ivasan07/ShyEngine`

The code has been developed in the Visual Studio 2022 integrated development environment (IDE) and written in C++, and the PDF generation has been carried out with LaTeX.

Finally, we have carried out the task management through the Trello project management system.

B.4. Work Plan

Our project will be divided into three main blocks: engine, editor and visual scripting.

The work will be divided into five phases: research and planning, initial development, core development, development closure and user testing.

- *Research and planning*: The first phase of the work will consist of researching different game engines and publishers to understand how they work and their different architectures. We will also look for libraries that fit the demands of our project to achieve a comfortable and efficient development. Finally we will plan the division of work as well as the future continuous integration of each of the parts.

- *Initial development*: For this phase, we will be developing the core of each project. Regarding the engine, initial testing of the libraries to be used will be conducted, and the basic game architecture will be implemented. As for the editor, a project will be created in which the selected graphical interface library will be integrated to test its functionality. Regarding visual scripting, the language will begin to be prototyped. Additionally, a continuous integration process will be carried out as this phase progresses to prevent potential incompatibilities in the next phase. This will involve defining a file exchange format that will serve as the bridge between the editor and the engine. The type of information, its basic structure, and how assets will be referenced.

- *Development core*: During this phase, being the longest one, we will add new functionalities in each of the parts, such as new components in case of the editor, system of windows and docking for the editor, etc. We will extend the information exchange between engine and editor so that we will continue to integrate and we will continue to integrate and create the visual scripting system in the editor to generate the first scripts and the logic to store and interpret them in the engine.
- *Closing of the development*: With the projects completed and fully integrated, we will continue to develop features related to improving the user experience, especially in the editor, to make it as polished as possible before user testing. This will involve testing/developing a game to identify possible errors and correct them.
- *Testing with users*: User testing will be conducted with individuals from different backgrounds: users with programming experience and users without experience. This way, we will leverage their feedback to address potential issues and refine details that enhance the user experience.

Apéndice C

Conclusions

In the development of our game engine, despite being a relatively simpler project compared to industry-leading engines and having a team of just three members, we have managed to create a tool with a fairly accurate finish and functionality comparable to that of other renowned engines in the industry. Furthermore, we have successfully achieved our goal of making game development more accessible to individuals with no prior experience in the field, thus expanding the possibilities for game development to a wider audience.

However, our engine does have certain limitations, such as the inability to move ImGUI windows outside of the main SDL window, which can be inconvenient in certain situations, especially when implementing scripts. Additionally, we had to implement reflection in C++, which in hindsight could have been simplified by choosing a language with built-in reflection capabilities. A higher-level language would have also reduced the need to manually manage memory in the editor project, making its development more comfortable.

Looking ahead, we have several areas of improvement and potential development for our engine:

- *File path abstraction*: We can work on abstracting file and directory paths by turning files into engine objects. This would simplify resource management and provide more user convenience.
- *Add explanatory texts*: Including text-explanatory boxes in areas that might be confusing to users is a great idea. This would improve the user experience and make it easier for users to understand the tool.
- *Contribution of documentation and tutorials*: As a complement to adding explanatory texts, developing a section with documentation and links to tutorials to facilitate the learning curve is also a valuable en-

hancement. This would provide users with a comprehensive resource to learn and understand the tool more effectively.

- *Improve scripting*: Expanding the scripting capabilities would allow users to express their ideas more fully. This could include adding more complex functionality, such as managing editable arrays from the editor, creating classes, recursion, timers, coroutines, and debugging running nodes.
- *Custom data structures*: Provide users with the ability to create custom data structures and group elements within the scripting system, which would increase flexibility and design options.
- *Execute in the editor*: Allow users to run their games directly in the editor itself rather than in a separate window, which would streamline the development and testing process.
- *Enhanced usability of entities in the scene window*: Work on improving the usability of entities in the scene window so that users can interact and manage their elements more efficiently.

In summary, while we have achieved a functional and accessible game engine, we acknowledge that there is always room for improvement and expansion. Our goals for the future include making the tool even more user-friendly and adding additional features that enable more comprehensive and versatile game development.

Bibliografía

[1] Bourg, D. y Bywalec, B. *Physics for Game Developers: Leverage Physics in Games and More*. O'Reilly Media, Incorporated, 2013. ISBN 9781449392512.

[2] de Figueiredo, L., Celes, W. y Ierusalimschy, R. *Lua Programming Gems*. Roberto Ierusalimschy, 2008. ISBN 9788590379843.

[3] Gómez, C., García, A. y de las Heras del Dedo, R. *Métodos Ágiles. Scrum, Kanban, Lean*. MANUALES IMPRESCINDIBLES. Anaya Multimedia, 2017. ISBN 9788441538887.

[4] Gregory, J. *Game Engine Architecture*. Game Engine Architecture. CRC Press, Taylor & Francis Group, 2018. ISBN 9781138035454.

[5] Hecker, K. *Unity: Scripting with C#*. linkedin.com, 2015.

[6] Lee, J., Doran, J. y Misra, N. *Unreal Engine: Game Development from A to Z*. Packt Publishing, 2016. ISBN 9781787124790.

[7] Lengyel, E. *Game Engine Gems, Volume One*. Jones & Bartlett Learning, 2010. ISBN 9780763798963.

[8] Mitchell, S. *SDL Game Development*. Community experience distilled. Packt Publishing, 2013. ISBN 9781849696838.

[9] Parberry, I. *Introduction to Game Physics with Box2D*. CRC Press, 2017. ISBN 9781315360614.

[10] Ruelas, L. *Unity y C# Desarrollo de videojuegos*. RA-MA S.A. Editorial y Publicaciones, 2018. ISBN 9788499648163.

[11] Sherif, W. y Whittle, S. *Unreal Engine 4 Scripting with C++ Cookbook*. Packt Publishing, 2016. ISBN 9781785884689.

[12] Valcasara, N. *Unreal Engine Game Development Blueprints*. Packt Publishing, 2015. ISBN 9781784391782.