
Editor y motor de juegos 2D para no programadores



TRABAJO DE FIN DE GRADO

Pablo Fernández Álvarez

Yojhan García Peña

Iván Sánchez Míguez

Grado en Desarrollo de Videojuegos

Facultad de Informática

Universidad Complutense de Madrid

Septiembre 2023

Documento maquetado con TEXIS v.1.0+.

Editor y motor de juegos 2D para no programadores

Memoria que se presenta para el Trabajo de Fin de Grado

**Pablo Fernández Álvarez, Iván Sánchez Míguez, Yojhan
García Peña**

*Dirigida por el Doctor
Pedro Pablo Gómez Martín*

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

Septiembre 2023

Agradecimientos

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

Resumen

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

Un motor de ejecución 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 abstracta 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 ejecución que proporcione esa tecnología independiente a las necesidades concretas del videojuego a desarrollar.

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

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

Aquí es donde entra en juego nuestro trabajo. 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.

Índice

Agradecimientos	v
Resumen	vii
1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Herramientas	3
1.4. Plan de trabajo	3
2. Estado del arte	5
2.1. Qué es un videojuego	5
2.1.1. Cómo se hace un juego	6
2.1.2. Qué es un juego dirigido por datos	6
2.2. Qué es un motor de videojuegos	7
2.2.1. División entre motor y gameplay	8
2.2.2. De que está compuesto un motor de videojuegos	9
2.3. Qué es una arquitectura en el contexto del desarrollo de videojuegos	11
2.3.1. En qué consiste la arquitectura basada en Entidades y Componentes (EC)?	12
2.3.2. Qué es una arquitectura basada en herencia?	13
2.4. ¿Qué es un editor?	14
2.4.1. Evolución histórica de los Editores	14
2.4.2. Importancia en el desarrollo de videojuegos	14
2.4.3. Enfoque histórico y actual de la integración entre editor y motor	15
2.4.4. TODO: COMUNICACION MOTOR-EDITOR (NO ENTENDO EL COMENTARIO DE PP)	16
2.5. ¿Qué es el Scripting?	16
2.5.1. ¿Qué diferencia hay entre script y componente?	16

2.5.2. ¿Como se convierte el comportamiento en datos para el motor?	16
2.5.3. ¿Como se comunica el motor con el scripting?	19
2.6. Ejemplos de motores	20
2.6.1. Unity	20
2.6.2. Unreal	21
3. Vision general	25
4. Motor	27
4.1. Cómo funciona y cómo está dividió	27
4.1.1. Utilidades	27
4.1.2. Recursos	28
4.1.3. Sonido	28
4.1.4. Input	29
4.1.5. Consola	30
4.1.6. Físicas	30
4.1.7. Renderizado	31
4.1.8. Arquitectura de gameplay	32
4.1.9. Bucle principal	35
4.2. Cómo se genera la información necesaria para el editor	36
4.3. Cómo funciona el scripting por nodos	36
4.4. Cómo se traduce un script a lógica en C++	36
5. Editor	37
5.1. Funcionamiento y arquitectura	37
5.2. Como se leen los componentes a partir de los datos del motor	39
5.3. Como se trasladan los datos del editor al motor	40
5.4. Scripting en el editor	41
5.4.1. Serialización y uso en el motor	41
5.4.2. Edición de scripts	41
6. Videojuegos de ejemplo	43
6.1. Space Invaders	43
6.2. Plataformas genérico	44
7. Pruebas con usuarios	45
8. Contribuciones	47
8.1. Pablo Fernández Álvarez	47
8.2. Yojhan García Peña	48
8.3. Iván Sánchez Míguez	50

9. Conclusiones	53
Bibliografía	57

Índice de figuras

2.1. Juegos desarrollados por Lucasfilm Games.	7
2.2. Ejemplos de juegos dirigidos por datos con juegos clásicos	8
2.3. División de juego en motor y gameplay.	9
2.4. Motor aporta funcionalidad al gameplay.	10
2.5. Partes fundamentales que componen un motor de videojuegos.	11
2.6. Diferencia entre arquitectura centrada en objetos vs propiedades.	13
2.7. Editor de WorldCraft.	16
2.8. Funcionamiento de una DLL.	18
2.9. Editor de Unity.	21
2.10. Editor de Unreal Engine.	22
5.1. Captura del editor de ShyEngine.	37
5.2. Flujo de funcionamiento del editor.	39
5.3. Arquitectura de las ventanas.	39
5.4. Lectura de componentes del motor en el editor.	40
5.5. Serialización de la información del juego.	40
6.1. Space Invaders desarrollado con ShyEngine.	43
6.2. Juego de plataformas genérico desarrollado con ShyEngine.	44

Índice de Tablas

Capítulo 1

Introducción

1.1. Motivación

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

En el primer caso, el 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 el desarrollo 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 desarrollar esas tecnologías los equipos deben trazar la línea que separa lo que son las tecnologías para desarrollar un videojuego y el propio desarrollo del videojuego. Ésta separación es necesaria para garantizar la reutilización de las tecnologías.

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

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

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

racterísticas. Los más cómodos para el desarrollo son los que tienen editor, pero un editor es muy costoso de desarrollar, por lo que los motores que lo proporcionan son aquellos que tienen una masa crítica de uso muy grande, como para que merezca la pena el esfuerzo de desarrollo del editor. Esto solo ocurre en motores generalistas que permiten realizar juegos de muchos tipos y con muy buena calidad. El precio a pagar por usar esos motores es 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 trabajo de fin de grado. Vamos a hacer un motor con su editor para hacer juegos pequeños en los que la experiencia de desarrollo sea equivalente a la vivida con editores de motores más grandes, pero que esté centrado en el desarrollo de juegos 2D mucho más pequeños para perfiles con poca experiencia en programación o en desarrollo 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:

- *Ejecución del juego desde el editor*: Esto significa la posibilidad de lanzar el juego desde el editor sin tener que hacer una versión ejecutable o buscar el archivo ejecutable a mano. Además, se podrá imprimir información por consola visible desde el editor ya sea para depurar o consultar errores.
- *Autosuficiencia*: Aporta funcionalidad para manejar recursos, crear escenas y objetos desde el editor y permite la creación de ejecutables finales del juego para su distribución. Con esta característica se busca la mínima dependencia de herramientas externas.
- *Programación visual*: Esta es la parte a destacar de nuestro motor. Debido a la complejidad que presentan algunos motores de la actualidad para desarrolladores principiantes o inexpertos, la programación visual es una herramienta muy útil e intuitiva para crear lógica y comportamiento en el videojuego.

1.2. Objetivos

El objetivo principal es desarrollar un motor de videojuegos 2D autosuficiente con editor integrado y una programación visual basada en nodos. Además, se podrá poblar 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++.

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 juego

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

El desarrollo ha pasado por muchas fases hasta el estado en el que se 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 la década de 2000 en adelante, la mayoría de los juegos se desarrollan digitalmente, 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.

2.1.2. Qué es un juego dirigido por datos

Con el concepto de videojuego como división de programación y recursos en mente, 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.

Algunos ejemplos de juegos que siguen esta filosofía son las aventuras gráficas desarrolladas por Lucasfilm Games:

Como ejemplos de juegos clásicos dirigidos por datos serían el Quake II y el Half-Life, desarrollados por idSoftware y Valve Corporation, respectivamente.

En todos ellos, se creaban nuevos datos/recursos para desarrollarlos.



(a) Maniac Mansion (1987)



(b) Indiana Jones and the Last Crusade: The Graphic Adventure (1989)



(c) Loom (1990)



(d) Sam and Max Hit the Road (1993)

Figura 2.1: Juegos desarrollados por Lucasfilm Games.

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 un motor para renderizar gráficos 2D y 3D, un motor físico 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”.



Figura 2.2: Ejemplos de juegos dirigidos por datos con juegos clásicos

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

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), y el flujo general de la experiencia de juego en su conjunto.

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 esta imagen 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 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

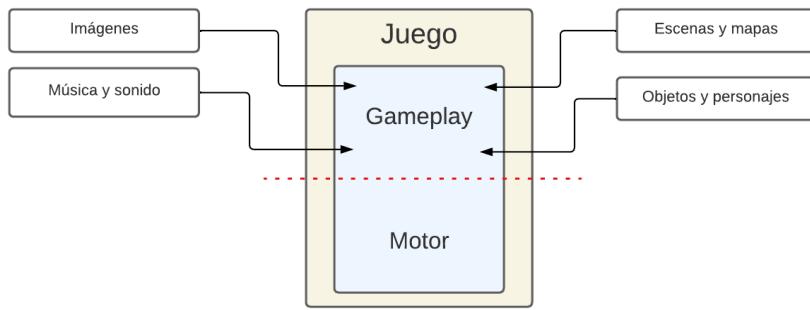


Figura 2.3: División de juego en motor y gameplay.

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 de Gregory (2018) 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. De que está compuesto 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 varias tecnologías con diferentes propósitos, todo necesarios para el videojuego. Para un motor de carácter general, es decir, sin implementación extra para un tipo de videojuegos concreto, estas tecnologías son

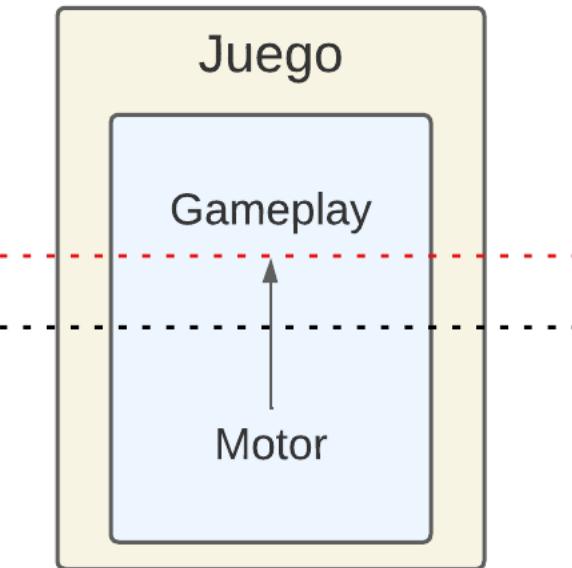


Figura 2.4: Motor aporta funcionalidad al gameplay.

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, mouse, 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

2.3. Qué es una arquitectura en el contexto del desarrollo de videojuegos 11

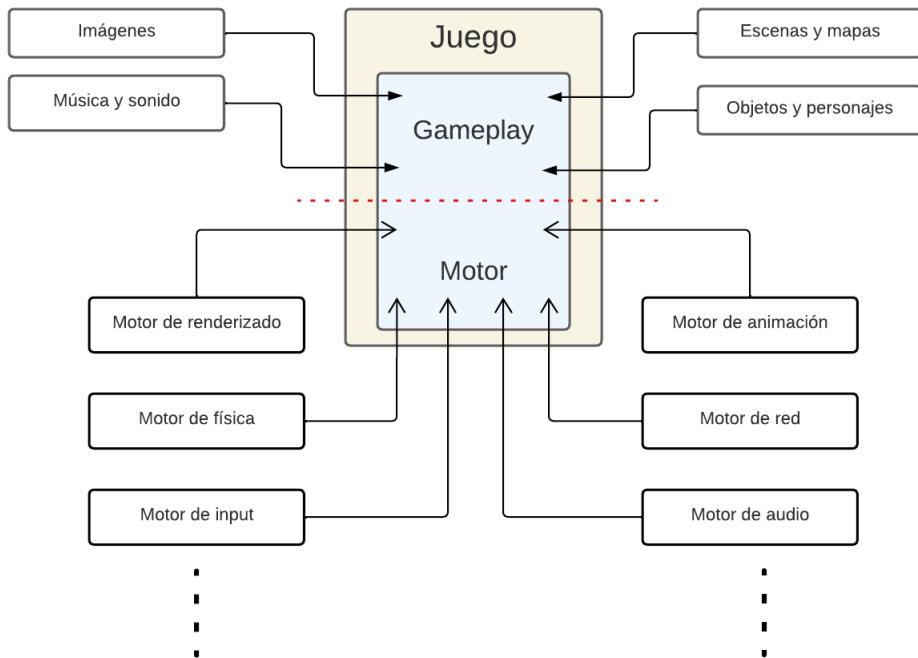


Figura 2.5: Partes fundamentales que componen un motor de videojuegos.

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.

El diagrama anterior quedaría completo de la siguiente manera:

2.3. Qué es una arquitectura en el contexto del desarrollo de videojuegos

Hasta ahora se 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 comportamiento del gameplay y sirven de conexión con el motor.

Un arquitectura define como se representan los diferentes tipos de entidades en el mundo junto con sus atributos y comportamientos.

Existen dos tipos de arquitecturas principales:

- *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.

Aquí una imagen que muestra la diferencia:

2.3.1. En qué consiste la arquitectura basada en Entidades y Componentes (EC)?

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*: Una entidad representa un objeto individual en el juego. Esto podría ser cualquier cosa, desde un personaje, un enemigo, un proyectil, hasta un objeto inanimado como una piedra pero realmente 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

2.3. Qué es una arquitectura en el contexto del desarrollo de videojuegos 13

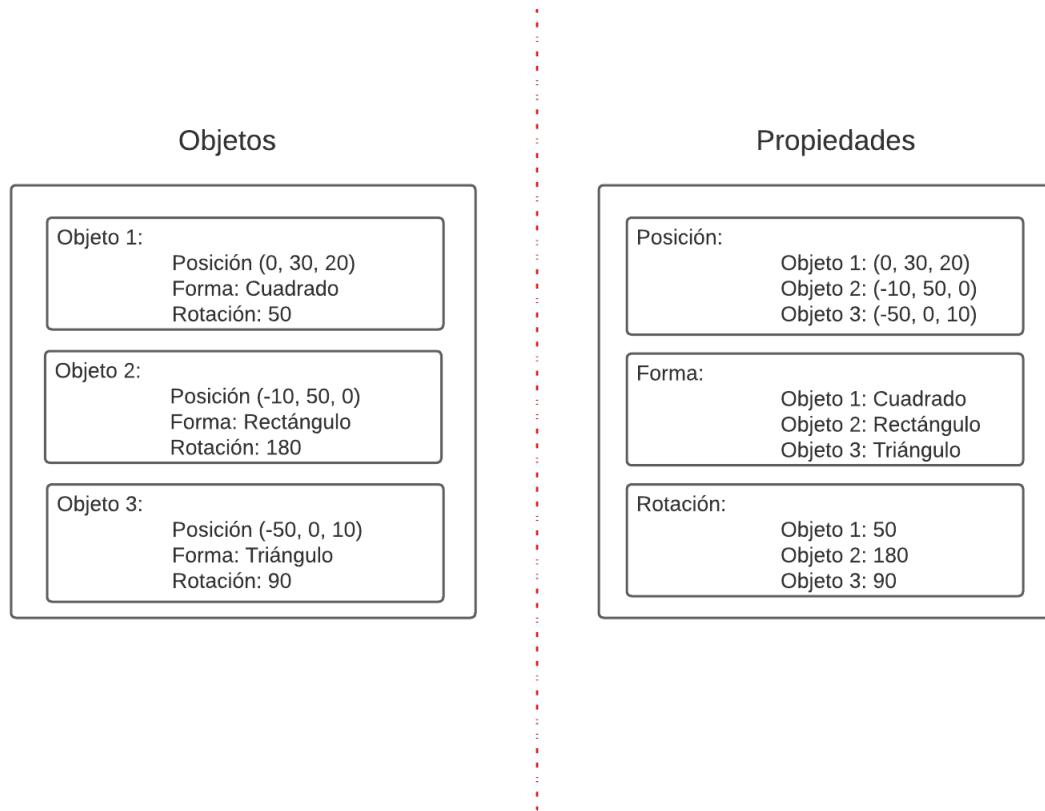


Figura 2.6: Diferencia entre arquitectura centrada en objetos vs propiedades.

componentes. Esto facilita la creación de nuevos tipos de objetos en el juego sin necesidad de escribir código específico para cada uno.

2.3.2. Qué es una 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 más 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. Además, en situaciones donde se requiere un alto rendimiento, la herencia puede ser más eficiente, ya que las clases son estáticas y el compilador puede realizar optimizaciones específicas.

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.

2.4. ¿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 facilidad de uso.

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.

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

En la actualidad, muchos editores de videojuegos están integrados con motores de juego, 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. Este proceso a menudo era laborioso y propenso a errores.

Un ejemplo histórico de esta separación entre motores y editores es el caso de WorldCraft, 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. En la imagen 2.7 observamos el editor WorldCraft.

Sin embargo, con la integración de editores y motores 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 aplican sin necesidad de complicados procesos de compilación.

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. La simplicidad de este enfoque también ha abierto las puertas a diseñadores y creadores

de contenido que pueden contribuir al desarrollo de juegos sin necesidad de conocimientos profundos en programación o procesos de compilación.

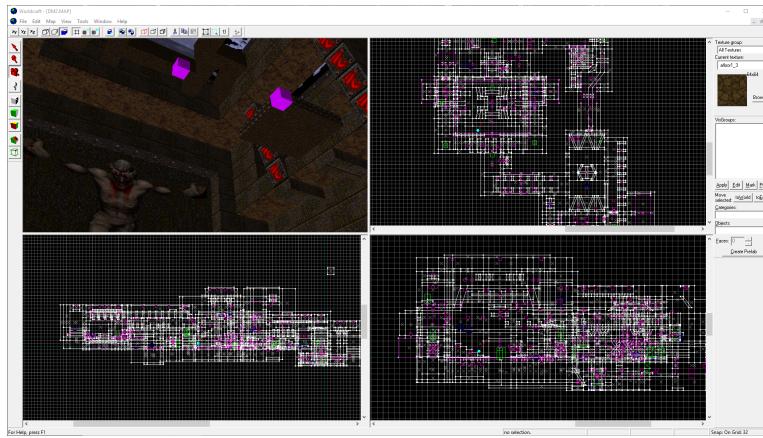


Figura 2.7: Editor de WorldCraft.

2.4.4. TODO: COMUNICACION MOTOR-EDITOR (NO ENTIENDO EL COMENTARIO DE PP)

2.5. ¿Qué es el Scripting?

El scripting, en el contexto del desarrollo de videojuegos, se refiere a la capacidad de definir el comportamiento y las reglas del juego utilizando scripts o instrucciones escritas en un lenguaje específico. Estos scripts permiten controlar eventos, acciones y la interacción dentro del juego, proporcionando una forma versátil de diseñar la experiencia del jugador.

2.5.1. ¿Qué diferencia hay entre script y componente?

La diferencia entre scripts y componentes es que, ambos definen lógica para el videojuego pero los componentes son comportamiento, en la mayoría de casos, fundamental y genérico (por ejemplo un componente “Imagen”), que proporciona el motor al usuario y los scripts son piezas de lógica que construye el usuario y que, en general, es comportamiento específico al videojuego que esté desarrollando el usuario.

2.5.2. ¿Como se convierte 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 del diseño del juego en sí.

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 (DLL) 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 (DLL) que se integran con el motor del juego.

La clave del funcionamiento de las DLL es el enlace dinámico. En lugar de incluir todo el código de la DLL en el ejecutable del juego, el programa principal solo contiene referencias a las funciones y datos en la DLL. Estas referencias se resuelven en tiempo de ejecución cuando el juego se carga. Esto significa que las DLL se cargan en memoria solo cuando se necesitan, lo que ahorra recursos y permite una gestión más eficiente de la memoria. Sin embargo, es necesario implementar un sistema de reflexión y serialización/deserialización en el motor del juego, ya que esto permite agregar nuevas entidades o comportamientos sin necesidad de recompilar o modificar el código del motor.

Este enfoque de DLLs 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.

En la figura 2.8 se observa de manera esquemática el funcionamiento de una DLL.

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.

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.

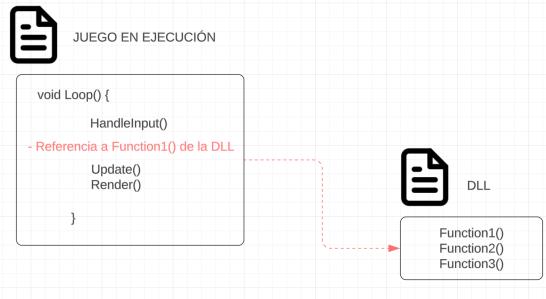


Figura 2.8: Funcionamiento de una DLL.

Por supuesto, también es importante considerar las limitaciones y desafíos que los lenguajes de script pueden presentar. Aunque son poderosos en muchos aspectos, pueden resultar complicados para los diseñadores de juegos y, en algunos casos, pueden volverse muy verbosos dependiendo de los objetivos específicos. Una posible solución 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.

2.5.3. ¿Cómo se comunica el motor con el 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.

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.

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

Unity utiliza varios lenguajes de programación. Para componentes de alto rendimiento, como el motor de física, utiliza C++. Para el desarrollo de scripts y el propio editor, utiliza C#.

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 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.

En la imagen 2.9 podemos ver el editor de Unity.

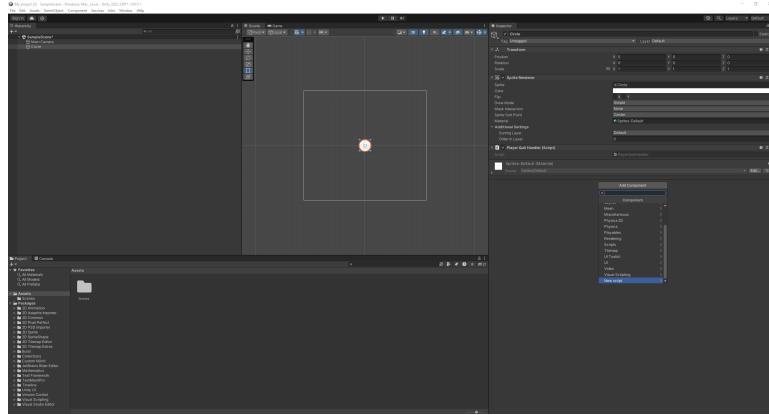


Figura 2.9: Editor de Unity.

2.6.1.3. Comunicación motor-editor

TODO

TODO: HABLAR DE HOT-RELOAD?

2.6.2. Unreal

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, UnrealEngine 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 videos y películas.

En cuanto al editor, tiene el siguiente aspecto:

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, puedes 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 orga-

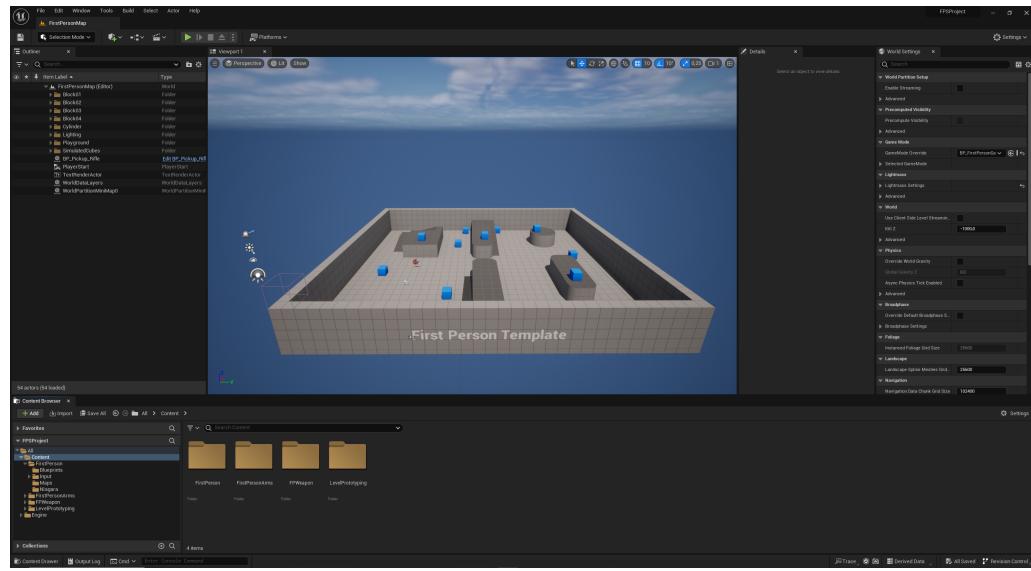


Figura 2.10: Editor de Unreal Engine.

nización de la funcionalidad común.

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 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 puedes 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 scripting, Unreal Engine tiene dos modos distintos pero compatibles:

- *Programación en C++*: esta forma de scripting 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 ren-

dimiento 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.

- *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 node-based (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.

En cuanto a la combinación de ambos enfoques, Unreal Engine permite integrar fácilmente código C++ y Blueprints en un mismo proyecto. Esto significa que puedes utilizar C++ para las partes críticas del rendimiento o para implementar sistemas complejos, mientras que utilizas 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.

2.6.2.3. Comunicación motor-editor

En este apartado debemos diferenciar de los elementos, escenas, personajes y todo lo que se pueda crear en el editor de Unreal Engine con los Blueprints.

En el primer caso, la información del editor se almacena en archivos de nivel (.umap) que contienen información sobre la ubicación de los elementos, las configuraciones de iluminación, las cámaras y más. Posteriormente, en tiempo de carga, qué sucede antes de la ejecución, Unreal Engine lee el archivo de nivel correspondiente. Este proceso implica cargar recursos desde disco, configurar la jerarquía del nivel y preparar la representación en memoria de los objetos y elementos del nivel. Por último

En el segundo caso, los Blueprints en Unreal Engine generan código C++. Cuando se ejecuta el juego o la simulación en Unreal Engine, el motor utiliza este código C++ generado a partir de los Blueprints para ejecutar la lógica de juego. Esto significa que el motor puede interpretar y ejecutar los Blueprints de la misma manera que ejecutaría código C++ escrito manualmente.

Capítulo 3

Vision general

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 de desarrollo de juegos. Sin embargo, esta versatilidad 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 datos que el motor puede leer e interpretar, permitiendo así desarrollar un motor especializado en juegos 2D simples y eficientes sin comprometerse con características innecesarias.

En nuestro modelo, la entidad fundamental es la escena, que contiene entidades individuales. Estas entidades están equipadas con componentes que, a su vez, almacenan atributos que representan datos específicos. Estos atributos pueden variar desde tipos de datos simples hasta referencias a otras entidades o recursos como imágenes. La serialización de estos datos se lleva a cabo en formato JSON, lo que permite una representación eficiente y legible de los objetos del juego.

Una decisión crítica en nuestro proyecto fue mantener el motor y el editor como entidades separadas en lugar de integrarlos. Esto nos otorga la flexibilidad de utilizar el motor y el editor con otros sistemas si fuera necesario, con la condición de establecer una forma coherente de transferir información entre ellos. Esta independencia de un motor de juego específico nos brinda la

libertad de elegir el motor más adecuado para las necesidades particulares de nuestro proyecto, un aspecto esencial cuando se trata de proyectos altamente personalizados.

Otro aspecto a tener en cuenta tanto en editor como en motor es el soporte de 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 simplificar el proceso de diseño y desarrollo. Además, modificar un prefab modifica también todas sus instancias en la escena.

TODO:————— Se decide tirar por un lenguaje visual porque el objetivo, como se dijo en el capítulo 1, era que pudieran usarlo no programadores. Vale, esta decisión ¿que nos supone a vista de pájaro, que es donde estamos ahora? Hay que ver qué queremos soportar porque hay que serializarlo/deserializarlo, crear el editor, y crear la "máquina virtual". Además hay que ver cómo se va a hacer "el puente" para que desde un dibujo (programa visual) se pueda llamar a un método de C++ del motor y al revés, sea cual sea. Y ¿vamos a usar eventos o update? Habrá que decidir qué eventos vamos a tener, cómo vamos a "dispararlos" desde el motor al gameplay y cómo vamos a "pintar" la reacción a un evento en el código".

Capítulo 4

Motor

4.1. Cómo funciona y cómo está dividio

El motor esta dividio en diez proyectos de tecnología y cada uno 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.

4.1.1. 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.

Entre estas clases destacan las siguientes:

- **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 y un porcentaje que representa la influencia que tendrá cada color en el color resultante.
- **EngineTime**: Por un lado, contiene información sobre el tiempo entre fotogramas del motor, tiempo entre pasos físicos, tiempo transcurrido desde el inicio del programa y número de fotogramas hasta el momento.

- **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.

4.1.2. 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.

El manager de recursos contiene un mapa por cada tipo de recurso donde la clave es la ruta del archivo y el valor un puntero a un objeto del tipo del recurso (**Texture***, **Font***, **Sound***, **Music***). El hecho de utilizar un mapa se debe a la complejidad constante de acceder a los recursos una vez creados.

Esto es importante porque 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.

Ya que la clave en los mapas es la ruta del archivo, los recursos pueden duplicarse en caso de tener el mismo archivo en diferentes directorios. El manager no contempla ese escenario ya que realmente el archivo también está duplicado y es responsabilidad del desarrollador ordenar sus archivos de assets.

4.1.3. Sonido

El objetivo de este proyecto es construir un envoltorio sobre la librería de audio **SDL_Mixer** para poder implementar posteriormente los componentes **MusicEmitter** y **SoundEmitter**.

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 número de canales, al contrario que con los efectos de sonido.

Este proyecto cuenta con tres clases:

- **SoundEffect**: Representa un efecto de sonido. Contiene la información de un MixChunk de **SDL_Mixer** y un identificador usado posteriormente por el componente **SoundEmitter**.

- **MusicEffect:** Representa un sonido de música de fondo. Contiene la información de un **MixMusic** de **SDL_Mixer** y un identificador usado posteriormente por el componente **MusicEmitter**.
Estas dos clases representan también los recursos que se usan para música y sonidos en el manager de recursos.
- **SoundManager:** Manager **Singleton** encargado de implementar el en voltorio de las funciones principales de **SDL_Mixer** para reproducir, parar, y detener sonidos, entre otros. Tiene dos métodos destinados al usuario para el modificar el volumen general y cambiar el número de canales disponibles para la reproducción de efectos de sonidos.

4.1.4. 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 de SDL y se actualizan debidamente.

Estos estados se dividen en:

1. *Down*: Una tecla esta siendo pulsada.
2. *Up*: Una tecla no esta siendo pulsada.
3. *Pressed*: Una tecla acaba de ser pulsada.
4. *Released*: Una tecla acaba de ser soltada.

- **Teclado:** Guarda la información sobre la mayoría de teclas importantes de un teclado. Letras, números y teclas especiales. Para ello, el manager cuenta con tres enumerados que contienen el nombre de cada una de las teclas para cada tipo.
- **Ratón:** Guarda la información de la posición del ratón, del estado del clic izquierdo, clic central (de la rueda), clic derecho y movimiento de la rueda.
- **Mando:** Cuenta con soporte para múltiples mandos y cada uno de ellos guarda la siguiente información:
 1. Nombre del mando.
 2. Estado de cada uno de los botones del mando.

3. Información del movimiento de los triggers del mando.
4. Información del movimiento de los joysticks del mando.

El manager tiene sporte además para conexiones y desconexiones durante la ejecución. 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 singleplayer, el usuario no tendrá que preocuparse por la posibilidad de múltiples mandos teniendo que indicar que identificador tiene su mando.

Por último, el manager implementa métodos de lógica para el usuario como movimiento horizontal y vertical, salto o acción.

4.1.5. 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 por consola con los colores por defecto, imprimir una advertencia, con color amarillo e imprimir un error, con color rojo, entre otros.

Además de ser útil para el desarrollo, sirve también para dar formato a los mensajes que aparecen por la consola del editor. Se utiliza una tubería o pipe para conectar la consola del motor y la del editor. Esto se cuenta más en detalle en el apartado de editor.

4.1.6. Físicas

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

Antes de nada, al igual que con **SDL_Mixer**, algunos conceptos sobre la librería:

- *Mundo físico*: La librería tiene una clase b2World que representa un mundo físico donde se pueden crear cuerpos físicos. Esta clase tiene un método fundamental **Step()**, al que se debe llamar para realizar un paso físico, lo que actualiza la simulación al avanzar el tiempo en un intervalo fijo, realiza cálculo de colisiones, resuelve restricciones y actualiza posiciones y velocidades.

- *Unidades*: Box2D trabaja con números de punto flotante y es necesario tener en cuenta alguna restricciones para que Box2D funcione correctamente. Estas restricciones han sido ajustadas para funcionar bien con unidades de metros-kilogramos-segundos (MKS). En particular, Box2D ha sido ajustado para funcionar adecuadamente con formas en movimiento que tienen dimensiones entre 0.1 y 10 metros.
- *Pixeles*: Es tentador usar pixeles como unidades para los tamaños, posiciones, fuerzas o velocidades pero desafortunadamente, esto llevaría a una simulación ineficiente y posiblemente a un comportamiento extraño. En la propia documentación de Box2D comentan que un objeto de 200 pixeles de longitud sería visto por Box2D como el tamaño de un edificio de 45 pisos.

Para resolver el problema de los pixeles, se usa un valor *screenToWorldFactor* usado para convertir de pixeles a unidades físicas y viceversa. Por lo tanto, a la hora de crear cuerpos físicos se convierte el tamaño en pixeles deseado por el usuario a unidades físicas utilizando ese factor de escala.

Las clases que tiene este proyecto son las siguientes:

- *PhysicsManager*: Clase, **Singleton**, que contiene la funcionalidad necesaria para manejar el filtrado de colisiones e información sobre gravedad del mundo físico así como la matriz de colisiones y capas existentes.

En cuanto al filtrado de colisiones, cada objeto físico tiene máscaras de bits donde guarda la información sobre en qué capa se encuentra y con qué capas colisiona. Para que se produzca una colisión, los cuerpos deben cumplir una condición, y es que, la capa del cuerpo A debe de estar marcada para que colisione con la del cuerpo B y viceversa.

El manager guarda un mapa para las capas donde la clave es el nombre de la capa y el valor un índice que la representa. Cuando se crea un nuevo cuerpo físico, se calculan sus máscaras de bits a partir de ese índice.

- *DebugDraw*: Clase que contiene la funcionalidad para dibujar los cuerpos físicos de Box2D. En concreto, puede dibujar polígonos, círculos, segmentos y puntos. El dibujado se realiza con SDL y antes de dibujar, se utiliza el *screenToWorldFactor* para devolver la escala a los cuerpos, es decir, de unidades físicas a pixeles.

4.1.7. Renderizado

Este proyecto hace uso de las librerías de SDL, **SDL_Image** y **SDL_TTF**.

Las clases que tiene este proyecto son las siguientes:

- *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, ícono, cursor, nombre y modo pantalla completa. Además, proporciona los métodos renderizar y para limpiar la pantalla.
- *Font*: Representa una fuente de texto y tiene la funcionalidad de crear una a partir de un 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*”.

Al igual que **SoundEffect** y **MusicEffect**, **Font** y **Texture** representan los recursos utilizado en el manager de recursos para almacenar fuentes de texto e imágenes.

4.1.8. Arquitectura de gameplay

Como arquitectura de gameplay se ha implementado de tipo entidades y componentes.

Este es el proyecto más importante del motor. Implementa la arquitectura, componentes fundamentales para el usuario y una serie de managers como el de escenas, prefabs, referencias y overlay.

Las partes fundamentales de esta arquitectura son las siguientes:

1. *Component*: Clase que representa a un componente. Contiene una referencia a la entidad a la que está asociado e información sobre si está activo o eliminado. 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.
2. *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, activa y eliminada, un identificador y su orden de renderizado.
3. *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. Por ello, las entidades únicamente se encarga de mantener actualizados a todos los componentes que tengan asociados. Además de los métodos para actualizar a los componentes, las entidades tiene métodos para añadir componentes, consultar si contienen un componente y eliminarlos.

4.1.8.1. 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:

1. *Transform*: Contiene la información sobre la posición, rotación y escala de la entidad. Además implementa algunos métodos para rotar, escalar y mover la entidad.
2. *Overlay*: Componente encargado de representar los elementos de la interfaz de usuario como textos, imágenes y botones.
3. *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.
4. *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 bastante propiedades físicas como el tipo de cuerpo (estático, cinemático, dinámico), el rozamiento o la escala de la gravedad. De esta clase heredan **BoxBody**, **CircleBody** y **EdgeBody**, que son cuerpos físicos cuyos colisionadores tienen formas especiales.
5. *SoundEmitter*: Componente encargado de cargar un sonido e implementar métodos para reproducirlo, detenerlo, pausarlo, etc. Como se comentó anteriormente, **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.
6. *MusicEmitter*: Componente encargado de cargar música e implementar métodos para reproducirla, detenerla, pausarla, rebobinarla, etc.

7. *ParticleSystem*: Componente encargado de implementar un sistema de partículas configurable. Tiene soporte para cargar texturas y mover las partículas con el motor de físicas Box2D.
8. *Animation*: Componente encargado de implementar la lógica de reproducción de animaciones.
9. *TopDownController*: Componente encargado de implementar un movimiento tipo Top-Down.
10. *PlatformController*: Componente encargado de implementar un movimiento de tipo plataformas.

Estos dos últimos componentes no son fundamentales pero aportan comodidad porque evitan al usuario tener que implementarlos usando el sistema de scripting, lo que puede ser algo avanzado. Mencionar también que es obligatorio que las entidades tengan al menos un componente, Transform u Overlay. Esto es así para poder distinguir entre entidades destinadas a la interfaz de usuario y el resto de entidades de la escena.

En cuanto a los managers:

1. *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:

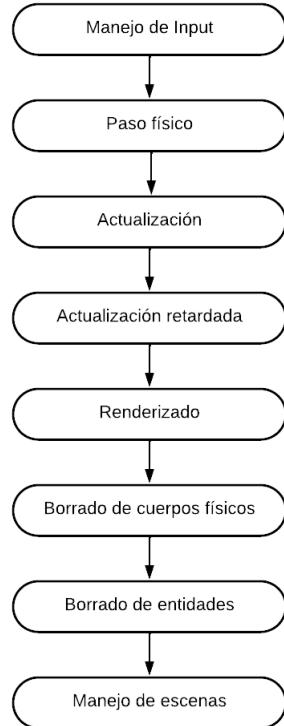
- *Operación PUSH*: carga la escena y la añade al top de la pila.
 - *Operación POP*: elimina la escena en el top de la pila y avisa, a la escena por debajo del top, si la hay, que va a empezar a actualizarse.
 - *Operación POPANDPUSH*: realiza una operación POP y posteriormente una operación PUSH.
 - *Operación CLEARANDPUSH*: vacía la pila de escenas y añade una nueva al top de la pila que va a empezar a actualizarse.
 - *Operación CLEAR*: vacía la pila de escenas.
2. *SceneLoader*: Clase encargado de leer la información de las escenas creadas en el editor. A la hora de construir las entidades diferencia entre entidades con **Transform** y entidades con **Overlay**.

3. *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`.
4. *RenderManager*: Encargado de renderizar por orden las entidades de la escena. A la hora de desarrollar en juego es deseable poder elegir elegir el orden en el que se renderizan las entidades. Esto también se conoce como profundidad o z-order.
5. *ReferencesManager*: Encargado de manejar una relación entre las entidades y sus identificadores. Necesario para evitar problemas de referencias entre entidades.

4.1.9. Bucle principal

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

En cuanto al bucle principal, tiene la siguiente estructura:



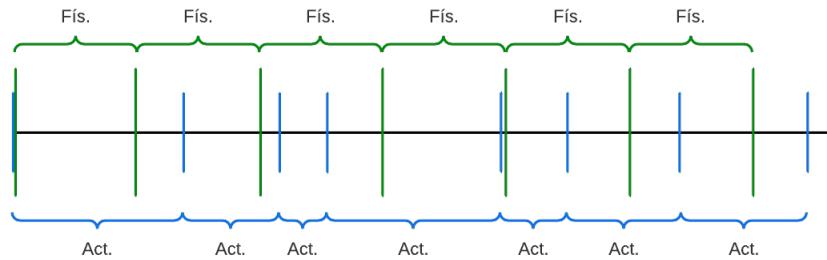
Además de esos métodos, que ejecutados en orden descendiente. se realizan cálculos de tiempo para proporcionar al usuario el `DeltaTime`, tiempo

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, y todas en general, requieren 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 qué 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.

Esto se explica mejor con el siguiente diagrama:



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 base a las necesidades del videojuego.

4.2. Cómo se genera la información necesaria para el editor

4.3. Cómo funciona el scripting por nodos

4.4. Cómo se traduce un script a lógica en C++

Capítulo 5

Editor

5.1. Funcionamiento y arquitectura

El editor está desarrollado en C++ a pesar de no tener el motor integrado debido a la elección de utilizar ImGui, una biblioteca de interfaz de usuario (UI) diseñada específicamente para C++. 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 imagen 5.1 podemos ver una imagen de nuestro editor.

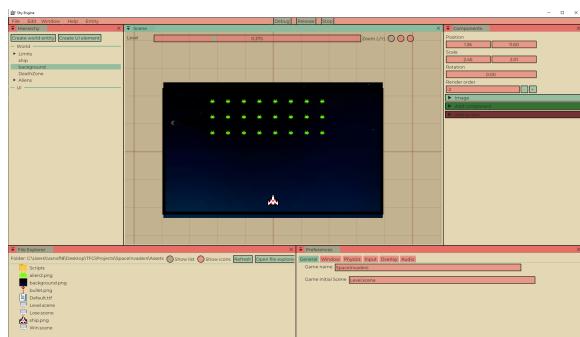


Figura 5.1: Captura del editor de ShyEngine.

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

1. *Gestor de proyectos*: Esta sección es la primera en iniciarse y se encarga de crear un archivo de configuración “.shyproject” y un directorio dedicado para cada videojuego que se desee crear, o para cargar un proyecto mediante la lectura de su archivo de configuración.
2. *Ventana de edición de scripts*: Desde esta ventana se maneja la creación

y edición de scripts. Permite a los desarrolladores definir el comportamiento y la lógica del juego.

3. *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.

El flujo de trabajo del editor comienza con la ventana del gestor de proyectos. Después se traslada a una pila de estados, que determina qué ventana se debe mostrar y manejar en un momento dado, ya sea la ventana de edición de scripts o el editor principal. Todas las operaciones en el editor se gestionan a través de un bucle principal que se encarga de manejar la entrada del usuario, actualizar el estado de las ventanas y renderizarlas.

En el dibujo 5.2 podemos ver dicho flujo.

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.

El editor cuenta con varias ventanas principales:

- *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 secundarias como la de preferencias o la de gestión de prefabs.

En el dibujo 5.3 observamos esta arquitectura de ventanas.

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.

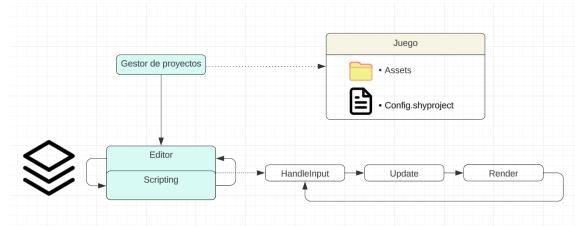


Figura 5.2: Flujo de funcionamiento del editor.

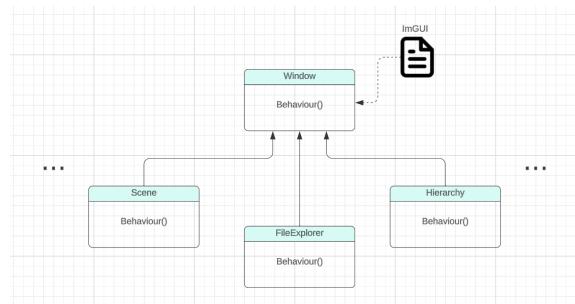


Figura 5.3: Arquitectura de las ventanas.

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

Como se mencionó anteriormente en (añadir referencia al apartado del motor que hable sobre esto), el motor del juego genera archivos en formato JSON que contienen los datos serializados de los componentes, sus atributos y su tipo correspondiente.

Los datos en formato JSON se interpretan y se utilizan para construir objetos de C++ en el editor. Estos objetos son los que se añaden a las entidades para definir su comportamiento.

En la imagen 5.4 se puede observar el funcionamiento de la lectura de componentes.

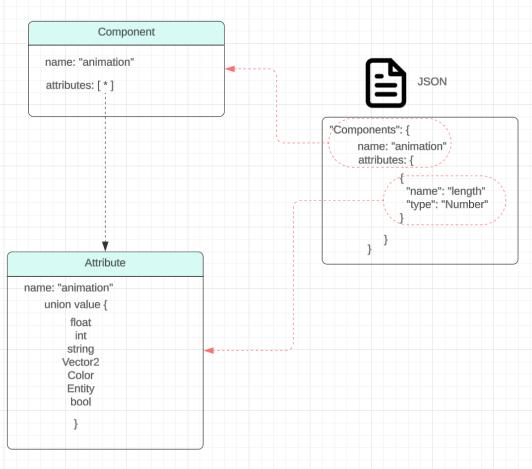


Figura 5.4: Lectura de componentes del motor en el editor.

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 (REFERENCIA).

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 5.5 representa como se serializa la información que será trasladada al motor.

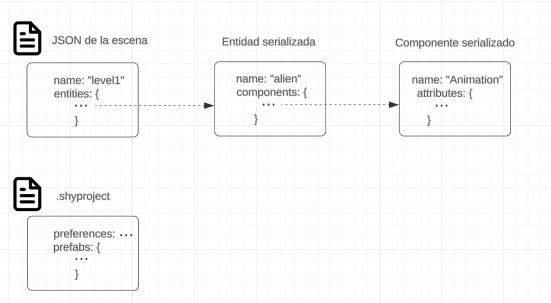


Figura 5.5: Serialización de la información del juego.

5.4. Scripting en el editor

Como hemos mencionado anteriormente, el editor contiene un estado completamente dedicado a la creación y modificación de scripts.

En el editor, los scripts se crean utilizando un sistema de programación visual [2.5.2] mediante nodos y conexiones. Los nodos representan acciones o eventos [2.5.3] 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.

Por ejemplo, un nodo de evento podría desencadenar una secuencia de nodos de lógica que determinan cómo responde el juego cuando un jugador alcanza cierta puntuación. Los nodos de valores se utilizan para almacenar y manipular datos dentro del script, lo que permite a los desarrolladores realizar cálculos y tomar decisiones basadas en variables específicas.

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, éstas contienen información sobre qué scripts tienen asignado.

Cuando un juego se ejecuta utilizando el motor, este archivo JSON se interpreta y se utiliza para guiar el comportamiento del juego. Los nodos y las conexiones se transforman en acciones y eventos en tiempo real, lo que permite que el juego responda a las interacciones del jugador y otros eventos del juego de acuerdo con el script creado en el editor.

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.

Capítulo 6

Videojuegos de ejemplo

Durante el proceso de desarrollo de nuestro motor de videojuegos y editor, hemos tenido la oportunidad de aplicar estas herramientas en la creación de varios videojuegos. 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. Space Invaders

Uno de nuestros primeros proyectos fue una versión simplificada del juego Space Invaders, este juego nos permitió poner a prueba cosas como los cambios de escenas, instanciacion de prefabs, manejo del input del jugador y las colisiones a través del scripting.

En la imagen 6.1 se puede observar una captura del juego.



Figura 6.1: Space Invaders desarrollado con ShyEngine.

6.2. Plataformas genérico

También hemos desarrollado un pequeño juego de plataformas que nos permitió usar animaciones, sistema de partículas y un movimiento de plataformaeo.

En la imagen 6.2 se puede observar una captura del juego.



Figura 6.2: Juego de plataformas genérico desarrollado con ShyEngine.

Capítulo 7

Pruebas con usuarios

Capítulo 8

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.

8.1. 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 implementados, comencé a implementar los primeros componentes básicos, como son el Transform, Image, PhyiscBody y SoundEmitter.

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

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

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

8.2. Yojhan García Peña

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

unión entre el editor y el motor.

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

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

Llegados a este punto pospuso un poco el desarrollo del scripting y empepéz a centrarme en cómo sería la unión entre el editor y el motor. Tras varios intentos fallidos finalmente terminé de desarrollar la clase ECSReader, con la cual se puede leer el contenido del motor gracias a unas marcas que se pueden ir añadiendo en el código. Y tras añadir al proyecto la biblioteca de lectura de JSON de nlohmann, toda la información relevante del motor pasaba a estar serializada en fichero en disco, siendo la idea que el motor pueda leer los valores desde ese fichero.

Aprovechando que el ECSReader ya estaba creado, fue usado para muchos más usos además de generar ficheros JSON. Finalmente terminamos utilizando el ECSReader para generar código en c++ que pueda leer el motor, de esta forma, automatizando bastante muchos aspectos tediosos. El ECSReader genera tres ficheros diferentes: ClassReflection, que permite serializar los atributos de una clase y poder dotarles de valor desde fuera conociendo el nombre de la variable; ComponentFactory, que con el nombre de un componente en formato string crea un componente de ese tipo, siendo una especie de factoría; y FunctionManager, el cual genera una función para cada método de los componentes y luego los agrupa en un mapa.

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

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

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

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

8.3. 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 imgui dock'de IMGUI, que permitía el anclaje de ventanas entre sí, lo que resultó ser una característica valiosa para el proyecto.

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

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

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

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

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

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

Capítulo 9

Conclusiones

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

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

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

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

Como trabajo futuro pensamos en la siguiente funcinalidad:

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

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

Ocornut Catto Community ifo Parberry (2017) Ericson (2004) Millington (2010) Fernando (2006) Eberly (2010) Eberly (2004) Romero et al. (2022) Bourg y Bywalec (2013) Shu (1988) Mitchell (2013) Wilde (2004) Gouveia (2013) Summers (2016) Madhav (2018)

Bibliografía

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

Miguel de Cervantes Saavedra

iforce2d. Disponible en <https://www.iforce2d.net/b2dtut/>.

BOURG, D. y BYWALEC, B. *Physics for Game Developers: Leverage Physics in Games and More.* O'Reilly Media, Incorporated, 2013. ISBN 9781449392512.

CATTO, E. Box2d. Disponible en <https://box2d.org/>.

COMMUNITY, S. Sdl2. Disponible en <https://wiki.libsdl.org/SDL2/FrontPage>.

EBERLY, D. *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic.* CRC Press, 2004. ISBN 9781482267310.

EBERLY, D. *Game Physics.* Taylor & Francis, 2010. ISBN 9780123749031.

ERICSON, C. *Real-Time Collision Detection.* Morgan Kaufmann Series in Inte. Taylor & Francis, 2004. ISBN 9781558607323.

FERNANDO, R. *GPU gems: programming techniques, tips, and tricks for real-time graphics.* Addison-Wesley, 2006.

GOUVEIA, D. *Getting Started with C++ Audio Programming for Game Development.* Community experience distilled. Packt Publishing, 2013. ISBN 9781849699105.

GREGORY, J. *Game Engine Architecture.* Game Engine Architecture. CRC Press, Taylor & Francis Group, 2018. ISBN 9781138035454.

MADHAV, S. *Game Programming in C++: Creating 3D Games.* Game Design. Pearson Education, 2018. ISBN 9780134597317.

- MILLINGTON, I. *Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game*. Taylor & Francis, 2010. ISBN 9780123819765.
- MITCHELL, S. *SDL Game Development*. Community experience distilled. Packt Publishing, 2013. ISBN 9781849696838.
- OCORNUT. Dear imgui. Disponible en <https://github.com/ocornut/imgui>.
- PARBERRY, I. *Introduction to Game Physics with Box2D*. CRC Press, 2017. ISBN 9781315360614.
- ROMERO, M., SEWELL, B. y CATALDI, L. *Blueprints Visual Scripting for Unreal Engine 5: Unleash the true power of Blueprints to create impressive games and applications in UE5*. Packt Publishing, 2022. ISBN 9781801818698.
- SHU, N. *Visual Programming*. Van Nostrand Reinhold, 1988. ISBN 9780442280147.
- SUMMERS, T. *Understanding Video Game Music*. Cambridge University Press, 2016. ISBN 9781107116870.
- WILDE, M. *Audio Programming for Interactive Games*. Taylor & Francis, 2004. ISBN 9781136125812.

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

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

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

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

