
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

Editor y motor de juegos 2D para no programadores

Memoria que se presenta para el Trabajo de Fin de Grado

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

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

Editor y motor de juegos 2D para no programadores

Un motor de videojuegos es un entorno de desarrollo que proporciona herramientas para la creación de videojuegos. Estas herramientas evitan al desarrollador implementar gran cantidad de funcionalidad para centrarse en mayor medida en el desarrollo del videojuego. Algunos ejemplos de funcionalidades que aportan los motores son: renderizado gráfico, motor de físicas, sistema de audio, gestión de input del jugador, gestión de recursos, gestión de red, etc.

Además, pueden llevar integrado un editor. Los editores son herramientas visuales cuyo objetivo es comunicar al motor las acciones que realiza el desarrollador. Por lo tanto, forman parte del entorno de desarrollo del motor. Los editores suelen tener una curva de aprendizaje lenta, especialmente para aquellos que no están familiarizados con el motor en particular o con el desarrollo de videojuegos en general. Sin embargo, una vez que los desarrolladores se familiarizan con las herramientas, pueden acelerar significativamente el proceso de creación del juego y mejorar la productividad.



Esto supone una gran ventaja a los desarrolladores experimentados pero motores como Unity o UnrealEngine pueden albergar demasiada complejidad para personas sin experiencia en programación, incluso aunque su objetivo sean juegos sencillos en 2D. Una herramienta muy útil para solucionar este problema es la programación visual. Este tipo de programación permite a los usuarios crear lógica mediante la manipulación de elementos gráficos en lugar de especificarlos exclusivamente de manera textual. Unity cuenta con su Unity Visual Scripting y UnrealEngine con los Blueprints.

El motor de este trabajo de fin de grado consiste en un entorno de desarrollo de videojuegos 2D autosuficiente. Esto quiere decir que permitirá gestionar los recursos del videojuego, las escenas y los elementos interactivos. Además, dará soporte para la creación de comportamientos a través de programación visual basada en nodos, la ejecución del juego en el editor y la creación de ejecutables finales del juego para su distribución.

Abstract

Game engine and editor 2D for non-programmers

A game engine is a development environment that provides tools for the creation of video games. These tools prevent the developer from having to implement a large amount of functionality in order to focus more on the development of the videogame. Some examples of functionalities provided by the engines are: graphic rendering, physics engine, audio system, player input management, resources management, network management, etc.

In addition, an editor may be integrated. Editors are visual tools whose purpose is to communicate to the engine the actions performed by the developer. They are therefore part of the engine's development environment. Editors tend to have a slow learning curve, especially for those who are not familiar with the engine. However, once developers become familiar with the tools, however, they can significantly speed up the process of game creation and improve productivity.

This is a great advantage to experienced developers, but engines like Unity or UnrealEngine can harbor too much complexity for non-programmers, even for people with no programming experience, even if they are aiming at simple 2D games. A very useful tool to solve this problem is visual programming. This type of programming allows users to create logic by manipulating graphical elements instead of specifying them exclusively textually. Unity has Unity Visual Scripting and UnrealEngine with Blueprints.

The engine of this graduate work consists of a self-sufficient 2D video game development environment. This means that it will allow manage the video game resources, scenes and interactive elements. In addition, it will support the creation of behaviors through node-based visual programming based on nodes, the execution of the game in the editor and the creation of final executables of the game for distribution.

Índice

Agradecimientos	V
Resumen	VII
Abstract	IX
1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Herramientas	2
1.4. Plan de trabajo	3
2. Introduction	5
2.1. Motivation	5
2.2. Goals	6
2.3. Project Management	6
2.4. Work Plan	7
3. Estado del arte	9
3.1. Motores de videojuegos 2D	9
3.2. Partes de un motor de videojuegos	9
3.3. Editores en motores de videojuegos	10
3.4. Diferencia entre editor y motor	11
3.5. Scripting vs programación	15
4. Editor	19
4.1. ImGUI	19
4.2. Gestión de escenas	20
4.3. Gestión de ventanas	22
4.4. La clase <code>Editor</code> y <code>WindowLayout</code>	24
4.5. Navegación y gestión de archivos con <code>FileExplorer</code>	24
4.6. Paso de assets entre escenas	25

4.7. Entidades	26
4.8. Jerarquía en las escenas	27
4.9. Prefabs y PrefabManager	29
4.10. Componentes	30
4.11. Ejecución del juego, estructura de carpetas y build del motor	31
4.12. Gestión de proyectos con ProjectManager	32
4.13. Gestión de recursos con ResourceManager	33
4.14. Configuración de preferencias y paletas de colores	34
4.15. Generación de build del juego	35
5. Motor	37
5.1. Utilidades	37
5.2. Recursos	38
5.3. Sonido	39
5.4. Input	40
5.5. Consola	42
5.6. Físicas	42
5.7. Renderer	44
5.8. Entity-Component-System	44
5.8.1. Managers	48
5.8.2. Componentes	49
5.9. Main	50
6. Scripting	53
6.1. Diseño del lenguaje	53
6.2. Implementación del scripting en el editor: edición de nodos	56
6.3. Implementación del scripting en el motor	62
7. Contribuciones	67
7.1. Pablo Fernández Álvarez	67
7.2. Yojhan García Peña	68
7.3. Iván Sánchez Míguez	70
8. Pruebas con usuarios	73
9. Conclusiones	75
10. Conclusions	77
I Apéndices	79

Capítulo 1

Introducción

1.1. Motivación



En lo relacionado a motores de videojuegos durante el grado, en primero aprendimos las bases de Unity, en segundo desarrollamos videojuegos 2D sencillos con SDL con una capa de abstracción para facilitar el aprendizaje y en tercero tuvimos que salir de la zona de **comfort** y aprender que hay detrás de las herramientas que nos proporcionaban durante el grado desarrollando así un motor de videojuegos 3D usando OGRE como motor gráfico, BulletPhysics como motor físico, FMOD como librería de sonido y Lua como lenguaje de scripting.

Con esta experiencia nos dimos cuenta de varias cosas. Es más cómodo trabajar con un motor que cuenta con un editor integrado en vez de acceder a él directamente a través de programación. Además, la forma de programar los scripts, que son fragmentos de código que definen comportamientos específicos para las entidades, debe ser cómoda e intuible ya que es la tarea que más tiempo va a ocupar al desarrollador.

Por último, en cuarto y con la reciente experiencia de desarrollo de un motor decidimos que la propuesta del trabajo de fin de grado sería un motor con las siguientes características:

- **Autosuficiencia:** Esto significa que aporta funcionalidad para manejar recursos, crear escenas y objetos desde el editor y **cuenta** con la creación de ejecutables finales del juego para su distribución. Además, el motor podrá mostrar la ejecución del juego directamente en el editor durante su desarrollo. Obviamente no tiene toda la funcionalidad necesaria como para no depender de ninguna herramienta externa durante el desarrollo pero sí aporta lo básico para el ciclo de desarrollo de un videojuego 2D.
- **Editor integrado:** La principal herramienta de un motor autosuficiente

es el editor, encargado de comunicar al motor las acciones del desarrollador. No teníamos experiencia en desarrollo de aplicaciones de escritorio, ya que para el motor de tercero no hicimos editor (por lo que sabíamos que podía ser un reto).

- *Programación visual basada en nodos*: 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. Sabíamos que podía ser desafiante a nivel técnico pero aportaría mucha comodidad, fluidez y por supuesto abriría las puertas de nuestro motor a muchos desarrolladores con poca experiencia en programación.

Por último, optamos por el 2D debido a la experiencia obtenida durante la carrera. Además, supone menor complejidad a nivel técnico durante el desarrollo del motor.

1.2. Objetivos

El objetivo principal del trabajo de fin de grado es desarrollar el motor de videojuegos 2D autosuficiente con editor integrado y programación visual basada en nodos. Con esto, los usuarios dispondrán de una herramienta para desarrollar cualquier tipo de videojuego 2D.

Con la autosuficiencia del motor queremos conseguir que los usuarios puedan desarrollar sus videojuegos con la mínima dependencia posible de herramientas externas de tal forma que todo el trabajo pase por el editor.

Con el editor queremos conseguir que el desarrollo sea cómodo, fluido y visual evitando así que el desarrollador se tenga que comunicar con el motor vía programación.

Con el sistema de programación visual basada en nodos queremos conseguir abrir las puertas de nuestro motor a desarrolladores inexpertos o con bajos conocimientos en programación para que así puedan desarrollar sus videojuegos. Además, supone un reto a nivel técnico que nos aportará experiencia y conocimiento.

1.3. Herramientas

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

En concreto, tal y como se cuenta en el plan de trabajo, el proyecto se ha dividido en motor, editor y programación visual.

Enlace al repositorio: <https://github.com/ivasan07/ShyEngine>

Para el desarrollo del motor se ha hecho uso de la librería de SDL (Simple DirectMedia Layer). En concreto, se ha utilizado SDL Image como sistema de gráficos, SDL Mixer como sistema de audio y SDL TTF como sistema de fuentes TrueType para renderizar texto. También se ha utilizado la librería Box2D para la simulación física y manejo de colisiones 2D.

Para el desarrollo del editor se ha hecho uso de la librería de interfaz gráfica ImGui.

El código ha sido desarrollado en el entorno de desarrollo integrado (IDE) Visual Studio 2022 y escrito en C++. Por último, para la organización de tareas hemos usado la herramienta de gestión de proyectos Trello.

1.4. Plan de trabajo

El trabajo se divide en tres partes: el desarrollo del motor, el desarrollo del editor y el desarrollo del sistema de scripting visual basado en nodos.



Etapas 1: Desarrollo Inicial

La primera fase del trabajo consistirá en el desarrollo del núcleo de cada una de las tres partes por separado:

- *Motor*: Para la parte del motor esta fase consistirá en la investigación de posibles librerías a utilizar, integración y funcionamiento básico en el entorno de trabajo de las librerías elegidas, implementación del sistema de entidades y componentes y desarrollo de componentes fundamentales.
- *Editor*: Para la parte del editor esta fase consistirá en investigar librerías gráficas, con especial énfasis en aquellas adecuadas para la interfaz de usuario (UI). Además, incluirá la implementación de una escena navegable de carácter básico, la creación y manipulación de entidades dentro de dicha escena, así como la modificación de su Transform, seguida de la correspondiente serialización de estos elementos.
- *Sistema de Scripting*: TODO

Etapas 2: Integración

La segunda fase del trabajo consistirá en la integración de las tres partes, posible reimplementación de funcionalidad no compatible y continuación con el desarrollo por separado:

- *Motor*: Para la parte del motor esta fase consistirá en el desarrollo del bucle principal, funcionalidad requerida para la comunicación motor-editor y terminar con el desarrollo de componentes y managers fundamentales.
- *Editor*: Para la parte del editor esta fase consistirá en la lectura de los componentes del motor, su visualización y edición, así como en la serialización y lectura de las escenas, abordando además otros aspectos fundamentales para un editor, tales como la implementación de la ventana de jerarquía y el explorador de archivos.
- *Sistema de Scripting*: TODO

Etapas 3: Mejoras y Pruebas

La tercera fase del trabajo consistirá en mejorar el editor para conseguir una experiencia de usuario satisfactoria, desarrollar algún que otro minijuego y realizar pruebas con usuarios:

- *Motor*: Para la parte del motor esta fase consistirá en realizar pruebas desde el editor y sistema de scripting para comprobar el correcto funcionamiento de todas las partes del motor y arreglar lo necesario.
- *Editor*: Para la parte del editor esta fase consistirá en la implementación de una jerarquía entre entidades, la implementación de prefabs, la lectura de escenas serializadas, la ejecución de las escenas en el motor, así como el arreglo de fallos y la incorporación de otras mejoras destinadas a aumentar la comodidad del desarrollador. Algún ejemplo de estas mejoras es la capacidad de arrastrar imágenes directamente desde el sistema operativo Windows hasta el editor, la depuración de errores a través de una consola, la creación de paletas, y muchas otras.
- *Sistema de Scripting*: TODO

TODO: Fase de prueba con usuarios...

Capítulo 2

Introduction



2.1. Motivation

Regarding game engines during our degree, in the first year, we learned the basics of Unity, in the second year, we developed simple 2D games with SDL using an abstraction layer to facilitate learning, and in the third year, we had to step out of our comfort zone and learn what's behind the tools provided to us during the degree. This led us to develop a 3D game engine using OGRE as the graphics engine, BulletPhysics as the physics engine, FMOD as the sound library, and Lua as the scripting language.

Through this experience, we realized several things. It's more convenient to work with an engine that has an integrated editor rather than accessing it directly through programming. Additionally, scripting should be comfortable and intuitive since it's the task that will take up the most time for the developer.

In the fourth year, with the recent experience of developing a game engine, we decided that the proposal for our final degree project would be an engine with the following characteristics:

- **Self-sufficient:** This means it provides functionality to manage resources, create scenes and objects from the editor, and can generate final game executables for distribution. Obviously, it doesn't have all the necessary functionality to be completely independent of any external tools during development, but it does provide the basics for the development cycle of a 2D game.
- **Integrated Editor:** The main tool of a self-sufficient engine is its editor, responsible for communicating the developer's actions to the engine. We had no experience in developing desktop applications, as we didn't create an editor for the third-year engine, so we knew it could be a challenge..

- **Visual Node-Based Programming:** This is the highlight of our engine. Due to the complexity that some modern engines present for beginner or inexperienced developers, visual programming is a very useful and intuitive tool for creating logic and behavior in the game. We knew it could be a technical challenge, but it would provide great convenience, fluidity, and, of course, open the doors of our engine to many developers with little programming experience.

Finally, we chose to focus on 2D due to the experience gained during our studies. Additionally, it involves less technical complexity during the development of the engine.

2.2. Goals

The main objective of the final degree project is to develop a self-sufficient 2D game engine with an integrated editor and node-based visual programming. With this, users will have a tool to develop any type of 2D video game.

By achieving self-sufficiency in the engine, we aim to enable users to develop their video games with minimal reliance on external tools, ensuring that all the work can be done within the editor.

With the editor, our goal is to make the development process comfortable, smooth, and visually-oriented, eliminating the need for developers to communicate with the engine through programming.

With the node-based visual programming system, we aim to open the doors of our engine to inexperienced developers or those with limited programming knowledge, allowing them to develop their videogames. In addition, it represents a technical challenge that will provide us with experience and knowledge.

2.3. Project Management

To start with, Git has been used as the version control system through the GitHub Desktop application. All the implemented code has been uploaded to a repository, dividing the work into branches. Specifically, as outlined in the work plan, the project has been divided into engine, editor, and visual programming.

Repository link: <https://github.com/ivasan07/ShyEngine>

For the development of the engine, the SDL (Simple DirectMedia Layer) library has been used. In particular, SDL Image has been used for graphics,

SDL Mixer for audio, and SDL TTF for TrueType font rendering. The Box2D library has also been used for 2D physics simulation and collision handling.

For the development of the editor, the ImGUI library has been used.

The code has been developed in the Visual Studio 2022 Integrated Development Environment (IDE) and written in C++. Finally, for task organization, we have used the Trello project management tool.

2.4. Work Plan

The work is divided into three main parts: engine development, editor development, and the development of a visual node-based scripting system.

Phase 1: Initial development

The first phase of the work will involve the development of the core of each of the three parts separately:

- *Engine*: For the engine part, this phase will involve researching potential libraries to use, integrating and ensuring basic functionality within the chosen library's working environment, implementing the entity and component system, and developing essential components.
- *Editor*: For the editor part, this phase will consist of researching graphic libraries, with special emphasis on those suitable for the user interface (UI). In addition, it will include the implementation of a basic navigable scene, the creation and manipulation of entities within this scene, as well as the modification of its **Transform**, followed by the corresponding serialization of these elements.
- *Scripting System*: TODO

Phase 2: Integration

The second phase of the work will focus on integrating the three parts, potentially reimplementing incompatible functionality, and continuing with separate development:

- *Engine*: For the engine part, this phase will involve developing the main loop, the required functionality for engine-editor communication, and completing the development of essential components and managers.
- *Editor*: For the editor part this phase will consist of reading the engine components, displaying and editing them, as well as serializing and reading the scenes, and other fundamental aspects for an editor, such as the implementation of the hierarchy window and the file explorer.

- *Scripting System*: TODO

Phase 3: Improvements and debugging

The third phase of the work will aim to improve the editor for a satisfactory user experience, develop some mini-games, and conduct user testing:

- *Engine*: For the engine part, this phase will involve conducting tests from the editor and scripting system to verify the correct functioning of all engine parts and making necessary fixes.
- *Editor*: For the editor part this phase will consist of the implementation of a hierarchy between entities, the implementation of prefabs, the reading of serialized scenes, the execution of the scenes in the engine, as well as bug fixing and the incorporation of other improvements aimed at increasing the developer's comfort. Some examples of these improvements are the ability to drag images directly from the Windows operating system to the editor, debugging errors through a console, palette creation, and many others, and many others.
- *Scripting System*: TODO

TODO: User testing phase...

Capítulo 3

Estado del arte

RESUMEN: En este capítulo se explica qué son los motores de videojuegos 2D, las partes que contienen y los principales motores de videojuegos a día de hoy. Por otro lado se hablará de qué es un editor, que relación tienen con los motores y su importancia en el desarrollo. Por último se explica en qué consiste el scripting visual, que tipos de scripting visual hay y en concreto qué características tiene el tipo basado en nodos.

3.1. Motores de videojuegos 2D

Un motor de videojuegos 2D es a un sistema de software diseñado para facilitar el desarrollo, diseño y ejecución de videojuegos que se desarrollan en un entorno bidimensional. Este componente tecnológico proporciona un conjunto de herramientas y funcionalidades predefinidas que permiten a los desarrolladores crear juegos visuales en dos dimensiones de manera eficiente y efectiva. Los motores de videojuegos 2D son esenciales para el proceso de producción de juegos, ya que simplifican tareas técnicas complejas y permiten a los creadores centrarse en la creatividad y la jugabilidad.



3.2. Partes de un motor de videojuegos

Un motor de videojuegos consta de varias partes esenciales que trabajan en conjunto para facilitar el desarrollo de videojuegos. Estas partes clave incluyen:

- *Motor gráfico:* Responsable de la representación visual de los elementos del juego, como objetos o interfaz, garantizando su adecuada superposición en el espacio bidimensional.

- *Motor físico*: Simula efectos físicos realistas, como gravedad, movimiento y colisiones entre objetos, para que los elementos del juego interactúen de manera coherente y realista, mejorando la experiencia del jugador.
- *Sistema de Input*: Captura y procesa la entrada del jugador, como pulsaciones de teclas y movimientos del ratón, permitiendo controlar la interacción del jugador con el juego.
- *Gestión de escenas*: Permite la creación y gestión de múltiples escenas o niveles dentro del juego, facilitando la transición entre partes del juego como menús, gameplay, inventario, mapa, etc.
- *Motor de audio*: Reproduce efectos de sonido y música para enriquecer la atmósfera del juego y proporcionar retroalimentación auditiva al jugador, lo que puede mejorar la inmersión y la retroalimentación.

Estas partes esenciales trabajan en conjunto para proporcionar un entorno de desarrollo completo y eficiente para juegos en 2D, permitiendo a los desarrolladores centrarse en la creatividad y la jugabilidad en lugar de crear todas estas funcionalidades desde cero.

3.3. Editores en motores de videojuegos

Un editor de videojuegos es una herramienta de software que facilita la creación, modificación y organización de diversos elementos de un videojuego. Este componente tecnológico proporciona una interfaz visual y funcionalidades específicas que permiten a los desarrolladores y diseñadores trabajar en la construcción y edición de contenido de juegos de manera eficiente. Los editores de videojuegos son esenciales en el proceso de producción, ya que permiten la creación y personalización de niveles, personajes, escenarios y otros componentes visuales.

- *Creación de Escenas y Niveles*: Los editores de videojuegos ofrecen herramientas para diseñar y crear los entornos de juego. Esto incluye la disposición de elementos, creación de geometría del nivel, etc.
- *Diseño de Personajes y Objetos*: Los editores permiten la creación y personalización de personajes jugables, personajes no jugables (PNJ), enemigos, objetos y elementos interactivos. Esto puede incluir la definición de apariencia, comportamiento y habilidades.
- *Gestión de Recursos Multimedia*: Los editores de videojuegos facilitan la importación y organización de gráficos, sonidos, música y otros activos multimedia que se utilizarán en el juego.

- *Asignación de Comportamientos y Lógica*: Los editores permiten definir el comportamiento de los elementos del juego mediante la asignación de reglas, scripts y lógica programada. Esto incluye la configuración de interacciones y eventos.
- *Personalización de Interfaz de Usuario*: Algunos editores permiten la adaptación de la interfaz de usuario del juego, lo que incluye la disposición de elementos de la pantalla, el diseño de menús y la presentación visual general.
- *Pruebas y Depuración*: Los editores de videojuegos a menudo incluyen herramientas de prueba y depuración que permiten a los desarrolladores evaluar y ajustar el comportamiento del juego durante el proceso de creación.
- *Exportación y Distribución*: Una función esencial de los editores es la capacidad de exportar el juego en un formato que pueda ser ejecutado por el motor de videojuegos correspondiente. Esto permite la distribución y el acceso al juego final.



3.4. Diferencia entre editor y motor

En la industria, es frecuente observar que ambos términos estén estrechamente vinculados ya que el editor y el motor de videojuegos desempeñan roles complementarios esenciales en el proceso de desarrollo de videojuegos al permitir un flujo de trabajo eficiente. Su sinergia constituye un pilar fundamental en el proceso de desarrollo de videojuegos, facilitando la creación y optimización de experiencias de juego únicas y atractivas.



El motor de videojuegos representa el núcleo tecnológico que sustenta el juego, encargándose de tareas críticas como la gestión de gráficos, física, sonido o lógica. Puede considerarse como el corazón del juego, sin el cual la mayoría de los juegos simplemente no podrían existir.

Por su parte, el editor de videojuegos se posiciona como una herramienta de creación y diseño que colabora estrechamente con el motor. Su principal función radica en permitir a los desarrolladores y diseñadores de juegos crear y modificar contenido mucho más rápida y cómodamente que si tuvieran que hacerlo a través del código directamente. Esto simplifica la personalización del mundo del juego y la configuración de sus componentes. En el editor se definen diversos datos, como escenas, componentes y entidades, que el motor posteriormente leerá para su procesamiento y ejecución.

Comunicación motor-editor

Durante el proceso de creación de un videojuegos a través de un motor de videojuegos con editor, todas las acciones de los desarrolladores se producen en el editor. Con todas estas acciones, el editor genera unos datos que serán leídos posteriormente por el motor para poder ejecutar el videojuego. Este proceso iterativo permite a los equipos de desarrollo perfeccionar gradualmente la experiencia de juego, realizando ajustes y corrigiendo errores a medida que avanzan en el proyecto.

La comunicación entre el motor y el editor se ilustra en la Figura 3.1.

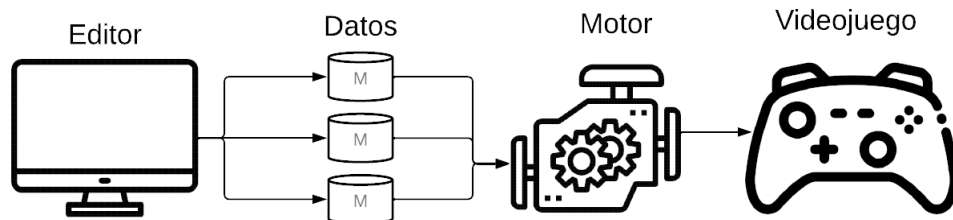


Figura 3.1: Comunicación motor-editor

Motor de física


Un motor de física en el desarrollo de videojuegos es un software o una biblioteca de programación que simula y calcula el comportamiento físico de objetos dentro del mundo virtual del juego. Estos motores permiten que los objetos en un videojuego interactúen con realismo según las leyes de la física, lo que añade un nivel de realismo y jugabilidad a la experiencia del jugador.

Algunas de las funciones que un motor de física puede proporcionar incluyen:

- *Simulación de colisiones*: Los motores de física pueden detectar y manejar colisiones entre objetos en el juego, calculando cómo se deben comportar los objetos cuando chocan entre sí, rebotan, se deslizan o se detienen.
- *Gravedad*: Los motores de física pueden aplicar la fuerza de la gravedad a los objetos en el juego, lo que significa que los objetos caen hacia abajo y pueden rebotar o interactuar de manera realista con el entorno.
- *Cinemática inversa*: Los motores de física a veces incluyen herramientas para calcular automáticamente animaciones realistas de personajes.

y objetos en función de las acciones del jugador. Esto es útil para animaciones de personajes en juegos, como caminar, correr o saltar.


- *Simulación de fluidos y partículas*: Algunos motores de física avanzados pueden simular efectos de fluidos, como agua o fuego, así como partículas, como humo o chispas.
- *Simulación de movimientos*: Los motores de física permiten aplicar fuerzas a los objetos para simular movimientos, como la aplicación de una fuerza para lanzar un objeto o empujarlo.
- *Restricciones y articulaciones*: Estos motores también pueden gestionar restricciones y articulaciones que definen cómo los objetos pueden moverse y girar entre sí, lo que es útil para simular conexiones como bisagras, ruedas o articulaciones de personajes.



Algunos ejemplos de motores de física populares en la industria de los videojuegos incluyen Unity Physics, NVIDIA PhysX, Havok y Bullet Physics, entre otros. Estos motores ofrecen diferentes características y niveles de realismo, lo que permite a los desarrolladores elegir el más adecuado para sus necesidades específicas.

Unity

Unity es un motor de desarrollo de videojuegos altamente influyente debido a su accesibilidad y versatilidad. Utiliza un motor gráfico propio que es conocido por permitir a desarrolladores de todos los niveles de experiencia crear juegos y aplicaciones interactivas para una amplia variedad de plataformas, desde PC y consolas hasta dispositivos móviles y realidad virtual. Su característica distintiva es su capacidad para facilitar el desarrollo multiplataforma, lo que lo convierte en una elección sólida para proyectos que buscan llegar a una audiencia diversa.



Como motor físico, Unity utiliza el motor de físicas NVIDIA PhysX, que es ampliamente reconocido por su capacidad para simular efectos físicos realistas en los juegos. La combinación del motor gráfico Unity Graphics y el motor de físicas NVIDIA PhysX proporciona a los desarrolladores las herramientas necesarias para crear experiencias visuales y físicas inmersivas en sus juegos.

Unreal Engine

Unreal Engine, por otro lado, se destaca por su impresionante potencia gráfica y capacidades en 3D. Utiliza su propio motor gráfico altamente avanzado que es la elección preferida para desarrolladores que buscan crear experiencias visuales de alta calidad, como juegos de acción, aventuras y

experiencias de realidad virtual inmersivas. Sus gráficos fotorrealistas, motor de física avanzada y sistema de partículas robusto lo convierten en una herramienta esencial para proyectos de vanguardia.



Además, Unreal Engine incorpora su propio motor de física llamado Chaos.

En las siguientes imágenes se muestran el editor del motor de videojuegos Unity 3.2 y el editor del motor de videojuegos Unreal Engine 3.3.

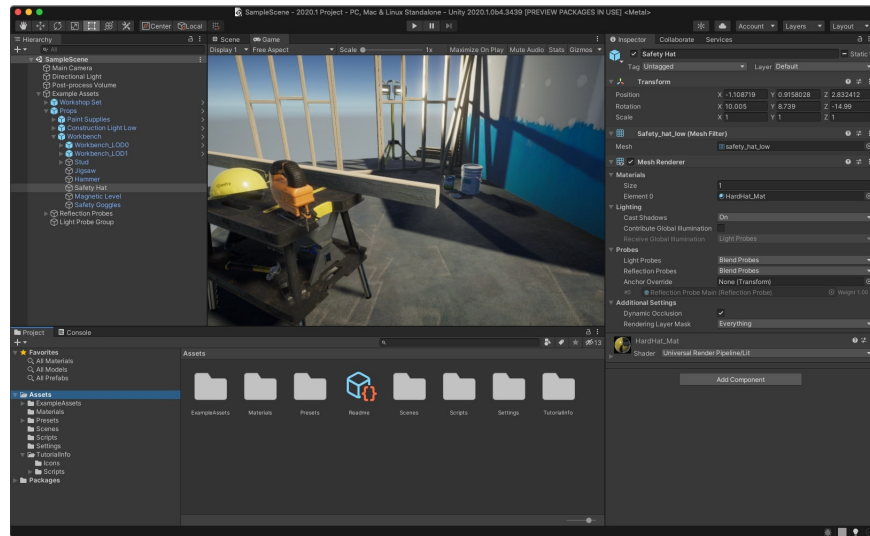


Figura 3.2: Editor del motor de videojuegos Unity

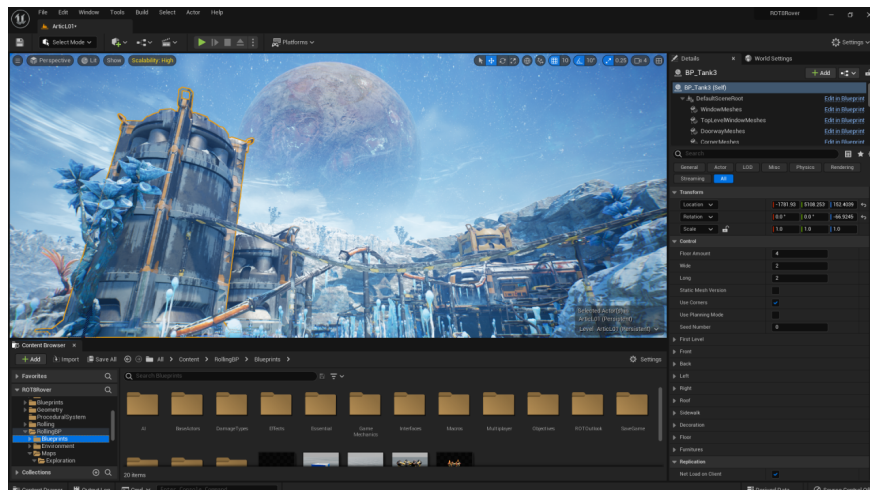


Figura 3.3: Editor del motor de videojuegos Unreal Engine



3.5. Scripting vs programación

El desarrollo de videojuegos puede llevarse a cabo utilizando diferentes enfoques, entre los que se destacan el scripting por nodos y la programación tradicional. Estos enfoques influyen en la manera en que se construye la lógica y la funcionalidad del juego. Además, existen sistemas específicos de scripting que simplifican la creación de videojuegos, como los Blueprints de Unreal Engine o Scratch. A continuación, se explorarán las diferencias entre ambos enfoques y se describirán los sistemas investigados.

Scripting por Nodos vs. Programación

- *Scripting por Nodos*: Este enfoque utiliza interfaces visuales y gráficas para representar la lógica del juego mediante nodos interconectados. Los desarrolladores ensamblan estos nodos para definir el flujo de control, las interacciones y las acciones del juego. No se requiere conocimiento profundo de programación, lo que lo hace más accesible para principiantes. Es común en herramientas como Unreal Engine y Scratch.
- *Programación*: La programación tradicional implica escribir código en lenguajes de programación como C++, CSharp, Python o JavaScript. Los desarrolladores crean instrucciones detalladas para definir el comportamiento del juego. Este enfoque es más poderoso y versátil, pero puede requerir un nivel más alto de experiencia en programación.

Ejemplo de scripting por nodos: Blueprints de Unreal Engine


Los Blueprints en Unreal Engine representan una implementación destacable del enfoque de Scripting por Nodos en el desarrollo de videojuegos. Esta herramienta esencial permite a los desarrolladores crear la lógica del juego utilizando una interfaz visual basada en nodos, lo que facilita enormemente la creación de juegos sin la necesidad de escribir código en lenguajes tradicionales como C++.

Funcionamiento de los Blueprints

- *Interfaz Visual Intuitiva*: Los Blueprints de Unreal Engine proporcionan a los desarrolladores una interfaz gráfica intuitiva y amigable. En lugar de escribir líneas de código, los diseñadores y desarrolladores pueden crear lógica utilizando una colección de nodos predefinidos, que representan acciones, eventos y operaciones lógicas.
- *Nodos Interconectados*: La lógica del juego se construye interconectando estos nodos de manera visual. Los nodos pueden representar even-


tos, como la pulsación de un botón o la colisión de objetos, así como acciones, como mover un personaje o cambiar la iluminación de una escena. Al conectar estos nodos de manera apropiada, se define el flujo de control y el comportamiento del juego.

- *Personalización y Reutilización:* Los Blueprints permiten una alta personalización y reutilización de la lógica. Los desarrolladores pueden crear sus propios nodos personalizados y guardarlos para utilizarlos en futuros proyectos, acelerando así el desarrollo y manteniendo un alto nivel de consistencia.
- *Facilita la Colaboración:* Dado que la interfaz de Blueprints es visual y gráfica, facilita la colaboración entre diseñadores y programadores. Los diseñadores pueden crear la lógica del juego de manera más independiente, lo que permite a los programadores centrarse en tareas más técnicas.
- *Combinación con C++:* Unreal Engine permite una integración fluida entre Blueprints y código C++. Esto significa que los desarrolladores pueden utilizar Blueprints para definir el comportamiento general del juego y luego recurrir a C++ para implementar funciones específicas o mejorar el rendimiento cuando sea necesario.
- *Depuración y Visualización:* Los Blueprints ofrecen herramientas de depuración visuales que permiten a los desarrolladores rastrear y solucionar problemas en la lógica del juego de manera efectiva. Esto incluye la capacidad de ver el flujo de ejecución de los nodos y detectar posibles errores.



En resumen, los Blueprints de Unreal Engine son una herramienta poderosa que democratiza el desarrollo de videojuegos al permitir que una amplia gama de profesionales, desde diseñadores hasta programadores, colaboren en la creación de experiencias interactivas sin la necesidad de una programación profunda. Esta versatilidad y accesibilidad los convierten en una elección popular en la industria del desarrollo de videojuegos.

Ejemplo de scripting por programación: MonoBehaviour de Unity



Unity, una de las plataformas líderes en desarrollo de videojuegos, se destaca por su enfoque en la programación tradicional a través de su componente llamado MonoBehaviour. A diferencia del enfoque de Scripting por Nodos, que hemos discutido previamente, MonoBehaviour permite a los desarrolladores utilizar lenguajes de programación como CSharp para definir la lógica y la funcionalidad del juego de manera más programática y detallada.

El uso de MonoBehaviour en Unity se basa en los siguientes conceptos:

- *Componentes en GameObjects*: En Unity, los objetos en el juego se representan como GameObjects, y cada GameObject puede tener uno o más componentes MonoBehaviour adjuntos. Estos componentes representan la funcionalidad y el comportamiento del GameObject.
- *Programación en CSharp*: Para definir la lógica y el comportamiento de un GameObject, los desarrolladores utilizan el lenguaje de programación CSharp en combinación con MonoBehaviour. Escriben scripts que heredan de la clase MonoBehaviour y anexan estos scripts como componentes a los GameObjects correspondientes.
- *Métodos y Eventos*: Los scripts de MonoBehaviour pueden implementar métodos que se ejecutan en momentos específicos durante la vida útil del GameObject, como `Start()` para la inicialización y `Update()` para la actualización continua. Además, pueden responder a eventos como colisiones, clics del mouse o entradas del teclado.
- *Flexibilidad y Control*: El uso de programación con MonoBehaviour brinda a los desarrolladores un alto grado de flexibilidad y control sobre la lógica del juego. Pueden definir comportamientos precisos y detallados para los elementos del juego.
- *Personalización y Extensibilidad*: Unity permite la creación de scripts personalizados que se pueden reutilizar en múltiples objetos o proyectos. Esto facilita la extensibilidad y la personalización de la funcionalidad del juego.
- *Combinación con Assets y Gráficos*: Los scripts de MonoBehaviour pueden interactuar con recursos gráficos, modelos 3D, sonidos y otros assets del juego, lo que permite una integración completa de la programación con los elementos visuales y auditivos del juego.

En resumen, mientras que el scripting por nodos simplifica la creación de videojuegos a través de interfaces visuales, MonoBehaviour en Unity representa un enfoque más programático que brinda a los desarrolladores un alto nivel de control y flexibilidad sobre la lógica y la funcionalidad del juego. Esto lo convierte en una opción preferida para proyectos que requieren una programación detallada y un mayor control sobre la interacción del juego.

Capítulo 4

Editor

4.1. ImGUI

ImGui es una biblioteca de interfaz de usuario (UI) que se destaca por su enfoque inmediato, generando la interfaz de usuario de manera efímera en cada fotograma en lugar de mantener objetos de UI a largo plazo. Esto simplifica el desarrollo y es ampliamente utilizado en la creación de ventanas, popups, botones, campos de texto y más. ImGui es de código abierto y está disponible en varios lenguajes de programación, lo que lo hace accesible para una amplia gama de desarrolladores.

Además, en nuestro proyecto, hemos aprovechado la funcionalidad de docking de ImGui, que permite conectar ventanas entre sí de manera flexible. Esto significa que los usuarios pueden organizar y anclar ventanas según sus preferencias, lo que facilita la personalización de la interfaz de usuario del editor.

Ventanas y Popups

ImGui facilita la creación de ventanas y popups de manera intuitiva y eficaz. Esto permite organizar y presentar información de manera clara y ordenada en la interfaz del editor. Los métodos de ImGui, como **Begin()**, **End()**, **OpenPopup()**, y **CloseCurrentPopup()**, son utilizados para gestionar ventanas y popups de manera dinámica.

Selección y Dropdowns

La biblioteca ImGui proporciona elementos interactivos como espacios con texto seleccionables y dropdowns. Los espacios con texto seleccionables permiten a los usuarios interactuar con información específica, mientras que los dropdowns ofrecen opciones ocultas que se despliegan cuando el usuario lo requiere, mejorando la experiencia de usuario y la organización de la

información.

Inputs

Para la entrada de datos, ImGui ofrece una variedad de métodos útiles. `InputText()`, por ejemplo, permite a los usuarios ingresar texto en campos designados. Además, se pueden utilizar otros métodos como `InputInt()`, `InputFloat()`, y `InputDouble()` para gestionar diferentes tipos de datos de entrada.

Recepción y Envío de Assets Entre Ventanas

ImGui es especialmente útil para recibir y enviar assets entre ventanas del editor. Esto se logra mediante la implementación de ventanas de ImGui que permiten la interacción del usuario para cargar, modificar o eliminar activos. Por ejemplo, se puede utilizar ImGui para agregar assets al explorador de archivos o para crear entidades arrastrando y soltando imágenes en la escena.

Inicialización

Para inicializar ImGui con SDL hemos seguido un procedimiento específico. Primero, se debe iniciar SDL, que es la librería utilizada para la gestión de ventanas y gráficos. Luego, se crea el contexto de ImGui utilizando `ImGui::CreateContext()`. Finalmente, se configuran las conexiones con la plataforma y el renderizador utilizando las funciones correspondientes, como:

```
ImGui_ImplSDL2_InitForSDLRenderer(window, renderer);  
ImGui_ImplSDLRenderer2_Init(renderer);
```

Este proceso asegura que ImGui funcione de manera efectiva junto con SDL en el proyecto.

4.2. Gestión de escenas

La clase `Scene` desempeña un papel de vital importancia en el editor, ya que asume la responsabilidad de gestionar las distintas escenas disponibles. Cada instancia de esta clase contiene listas de objetos y overlays, que serán renderizados en la ventana principal del editor. Además, `Scene` presenta funcionalidades clave para ejecutar operaciones esenciales. Por ejemplo, facilita la adición de entidades y superposiciones, la capacidad de guardar y cargar escenas desde archivos en formato JSON, así como también la gestión tanto de la interfaz de usuario como de la representación visual de los elementos



presentes en la escena. La clase **Scene** incluye una cámara virtual que se integra en la escena y permite al usuario explorar y visualizar el entorno desde diferentes perspectivas.



Renderizado de las entidades y cámara virtual

Para renderizar la escena, la cámara virtual contiene una textura de destino donde se renderizarán todas las entidades utilizando la biblioteca SDL. Este proceso tiene en cuenta la posición, tamaño y nivel de zoom de la cámara virtual. Una vez que todas las entidades se han renderizado en esta textura, se utiliza una ventana de ImGui para renderizar esa textura target y mostrar la escena completa en la ventana principal del editor. Esto proporciona una representación visual precisa de la escena, permitiendo al usuario ver todos los elementos presentes en la misma.

Es importante destacar que las entidades y overlays no se manejan de la misma manera en el proceso de renderizado.

Métodos relevantes de la clase Scene

- **AddEntity()**: Este método permite añadir una nueva entidad a la lista de objetos de la escena. Las entidades representan elementos visuales u objetos interactivos presentes en la escena. **AddEntity()** tiene dos versiones, una que recibe la propia entidad ya creada y otra que la construye a partir de la ruta de una imagen.
- **AddOverlay()**: Similar al método anterior, **AddOverlay()** agrega elementos de la interfaz a la lista de superposiciones de la escena. Las superposiciones son elementos que se muestran por encima de las entidades y pueden contener información adicional. **AddOverlay()** tiene dos versiones, una que recibe la propia entidad ya creada y otra que la construye a partir de la ruta de una imagen.
- **SaveScene()**: El método **SaveScene()** cumple con la tarea de llevar a cabo la serialización integral de la información de la escena y sus entidades asociadas en un archivo **'scene'** en formato JSON. Esto garantiza la preservación y almacenamiento de la configuración completa de la escena, incluyendo tanto los detalles de la propia escena como las propiedades de sus entidades. Esta información podrá ser recuperada en el futuro de manera precisa y coherente en el método **LoadScene()**.
- **LoadScene()**: Mediante **LoadScene()**, es posible cargar una escena previamente guardada en un archivo **'scene'**. Esta función reconstruye la estructura de la escena y de sus entidades a partir de los datos almacenados en el archivo.

- *RenderUI()*: Esta función tiene la responsabilidad de renderizar la interfaz de usuario (UI) asociada a la escena, incluyendo los overlays mencionados previamente. Es importante destacar que renderizar elementos de **Overlay** difiere de renderizar objetos con **transform**, una similitud que aquellos familiarizados con plataformas como Unity podrían encontrar. Mientras que los objetos con transform representan entidades en la escena con atributos de posición, orientación y escala, los overlays son elementos de interfaz que se superponen en la escena, proporcionando información contextual o funcionalidades adicionales sin afectar directamente la posición o estructura de los objetos en la escena tridimensional.
- *RenderEntities()*: El método **RenderEntities()** desempeña la función específica de renderizar las entidades que poseen transform en la escena, presentándolas visualmente en la ventana principal del editor. Mientras **RenderUI()** se enfoca en la interfaz de usuario y superposiciones, **RenderEntities()** se centra únicamente en la visualización de las entidades con atributos con **Transform**.
- *HandleInput()*: La función **HandleInput()** desempeña un papel crucial al gestionar las entradas del usuario, que comprenden acciones como clics de ratón y pulsaciones de teclas. A través de esta función, el usuario tiene la capacidad de interactuar con la escena y sus componentes. Además de permitir la selección y manipulación de entidades, este método también cumple un rol esencial en el control de la cámara. Mediante la interpretación de las entradas, es posible ajustar la vista de la cámara, realizar movimientos y, en definitiva, modificar la perspectiva con la cual se visualiza la escena.
- *Behaviour()*: La función **Behaviour()** orquesta el funcionamiento general de la escena en el editor. Esto incluye la ejecución de la interfaz de usuario, la representación visual de las entidades y superposiciones, y la respuesta a las interacciones del usuario.

En conjunto, estos métodos permiten que la clase **Scene** desempeñe una función esencial en la creación y manipulación de escenas dentro del editor, garantizando una experiencia interactiva y eficiente para los usuarios.

4.3. Gestión de ventanas

Las ventanas en nuestro editor se gestionan a través de un vector de la clase **Window** que se encuentra dentro de la clase **Editor**. El editor tiene la responsabilidad de renderizar y manejar el input de estas ventanas de manera centralizada.

La clase `Window` sirve como la clase base para cada ventana en el editor, y todas las ventanas heredan de ella. Esta clase base proporciona funcionalidades comunes para todas las ventanas. Algunos de los métodos relevantes incluyen:

- *IsFocused()*: Este método permite verificar si la ventana está enfocada o activa, lo que puede ser útil para determinar la interacción del usuario.
- *IsDocked()*: Verifica si la ventana está acoplada a otra ventana o se encuentra en modo flotante, lo que puede afectar su diseño y ubicación.
- *ReceiveAssetDrop()*: Gestiona la recepción de activos que se arrastran y sueltan en la ventana, permitiendo la incorporación de recursos a la ventana de manera eficiente.
- *GetPosition()*: Obtiene la posición actual de la ventana.
- *SetPosition()*: Establece la posición de la ventana.
- *GetSize()*: Obtiene el tamaño actual de la ventana.
- *SetSize()*: Establece el tamaño de la ventana.
- *Hide()* y *Show()*: Controlan la visibilidad de la ventana, lo que puede ser útil para mostrar u ocultar paneles según las necesidades del usuario.
- *IsMouseHoveringWindow()*: Determina si el cursor del mouse se encuentra sobre la ventana en ese momento, lo que puede ser relevante para eventos de interacción.
- *HandleInput()*: Se encarga de gestionar la entrada de usuario, lo que incluye la capacidad de responder a eventos como clics de ratón y pulsaciones de teclas.
- *Behaviour()*: Este método se encarga de inicializar la ventana y establecer su tamaño utilizando las funciones de ImGui, como `ImGui::Begin()`, `ImGui::End()`, `ImGui::SetNextWindowSize()`, y `ImGui::SetNextWindowPos()`. En medio de este método, se llama a `Behaviour()`, que define el comportamiento específico de cada ventana que hereda de la clase `Window`. Por ejemplo, las ventanas de jerarquía `Hierarchy` o componentes `Components` tendrán sus propios comportamientos personalizados para interactuar con el usuario de manera adecuada.

Esta estructura permite una gestión flexible y coherente de las ventanas en el editor, lo que facilita la creación de una interfaz de usuario rica y eficiente.

4.4. La clase Editor y WindowLayout

La clase Editor y sus estados

La clase **Editor** desempeña un papel fundamental en la estructura de nuestro proyecto. Esta clase centraliza la gestión de las ventanas y su estado en el editor. Una característica destacada es la presencia de una pila de estados, que permite alternar entre dos estados principales: la ventana de scripting y el propio entorno del editor. Esta funcionalidad proporciona una forma eficaz de cambiar el contexto de trabajo y facilita la programación y el diseño visual en el editor.

La clase **Editor** también incluye métodos cruciales para guardar y cargar el estado de las distintas ventanas, determinando si están visibles o no. Los métodos **StoreWindowsData()** y **LoadWindowsData()** permiten mantener la configuración de la interfaz de usuario persistente entre sesiones del editor.

Window Layout y su utilidad

El proceso de renderizado de las ventanas en el editor se beneficia de la clase **WindowLayout**. Esta clase juega un papel importante al administrar diferentes layouts de ventanas. Los diseños se gestionan utilizando **ImGui::Dockbuilder()**, lo que permite una organización flexible de las ventanas en el espacio de trabajo.

Los métodos de **WindowLayout**, como **Update()** y **GetAllLayouts()**, facilitan la gestión y selección de diseños específicos para las ventanas. Cuando el usuario selecciona un diseño, este se aplica automáticamente al renderizar, lo que permite una experiencia de usuario personalizada y adaptable según las necesidades de cada tarea en el editor. La capacidad de cambiar rápidamente entre diseños de ventanas mejora la eficiencia y la comodidad en el flujo de trabajo del usuario.

4.5. Navegación y gestión de archivos con FileExplorer

La clase **FileExplorer** desempeña un papel crucial en nuestro editor al gestionar la ventana del explorador de archivos. Esta clase se encarga de controlar y mostrar el contenido de los directorios, así como de interactuar con los archivos y directorios presentes en el sistema de archivos del proyecto.

FileExplorer utiliza una variable auxiliar llamada **Entry** para almacenar información detallada sobre un archivo o directorio, incluyendo su ruta, nombre y extensión. Además, mantiene una cola de ficheros que representan el contenido del directorio actual. Esta cola se actualiza dinámicamente al

cambiar de directorio o al activar la función de `Refresh()` para reflejar el contenido más reciente.

Para navegar por los directorios y obtener información sobre ellos, `FileExplorer` hace uso de la biblioteca `filesystem`, que proporciona acceso a las clases `path` y `directory iterator` para la manipulación de rutas y la exploración de directorios. Entre los métodos relevantes de la clase `FileExplorer`, se encuentran:

- *`ProcessPath()`*: Este método se encarga de procesar la ruta del directorio actual y actualizar la cola de ficheros para reflejar su contenido. Facilita la navegación y actualización del explorador de archivos.
- *`DrawList()`*: Controla la representación visual de la lista de archivos y directorios en la ventana del explorador. Este método se encarga de mostrar el contenido de manera legible y accesible para el usuario.
- *`OnItemSelected()`*: Establece las acciones a realizar al seleccionar un elemento en la lista. Además, define comportamientos específicos para las extensiones de archivo, como `'scene'` o `'script'`, cuando se realiza doble clic sobre ellos, lo que facilita la interacción y edición de archivos relevantes en el proyecto.

La clase `FileExplorer` mejora la eficiencia y la comodidad del flujo de trabajo del usuario al proporcionar una interfaz intuitiva para la navegación y gestión de archivos y directorios en el proyecto del editor.

4.6. Paso de assets entre escenas

Una funcionalidad esencial de nuestro editor es la capacidad de transferir assets entre escenas de manera eficiente. Para lograr esto, hacemos uso de las capacidades de arrastrar y soltar proporcionadas por `ImGui`, específicamente a través de las funciones `BeginDragAndDropSource()`, `SetDragDropPayload()`, y `BeginDragAndDropTarget()`.

Hemos desarrollado una clase auxiliar llamada `Asset` que desempeña un papel crucial en este proceso. La clase `Asset` contiene información detallada sobre un activo, incluyendo su nombre, extensión, ruta, ruta relativa, indicador de si es un `prefab` y su identificador de `prefab`. Esta información es esencial para garantizar que los activos se transfieran de manera precisa y coherente entre las escenas.

Además, cada ventana en el editor, como parte de la clase `Window`, implementa el método `ReceiveAssetDrop()`. Este método es responsable de procesar un activo recibido, y cada ventana puede personalizar su propia lógica para manejar activos específicos según sus necesidades. Por ejemplo, una ventana de escena podría procesar activos gráficos, mientras que una

ventana de lógica podría manejar scripts. La capacidad de personalizar el comportamiento de la transferencia de activos es fundamental para adaptarse a las necesidades de cada ventana y facilitar la manipulación de activos en el proyecto.

4.7. Entidades



En el contexto del editor, las entidades son elementos fundamentales que componen la escena. Cada entidad se distingue por un identificador único asignado a ella, lo que permite una diferenciación clara entre las diversas entidades presentes. Además, una entidad también pueden estar asociada con una textura, aunque esta asociación no es obligatoria, lo que significa que una entidad podría ser simplemente una entidad vacía. Destacar la existencia también del componente **Image**, que permite modificar la ruta de la imagen asociada a la textura.

Métodos relevantes de la clase **Entidad**

- *AssignId()*: El método **AssignId()** gestiona la asignación y **desasignación** de identificadores a las entidades. Esto permite que cada entidad tenga un identificador único que la distinga de otras en la escena.
- *RenderTransform()*: Con el método **RenderTransform()**, las entidades se presentan en la pantalla, lo que implica su visualización en la ventana principal del editor.
- *Update()*: El método **Update()** se encarga de actualizar ciertos atributos de la entidad en cada frame. Esta función es esencial para mantener la coherencia y la actualización constante de las propiedades de las entidades durante la ejecución del editor.
- *HandleInput()*: permite a las entidades responder a las entradas del usuario, como clics de ratón y pulsaciones de teclas.
- *AddComponent()*: posibilita la adición de componentes a la entidad. Los componentes son módulos que agregan funcionalidad a la entidad, como algún Collider o Animación, entre otras.
- *AddScript()*: De manera similar a **AddComponent()**, **AddScript()** permite agregar scripts a la entidad. Los scripts son fragmentos de código que definen comportamientos específicos para la entidad. En el caso de nuestro editor, dichos scripts se generan automáticamente a través de nodos visuales. Estos nodos proporcionan una interfaz visual para crear comportamientos y lógica sin necesidad de escribir código directamente.

- *SetComponents()*: `SetComponents()` se utiliza para establecer la lista de componentes asociados a la entidad. Util a la hora de crear una entidad copia de un prefab
- *SetScripts()*: Con `SetScripts()`, es posible definir los scripts que se aplicarán a la entidad. Util a la hora de crear una entidad copia de un `Prefab`
- *ToDelete()*: Mediante `ToDelete()`, se marca la entidad para su eliminación posterior. Este método permite gestionar la eliminación de entidades de manera controlada.
- *IsTransform()*: se emplea para determinar si una entidad es de tipo `Transform` o si se trata de un `Overlay`. Esto permite una diferenciación en el manejo de las entidades según su naturaleza.



Estos métodos, en conjunto, definen la funcionalidad y el comportamiento de las entidades en el editor, permitiendo su manipulación, renderizado y gestión de manera efectiva y coherente.

4.8. Jerarquía en las escenas

En el entorno del editor, la organización jerárquica de los elementos es una característica fundamental que permite una gestión coherente y eficiente de las entidades presentes en la escena. La clase `Entidad` se convierte en un componente clave para establecer esta jerarquía, ya que cada instancia incluye punteros tanto a su entidad padre como a una lista de entidades hijas.

La jerarquía de entidades también se refleja en la gestión de los `transform`. Si una entidad tiene un padre, estos valores locales se vuelven relativos al padre, lo que garantiza que los movimientos y ajustes de transformación sean coherentes respecto a la jerarquía.

Para gestionar estos aspectos, la clase `Transform` se encarga de proporcionar métodos para obtener y establecer tanto los valores globales como los relativos.

Métodos relevantes de la clase Entidad para la gestión de la jerarquía

- *AddChild()*: permite agregar una entidad como hijo de la entidad actual, estableciendo así una relación jerárquica entre ambas.
- *RemoveChild()*: permite eliminar una entidad de la lista de hijos de la entidad actual, rompiendo la relación jerárquica.

- *SetParent()*: se encarga de establecer la entidad padre de la entidad actual, ajustando sus valores locales de transformación de acuerdo con la jerarquía. También es posible llamar al método con `nullptr` establecer que la entidad carece de padre.

Métodos relevantes de la clase Transform para la gestión de la jerarquía

- *GetWorldPosition()*: se obtiene la posición global de la entidad, considerando la transformación jerárquica en la estructura.
- *GetWorldScale()*: se obtiene la escala global de la entidad, considerando la estructura jerárquica.
- *GetWorldRotation()*: devuelve la rotación global de la entidad, teniendo en cuenta la jerarquía en la transformación.
- *SetWorldPosition()*: permite establecer la posición global de la entidad, ajustando sus valores locales y considerando la jerarquía.
- *SetWorldScale()*: permite establecer la escala global de la entidad, ajustando sus valores locales y respetando la jerarquía.
- *SetWorldRotation()*: se utiliza para establecer la rotación global de la entidad, considerando la jerarquía y ajustando sus valores locales.
- *GetLocalPosition()*: devuelve la posición local de la entidad, que es relativa a su entidad padre en la jerarquía.
- *GetLocalScale()*: devuelve la escala local de la entidad, que se relaciona con su entidad padre en la jerarquía.
- *GetLocalRotation()*: devuelve la rotación local de la entidad, en relación con su entidad padre en la jerarquía.
- *SetLocalPosition()*: permite establecer la posición local de la entidad, considerando su entidad padre en la jerarquía.
- *SetLocalScale()*: permite establecer la escala local de la entidad, considerando su entidad padre en la jerarquía.
- *SetLocalRotation()*: permite establecer la rotación local de la entidad, considerando su entidad padre en la jerarquía.



Estos métodos, en conjunto, permiten establecer y mantener la jerarquía entre entidades y gestionar sus transformaciones de manera coherente, garantizando la organización precisa y eficiente de la escena en el editor. En cuanto a su visualización, la jerarquía se representa mediante indentaciones y dropdowns en el editor.

4.9. Prefabs y PrefabManager



Los prefabs son copias de entidades que se guardan como plantillas para su posterior instanciación o para manejar varias instancias de un mismo prefab mientras se comparte una base común. Estos prefabs tienen la particularidad de que su identificador ID es negativo, lo que los distingue de las entidades regulares. Esta característica les permite limitar ciertas funcionalidades, como la capacidad de referenciar otras entidades a través de scripts.

Cada entidad tiene un atributo llamado `'prefabId'`, el cual, en caso de ser negativo, indica que se trata de una instancia de un prefab. La gestión de prefabs se lleva a cabo mediante la clase `PrefabManager`, que mantiene una lista de todos los prefabs disponibles, junto con un mapa donde las claves son las IDs de los prefabs y los valores son vectores que contienen los ids de las entidades que son instancias de ese prefab.

Métodos relevantes de PrefabManager para la gestion de prefabs

- *UpdatePrefabInstances()*: Este método se encarga de actualizar las instancias de los prefabs en el escenario, asegurando su coherencia y consistencia.
- *AddPrefab()*: permite agregar un nuevo prefab a la lista de prefabs disponibles.
- *AddInstance()*: agrega una referencia a una instancia de un prefab al vector correspondiente en el **mapa de instancias**.
- *RemoveInstance()*: elimina la referencia a una instancia de un prefab del vector correspondiente en el mapa de instancias. Este método acepta tanto un puntero a la propia entidad que queremos quitar de la lista o bien dos ids, **la del prefab y la de la entidad instanciada**.
- *GetPrefabs()*: devuelve la lista de todos los prefabs disponibles.
- *GetPrefabById()*: permite obtener un prefab específico según su id.

Métodos relevantes de la clase Entidad para la gestion de prefabs

- *IsPrefab()*: indica si la entidad es un prefab o no.
- *IsPrefabInstance()*: verifica si la entidad es una instancia de un prefab.
- *GetPrefabId()*: devuelve la id del prefab al que pertenece la entidad en caso de ser una instancia de un prefab.

- *SetPrefabId()*: establece la id de prefab para una entidad, lo que la convierte en una instancia de ese prefab.
- *GetTopParentPrefab()*: devuelve la entidad de nivel superior dentro de la jerarquía de instancias de un mismo prefab.

La gestión de prefabs mediante la clase **PrefabManager** permite mantener un control organizado de las plantillas y sus instancias, facilitando la edición y manipulación coherente de la escena en el editor. Para su diferenciación visual con el resto de entidades, se dibujan de otro color dentro del editor.

4.10. Componentes

En el contexto del sistema descrito, los componentes juegan un papel fundamental al definir el comportamiento y las propiedades de las entidades en el motor. Los componentes son leídos desde un archivo JSON del motor, el cual contiene información sobre cada componente, sus atributos y funciones.



La estructura de un componente se organiza en clases que facilitan su manejo y uso en el motor. Cada componente se compone de atributos y funciones que definen su comportamiento y propiedades. La información sobre atributos y funciones se almacena en clases específicas **Attribute** y **Function**, y todo esto se agrupa bajo la clase **Component**.

La clase **Entidad** desempeña un papel esencial en la gestión de componentes. Cada entidad contiene una lista de componentes que define sus características y comportamientos. Los componentes se serializan junto a la entidad a la que pertenecen.

Métodos relevantes de la clase **Attribute**

- *GetValue()*: devuelve el valor actual del atributo.
- *SetValue()*: establece el valor del atributo.
- *GetType()*: devuelve el tipo del atributo.
- *GetName()*: devuelve el nombre del atributo.

Métodos relevantes de la clase **Function**

- *SetReturn()*: establece tipo de retorno de la función.
- *GetReturn()*: devuelve el tipo de retorno de la función.
- *AddInput()*: añade un posible input a la función.

- *GetName()*: devuelve el nombre de la función.
- *GetComponent()*: devuelve el nombre del componente al que pertenece la función.

Métodos relevantes de la clase Component

- *GetName()*: devuelve el nombre del componente.
- *GetAttribute()*: permite obtener un atributo específico de un componente mediante su nombre.
- *GetFunction()*: permite obtener una función específica de un componente utilizando su nombre
- *FromJson()*: reconstruye un componente a partir de un fragmento en formato JSON.
- *ToJson()*: lleva a cabo la serialización integral de la información del componente y sus atributos y funciones asociadas en un archivo en formato JSON.

Métodos relevantes de la clase Entidad para la gestión de componentes

- *AddComponent()*: añade un componente a la entidad.
- *GetComponents()*: devuelve el mapa de componentes de la entidad.
- *SetComponents()*: recibe y una lista de componentes para asignarsela a la entidad.

En resumen, los componentes y su relación con las entidades en el motor permiten una estructuración eficiente y una personalización precisa de la funcionalidad de cada elemento en el mundo virtual, enriqueciendo la experiencia del usuario y posibilitando un proceso de desarrollo más fluido y adaptativo.

4.11. Ejecución del juego, estructura de carpetas y build del motor

Ejecución del juego y redirección de la salida mediante tuberías

La ejecución del juego en nuestro editor se maneja a través de dos botones ubicados en la parte superior, que permiten ejecutar el juego en modo de



depuración (*'debug'*) o en modo de lanzamiento (*'release'*). Estos botones invocan el método `Play()` de la clase `Game`. El método `Play()` se encarga de configurar una tubería para redirigir la salida del juego hacia el editor, y ser mostrado así en la consola. Además, inicia un hilo que se encarga de capturar y guardar los datos provenientes de esa tubería. Cuando el juego se cierra, se llama al método `Stop()`, que cierra el hilo y finaliza el proceso del juego de manera ordenada.

Estructura de carpetas



En cuanto a la estructura de carpetas, en la solución del Editor se encuentra una carpeta `Editor` dedicada para almacenar los assets y configuraciones necesarios para el proyecto. Dentro de esta carpeta, se encuentra otra denominada `Engine` que alberga los ejecutables y recursos generados a partir de la build del motor del juego. Por otro lado, al ejecutar el editor, se creará una carpeta adicional a través del `ProjectManager` que contendrá exclusivamente los recursos específicos de nuestro proyecto de videojuego.

Build del motor



Para realizar la construcción del motor, es esencial configurar el directorio de salida en la ubicación mencionada anteriormente. Además, se requiere copiar los archivos *'Components.json'* y *'Managers.json'* desde la carpeta *'src/ecs/ECSUtilities'* hacia esta misma carpeta, asegurando que todos los recursos necesarios estén disponibles para el funcionamiento adecuado del motor.

En resumen, la ejecución del juego se gestiona mediante botones en el editor, con un sistema que redirige la salida y captura los datos del juego. La estructura de carpetas está organizada de manera que los activos y configuraciones se encuentren separados de los recursos generados por la construcción del motor, facilitando así la gestión del proyecto.

4.12. Gestión de proyectos con ProjectManager

El `ProjectManager`, o gestor de proyectos, es una clase esencial en nuestro editor que permite una organización eficiente de varios proyectos de videojuegos. Su funcionalidad incluye la creación y administración de directorios dedicados para cada proyecto, donde se almacenan todos los recursos relacionados con ese juego en particular. Además, el gestor serializa información esencial de cada proyecto en un archivo con extensión *'shyproject'*. Esta información incluye detalles como los prefabs, la última fecha de apertura del proyecto, el directorio del proyecto y las preferencias específicas.

Una característica destacada es la capacidad de cargar proyectos exis-

tentes a partir de archivos `'shyproject'`. Esto facilita el acceso a proyectos previamente creados y su restauración con toda su configuración previa.

El **ProjectManager** también implementa una funcionalidad de historial que almacena una lista de proyectos recientes. Esto permite a los usuarios acceder rápidamente a proyectos que han sido abiertos anteriormente. El historial se gestiona mediante la creación de un archivo en el directorio `App-Data` que contiene los directorios de los últimos proyectos abiertos.

Entre los métodos relevantes en la clase **ProjectManager**, se encuentran:

- *NewProject()*: inicia la creación de un nuevo proyecto, creando un directorio dedicado y serializando la información esencial en un archivo `'shyproject'`.
- *OpenProject()*: permite cargar proyectos existentes a partir de archivos `'shyproject'`, restaurando toda la configuración y recursos asociados al proyecto.
- *SaveProject()*: guarda los cambios y la información esencial del proyecto actual, asegurando que las últimas modificaciones se reflejen en el archivo `'shyproject'`.

El **ProjectManager** desempeña un papel fundamental en la administración y organización de proyectos de videojuegos en nuestro editor, lo que mejora la eficiencia y la comodidad para los desarrolladores.

4.13. Gestión de recursos con **ResourceManager**

El **ResourceManager**, o gestor de recursos, es un componente crucial en nuestro editor que se encarga de gestionar recursos como texturas y fuentes de manera eficiente. Se implementa como un patrón Singleton para garantizar que una única instancia gestione todos los recursos del editor de manera centralizada.

La funcionalidad principal del **ResourceManager** se basa en la utilización de dos mapas, donde las texturas y fuentes se almacenan como valores asociados a sus respectivos nombres clave. Esto permite que una misma imagen o fuente se reutilice en todo el editor sin necesidad de crear instancias adicionales cada vez que se requieran. Cuando se llama a métodos como `AddTexture()` o `AddFont()`, el gestor de recursos verifica si el recurso ya existe en el mapa. Si existe, no se crea una nueva instancia, lo que ahorra recursos y mejora la eficiencia.

Algunos de los métodos relevantes en la clase **ResourceManager** incluyen:

- *AddTexture()*: agrega una textura al mapa de recursos. Si la textura no existe, se devuelve la instancia existente; de lo contrario, se crea y almacena en el mapa. Además devuelve dicha textura.

- *AddFont()*: similar a *AddTexture()*, este método agrega una fuente al mapa de recursos y verifica si ya existe antes de crearla. Además devuelve la fuente.
- *GetInstance()*: para acceder al gestor de recursos, se utiliza el método *GetInstance()*, que garantiza que solo exista una instancia de la clase **ResourceManager** en todo el editor.



El **ResourceManager** contribuye significativamente a la eficiencia y la gestión de recursos en el editor al permitir la reutilización de texturas y fuentes, lo que es fundamental para un flujo de trabajo fluido y una interfaz de usuario rica en detalles.

4.14. Configuración de preferencias y paletas de colores

Dentro de la opción Edit en la barra de menú de nuestro editor se encuentra la ventana de preferencias. La clase **Preferences** se encarga de almacenar información relacionada con la configuración del editor. Esto incluye detalles como las capas de colisión, la escena inicial, las opciones de audio, el mapeo de controladores o teclado y la ubicación para generar la compilación final del proyecto.

Una característica destacada es la capacidad de personalización que ofrece **Preferences**, permitiendo a los usuarios adaptar el entorno del editor según sus preferencias y necesidades de desarrollo.

Además, en nuestro editor contamos con la clase **ColorPalette**, que desempeña un papel esencial en la gestión de paletas de colores. Para ello, hace uso de la clase **Palette**, que almacena distintos colores para una variedad de elementos y opciones dentro del editor. Estos colores pueden variar desde el fondo de la interfaz hasta los colores de las flechas utilizadas en la representación visual de nodos en el scripting.

Para una gestión eficiente de múltiples paletas de colores, **ColorPalette** almacena las paletas en un mapa, utilizando el nombre de cada paleta como clave. Esto permite una fácil selección y cambio de paletas según las necesidades del usuario y el contexto de trabajo en el editor.

En conjunto, las clases **Preferences** y **ColorPalette** contribuyen a la versatilidad y personalización de nuestro editor al proporcionar opciones de configuración y una gestión de colores flexible y adaptable.

4.15. Generación de build del juego

Dentro del editor, una funcionalidad crucial es la capacidad de generar una build ejecutable del juego. Esto permite a los desarrolladores crear una versión independiente del juego que puede ser distribuida y ejecutada por los usuarios finales. Para llevar a cabo esta tarea, utilizamos la clase `Build`.

El método principal en la clase `Build` es `GenerateBuild()`, que se encarga de iniciar la generación de la build del juego. Para asegurar que este proceso no afecte negativamente la interacción con el editor, se ejecuta en un hilo separado para evitar bloqueos y mantener la fluidez del entorno de desarrollo.

Dentro de `GenerateBuild()`, se llama al método `BuildThread()` mediante un hilo, que es responsable de realizar las operaciones de copiado necesarias para construir la versión del juego. Esto incluye la copia de archivos esenciales como assets, bibliotecas DLL y cualquier otro recurso requerido para la ejecución del juego.

Entre los métodos relevantes en la clase `Build`, se encuentran:

- *GenerateBuild()*: inicia el proceso de generación de la build del juego, ejecutando `BuildThread()` en un hilo separado.
- *BuildThread()*: este método se encarga de copiar los archivos esenciales necesarios para la build del juego. La ejecución en un hilo separado garantiza que el proceso no afecte la experiencia del usuario en el editor.
- *Copy()*: método utilizado para copiar archivos desde el directorio de desarrollo del proyecto a la ubicación de la build, asegurando que todos los recursos necesarios estén disponibles para la ejecución del juego.

La capacidad de generar builds del juego es esencial para la distribución y prueba de los proyectos de desarrollo, y la clase `Build` en nuestro editor simplifica este proceso, permitiendo a los desarrolladores crear versiones jugables del juego de manera eficiente.

Capítulo 5

Motor

El motor esta dividido en diez proyectos de Visual Studio, todos dentro de la misma solución. Cada proyecto cumple una función específica de la que pueden depender otros proyectos y todo el código está escrito en C++

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.

5.1. 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. En este caso, esta clase se puede usar simplemente como un contenedor de información en el que se pueden asociar dos números reales pero también se puede usar para hacer cálculos geométricos en dos dimensiones como rotaciones, cálculo de ángulos, etc.
- **Random**: Contiene métodos estáticos útiles para calcular aleatoriedad entre números enteros, números reales, ángulos, y colores. En este caso esta clase solo tiene como objetivo proporcionar funcionalidad.
- **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. También existen

métodos que aportan comodidad a la hora de crear colores como Red, Green, Blue, Orange, Black, que simplemente construyen el color por dentro sin necesidad de conocer su valor en el modelo RGB.

- **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. Por otro lado, implementa funcionalidad para conocer la tasa de frames o convertir un valor de tiempo en una cadena de texto formateada. Mencionar también que esta clase es un **Singleton**.
- **Singleton**: Una plantilla para crear instancias estáticas a través de herencia. Es decir, en caso de querer convertir una clases en un **Singleton**, muy útiles para managers, simplemente hay que heredar de esta clases para conseguirlo. Aporta mucha comodidad ya que evita tener que implmentar la instancia estática de la clase y sus métodos para manejarla. Solo tiene un inconveniente y es borrar los **Singleton** en el orden adecuado si dependen entre ellos.

5.2. 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***, **SoundEffect***, **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 esta duplicado y es responsabilidad del desarrollador ordenar sus archivos de assets.

Por último, en la destructora de la clase se borran todos los recursos de todos los mapas.

5.3. Sonido

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

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

Para la música **MixMusic**, 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 **MixChunk**.

Este proyecto cuenta con tres clases:

- **SoundEffect**: Representa un efecto de sonido. Contiene la información de un **MixChunk** de **SDLMixer** y un identificador usado posteriormente por el componente **SoundEmitter**.
- **MusicEffect**: Representa un sonido de música de fondo. Contiene la información de un **MixMusic** de **SDLMixer** 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 envoltorio de las funciones principales de **SDLMixer** para reproducir, pausar, 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.

En cuanto a los sonidos, todos los métodos de **SDLMixer** necesitan un canal y un **MixChunk***. Esto choca con la idea del componente **SoundEmitter**, que visto desde la perspectiva del usuario, simplemente se le establece un sonido y ya se puede reproducir, sin necesidad de conocer la existencia de canales. Esto se contará más en detalle en la implementación de **SoundEmitter**.

Las funciones disponibles para los canales de sonido son: reproducir, fade-in, fade-out, pausar, detener (la diferencia con pausar es que si se detiene no se puede renaudar), renaudar, consultar si un canal está reproduciendo un sonido, establecer el sonido de un canal, consultar el volumen de un canal, establecer la posicion en el espacio 2D de un canal y establecer el paneo de un canal.

En cuanto a la música, los métodos de **SDLMixer** solo necesitan un **MixMusic** ya que solo hay un canal por lo que el problema de los canales des-

aparece. Las funciones disponibles son: reproducir, fade-in, fade-out, pausar, detener, renaudar, modificar el volumen de la música y rebobinar.

5.4. 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. Con estos enumerados se crean posteriormente arrays con la información del estado de cada tecla.

Los eventos de SDL relacionados con el teclado son **KEYDOWN** y **KEYUP**. El manager implementa métodos de usuario para conocer si se ha pulsado o soltado alguna tecla o si una tecla está pulsada, acaba de ser pulsada, acaba de ser soltada o no está pulsada.
 - *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. Para ello, el manager cuenta con un **Vector2D** para la posición, booleanos para el estado de pulsado/no pulsado de los botones y un número entero para representar si la rueda del ratón está haciendo scroll hacia abajo o hacia arriba.

Los eventos de SDL relacionados con el ratón son **MOUSEWHEEL**, **MOUSEMOTION**, **MOUSEBUTTONDOWN** y **MOUSEBUTTONUP**. El manager implementa métodos de usuario para conocer si ha habido algún movimiento con el ratón o con la rueda, si se ha pulsado o soltado algún botón, la posición del ratón y el scroll actual de la rueda.

- *Mando*: Cuenta con soporte para múltiples mandos y cada uno de ellos guarda la siguiente información:
 1. Referencia al GameController creado por SDL.
 2. Id del GameController creado por SDL.
 3. Nombre del GameController.
 4. Estado de cada uno de los botones del GameController.
 5. Información del movimiento de los triggers del GameController.
 6. Información del movimiento de los joysticks del GameController.

El manager tiene soporte 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.

Los eventos de SDL relacionados con el ratón son `CONTROLLERDEVICEADDED`, `CONTROLLERDEVICEREMOVED`, `JOYBUTTONDOWN`, `JOYBUTTONUP`, `JOYAXISMOTION`.

El manager implementa métodos de usuario para conocer el número de mandos conectados, si algún mando ha pulsado o soltado algún botón, si algún mando ha movido los joysticks o los triggers y si ha habido alguna conexión o desconexión.

Comentar también que, antes de esta implementación, el manager usaba `SDLJoystick` para el manejo de los mandos pero debido a cierta incomodidad con los eventos se hizo el cambio a `SDLGameController`. La diferencia entre ambos es que `SDLJoystick` es la API más antigua de SDL para manejar mandos y joysticks y proporciona una interfaz de bajo nivel para interactuar con dispositivos de entrada mientras que `SDLGameController` proporciona una abstracción de más alto nivel para interactuar con mandos, lo que facilita la detección de mandos y el acceso a sus entradas y es la opción recomendada para la mayoría de los desarrolladores.

Por último, el manager implementa métodos de lógica para el usuario como movimiento horizontal y vertical, salto o acción. Estos métodos usan la información que se haya establecido en el editor de las teclas y botones que se desean usar para moverse, saltar o realizar una acción.

5.5. 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.

5.6. 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 `SDLMixer`, algunos comentarios 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.
- *Píxeles*: Es tentador usar píxeles 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 píxeles de longitud sería visto por `Box2D` como el tamaño de un edificio de 45 pisos.

Para resolver el problema de los píxeles, el manager declara una variable `'screenToWorldFactor'` usada para convertir de píxeles a unidades

físicas y viceversa. Por lo tanto, a la hora de crear cuerpos físicos se convierte el tamaño en píxeles 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, Box2D proporciona los **CategoryBits** y los **MaskBits** para ello. Los **CategoryBits** especifican la capa en la que se encuentra un objeto y los **MaskBits** las capas con las que 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.

Tanto los **CategoryBits** como los **MaskBits** se representan en hexadecimal y el su valor por defecto es *'0x0001'* en caso de los **CategoryBits** y *'0xFFFF'* en caso de los **MaskBits**.

La comprobación de colisión tiene este aspecto:

```
bool colision = (bodyA.maskBits AND bodyB.categoryBits) == 0
&& (bodyA.categoryBits AND bodyB.maskBits) == 0
```

Por lo tanto, por defecto, todos los cuerpo creados va a estar en la capa *'0x0001'* y van a colisionar con todas las capas.

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 calcula su **CategoryBits** a partir de ese índice.

Para calcular sus **MaskBits** entra en juego otro funcionalidad que es la matriz de colisiones. En ella se ajustan la colisión entre las capas existentes y posteriormente se calculan los **MaskBits** de una capa dada.

Como métodos al usuario, el manager proporciona poder cambiar la gravedad del mundo físico, añadir o eliminar capas, establecer colisión entre capas y consultar si dos capas colisionan.

- *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 píxeles.

5.7. Renderer

Este proyecto tiene como objetivo iniciar la librería de SDL y SDLImage. Las clases que tiene este proyecto son las siguientes:

- *RendererManager*: Clase, **Singleton**, encargada de inicializar y cerrar la librería de SDL, SDLImage y SDLTTF. Contiene información y funcionalidad relacionada con la ventana como su tamaño, borde, icono, cursor, nombre y modo pantalla completa. Además, tiene la información de la cámara como su posición y escala. 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'*. Tiene métodos para obtener la **SDLTexture*** y para consultar el ancho y el alto de la misma.

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

5.8. Entity-Component-System

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

Para empezar, la idea de un sistema de componentes y entidades es la siguiente:

Es un patrón de diseño utilizado en el desarrollo de videojuegos y otros sistemas interactivos para organizar y gestionar la lógica y la funcionalidad de los objetos dentro del juego. En lugar de utilizar una jerarquía de clases tradicional o un sistema basado en objetos, el ECS descompone los elementos del juego en dos partes principales: entidades y componentes.

- *Entidades*: Las entidades son objetos vacíos que actúan como contenedores para componentes. Cada entidad representa un objeto o entidad en el juego, como un personaje, un enemigo, un objeto interactivo, etc. Las entidades no tienen lógica o comportamiento en sí mismas, simplemente contienen uno o más componentes.

- *Componentes*: Los componentes son bloques de datos y lógica que contienen información específica sobre el comportamiento o las propiedades de una entidad en el juego. Cada componente se enfoca en una única funcionalidad o característica del objeto. Por ejemplo, puedes tener un componente de *Posición* para almacenar la posición de una entidad en el mundo, un componente de *Renderizado* para pintar la entidad en la escena, un componente de *Física* para gestionar su comportamiento físico, etc.

Para llevar a cabo la implementación de este sistema es imprescindible el uso de programación orientada a objetos (POO) junto con herencia y polimorfismo.

En cuanto a las clases implementadas:

1. *Component*: Clase que representa a un componente. Contiene una referencia a la entidad a la que está asociado e información sobre si esta 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.

Estos métodos son los siguientes:

- *Init()*: método reservado para el motor donde se realiza toda la inicialización que necesita el componente para funcionar correctamente.
- *Start()*: método llamado inmediatamente después del *Init* donde el usuario puede realizar su propia inicialización.
- *Update()*: método llamado en cada vuelta del bucle principal.
- *LateUpdate()*: método llamado en cada vuelta del bucle principal inmediatamente después del *Update*.
- *Render()*: método llamado después del *Update* y *LateUpdate*. Destinado a implementar el renderizado del componente.
- *FixedUpdate()*: método llamado en un intervalo de tiempo fijo denominado paso físico. Destinado a implementar la física del componente.
- *OnActive()*: método llamado cuando se activa el componente.
- *OnDeactive()*: método llamado cuando se desactiva el componente.
- *OnSceneUp()*: método llamado cuando el componente se encuentra en la escena que se acaba de empezar a actualizar.

- *OnSceneDown()*: método llamado cuando el componente se encuentra en la escena que se ha dejado de actualizar.
- *OnDestroy()*: método llamado cuando el componente es eliminado.

Métodos para el manejo de colisiones:

- *OnCollisionEnter()*: método llamado cuando la entidad que contiene este componente ha colisionado con otra entidad.
- *OnCollisionStay()*: método llamado cuando la entidad que contiene este componente está colisionando con otra entidad.
- *OnCollisionExit()*: método llamado cuando la entidad que contiene este componente ha dejado de colisionar con otra entidad.
- *OnTriggerEnter()*: método llamado cuando la entidad que contiene este componente ha colisionado con otra entidad y, o bien el componente físico de esta entidad o bien el de la otra están marcados como trigger.
- *OnTriggerStay()*: método llamado cuando la entidad que contiene este componente está colisionando con otra entidad y, o bien el componente físico de esta entidad o bien el de la otra están marcados como trigger.
- *OnTriggerExit()*: método llamado cuando la entidad que contiene este componente ha dejado de colisionar con otra entidad y, o bien el componente físico de esta entidad o bien el de la otra están marcados como trigger.

Métodos para el Overlay (UI):

- *OnClickBegin()*: método llamado cuando se hace click sobre un elemento del componente **Overlay** de la entidad.
- *OnClickHold()*: método llamado cuando se mantiene clickado un elemento del componente **Overlay** de la entidad.
- *OnDoubleClick()*: método llamado cuando se hace doble click sobre un elemento del componente **Overlay** de la entidad.
- *OnRightClick()*: método llamado cuando se hace click derecho sobre un elemento del componente **Overlay** de la entidad.
- *OnMouseEnter()*: método llamado cuando el ratón entra sobre un elemento del componente **Overlay** de la entidad.
- *OnMouseHover()*: método llamado cuando el ratón se encuentra sobre un elemento del componente **Overlay** de la entidad.
- *OnMouseExit()*: método llamado cuando el ratón sale de un elemento del componente **Overlay** de la entidad.

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.

En cuanto a funcionalidad, contiene los mismos métodos que los componentes pero con implementación. Esta implementación simplemente consiste en llamar a los métodos de todos los componentes asociados a la entidad. Por ejemplo, el método **Render()** de la entidad recorre la lista de componentes asociados y llama al método **Render()** de cada uno. Aquí es donde entra la importancia de la herencia y el polimorfismo. Para crear un componente con funcionalidad, se debe heredar de la clase *Componente* e implementar los métodos virtuales disponibles. De esta manera, serán llamados por la entidad que contenga el componente creado.

Además de los métodos de los componentes, las entidades tienen métodos para añadir componentes, consultar si contienen un componente, eliminar componentes y obtenerlos. Estos métodos hacen uso de templates de C++ con un tipo genérico **T** y una restricción para asegurar que el **T** debe ser de tipo componente. Además, los parámetros de estos métodos reciben un paquete de parámetros variados de categoría **RValue**, por lo que dentro se utiliza la función **std::forward** para preservar esa categoría y evitar así copias innecesarias y garantizar un comportamiento predecible.

Por último mencionar que las entidades también tienen métodos para añadir scripts. 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, que proporciona el motor al usuario y los scripts son piezas de lógica que construye el usuario a partir del sistema de scripting visual y que, en general, es comportamiento específico al videojuego que esté desarrollando el usuario.

3. *Scene*: La última pieza que compone este ECS 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, la posición y escala de la cámara, y bastantes métodos comunes a las entidades y componentes. No todos porque hay algunos que no tienen sentido en las escenas, como los de físicas o los de UI. Al igual que las entidades, estos métodos cuentan

con funcionalidad y simplemente se dedican a llamar al método correspondiente de cada una de las entidades que contiene. Por ejemplo, cuando se actualiza una escena, el método **Update** de la escena lo único que hace es llamar al **Update** de sus entidades comprobando si el estado de la entidad le permite actualizarse, es decir, que esté activa y no esté eliminada.

Además, las escenas cuentan con métodos para crear entidades con identificador y sin identificador, buscar entidades por nombre, y eliminar entidades.

5.8.1. 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, por lo que la escena añadida pasa a ser la que se actualiza en el juego. Antes de añadirla al top, llama al método **OnSceneDown()** de la escena que está actualmente en el top para avisar de que esa escena va a dejar de actualizarse. Posteriormente, una vez añadida al top, se llama al **Start()** para la inicialización de la escena.
 - *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 ejecutarse.
 - *Operación CLEAR*: vacía la pila de escenas.

Por último comentar que para evitar problemas de ejecución, realmente el cambio de escenas se produce al final del bucle principal. Por lo que el método de cambiar escenas simplemente marca que escena se va a cambiar y al final del bucle principal se cambia.

2. *SceneLoader*: Clase encargado de leer la información de la escenas creadas en el editor. A la hora de construir la entidades diferencia entre entidades con **Transform** y entidades con **Overlay**. Para construir los componentes de las entidades se usa la clase **ClassReflection**, que es un **Singleton** encargado de guardar en un mapa la información de los componentes donde la clave es el nombre del atributo de los componentes y el valor es el propio valor del atributo.

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**. Esto se comenta en el apartado de componentes del motor pero todas las entidades contienen al menos un componente, **Transform** u **Overlay**. El **Overlay** lo contienen aquellas entidades destinadas a ser parte de la interfaz y el **Transform** todo el resto de entidades.
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.

5.8.2. 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. *Image*: Componente encargado de cargar una imagen y renderizarla en pantalla en la posición indicada por el transform de la entidad. Por ello, tanto este componente como todos los que requieran componente **Transform** para su correcto funcionamiento. Para cargar la imagen hace uso del manager de recursos para reutilizar la imagen en caso de estar ya creada por otra entidad.
3. *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 si es *'trigger'*, la fricción que genera, el rebote, 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.
4. *SoundEmitter*: Componente encargado de cargar un sonido e implementar métodos para reproducirlo, detenerlo, pausarlo, etc. Como se comentó anteriormente, **SDLMixer** dispone de un conjunto de canales para reproducir sonidos pero este componente es abstraer la necesidad de canales desde la perspectiva del usuario.
5. *MusicEmitter*: Componente encargado de cargar música e implementar métodos para reproducirla, detenerla, pausarla, rebobinarla, etc.

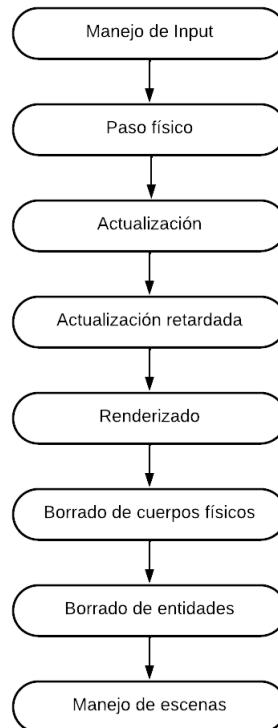
6. *ParticleSystem*: Componente encargado de implementar un sistema de partículas configurable. Tiene soporte para cargar texturas y mover las partículas con el motor de físicas Box2D.
7. *Animation*: Componente encargado de implementar la lógica de reproducción de animaciones a partir de una hoja de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

5.9. Main

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:

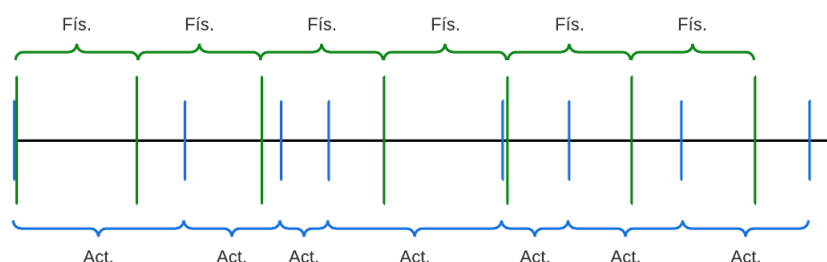


Además, de esos métodos, se realizan cálculos de tiempo para proporcionar al usuario el **DeltaTime**, tiempo transcurrido desde el inicio de la ejecución del programa o el número de frames/actualizaciones hasta el momento. El **DeltaTime** es una medida de tiempo, generalmente en milisegundos, que informa 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, sobretodo por motivos de estabilidad. Por ello, es necesario hacer cálculos adicionales para saber en que momentos se debe ejecutar el Paso Físico ya que no se puede llamar en cada frame, a diferencia de la **Actualización**.

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

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. Para llevar esto a cabo se necesitan dos contadores de tiempo, uno para la **Actualización** y otro para el **Paso físico**. Mientras que el contador de tiempo para el **Paso físico** esté por detrás temporalmente que el de **Actualización**, se llama al método **Paso físico** y se suma al contador el tiempo fijo. El tiempo fijo es un valor que se puede modificar en base a las necesidades del videojuego.

Capítulo 6

Scripting

6.1. Diseño del lenguaje



Nosotros nunca habíamos utilizado ninguna herramienta para programar usando nodos y flechas, y lo más parecido que habíamos usado era Construct 2 el cual es un motor en el que se programa usando bloques. Pero pese a nuestra inexperiencia en este ámbito no queríamos basarnos en editores de programación de nodos ya existentes como Unity o Unreal, ni usar ninguna biblioteca ya existente de visual scripting, sino desarrollar nuestra propia herramienta y adaptarla a las necesidades de nuestro motor.

Inicialmente teníamos una idea muy sencilla y simplificada de cómo sería nuestro sistema de nodos. Se basa en una serie de nodos, representados mediante cajas con muescas a la izquierda y a la derecha. Cada nodo representaría una función (o método) dentro del motor siendo las muescas de la izquierda los parámetros que recibiría dicha función y a la derecha el valor de salida.

Rápidamente nos dimos cuenta de las limitaciones de este sistema, siendo principalmente el hecho de que no permite introducir valores o variables, y solo permite operar mediante el uso de funciones. Entonces decidimos dividir los nodos por tipos según su función. Estaría el nodo que represente métodos (el mencionado anteriormente) y un nuevo tipo de nodo el cual representaría un valor constante, que podría ser usado para pasarse a la entrada de nodo método. En este nodo, el usuario podría decidir de qué tipo es el valor que quiere usar y asignar el valor correspondiente. Fue en este momento cuando tuvimos que decidir qué cuales eran los tipos que querríamos que soporte este lenguaje, y al estar pensado principalmente para programadores nos decantamos por los siguientes: Números, vectores, cadenas de texto, letras, booleanos, colores y entidades.

Sobre los tipos hay que mencionar que decidimos decantarnos por un conjunto de números para conseguir mayor simplicidad, no teniendo que dis-

tinguir entre distintos tipos como `int`, `uint`, `float`, `double`. . . y agruparlos todos en un conjunto números a secas, parecido al funcionamiento de javascript. Este nuevo tipo de nodo no tendría muescas de entrada al tratarse de un valor constante, y únicamente contaría con la muesca de salida para poder conectar el valor a un método. Estos nodos, como se esperaría no ejecutan nada dentro del motor y solo devuelven el valor asociado.

Con esto ya conseguimos tener un lenguaje funcional y procedimos con los primeros prototipos, hechos en Unity. Podíamos ejecutar programas simples y nos funcionaba correctamente para la depuración, pero entonces nos encontramos con el segundo bloqueo importante, siendo el control del flujo de los nodos, o más bien su ausencia.

Por control de flujo nos referimos a la posibilidad de ejecutar varios nodos función de forma consecutiva, ya que actualmente sólo se ejecuta el nodo inicial y sus nodos de entrada de forma recursiva, pero por ejemplo no podríamos ejecutar dos funciones independientes una detrás de la otra, ni tampoco tendríamos una forma de controlar el orden en que ejecutar estas funciones.

Para solucionar se creó un nuevo tipo de flecha. De esta forma cada nodo tendría una conexión de flujo con el nodo que se ejecutará inmediatamente después. Cuando un nodo está siendo ejecutado se dice que tiene el flujo actual.

Para ejecutar el nodo A, primero se ejecutan todos sus nodos entrada, que a su vez ejecuta sus propios nodos entrada y así sucesivamente hasta llegar o a un nodo de valor constante o a un nodo función sin nodos de entrada. Cuando toda la entrada haya sido procesada, se ejecuta el propio nodo y luego el flujo pasa al nodo siguiente. Este proceso se ejecutará constantemente hasta que el nodo siguiente del nodo con el flujo actual sea inexistente.

De este sistema cabe mencionar que un nodo de entrada no puede tener conexión con el nodo siguiente y además si en el proceso de ejecutar los nodos de entrada alguno de ellos tiene un nodo siguiente esa conexión será ignorada pues ninguno de esos métodos tiene el flujo actual.

Ahora con el flujo hecho el siguiente paso para mejorar el scripting es añadir funcionalidad para poder modificarlo. Para hacerlo hemos tenido que implementar en el lenguaje condicionales y bucles como nuevos tipos de nodos, al que llamamos bifuración. Este tipo de nodos tiene un nodo de entrada con la condición para el control del flujo, y tiene dos salidas de flujo que tendrán una función dependiendo del tipo de controlador que sea.

Existen tres tipos de controladores:

- *If*: Este tipo de controlador recibe como entrada un tipo booleano, y dependiendo del valor de la condición el flujo continuará por el primer flujo de salida o por el segundo.

- *While*: Este tipo de controlador recibe como entrada un tipo booleano, y mientras la condición sea cierta ejecutará el primer flujo de salida. Cuando este flujo haya terminado, se vuelve a procesar la entrada para ver si es necesario ejecutar una nueva iteración. Procesar la entrada significa volver a ejecutar ese nodo (y todos sus nodos entrada) por si el valor hubiera cambiado. Cuando el resultado devuelto sea falso el flujo continuará por el segundo flujo de salida.
- *Repeat*: Este tipo de controlador recibe como entrada un número, y ejecutará esa cantidad de veces el primer flujo de salida. El valor de la entrada no se procesa en cada iteración por lo que solo se tendrá en cuenta el valor inicial. Cuando se haya completado el número de vueltas el flujo continúa por el segundo flujo de salida.

Para terminar, uno de los cambios imprescindibles que faltaba por añadir es la adición de eventos. Actualmente, el scripting le da el flujo al primer nodo que encuentra, pero esto no es ni lo deseable ni lo óptimo. Entonces diseñamos un nuevo tipo de nodo, que es el nodo de eventos. Este nodo no tiene valor de entrada ni de salida y solo cuenta con una conexión de flujo de salida. Los valores de eventos que haya dependerán de las necesidades del motor, pero los dos esenciales serían el evento de inicio y el evento de refresco, los cuales serán ejecutados desde el motor al inicio del programa y en cada fotograma siguiente respectivamente.

Este fue el desarrollo principal del lenguaje de scripting, aunque más adelante se fueron añadiendo mejoras y modificaciones a la idea inicial. El resultado final cuenta con los siguientes elementos y características:

- *Nodo función*: Ejecuta una función dentro del motor. Puede recibir un número indeterminado de nodos como entrada y puede tener un valor de salida, puede recibir y pasar el flujo y puede tener nombre. Cuando un nodo tiene nombre puede ser ejecutado desde otro script (como si se tratase de un método en un lenguaje de programación convencional).
- *Nodo valor constante*: Tiene un tipo determinado y tiene un valor asociado a ese tipo. No tiene valor de entrada, pero siempre tiene uno de salida. No puede interactuar con el flujo del script. Puede ser serializado poniéndole un nombre. De esta forma, el valor puede modificarse desde fuera. Esto es útil pues de esta forma se puede modificar el valor del nodo desde fuera (por ejemplo, el editor) evitando tener que repetir el script entero para modificar únicamente un valor.
- *Nodo evento*: Representa cada evento del motor, no tiene nodos de entrada ni valor de salida. Es el elemento que empieza el flujo en el script, pero luego no puede recibir el flujo.

- *Nodo bifurcación*: Puede modificar el flujo del script, seleccionando la ruta en la que continúa el flujo o repitiendo en bucle partes del script. Recibe un único nodo de entrada, pero no tiene nodo de salida. Hay tres tipos de nodos de bifurcación, permitiendo bifurcar el flujo, repetir una sección del script en bucle determinado por una condición o repetir una sección un número determinado de veces.
- *Nodo comentario*: Este nodo no tiene efecto en la ejecución del script, pero es para proporcionar una mayor claridad al usuario permitiéndole escribir aclaraciones sobre los nodos.
- *Nodo enumerado*: Este nodo es un caso especial del nodo de valor. Permite seleccionar un tipo dentro de una lista de enumerados, y luego seleccionar un valor dentro de ese enumerado. La función de este nodo también es para proporcionar una mayor claridad al usuario y no tener que memorizar la asociación de algunos valores a un número.

El lenguaje

Este lenguaje no es 'Turing' completo, principalmente porque está pensado para un entorno de videojuegos más que propósito general, por lo que el manejo de memoria y recursos es muy limitado. Pero obviando eso, otro motivo importante por el que el lenguaje no tiene completitud 'Turing' es porque no tiene una de las características fundamentales para ello, la recursión. Esto no es por error ya que este comportamiento es el esperado por dos motivos, el primero es que no creemos que este lenguaje sea el idóneo para usar reflexión, y permitir enlazar nodos de esa forma puede llegar a ser bastante confuso. El segundo motivo es que gracias a prohibir la recursión hay varias partes del lenguaje que pueden ser optimizadas.

Otras cosas adicionales que permite el lenguaje con ayuda del motor es poder crear y modificar variables, arrays, crear e invocar eventos globales y operaciones básicas para manipular los distintos atributos, entre otros.

6.2. Implementación del scripting en el editor: edición de nodos

El scripting ocurre en su propia ventana del editor. Esta ventana es especial pues esta ocupa el tamaño completo de la ventana. El editor cuenta con una máquina de estados interna para saber cuándo se está en estado de edición de nodos. Cuando el editor entra en estado de scripting, las demás ventanas dejan de poder actualizarse y renderizarse hasta que se cierre el editor de nodos. Esta ventana será referida como ventana de scripting o de editor de nodos. La lógica de esta ventana está manejada en parte por el editor que es quien se encarga de mantener los valores de la máquina de estados,

y por otro lado la clase `ScriptCreation` que hereda de `Window`. Esta clase se encarga de representar la ventana de scripting, así como la información del script que se quiere editar. El punto de entrada de esta clase es el mismo que el de otras ventanas del editor, el método `Behaviour` heredado de las ventanas.

Inicialización

Cuando se decide cambiar el estado del editor del estado por defecto al estado de edición de nodos no se cambia exclusivamente el valor de la máquina de estados, también se modifica el valor del nombre del script que se está editando (representado mediante un string), dependiendo de la forma en la que se haya abierto el editor nodos. Esto se debe a que el editor cuenta con varias formas de abrir el editor de nodos, pero se pueden dividir en dos categorías: la creación de un script nuevo y la edición de un script existente. En el momento en el que la máquina de estados cambia de valor al estado de edición de nodos se llamará al método público `Load()`, donde en el caso de que la cadena proporcionada para el nombre no esté vacía, se cargará de memoria el valor de ese script (ver apartado Serialización y deserialización). En el caso de que el nombre esté vacío o que la lectura del fichero falle, se procederá a crear un nuevo fichero de scripting. Un fichero recién creado de scripting contará con dos nodos representando el evento de inicialización y el evento de actualización en cada fotograma (`Start()` y `Update()`).

Una vez hecho este proceso el editor de nodos está listo para utilizarse.

Behaviour

Esta función, heredada de la clase `Window`, es la encargada de brindar la funcionalidad principal a la ventana de scripting. Esta función es llamada en cada fotograma y se encarga de una gran funcionalidad de la ventana, en este orden:

1. Garantizar que la ventana se encuentra ocupando la completitud de la ventana de SDL .
2. Acceder a los datos de la paleta de colores para cambiar los valores y colores correspondientes (pe. Grosor y color de las líneas, color del fondo, tamaño de las muescas de arrastre, entre otros)
3. Manejar el desplazamiento de la **cámara**. La ventana cuenta con una posición ficticia, la cual es usada para saber dónde posicionar los distintos nodos y elementos de la ventana. De este modo, se simula que el usuario tiene la opción de desplazarse a través de un espacio infinito. Se puede controlar la posición de dos formas, siendo la más común moviendo el ratón mientras se mantiene presionado el botón de la rueda.

La otra forma de mover la cámara es haciendo uso de un interpolado. Esto ocurre cuando el usuario decide situarse sobre un nodo en concreto desde la barra de menú (explicado más adelante). El control del interpolado está dividido en dos métodos: `Lerp()` y `ManageLerp()`. El primero se encarga de inicializar los valores necesarios para llevar a cabo el interpolado, guardando la posición inicial y la posición final del movimiento. Además, se recibe como parámetro un valor adicional, siendo la duración del proceso. Por otro lado, el método `ManageLerp()` es el encargado de calcular y posicionar el valor actual de la cámara en cada fotograma que esté activo el interpolado. Para procesar el interpolado adecuadamente se guarda un valor adicional en la clase `ScriptCreation` encargado de llevar la cuenta del tiempo, incrementando su valor en cada fotograma haciendo uso del valor `DeltaTime` proporcionado por `ImGui`. Una vez se haya superado el tiempo máximo, se desactiva el proceso de interpolado. Para calcular la posición de la cámara en cada iteración, se hace uso de una combinación de un interpolado lineal clásico, junto con un interpolado cúbico. El interpolado cúbico recibe como parámetro exclusivamente el valor `t` (calculado como tiempo actual dividido entre el tiempo máximo) y devuelve el valor `'suavizado'`, de forma que el movimiento de la cámara aparenta tener aceleración al principio y deceleración al llegar al final del recorrido. Mientras el interpolado está en movimiento, el usuario pierde el control de la cámara para evitar comportamientos indeseados. Esta cámara no tiene soporte para alejarse y acercarse, por lo que el tamaño de los elementos será siempre constante.

4. Dibujado de gráficos: Se pinta el color de fondo de la ventana con el color aportado por la paleta de colores, después se pinta por encima una cuadrícula. La cuadrícula está hecha con fines estéticos y para ayudar con la orientación al usuario, pues mover la cámara sin tener ningún punto de referencia puede resultar confuso. Por ello, la cuadrícula también tiene que ajustarse a la posición de la cámara para simular que también se mueve cuando se mueve la cámara. Para implementar la cuadrícula se ha hecho uso del api de `ImGui`, basándose además en su implementación para que su uso se asemeje al uso de otras partes del api de `ImGui`. La clase `Grid` está compuesta por una serie de variables estáticos que representan la información de la cuadrícula, como su posición, grosor o desplazamiento. Para modificar o acceder a estos valores la clase cuenta con otros métodos estáticos `Getters` y `Setters` para manipular el estado de la cuadrícula. Todo esto culmina con el método `Draw`, que dibuja sobre la ventana actual de `ImGui` la cuadrícula con la configuración establecida. Para el dibujado, se hace uso de la clase `DrawList` de `ImGui`, que proporciona la capacidad de pintar sobre la ventana geometría básica. Otra clase que se ha implementado de forma

similar es la clase **Bezier**. Esta clase dibuja una curva bezier cuadrática entre dos puntos dados. Al igual que la **Grid**, se implementa mediante valores estáticos y llamadas a métodos estáticos que cambien sus valores internos. Para calcular la lógica de la línea bezier, se establecen como punto inicial y final los puntos dados,

5. Dibujado de nodos: En este punto, se dibujan y actualizan todos los nodos que contiene el script. Este proceso se explicará en más detalle más adelante. Cabe destacar, que durante la iteración por cada nodo se comprueba si alguno ha sido marcado para ser eliminado. Si eso es así, ese nodo se elimina cuando todos los nodos hayan sido actualizados. Como cada nodo cuenta con un identificador único ascendente y la lista de nodos permanece siempre ordenada (los nuevos nodos se añaden al final), para evitar que el identificador de los nodos crezca de forma innecesaria, cuando un nodo es eliminado se reduce en uno el valor de los identificadores de los nodos siguientes. Por cómo está implementado este sistema, no es posible eliminar dos nodos en la misma iteración, aunque tampoco supone un problema porque no sería posible pulsar varios botones de cierre de nodos simultáneamente.
6. Dibujado del desplegable de nodos: Esta clase auxiliar se encarga de manejar el desplegable que aparece al pulsar el clic derecho. Este desplegable contiene la lista con todas las funciones disponibles por el motor. La clase controla el input, y en caso de haber pulsado muestra el desplegable usando funciones de ImGui. El desplegable muestra información de métodos tanto para los componentes como para los managers. Si se ha hecho clic sobre alguno de los elementos, se creará un nuevo nodo de tipo función asociado a esa función donde el usuario hubiera hecho clic para abrir el desplegable.
7. Dibujado de la barra de menú: En la parte superior de la ventana, aparece funcionalidad adicional necesaria para el editor de scripts. Desde la barra se permite al usuario poder cambiar el nombre del script, crear nodos para cada tipo de variable constante, crear los nodos para manejar la lógica y el control del flujo, crear nodos para los distintos eventos proporcionados por el motor, un botón para guardar el script, un botón para cerrar la ventana de scripting y volver a la ventana principal del editor, una barra de búsqueda para buscar funciones del desplegable de nodos mostrando los resultados en un desplegable, un desplegable con la lista de nodos creados, lo cual comenzará la secuencia de interpolado de la cámara hacia ese nodo en concreto; un botón para crear un nodo comentario y por último, un botón que abre la consola, útil en el caso de que se esté depurando un script y ver los mensajes que se hayan escrito sin tener que cerrar la ventana de scripting. Además, proporciona la lógica para mostrar una ventana emergente en caso de

que se quiera cerrar el editor de nodos, pero haya cambios sin guardar.

Serialización y deserialización

Para la serialización y deserialización se hace uso de la biblioteca de procesamiento de JSON de `ñlohmann`.

- *Serialización*: Ocurre en el método `Save()` cuando el usuario pulsa el botón de guardar un script. Para serializar se distingue cada nodo por tipos, agrupando cada tipo en un array diferente. Cada tipo de nodo tiene funcionalidad para generar un fichero JSON con su información, simplificando el proceso para la ventana. Además de guardar la información necesaria para el correcto funcionamiento del motor, se guarda información como la posición y tamaño del nodo para que todo permanezca igual tras volver a abrir el script para su edición. Una particularidad de la serialización de los nodos ocurre con los nodos de tipo valor constante, ya que estos pueden ser serializados. En este caso, ese nodo es también incluido en otro vector con todos los valores serializados. Cuando todos los nodos han sido serializados, todos los vectores se añaden a la raíz de JSON para ser guardado en disco. Por defecto, el editor guarda todos los scripts en una carpeta `'Scripts'` dentro de `assets`, aunque no es necesario que un script esté en esa carpeta para su lectura por el motor o el editor. Una vez terminada la serialización, se actualizan los scripts del `ComponentManager` del editor, para poder utilizar el nuevo script en las escenas.
- *Deserialización*: Ocurre en el método `Load`, el cual se llama cuando el editor pasa al estado de edición de nodos. Se comprueba la validez del nombre y del fichero a leer, y en caso de que el nombre esté vacío o que la lectura falle se creará un script vacío con ese nombre. Para de serializar un script, se va leyendo cada nodo del fichero, y creando nodo del tipo correspondiente según los datos dentro del propio JSON. Una vez todos los nodos han sido creados, se crean las uniones entre ellos. Dichas uniones también se extraen del propio JSON. Las uniones incluyen las conexiones de entrada y salida de valores, y las conexiones de flujo.

Nodos

Para entender el funcionamiento de los nodos en el editor primero es conveniente entender la jerarquía de clases que los representa.

La clase principal sería la clase `ScriptNode`. Esta clase contiene la lógica para el funcionamiento de la ventana de cada nodo, conociendo su tamaño y posición, distintos controles de la ventana como la posibilidad de cambiar su

tamaño y un botón para poder cerrarlo. Además del funcionamiento general, también se encarga de manejar el valor de salida de los nodos, pero como no todos los nodos tienen valor de salida cuenta también con un flag para ignorar este valor y no mostrar nada al usuario. De esta clase heredan otras cuatro clases usadas para representar distintos tipos de nodos: **ScriptFork**, **ScriptEvent**, **ScriptFunction**, **ScriptInput** y **ScriptComment**.

Como en el editor los scripts no se llegan a ejecutar, la lógica para implementar cada tipo de nodo es bastante sencilla, pues cada nodo solo tiene que almacenar los valores necesarios para cada tipo de nodo.

- *ScriptFork* almacena su valor de entrada y sus dos nodos de salida de flujo.
- *ScriptEvent* almacena el nombre del evento y su nodo de salida de flujo.
- *ScriptFunction* almacena un vector para sus nodos de entrada y el nombre de la función. En la constructora recibe como parámetro un puntero a una clase **Function** con toda la información asociada al método, como el nombre de la función, el número de nodos de entrada que necesita, el tipo y nombre asociado a cada nodo de entrada y tipo del valor de salida. Además de eso, también almacena el nodo de salida de flujo.
- *ScriptInput* almacena tanto el tipo del nodo como su valor. En la constructora recibe el tipo asociado al nodo y éste no puede ser cambiado. Dependiendo del tipo del nodo, al usuario le aparecerán distintos inputs para almacenar los distintos tipos de entrada. Tiene además un botón para serializar el valor, de esta forma, el valor del script se puede modificar desde el editor. **ScriptInput** no tiene un nodo de salida de flujo ya que no puede recibirlo.
- *ScriptComment* almacena un mensaje de texto editable por el usuario. Este nodo es completamente auxiliar y su única función es que el usuario pueda ir dejando mensajes por los nodos para poder documentarlo y dejar aclaraciones.
- *ScriptFlow* sirve para poder manejar la salida de flujo. Esta clase se asocia a un nodo, y tiene funcionalidad para controlar el flujo del script. Esto se hace uniendo las muescas del flujo (representadas mediante rombos), arrastrando la salida del flujo hasta la ventana del nodo que se quiere que se ejecute después. Si la ventana soporta el manejo de flujo aparecerá coloreada cuando se arrastre el ratón por encima mientras se arrastra un flujo de salida. Un nodo solo puede tener como máximo una entrada y una salida, con la excepción de **ScriptFunction** que puede tener hasta dos salidas. Para eliminar una conexión de flujo basta con

hacer doble clic sobre el inicio o final del nodo. La muesca de entrada de un nodo por defecto no se dibuja, hasta que se cree una conexión con otro nodo.



La unión entre dos nodos vendrá representada mediante una línea hecha usando la clase **Bezier**.

La conexión de entrada y salida de un nodo se comporta de forma similar al flujo. Para crear una conexión se debe arrastrar sobre la muesca de salida de un nodo (representado mediante un triángulo apuntando hacia la derecha) y se soltará sobre el nodo de entrada (representado mediante un triángulo apuntando hacia la izquierda). Si la conexión ha resultado exitosa, ambos triángulos pasarán a estar coloreados. Un nodo de salida puede tener varias conexiones, mientras que un nodo de entrada puede tener solo una, por ello, si se conecta un nodo de salida diferente a un nodo de entrada, se eliminará la conexión anterior y se creará una nueva con los nuevos nodos. Una condición para que se pueda crear la conexión es que tanto la salida como la entrada deben de ser del mismo tipo. Se puede consultar el tipo de una entrada posando el ratón encima del texto con el nombre de la entrada, y se puede consultar el tipo de la salida posando el ratón sobre el triángulo de salida.

Para eliminar una conexión se puede hacer doble clic sobre el nodo de entrada. Las conexiones se representan haciendo uso de una línea hecha usando la clase **Bezier**. Cuando se elimina un nodo, haciendo uso de llamadas virtuales, se notifica a cada clase de la inminente destrucción. Esto es importante, por las conexiones se almacenan en ambos sentidos, es decir, el nodo de entrada guarda un puntero al nodo de salida y viceversa. Por ello, cuando se elimina un nodo, hay que notificar a todas las conexiones para que los punteros se liberen correctamente y pasen a ser *nullptr*. Esto es cierto tanto para las conexiones de entrada y salida como las conexiones de flujo.

6.3. Implementación del scripting en el motor

El punto de conexión entre el motor y el scripting es a partir del componente **Script**. La función de esta clase es principalmente recibir las llamadas a los distintos eventos desde el motor y se las redirigirlas a los eventos del propio script. Además, también funciona como clase para almacenar distintos valores que se pueden utilizar en los scripts, aunque de esto se entrará en profundidad más adelante. La clase contiene una estructura de datos, en los que hay punteros a los distintos nodos de eventos del script. Si un script no tiene un evento en concreto su valor será *nullptr* y ese evento en cuestión se omitirá.

La clase `script` no contiene el script y sus nodos como tal, pues esto podría suponer una gran cantidad de datos repetidos en el momento en el que varias entidades compartieran el mismo script. Para ello existe la clase **ScriptManager**. Esta clase es un **Singleton** que carga y almacena todos los scripts que se vayan utilizando a lo largo de la ejecución del juego. De esta forma, cuando varias entidades usen el mismo script no tendrá que leerse el fichero nuevamente en memoria y se evitan tener duplicados. Además de almacenar los scripts, también se encarga de manejar la lógica básica de los nodos. Pero antes de seguir explicando el **ScriptManager**, expliquemos el funcionamiento de los nodos.

La clase `nodo` es una clase abstracta que contiene el identificador único del script, métodos virtuales para proporcionar funcionalidad, un identificador de iteración y el método **Cycle()**. Este método, no virtual, maneja el flujo de los nodos y lo que hace es llamar al método virtual **Operate()**. **Operate()** proporciona un puntero al siguiente nodo a ejecutar, y si este puntero es válido (distinto de *nullptr*) se llama al método **Cycle()** del nuevo nodo. Por lo tanto, este método se ejecutará de manera recursiva hasta que se hayan ejecutado todos los nodos del flujo. Este proceso, en el que se llama por primera vez a **Cycle()** desde el script manager hasta que termina su ejecución en todos los nodos es lo que llamamos iteración.

Al hacer la llamada desde el **ScriptManager** hay que proporcionar un entero, que contiene el índice de la iteración actual, que se guarda dentro de la clase **Node**. Esto es importante pues optimiza enormemente la velocidad del lenguaje ya que, si la salida de una función es usada por varios nodos, no queremos tener que recalcular tanto los valores de entrada de forma recursiva y la función de nuevo, por lo que al comienzo del método se comprueba si ambos índices, el guardado por el nodo y el recibido como entrada al método, son distintos. En caso de ser distintos sabemos que el nodo no ha sido ejecutado esta iteración y procedemos a la ejecución normal, pero en caso de ser iguales, este nodo ya ha sido procesado y procesarlo de nuevo sería innecesario.

No hay que confundir una iteración con un fotograma dentro del juego, ya que si varios scripts se ejecutan en el mismo fotograma tendrán identificadores de iteración diferentes.

De la clase **Node** hereda la clase **OutputNode**. Esta clase, también abstracta está pensada para los distintos nodos que tienen un valor de salida, siendo los nodos de función, y los nodos de valor constante. Por ello, esta clase proporciona esencialmente un atributo nuevo de tipo **Variable**, usado para almacenar el valor devuelto por el nodo.

Ahora es un buen momento para explicar la clase **Variable**. Esta clase está pensada para poder tomar cualquier valor utilizado desde el scripting,

independientemente del tipo. Para ello se hace uso de un tipo enumerado indicando el valor de la clase, y una unión con cada tipo disponible para el scripting. La motivación para usar esta clase es simplicidad, ya que por ejemplo de esta forma podemos hacer que las funciones del scripting tengan siempre la misma entrada y salida (de entrada, un vector de variables y de salida una variable) y poder almacenar todo en un único mapa, aunque esto se explicará en más detalle más adelante. Volviendo a los nodos, de la clase **OutputNode** heredan otras dos clases: **ConstNode** y **Function**.

1. La primera es la encargada de almacenar los valores constantes del scripting, siendo por ejemplo números, vectores o cadenas de texto. El nombre de la clase viene ya que para el usuario el valor dado al nodo es inmutable. Podemos distinguir dos casos de uso para esta clase, el primero siendo únicamente un valor dado desde el script, en cuyo caso el método **Operate** es bastante simple ya que únicamente se encarga de hacer que el siguiente nodo sea *nullptr*. El otro caso de uso es cuando el valor ha sido serializado, por esto nos referimos a que el usuario tiene la posibilidad de modificar el valor del nodo, por ejemplo, desde el editor, pudiendo de esta manera reutilizar el script con valores diferentes. Para este caso de uso la clase **ConstNode** cuenta dos atributos adicionales, uno de tipo *'string'* que representa el nombre de la variable, y otro de tipo **Variable** que representa el valor original que tenía el nodo antes de la serialización. La diferencia entre ambos casos de uso reside esencialmente en el método **Operate**, donde comprobamos si la variable está serializada (podemos saber esto si tiene un nombre asociado, es decir, distinto de nulo), en cuyo caso accederíamos a uno de los valores que mencionamos anteriormente que almacenaba la clase **Script**, siendo el valor de la variable serializada. Si el valor asociado a ese nombre que almacena la clase es válido, entonces ese será el nuevo valor devuelto. En caso de no ser válido, porque no se haya asignado un valor a todas las entidades que usen el script, se usará el valor por defecto que es el motivo por el que se guardaba.
2. La otra clase que hereda de **OutputNode** es **Function**. Esta clase es la encargada de ejecutar las funciones del editor. Contiene un puntero al siguiente nodo, el nombre del método y un vector de **OutputNode** con todos los inputs de la función. El método **Operate()** se encarga de generar el vector para la entrada de la función. Esto lo hace iterando por cada nodo de entrada, llamando a su método **Cycle()** y luego obteniendo el valor guardado que se guarde en *output*. Para la llamada al método se hace a través del **ScriptManager**, el cual contiene un mapa con todas las funciones del motor, accediendo a la función con un string con su nombre. Este mapa de funciones se genera haciendo uso del **FunctionManager**, clase la cual tiene un método para generar por

completo el mapa, añadiendo uno a uno cada función del motor con su nombre asociado. Estas funciones, también incluidas en el mismo fichero que **FunctionManager** se encargan de llamar a cada método del motor, recibiendo como entrada un vector de tipo **Variable** y devolviendo un valor de tipo **Variable** (Variable tiene soporte para devolver null en caso de que la función no devuelva nada).

El último tipo de nodo es la clase **Fork**. Esta clase representa la lógica condicional y bucles que se pueden implementar en el scripting, pudiendo ser de tres tipos: un condicional *if*, un bucle *while*, y un tipo de bucle que repite el contenido un número determinado de veces al que llamaremos *for* aunque realmente no lo sea. Este nodo contiene un puntero a otros dos nodos, llamados A y B, que tienen uso diferente en caso de ser un condicional o un bucle. Si es un condicional, si el valor de entrada es A, el nodo que se marque como next será A, en caso contrario será B. Si se trata de un bucle A es el contenido inicial del bucle, y el nodo B es el nodo siguiente a la terminación del bucle. Una peculiaridad de este nodo es que es el único que modifica el índice de iteración. Esto es así pues queremos que los nodos de dentro del bucle se puedan ejecutar varias veces dentro de la misma iteración, por lo que se incrementa el índice de iteración en cada vuelta del bucle.

Capítulo 7

Contribuciones

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

7.1. Pablo Fernández Álvarez

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

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

funciones de dibujado con SDL.

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

7.2. 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 pospuse un poco el desarrollo del scripting y empecé 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 relevantes del motor pasaba a estar serializada en fichero en disco, siendo la idea que el motor pueda leer los valores desde ese fichero.

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

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

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

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

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

7.3. Iván Sánchez Míguez

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

Una vez seleccionada esta biblioteca, procedí a crear el proyecto en Visual Studio 2022. Inicié con la creación de un programa simple inicial para comprender cómo se inicializa y funciona ImGui, el cual incluía un proyecto con numerosos ejemplos. Mi enfoque inicial se centró en el diseño de las ventanas principales, como la barra de menú, la jerarquía, la escena, los componentes y el explorador de archivos. Esto se hizo de manera rápida y provisional con el único propósito de familiarizarme rápidamente con ImGui. Durante este proceso, descubrí la rama `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 padre y sus hijos, lo que debía

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

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

Capítulo 8

Pruebas con usuarios

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 puertas 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 funcionalidad:

- 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.

Capítulo 10

Conclusions

The aim of this final degree project was to develop a 2D game engine for non-programmers. We have successfully achieved the proposed goal. Our engine opens the doors to developers with little programming experience while providing enough functionality to create competent 2D video games.

In practical applications, our engine can be used in the indie game development world or even for educational purposes.

From a technical perspective, we have reached the following conclusions:

- With the current implementation, ImGui windows cannot be moved outside the main SDL window, which can be inconvenient in some situations, such as when implementing a script where it's essential to view the scene or some editor parameters to make decisions.
- We had to implement reflection in C++. It would have been more convenient to choose a language with reflection, which would have provided us with greater flexibility.

As for future work, we are considering the following functionalities:

- Enabling the game to be launched within the editor itself rather than in a separate window.
- Modifying the current editor implementation to allow the movement of ImGui windows outside the main SDL window.
- Adding an animation and rendering system to reduce the dependence on external tools.
- Abstracting the user from file paths and directories by converting files into engine objects.
- To delve deeper into scripting, adding more complex functionality that allows the user greater expressiveness, for example, by adding editable

arrays from the editor, class creation, recursion, timers, coroutines, and debugging of running nodes.

Parte I

Apéndices

Ocornut Catto Community ifo Gregory (2018) 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 cerebro 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 ingui. Disponible en <https://github.com/ocornut/ingui>.
- 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*

