Editor y motor de juegos 2D para no programadores

TRABAJO DE FIN DE GRADO

Pablo Fernández Álvarez
Yojhan García Peña
Iván Sánchez Míguez

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

Septiembre 2023

Documento maquetado con TEXIS v.1.0+.

Editor y motor de juegos 2D para no programadores

Memoria que se presenta para el Trabajo de Fin de Grado

Pablo Fernández Álvarez, 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 centrase en mayor medida en el desarrollo del videojuego. Algunos ejemplos de funcionalidades que aportan los motores son: rendereizado 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. tools for the creation of video games. These tools prevent the developer from developer to implement a large amount of functionality in order to focus more on the 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. especially for those who are not familiar with the engine in particular or with game development in general. game development in general. However, once developers become familiar with the Once developers become familiar with the tools, however, they can significantly speed up the process of game the game creation process and improve productivity.

This is a great advantage to experienced developers, but engines like Unity or UnrealEngine engines such as Unity or UnrealEngine can harbor too much complexity for non-programmers, even for for people with no programming experience, even if they are aiming at simple 2D games. simple 2D games. A very useful tool to solve this problem is visual programming. problem is visual programming. This type of programming allows users to users to create logic by manipulating graphical elements instead of specifying them exclusively textually. rather than specifying them exclusively in textual form. 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 visual programming based on nodes, the execution of the game in the editor and the creation of nal the creation of final executables of the game for distribution.

Índice

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 comodo 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, queson fragmentos de código que definen comportamientos específicos para las entidades, debe ser cómoda e intuible ya que es la tarea que más más tiempo va a ocupar al desarrollaor.

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 la siguientes características:

- `Autosuficiencia`: Esto siginifca 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 herrmienta externa durante el desarrollo pero si 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 accciones 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 cononcimientos 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.

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.

- El motor consta de varios proyectos de Visual Studio donde se dividen las partes fundamentales del mismo:

- `Proyecto principal`: Este proyecto implementa el bucle principal del motor y de generar el ejecutable de juego.
- `Proyecto de físicas`: Implementa un manager con el que acceder a la configuración del mundo físico.
- `Proyecto de gráficos`: Implementa un manager con el que acceder a la configuración de renderizado de SDL, como la ventana de juego o la cámara.
- `Proyecto de recursos`: Implementa un manager para el guardado de recursos, como imágenes o fuentes de texto.
- `Proyecto de sonido`: Implementa un manager para cargar, reproducir o detener efectos de sonido o música.
- `Proyecto de utilidades`: Implementa varias clases genéricas útiles para el resto de proyectos.
- `Proyecto de input`: Implementa un manager para manejar el input del usuario, tanto de teclado y ratón como de mando, con soporte para Dualshock 4.
- `Proyecto de Entity-Component-System`: Implementa el ECS y contiene todos los componentes proporcionados por el motor.

- `Proyecto de Consola`: Implementa funcionalidad para imprimir información por consola. Útil para el desarrollo y la consola de debug del editor.

- `Proyecto de Scripting`: Implementa funcionalidad leer y crear clases en C++. Explicada en detalle posteriormente.

- El Editor consiste en 2 proyectos de Visual Studio:

- `Proyecto de componentes`: Este proyecto tiene la responsabilidad de gestionar la adquisición y procesamiento de scripts y componentes procedentes del motor subyacente. Estos elementos son preparados para su utilización posterior en el editor.

- `Proyecto de ImGUI`: En este proyecto se encuentra toda la funcionalidad relativa al editor, haciendo uso de la biblioteca ImGUI. Aquí se engloba la creación y manejo de ventanas, cuadros de diálogo emergentes, menús desplegables, renderización de la escena en edición, así como la gestión de archivos, entre otras características.

- `Proyecto main`: Este proyecto ejecuta el bucle principal del editor y genera el ejecutable.

TODO: Scripting...

- La integración entre el editor y el motor se materializa mediante un flujo de intercambio de datos. Inicialmente, el motor lleva a cabo la serialización de los componentes disponibles en un formato JSON. Posteriormente, el editor se encarga de leer y procesar estos componentes serializados, permitiendo su asignación a las diferentes entidades en desarrollo. Una vez que el proceso de desarrollo del videojuego ha concluido, el editor toma la iniciativa de serializar la escena completa. Esto abarca las entidades presentes en la escena, junto con sus respectivos componentes y scripts asociados. Con la escena debidamente serializada, el editor ejecuta el archivo ejecutable (.exe) del motor. En este punto, es el motor el que asume la responsabilidad de interpretar y cargar las escenas previamente serializadas por el editor.

Este enfoque de intercambio de datos entre el editor y el motor permite una sincronización efectiva de la información esencial para la construcción y ejecución del videojuego, al tiempo que mantiene una clara separación de las tareas y roles entre las dos entidades.

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.

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 (Immediate Mode Graphical User Interface) 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 parts: the engine development, the editor development, and the development of the node-base visual scripting system.

- The engine consists of several Visual Studio projects where its fundamental parts are divided:

- `Main Project`: This project implements the main loop of the engine and generates the game executable.
- `Physics Project`: Implements a manager to access the physical world's configuration.
- `Graphics Project`: Implements a manager to access SDL rendering configuration, such as the game window or the camera.
- `Resources Project`: Implements a manager to handle resources such as images or text fonts.
- `Sound Project`: Implements a manager to load, play, or stop sound effects or music.
- `Utilities Project`: Implements various generic classes useful for the rest of the projects.
- `Input Project`: Implements a manager to handle user input, including keyboard, mouse, and controller input, with support for Dualshock 4.
- `Entity-Component-System Project`: Implements the ECS and contains all the components provided by the engine.
- `Console Project`: Implements functionality to print information through the console, useful for development and debugging within the editor.
- `Scripting Project`: Implements functionality to read and create C++ classes. Detailed explanation follows later.

- The Editor consists of 2 Visual Studio projects:

- `Componentes project`: This project is responsible for managing the acquisition and processing of scripts and components from the underlying engine.
- `ImGUI project`: In this project you will find all the functionality related to the editor, making use of the ImGUI library. This includes the creation and management of windows, pop-up dialog boxes, dropdown menus, rendering of the editing scene, as well as file management, among other features. as well as file management, among other features.
- `main project`: This project implements the main loop of the editor and generates the executable.

TODO: Scripting...

- The integration between the editor and the engine takes the form of a data exchange flow. Initially, the engine performs the serialization of the available components in a JSON format.components available in a JSON format. Subsequently, the editor is in charge of reading and processing these serialized components, allowing their assignment to the different entities under development. different entities under development. Once the video game development process has been completed, the editor takes the initiative to serialize the entire scene. This encompasses the entities present in the scene, along with their respective components and associated scripts. With the scene duly serialized, the editor runs the executable file (.exe) to the engine. At this point, it is the engine that assumes responsibility for interpreting and loading the scenes previously serialized by the editor.

This approach to data exchange between the editor and the engine allows for effective synchronization of the information essential for the construction and execution of the while maintaining a clear separation of tasks and roles between the two entities.

TODO: Fase de prueba con usuarios...

En el próximo capítulo...

En capítulo *3 Estado del Arte* se explica qué son los motores de videojuegos 2D y cuáles hemos tomado como referencia para desarrollar el nuestro. Asimismo, se expondrán las librerías que hemos decidido emplear en nuestro proyecto y se justificará la elección de cada una. Además, se menciona la diferencia entre el desarrollo de un videojuego a través de scripting y programación. Por último, se habla sobre el papel que juegan los motores en el mercado actual de los videojuegos.

Capítulo 3

Estado del arte

3.1. Motores de videojuegos 2D

Un motor de videojuegos 2D se refiere a un sistema de software integral 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.

Los puntos clave que caracterizan un motor de videojuegos 2D incluyen:

- `Gestión de Gráficos y Renderizado`: El motor maneja la representación visual de los elementos del juego, como personajes, escenarios y objetos, asegurando la colocación precisa y la superposición adecuada de estos componentes en el espacio bidimensional. Proporciona herramientas para el dibujo, la animación y el manejo de capas gráficas.

- `Física y Colisiones`: Los motores 2D permiten simular efectos físicos realistas, como gravedad, movimiento y colisiones entre objetos. Esto permite que los elementos del juego interactúen de manera coherente y creíble, mejorando la experiencia de juego.

- `Entrada del Usuario`: Los motores de videojuegos 2D ofrecen una interfaz para capturar y procesar la entrada del jugador, como pulsaciones de teclas y movimientos del mouse. Estos datos se utilizan para controlar la interacción del jugador con el juego.

- `Lógica del Juego`: Incluye el conjunto de reglas, mecánicas y comportamientos que definen la jugabilidad del videojuego. El motor facilita la

implementación de esta lógica y proporciona estructuras para gestionar eventos y estados del juego.

- `Gestión de Escenas y Niveles`: Los motores 2D permiten la creación y gestión de múltiples escenas o niveles dentro de un juego. Esto posibilita la transición suave entre diferentes partes del juego y la estructuración efectiva de la experiencia del jugador.

- `Sonido y Música`: Los motores de videojuegos 2D permiten la integración de efectos de sonido y música para enriquecer la atmósfera del juego y proporcionar retroalimentación auditiva al jugador.

3.1.1. 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 de Gráficos y Renderizado:** Responsable de la representación visual de los elementos del juego, como personajes y objetos, garantizando su colocación precisa y superposición adecuada en el espacio bidimensional. Proporciona herramientas para dibujar, animar y gestionar capas gráficas.

- **Motor de Física:** Simula efectos físicos realistas, como gravedad, movimiento y colisiones entre objetos, para que los elementos del juego interactúen de manera coherente y creíble, mejorando la experiencia del jugador.

- **Motor de Entrada del Usuario:** Captura y procesa la entrada del jugador, como pulsaciones de teclas y movimientos del ratón, permitiendo controlar la interacción del jugador con el juego.

- **Motor de Lógica del Juego:** Incluye reglas, mecánicas y comportamientos que definen la jugabilidad del videojuego. Facilita la implementación de esta lógica y proporciona estructuras para gestionar eventos y estados del juego.

- **Gestión de Escenas y Niveles:** Permite la creación y gestión de múltiples escenas o niveles dentro del juego, facilitando la transición entre partes del juego y la estructuración de la experiencia del jugador.

- **Motor de Sonido y Música:** Integra efectos de sonido y música para enriquecer la atmósfera del juego y proporcionar retroalimentación auditiva al jugador, mejorando la inmersión.

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. Puedes copiar y pegar este código LaTeX en tu documento según sea necesario.

3.2. Editores en motores de videojuegos

Un editor de videojuegos es una herramienta de software que facilita la creación, modificación y organización de diversos elementos que componen 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 y efectiva. 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 y jugables.

- `Creación de Escenarios y Niveles`: Los editores de videojuegos ofrecen herramientas para diseñar y crear los entornos jugables en los que se desenvuelven los juegos. Esto incluye la disposición de elementos, la configuración de obstáculos y la definición de rutas.
- `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.
- `Edición de Eventos y Secuencias`: Los editores posibilitan la creación y edición de eventos específicos del juego, así como secuencias de
  eventos que pueden desencadenar acciones en el juego en respuesta a las acciones del jugador.
- `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.

A pesar de no abarcar todas las características de forma exhaustiva, nuestro editor está diseñado para proporcionar un conjunto de herramientas eficientes y efectivas para el desarrollo de juegos. Por ejemplo, nuestro editor ofrece funcionalidades como la creación de escenarios y niveles mediante escenas, la gestión de assets a través del explorador de archivos, la personalización de la interfaz de usuario con overlays, pruebas en modo de depuración para facilitar la detección de errores, exportación en modo release para la distribución final y un sistema de scripting basado en nodos que permite asignar comportamientos y lógica a las entidades del juego.

3.3. 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.

3.3.1. El motor

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

3.3.2. El editor

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, como niveles, personajes y escenarios, entre otros elementos. 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.

3.3.3. El flujo de trabajo en el desarrollo de videojuegos

El flujo de trabajo en el desarrollo de videojuegos sigue típicamente un patrón colaborativo entre el editor y el motor. Los diseñadores y desarrolladores utilizan el editor para crear y ajustar los elementos del juego, especificando cómo se verán y se comportarán en el mundo virtual. Posteriormente, el motor interpreta estos datos y los utiliza para dar vida al juego en tiempo real. 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.

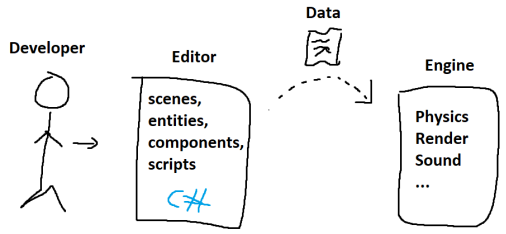


Figura 3.1: Difference between editor and engine

3.4. Importancia de los motores y editores de videojuegos

Los motores y editores de desarrollo desempeñan un papel fundamental en la industria actual al proporcionar las herramientas esenciales que permiten dar vida a la creatividad y la innovación en el mundo digital. Estas plataformas no solo simplifican el proceso de creación, reduciendo los obstáculos técnicos para los desarrolladores, sino que también fomentan la experimentación y el descubrimiento de nuevas formas de contar historias, entretener y educar. Además, promueven la colaboración entre equipos multidisciplinarios al ofrecer un entorno unificado para diseñadores, artistas, programadores y otros profesionales. Los motores y editores también impulsan la producción más eficiente, lo que es crucial en una industria que requiere un equilibrio entre calidad y tiempo de desarrollo.

3.4.1. 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.

Además, 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.

3.4.2. 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 basado en PhysX que complementa su potente capacidad gráfica.

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 y Scratch. A continuación, se explorarán las diferencias entre ambos enfoques y se describirán los sistemas investigados.

3.5.1. 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.

3.5.2. 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

1. **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.

2. **Nodos Interconectados:** La lógica del juego se construye interconectando estos nodos de manera visual. Los nodos pueden representar eventos, 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.

3. **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.

4. **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.

5. **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.

6. **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.

3.5.3. 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.

Funcionamiento de MonoBehaviour en Unity

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

1. **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.

2. **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.

3. **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.

4. **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.

5. **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.

6. **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.

3.5.4. Sistemas de Scripting Investigados (MOVER A OTRO APARTADO)

En el proceso de investigación y desarrollo, optamos por no basarnos en sistemas de scripting preexistentes, ya que nuestro objetivo era crear una solución propia que se ajustara a nuestras necesidades particulares. Sin embargo, en cuanto a la interfaz visual del scripting, hemos tomado referencia de la conocida herramienta de modelado y animación 3D Blender, ya que consideramos que atractiva. Diseñamos nuestro propio sistema de scripting con el propósito de cumplir con las siguientes características clave:

- `Flexibilidad y comodidad para no desarrolladores`:
  Reconociendo la amplia variedad de habilidades entre los usuarios, hemos eliminado la necesidad de conocimientos en programación. Nuestro
  enfoque visual y basado en nodos significa que no es necesario escribir código. Los usuarios pueden construir interacciones y comportamientos simplemente conectando bloques visuales.

- `Aprendizaje Sencillo`: Valoramos la importancia de una curva de aprendizaje suave. Hemos diseñado nuestro sistema con claridad en mente, asegurándonos de que los conceptos sean fáciles de entender. Esto permite a los usuarios adquirir rápidamente la confianza necesaria para utilizar el sistema y plasmar sus ideas en el juego.
- `Enfoque Visualmente Intuitivo`: Inspirados en los principios de diseño de interfaz de usuario, hemos creado un entorno visualmente atractivo y fácil de navegar. Los usuarios pueden interactuar con elementos gráficos que representan conceptos y funciones de juego, lo que facilita la creación de lógica y la toma de decisiones.

3.6. Librerias utilizadas (MOVER A OTRO APARTADO)

Hemos seleccionado cuidadosamente un conjunto de librerías para respaldar el desarrollo de nuestro sistema de scripting personalizado. Entre estas librerías, ImgUI, SDL y Box2D han sido esenciales debido a sus características y ventajas específicas:

3.6.1. ImGUI

Optamos por ImgUI para potenciar la interfaz de usuario de nuestro editor. Su flexibilidad y capacidad para personalización han sido especialmente valiosas, permitiéndonos crear ventanas, cuadros emergentes y controles altamente adaptables para nuestras necesidades específicas. La documentación detallada y la amplia gama de ejemplos proporcionados han facilitado la implementación de soluciones de interfaz visualmente atractivas y funcionales.

3.6.2. SDL (Simple DirectMedia Layer)

La elección de SDL se ha extendido a ambos componentes, el editor y el motor, debido a su versatilidad y amplias capacidades. Para el editor, SDL ha proporcionado una forma eficiente de gestionar elementos multimedia, como fuentes y renderización, mejorando la calidad visual y la interacción del usuario. En el motor, SDL se ha empleado para lograr la misma calidad visual, gestionar elementos multimedia y para facilitar la interacción con el usuario en la ejecución del juego.

3.6.3. Box2D

Hemos decidido implementar Box2D para el motor del videojuego. Box2D se ha utilizado para gestionar las simulaciones físicas que queríamos lograr en el entorno de juego. Esta librería ha sido especialmente útil para crear

interacciones físicas precisas y realistas entre los elementos del juego. Si bien no se ha aplicado en el editor, su incorporación en el motor ha enriquecido la experiencia de juego al agregar un componente realista de movimiento y colisiones.

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.

4.1.1. 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.

4.1.2. 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.

4.1.3. 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.

4.1.4. Recepción y Envío de Assets Entre Ventanas

ImGui es especialmente útil para recibir y enviar activos (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.

4.1.5. Inicialización

Para inicializar ImGUI con SDL hemos seguido un procedimiento específico. Primero, se debe iniciar SDL, que es la biblioteca Simple DirectMedia Layer 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 superposiciones (*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.

4.2.1. Renderizado de las entidades y cámara virtual

Para renderizar la escena, la cámara virtual contiene una textura de destino (texture target) 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 las superposiciones (overlays) no se manejan de la misma manera en el proceso de renderizado, como se habla en (REFERENCIA).

4.2.2. 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 elemententos 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 la entrada (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

4.4.1. 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.

El '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.

4.4.2. Window Layout y su utilidad

El proceso de renderizado de las ventanas en el editor se beneficia de la clase 'Window Layout'. Esta clase juega un papel importante al administrar diferentes diseños (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 'Window Layout', 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 actualización (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 activos (assets) entre escenas de manera eficiente. Para lograr esto, hacemos uso de las capacidades de arrastrar y soltar (drag-and-drop) 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. Ademas, una entindad 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.

4.7.1. 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()`: `HandleInput()` permite a las entidades responder a las entradas del usuario, como clics de ratón y pulsaciones de teclas.

- `AddComponent()`: `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 automaticamente 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()`: `IsTransform()` se emplea para determinar si una entidad es de tipo transform (con atributos de posición, escala y rotación) 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.

4.8.1. 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.

4.8.2. 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()`: Este método, `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 visualizción, 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.

4.9.1. 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.

4.9.2. Métodos relevantes de la clase `Entidad` para la gestion de prefabs

- `IsPrefab()`: Indica si la entidad es un prefab (id negativo) 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, como vemos en la figura **??**.

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.

4.10.1. 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.

4.10.2. 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.

4.10.3. 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.

4.10.4. 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

4.11.1. 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.

4.11.2. 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 Project Manager que contendrá exclusivamente los recursos específicos de nuestro proyecto de videojuego.

IMAGEN

4.11.3. 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 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 configura-

ciones se encuentren separados de los recursos generados por la construcción del motor, facilitando así la gestión del proyecto.

Capítulo 5

Motor

En este capítulo...

5.1. Motor

El motor esta dividio 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.

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.

5.1.1. Proyecto de 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 funcionalida 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.

- EngineTine: 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 incoveniente y es borrar los Singletons en el orden adecuado si dependen entre ellos.

5.1.2. Proyecto de 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.

5.1.3. Proyecto de 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, parar, y detener sonidos, entre otros. Tiene dos métodos destinados al usuario para el modificar el volumen general y cambiar el número de canales disponibles para la reproducción de efectos de sonidos.

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

5.1.4. Proyecto de 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:

- Down: Una tecla esta siendo pulsada. - Up: Una tecla no esta siendo pulsada. - Pressed: Una tecla acaba de ser pulsada. - 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:

- Referencia al GameController creado por SDL. - Id del GameController creado por SDL. - Nombre del GameController. - Estado de cada uno de los botones del GameController. - Información del movimiento de los triggers del GameController. - Información del movimiento de los joysticks del GameController.

El manager tiene sporte además para conexiones y desconexiones durante la ejecución. Debido a la posibilidad de tener varios mandos conectados el manager diferencia entre métodos con identificador y métodos sin identificador. Los métodos con identificador reciben el identificador del mando del que se quiere consultar el estado y los métodos sin identificador devuelven la información del estado del último mando que registró input. De esa manera, si se quiere desarrollar un singleplayer, el usuario no tendrá que preocuparse por la posibilidad de múltiples mandos teniendo que indicar que identificador tiene su mando.

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.

5.1.5. Proyecto de 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.

5.1.6. Proyecto de 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.

- Pixeles: Es tentador usar pixeles como unidades para los tamaños, posiciones, fuerzas o velocidades pero desafortunadamente, esto llevaría a una simulación ineficiente y posiblemente a un comportamiento extraño. En la propia documentación de Box2D comentan que un objeto de 200 píxeles de longitud sería visto por Box2D como el tamaño de un edificio de 45 pisos.

  Para resolver el problema de los pixeles, el manager declara una variable 'screenToWorldFactorúsada para convertir de pixeles a unidades físicas y viceversa. Por lo tanto, a la hora de crear cuerpos físicos se convierte el tamaño en pixeles deseado por el usuario a unidades físicas utilizando ese factor de escala.

Las clases que tiene este proyecto son las siguientes:

- PhysicsManager: Clase, Singleton, que contiene la funcionalidad necesariara 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 bobyB.categoryBits) == 0 Y (bodyA.categoryBits AND bobyB.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 indice que la representa. Cuando se crea un nuevo cuerpo físico, se calcula su CategoryBits a partir de ese indice.

  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 pdoer 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 pixeles.

5.1.7. Proyecto de 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 una fichero con .ttf como extensión. Tiene también la funcionalidad de crear un texto o un texto ajustado mediante la creación de una textura.

- Texture: Representa una textura y tiene la funcionalidad de crear una a partir de un fichero con una extensión de imagen como .png o .jpg. Tiene métodos para obtener la textura de SDL (SDLTexuture*) y para consultar el ancho y el alto de la misma.

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

5.1.8. Proyecto de 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 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:

- 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.

- 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 tiene 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 éstos métodos reciben un paquete de parámetros variados de categoria 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.

- 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.

Managers

- SceneManager: Encargado de manjear las escenas. Para ello, cuenta con una pila en la que va almacenando las escenas que se crean. La escena que se va actualizar en el juego es la que se encuentra en el top de la pila. Hay 5 operaciones que se pueden realizar:

1.- Operación PUSH: Carga la escena y la añade al top de la pila, 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.

2.- Operación POP: Elimina la escena en el top de la pila y avisa, a la escena por debajo del top, si la hay, que va a empezar a actualizarse.

3.- Operación POPANDPUSH: Realiza una operación POP y posteriormente una operación PUSH.

4.- Operación CLEARANDPUSH: Vacia la pila de escenas y añade una nueva al top de la pila que va a empezar a ejecutarse.

5.- Operación CLEAR: Vacia 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.

- 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.

- 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.

- ReferencesManager: Encargado de manejar una relación entre las entidades y sus identificadores. // TODO Se va a cambiar

Componentes

- Transform: Contiene la información sobre la posición, rotación y escala de la entidad. Además implementa algunos métodos para rotar, escalar y mover la entidad.

- 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.

- PhysicBody: Componente encargado de crear un cuerpo físico de Box2D. Implementa la funcinalidad de sincronizar posición, rotación y escala del Transform de la entidad al cuerpo físico. Contiene la información sobre bas-

tante 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.

- SoundEmitter: Componente encargado de cargar un sonido e implementar métodos para reproducirlo, detenerlo, pausarlo, etc. Como se comentó anteriomente, SDLMixer dispone de un conjunto de canales para reproducir sonidos pero este componente es abstrae la necesidad de canales desde la perspectiva del usuario.

- MusicEmitter: Componente encargado de cargar música e implementar métodos para reproducirla, detenerla, pausarla, rebobinarla, etc.

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

- Animation: Componente encargado de implementar la lógica de reproducción de animaciones a partir de una hoja de Sprites.

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

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

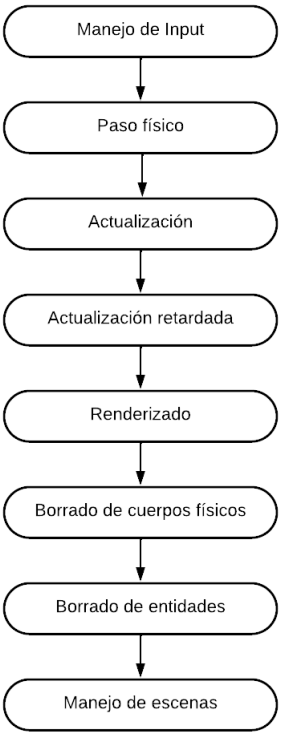
Estos dos útlimos componentes no son fundamentales pero aportan comodidad porque evitar al usuario tener que implementarlos usando el sistema de scripting, lo que puede ser algo avanzado.

//TODO Componentes de Overlay

5.2. Proyecto principal

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

En cuanto al bucle principal, tiene la siguiente estructura:

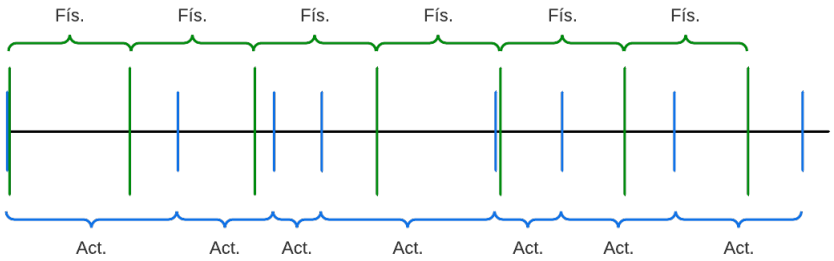


Además, de esos métodos, se realizan cálculos de tiempo para proporcionar al usuario el DeltaTime, tiempo transcurrido desde el inicio de la ejecución del programa o el número de frames/actualizaciones hasta el momento. El DeltaTime es una medida de tiempo, generalmente en milisegundos, que informa sobre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

Algo a comentar es la diferencia entre el Paso físico y la Actualización. La lbrería 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 acabo 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 detras 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

En este capítulo
TODO: Completar...

Capítulo 7

Contribuciones

7.1. Contribuciones de Pablo

Contribuciones de Pablo Fernández Álvarez

7.2. Contribuciones de Yojhan

Contribuciones de Yojhan García Peña

7.3. Contribuciones de Iván

Contribuciones de Iván Sánchez Míguez

Parte I

Apéndices

Bibliografía

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

Miguel de Cervantes Saavedra

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

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

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

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