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.

3.2. 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.

3.3. 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.

3.3.1. 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.

3.3.2. Sistemas de Scripting Investigados

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.

3.4. Librerias utilizadas

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:

3.4.1. 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.

3.4.2. 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.

3.4.3. 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

Implementación

4.1. Editor

4.1.1. 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.

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.

TODO: Mostrar imagenes de las distintas escenas ¿?...

4.1.2. 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.

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.

4.1.3. Jerarquía en las escenas

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

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

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

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

- `AddChild()`: Este método permite agregar una entidad como hijo de la entidad actual, estableciendo así una relación jerárquica entre ambas.

- `RemoveChild()`: Usando `RemoveChild()`, es posible eliminar una entidad de la lista de hijos de la entidad actual, rompiendo la relación jerárquica.
- `SetParent()`: El método `SetParent()` se encarga de establecer la entidad padre de la entidad actual, ajustando sus valores locales de transformación de acuerdo con la jerarquía. También es posible llamar al método con *nullptr* establecer que la entidad carece de padre.

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

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

- `SetLocalRotation()`: `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. **??**

4.1.4. Prefabs y PrefabManager

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

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

Métodos relevantes de `PrefabManager` para la gestion de prefabs

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

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

- `IsPrefab()`: Indica si la entidad es un prefab (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 **??**.

4.1.5. Componentes

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

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

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

Métodos relevantes de la clase `Attribute`

- `GetValue()`: Devuelve el valor actual del atributo.
- `SetValue()`: Establece el valor del atributo.
- `GetType()`: Devuelve el tipo del atributo.
- `GetName()`: Devuelve el nombre del atributo.

Métodos relevantes de la clase `Function`

- `SetReturn()`: Establece tipo de retorno de la función.
- `GetReturn()`: Devuelve el tipo de retorno de la función.
- `AddInput()`: Añade un posible input a la función.
- `GetName()`: Devuelve el nombre de la función.
- `GetComponent()`: Devuelve el nombre del componente al que pertenece la función.

Métodos relevantes de la clase `Component`

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

Métodos relevantes de la clase `Entidad` para la gestión de componentes

- `AddComponent()`: Añade un componente a la entidad.
- `GetComponents()`: Devuelve el mapa de componentes de la entidad.
- `SetComponents()`: Recibe y una lista de componentes para asignarsela a la entidad.

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

4.1.6. Scripts

TODO: Completar...

4.2. 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.

A continuación se entrará en detalle sobre la función y detalles de implementación de cada proyecto y de las librerías asociadas al mismo, si las tiene.

4.2.1. 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.

4.2.2. 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.

4.2.3. 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: 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 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 el sonido, 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 sonido, hacer fade-in de un sonido, hacer fade-out de un sonido, pausar un sonido, detener un sonido (la diferencia con pausar es que si se detiene no se puede renaudar), renaudar un sonido, consultar si un canal esta reproduciendo un sonido, establecer el sonido de un canal en específico, consultar el volumen de un canal, establecer la posicion en el espacio 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 y rebobinar.

4.2.4. Proyecto de input

Capítulo 5

Contribuciones

5.1. Contribuciones de Pablo

Contribuciones de Pablo Fernández Álvarez

5.2. Contribuciones de Yojhan

Contribuciones de Yojhan García Peña

5.3. Contribuciones de Iván

Contribuciones de Iván Sánchez Míguez

Parte I

Apéndices

–¿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