
**Motor y editor de videojuegos en 2D enfocado al
desarrollo de juegos RPG**

**2D video game engine and editor focused on
RPG game development**



**Trabajo de Fin de Grado
Curso 2024–2025**

Autores

Miguel Curros García
Alejandro González Sánchez
Alejandro Massó Martínez

Director

Pedro Pablo Gómez Martín

Grado en Desarrollo de Videojuegos

Facultad de Informática

Universidad Complutense de Madrid

Motor y editor de videojuegos en 2D
enfocado al desarrollo de juegos RPG
2D video game engine and editor focused
on RPG game development

Trabajo de Fin de Grado en Desarrollo de Videojuegos

Autores

Miguel Curros García
Alejandro González Sánchez
Alejandro Massó Martínez

Director

Pedro Pablo Gómez Martín

Convocatoria: *Junio 2025*

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

26 de mayo de 2025

Dedicatoria

Dedicatoria de Miguel. - Miguel

Dedicatoria de Alex. - Alex G.

Dedicatoria de Alex. - Alex M.

Agradecimientos

A Guillermo, por el tiempo empleado en hacer estas plantillas. A Adrián, Enrique y Nacho, por sus comentarios para mejorar lo que hicimos. Y a Narciso, a quien no le ha hecho falta el Anillo Único para coordinarnos a todos.

Resumen

Motor y editor de videojuegos en 2D enfocado al desarrollo de juegos RPG

Pese a que los videojuegos de rol (RPG) son uno de los géneros más demandados actualmente en la industria, hay una gran escasez de motores y editores específicos para este tipo de juegos. Los motores más utilizados tienden a ser muy genéricos y no están centrados específicamente en los RPG.

A este problema, se le añade que los motores y editores para RPG de código abierto disponibles no suelen ser muy amigables con los nuevos usuarios, ya que suelen utilizar mecanismos y estructuras pensados para gente ducha en la materia. Los que sí que tienen una interfaz más sencilla y más práctica suelen estar bloqueados bajo un muro de pago, por lo que muchas veces, diseñadores aficionados se ven gastando una elevada cantidad de dinero en la propia licencia de un *software* que van a utilizar en contadas ocasiones.

Para dar respuesta a estos problemas, en este trabajo se presenta el desarrollo de un motor y editor de videojuegos 2D orientado específicamente a los RPG. El motor cuenta con soporte multiplataforma en Windows, MacOS, Linux y Android, mientras que el editor está disponible en Windows, MacOS y Linux. Su objetivo es facilitar la creación de este tipo de juegos a usuarios sin experiencia en el desarrollo o la programación, manteniendo también algunos elementos más complejos para que usuarios más experimentados puedan generar juegos más completos, todo ello utilizando herramientas de libre acceso.

Palabras clave

Videojuegos de Rol, Desarrollo de Videojuegos, Editor de Videojuegos, Herramienta de Desarrollo, Motor de Videojuegos.

Abstract

2D video game engine and editor focused on RPG game development

Although role-playing video games (RPGs) are among the most demanded genres in today's industry, specific engines and editors designed for these type of games are scarce. The most used ones tend to be too generic and are not focused on RPGs specifically.

In addition to this, the available open source RPG engines and editors are often not very user-friendly for newcomers, as they use mechanisms and structures designed for users with advanced knowledge. Those that do offer a more straightforward and practical interface are often locked behind a paywall, so amateur designers find themselves spending a large amount of money on the license for a software they will rarely use.

To address these problems, this project presents the development of a 2D-video game engine and editor specifically oriented towards RPGs. The engine offers cross-platform support for Windows, MacOS, Linux, and Android, while the editor is available for Windows, MacOS, and Linux. Its goal is to ease the creation of this type of games for users without experience in development or programming, while maintaining more advanced features so that experienced users can generate more complete games, all using open-source and freely accessible tools.

Keywords

Role-playing Video Games, Video Game Development, Video Game Editor, Development Tool, Video Game Engine.

Índice

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Plan de trabajo	2
1.4. Herramientas y Metodología	4
1.5. Estructura de la memoria	4
 Introduction	 7
2. Juegos de rol	13
2.1. ¿Qué es un juego de rol?	13
2.1.1. Tipos de juegos de rol	14
2.1.1.1. Juegos de rol de mesa o tablero	14
2.1.1.2. Juegos de rol en vivo	15
2.1.1.3. Juegos de rol por foro o texto	16
2.1.1.4. Juegos de rol digitales	16
2.1.1.5. Comparativa	16
2.2. Historia de los juegos de rol	16
2.3. Análisis de los principales juegos de rol	18
2.3.1. <i>Dungeons & Dragons</i>	18
2.3.2. <i>Pathfinder</i>	19
2.3.3. <i>Call of Cthulhu</i>	20
2.3.4. <i>Warhammer Fantasy Roleplay</i>	20
 3. Videojuegos de rol	 23
3.1. ¿Qué es un videojuego?	23
3.1.1. Historia de los videojuegos de rol	26
3.2. Sobre el desarrollo de videojuegos	29
3.2.1. Roles en el desarrollo de videojuegos	29
3.2.2. Motor de videojuegos	30
3.2.2.1. Componentes de un motor de videojuegos	31
3.2.2.2. Patrones ECS y EC	34
3.2.2.3. Bucle de juego	34
3.2.2.4. Separación entre motor y <i>gameplay</i>	35

3.2.2.5. Programación dirigida por datos (DDP) en videojuegos	36
3.2.2.6. Programación multiplataforma en videojuegos	37
3.2.3. Editor de videojuegos	39
3.2.3.1. Editores específicos de videojuegos para desarrollo de RPG	41
4. Planteamiento del proyecto	43
4.1. Objetivos principales de RPGBAKER	43
4.2. Toma de decisiones	44
4.2.1. Diseño del motor	45
4.2.1.1. Base con el sistema entidad-componente	45
4.2.1.2. Componentes genéricos de juego	46
4.2.1.3. Componentes específicos de RPG	47
4.2.1.4. Sistema de eventos	47
4.2.2. Diseño del editor	48
4.2.2.1. Funcionalidad del editor	48
4.2.2.2. Modularidad y arquitectura	50
4.2.2.3. Tecnologías utilizadas	50
5. Motor	51
5.1. <i>Core</i>	51
5.1.1. Sistema de entidad-componente	51
5.1.2. Sistema de gestión de recursos	52
5.1.3. Carga de escenas a partir de datos	54
5.2. Componentes genéricos de juego	55
5.2.1. <i>Render</i>	55
5.2.2. Entrada	57
5.2.3. Audio	57
5.2.4. Colisiones	58
5.3. Componentes específicos de RPG	59
5.3.1. Movimiento	59
5.3.2. Mapas	59
5.3.3. Diálogos	60
5.4. Sistema de eventos	60
5.4.1. Condiciones	61
5.4.2. Comportamientos	62
6. Editor	65
6.1. Base del editor	65
6.1.1. Elementos comunes	65
6.1.2. Elementos de lectura-escritura	66
6.1.3. Elementos de dibujado	69
6.1.4. Recursos	70
6.2. Interfaz del editor	73

7. Evaluación y análisis	75
7.1. Objetivo de la evaluación	75
7.2. Metodología	75
7.3. Resultados	76
7.4. Análisis de los resultados y conclusiones	77
8. Conclusiones y trabajo futuro	79
Conclusions and future work	81
9. Contribuciones Personales	83
Bibliografía	91
A. Guía de pruebas de usuario	97

Índice de figuras

1.1.	Diagrama de Gantt mostrando la planificación temporal del trabajo.	3
1.2.	Gantt diagram showing the temporal planning of the work.	9
2.1.	Partida de LARP nórdico, extraída de Pohjola (2021).	15
2.2.	Tablero de juego moderno de <i>Kriegsspiel</i> (von Reisswitz y von Reiswitz, 1812), extraída de Heinemann (2022).	17
2.3.	Imágenes de distintas variedades de partidas de <i>Dungeons & Dragons</i> . .	19
2.4.	Partida de <i>Warhammer Fantasy Battle</i> , extraída de Paddington (2023).	21
3.1.	Escena de juego de <i>The Legend of Zelda: Breath of the Wild</i> (Nintendo, 2017), extraída de Barder (2017).	25
3.2.	Interfaz de <i>dnd</i> (Whisenhunt y Wood, 1975), extraída de Addict (2019).	27
3.3.	Capturas de <i>Rogue</i> y <i>Ultima</i>	28
3.4.	Representación esquemática de la estructura de un juego y su motor con algunos de los componentes principales	32
3.5.	Ejemplos de separaciones entre motor y <i>gameplay</i> en los distintos motores.	35
3.6.	Representación esquemática de la arquitectura de un motor multiplataforma	38
3.7.	Vista de la interfaz de <i>Unreal Engine 5.2</i> , extraída de Wadstein (2023).	39
3.8.	Vista del <i>scripting</i> visual mediante nodos de <i>Godot</i> , extraída de Linetsky y Manzur (2022).	40
3.9.	Capturas de distintos elementos de la interfaz de <i>RPG Maker</i>	41
4.1.	Editor de mapas de RPGBAKER.	49
5.1.	Esquema de la estructura del sistema de entidad componente del motor.	53
5.2.	Esquema de la estructura del sistema de gestión de recursos del motor.	55
5.3.	Representación de un ejemplo de las conexiones de los elementos del sistema de audio del motor.	58
5.4.	Esquema de la estructura de un evento del motor.	61
6.1.	Contenido del fichero <code>es_ES.lua</code>	67
6.2.	Contenido del fichero <code>en_EN.lua</code>	67
6.3.	Distintas ventanas del editor de RPGBAKER.	73

Capítulo 1

Introducción

“Una vez tuve una conversación bastante rara con un par de abogados y estaban hablando sobre: «¿Cómo elegís a vuestro público objetivo? ¿Hacéis “focus groups”, encuestáis a gente y todo eso?» Y es como: «No, simplemente hacemos juegos que creemos que molan.»”
— John Carmack

1.1. Motivación

Los videojuegos de rol constituyen uno de los géneros más influyentes de la industria¹ desde sus orígenes en la década de los años 80 hasta la actualidad, donde concentran una gran parte de la cuota de mercado².

El desarrollo de este tipo de juegos ha sufrido cambios mayúsculos con el paso de los años y con las consecuentes mejoras *hardware* y *software*, que los han permitido evolucionar desde simples escenas monocromas y con interfaces basadas en texto hasta las representaciones tridimensionales en tiempo real, mundos abiertos complejos y sistemas de interacción avanzados. Estas mejoras han influido no solo en el apartado visual, sino también en la narrativa, la jugabilidad y la inmersión del jugador, permitiendo experiencias cada vez más ricas y personalizadas.

Numerosos programas de edición y desarrollo de videojuegos han aparecido en las últimas dos décadas, pero los más habitualmente usados están pensados para juegos de todo tipo, por lo que suelen tener características generales y no estrechamente relacionadas con el desarrollo de juegos de rol, y muchas veces restringen algunas de sus funcionalidades, lo que dificulta el desarrollo.

Aquellos programas que sí que están pensados para el desarrollo específico de videojuegos de rol tienen dos problemas fundamentales:

¹En total, se han vendido más de mil millones de copias de videojuegos de rol a lo largo de la historia, según *VGChartz* (<https://shorturl.at/VI2Zy>).

²Según *Rocket Brush Studio*, los videojuegos de rol corresponden al sector de videojuegos más vendido en el mercado móvil, el tercero en el mercado de PC y el quinto en el mercado de las videoconsolas (<https://rocketbrush.com/blog/most-popular-video-game-genres-in-2024-revenue-statistics-genres-overview>).

- Los que ofrecen una interfaz intuitiva, sencilla de utilizar, y bastante amigable con nuevos usuarios que están introduciéndose en el mundo del desarrollo de videojuegos están bajo un muro de pago. Este muro de pago hace que nuevos usuarios paguen por un *software* que rara vez utilizarán si no se afianzan finalmente en el mundo del desarrollo.
- Aquellos que no están bajo un muro de pago o que son *software* de código libre suelen tener interfaces y sistemas complejos, difíciles de entender para usuarios noveles, quienes a menudo abandonan el desarrollo debido a dicha complejidad.

La motivación principal de este trabajo surge de la necesidad de contar con herramientas sencillas de utilizar y flexibles, tanto para desarrolladores independientes experimentados que quieran crear juegos sin las limitaciones impuestas por los motores de uso general, como para personas sin amplio conocimiento en el desarrollo de videojuegos o en la programación de estos.

1.2. Objetivos

Este trabajo de fin de grado tiene marcados como objetivos el desarrollo de RPGBAKER, un entorno de desarrollo de videojuegos de rol 2D, compuesto por un motor multiplataforma, que pueda funcionar en Windows, MacOS, Linux y Android; acompañado de un editor, también multiplataforma, que pueda funcionar en Windows, MacOS y Linux, y que permita un desarrollo sencillo de videojuegos para este motor. El editor podrá generar ejecutables que el usuario solamente necesite distribuir sin la necesidad de hacer ningún paso extra posterior al desarrollo.

La interfaz del editor estará pensada para usuarios primerizos en el desarrollo, sin eliminar la posibilidad a usuarios más experimentados que quieran hacer juegos de mayor envergadura.

Para ello:

- Se desarrollará un motor multiplataforma (en Windows, MacOS, Linux y Android) que permita la implementación de videojuegos de rol.
- Se desarrollará un editor multiplataforma (en Windows, MacOS y Linux) que facilite la implementación de los videojuegos de rol en el motor desarrollado.
- Se probarán ambas herramientas con usuarios para demostrar el funcionamiento de RPGBAKER y se extraerán las conclusiones necesarias, así como posibles mejoras de cara al futuro.

1.3. Plan de trabajo

Para cumplir con los objetivos anteriores, se dividirá el plan de trabajo en cuatro fases:

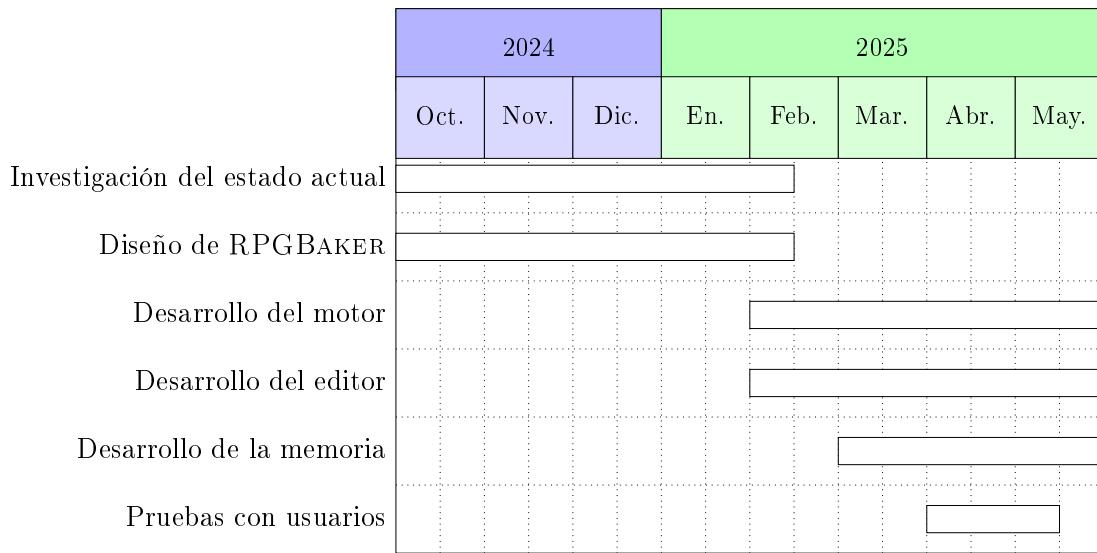


Figura 1.1: Diagrama de Gantt mostrando la planificación temporal del trabajo.

- Investigación del estado actual sobre los motores y editores específicos para videojuegos de rol, así como de los propios videojuegos de rol. En esta parte se intentarán abstraer las características comunes entre todos los motores, editores y videojuegos, tanto los de código libre como los que están bajo una capa de pago; y se intentarán proponer mejoras a los problemas que estos puedan tener de cara a nuevos usuarios poco experimentados.
- Diseño de RPGBAKER. Con las características abstraídas en la fase anterior, se planteará un diseño inicial que servirá como base durante el desarrollo del trabajo. Este diseño, si bien no será inmutable, debería ser lo más «final» posible para evitar problemas durante la fase de desarrollo.
- Desarrollo de RPGBAKER. Una vez finalizado el diseño, comenzará el desarrollo. Esta fase, a su vez, se dividirá en varias fases:
 - Puesta en marcha del entorno de desarrollo. Se elegirá un entorno de desarrollo dependiendo de las necesidades del motor y del editor, y se configurará todo para que sea lo menos trabajoso posible durante el desarrollo.
 - Desarrollo del motor. Se desarrollará un motor de acuerdo a los planes diseñados anteriormente, con soporte multiplataforma tanto para ordenador como para dispositivos móviles Android.
 - Desarrollo del editor. Al igual que con el motor, se desarrollará el editor de acuerdo al diseño preestablecido. Tendrá que tener soporte multiplataforma en PC, es decir, Windows, MacOS y Linux, no así con Android.
- Pruebas con usuarios. Para garantizar el correcto funcionamiento de RPGBAKER, se probarán las herramientas finales con usuarios de diversa índole ajenos al desarrollo de estas. Al finalizar las pruebas, se extraerán las conclusiones necesarias, y se retocarán todas aquellas funcionalidades críticas que

requieran de un arreglo antes de poder publicar la versión pública, dejando como trabajo futuro aquellas que supongan una excesiva carga como para poder desarrollarlas en el tiempo restante.

La figura 1.1 muestra un diagrama de Gantt con la planificación temporal del trabajo, que se empezó en el mes de octubre de 2024 y ha terminado en mayo de 2025. La parte de investigación y diseño ha correspondido a la primera parte del trabajo, desde el inicio del proyecto hasta mediados de febrero; la implementación del proyecto, desde mediados de febrero hasta finales de mayo; el desarrollo de la memoria, desde principios de marzo hasta finales de mayo; y, finalmente, las pruebas de usuario, que se hicieron entre la segunda quincena de abril y la primera de mayo.

1.4. Herramientas y Metodología

En cuanto a las herramientas, se usará Git como sistema de control de versiones, usando un repositorio alojado en GitHub, con la ayuda de GitHub Desktop como herramienta de manejo del este. La gestión de tareas se llevará a cabo a través de los proyectos que GitHub incorpora en su página web.

El acceso al repositorio con el código puede hacerse a través de esta dirección: <https://github.com/almasso/rpgbaker>.

Con respecto a las herramientas de desarrollo de RPGBAKER, se utilizarán como entornos de desarrollo CLion, para programación en C++, y Android Studio, para la programación en Java de Android; se usará CMake como herramienta de generación de librerías externas; y MinGW como compilador en Windows, Clang como compilador en MacOS, y GCC como compilador en Linux . Las razones detalladas del uso de estas herramientas se encuentran en la sección 4.2.

Por otra parte, la generación del PDF de la memoria se llevará a cabo mediante L^AT_EX, utilizando la plantilla de T_EXIS, y utilizando como entorno de edición de esta Texmaker.

En cuanto a la metodología de trabajo, se propondrán reuniones cada dos semanas con el tutor, pudiendo variar el número de semanas dependiendo del progreso realizado. En estas reuniones se expondrá el estado del trabajo, así como se realizarán consultas referidas al diseño o al desarrollo de este.

La comunicación con el equipo se establecerá, tanto mediante reuniones presenciales cuando se tengan que abordar problemas importantes, como mediante *software* que permita mensajería y chat de voz, como Discord. Es mediante esta herramienta que también se tratará de hacer las pruebas finales con los usuarios.

1.5. Estructura de la memoria

En el capítulo 2, *Juegos de rol*, se habla de los juegos de rol, sus características, orígenes así como un análisis de alguno de los juegos de rol más importantes.

En el capítulo 3, *Videojuegos de rol*, se introducen los videojuegos de rol y su historia, así como el proceso de desarrollo y el uso de motores y editores de videojuegos.

En el capítulo 4, *Planteamiento del proyecto*, se trata en profundidad el diseño planteado y las decisiones tomadas previas al desarrollo de las aplicaciones.

En el capítulo 5, *Motor*, se habla de la implementación del motor de RPGBAKER, con algunas de las decisiones tomadas en cuanto al desarrollo.

En el capítulo 6, *Editor*, se habla de la implementación del editor de RPGBAKER, con algunas de las decisiones tomadas en cuanto al desarrollo.

En el capítulo 7, *Evaluación y análisis*, se exponen las preguntas y objetivos de investigación, el desarrollo de las pruebas con los usuarios, y un análisis acerca de las pruebas, del que se han extraído conclusiones.

Finalmente, en el capítulo 8, *Conclusiones y trabajo futuro*, se detallan las conclusiones del proyecto así como posibles mejoras para una futura actualización de las herramientas que pueden ser interesantes.

Introduction

“I had this really bizarre conversation once with a couple of lawyers and they were talking about «How do you pick your target market? Do you use focus groups and poll people and all this?» It’s like «No, we just write games that we think are cool.»”

— John Carmack

Motivation

Role-playing video games are one of the most influential genres in the industry³, from their origins in the 1980s to the present day, where they account for a large portion of the market share⁴.

The development of this type of game has undergone significant changes over the years, along with consequent improvements in hardware and software, which have allowed them to evolve from simple monochrome scenes with text-based interfaces to real-time 3D graphics, complex open worlds, and advanced interaction systems. These improvements have influenced not only the visual aspect but also the narrative, gameplay, and player immersion, enabling increasingly rich and personalized experiences.

Numerous video game development and editing tools have appeared in the last two decades, but the most commonly used ones are designed for games of all kinds. As a result, they tend to offer general features that are not closely related to role-playing game development, and they often restrict certain functionality, making the development process more complicated.

Those tools that are specifically designed for role-playing game development have two fundamental flaws:

- Those that offer an intuitive, easy-to-use interface that is friendly to newcomers in game development are locked behind a paywall which, although not very expensive, forces amateur users to pay for a software they will rarely unless they consolidate as developers.

³In total, more than one billion copies of role-playing video games have been sold throughout history, according to *VGChartz* (<https://shorturl.at/VI2Zy>).

⁴According to *Rocket Brush Studio*, role-playing video games correspond to the best-selling genre in the mobile market, the third-best in the PC market, and the fifth-best in the console market (<https://rocketbrush.com/blog/most-popular-video-game-genres-in-2024-revenue-statistics-genres-overview>).

- Those that are not locked behind a paywall or are open source, often have interfaces and systems that are difficult for beginners to understand, leading many to abandon game development due to the complexity of these tools.

The main motivation behind this project arises from the need to have easy-to-use and flexible tools, both for experienced independent developers who desire to create games without the limitations imposed by general-purpose engines, and for people with little knowledge of video game development or programming.

Objectives

This bachelor's thesis aims to develop RPGBAKER, a 2D role-playing game development environment consisting of a cross-platform engine capable of running on Windows, MacOS, Linux and Android, along with an editor, also cross-platform, that works on Windows, MacOS and Linux, and enables simple and straightforward game development for this engine. The editor will be capable of generating executable files that users can distribute without the need for any additional steps after development.

The editor's interface will be designed for beginners in game development, while still allowing more experienced users to create larger and more ambitious projects.

To accomplish this:

- A cross-platform (on Windows, MacOS, Linux and Android) engine that allows the user to implement role-playing games will be developed.
- A cross-platform (on Windows, MacOS and Linux) editor that eases the implementation of a role-playing game in the developed engine will be developed.
- RPGBAKER will be tested with users to demonstrate its functionality, and necessary conclusions will be drawn, along with potential improvements for the future.

Work Plan

To accomplish the previous objectives, the work plan will be divided in four phases:

- Research on the current state of engines and editors specifically designed for role-playing games, as well as on role-playing games themselves. This section will aim to identify the common features shared by all engines, editors and games, both open-source and commercial ones, and propose improvements to address the issues these tools may present for inexperienced new users.
- RPGBAKER design. Based on the common features identified in the previous phase, an initial design will be proposed to serve as a basis for the development of the project. This design, while not immutable, should be as close to final as possible to avoid any issues during the development phase.

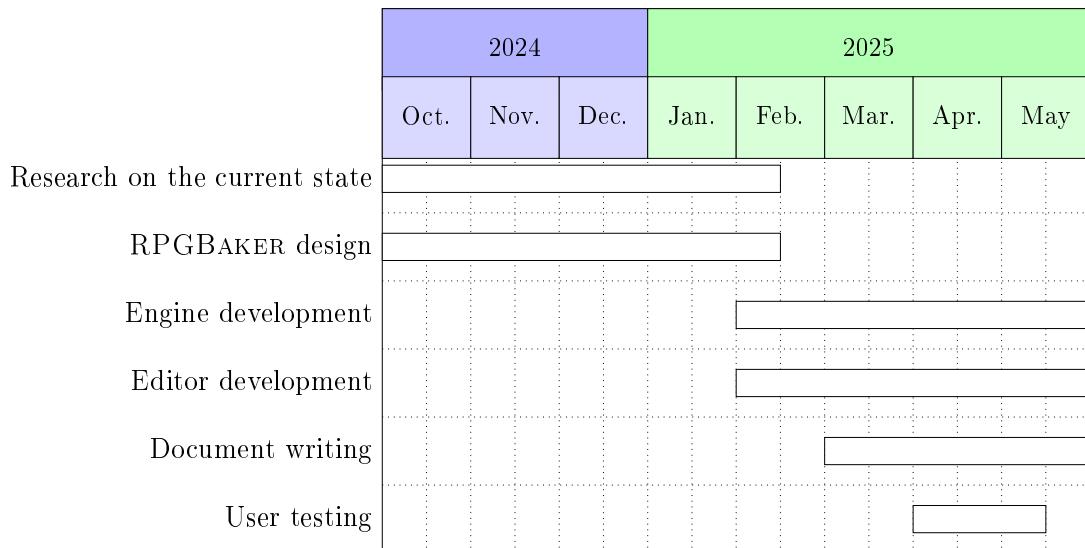


Figure 1.2: Gantt diagram showing the temporal planning of the work.

- RPGBAKER development. Once the design phase is completed, development will begin. This phase will, in turn, be split across multiple phases:
 - Setting up the development environment. A development environment will be chosen based on both applications' needs, and everything will be configured to minimize effort during the development.
 - Engine development. An engine will be developed according to the previously established plans, with cross-platform support for both PCs and Android mobile devices.
 - Editor development. Like the engine, the editor will be developed according to the established design. It must support multiple platforms on PC, namely Windows, MacOS and Linux, but not Android.
- User testing. To ensure the correct functioning of RPGBAKER, the final tools will be tested by a variety of users not involved in their development. After the tests are completed, the necessary conclusions will be drawn, and the tools will be adjusted to address any critical issues that require attention before releasing the public version. Features that would require excessive effort to implement within the available time will be left as future work.

Figure 1.2 shows a Gantt chart illustrating the project timeline, which began in October 2024 and ended in May 2025. The research and design phase took place during the first part of the project, from its inception until mid-February; the implementation phase spanned from mid-February to the end of May; the writing of this document was carried out from early March to late May; and finally, user testing was conducted between the second half of April and the first half of May.

Tools and Methodology

Regarding the tools, Git will be used as the version control system, with a repository hosted on GitHub and managed through GitHub Desktop. Task management will be carried out using GitHub Projects, available on the GitHub website.

Access to the repository containing the code may be done through this link: <https://github.com/almasso/rpgbaker>.

Regarding the development tools for RPGBAKER, CLion will be used as the C++ development environment, and Android Studio for Android development in Java. CMake will be used as the tool for generating external libraries, and MinGW for handling compilation on Windows, Clang on MacOS, and GCC on Linux. Detailed reasons for choosing these tools can be found in section 4.2.

On the other hand, PDF generation of the document will be carried out using L^AT_EX, with the T_EX^IS template, and Texmaker will be used as the editing environment.

Regarding the working methodology, bi-weekly meetings will be proposed with the tutor, although the number of weeks may vary depending on the progress made. During these meetings, the current status of the project will be presented, and advice related to the design or development will be requested.

Communication with the team will be established through both in-person meetings when important issues need to be addressed and via software that allows messaging and voice chat, such as Discord. This tool will also be used for conducting user testing.

Document Structure

In Chapter 2, *Role-playing games*, the topic of role-playing games is discussed, including their characteristics, origins, and an analysis of some of the most important titles in the genre.

In Chapter 3, *Role-playing video games*, role-playing video games and their history are introduced, along with the development process and the use of game engines and editors.

In Chapter 4, *Project planning*, the design and decisions made prior to the development of the applications are discussed in detail.

In Chapter 5, *Engine*, the implementation of RPGBAKER's engine is discussed, along with some of the decisions made regarding its development.

In Chapter 6, *Editor*, the implementation of RPGBAKER's editor is discussed, along with some of the decisions made regarding its development.

In Chapter ??, *Evaluation and analysis*, the research questions and objectives are presented, along with the development of the user testing phase and an analysis of the tests, from which conclusions are drawn.

Finally, in Chapter 8, *Conclusions and future work*, the project's conclusions and potential improvements for the final implementation of the applications that may be of interest are detailed.

Capítulo 2

Juegos de rol

RESUMEN: En este capítulo se explicará qué son los juegos de rol, se introducirá su historia, y se examinarán algunos de los juegos de rol más importantes.

2.1. ¿Qué es un juego de rol?

Los juegos de rol (RPG, *role-playing game*) son, en palabras de Lortz (1979, como se cita en Fine (1983)), «todos aquellos juegos que permiten a un determinado número de jugadores asumir los roles de personajes imaginarios y operar con cierto grado de libertad en un entorno imaginario».

Este tipo de juegos se caracteriza por una base muy sólida, que Tychsen et al. (2006) resumen en los siguientes elementos:

- **Narrativa estructurada por reglas:** los juegos de rol se basan en contar una historia dentro de un marco de reglas específicas. Tanto la narrativa como las reglas son únicas para cada juego, proporcionando una estructura que guía la progresión de este y las decisiones de los jugadores.
- **Participación múltiple y mundo ficticio compartido:** los juegos de rol requieren de la participación de múltiples jugadores (al menos dos), quienes interactúan dentro de un mundo ficticio común. Es esencial que todos los participantes comprendan la ambientación, el entorno y las reglas antes de comenzar la partida.
- **Control de personajes por parte de los jugadores:** la mayoría de los participantes asumen el control de al menos un personaje durante toda la partida. A través de estos personajes, los jugadores interactúan con el entorno y con otros personajes, desarrollando la narrativa del juego.
- **Dirección del juego mediante un instructor:** por lo general, existe la figura del director de juego (GM, *gamemaster*) que administra los aspectos del juego no controlados directamente por los jugadores. El *gamemaster* facilita el flujo del juego, proporciona contenido ambiental del mundo ficticio y arbitra las reglas y los conflictos que puedan surgir.

Sumado a estos elementos estructurales, se encuentran otras características igualmente esenciales para definir la experiencia de un juego de rol:

- **Progresión del personaje:** los personajes evolucionan mecánica, narrativa y emocionalmente a lo largo del tiempo mediante la obtención de puntos de experiencia que se pueden «gastar» en un sistema de niveles (Barton, 2008).
- **Inmersión:** resulta fundamental para comprender el atractivo de este género, permitiendo a los jugadores experimentar emociones y tomar decisiones desde la perspectiva de sus personajes (Montola y Stenros, 2010).
- **Narrativa emergente:** es decir, la construcción dinámica y colaborativa de la historia a partir de las acciones de los participantes (Fine, 1983).
- **Decisiones significativas:** las decisiones de los jugadores deben tener consecuencias reales en el desarrollo de la partida (Tekinbas y Zimmerman, 2003).
- **Alto grado de flexibilidad y adaptabilidad:** las reglas, ambientaciones o mecánicas se pueden modificar dependiendo de las necesidades del grupo, lo que evidencia un carácter modular y abierto (Edwards, 2001).

Por otra parte, existen algunos aspectos menos formalizados pero igualmente importantes para el funcionamiento y disfrute de los juegos de rol, como por ejemplo, un contrato social, en el que los jugadores acuerden las temáticas permitidas, el tono de la partida y las dinámicas sociales; una coherencia tonal y de género, que permita mantener una ambientación y tonos coherentes para la inmersión en la partida; fomento de la expresividad de los jugadores, es decir, beneficiarse del uso de la voz, gestos y una narración detallada para enriquecer la interpretación de los personajes; libertad de improvisación frente a lo inesperado; y la exploración emocional y empática con realidades ajenas.

En conjunto, todos estos elementos conforman una experiencia lúdica rica, dinámica y profundamente humana, donde la colaboración, interpretación y narrativa se entrelazan de forma única. Lejos de limitarse a un solo modelo, los juegos de rol han dado lugar a una amplia variedad de formas y enfoques, cada uno con sus propias mecánicas, estilos narrativos y niveles de complejidad.

2.1.1. Tipos de juegos de rol

A lo largo del tiempo, los juegos de rol han evolucionado en múltiples direcciones, dando lugar a distintas topologías con enfoques, estructuras y estilos de juego muy variados. Atendiendo a estas características, se presenta una clasificación general de los tipos más representativos:

2.1.1.1. Juegos de rol de mesa o tablero

Los juegos de rol de mesa (TTRPG, *tabletop role-playing games*), constituyen la forma más tradicional de juegos de rol. En este tipo de juegos, los jugadores se reúnen en torno a una mesa física o virtual y cada uno asume el papel de un



Figura 2.1: Partida de LARP nórdico, extraída de Pohjola (2021).

personaje ficticio. Uno de los participantes actúa como GM, responsable de narrar la historia, interpretar a los NPC (*non-player character*, personaje no jugador) y arbitrar las reglas.

Uno de los elementos más representativos de los TTRPG es el uso de dados poliédricos para resolver acciones inciertas. Por ejemplo, en *Dungeons & Dragons* (Gygax y Arneson, 1974), se utiliza un dado de veinte caras (popularmente conocido como «d20») para realizar la mayoría de las pruebas. Si un personaje intenta realizar una acción desafiante, como escalar una pared o atacar a un enemigo, el jugador tira el dado y suma ciertos modificadores; si el resultado iguala o supera una dificultad determinada por el GM, la acción tendrá éxito.

Estos juegos también incluyen hojas de personaje con estadísticas numéricas (como fuerza, inteligencia o destreza), inventarios, puntos de vida y habilidades especiales. Además, el progreso de los personajes se da mediante la adquisición de experiencia, lo que permite mejorar atributos, aprender nuevas habilidades o subir de nivel.

2.1.1.2. Juegos de rol en vivo

Los juegos de rol en vivo (LARP, *live-action role-playing game*) son una variante en la que los participantes interpretan físicamente a sus personajes en un espacio real, vistiéndose y actuando conforme a su rol. No se juega en torno a una mesa, sino que se representa la historia a través de la acción directa, muchas veces con elementos de escenografía, vestuario y una mayor carga teatral.

Las reglas suelen ser simplificadas o adaptadas para no interrumpir el flujo de la interpretación, y en muchos casos se utilizan sistemas de señales o puntos para resolver conflictos, en lugar de dados. En ciertos LARP de combate, se emplean armas acolchadas para simular enfrentamientos físicos.

Este formato se presta especialmente para la exploración emocional, la inmersión profunda y la representación de tramas políticas, sociales o dramáticas.

Dentro del género LARP, encontramos el subgénero de LARP nórdico, inspirado en la cultura nórdica en la que el fracaso es parte de la trama y es mucho más artístico que los LARP tradicionales, como se puede apreciar en la figura 2.1, con la recreación de un dragón a tamaño real y el involucramiento de numerosos jugadores.

2.1.1.3. Juegos de rol por foro o texto

Este tipo de juegos se desarrollan completamente por escrito, ya sea en foros de internet, chats o correos electrónicos. Los participantes escriben mensajes en los que narran las acciones, pensamientos y diálogos de sus personajes, construyendo la historia de manera colaborativa.

A menudo carecen de un sistema rígido de reglas o tiradas de dados, y dependen más de la narración consensuada y de la calidad interpretativa. En algunos casos se incorporan sistemas de puntos, turnos o moderados para mantener la coherencia y el equilibrio del juego.

Este tipo de juego permite un desarrollo más introspectivo de los personajes y tramas más complejas, ya que los jugadores disponen de tiempo para redactar sus intervenciones.

2.1.1.4. Juegos de rol digitales

Los juegos de rol digitales (CRPG, *computer role-playing games*, juegos de rol de ordenador) son videojuegos que adoptan las mecánicas y estructuras de los juegos de rol tradicionales.

Sobre este tema se ampliará más en el capítulo 3, donde se tratará en profundidad todo lo relacionado con este género de videojuegos.

2.1.1.5. Comparativa

A modo de resumen, se incluye una tabla comparativa entre los distintos tipos anteriormente vistos.

Tipo	Medio	Resolución de acciones	Inmersión
De mesa (TTRPG)	Presencial / Virtual	Dados y reglas estructuradas	Alto
En vivo (LARP)	Presencial	Señales, puntos o actuación directa	Alto
Por foro / texto	Digital (escrito)	Narración compartida y consenso	Alto
Digital (CRPG)	Videojuegos	Algoritmos computacionales	Variable

2.2. Historia de los juegos de rol

La historia de los juegos de rol está íntimamente ligada a la evolución de los juegos de estrategia, la literatura de fantasía y la experimentación lúdica del siglo XX. Su origen puede rastrearse a las décadas de 1960 y 1970, con influencias que abarcan desde los *wargames* (juegos de guerra) hasta las obras de J.R.R. Tolkien.



Figura 2.2: Tablero de juego moderno de *Kriegsspiel* (von Reisswitz y von Reisswitz, 1812), extraída de Heinemann (2022).

Los juegos de rol tienen sus raíces en los *wargames*, juegos de estrategia militar que simulan conflictos bélicos con miniaturas y mapas. Entre ellos se destaca el pionero *Kriegsspiel* (von Reisswitz y von Reisswitz, 1812), diseñado por un noble prusiano que quería dotar al ejército de un entrenamiento táctico en caso de guerra. En la figura 2.2 podemos ver el tablero de juego, consistente en una representación cartográfica de un territorio real, y, encima, las unidades militares de ambos bandos (cada una de un color, y cada rango dependiendo del patrón dibujado).

Este tipo de juegos de mesa se fueron desarrollando con el paso del tiempo, apareciendo también libros que contenían una serie de reglas para desarrollar partidas más interesantes, como es el caso de *Little Wars* (Wells, 1913), convirtiendo a este tipo de juegos en simulaciones mucho más narrativas.

No es hasta la década de los 70, con la aparición de juegos como *Chainmail* (Gygax y Perren, 1971), que la comunidad comienza a incorporar elementos de personajes individuales y no colecciones de unidades militares. Con este suceso, se comienza a hablar propiamente de «juegos de rol», y se empiezan a ver los primeros juegos comerciales de este tipo.

Se considera que *Dungeons & Dragons* (Gygax y Arneson, 1974) es el primer juego de rol comercial. Este juego combinaba reglas derivadas de los *wargames* con elementos narrativos inspirados en la literatura fantástica. Cada jugador asumía el papel de un aventurero (por ejemplo, un guerrero o un mago) y exploraba mazmorras, resolvía acertijos y combatía monstruos, todo bajo la guía de un GM. Esta revolucionaria idea de interpretar un personaje y tomar las decisiones desde su perspectiva sentó las bases de este género, que poco a poco se fue popularizando.

En la década de los 80, este género se expandió, dando a lugar a nuevos sistemas y ambientaciones, como la saga de horror *Call of Cthulhu* (Petersen y Willis, 1981)

o la de fantasía *RuneQuest* (Perrin et al., 1983). Los temas y las mecánicas comienzan a diversificarse, y se comienzan a publicar suplementos, novelas y productos derivados que expandían los universos de los juegos. También aparecen los primeros videojuegos que incorporaban mecánicas de estos juegos de tablero, apareciendo el género de los CRPG. Es a finales de esta época cuando comienzan a surgir los primeros LARP, y se empiezan a organizar convenciones temáticas, consolidando la cultura de jugadores de este género.

Entrando en la nueva década, crecen las comunidades LARP tanto en Estados Unidos como en Europa gracias a juegos como *Mind's Eye Theatre* (Rein-Hagen y Davis, 1993), mientras que la normalización de internet en los hogares permitió el surgimiento del rol por foro y correo electrónico, ampliando el acceso a este tipo de contenidos.

Finalmente, en el siglo XXI, el rol se diversificó aún más. Los CRPG se volvieron cada vez más complejos y tenían un mayor alcance, surgen los MMORPG (*massively multiplayer online role-playing game*, videojuegos de rol multijugador masivos en línea), que ofrecen experiencias compartidas masivas, mientras que el auge de plataformas como Discord, Roll20 o Foundry Virtual Tabletop revitalizó el juego de rol de mesa en línea.

La popularización del *streaming* en la década del 2010, y la aparición de *podcasts* de rol como *Critical Role* ayudaron a introducir el género a nuevas generaciones, mostrando su gran potencial narrativo y creativo. Hoy en día, los juegos de rol son practicados por millones de personas en todo el mundo, en formatos que van desde campañas caseras hasta multitudinarios eventos de LARP, o desde videojuegos AAA hasta partidas por texto en móviles.

2.3. Análisis de los principales juegos de rol

2.3.1. *Dungeons & Dragons*

Como se ha mencionado anteriormente, *Dungeons & Dragons* (Gygax y Arneson, 1974), a veces conocido únicamente como D&D, es ampliamente conocido como el primer juego de rol comercial y el más influyente del género, y su origen se basa en el sistema de reglas de *Chainmail* (Gygax y Perren, 1971), al cual se le añadieron elementos de exploración, personajes individuales y narrativa.

D&D se estructura en torno al concepto de un grupo de aventureros que exploran mundos de fantasía, enfrentan criaturas, resuelven conflictos y evolucionan mediante un sistema de niveles y experiencia. El director de juego (aquí llamado *dungeon master*, DM) narra la historia, describe el mundo y controla a los NPC y eventos.

A partir de la tercera edición, del año 2000, se introduce el *sistema d20*, un dado de veinte caras en el que se basa todo el sistema de reglas, así como hojas de personaje detalladas con diversos atributos.

En la figura 2.3 podemos ver varios ejemplos de partidas de D&D, siendo la que está capturada en la figura 2.3a un ejemplo de una partida normal, con un tablero



(a) Imagen de una partida común de D&D, extraída de Anh y Nguyén (2021).



(b) Imagen de un diorama para el escenario de una partida de D&D, extraída de García (2016).

Figura 2.3: Imágenes de distintas variedades de partidas de *Dungeons & Dragons*.

y sus diferentes *d20*, y, en la figura, 2.3b, un ejemplo de un diorama creado por un aficionado a la maquetación, que cuenta con todo lujo de detalles y que llega a ocupar toda una sala.

El juego ha pasado por múltiples ediciones, cada una introduciendo cambios en las mecánicas, balance y filosofía de diseño. Su edición más reciente, la quinta, del año 2014, ha tenido un gran éxito comercial y ha contribuido al resurgimiento del juego de rol gracias a su gran accesibilidad, su enfoque narrativo y la visibilidad en plataformas como YouTube o Twitch.

El impacto de este juego en la cultura popular y en el diseño de otros juegos es difícil de sobreestimar, ya que está considerado el estándar de facto en los TTRPG y ha inspirado hasta adaptaciones cinematográficas¹.

2.3.2. *Pathfinder*

Pathfinder (Jason Bulmahn, 2009) nació como una evolución del *sistema d20* de D&D. Ante el cambio de enfoque que supuso la cuarta edición de D&D, muchos jugadores buscaron una alternativa que conservara la complejidad y flexibilidad del sistema anterior. Se aprovechó de la licencia OGL (*Open Game License*, Licencia de Juego Abierto)² para crear un sistema propio que mejorara y expandiera el marco de reglas original.

El juego destaca por su nivel de detalle, la personalización de personajes y la riqueza de opciones tácticas durante el combate. El sistema de creación de personajes permite una gran variedad de combinaciones de clases, habilidades, dotes y objetos mágicos. Esto atrae especialmente a jugadores que valoran la optimización mecánica y la profundidad estratégica.

¹La saga *Dungeons & Dragons* (Solomon, 2000) ha recaudado más de 250 millones de dólares en total sumando las cuatro películas, siendo la última entrega, de 2023, la más exitosa de todas.

²Esta licencia se usa principalmente en TTRPG y concede, por parte de los diseñadores, permisos de modificación, copia y redistribución de algunos de los contenidos diseñados para sus juegos, principalmente mecánicas.

Narrativamente, mantiene el enfoque épico y de alta fantasía de D&D, pero con mundos propios, combinando elementos clásicos con tramas políticas, horror cósmico o mitologías exóticas.

En 2019 se publicó su segunda edición, que renovó el sistema con nuevas mecánicas más modernas, como un sistema de tres acciones por turno y una progresión más clara. Pese a que no ha alcanzado el nivel de popularidad de D&D, *Pathfinder* ha consolidado una gran comunidad y ha mantenido una identidad propia, por lo que muchas páginas lo sitúan siempre en segundo o tercer lugar de popularidad³.

2.3.3. *Call of Cthulhu*

Call of Cthulhu (Petersen y Willis, 1981) está basado en los mitos creados por H.P. Lovecraft, y representa un enfoque radicalmente distinto al de otros juegos de rol de la época, ya que se centra en la investigación, el horror psicológico y la fragilidad humana ante lo desconocido.

El sistema utiliza como base las reglas BRP *Basic Role-Playing*, Juego de Rol Básico, que utiliza porcentajes para determinar el éxito de las acciones. Los personajes son investigadores en los años 20 (aunque hay adaptaciones a otras épocas), y deben enfrentarse a cultos secretos, criaturas indescriptibles y horrores cósmicos. A diferencia de otros juegos, la muerte o locura de los personajes es común y está integrada en la narrativa, lo que fomenta una experiencia inmersiva y tensa.

El atributo de cordura (SAN, del inglés *sanity*) es central, ya que los encuentros con lo sobrenatural pueden erosionar la mente de los personajes hasta conducirlos a la locura. Esta mecánica refleja la atmósfera opresiva de las obras *lovecraftianas* y crea un estilo de juego más introspectivo y sombrío.

Call of Cthulhu ha sido aclamado por su capacidad de generar tensión narrativa, y ha influido en muchos otros juegos de rol e investigación, convirtiéndolo en un pilar fundamental del rol narrativo.

2.3.4. *Warhammer Fantasy Roleplay*

Warhammer Fantasy Roleplay (Halliwell et al., 1986) comparte ambientación con el universo de miniaturas de *Warhammer Fantasy Battle* (Ansell et al., 1983) (ver figura 2.4), pero se diferencia por su tono más oscuro, realista y decadente.

Ambientado en un mundo inspirado en la Europa del Renacimiento, pero plagado de corrupción, magia caótica y criaturas monstruosas, WFRP pone al jugador en la piel de personajes que suelen comenzar como personas comunes (rateros, campesinos o aprendices), y no como héroes épicos. El juego enfatiza la progresión lenta, la supervivencia y la toma de decisiones difíciles.

Su sistema de reglas, basado también en porcentajes, incluye un sistema de carreras profesionales que estructura el desarrollo del personaje de forma naturalista. Las

³La página *The Dragon's Trove* lo sitúa en tercer lugar en su histórico: <https://www.dragonstrove.com/blogs/news/the-best-tabletop-rpgs-of-all-time>.



Figura 2.4: Partida de *Warhammer Fantasy Battle*, extraída de Paddington (2023).

heridas son graves, la corrupción mágica puede deformar al personaje, y la muerte puede llegar de forma repentina. Esto genera un entorno de juego más peligroso y centrado en la narrativa emergente.

WFRP es un referente del estilo conocido como *grimdark*⁴, que enfatiza mundos crueles, moralidades grises y destinos trágicos. Esta saga ha sido particularmente influyente en Europa, y ha servido de inspiración para sagas de videojuegos como la de *The Witcher*.

⁴La expresión *grimdark* proviene de una de las entregas de la saga *Warhammer*, concretamente, *Warhammer 40000*, en la que se explica que «In the **grim** darkness of the far future there is only war» (En la sombría oscuridad del futuro lejano solo hay guerra).

Capítulo 3

Videojuegos de rol

RESUMEN: En este capítulo se explicará qué son los videojuegos, haciendo un enfoque en los videojuegos de rol, se introducirá su historia y se hablará del proceso de desarrollo de estos, analizando la importancia de los motores y editores en el desarrollo de videojuegos.

3.1. ¿Qué es un videojuego?

Definir qué es un videojuego es una tarea bastante complicada, sobre todo teniendo en cuenta los matices con los que se puede definir (académicos, de diseño, experimentales o tecnológicos). Una de las definiciones más acertadas la da Esposito (2005), que lo define como «un *juego* que se *juega* gracias a un *aparato audiovisual*, y que puede estar basado en una *historia*». El propio Esposito se encarga de definir qué es el *juego*, qué es *jugar*, qué es el *aparato audiovisual* y qué es la *historia*.

La diferencia fundamental de los juegos tradicionales frente a los videojuegos es la existencia de ese *aparato audiovisual* (videoconsolas, ordenador, dispositivos móviles) que pueda ofrecer una interacción «humano-máquina», ya que es esta interacción recíproca la que hace que los videojuegos sean videojuegos y no otro tipo de entretenimiento multimedia.

A la hora de diseñar un videojuego, hay que tener en cuenta tres conceptos importantes, dos de ellos propuestos por Hunicke et al. (2004) en su modelo MDA¹, las mecánicas, las dinámicas y la jugabilidad:

- Las **mecánicas** son las reglas, sistemas y acciones posibles dentro del juego. Incluyen elementos como el movimiento, la recolección de objetos, el combate o el sistema de progresión de los personajes. Se podrían definir como «los bloques de construcción del juego».

¹El modelo MDA de Hunicke et al. intenta cerrar las posibles brechas entre el diseño y el desarrollo de un juego basándose en tres aspectos fundamentales: mecánicas, dinámicas y estética (*Mechanics-Dynamics-Aesthetics*).

- Las **dinámicas** son los comportamientos emergentes que surgen de la interacción entre el jugador y las mecánicas, y están estrechamente relacionadas con la experiencia del jugador. Por ejemplo, una mecánica de sigilo puede generar una dinámica de tensión y precaución.
- La **jugabilidad** (o *gameplay*) es la experiencia interactiva del jugador dentro de un videojuego. Pese a que ni Tekinbas y Zimmerman (2003) ni Schell (2019) dan una definición concreta, hablan de cómo la interacción entre el jugador con las reglas del juego y con las mecánicas (es decir, las dinámicas) dan lugar a la jugabilidad. No solo cada juego tiene una jugabilidad distinta que depende de las dinámicas, mecánicas y reglas, sino que puede haber una mecánica emergente que surja a través de las decisiones que cada jugador tome.

Dependiendo de las mecánicas principales, los objetivos y las formas de interacción que tenga un videojuego, se pueden clasificar en una amplia variedad de géneros.

Los primeros intentos de clasificar los videojuegos mediante características comunes surgen en la década de los 80, principalmente por diseñadores y desarrolladores, como Crawford (1984), quien los categorizó entre aquellos que *requieren de habilidades previas por parte del usuario* (como juegos de combate o de carreras), y *juegos de estrategia* (que engloba al resto de juegos, como los de aventuras, los educativos, y los RPG). Estas categorías son *funcionales*, se centran en las mecánicas de juego y enfatizan cómo los jugadores interactúan con el sistema.

Esta categorización ha ido cambiando a lo largo de la historia, hasta llegar a la actualidad, donde los distintos géneros vienen dados por una mezcla de mecánicas, temas, elementos narrativos, estética o lugar de origen y plataforma, y están gravemente influenciados por las tendencias del mercado, discursos mediáticos y la propia percepción de los jugadores. También, muchas veces se quiere categorizar a los videojuegos con etiquetas propias de otras modalidades, como el cine o la literatura, que son incapaces de capturar los aspectos únicos que definen a un videojuego.

Entre esta categorización más «clásica» se encuentran los siguientes géneros:

- Acción, que son todos aquellos juegos que se centran en la velocidad de reacción del jugador y en la coordinación, como por ejemplo juegos de disparos (*shooters*) o plataformas.
- Aventura, que prioriza la exploración, la narrativa y, a veces, la resolución de acertijos, como en los juegos *point-and-click*, en los que se interactúa con el entorno clicando en elementos de la pantalla.
- Estrategia, que requieren una planificación táctica, ya sea en tiempo real (RTS, *real-time strategy*) o por turnos (TBS, *turn-based strategy*)
- Simulación, que buscan reproducir aspectos de la vida real, como la gestión de recursos, la conducción de un vehículo o pilotaje de un avión.
- Deportes y carreras, que emulan disciplinas deportivas o competiciones de velocidad.



Figura 3.1: Escena de juego de *The Legend of Zelda: Breath of the Wild* (Nintendo, 2017), extraída de Barder (2017).

- Puzzles, que plantean desafíos cognitivos lógicos o espaciales.
- Rol, que atendiendo a la definición dada en el capítulo 2, son todos aquellos juegos que permitan a sus usuarios encarnar el rol de uno o varios personajes en un mundo ficticio, donde puedan tomar decisiones, interactuar con su entorno y desarrollar una narrativa para conseguir un determinado objetivo.

Otro aspecto importante en la clasificación de videojuegos es la dimensión en la que se desarrolla la acción. Tradicionalmente, se distingue entre juegos en 2D , 2.5D y 3D.

Los juegos en 2D se caracterizan por usar gráficos bidimensionales, donde el movimiento y la interacción transcurren en un plano. Este tipo de juegos incluye una gran variedad de géneros clásicos, como los plataformas, puzzles o juegos de aventura gráfica.

Los juegos en 3D desarrollan su acción en un espacio tridimensional completo, donde el jugador puede desplazarse libremente en los tres ejes cartesianos. Suelen ser juegos de exploración, simulación y realismo, predominantes en géneros como *shooters*, juegos de acción y algunos RPG modernos.

Por último, los juegos en 2.5D combinan elementos de ambos mundos: presentan gráficos que aparentan ser tridimensionales mediante técnicas como proyecciones isométricas, o bien emplean entornos 3D con movimiento restringido a dos dimensiones. Este enfoque permite aprovechar la profundidad visual y efectos modernos sin perder la claridad o simplicidad propia del 2D.

Clarke et al. (2015) argumentan que las categorías existentes a día de hoy se quedan cortas para satisfacer los propósitos del género en videojuegos (identidad, agrupación, *marketing* y educación), ya que dada la gran diversidad de juegos que hay ahora, resulta casi imposible poder agrupar la naturaleza multifacética de estos en etiquetas tan «tradicionales».

El ejemplo más claro se ve en uno de los juegos más exitosos de la última década, *The Legend of Zelda: Breath of the Wild* (Nintendo, 2017), que mezcla elementos de libre exploración (lo que lo convertiría en un *sandbox* o «juego libre»), como bien se puede apreciar en la figura 3.1 por el enorme mundo en el que se desarrolla la acción; acción en tiempo real e investigación y resolución de rompecabezas (lo que lo convertiría en un juego de «acción-aventura»); y la progresión en tiempo real del personaje característica de los RPG (se puede ir consiguiendo nuevas habilidades o mejorando el equipamiento). Es por eso que muchas veces hay géneros intermedios, como en este caso, que se podría considerar a *Breath of the Wild* como un RPG de acción (ARPG, *action role-playing game*), que igualmente siguen sin englobar a la increíble diversidad de juegos. Este problema también se puede aplicar a la inversa, donde juegos como *Undertale* (Toby Fox, 2015), esencialmente un RPG, tiene elementos, como el combate, propios de otro género de juegos.

Hay muchas formas de evitar este problema, y quizá la mejor solución sea dejar de considerar a los géneros como «cajones estancos» en los que un videojuego no pueda pertenecer a dos de estas categorías simultáneamente (Apperley, 2006), sino que sean más bien un espectro, sin fronteras establecidas, en el que un videojuego pueda caer entre dos categorías distintas.

En resumen, antes de categorizar un videojuego en según qué género, hay que entender que resulta imposible que este se reduzca a una fórmula fija, sino que hay que entenderlo como una combinación fluida de mecánicas, elementos narrativos, temas y estética, que varían de juego a juego.

3.1.1. Historia de los videojuegos de rol

Es a mediados de la década de los 70 cuando se puede hablar del nacimiento de los videojuegos de rol. Dadas las limitaciones tecnológicas de la época, los RPG primitivos no eran más que simples adaptaciones de juegos de mesa ya existentes por entonces, como *dnd* (Whisenhunt y Wood, 1975), una adaptación del anteriormente mencionado *Dungeons & Dragons*, de donde se tomó su sistema de combate e historia; y posteriormente, en juegos más modernos, se ha tomado el *sistema d20*, las estadísticas de los personajes, o los sistemas de niveles.

Estos primeros juegos contaban con interfaces basadas en texto, resaltadas con colores muy vivos, y pocos *sprites*, muchas veces llegando a dibujarlos utilizando caracteres ASCII (ver figura 3.2), y normalmente estaban programados en los grandes ordenadores que se encontraban en campus universitarios como los de Harvard o la Universidad de Illinois. Esta primera etapa, Barton (2008) la denomina como la etapa de «años oscuros»², ya que es escasa la información que hay hoy en día sobre estos juegos, autores o desarrollo, ya que muchos no se han conservado y se consideran *lost media*, es decir, materiales multimedia que ya no existen en ningún formato y para los cuales no hay ninguna copia disponible.

De esta primera etapa también cabe mencionar tres videojuegos que dieron comienzo a tres distintos subgéneros dentro de los RPG:

²En un símil con los años oscuros del Medievo europeo, ya que es un período caracterizado por la falta de literatura e historia escritas y por una decadencia constructiva y cultural.

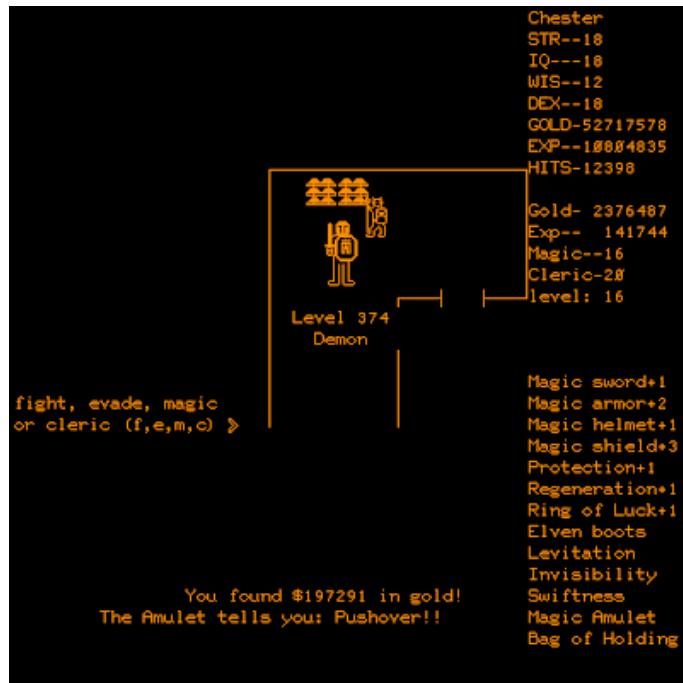


Figura 3.2: Interfaz de *dnd* (Whisenhunt y Wood, 1975), extraída de Addict (2019).

- *Rogue: Exploring the Dungeons of Doom* (A.I. Design, 1980), que dio lugar a los videojuegos de mazmorra o *roguelikes*, caracterizados por la generación aleatoria de un laberinto o mazmorra (ver figura 3.3a) en el que se desarrolla una aventura basada en turnos. En este tipo de juegos la muerte es permanente, y al perder la partida se empieza en una nueva desde cero.
- *Wizardry: Proving Grounds of the Mad Overlord* (Sir-Tech Software, 1981), que dio lugar a los videojuegos de exploración de mazmorra o *dungeon crawlers*, similares a los anteriores, pero centrados en la exploración de la mazmorra, con un énfasis en la progresión de los personajes y de la gestión de la *party* o escuadrón (el conjunto de personajes que juntos intentan alcanzar objetivos comunes).
- *Ultima I: The First Age of Darkness* (Origin Systems, 1981), que dio lugar a los RPG de mundo abierto. En esta clase de juegos, los jugadores pueden explorar libremente ciudades, mazmorras, bosques y otro tipo de entornos, manteniendo las mecánicas tradicionales de otros RPG (ver figura 3.3b).

Con el salto tecnológico que hubo a mediados de la década de los 80, los RPG comienzan a separarse cada vez más de ser meras adaptaciones de juegos ya existentes a desarrollar sus propias historias. A partir de esta época se pueden encontrar dos corrientes bastante diferenciadas de RPG, los «occidentales» (WRPG, *Western role-playing game*), con más libertad de decisión para los jugadores tanto en personalización como en la historia, y con temáticas realistas (como los anteriormente mencionados *Rogue*, *Wizardry* y *Ultima*); y los «orientales» o «japoneses» (JRPG,

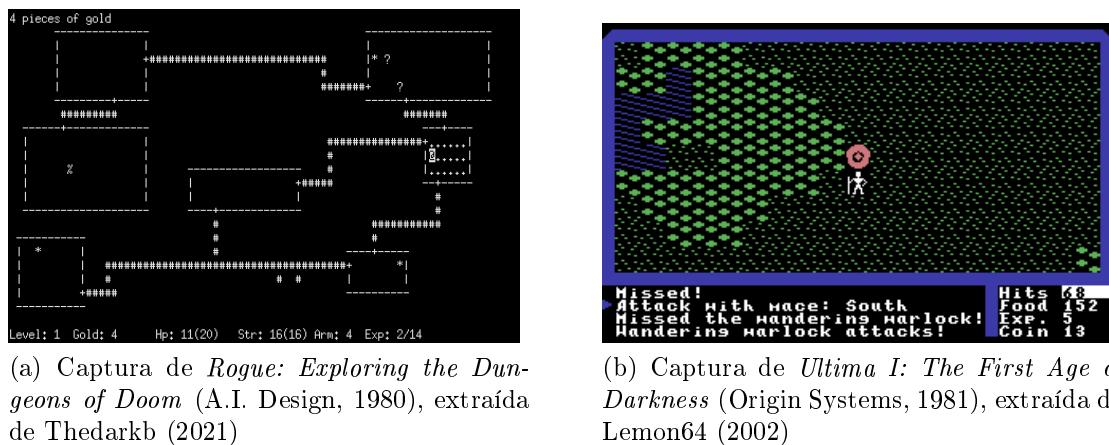


Figura 3.3: Capturas de *Rogue* y *Ultima*.

Japanese role-playing game, por ser Japón el país que más videojuegos de este tipo produce), centrados en una narrativa lineal con temáticas fantásticas y mecánicas basadas en turnos. Dos grandes videojuegos que definieron el género de los JRPG son *Dragon Quest* (Chunsoft, 1986) y *Final Fantasy* (SquareSoft, 1987), cuyas sagas permanecen hasta la actualidad con nuevas entregas cada pocos años. Son estos años de apogeo de los RPG los que Barton denomina como «etapa dorada».

La década de los 90 supuso un grave declive para los RPG «occidentales», ya que la aparición de juegos de acción en 3D, como *Doom* (id Software, 1993), *Quake* (id Software, 1996) o *Tomb Raider* (Core Design, 1996), hizo que el mercado cambiase hacia este tipo de juegos, mucho más rápidos y frenéticos que los RPG, que se consideraban obsoletos, con una pesada carga textual y mucho más lentos de jugar. También, la aparición de videoconsolas mucho más potentes y baratas, como la *PlayStation* (Sony, 1994) o la *Nintendo 64* (Nintendo, 1996), para las cuales estos RPG no tenían portabilidad³, y los altos costes de desarrollo y producción que supuso el cambio de cartuchos tradicionales a nuevos formatos como CD-ROM, hicieron que los WRPG tuviesen este gran declive que solo pudo recuperarse a mediados de la siguiente década.

Este declive, sin embargo no afectó a los JRPG, ya que estos vivieron una etapa de auge y consolidación, especialmente gracias al éxito de las consolas mencionadas y a una fuerte apuesta por el mercado internacional. Sagas como *Final Fantasy* lograron gran reconocimiento global, en especial, con su séptima entrega, *Final Fantasy VII* (SquareSoft, 1997), que supuso un punto de inflexión gracias a su espectacular apartado gráfico para la época y a su narrativa cinematográfica.

Otro fenómeno destacado fue la aparición de la saga *Pokémon* (Game Freak, 1996), que redefinió el género al introducir mecánicas de colección y combate por turnos simplificado, orientado a un público mucho más joven, pero sin perder la

³Son muchas las decisiones por las cuales las empresas «occidentales» no quisieron portar sus juegos hacia las nuevas consolas, como por ejemplo las limitaciones que iba a suponer (algunos requerían teclado y ratón), los altos costes que conllevaba el rediseño de muchos de los juegos y la poca demanda que se consideraba que los juegos iban a tener en estas plataformas.

profundidad estratégica. Otros títulos como *Chrono Trigger* (SquareSoft, 1995), *Secret of Mana* (SquareSoft, 1993), o *Suikoden* (Konami, 1995) aportaron innovaciones tanto en jugabilidad como en narrativa. Gracias a esta explosión de creatividad y accesibilidad, los JRPG no solo sobrevivieron al cambio de tendencias de la industria, sino que se convirtieron en uno de los géneros más representativos de las consolas de 16 y 32 bits.

En el nuevo milenio, que para Barton es la «etapa de platino», surgen sagas con juegos con gráficos mucho más sofisticados y con narrativas mucho más profundas, como la saga *The Elder Scrolls*, más concretamente, su tercera entrega, *The Elder Scrolls III: Morrowind* (Bethesda Game Studios, 2002); la saga *Fallout: A Post Nuclear Role Playing Game* (Interplay Productions, 1997); la saga *Baldur's Gate* (BioWare, 1998); o la saga *Diablo* (Blizzard North, 1997), que llevan hasta el límite las propias definiciones del género por las mezclas con otros géneros (llegando a ser juegos «híbridos»). La potencia del *hardware* va en aumento, lo que permite que haya un salto cualitativo, tanto gráfico como en jugabilidad, mientras que la tendencia de los mundos abiertos continúa y se mejora, llegando a ser algunos RPG como *The Elder Scrolls V: Skyrim* (Bethesda Game Studios, 2011) o *The Witcher 3: Wild Hunt* (CD Projekt Red, 2015) los más vendidos⁴.

3.2. Sobre el desarrollo de videojuegos

Para entender cómo, bien las empresas profesionales o bien equipos *indies* desarrollan un videojuego desde cero, es imprescindible saber con exactitud cómo funciona el software internamente, y qué formas tienen los programadores o desarrolladores para comunicarse con las entrañas de este durante el proceso de desarrollo. Es por eso que en este apartado se explicará qué es un *motor* y qué es un *editor*.

3.2.1. Roles en el desarrollo de videojuegos

El desarrollo de un videojuego es una tarea multidisciplinar que requiere la colaboración de múltiples individuos expertos en múltiples áreas. Por lo general, los roles más comunes dentro de un equipo de desarrollo profesional⁵ son los siguientes:

- **Diseñador:** es el responsable de definir las mecánicas del juego, las reglas, los objetivos, la progresión y la experiencia general del jugador. También se encarga del diseño de los niveles, de los sistemas de juego o del equilibrio de la dificultad.
- **Programador:** se encarga de implementar técnicamente el videojuego utilizando lenguajes de programación y motores de desarrollo. Es común que haya

⁴Según la revista *TheGamer*, en un artículo del año 2021 (<https://www.thegamer.com/highest-selling-rpgs-all-time/>), sitúan a *The Witcher 3* como el cuarto juego más vendido con treinta millones de copias, al igual que a *The Elder Scrolls V*, que se sitúa en segundo lugar con las mismas ventas.

⁵En proyectos pequeños o empresas *indies* estas funciones pueden solaparse, ya que no siempre se puede contar con expertos en todas las materias.

programadores especializados en distintas áreas, como la inteligencia artificial, gráficos, físicas o red.

- **Artista:** se encarga de diseñar los elementos visuales del juego, como personajes, escenarios, animaciones y efectos visuales. Dependiendo del estilo del juego, puede haber artistas 2D, 3D, conceptuales (*concept artist*) o de interfaces.
- **Diseñador de sonido:** se encarga del diseño y producción de los efectos de sonido, música y ambiente del juego. A menudo colabora con compositores y actores de voz para dotar de identidad sonora al juego.
- **Guionista:** se encarga de crear la historia, los diálogos, el trasfondo y el desarrollo narrativo del juego. Es fundamental en aquellos géneros donde la narrativa tiene un peso importante, como es el caso de algunos RPG.
- **Probador:** su misión es detectar errores, evaluar la jugabilidad y asegurar que el producto final funcione correctamente. Esta fase se conoce comúnmente como QA (*quality assurance*, aseguramiento de la calidad), y es esencial para garantizar una buena experiencia de usuario.
- **Productor:** se encarga de coordinar al equipo, gestionar los tiempos, recursos y comunicación entre los distintos departamentos. Es el vínculo entre el equipo creativo y las partes interesadas externas (como clientes, editoras o inversores).
- **Director creativo:** supervisa la visión general del juego y asegura que todos los elementos (jugabilidad, arte, narrativa...) estén alineados con el concepto original.

3.2.2. Motor de videojuegos

Gregory (2018) define un motor de videojuegos (*game engine*) como «todo aquel *software* extensible que, sin apenas modificaciones, puedan servir de base o cimiento para múltiples videojuegos distintos». Este *software* es todo el conjunto de herramientas que hacen que por detrás funcione un juego, como por ejemplo, todas las herramientas que se encarguen del *renderizado* o dibujado en pantalla, bien sea en 2D o en 3D, aquellas que se encarguen de la simulación física y detección de colisiones con el entorno, las que se encarguen del sonido, las del *scripting*, animaciones, inteligencia artificial... A todas estas herramientas también se les denomina *motores* (por ejemplo, *motor de render*, *motor de físicas*...).

Estos *motores de tecnología* conforman la infraestructura básica técnica del motor más complejo que las usa, y se usan para abstraer la interacción con el *hardware* o sistema operativo; por lo que un motor de videojuegos también podría describirse como «una capa de abstracción y herramientas orientadas al desarrollo de videojuegos elaborada sobre *motores de tecnología*».

Debido a las limitaciones tecnológicas de los años 70 y principios de los 80, los primeros juegos se desarrollaban todos desde cero, sin apenas compartir código un

juego con otro, ya que cada uno necesitaba una lógica optimizada de una determinada manera que otros no necesitaban o no podían utilizar. Además, los juegos solían ser lanzados para una única plataforma, ya que portar un juego a otra distinta con otra serie de requisitos y limitaciones implicaba reescribirlo desde cero, y muchas veces los desarrolladores eran equipos muy pequeños (llegando a ser incluso de una única persona) y no todas las grandes empresas disponían de departamentos de portabilidad.

No es hasta mediados y finales de la década de los 80 cuando los desarrolladores comienzan a reutilizar código entre juegos y surgen los primeros ejemplos de lo que hoy se podría llamar «motor». Uno de los primeros fue el que Shigeru Miyamoto desarrolló en Nintendo para la *Nintendo Entertainment System* (según explica Williams (2017)), que se utilizaría en juegos como *Excitebike* (Nintendo, 1984) o *Super Mario Bros.* (Nintendo, 1985).

A principios de los 90 surgen los primeros «motores modernos», de la mano de desarrolladoras como *id Software* y juegos como *Doom* o *Quake*, quienes decidieron reutilizar toda la lógica de *renderizado* y los sistemas de simulación física, ya que cada parte estaba desarrollada de manera independiente. Tal fue el éxito que alcanzaron estos dos juegos que muchas empresas prefirieron pagar a *id Software* por una licencia del núcleo del motor y diseñar sus propios recursos, que desarrollar su propio motor desde cero. Una de estas empresas fue *Valve*, quienes desarrollaron uno de los mejores juegos de la historia, *Half-Life* (Valve, 1998), utilizando el motor *GoldSrc* (que es el antecesor del actual motor de Valve, *Source*), una versión modificada del motor de *id Software*.

Con la generalización de internet a principios de los 2000, comenzó el auge de comunidades de *modding* en línea, y muchas desarrolladoras comenzaron a lanzar motores de código abierto acompañados por editores de niveles o herramientas de *scripting* (es decir, código de alto nivel, normalmente no compilado, que solo modifica lógica del juego o eventos sin modificar el núcleo del motor).

A día de hoy, las empresas dedican numerosos recursos a la hora de desarrollar nuevos motores, ya que son la parte fundamental del desarrollo de videojuegos, y cada vez son más sofisticados y requieren de un gran conocimiento en programación o en diseño.

Los desarrolladores *indie*, por su parte, tienen la posibilidad de poder desarrollar sus propios motores, cuyo contenido no es equiparable al de empresas que producen juegos *triple A* (aquellos juegos producidos por grandes empresas a los que se suelen destinar un alto presupuesto en desarrollo y publicidad); usar motores propietarios de empresas con licencias gratuitas o de poco coste, como por ejemplo *Unity* o *Unreal Engine*; o motores de código abierto, como *Godot*. Por lo general, estos motores suelen venir acompañados de un editor, que facilita el desarrollo de los juegos implementados para ese motor.

3.2.2.1. Componentes de un motor de videojuegos

Cada motor de videojuegos es distinto, y cada uno incorpora según qué motores de tecnología dependiendo de las necesidades de los programadores o del juego que

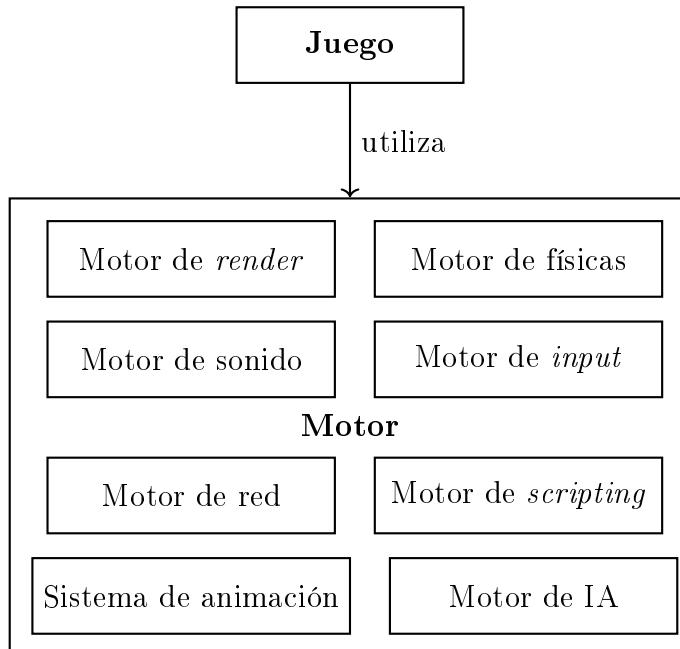


Figura 3.4: Representación esquemática de la estructura de un juego y su motor con algunos de los componentes principales

se esté programando. Siguiendo el esquema provisto en la figura 3.4, estos son los componentes principales de un motor de videojuegos tanto de grandes empresas, como motores con licencia gratuita, como aquellos de código abierto:

- **Motor de render o de dibujado:** se encarga de gestionar todas las tareas relacionadas con los gráficos que se muestran en pantalla. Para ello, dibuja objetos bidimensionales o tridimensionales, representados generalmente mediante «mallas», a través de técnicas de informática gráfica, como pueden ser la *rasterización* o el trazado de rayos. También es el encargado de gestionar la cámara, luces, sombras, materiales y texturas, y puede aplicar diversos efectos de posprocesado al fotograma final. Muchas de estas tareas las puede definir el propio programador haciendo uso de un tipo de *script* especial llamado *shader*, que en lugar de ejecutarse en el procesador de un ordenador se ejecuta en el procesador de las tarjetas gráficas. Estos motores suelen ser una capa de abstracción sobre especificaciones estándar para gráficos como *OpenGL*, *Vulkan*, *DirectX* o *Metal*.
- **Motor de físicas:** se encarga de simular comportamientos físicos realistas. Entre estos comportamientos físicos se encuentran las colisiones de objetos con otros objetos o con el entorno, aplicar la fuerza de gravedad a unos determinados objetos, simular las dinámicas de cuerpos rígidos (es decir, el movimiento de cuerpos interconectados bajo la acción de una fuerza externa, como por ejemplo, cajas que se pueden tirar, deslizar o romper), simular dinámicas de cuerpos blandos (similares a los cuerpos rígidos, pero con la posibilidad de que estos se deformen), y, en algunos casos, simulaciones de fluidos y de materiales textiles. Ya que hacer una simulación física es complicado, se suelen

utilizar motores de terceros, como por ejemplo *NVIDIA PhysX*, *Havok*, *Bullet* o *Box2D*.

- **Motor de sonido:** se encarga de gestionar los efectos de sonido, la música, el audio espacial o posicional en dos o tres dimensiones, y todos los efectos sonoros que se pueden aplicar al sonido (reverberación, eco, tono, barrido, mezclado de pistas, oclusión sonora, efecto Doppler, etc...). Algunas librerías utilizadas son *FMOD*, *OpenAL*, *Wwise* o *irrKlang*.
- **Motor de *input* o de gestión de la entrada:** se encarga de capturar y procesar eventos de entrada (*input*) del usuario a través de los diversos dispositivos para los que se programe (teclado y ratón, mandos o controladores y pantallas táctiles) y asociarlos a las acciones definidas en el juego.
- **Sistema de animación:** gestiona animaciones de *sprites* (los elementos gráficos bidimensionales básicos que representan visualmente objetos dentro del juego) o de esqueletos (*rigging*, usados para personajes tridimensionales). Suele tener soporte para árboles de mezcla y cinemática inversa.
- **Motor de *scripting*:** permite escribir lógica específica del juego utilizando lenguajes de programación de alto nivel (como por ejemplo JavaScript, Lua, C# o Python), separando la implementación de la jugabilidad de la implementación del motor. El *scripting* se suele hacer mediante lenguajes de programación interpretados y no compilados, por lo que los cambios más pequeños no requieren volver a iterar por todo el proceso de compilado del motor o del juego.
- **Motor de red:** se encarga de gestionar el soporte multijugador, es decir, la comunicación y sincronización cliente-servidor o cliente-cliente. El motor de red también incluye sistemas de emparejamiento (*matchmaking*) y de predicción o interpolación del juego para una simulación fluida en los juegos multijugador.
- **Motor de inteligencia artificial:** ofrece herramientas para poder crear comportamientos inteligentes artificiales, como por ejemplo, algoritmos de búsqueda de caminos (*pathfinding*, como los algoritmos *A** o el de Dijkstra), árboles de comportamiento y máquinas de estado, o sistemas de toma de decisiones. Este motor suele tener una conexión directa con el motor de *scripting* para poder definir comportamientos mucho más fácilmente.
- **Gestor de recursos:** maneja la carga y descarga de recursos como texturas, mallas, sonidos o animaciones, muchas veces bajo demanda del juego mediante gestores de memoria, compresión y de transmisión de datos altamente optimizados.
- **Sistema de interfaz de usuario:** se encarga de gestionar la barra de estado (HUD, *head-up display*), los menús, texto, botones y otros elementos de la interfaz con los que el usuario pueda interactuar.

3.2.2.2. Patrones ECS y EC

En cuanto a los patrones arquitectónicos utilizados en el diseño de motores de videojuegos, uno de los más usados es el patrón ECS (entidad, componente, sistema), cuya principal ventaja es la separación total entre los datos y el comportamiento. Este patrón busca eliminar los problemas de herencia profunda y acoplamiento que surgen con el enfoque clásico de la orientación a objetos, y es especialmente útil en contextos donde se manejan muchos objetos heterogéneos.

El patrón ECS se basa en tres elementos fundamentales:

- Entidades: son identificadores únicos que representan objetos del juego (por ejemplo, jugadores, enemigos, proyectiles...). Por sí solas, las entidades no contienen ni datos ni comportamiento.
- Componentes: son estructuras de datos puras que contienen información específica de una entidad, como su posición, velocidad, salud, animación o inteligencia artificial. Cada entidad puede tener múltiples componentes.
- Sistemas: son unidades lógicas encargadas de procesar todas las entidades que contienen un conjunto específico de componentes. Por ejemplo, un sistema de física puede procesar todas las entidades que tengan componentes de posición y velocidad, actualizando su movimiento en función del tiempo.

En algunas implementaciones mucho más sencillas, se eliminan los sistemas, dejando únicamente a las entidades y los componentes como base de la arquitectura. Esta implementación, conocida como patrón EC, es mucho menos escalable y modular que el patrón ECS, pero es más próxima a la orientación a objetos. En este caso, son las propias entidades o los componentes los que se encargan del procesado de la lógica en lugar de los sistemas.

3.2.2.3. Bucle de juego

El bucle de juego (*game loop*) es una estructura central en cualquier motor de videojuegos. Su propósito es mantener el flujo continuo de ejecución del juego, actualizando su estado y renderizando el resultado en pantalla de forma cíclica. Se trata de un patrón esencial que permite que el juego responda a las entradas del usuario, avance en el tiempo, y se represente visualmente de forma coherente.

Típicamente, un bucle de juego sigue los siguientes pasos en cada iteración (o fotograma):

1. Procesamiento de entrada: se recogen y procesan las entradas del jugador, ya sea desde el teclado, ratón, *gamepad*, red...
2. Actualización del estado del juego: se actualiza la lógica del juego, incluyendo la física, la inteligencia artificial, las colisiones y cualquier otro sistema relevante.
3. Renderizado: se genera la imagen correspondiente al nuevo estado del juego y se envía a la pantalla.

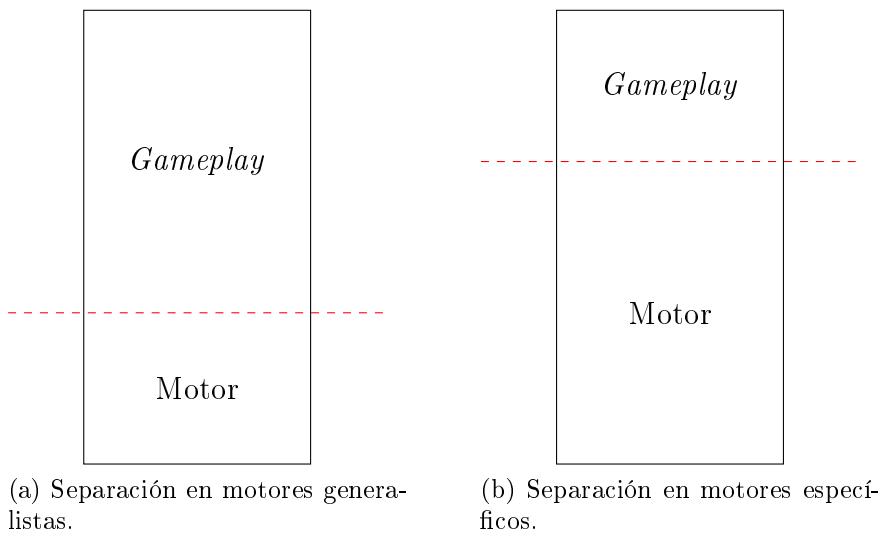


Figura 3.5: Ejemplos de separaciones entre motor y *gameplay* en los distintos motores.

Este ciclo se repite de forma constante durante toda la ejecución del juego. En juegos modernos, suele buscarse una tasa de refresco constante (por ejemplo, 60 fotogramas por segundo), aunque también pueden implementarse bucles desacoplados, donde la actualización lógica y el *renderizado* ocurren a distintas frecuencias para mejorar el rendimiento o la estabilidad.

Existen variantes del bucle de juego adaptadas a distintas arquitecturas o estilos de motor. Por ejemplo, en motores basados en el patrón ECS, los sistemas se ejecutan durante la fase de actualización para aplicar transformaciones a las entidades y sus componentes.

3.2.2.4. Separación entre motor y *gameplay*

Para poder reusar un motor en múltiples juegos con el menor número de cambios, es necesario saber *cómo y para qué vamos a desarrollar el motor*, y de acuerdo a la decisión que se haya tomado, se establecerá la barrera de separación entre el motor y la jugabilidad del juego.

Los motores más generalistas, como los anteriormente mencionados *Unity*, *Unreal Engine* o *Godot*, debido a que están pensados para poder ejecutar todo tipo de juegos, tienen una barrera de separación entre el motor y el *gameplay* que se suele establecer a la mitad. La implementación del juego final que tienen este tipo de motores es mínima (el motor es puramente un conjunto de *motores de tecnología*) y es el desarrollador, o a veces el diseñador⁶, el que se tiene que encargar del desarrollo de la jugabilidad. En la figura 3.5a se puede ver esta «separación» de manera esquemática, donde la parte de la jugabilidad es mucho más amplia que la del motor,

⁶En muchos casos, hay veces que los propios diseñadores también pueden implementar parte del juego sin la necesidad de tener conocimientos de programación. Para ello, existen los editores, que se explicarán en la sección 3.2.3.

que solo incluye elementos básicos para que un juego funcione.

La gran ventaja que tienen los motores generalistas frente a los específicos es la libertad que dejan al desarrollador (o diseñador) para poder implementar la jugabilidad a su manera, con las reglas y mecánicas que se hayan establecido previamente.

En el caso de aquellos motores que fijan gran parte de la jugabilidad de los juegos que se puedan hacer con ellos, y que lo único modificable son algunos parámetros del *gameplay* y las partes artísticas y visuales, se tendrán que introducir elementos propios del juego en el motor, dejando menos libertad de implementación en el *gameplay*. En la figura 3.5b se aprecia cómo la barrera de separación entre la implementación del juego que contiene el motor frente a la de la jugabilidad es mucho más alta que en la figura 3.5a, por lo que el motor ya no solo incluirá elementos básicos para el funcionamiento del juego, sino elementos específicos de un tipo de juego en concreto.

Este tipo de motores son mucho más rígidos a la hora de poder implementar nuevas funcionalidades y limitan bastante los juegos que se pueden crear con ellos, pero simplifican bastante el desarrollo, y son especialmente útiles para nuevos programadores.

3.2.2.5. Programación dirigida por datos (DDP) en videojuegos

La programación dirigida por datos (DDP, *data-driven programming*) es un paradigma de diseño en el que gran parte del comportamiento y lógica de un programa están controlados por datos externos en lugar de estar programados en el código fuente de este. Este paradigma, según Gregory, ampliamente extendido entre las empresas desarrolladoras de juegos *triple A*, permite a los desarrolladores modificar o expandir el comportamiento del juego sin la necesidad de alterar el motor o el código base de este.

Los motores de videojuegos suelen estar programados utilizando lenguajes de *medio nivel*⁷, ya que se espera poder optimizar al máximo el programa, así como poder tener un soporte multiplataforma. De todo el gran abanico de lenguajes de nivel medio, un gran porcentaje son lenguajes compilados, es decir, requieren de un «traductor» (compilador) para generar el código máquina antes de poder ejecutar el programa. Esta compilación depende del tamaño del proyecto y del número de ficheros a compilar, y, en el caso de muchos motores, la compilación puede llegar a tardar decenas de minutos.

Para evitar tener que esperar estos minutos recompilando todo un juego en caso de cambiar un simple parámetro referente al *gameplay*, se opta por la solución más flexible, que es desarrollar todo el juego utilizando datos, generalmente en lenguajes de *scripting*, de alto nivel, como Lua, que no necesitan ser compilados, sino interpretados (la «traducción» a lenguaje máquina se realiza en ejecución del programa y a medida que sea necesaria) por el motor del juego.

Además de reducir los tiempos de compilación en los juegos, lo cual ayuda a re-

⁷Los lenguajes de *medio nivel*, como C o C++, son aquellos lenguajes de alto nivel que proporcionan estructuras de acceso a *hardware* propias de los lenguajes de bajo nivel.

ducir los tiempos en las iteraciones de desarrollo, la programación dirigida por datos permite, tanto a desarrolladores como a diseñadores, modificar comportamientos, ajustar parámetros o añadir contenido utilizando archivos de configuración o mediante herramientas visuales.

Este paradigma refuerza la ya mencionada separación entre motor y *gameplay*, ya que un mismo motor puede ejecutar distintos juegos (es decir, distintos datos) sin la necesidad de tener que ser recompilado, siempre y cuando los datos estén en un formato y estructura que el motor sea capaz de interpretar.

Los datos se suelen almacenar utilizando lenguajes de definición de datos (DDL, *data definition language*), que permiten describir la estructura y organización de estos que el motor o el sistema de *scripting* van a usar. Estos lenguajes, como JSON, XML o incluso formatos binarios diseñados específicamente, que permiten mantener una clara separación entre la lógica del juego y los contenidos, facilitando, a su vez, la escalabilidad del proyecto.

Aunque DDP se asocia al uso de motores de videojuegos, su implementación no depende estrictamente de estos. De hecho, es perfectamente posible aplicar DDP sin utilizar un motor formal, siempre que exista una arquitectura que permita interpretar y ejecutar los datos de forma dinámica. Muchos desarrolladores independientes o juegos clásicos sin motor como tal ya hacían uso de este enfoque mediante archivos de configuración, tablas o *scripts* embebidos.

Un ejemplo histórico de DDP puede encontrarse en el motor *SCUMM (Script Creation Utility for Maniac Mansion)*, desarrollado por Lucasfilm Games en los años 80 para facilitar el desarrollo de su aventura gráfica *Maniac Mansion* (Lucasfilm Games, 1987). Este motor permitía definir el comportamiento de escenas y personajes mediante *scripts* externos, lo que habilitaba la creación de aventuras gráficas sin necesidad de reescribir el código base. Posteriormente, el motor *GrimE (Grim Engine)*, también desarrollado por LucasArts⁸ y utilizado en juegos como *Grim Fandango* (LucasArts, 1998), amplió este enfoque incorporando Lua como lenguaje de *scripting*, mostrando una evolución clara hacia un diseño aún más flexible y dirigido por datos.

Para poder conectar la programación dirigida con datos con el propio código del motor o del videojuego se suele usar un patrón de diseño creacional, llamado patrón factoría (*factory pattern*), que proporciona una interfaz para crear objetos en una superclase permitiendo a las subclases alterar el tipo de objeto que se crea. De esta manera, se consigue desacoplar el proceso de creación de objetos del código que los utiliza.

3.2.2.6. Programación multiplataforma en videojuegos

La programación multiplataforma es una práctica esencial en el desarrollo de videojuegos modernos. Esta característica permite que un mismo juego funcione idénticamente en distintos dispositivos y sistemas operativos (por ejemplo, en el

⁸Lucasfilm Games cambiaría su nombre a LucasArts en 1990, y, posteriormente, en 2021, lo volvería a cambiar por Lucasfilm Games.

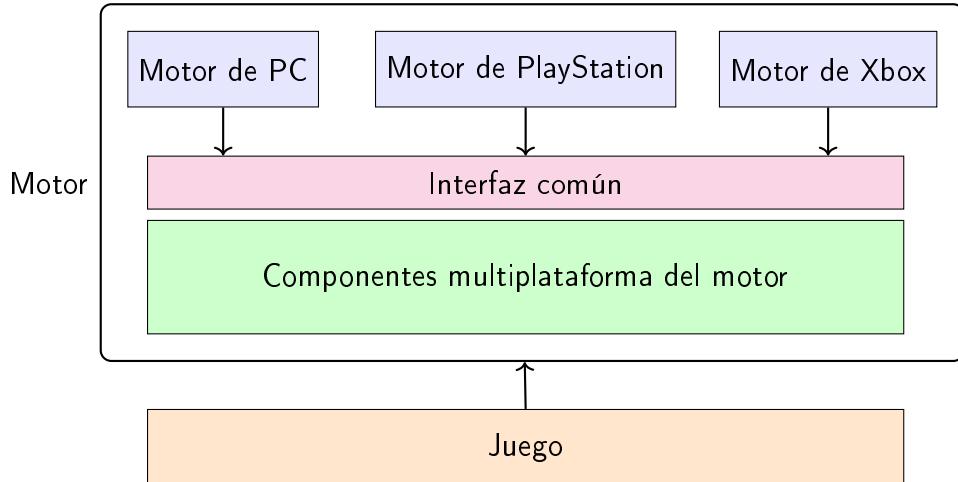


Figura 3.6: Representación esquemática de la arquitectura de un motor multiplataforma

caso de los videojuegos, que funcione en ordenadores Windows, Linux o Mac; en consolas, como PlayStation, Xbox o Nintendo Switch; o en dispositivos móviles, como Android o iOS). Pese a que aumenta las labores de desarrollo del motor (ya que en muchos casos hay que desarrollar módulos específicos para cada una de las plataformas, principalmente de *renderizado*), reduce los costes de mantener un juego y maximiza el alcance de este.

Para lograr que un motor sea multiplataforma, hay que tener en cuenta para qué plataformas se va a desarrollar y qué diferencias hay entre cada una de ellas. Es por ello que se suelen utilizar capas de abstracción, que permiten que el núcleo del motor permanezca intacto entre las distintas implementaciones, mientras que son aquellos módulos que difieren de un sistema a otro los que se modifican.

Otro factor a tener en cuenta es la compilación cruzada, es decir, el proceso por el cual el código se compila en una plataforma (principalmente un ordenador con Windows o Linux), pero el ejecutable se genera para una plataforma distinta (por ejemplo, para Android o para PlayStation). Para ello, se deben disponer de herramientas que faciliten la configuración de estas compilaciones.

Los principales desafíos en la programación multiplataforma de videojuegos son debidos a las diferencias entre las distintas API (*application programming interface*, interfaz de programación de aplicaciones, es decir, todo el código que permite a dos aplicaciones comunicarse entre sí) de las videoconsolas, tanto en el *render*, como en el manejo de archivos; o las compatibilidades que una librería pueda tener en un sistema o en otro.

En la figura 3.6 se puede apreciar el esquema de una arquitectura multiplataforma: por una parte, los nodos azules corresponden a las implementaciones de los *motores de tecnología*, cada una distinta dependiendo de los requerimientos de cada una de las plataformas; el nodo rosa correspondería a la interfaz común de cada uno

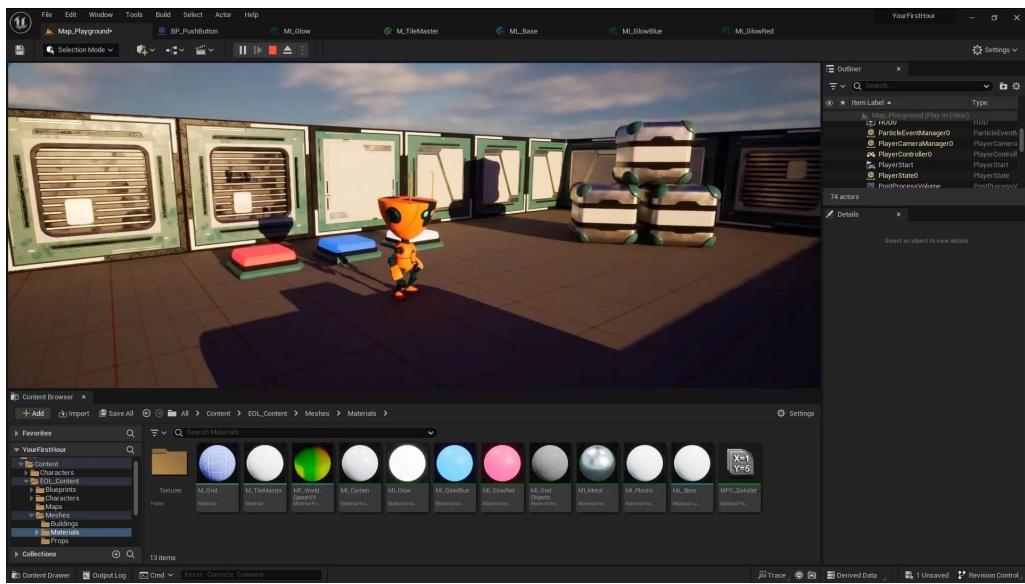


Figura 3.7: Vista de la interfaz de *Unreal Engine 5.2*, extraída de Wadstein (2023).

de los *motores de tecnología*⁹; el nodo verde corresponde a toda aquella parte del motor que no depende de ninguna plataforma; y finalmente, el nodo naranja es el juego, que utiliza todo el conjunto del motor para poder funcionar.

3.2.3. Editor de videojuegos

Un editor de videojuegos es una herramienta, generalmente visual, que permite crear, modificar y probar contenido de un juego sin la necesidad de programar directamente en código. Un editor también es especialmente útil para *no programadores*, es decir, diseñadores o artistas que pueden estar involucrados en el desarrollo del juego sin necesariamente tener que saber programar.

Por lo general, los editores como el de *Unreal Engine* permiten al usuario diseñar y editar los niveles, mapas o escenarios sobre los que se desarrolla la acción; editar entidades o actores, como personajes, enemigos o NPC; editar *scripts* o crear eventos que definen el comportamiento o la jugabilidad; gestionar y editar recursos como materiales, texturas, partículas o animaciones; y poder ejecutar y depurar una versión del juego para probar y solucionar problemas sin la necesidad de generar un ejecutable final.

Una característica esencial de los editores es un sistema de «hacer-deshacer-rehacer», que permite a los desarrolladores experimentar con cambios sin riesgo a perder el progreso.

En el editor de *Unreal Engine*, mostrado en la figura 3.7, apreciamos varias de

⁹Es decir, tanto el motor como el juego deben poder llamar a funciones y clases que estén implementadas de distinta forma en cada uno de los *motores de tecnología* y poder cambiar de plataforma sin que haya ningún error. Para ello se define esta interfaz, que contendrá las definiciones de los nombres pero no su implementación.

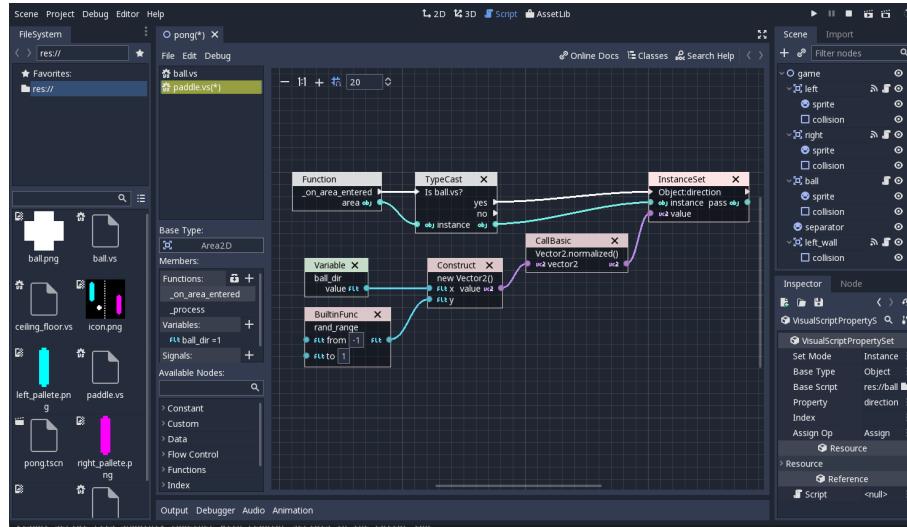


Figura 3.8: Vista del *scripting* visual mediante nodos de *Godot*, extraída de Linietsky y Manzur (2022).

estas características, como la edición de materiales y texturas (cada una de las esferas que se muestran en el cajón inferior representa un material o textura que se puede aplicar a los objetos en la escena), un visor de las entidades que se encuentran en la escena (en la parte derecha, que permite la edición individual de cada una de las entidades, bien sean parte del escenario o personajes y la propia escena (o *viewport*, en grande, en el centro del editor, en la cual el usuario puede mover libremente la cámara y mover, rotar y escalar cada objeto).

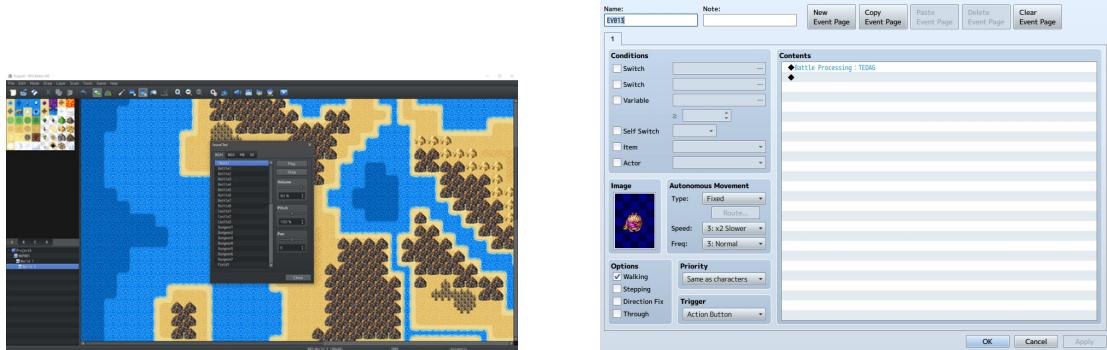
El editor se aprovecha de los principios de la programación dirigida por datos (ver apartado 3.2.2.5), haciendo posible que los cambios realizados por el usuario se reflejen en el juego sin la necesidad de recompilar el motor ni generando un ejecutable externo. Esto permite poder ejecutar el juego directamente desde el propio editor (*Unreal Engine* lo denomina *Play In Editor*), lo cual agiliza las iteraciones durante el desarrollo.

Es importante destacar que el editor no es el motor del juego en sí, aunque está estrechamente ligado a él y hace uso directo de muchas de sus funcionalidades. Mientras que el motor es el componente responsable de ejecutar el juego, el editor actúa como una capa de interfaz que permite manipular los datos que el motor utiliza. Por ejemplo, al colocar un objeto en una escena desde el editor, lo que realmente ocurre es que se modifica un conjunto de datos que el motor luego interpretará en tiempo de ejecución.

Algunos editores modernos suelen ofrecer herramientas de *scripting* visual, como es el caso de los *blueprints* en *Unreal Engine* o el sistema de nodos en *Godot* (ver figura 3.8), que permiten implementar la lógica sin tener que escribir código tradicional. Estas herramientas, junto con el patrón ECS (*entity-component-system*, entidad-componente-sistema), una importación automatizada de los recursos, y el poder extender la funcionalidad del editor mediante *plugins* (complementos o extensiones), convierten a los editores modernos en los ejes centrales del flujo de trabajo.

en el desarrollo de videojuegos.

3.2.3.1. Editores específicos de videojuegos para desarrollo de RPG



(a) Vista de la interfaz de *RPG Maker MZ* (Gotcha Gotcha Games, 2020).

(b) Vista del editor de eventos de *RPG Maker MV*, extraída de Cenance (2019).

Figura 3.9: Capturas de distintos elementos de la interfaz de *RPG Maker*.

Si bien es cierto que una gran parte de los juegos más jugados a diario son RPG¹⁰, hay muy pocos motores y editores específicos para este tipo de juegos, y se tiende a usar motores más generalistas. De entre los específicos, necesariamente tenemos que mencionar a la serie de herramientas más conocida de todas, *RPG Maker* (ASCII, 1992)¹¹. Se trata de un entorno de desarrollo completo, que combina un motor de ejecución con un editor visual, pensado para permitir a usuarios sin conocimientos avanzados de programación crear juegos completos mediante una interfaz sencilla (ver figura 3.9a).

El editor incluye un sistema de creación de mapas basado en baldosas, un generador de personajes y una base de datos para poder definir los enemigos, objetos, habilidades, clases, estados y eventos mediante lógica visual (figura 3.9b).

Este sistema de eventos es una de las características más potentes y distintivas de *RPG Maker*, ya que permite definir la lógica utilizando una lista secuencial de comandos que representan *acciones*¹² y que puede ser utilizado por usuarios sin experiencia alguna, así como otros más curtidos en la materia que pueden crear sistemas más complejos.

También se incluye un sistema de batallas tradicional, en el cual los combatientes se van turnando, inspirado en los JRPG clásicos, aunque es posible modificarlo

¹⁰Según la página SteamDB, y, a día 21 de mayo de 2025, un 34 % de los juegos jugados a diario, solamente en PC, son RPG (<https://steamdb.info/stats/globaltopsellers/?displayOnly=Game&tagId=122>).

¹¹Según itch.io, *RPG Maker* se sitúa como la octava herramienta más utilizada, siendo la primera específica de RPG que aparece en el listado, con más de diez mil proyectos creados a día 21 de mayo de 2025 y subidos a su web (<https://itch.io/game-development/engines/most-projects>).

¹²Las acciones son instrucciones específicas dentro de un sistema de eventos. Estas acciones son «bloques» con instrucciones básicas que se ejecutan secuencialmente para definir la lógica del juego, como por ejemplo, *mostrar diálogo*, *mover personaje* o *cambiar una variable*.

Característica	RPG Maker	RPG Paper Maker	RPG JS	EasyRPG	OHRRPGCE
Licencia	Comercial	Gratuita	Código abierto	Código abierto	Código abierto
Exporta a	PC ¹³ , Web, Android e iOS	PC, Web	Web	PC, Android, iOS, Web	PC, Android
Gráficos	2D	3D retro	2D	2D clásico	2D clásico
Scripting	JavaScript (MV/MZ)	JavaScript	TypeScript y HTML5	Eventos visuales	HamsterSpeak
Facilidad de uso	Muy fácil	Fácil	Compleja	Muy fácil	Media
Personalización	<i>Plugins/Scripts</i>	<i>Scripts</i>	Código fuente	<i>Scripts</i>	<i>Scripts</i>
Desarrollo activo	Sí	Sí	Parcial	Sí	Sí
Sistema de combate	Personalizable	Personalizable	-	Por turnos	Por turnos

Tabla 3.1: Comparativa de características entre diferentes editores RPG.

utilizando *plugins* o *scripts* (desarrollados en las últimas versiones en JavaScript), así como la posibilidad de poder exportar los juegos a múltiples plataformas (desde las últimas versiones), como PC, web y dispositivos móviles.

El editor de *RPG Maker* permite definir todos estos elementos mencionados anteriormente de forma visual, mientras que el motor es el que se encarga de ejecutar el resultado final. Ambos están estrechamente integrados, permitiendo una experiencia fluida durante el desarrollo, sin la necesidad de recurrir a herramientas externas ni a complejos procesos de compilación.

Además de *RPG Maker*, existen varias alternativas de código abierto, como *RPG Paper Maker* (Wano, 2018), que permite crear RPG en 3D con estética retro. Otras opciones, como *RPG JS* (Ronce et al., 2020) permiten desarrollar los juegos utilizando lenguajes como TypeScript o HTML5 para navegadores¹⁴.

También encontramos proyectos de código abierto como *EasyRPG*, que busca recrear los contenidos de *RPG Maker 2000/2003* de manera gratuita, facilitando la ejecución de RPG antiguos que ya no tienen soporte en plataformas modernas así como la creación de nuevos juegos sin la necesidad de utilizar un software propietario; o *Official Hamster Republic Role Playing Game Construction Engine* (Paige y Fisher, 1998) (conocido como OHRRPGCE), que permite crear juegos al estilo de los primeros juegos de la saga *Final Fantasy*¹⁵.

Por último, en la tabla 3.1, se encuentra una comparativa más resumida entre los distintos *softwares* anteriormente mencionados con algunas de sus características más importantes.

¹³En la tabla, para abreviar, PC se refiere tanto a Windows, MacOS como Linux.

¹⁴<https://docs.rpgjs.dev/guide/get-started.html>.

¹⁵<https://rpg.hamsterrepublic.com/ohrrpgce/About>.

Capítulo 4

Planteamiento del proyecto

RESUMEN: En este capítulo se tratarán los objetivos principales del proyecto, así como las decisiones tomadas en cuanto a diseño del motor y del editor.

4.1. Objetivos principales de RPGBAKER

La idea principal es el desarrollo de un motor de videojuegos, enfocado a los RPG 2D, acompañado de un editor que permita un desarrollo rápido y simple de juegos de este tipo para el motor desarrollado. El editor está pensado principalmente para gente no programadora o sin experiencia en el desarrollo de videojuegos, por lo que la interfaz tiene que ser intuitiva y fácil de utilizar y aprender.

El editor debe ser capaz de generar un ejecutable, que por debajo utilice el motor desarrollado previamente, con el diseño de juego realizado por el usuario en el propio editor, y que pueda ejecutarse en Windows, MacOS, Linux y Android. El editor, por su parte, ha de poder ser ejecutado tanto en Windows, MacOS o Linux, descartando la ejecución en dispositivos móviles¹.

El usuario podrá elegir la plataforma para la cual se va a generar el ejecutable, y el editor se encargará de transferir el contenido desarrollado en el proyecto a la *build*, asegurándose de que el comportamiento volcado es el mismo que el diseñado previamente y generando una *build* lista para empaquetar y distribuir, sin requerir pasos adicionales por parte del usuario.

Por otra parte, se espera que el editor pueda generar diversos proyectos (es decir, distintos juegos), y sea capaz de guardar el estado de un proyecto y recuperarlo cuando el usuario desee, sin que se hayan perdido los cambios que se hayan realizado. Y, también, debe poder exportar proyectos, así como importarlos incluyendo aquellos que otros usuarios puedan haber diseñado en otras plataformas o sistemas sin mayor dificultad.

El motor, por su parte, aportará la mayor parte del *gameplay*, para que el usuario solo tenga que desarrollar la parte de diseño (principalmente el diseño artístico y

¹Android no se utiliza como sistema operativo en el que se puedan desarrollar aplicaciones. Es más, ni siquiera las propias APK de Android se desarrollan en Android.

visual). Tendrá que tener las funcionalidades básicas que se esperan de un RPG, así como soporte para periféricos de entrada/salida tradicionales (teclado y ratón) y entrada táctil (para los dispositivos móviles).

4.2. Toma de decisiones

La primera decisión a tomar fue la plataforma de desarrollo de RPGBAKER. Se decidió desarrollar el grueso en C++, ya que se quería aprovechar el alto rendimiento que ofrece en comparación a otros lenguajes², el soporte multiplataforma que tiene la familia C/C++ tanto en dispositivos de sobremesa como en móviles, y el uso de librerías más avanzadas que facilitarían el desarrollo del trabajo.

Debido a la premisa de un desarrollo multiplataforma, se necesitaba usar un IDE (*integrated development environment*, entorno de desarrollo integrado) que fuese compatible tanto con compiladores específicos de Windows, MacOS, y Linux. La opción que en un principio se había valorado era la de utilizar *Visual Studio*, una de las herramientas más populares para el desarrollo en C++; sin embargo, la configuración que ofrece *Visual Studio* con otros compiladores es muy limitada y complicada, por lo que se optó por hacer el desarrollo de RPGBAKER en *CLion*, un IDE con soporte para CMake, que facilitaría a la hora de agilizar el trabajo (por su rapidez en la generación de proyectos complejos) y con la gestión de las dependencias externas.

Pese a que el aprender a usar CMake ocupó gran parte del inicio del desarrollo, las ventajas que ha supuesto a la hora del manejo de las distintas dependencias externas frente a otras alternativas que se habían manejado a lo largo del transcurso del Grado, han hecho que la inversión temporal en esta opción haya resultado beneficiosa.

Por otra parte, en el desarrollo del motor, se tendría que utilizar otra herramienta para el desarrollo de la APK (*Android Application Package*, paquete de aplicaciones Android, es decir, el ejecutable de Android), ya que esta se debe desarrollar utilizando Java. Por esto se decidió utilizar *Android Studio*, la herramienta oficial de desarrollo para Android, que proporciona máquinas virtuales de dispositivos Android con distintas versiones y generaciones para poder probar la *build*. Otra ventaja añadida al uso de Android Studio es el soporte que tiene para CMake, que ha permitido disponer de un único archivo de configuración para ambas herramientas.

Dentro de *Android Studio*, se tiene que añadir también el módulo de NDK (*Native Development Kit*, kit de desarrollo nativo) que permite el desarrollo de aplicaciones para Android utilizando llamadas a C/C++ gracias a JNI (*Java Native Interface*, interfaz nativa de Java) integrada en el SDK de Java. JNI es un ejemplo de una FFI (*foreign function interface*, interfaz de funciones foráneas), es decir, un mecanismo por el cual un lenguaje de programación puede llamar a funciones o rutinas

²El hecho de poder gestionar la memoria utilizada en cualquier momento, así como la ausencia de una máquina virtual intermedia hacen que C++ sea el lenguaje idóneo para proyectos donde el rendimiento es crítico.

programadas o compiladas en otro lenguaje distinto, lo cual es necesario para poder ejecutar el juego, ya que la entrada de la aplicación Android está en Java.

El resto de decisiones son propias de cada una de las partes de RPGBAKER y se detallan a continuación.

4.2.1. Diseño del motor

4.2.1.1. Base con el sistema entidad-componente

El objetivo del motor es poder ejecutar juegos RPG, por ello deberá implementar sus sistemas y permitir su uso a alto nivel. Sin embargo, aún teniendo la separación entre motor y *gameplay* de un motor específico en este alto nivel, a bajo nivel estos sistemas estarán construidos alrededor de estructuras modulares propias de motores generalistas.

La base del motor se estructurará atendiendo a un patrón EC (entidad/componente) y los sistemas específicos de los juegos RPG se construirán sobre esa base. Lo primero a tener en cuenta en esta parte es el bucle de juego.

En este bucle de juego es en el que se actualizarán los elementos de cada juego. Estos elementos están organizados según una jerarquía. La primera pieza son las escenas. Cada escena se define como un conjunto de entidades. A su vez, cada una de estas entidades es quien representa cualquier elemento del juego: un personaje, un obstáculo, un cuadro de texto... Para definir cuál será el comportamiento de cada una de las entidades estas contienen un conjunto de componentes. Los componentes encapsulan funcionalidades específicas, se pueden crear de múltiples tipos permitiendo una gran flexibilidad a la hora de marcar el funcionamiento de cada entidad. Gracias a esta estructura se podrán implementar una amplia variedad de mecánicas mediante la creación de distintos componentes.

Una vez con esta estructura montada, para funcionar necesita conocer qué escenas, entidades y componentes debe utilizar. Para este propósito se crea un sistema de gestión de recursos. Definimos recurso, o *asset*, como todo aquel elemento externo al código del programa que este usará para obtener el comportamiento deseado, estos pueden ser por ejemplo imágenes, audios, fuentes de texto... La forma en la que este sistema se conecta con el anterior es a través de estos recursos. Cada una de las escenas se definirá en un fichero que el motor interpretará como un recurso con el que crearlas dentro del programa. De esta forma el motor no deberá conocer de antemano estas escenas y su contenido permitiendo un uso mucho más ligero.

Estos recursos de escenas estarán descritos cada uno en un fichero Lua y son lo que definirá el *gameplay* de cada juego. Lua ha sido elegido por su simplicidad, su integración fluida con C++ mediante librerías como **sol2**, y su compatibilidad multiplataforma. Estas librerías proporcionan *bindings*, es decir, código que facilita el uso de Lua desde C++ y viceversa.

Además de la carga de escenas, el sistema de carga de recursos permitirá gestionar múltiples tipos de recursos que pueden ser utilizados por el resto del motor.

Cuando se solicite un recurso, este se cargará en memoria si no lo estaba previamente. Esta operación se realizará hasta alcanzar un límite de memoria, definido por el usuario. Una vez alcanzado dicho límite, si se solicitase un nuevo recurso, se aplicará un algoritmo LRU (*least-recently-used*, usado menos recientemente), que liberará el recurso que más tiempo lleve sin usarse para hacer espacio al nuevo.

4.2.1.2. Componentes genéricos de juego

Para desarrollar videojuegos, es necesario contar con ciertas funcionalidades básicas que faciliten la implementación de los sistemas específicos del juego. Estas funcionalidades han sido mencionadas en el capítulo 3, y se comentarán a continuación.

En el motor, se utilizará la librería **SDL3** para implementar estos sistemas. Esta elección se debe principalmente, a su sencillez de uso y a su robusto soporte multiplataforma, que permite ejecutar los juegos tanto en sistemas de escritorio como en dispositivos Android.

Los sistemas que se implementarán serán los siguientes:

- Sistema de *renderizado*: es imprescindible contar con un mecanismo que permita mostrar visualmente lo que ocurre en el juego. Dado que se trata de un motor para videojuegos 2D, se usarán imágenes y texto para cubrir esta necesidad. Estos sistemas de *renderizado* se expondrán al usuario a través de componentes que permitirán mostrar y animar imágenes, mostrar texto y controlar una cámara desplazable. Además, se implementará un sistema de *renderizado* basado en capas y escenas, que permitirá superponer diferentes escenas, lo cual resulta útil para mostrar elementos (como menús) en forma de *overlay*, y permite tener un mayor control sobre el orden de *renderizado*.
- Sistema de entrada: se desarrollará un sistema de *input* sencillo basado en clics y toques. Este diseño garantiza la compatibilidad multiplataforma y permite centrar la jugabilidad en la interacción mediante botones, en línea con la experiencia tipo *point-and-click* que se quiere ofrecer. El motor unificará la entrada mediante clic y toque en una estructura común que incluya la posición del evento y su estado (inicio, mantenido o final) en un determinado fotograma.
- Sistema de sonido: la retroalimentación auditiva es un componente esencial en los videojuegos. Para su implementación, se empleará el nuevo sistema de sonido incluido en **SDL3**. Para lograr este propósito se creará un componente que permita reproducir, pausar, detener y reanudar los sonidos. Además, estos pueden agruparse en diferentes conjuntos, lo que permite controlar de forma independiente el volumen general, el de música y el de los efectos sonoros.
- Sistema de colisiones: está diseñado para cubrir las necesidades de los juegos previstos, donde será suficiente con detectar intersecciones entre rectángulos. A través del componente correspondiente, será posible comprobar si un objeto colisiona con otro, si acaba de entrar en colisión o si ha dejado de colisionar.

4.2.1.3. Componentes específicos de RPG

Para la navegación por el mundo el motor ofrece un sistema de mapas. Cada mapa será una de las áreas de un juego especificada por el diseñador. Estos mapas están formados por casillas y cada una de ellas puede contener *tiles*. Los *tiles* son imágenes preparadas para combinarse modularmente de forma que en conjunto dibujen un escenario. En estos mapas también se podrán definir los elementos interactivos del juego como, por ejemplo, los NPC.

Sobre estos mapas, se ha diseñado un sistema de movimiento automático y detección de colisiones basado en una cuadrícula de casillas y un algoritmo A* de búsqueda de caminos. Para ello, se mantendrán actualizadas las posiciones ocupadas dentro de la cuadrícula, tanto por elementos estáticos como por elementos dinámicos, y se recalculará la ruta siempre que sea necesario. Este sistema se aplicará tanto a los NPC, a través del sistema de eventos y sus comportamientos asociados, como al jugador, mediante un sistema de entrada de tipo *point-and-click*.

También se implementará un sistema de carga dinámica de mapas y transición entre ellos. Mientras el jugador se encuentra en un determinado mapa, el motor cargará en segundo plano los mapas adyacentes. Al cambiar de un mapa a otro, se descargarán aquellos que ya no sean necesarios. Esta estrategia permite equilibrar el uso de memoria y los tiempos de carga, ofreciendo una solución eficiente y escalable.

Adicionalmente, se dispondrá de un sistema de diálogos con cuadros de texto que muestren el contenido de forma dinámica. El jugador podrá avanzar en la conversación mediante una entrada sencilla, mientras que el motor gestionará automáticamente los saltos de línea y el ajuste del tamaño del texto. Asimismo, se integrará un selector de opciones que permita al jugador elegir entre distintas alternativas mediante botones interactivos.

Las decisiones tomadas se registrarán, lo que permitirá diseñar desde el editor sistemas de interacción complejos al estilo clásico de los RPG. Este sistema hará uso de variables locales (propias de cada objeto del mapa) y globales del jugador (asociadas a la partida), permitiendo un control detallado de la progresión sin requerir programación adicional.

Por último, se desarrollará un componente destinado a facilitar el diseño de la lógica específica del juego mediante comandos sencillos: el gestor de eventos. Este componente constituirá el núcleo del sistema de interacción y comportamiento en el mundo del juego.

4.2.1.4. Sistema de eventos

Uno de los objetivos principales del motor es ofrecer la posibilidad de crear un RPG sin necesidad de conocimientos de programación. Para ello, se ha diseñado un sistema de eventos: una estructura que permite controlar la lógica del juego mediante instrucciones de alto nivel.

La idea fundamental es sencilla: gracias al componente de gestión de eventos, una entidad podrá contener un conjunto de eventos. Dentro del motor se define un evento como una combinación de una condición y una serie de comportamientos.

La condición determinará bajo qué circunstancias deberá activarse el evento. Existirán múltiples tipos de estas condiciones de alto nivel, encargadas de evaluar distintos aspectos del estado del juego. Algunas de ellas son, por ejemplo, si el jugador ha interactuado con un elemento o si ha pasado un cierto tiempo desde un instante. Si una condición se cumple su evento asociado comenzará su ejecución. Además, será posible combinar condiciones mediante operadores lógicos (*not*, *or*, *and*), lo que permitirá definir reglas más complejas y flexibles.

La segunda parte del evento serán los comportamientos, encargados de definir qué acciones se deben realizar una vez activado el evento. Estos comportamientos actuarán como una lista de instrucciones que modifiquen el estado del juego a través de parámetros sencillos. Estarán escritos en Lua y se encargarán de invocar distintas funciones del motor: desde mover objetos, cambiar animaciones o modificar la música, hasta iniciar diálogos.

El sistema está diseñado para que la ejecución de los comportamientos sea progresiva: en cada actualización del juego se ejecutará uno de los comportamientos activos del evento. En los casos en los que una acción requiera más de una actualización para completarse, el comportamiento se limitará a establecer un objetivo, que será procesado por el resto de sistemas mientras el evento continúa ejecutando el resto de sus comportamientos.

Existirán además comportamientos orientados al control del flujo de ejecución. Por ejemplo, uno de ellos permitirá esperar a que se cumpla una condición antes de continuar, actuando como un mecanismo de bloqueo temporal. También, se implementarán instrucciones de salto que permitirán la creación de bucles o la bifurcación de la ejecución según el estado del juego.

Gracias a este sistema, que actúa como lenguaje de *scripting* simplificado, será posible crear lógicas de juego complejas mediante herramientas accesibles y visuales, sin la necesidad de escribir código complejo.

4.2.2. Diseño del editor

4.2.2.1. Funcionalidad del editor

La esencia del editor es la de una interfaz intuitiva y sencilla de utilizar y aprender para aquellos usuarios noveles que nunca hayan utilizado una herramienta similar. Para evitar cargar cognitivamente a los usuarios con información textual, se ha optado por el uso de símbolos e iconos acompañados por descripciones emergentes cortas en la mayoría de elementos.

Las características fundamentales que el editor tiene corresponden con aquellos elementos que pueden ser modificados en el motor, es decir:

- Editor de mapas (figura 4.1), que permite al usuario cargar sus propios *tilesets*; decidir el tamaño de la cuadrícula que ocupa el mapa; dibujar el mapa sobre la cuadrícula utilizando las baldosas del *tileset*, con la posibilidad de tener varias capas para poder simular un efecto de profundidad; y establecer las regiones de colisión del mapa.

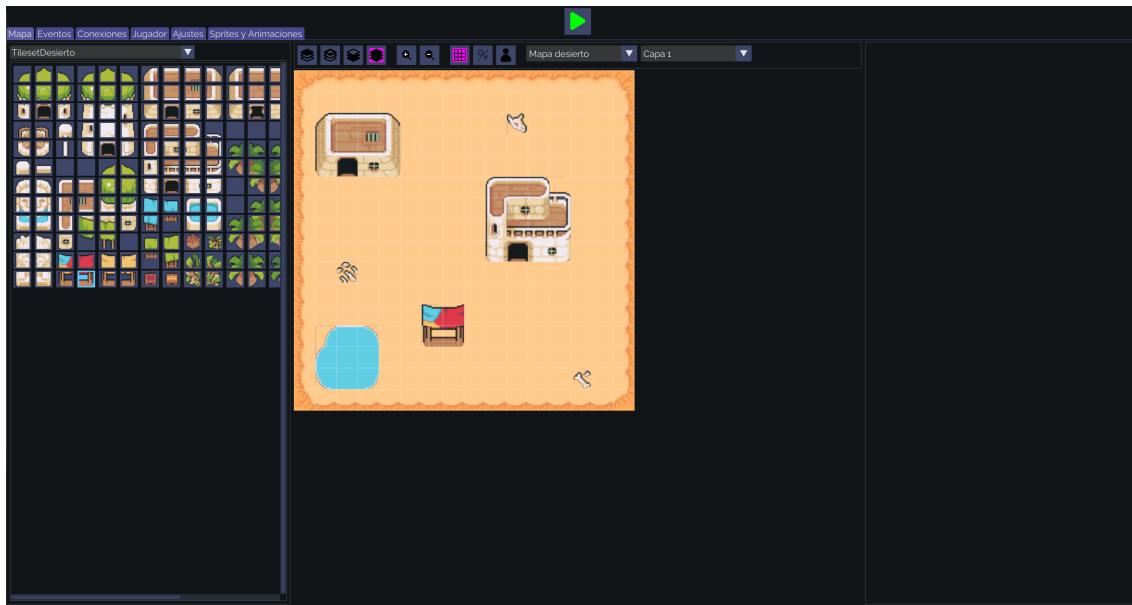


Figura 4.1: Editor de mapas de RPGBAKER.

- Editor de eventos, que permite al usuario la creación de eventos, asignándoles la condición de lanzamiento y los comportamientos que se ejecutarán cuando la condición se cumpla. El editor de eventos tiene soporte para eventos complejos, en el caso de que el usuario requiera de la creación de uno de ellos.
- Editor de *sprites* y de animaciones, que permite al usuario generar un *sprite* dada una imagen o *spritesheet*, y, posteriormente, animaciones dada una serie de *sprites*. Ambos editores tienen una previsualización del *sprite* o animación, añadiéndose controles para la reproducción en este último caso.
- Editor de objetos, que permite al usuario generar un objeto para cada una de las baldosas en la cuadrícula del mapa y asignarle un *sprite* y un evento.
- Editor de personaje, que permite al usuario personalizar su *sprite*, añadirle animaciones de movimiento y configurar diversos parámetros adicionales, como su posición de origen en el mapa.
- Editor de conexiones entre los mapas, que permite al usuario establecer las posiciones de los distintos mapas en el mundo, de manera visual e intuitiva.
- Editor de ajustes generales del ejecutable final, como por ejemplo el nombre del juego, dimensiones de la cámara, la fuente por defecto a utilizar en los textos, o el mapa inicial del juego.

Todos estos elementos tienen la capacidad de poder ser configurados al gusto del usuario, permitiendo ser editados y eliminados cuando este desee.

Por otra parte, el editor cuenta con un sistema de persistencia, que permite una carga y guardado de los *proyectos*, es decir, la representación de un juego en el editor.

Todos los recursos utilizados (es decir, imágenes, sonidos, animaciones...) tienen que estar referenciados en archivos de configuración del *proyecto*, y el guardado actualizará estos archivos de configuración.

4.2.2.2. Modularidad y arquitectura

En cuanto a la estructura de la interfaz, se ha optado por utilizar una arquitectura típica en el desarrollo de *software* no relacionado con videojuegos, basado en ventanas (o subventanas) anidadas en otras ventanas, con el uso de ventanas modales que aparecen sobre estas. Esta estructura permite una escalabilidad del proyecto mucho más sencilla en caso de futuras expansiones y una mayor modularidad con cada uno de los componentes que se quisiese integrar.

Las ventanas tienen comunicación unas con otras utilizando al *proyecto*, que almacena toda la información referente a cada uno de los juegos que el usuario crea (por ejemplo, referencias a los *tilesets*, *sprites*, animaciones o mapas creados).

Se tienen también diversos gestores, tanto de *scripting*, como de elementos de entrada/salida de ficheros, como de preferencias del usuario y hasta un gestor de idiomas, que permite que RPGBAKER sea multilingüe³ con una amplia escalabilidad en el caso de que se quisiesen añadir más idiomas.

4.2.2.3. Tecnologías utilizadas

Para conseguir todos los objetivos anteriores, se investigó qué librerías se iban a poder utilizar para conseguir desarrollar toda la interfaz y la funcionalidad básica. Al ofrecer las facilidades necesarias para los objetivos de esta parte del proyecto, como el soporte multiplataforma, se optó por SDL3 como base para el editor, acompañada de DearImGui para el dibujado y manejo de los elementos de la interfaz.

También, al ya haber elegido Lua como lenguaje de *scripting* y de definición de datos para el motor, se ha optado por el uso del mismo para el editor, acompañados por la anteriormente mencionada librería sol2.

Los archivos de configuración del proyecto difieren de los que el motor espera recibir, ya que muchas veces el editor espera recibir más datos de los que el motor necesita (por ejemplo, rutas específicas de *assets*, guardado de *tilesets*...). Es por eso que el editor se tiene que encargar de «traducir» estos ficheros de definición de datos a los datos que el motor espera; esto se hará en el tiempo de generación del ejecutable final. Esta decisión se ha tomado para que, si no se dispone de los ficheros de configuración del *proyecto*, un usuario ajeno al diseño del juego sea incapaz de poder modificarlo tan fácilmente, al igual que ocurre en la gran mayoría de motores⁴.

³En un principio, únicamente en castellano y en inglés, pero debido al sistema implementado, es muy sencillo añadir nuevos idiomas en el caso que fuese necesario.

⁴Esto es debido a que, en muchos casos, se quiere que el jugador final evite poder hacer modificaciones al juego si no dispone del código original de este, evitando posibles casos de piratería o distribución no autorizada.

Capítulo 5

Motor

El desarrollo del motor se fundamenta en 4 capas, cada una con distintos sistemas y utilidades.

5.1. *Core*

En primer lugar, la base del *Core* está formada por un sistema de entidades y componentes organizados en escenas, junto con un sistema de carga de recursos que lee tanto archivos fuente como archivos .lua.

5.1.1. Sistema de entidad-componente

El entorno de juego consta de múltiples componentes, cada uno contenido en una entidad. Estas entidades se agrupan en escenas, que gestiona el *SceneManager*.

La lógica de cada componente se rige por un ciclo de vida con las siguientes fases:

- **Init:** se invoca solo una vez al crear el componente, durante la construcción de la entidad o más adelante en tiempo de ejecución. Su función es inicializar atributos, ya sea obteniendo referencias de otros elementos de la escena o leyendo parámetros desde un objeto *ComponentData* que contiene datos leídos del archivo .lua.
- **Update:** se llama en cada iteración del bucle principal, aquí reside la mayor parte de la lógica del componente. Se puede evitar que esta llamada ocurra desactivando el componente y viceversa mediante un método *setEnabled(true/false)*.
- **OnEnable / OnDisable:** se disparan al activar o desactivar el componente respectivamente. Además, si un componente comienza habilitado al cargar la escena, *OnEnable* se invoca automáticamente.

Cada entidad mantiene sus componentes en un diccionario ordenado, donde la clave es un entero que define el orden de ejecución. Además, cada una de ellas puede tener otras entidades hijas: cada una almacena un conjunto sin ordenar de punteros a sus hijas y un puntero a su padre. La propia entidad propaga las llamadas de

Update y el resto del ciclo de vida a sus hijos, la escena solo conserva referencias a las raíces de la jerarquía. Para eliminar una entidad, se marca su atributo `Alive` en `false` y luego se purga en la llamada a `refresh` que ocurre tras cada Update.

Las escenas guardan sus entidades raíz en un conjunto sin orden y ejecutan recursivamente las llamadas correspondientes sobre cada rama de la jerarquía. También mantienen un diccionario de «handlers», claves únicas de tipo *string*, para referenciar rápidamente entidades desde cualquier componente. Además, almacenan referencias a los `RenderComponent` registrados, organizados por capas, que se explicarán más adelante.

Por último, tenemos al `SceneManager`, quien se encargará de la gestión de escenas, de instanciar nuevos objetos a partir de *blueprints* y de conectar las llamadas del bucle principal con la escena activa. Las escenas en el `SceneManager` se organizan en una lista que hace las veces de pila: podemos añadir escenas con `push_back` y eliminarlas con `pop_back`, y la que esté al final (`back`) es la que se considera activa. Sin embargo, mantenemos la estructura como lista para poder iterar sobre ella durante el renderizado, ya que queremos pintar todas las escenas, no solo la activa.

Esta base está preparada para funcionar con múltiples escenas distintas con diferentes entidades y componentes que se carguen desde archivos de datos. Para que esto funcione así, se ha implementado un sistema de gestión de recursos en el motor.

5.1.2. Sistema de gestión de recursos

La segunda parte fundamental de la base del motor es el sistema de gestión de recursos. Su objetivo es poder cargar distintos tipos de recursos de forma genérica y flexible para poder implementar funcionalidades dependientes de archivos externos. Para poder hacerlo este sistema consta de tres partes esenciales:

- `Resource` es una clase base que sirve como interfaz para poder crear cualquier tipo de recurso que se necesite. Permite implementar dos métodos `load()` y `unload()`. El primero es el que se encargará de leer el archivo correspondiente a ese recurso y almacenar su contenido en memoria. El segundo liberará la memoria reservada por el primero invalidando los accesos posteriores que se puedan hacer sobre el recurso. Además de esto, esta clase da acceso a un parámetro en el que cada uno de los recursos registrará el espacio ocupado en memoria. Este podrá ser consultado desde otras partes del código para poder tener una gestión de memoria más profunda.
- `ResourceHandler` es una clase plantilla que sirve para poder acceder a los distintos tipos de recursos que implementen la interfaz `Resource`. Usa el patrón *singleton*, pues queremos permitir que se pueda acceder a los recursos desde cualquier punto del motor. Su función principal es devolver un recurso que se pueda acceder a partir de un archivo dado. Con su método `get(key)` se encargará de crear el recurso solicitado en el motor si no lo estaba y retornarlo.

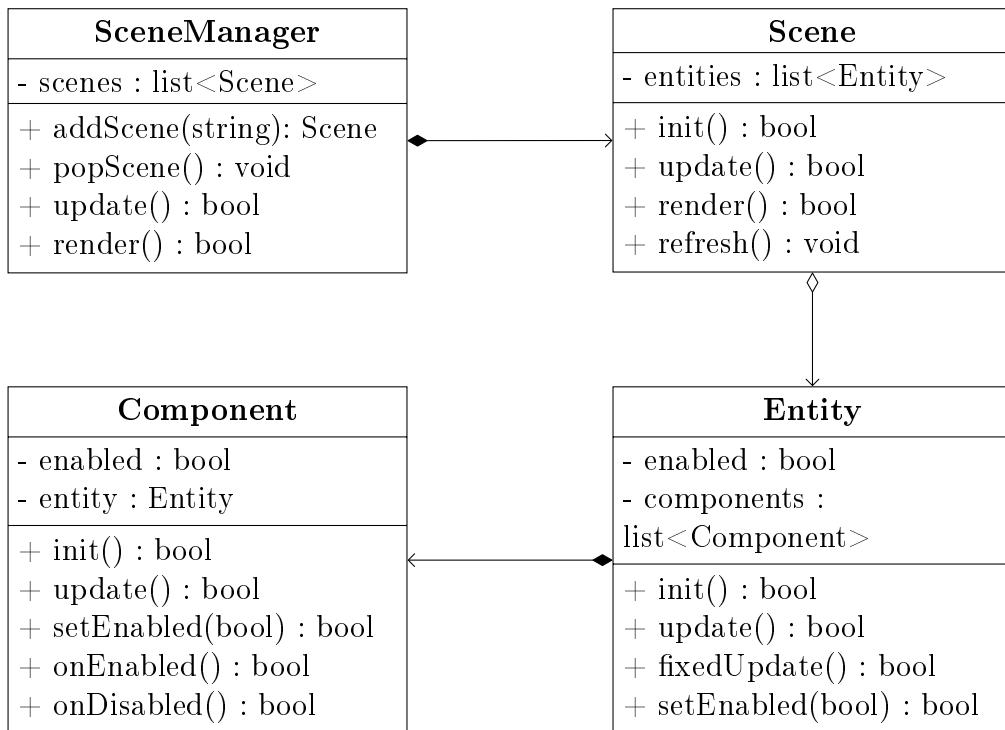


Figura 5.1: Esquema de la estructura del sistema de entidad componente del motor.

- **ResourceMemoryManager** se encarga de controlar qué recursos permanecen cargados y cuánto espacio de memoria están ocupando. A través de un archivo de configuración se puede establecer cuál es el límite de memoria que se quiere usar para los recursos; esta clase es la que se encarga de que ese umbral no se supere. A través de un algoritmo LRU se gestiona qué recursos están cargados en cada instante. Cada vez que se quiera usar alguno de estos, es decir, cada vez que se llame a `get(key)` de un **ResourceHandler**, se le solicitará a este gestor de memoria que lo active. Esta activación consiste en tres fases. Primero comprobar si el recurso está ya cargado, si es así simplemente se actualizará en el LRU. En caso contrario, iría al segundo paso que es cargarlo y comprobar si hay espacio suficiente, de ser así se almacenaría. Si no, el tercer paso es crear ese espacio y, a través del LRU, se irán descargando aquellos recursos que lleven más tiempo sin ser utilizados, hasta que haya espacio suficiente para que el nuevo se almacene.

Sobre estas tres piezas fundamentales está estructurado el sistema de gestión de recursos y, adicionalmente, existen otras dos clases que ayudan en esta tarea. **ResourceManager**, que se encarga de inicializar el sistema con la configuración dada y establecer la conexión entre **ResourceMemoryManager** y los **ResourceHandler**. Y **LuaReader**, que ofrece una interfaz sencilla para poder extraer tablas de Lua guardadas en un archivo o dentro de otra tabla de Lua. La lectura de estos archivos se hace a través de **SDL**, para poder mantener la funcionalidad entre plataformas, y la interpretación a través de **sol2**.

5.1.3. Carga de escenas a partir de datos

Una parte fundamental de los motores de videojuegos es que han de permitir ejecuciones con resultados distintos dependiendo de los datos que se le aporten. Esto es para poder reutilizar en videojuegos completamente distintos las herramientas que este ofrece. Se usará un paradigma conocido como programación dirigida por datos, *data-driven programming*.

Las escenas son un conjunto de entidades que se definen en archivos .lua. Estos archivos se leen en tiempo de ejecución y se almacenan en diferentes *blueprints*, que guardan la información esencial para que el motor pueda instanciar escenas, entidades y componentes a partir de ellos. Podemos diferenciar cuatro tipos de *blueprints*:

- **SceneBlueprint:** se guarda como recurso del sistema de carga. Lee un archivo .lua con el nombre de la escena y almacena todas las entidades definidas en él en un vector de `EntityBlueprints`. Más adelante, el `SceneManager` lo utiliza para crear la escena, iterando sobre el vector de entidades e instanciándolas.
- **EntityBlueprint:** contiene la información necesaria para crear e instanciar una entidad. Normalmente, forma parte de un `SceneBlueprint`. Guarda todos los hijos como `EntityBlueprints`, en un vector `children`, y todos sus componentes en un vector de `ComponentData`. Adicionalmente, almacena dos variables: el `handler`, un *string* que, si no está vacío, se registra en la escena para poder referenciar esa entidad; y un *booleano*, `active`, que indica si la entidad se instancia activa o inactiva en la jerarquía de la escena, esto significa que no recibirá llamadas a `update` ni a `render`. El `SceneManager` recorre recursivamente los hijos llamando al método de creación de entidad y luego itera sobre los componentes para añadirlos e inicializarlos con los parámetros almacenados en cada `ComponentData`.
- **PrefabBlueprint:** es una subclase de `EntityBlueprint` que se guarda como recurso independiente mediante herencia múltiple. Funcionalmente es idéntico, pero en lugar de formar parte de un `SceneBlueprint`, se lee desde su propio archivo de recurso y se mantiene de forma separada. El `SceneManager` lo utiliza para instanciar entidades en tiempo de ejecución, por ejemplo, en el sistema de mapas.
- **ComponentData:** almacena toda la información necesaria para crear e inicializar un componente. Consta de dos elementos:
 - El identificador de componente, un *string* que se usa para crear la una instancia de la clase de componente correspondiente.
 - `data`, una tabla de `sol2` donde se guardan los valores de inicialización de los distintos parámetros.

Para crear los componentes a partir de la información de estos `ComponentData` surge el problema de tener que crear objetos de clases distintas a partir de identificadores. La solución aplicada en este caso es el uso del patrón factoría. En una clase

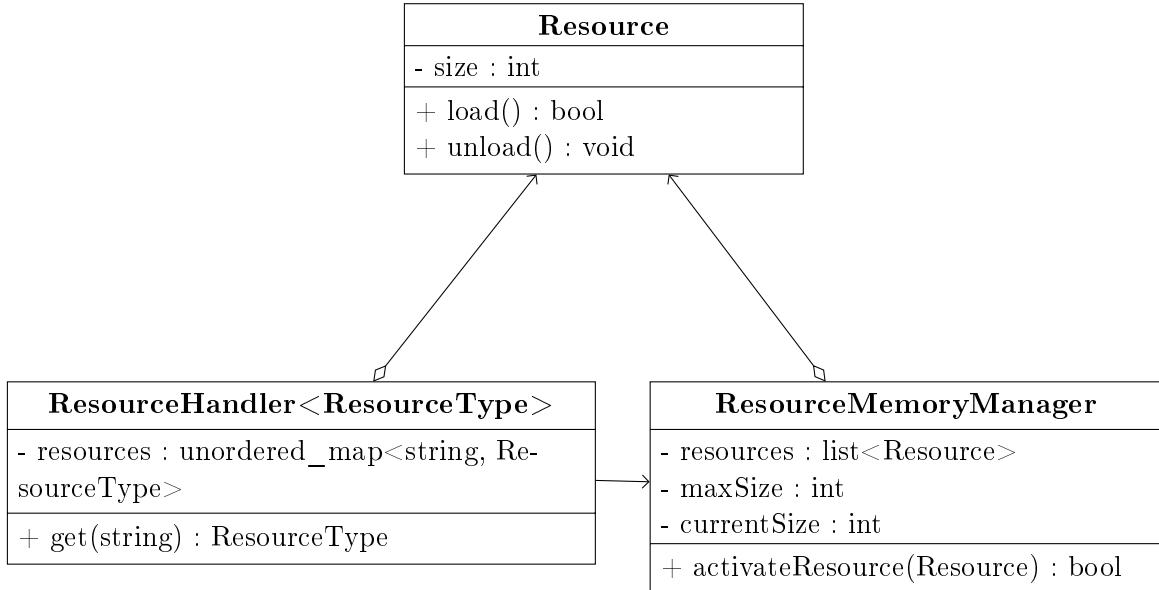


Figura 5.2: Esquema de la estructura del sistema de gestión de recursos del motor.

`ComponentFactory` se registrará la creación de cada tipo de componente asociada a un identificador. Con esto se le podrá solicitar a esta clase crear cada tipo de componente haciéndole llegar un identificador.

Para poder definir ese identificador fácilmente en cada componente se ha creado un paso intermedio en la herencia de estos, `ComponentTemplate`. Cada componente derivará de este permitiendo definir el identificador de cada componente fácilmente en su declaración.

Con esto, a partir de un objeto `ComponentData`, el `SceneManager` crea los componentes mediante `ComponentFactory`, estos *blueprints* conservan la tabla `data` para usarla en su llamada a `init`. Para evitar el uso directo de los métodos de `sol2` en cada componente, `ComponentData` proporciona las interfaces `getData` y `getArray`, que permiten obtener los valores almacenados. Además, `ComponentData` incluye en su propia tabla `data` un indicador que señala si el componente debe instanciarse habilitado o deshabilitado.

5.2. Componentes genéricos de juego

De cara a facilitar el desarrollo de videojuegos los motores ofrecen implementaciones de sistemas que se usan en la gran mayoría de estos. Hay múltiples bibliotecas que facilitan la adición de estas funcionalidades, en este caso se usará `SDL3` que ofrece todo lo necesario para desarrollar de forma sencilla lo diseñado además de ofrecer soporte multiplataforma.

5.2.1. *Render*

Para el apartado de *renderizado* se utilizan las funciones que ofrece `SDL`. El `RenderManager` se encarga de encapsular estas funciones en métodos para acceder a

ellos de forma centralizada y simple. Desde la llamada de render de cada componente estos podrán acceder a dichos métodos para realizar tareas como dibujar imágenes, texto o cambiar la zona del mundo que se dibuja.

El ciclo de *renderizado* de cada fotograma es:

1. Llamar a `clear` para limpiar el buffer.
2. Invocar `render` en el `SceneManager`, que lo replica en cada escena y, a su vez, en todos los `RenderComponent` registrados.
3. Llamar a `present` para mostrar el resultado.

Este proceso se repite durante todo el ciclo de vida de la aplicación.

Además, el `RenderManager` gestiona la inicialización y el cierre de los subsistemas de *SDL* relacionados, así como la creación de la ventana de vídeo.

El resto de la funcionalidad de vídeo se concentra en los componentes de *render* (`RenderComponent`) y en los recursos que utilizan. Implementa el método `Render(RenderManager*)` en el que realiza las llamadas a los métodos de *renderizado* necesarios. Todos dependen de un `Transform` que indica posición, rotación y escala, y especifican su capa para organizar el orden de dibujo. Los `RenderComponent` se registran en la escena en `OnEnable` y se des registran en `OnDisable`, con lo que la escena controla qué componentes están activos en cada momento.

Las llamadas de *renderizado* se realizan en orden inverso de la pila de forma que las escenas que estén más abajo en la pila se llaman primero y por lo tanto se ven por debajo de las siguientes. Además, dentro de cada escena los `RenderComponent` se llaman en orden de capas de forma que las capas más altas se pintan las últimas.

Los `RenderComponent` implementados son:

- `Camera`: modifica el área de *renderizado* llamando a `setViewRect`. Tiene un tamaño lógico en píxeles y su posición se controla mediante su `Transform`. Por defecto usa la capa -1 para que su llamada `Render` sea la primera de la escena.
- `Rectangle`: guarda tamaño y color, y dibuja un rectángulo en la posición del `Transform`.
- `Text`: crea una `TextTexture` a partir de fuente, tamaño, color, centrado y texto, y la dibuja en la posición del `Transform`.
- `SpriteRenderer`: dibuja un `Sprite` (fragmento de textura cargada desde una imagen) en la posición del `Transform`.
- `Animator`: subclase de `SpriteRenderer` que añade funcionalidad de animación mediante `Animation`.

Las clases auxiliares de recursos son:

- `Animation`: guarda un vector con los identificadores de `Sprite` para cada fotograma, un tiempo de duración y si debe reproducirse en bucle. Se crea desde un archivo `.lua` con esta información.

- **Sprite**: almacena el identificador de una textura y el rectángulo correspondiente a la zona a dibujar.
- **Font**: crea y gestiona una `TTFFont` de `SDL` a partir de un archivo `.ttf` y un tamaño.
- **Texture**: crea y gestiona una `SDLTexture` a partir de un archivo de imagen.
- **TextTexture**: genera y administra una textura de texto de `SDL` a partir de un texto y unas características de fuente.
- **TextureLoader**: *singleton* encargado de crear texturas usando el *renderer* de `RenderManager`.

Con esta infraestructura podemos dibujar en pantalla todo lo necesario para representar las entidades de nuestros juegos.

5.2.2. Entrada

El sistema de entrada gestiona toda la información que recogemos de los eventos de ratón y pulsación de `SDL` en un `struct` al que podemos acceder desde cualquier lugar del motor mediante un patrón *singleton*. Aunque no se expone directamente la instancia del `InputManager`, sí es posible obtener una referencia constante al `struct` que contiene toda la información necesaria para implementar el *gameplay* de los juegos. Decidimos no implementar ningún tipo de entrada por teclado o mando, con el fin de garantizar un diseño multiplataforma sencillo, ya que todas las funcionalidades de *input* que necesitábamos se pueden abstraer con clics o pulsaciones.

En coordinación con este sistema, también creamos el componente `Button`. Este almacena una `sol::function` como *callback*, que se invoca con unos parámetros de tipo `sol::object` cuando detecta un *input* dentro de su área. Gracias a esto, podemos conectar el motor genérico en C++ con la lógica específica de *gameplay* programada en Lua.

5.2.3. Audio

Este es el sistema que se encarga de controlar la reproducción de sonidos para los juegos. Envuelve al nuevo sistema de sonido de `SDL3` para ofrecer una interfaz adecuada al resto de sistemas del motor. Está hecho centrándose estas funcionalidades alrededor de un componente `AudioSource`, cuya función es permitir controlar la reproducción de una sola instancia de un sonido. Lo que se busca en `AudioSource` se puede separar en otras dos clases:

- `AudioClip` se encarga de envolver a `SDL_AudioStream` ofreciendo una interfaz simplificada. Representa una pista de sonido, se puede reproducir, pausar, detener o cambiar su volumen entre otros. Almacena un identificador de la pista de audio que representa. Para su funcionamiento se apoya en el sistema de gestión de recursos, con una clase que extiende de `Resource`, `AudioClipData`.

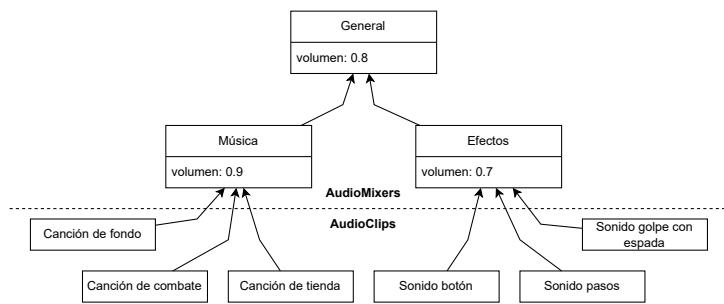


Figura 5.3: Representación de un ejemplo de las conexiones de los elementos del sistema de audio del motor.

Cada vez que se quiera hacer una modificación en el estado de la pista se solicitará el acceso al recurso correspondiente a través de su identificador.

- **AudioMixer** simula una simplificación de cómo funciona un mezclador de audio. Puede recibir múltiples entradas de audio y ofrece una salida. Esto significa que, a un **AudioMixer**, se le pueden conectar tanto múltiples **AudioClip** como otros **AudioMixer**. Permite modificar el volumen de reproducción de todos sus elementos conectados. Esto es así para poder permitir un control más preciso de distintos tipos de sonidos. En los videojuegos es típico permitir al jugador controlar tanto el volumen general como el de la música o los efectos de sonido, por ejemplo. De esta manera se pueden estructurar en forma de árbol las distintas salidas de audio para ofrecer esta funcionalidad, como se exemplifica en la figura 5.3. Con esto y con una clase **AudioManager**, encargada de gestionar el dispositivo de salida y las conexiones entre los **AudioMixer**, se completa la implementación del componente **AudioSource**.

5.2.4. Colisiones

En lo que refiere a detección de colisiones no se busca que el sistema sea especialmente complejo. Los videojuegos que se ofrecerán crear con el motor no necesitan más que comprobaciones sencillas obtenidas a partir de intersecciones entre rectángulos. Para hacer esto el sistema de colisiones consta de dos elementos:

- **Collider** es el componente que representa un elemento que puede colisionar. Permite la comprobación de si otro **Collider** acaba de empezar a colisionar, está colisionando o acaba de dejar de colisionar con este.
- **CollisionManager** es quien se encarga de mantener al día el estado de cada **Collider**. Cada vez que se crea una instancia del componente se registra en esta clase. En cada actualización de juego se encargará de cambiar los estados en función de las áreas representadas por cada componente. Esto lo hace a través de una sencilla función que proporciona **SDL** que comprueba si dos rectángulos tienen intersección.

5.3. Componentes específicos de RPG

Sobre toda la infraestructura creada en las secciones anteriores se construyen algunos componentes específicos orientados a los juegos RPG que se podrán diseñar en el editor.

5.3.1. Movimiento

El sistema de movimiento es autónomo y está basado en el cálculo de una ruta entre dos casillas. Para ello se utiliza un algoritmo de *A** sobre casillas, con distancia *Manhattan* y movimiento en cuatro direcciones. El sistema consta de tres componentes:

- **MovementManager**: componente único en escena encargado de registrar las casillas ocupadas, ya sea por elementos estáticos (**MovementObstacle**) o dinámicos (**MovementComponent**), y de calcular las rutas más eficientes bajo demanda de los **MovementComponent**.
- **MovementObstacle**: representa casillas estáticas ocupadas en el sistema de movimiento. Se registra en el **MovementManager** en **OnEnable** y se desregistra en **OnDisable**.
- **MovementComponent**: hereda de **MovementObstacle**. Obtiene un **Path**, un *vector* de movimientos, e itera por él moviendo el **Transform** a cada paso. Actualiza su posición en el **MovementManager** cada vez que cambia de casilla. Su velocidad de movimiento es parametrizable y se le pueden asignar animaciones que se reproducen en coordinación con cada dirección de desplazamiento.

Este sistema lo utilizan tanto los *NPC*, vía el sistema de eventos y su comportamiento, como el jugador, mediante el sistema de entrada que obtiene la casilla libre más cercana al clic o pulsación del jugador y la establece como objetivo para su **MovementComponent**.

5.3.2. Mapas

Los diferentes mapas creados en el editor no se instancian en el motor como escenas independientes, sino como entidades que se activan y desactivan dinámicamente en la escena del mundo. Para gestionar esto existen dos componentes:

- **OverworldManager**: mantiene dos conjuntos de mapas, los activos y los cargados pero inactivos. Cuando el jugador cambia de mapa, activa los colindantes, carga los que no estaban cargados y desactiva los que ya no son necesarios.
- **MapComponent**: detecta cuándo el jugador entra en un nuevo mapa y notifica al **OverworldManager**. Guarda los nombres de los mapas adyacentes y el propio y se los envía al **OverworldManager** para que gestione la carga/descarga de estos.

Con este sistema la carga de mapas ocurre dinámicamente durante la partida y no es necesario cargar todo el mundo de una vez o estar constantemente cargando un mapa cada vez que pasemos de uno a otro, obteniendo así un buen equilibrio entre tiempos de carga y uso de memoria.

5.3.3. Diálogos

Un elemento indispensable en un RPG son los diálogos. El motor ofrece dos elementos clave:

- **Textbox:** divide un texto largo (en formato *string*) en fragmentos que caben en el espacio parametrizado. Muestra cada carácter de forma progresiva y, al llenar la caja, espera una entrada para pasar al siguiente fragmento, facilitando la presentación dinámica de los diálogos.
- **Choices:** conjunto de hasta tres botones a los que se les asignan, mediante eventos, diferentes textos y valores que guardan la elección en una variable de juego. El *script* en *.lua* define los textos y asocia una función *callback* que se invoca con el índice de la opción seleccionada, permitiendo interactividad entre el jugador y los eventos diseñados.

5.4. Sistema de eventos

Este sistema es la estructura que permite agregar lógica a un juego sin necesidad de programar. A través de añadir eventos a entidades se puede conseguir que estas se comporten de formas completamente diferentes. La forma en la que este sistema se conecta con el resto del juego es con un componente que se encarga de manejar todos los eventos asociados a una entidad, **EventHandler**. Este componente funciona como un conjunto de eventos, se encarga de actualizarlos y permite acceder a ellos individualmente a partir de su nombre.

Un evento es una estructura que consta de una condición y una secuencia de acciones. En cuanto la condición de ese evento se cumpla se comenzará a ejecutar su secuencia de acciones. Dentro del motor está definido a través de la clase **Event**. Tienen un método **update()** que marca cada una de las actualizaciones del evento, dentro de este es donde ocurre su lógica definida. Cada actualización consta de tres partes. Primero, comprobar si la condición se cumple, en cuyo caso se comenzará con la ejecución de los comportamientos desde el primero. Segundo, ejecutar la acción del comportamiento actual. Y, tercero, si se completó la acción, avanzar al siguiente comportamiento para la próxima actualización.

Además de esto los eventos también ofrecen una interfaz que permite pausar, reanudar o detener su ejecución y cambiar cuál será el próximo comportamiento en actuar. Esta última funcionalidad permite una gran flexibilidad en la posible lógica de estos eventos. A través de este método **jump(index)** y el uso de condiciones se puede simular en la ejecución de los comportamientos el funcionamiento de instrucciones de salto en una CPU. Se explicará más al respecto al hablar en profundidad de los comportamientos.

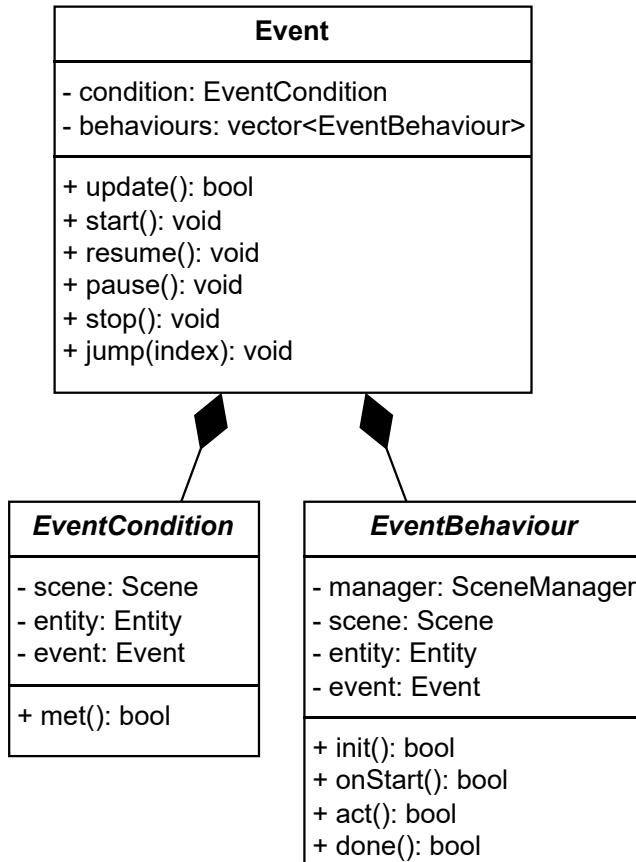


Figura 5.4: Esquema de la estructura de un evento del motor.

5.4.1. Condiciones

Las condiciones, **EventCondition**, se encargan de determinar si ciertos elementos del juego están en algún estado concreto. A través de la función virtual `met()`, definiendo nuevas clases que hereden de esta, la comprobación de cada condición podrá tener definiciones distintas. Cada evento tiene una de estas condiciones asociadas y se iniciará en cuanto esta se cumpla.

Cada implementación de condición tiene alguna comprobación esencial para la creación de este tipo de juegos como, por ejemplo, si el jugador intenta interactuar con algún elemento. Además de condiciones directamente ligadas al estado del propio sistema de eventos como, por ejemplo, si se completó un comportamiento. Otras comprobaciones cruciales de cara a ofrecer un sistema flexible que permite la creación de sistemas complejos son las puertas lógicas. Existen las condiciones *and*, *or* y *not* que permiten combinar condiciones cualesquiera con otras.

Uno de los problemas de cara a la implementación de estas condiciones es el mismo uno que surgió con los componentes, se deben poder construir los distintos tipos de condiciones a partir de información extraída de Lua. Para ello la solución aplicada es la misma, crear una factoría, `EventConditionFactory`, que permita la creación de cada una de las condiciones a partir de un identificador. A través de una clase plantilla como paso intermedio entre `EventCondition` y sus implementaciones, cada una de estas puede definir su identificador al declararse, que será el que usará la factoría.

5.4.2. Comportamientos

Los comportamientos son cada una de las acciones que se pueden realizar a través de un evento. Cada evento consta de un listado de comportamientos que funcionan de forma similar a una lista de instrucciones. Cada uno de estos recibe unos parámetros que harán que se realicen unas modificaciones u otras en el juego.

Buscando que estos comportamientos sean flexibles y se puedan crear nuevos sin necesidad de recompilar el motor se decidió que se implementaran en Lua. De esta manera se pueden añadir a las escenas muy fácilmente al estar estas también escritas en Lua. Aún con esto, surge el problema de que en Lua no existe de forma nativa la infraestructura de la POO (Programación orientada a objetos) que encaja perfectamente con el diseño planteado. Sin embargo, existen formas sencillas de simular este comportamiento a través de las tablas, que son la estructura básica de la programación en Lua. Gracias a poder asignar funciones dentro de las tablas y modificando las *metatablas*, tablas internas de cada tabla que contienen información sobre esta, se puede simular la herencia de clases.

Con esto se creó en Lua una clase `EventBehaviour` que consta de métodos de inicialización, un método de acción y métodos de comprobación de compleción. En estos, cada uno de los comportamientos se comunica con el motor en C++ para identificar y modificar el estado del juego. Para que funcionara esta comunicación se definió, a través de `sol2`, qué clases de C++ se pueden acceder desde Lua y a cuáles de sus atributos dan acceso.

Los comportamientos tienen dos comprobaciones de compleción distintas: `done()` y `ended()`. La primera será cierta cuando el comportamiento haya finalizado su acción, dando paso a que el evento continúe al siguiente comportamiento. La segunda en cambio se encarga de confirmar que la instrucción lanzada por el comportamiento se haya completado. Esto es porque hay comportamientos que le comunican al juego cuál es el cambio a realizar y este puede ser ejecutado de forma simultánea al resto del evento. Aún con esto en casos concretos se puede querer que el evento no continúe sin que se complete en su totalidad la acción solicitada. Para ese caso, entre otros, existe la segunda comprobación, esta se puede combinar con uno de estos comportamientos, `WaitForBehaviour`.

Hay ciertos comportamientos que permiten alterar el flujo de estos dentro de un evento. `WaitForBehaviour` detiene este flujo hasta que se cumpla una `EventCondition` concreta. Combinándolo con la condición `BehaviourEndedCondition`, que comprueba si la acción lanzada por un comportamiento concreto se ha completado,

se puede obtener el resultado recién mencionado, por ejemplo. Hay otros dos comportamientos que modifican el flujo de ejecución de un evento: `JumpBehaviour` y `JumpIfBehaviour`. Estos simulan el comportamiento de una instrucción de salto en una CPU, es decir, establecen cuál es el siguiente comportamiento desde el que se seguirá ejecutando la lista, independientemente de su posición en esta. La diferencia entre estos dos es que el primero siempre hará este salto, mientras que el segundo solo lo hará si se cumple una `EventCondition`. Gracias a estos se puede conseguir que ciertos comportamientos se ejecuten, no lo hagan o lo vuelvan a hacer según el estado del juego.

Las acciones del resto de comportamientos permiten establecer movimiento, cambiar animaciones, reproducir sonido o establecer diálogos.

Por último, se pueden también definir y cambiar valores propios en cada entidad de cara al diseño del videojuego. En un componente `LocalVariables` cada entidad puede tener las variables que se quieran con los valores que se quieran. A través de un comportamiento `ModifyVariableBehaviour` se pueden modificar estas variables de las que se podrán comprobar sus valores gracias a condiciones.

En C++ se definen los comportamientos a través de la clase `EventBehaviour` cuya función es envolver una instancia de Lua de alguno de estos comportamientos para el uso del resto del motor.

Capítulo 6

Editor

6.1. Base del editor

La base del desarrollo del editor se organiza en distintas secciones, clasificadas según su función principal dentro del sistema:

- Elementos comunes: componentes comunes para todo el resto de componentes del editor, y que también permiten la comunicación entre estas distintas partes. Por ejemplo, la clase `Project`, que contiene toda la información referida a cada uno de los proyectos.
- Elementos de lectura-escritura: encargados de gestionar la entrada y salida de datos mediante archivos externos, permitiendo tanto la carga como el guardado de información.
- Elementos de dibujado: componentes encargados de la representación visual en pantalla. Aquí se incluyen todas las ventanas y *subventanas* que componen la interfaz gráfica de la aplicación.
- Recursos: elementos del motor que pueden ser modificados desde el editor. Entre ellos, por ejemplo, mapas, objetos o *sprites*.

6.1.1. Elementos comunes

Los elementos comunes son todos aquellos que no pertenecen exclusivamente a ninguna de las áreas especificadas (por ejemplo, gestión de la interfaz gráfica o gestión de recursos), sino que forman la base del funcionamiento del editor y su funcionalidad es compartida por múltiples partes del sistema.

- `Editor`: implementado utilizando un patrón *Singleton*, se encarga de crear y mantener una única instancia centralizada del editor. Su responsabilidad principal es inicializar todos los componentes esenciales (ventana principal, gestor de dibujado, gestor de entrada de usuario...) y coordinar su funcionamiento. Incorpora un método `mainLoop()` que ejecuta el bucle principal de la aplicación, el cual sigue una estructura típica de aplicaciones interactivas: captura

de eventos de entrada del usuario, actualización del estado y dibujado de la interfaz. Gracias a esta arquitectura, se asegura una separación clara entre la lógica de control y la presentación visual.

- **Project:** representa la estructura y el contenido de cada proyecto creado con el editor. Esta clase centraliza la información relacionada con los recursos del proyecto, incluyendo mapas, *sprites* u objetos. También, almacena las rutas relativas a los archivos en disco y proporciona métodos para modificar, cargar y guardar estos elementos. Además, ofrece una interfaz para la exportación o generación de la *build* final del proyecto, asegurando así la persistencia y portabilidad del trabajo realizado por el usuario.
- **EditorError:** se encarga de gestionar los posibles errores, tanto de programación como de uso de la aplicación. Para ello, se utiliza la librería multiplataforma `tinyfiledialogs`, que permite mostrar alertas en pantalla de manera muy sencilla y sin inicialización previa.

En conjunto, estos elementos comunes proporcionan una infraestructura robusta y reutilizable sobre la que se construyen el resto de funcionalidades del editor, facilitando tanto su mantenimiento como su escalabilidad futura.

6.1.2. Elementos de lectura-escritura

Los elementos de lectura-escritura son todos aquellos componentes responsables de gestionar tanto la entrada del usuario como la lectura y escritura de datos del sistema, incluyendo preferencias, localización y proyectos.

- **InputManager:** se encarga de inicializar y gestionar la entrada del usuario mediante teclado y ratón. Para ello, utiliza por debajo el método `SDL_PollEvent()`, de la librería `SDL3`, que permite obtener el último evento registrado en cualquiera de los periféricos del usuario¹. Este evento, posteriormente, es discriminado por el gestor dependiendo de si lo procesa la librería `DearImGui` mediante la llamada al método `ImGui_ImplXXXX_ProcessEvent()`², o si se procesa de manera independiente, por ejemplo, comprobando que el evento lanzado es el de cerrar la aplicación (`SDL_EVENT_QUIT`).
- **LuaManager:** se encarga de inicializar y gestionar la librería `sol2`, que permite interactuar mucho más amigablemente con Lua. Esta clase define métodos de serialización de tablas Lua utilizando un serializador externo, `serpent`, creado por Paul Kulchenko, que mantiene la legibilidad del archivo generado, facilitando tanto la depuración como la edición manual del mismo; un método que permite la escritura de una tabla Lua a un fichero `.lua`; y métodos para

¹La implementación de `SDL` es esta, no puede haber dos eventos simultáneos y solo se registrará el último que se haya mandado. Por lo general, es complicado que haya dos eventos simultáneos, siempre hay una demora, por muy pequeña que sea, entre dos eventos.

²Se utiliza `XXXX` en medio del método ya que este es dependiente de la librería con la que se hubiese creado la ventana. En este caso, al estar creada utilizando `SDL3`, el método sería `ImGui_ImplSDL3_ProcessEvent()`.

```
return {
  ["action.settings"] = "Ajustes",
  ["action.add"] = "Añadir",
  ["window.mapeditor"] = "Mapa"
}
```

Figura 6.1: Contenido del fichero `es_ES.lua`.

```
return {
  ["action.settings"] = "Settings",
  ["action.add"] = "Add",
  ["window.mapeditor"] = "Map"
}
```

Figura 6.2: Contenido del fichero `en_EN.lua`.

la creación de tablas Lua y lectura de tablas Lua desde un fichero de datos o un *script*.

- **LocalizationManager**: se encarga de inicializar y gestionar la localización del proyecto, es decir, el idioma en el que aparecerá la interfaz. Para entender mejor la función de este gestor, se explicará el funcionamiento de los ficheros de idiomas del editor.

- El editor cuenta con ficheros `.lua`, nombrados siguiendo la especificación IETF BCP 47 (Phillips y Davis, 2009)³, que contienen una tabla de Lua cuya clave es un identificador definido para un determinado texto en el editor, y el valor es la frase requerida en el idioma correspondiente. Por ejemplo:

En la tabla Lua de la figura 6.1 se observa la traducción en español, y en la tabla de la figura 6.2, la traducción en inglés. Ambos ficheros comparten las mismas claves, por ejemplo, `action.settings`, que se referirá al texto que aparece en un botón que lleve hacia la pantalla de configuración del editor.

- El gestor de localización obtendrá la *locale*⁴ preferida del usuario, utilizando el método `SDL_GetPreferredLocales()`, e intentará cargar el archivo `.lua` cuyo nombre sea el de la *locale* preferida que haya devuelto el método. Si no se encuentra un archivo con el mismo identificador BCP 47, se intentará cargar el fichero con el mismo idioma⁵, y si el idioma tampoco se encuentra, cargará por defecto el idioma inglés.
- Una vez cargado el fichero, se guardará el contenido en un diccionario clave-valor, y se podrá acceder a cualquier frase requerida utilizando la clave guardada en el fichero Lua. Por ejemplo, si se quisiese, desde código, obtener la frase detrás de `action.add`, bastará con llamar a `getString("action.add")`.

³Esta especificación define un idioma de la siguiente manera: «*id_PA*», donde *id* es una abreviación del idioma y *PA* una abreviación del país donde se habla ese idioma. Por ejemplo, para el español de España, su código BCP 47 sería «`es_ES`», y para el inglés estadounidense, «`en_US`».

⁴Las *locales* o configuraciones regionales son todos los parámetros, definidos en el sistema operativo de un ordenador, que definen el idioma, país y preferencias especiales que el usuario quiera ver en su interfaz (por ejemplo, el formato de la fecha y hora o la divisa).

⁵Por ejemplo, una persona argentina tendrá configurado su ordenador de tal manera que la *locale* será «`es_AR`». Al no haber un fichero `es_AR.lua`, cargará uno que tenga el mismo idioma, es decir, `es_ES.lua`.

- También está contemplado el cambio de idioma por parte del usuario, por lo que el gestor también cuenta con un método, `changePreferredLocale()` que permite el cambio de idioma del editor sin más complejidad.
 - Este sistema es bastante ampliable, ya que simplemente para añadir un nuevo idioma, basta con crear un fichero `.lua` cuyo nombre sea el identificador BCP 47 del idioma correspondiente, copiar todas claves de uno de los ficheros existentes, y posteriormente generar la traducción. Este tipo de ampliaciones se suele hacer utilizando plataformas como *Crowdin* (<https://crowdin.com/>), que permiten traducciones hechas por individuos de manera altruista en múltiples idiomas.
- **PreferencesManager:** gestiona las preferencias del usuario en cuanto al editor. El sistema está preparado para leer y guardar las preferencias del usuario desde un archivo `userPreferences.lua`, almacenado junto al ejecutable del editor, y poder utilizarlas durante el código, haciendo una llamada al método `getPreference()` con el identificador de la preferencia. El proyecto, de momento, solo almacena la preferencia de idioma del usuario, pero está implementado de manera que puede ser escalable a cualquier preferencia nueva que se quiera añadir. Las preferencias se guardan automáticamente cuando se modifican por parte del usuario, por lo que no es necesario tener que hacer un guardado manual de estas.
- **ProjectManager:** gestiona todos los proyectos creados por el usuario en su ordenador. **ProjectManager** se encarga de leer el fichero `projects.lua`, que contiene todas las rutas de todos los proyectos y almacenar instancias de la clase `Project` sin inicializar⁶. Posteriormente, se comprueba que la ruta guardada por el editor en el fichero `projects.lua` existe y en ella se encuentra un archivo `ProjectSettings.lua`. En el caso de que este fichero no exista o tenga un formato incorrecto, el gestor ignora dicha entrada y la interfaz del editor se encargará de mostrar una advertencia al usuario. El gestor permite también eliminar proyectos, crear nuevos, añadir otros ya existentes y reubicar aquellos en los que no se encuentre un fichero `ProjectSettings.lua` válido.

Cabe destacar, que todos estos gestores están implementados utilizando un patrón *singleton*, por lo que una única instancia es válida para todas las clases del proyecto y no es necesario tener que estar gestionando el paso de punteros entre las distintas clases.

En conjunto, estos gestores permiten que el editor funcione de manera modular y adaptable a las necesidades del usuario, facilitando tanto la personalización como la persistencia de datos y configuraciones entre sesiones.

⁶Si se inicializase la instancia de `Project` en el momento de la lectura, y, el usuario tuviese creados muchos proyectos, sería un gran gasto de memoria innecesario, por lo que la instancia solo se inicializa en el momento en el que el editor va a necesitar poder obtener sus datos y modificarlos.

6.1.3. Elementos de dibujado

Los elementos de dibujado son los responsables del dibujado en pantalla de la interfaz del editor. Estas clases están diseñadas para trabajar conjuntamente con la librería `DearImGui` y el motor de renderizado de `SDL3`, facilitando la gestión de la interfaz gráfica y el contenido visual del entorno de desarrollo.

- `RenderObject`: es una interfaz que implementan todos los objetos que dibujan algo en pantalla. Define el método `render()`, que las clases que implementen esta interfaz tienen que redefinir.
- `WindowStack`: es una pila de objetos `RenderObject` a *renderizar* de abajo a arriba⁷, es decir, se dibujará primero el elemento que esté en la parte inferior de la pila, y sucesivamente hasta el superior. Esta clase contiene métodos para añadir y eliminar objetos de `render`, y un método, `renderWindows()` para poder recorrer todos estos elementos en orden y llamar a su método `render()`.
- `RenderManager`: se encarga de gestionar todo el apartado gráfico del editor, y, como el resto de gestores, está implementado utilizando un patrón *singleton*. Inicializa la ventana de la aplicación, así como el aparado gráfico de `SDL3` y `DearImGui`, y tiene métodos para el manejo y obtención de parámetros de la ventana principal de la aplicación, métodos que permiten la carga de un fichero `.ttf` de fuentes vectoriales o que permiten la carga de texturas e imágenes. Por otra parte, cuenta con el método `render()`, llamado desde el bucle principal del editor, y que gestiona el ciclo de dibujado de la aplicación, llamando a la función `renderWindows()` de `WindowStack` y posteriormente vuelca en la pantalla del usuario todo el contenido de `render` actualizado desde la última llamada.
- `Window`: es una interfaz que implementan todas las ventanas principales de la aplicación. Esta interfaz, que hereda de `RenderObject`, implementa el método `render()` de la siguiente forma:

```
void Window::render() {  
    beforeRender();  
    ImGui::Begin();  
    onRender();  
    ImGui::End();  
}
```

De esta manera, cada ventana que se quiera generar implementando esta interfaz solo tiene que redefinir los métodos `beforeRender()`, que se usa principalmente para modificar atributos asociados a `DearImGui`; y el método `onRender()`, que contiene el dibujado propiamente dicho del contenido de la ventana.

⁷Esto, que puede parecer contradictorio ya que en principio rompería el propósito de una pila, cuya función es solo tener acceso al elemento más superior de esta. El diseño de esta pila está pensado para, gestionar el añadir y quitar ventanas como una pila tradicional, pero poder recorrerla para su renderizado de abajo a arriba.

- **Subwindow:** se denomina «subventana» a todos aquellos elementos que componen una ventana (es decir, una ventana está compuesta por subventanas). Estas «subventanas» son un *wrapper*⁸ de las clases **Child** pertenecientes a la librería **DearImGui**. La implementación de esta clase es idéntica a la de la clase **Window**, sustituyendo las llamadas a `ImGui::Begin()` por `ImGui::BeginChild()`.
- **ModalWindow:** una ventana modal es una ventana o caja de diálogos que aparece en la interfaz y que bloquea la interacción con el resto de la aplicación hasta que el usuario la cierra o responde. Este tipo de ventanas se utiliza para acciones críticas o que requieran de atención inmediata, por ejemplo, asistentes de creación de elementos (*wizards*). **DearImGui** permite el uso de modales utilizando las clases **Popup** y **PopupModal**; sin embargo, este uso tiene una limitación, y es que solo un modal puede estar activo en un determinado momento, y para poder lanzar otro es necesario cerrarlo. Sin embargo, esta limitación no ha entorpecido el desarrollo del proyecto, y se ha podido desarrollar toda la funcionalidad sin problema ninguno.
- **WindowItem:** esta clase es un *wrapper* de la clase **TabItem** de **DearImGui**, que representa una pestaña dentro de una barra de pestañas (**TabBar**). Esta clase permite organizar el contenido en secciones separadas dentro de una misma ventana, sin la necesidad de tener que gestionarlo mediante «subventanas» y con la posibilidad del cambio entre pestañas sin abrir nuevas ventanas.

En conjunto, estos elementos permiten una arquitectura modular y extensible para la construcción de la interfaz gráfica del editor, facilitando tanto el mantenimiento del código como la integración de nuevas funcionalidades visuales de forma ordenada y eficiente.

6.1.4. Recursos

Los recursos permiten representar en el editor todos aquellos elementos que posteriormente serán utilizados por el motor del juego para su funcionamiento.

- **EditorResource:** es la interfaz común que deben implementar todos los recursos. Estos tienen que tener, necesariamente, sobrescritas e implementadas las siguientes tres funciones:
 - **readFromLua()**: lee los datos desde un fichero `.lua` de uno de los recursos utilizando llamadas a métodos de **LuaManager**.
 - **writeToLua()**: implementa la escritura de uno de los recursos a un fichero `.lua` para su guardado en disco, también utilizando llamadas a métodos de **LuaManager**.
 - **writeToEngineLua()**: implementa la «traducción» de uno de los recursos del editor a la sintaxis esperada por los recursos del motor, escribiéndolo también en un fichero `.lua`.

⁸Es decir, un envoltorio o estructura que encapsula una funcionalidad para simplificar su funcionamiento y uso.

- **Sprite:** define un *sprite*, es decir, una imagen bidimensional que representa un objeto o un personaje en pantalla. De los *sprites* se guarda su textura (generada utilizando el *RenderManager*) y las dimensiones de esta, así como métodos que permiten su obtención y modificación.
- **Animation:** define una animación, es decir, una lista de **Sprite** que actúan como fotogramas y que se suceden rápidamente para representar movimiento o cambios visuales de un objeto. De las animaciones se guarda una lista de **Sprite**, el tiempo entre fotogramas (fijado en el proyecto de 0.01s hasta 5s) y si la animación se repite en bucle.
- **Tile:** representa una baldosa, es decir, la unidad básica para construir los mapas en el editor. La baldosa es una imagen de dimensiones fijas (en el caso de este Proyecto, mínimo de 8x8 píxeles y máximo de 256x256), y las baldosas se generan a partir de un **Tileset**, por lo que simplemente se almacena su textura y su índice dentro del *tileset*.
- **Tileset:** representa una colección de objetos **Tile** organizados en una sola imagen (habitualmente en forma de cuadrícula). Es decir, en el caso del editor, un **Tileset** es una lista de **Tile**, ordenados de izquierda a derecha y de arriba a abajo. Para ahorrar trabajo a la hora de establecer las colisiones en el mapa a los usuarios, cada una de las *tiles* tiene asociada un *booleano* indicando si se puede colisionar o no con ese tipo de baldosa en concreto.
- **Event:** los eventos, explicados en el apartado 5.4, permiten agregar lógica a un juego sin la necesidad de programar. Cada evento tiene asociado uno o varios comportamientos y una única condición de ejecución. Estos comportamientos y condiciones corresponden a las clases **EventBehaviour** y **EventCondition**. Las condiciones se ejecutan en orden, de arriba a abajo, por lo que la clase **Event** permite el cambio de orden de ejecución de las condiciones. Tanto los eventos como las condiciones se crean utilizando el «patrón factoría», por lo que también se disponen de las clases **EventBehaviourFactory** y **EventConditionFactory**, que en su método **Create()** devuelven bien un comportamiento o una condición solamente con el identificador de cada uno de ellos. A continuación, se explican brevemente cada uno de los comportamientos:
 - **AnimationBehaviour:** permite reproducir, detener, reiniciar y cambiar una animación.
 - **ChoicesBehaviour:** permite generar, en un diálogo, tres opciones como máximo a escoger y cambiar el valor a una variable dependiendo de la opción elegida.
 - **DialogueBehaviour:** permite mostrar un texto en pantalla.
 - **JumpBehaviour:** permite hacer un salto a otros comportamientos, parecido a la instrucción **goto** en algunos lenguajes de programación.
 - **JumpIfBehaviour:** permite hacer un salto a otros comportamientos si se cumple una determinada condición, es decir, es un salto condicional.

- **ModifyVariableBehaviour:** permite cambiar el valor de una variable, bien sea de un objeto o de un jugador.
- **MoveBehaviour:** permite mover un objeto un determinado número de casillas, tanto en horizontal como en vertical.
- **MusicBehaviour:** permite reproducir, detener, reanudar, pausar, cambiar audio, cambiar volumen y establecer si una pista de sonido está o no en bucle.
- **PlaySFXBehaviour:** permite reproducir un efecto de sonido.
- **WaitForBehaviour:** hace una espera hasta que se cumpla una determinada condición.

Y, a continuación, se explican brevemente cada una de las condiciones:

- **AlwaysCondition:** esta condición siempre se cumple. Equivale al `true` en los lenguajes de programación.
 - **AndCondition:** esta condición permite concatenar otras dos condiciones, y se cumplirá siempre y cuando las dos condiciones que concatene se cumplan.
 - **BehaviourEndedCondition:** esta condición se cumple cuando un comportamiento termine.
 - **CollidesWithPlayerCondition:** esta condición se cumple cuando un objeto colisione con el jugador.
 - **InteractionCondition:** esta condición se cumple cuando se interactúe con un objeto.
 - **NotCondition:** niega todas las condiciones, por lo que se cumplirá siempre y cuando la condición que niegue no se cumpla.
 - **OnStartCondition:** esta condición se cumple al inicio de un evento.
 - **OrCondition:** esta condición también permite concatenar otras dos condiciones, y se cumplirá siempre y cuando al menos una de las dos condiciones que concatene se cumpla.
 - **TimePassedCondition:** esta condición se cumple cuando haya transcurrido un determinado tiempo.
 - **ValueEqualsCondition:** esta condición se cumple cuando el valor de una variable equivalga a un determinado valor.
- **Object:** define un objeto de juego. Esta clase añade métodos para establecer la posición del objeto en el mapa, establecer la capa del mapa en la que se encuentra, añadir o eliminar variables y eventos, y establecer su **Sprite**.
 - **Map:** define un mapa. Los mapas tienen un máximo de 16 capas, y están compuestos por elementos de la clase **Tile**. Los mapas, a su vez, tienen colisiones en cada una de las casillas de la cuadrícula y también pueden albergar un objeto por cada una de ellas.

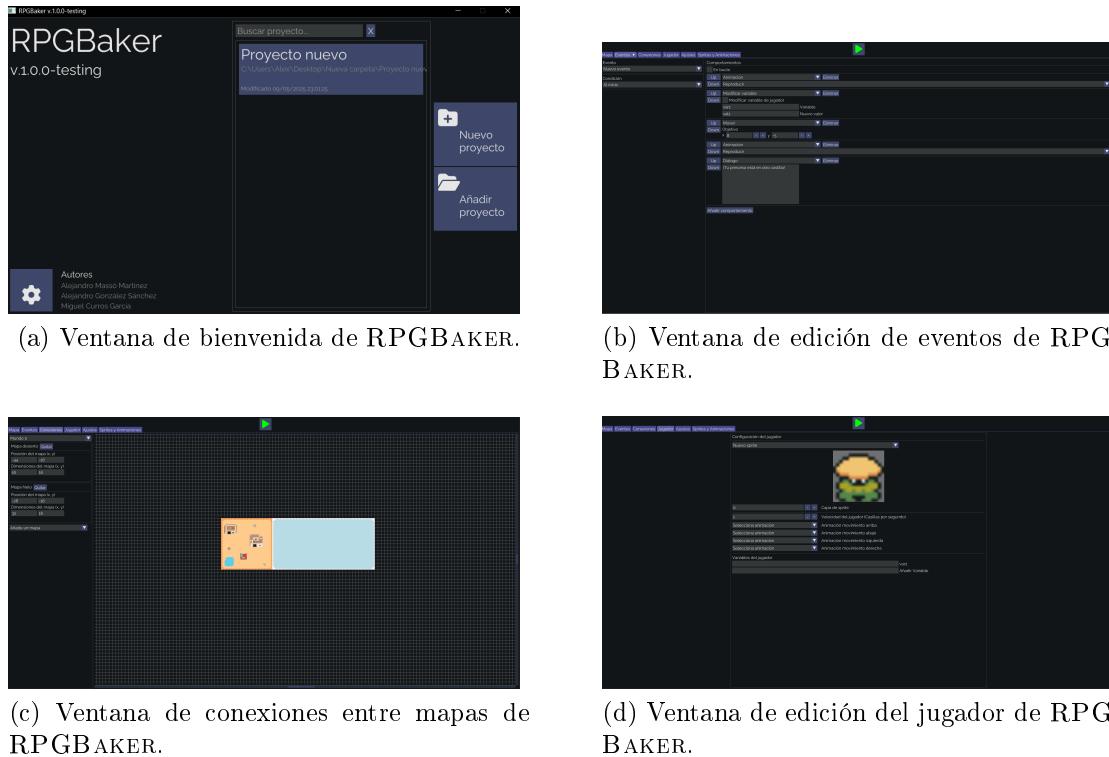


Figura 6.3: Distintas ventanas del editor de RPGBAKER.

Estas clases constituyen la base del sistema de edición de datos del proyecto, proporcionando estructuras reutilizables y extensibles que permiten almacenar, modificar y traducir los datos del editor al formato legible del motor, facilitando la interacción entre el editor y el entorno de ejecución.

6.2. Interfaz del editor

En cuanto a la interfaz que compone el editor, toda ella está elaborada utilizando la librería DearImGui junto con todos los elementos de dibujado mencionados en el apartado 6.1.3.

- La ventana de bienvenida (figura 6.3a), que permite la gestión de todos los proyectos creados por el usuario, así como crear uno nuevo y añadir otro ya existente, y también cambiar el idioma de la interfaz.
- La ventana principal de la aplicación, que permite la edición de un proyecto en concreto y está dividida en varias pestañas, cada una con una función distinta:
 - El editor de mapas (figura 4.1), permite la creación y modificación de un mapa utilizando diversos *tilesets*. En la parte izquierda se encuentra el visor de *tilesets*, que permite crear nuevas a partir de un fichero de imagen, editar las ya presentes o eliminarlas. En la parte central se encuentra el editor *per se*, mostrándose una cuadrícula con el tamaño elegido por el

usuario. Esta cuadrícula se puede ampliar y alejar, y tiene varios modos de visualización:

- Visualización de la capa actual exclusivamente.
- Visualización de la capa actual y las de abajo con transparencia.
- Visualización de la capa actual y las de abajo completamente opacas.
- Visualización de todas las capas en conjunto, empezando desde abajo hasta arriba.

Finalmente, en la parte derecha, se encuentra el editor de objetos, que permite crear y eliminar un objeto y agregarle o eliminarle un *sprite*, un evento o una variable.

- El editor de eventos (figura 6.3b), que permite la creación y edición de eventos, añadiendo los comportamientos esperados y las condiciones de lanzamiento. Eventos complejos se encapsulan dentro de otros y todos se pueden ordenar por orden de prioridad de ejecución utilizando las flechas que aparecen al costado izquierdo.
- El editor de conexiones (figura 6.3c), que permite crear mundos⁹ y añadirle los mapas, modificando en la cuadrícula la posición final de cada uno de los mapas.
- El editor del jugador (figura 6.3d), que permite establecer ajustes del jugador, como por ejemplo su *sprite*, sus animaciones de movimiento en las cuatro direcciones cardinales básicas y variables propias de este.
- El editor de ajustes, que permite editar información relevante al ejecutable final, como por ejemplo, el nombre de la ventana, las dimensiones que la cámara puede mostrar en pantalla de una sola vez, el mapa en el que empezará el jugador o la fuente por defecto que mostrarán los textos.
- El editor de *sprites* y animaciones, que permite generar un *sprite* a partir de un *spritesheet* o de una imagen, estableciendo el *offset* que se desee, y, posteriormente, a partir de los *sprites*, poder generar una animación. El asistente de creación de animaciones contiene un visor en tiempo real de la animación, así como controles para poder navegar por los diversos fotogramas.

⁹Un mundo, en la jerarquía de nuestro Proyecto, es aquel elemento que puede contener diversos mapas.

Capítulo 7

Evaluación y análisis

Al conocer en profundidad cómo funcionan todos los sistemas del motor y el editor, no se puede tener la certeza de que estos funcionen como un usuario promedio espera. Por esto, realizar pruebas con usuarios sin sesgos es crucial para obtener la información de las posibles brechas entre los programas y esos usuarios. De cara a ofrecer una experiencia de uso agradable se analizarán las pruebas en busca de formas de mejorarla. Se buscarán distintos tipos de perfiles para obtener una mayor cantidad de datos sobre los posibles conflictos que estos tengan con los sistemas.

7.1. Objetivo de la evaluación

Los objetivos de las pruebas con usuarios son principalmente dos. El primero es evaluar si los usuarios entienden los sistemas y son capaces de usarlos cómodamente. Es importante que el entorno sea agradable de usar para permitir que los usuarios entiendan las capacidades del mismo y las puedan aprovechar. El segundo es evaluar si cada usuario es capaz de usar el editor de forma creativa, de tal manera que los videojuegos que cree un usuario supongan una experiencia diferente a los que haga otro, comprobando así la flexibilidad de uso del editor.

7.2. Metodología

Para llevar a cabo estas pruebas, una vez definidos los objetivos, se diseñó el procedimiento a seguir. Cada usuario realizará las pruebas de manera individual, acompañado por un supervisor, y con una duración estimada de una hora.

Al inicio, se le pregunta a cada probador sobre su experiencia tanto en jugar como en crear videojuegos, haciendo especial énfasis en los RPG. Con esta información recopilada, se le entregan varios archivos: el ejecutable del editor, un paquete de recursos para evitar que se pierda tiempo buscando durante la prueba, y una breve guía de uso del editor.

Esta guía explica de forma concisa cómo utilizar cada uno de los sistemas disponibles en el editor. Con esta información, el probador seguirá la guía para entender los mecanismos y, posteriormente, podrá usarlos libremente según su criterio.

El investigador será responsable de registrar todas las interacciones relevantes de los probadores con el editor, además de brindar asistencia en caso de que surja algún error durante las pruebas. Para observar una experiencia lo más real posible, el investigador no intervendrá en las acciones del usuario, salvo en situaciones excepcionales, como errores del programa o bloqueos en los que el probador no pueda continuar debido a alguna dificultad.

Estas pruebas están diseñadas para realizarse tanto de manera presencial como telemática, adaptándose a la mayor comodidad de cada caso particular. Al finalizar la prueba, se solicitará al usuario una valoración general sobre los sistemas y su experiencia de uso, incluyendo sus quejas e ideas.

7.3. Resultados

Los usuarios pueden agruparse en 3 categorías según su experiencia con los videojuegos y, en particular, con los RPG.

En primer lugar, están los usuarios menos familiarizados con el mundo de los videojuegos, con poca experiencia jugando en general y, en particular, con los juegos RPG. Con este tipo de usuario, el objetivo es evaluar qué tan intuitivas son las herramientas y la interfaz de usuario del editor, así como comprobar si se entienden bien las instrucciones de las pruebas. Este primer grupo logró crear mapas con cierto sentido, pero la mayoría utilizó únicamente una o dos capas, lo que limitó un poco el diseño. No dedicaron mucho tiempo a explorar los *tilesets* y realizaron diseños bastante sencillos. Las configuraciones las entendieron bastante bien y, siguiendo la guía, lograron completarlas. En la creación de eventos se quedaron en los comportamientos más simples y no intentaron crear eventos complejos usando variables y condiciones. En general, la prueba fue satisfactoria y los resultados estuvieron dentro de lo esperado.

En segundo lugar, están los usuarios que sí juegan videojuegos y algunos incluso están relacionados con la industria, pero no tienen conocimientos de programación o desarrollo más allá de sus áreas específicas. Estos usuarios aprovecharon mucho más el apartado de *mapeado*, utilizando más capas y varios *tilesets* para crear mapas mucho más interesantes. En cuanto a los eventos, se les animó más que al primer grupo a intentar crear eventos algo más complejos, pero al parecer no tenían claro cómo estructurarlos para aprovechar al máximo las funcionalidades, y la mayoría se quedó en cosas más sencillas o requirió bastante ayuda por parte de los supervisores. Aunque los eventos no requieren conocimientos de programación, su estructura es similar a la de un sistema basado en lógica condicional, lo que exige más tiempo del disponible durante la prueba para entender cómo organizarlos correctamente.

Por último, está el grupo de personas con conocimientos en el desarrollo de videojuegos, incluso en el desarrollo específico en *RPG Maker*, lo cual fue muy útil para obtener *feedback* comparativo. Este grupo comprendió bien las herramientas de *mapeado* y señaló que echaba en falta algunas opciones más avanzadas que agilizaran el proceso, como la posibilidad de seleccionar varios *tiles* a la vez o llenar áreas con *tiles* de un mismo tipo. También comentaron que el proceso de creación de *sprites*

y animaciones es lento, proponiendo algún tipo de automatización similar a la que existe para los *tilesets*. En cuanto a los eventos, lograron entender cómo montar estructuras más avanzadas, lo cual fue un resultado positivo.

Adicionalmente a todas las observaciones y el *feedback* recibido, las pruebas también sirvieron para detectar varios *bugs* y problemas de funcionamiento, lo que fue de gran ayuda para dar un pulido final a la versión del editor.

7.4. Análisis de los resultados y conclusiones

Aunque las pruebas no alcanzaron la extensión ni la variedad deseadas debido a limitaciones logísticas, se pueden extraer algunas conclusiones interesantes para el desarrollo actual y, sobre todo, para un posible trabajo futuro.

Una de las prioridades identificadas es mejorar la experiencia de diseño de mapas y la creación de *sprites* y animaciones. Con este objetivo, se proponen las siguientes mejoras funcionales:

- Incorporar la posibilidad de seleccionar varios *tiles* simultáneamente y pintarlos en el mapa de forma conjunta.
- Añadir una herramienta para llenar automáticamente una zona completa con el *tile* seleccionado.
- Permitir la selección de *tiles* directamente desde el mapa, sin necesidad de localizarlos manualmente en el *tileset*.
- Incluir una función para deshacer la última edición realizada.
- Automatizar el proceso de creación de múltiples *sprites* y su conversión en animaciones a partir de una única fuente, de manera similar al funcionamiento del editor de *tilesets*.

Además, se considera conveniente implementar mejoras en forma de *tooltips* en la sección de eventos, con el objetivo de explicar de manera clara todas las funcionalidades disponibles. También se sugiere incluir eventos predefinidos que sirvan como ejemplos prácticos para facilitar su comprensión y uso.

Capítulo 8

Conclusiones y trabajo futuro

Gracias al desarrollo de RPGBAKER se ha conseguido implementar una herramienta útil de cara a la creación de videojuegos 2D de estilo RPG. Aún siendo un proyecto con menor alcance a otros motores específicos del género, es capaz de generar juegos muy completos cubriendo muchas de las necesidades que estos requieren. Con él se acerca a un público sin experiencia en programación a la creación de este tipo de juegos de forma libre, abriendo las puertas a que una mayor cantidad de gente se pueda dedicar a ello. Además, al ser multiplataforma tanto el editor como el motor, se permite un mayor alcance y facilidades para los usuarios. Al serlo el editor se ofrece una mayor comodidad de cara a los creadores, adaptándose con mayor holgura a sus entornos de desarrollo. Siéndolo el motor se permite un mayor alcance en la publicación de los videojuegos, accediendo con el soporte de Android a uno de los mayores mercados en la industria como lo es el de los dispositivos móviles.

Con todo esto, aún cumpliendo todos nuestros objetivos propuestos, RPGBAKER podría llegar a ofrecer más herramientas que permitan añadir otros sistemas propios de los RPG. Como posibles ampliaciones en este aspecto se podrían crear, por ejemplo:

- Sistema de combate: Al tener típicamente esta clase de juegos enfrentamientos entre personajes, sería un muy buen añadido el permitir crear y personalizar estas batallas. Idealmente se permitiría personalizar, lo primero, a aquellos personajes que vayan a combatir, definiendo sus estadísticas, sus formas de ataque y las maneras en las que estos progresarían tanto del lado del jugador como de los enemigos. La segunda parte esencial para conformar este apartado sería la creación de escenas de combate. Siendo los enfrentamientos por turnos típicos en los RPG encajaría perfectamente en el entorno, se podría personalizar qué personajes se enfrentarían, en qué momento se escogerían las acciones de un turno y en qué orden actuarían los combatientes.
- Sistema de *items*: Otra parte clásica de los RPG son los *items*, objetos que puede guardar un jugador, como armas o pociones. Estos irían asociados a un inventario donde guardarlos que podría tener el jugador. Estos objetos aplicarían efectos que modificarían el estado de los personajes o el juego. Cada uno de ellos tendría alguno de estos efectos asociados que podrían aplicarse tanto navegando el mundo como en los combates dependiendo del caso.

En resumen, nuestro editor permite de forma accesible crear, con el motor, videojuegos RPG sencillos en los que navegar por un mundo de manera dinámica. Sobre esto, aún siendo ya una herramienta potente, se podrían hacer añadidos que ofrezcan una experiencia más amplia para crear RPG. Permitiendo así a aquella gente menos experimentada en el desarrollo construir juegos aún más completos.

Conclusions and future work

Thanks to the development of RPGBAKER, a useful tool has been implemented for the creation of 2D RPG-style video games. While it may have a smaller scope compared to other engines specialized in the genre, it is capable of producing highly complete games that cover many of the typical requirements. It aims to bring game development closer to users without programming experience, allowing them to freely create this type of game and thereby opening the door for a wider audience to get involved in game creation. Furthermore, since both the editor and the engine are multiplatform, this broadens its accessibility and usability. A multiplatform editor offers greater convenience for creators by adapting more easily to their development environments. Likewise, a multiplatform engine increases the potential reach of the published games, especially by supporting Android, one of the largest markets in the video game industry due to the popularity of mobile devices.

Despite having achieved all the proposed objectives, Baker could be expanded to include more tools that support additional systems typically found in RPGs. Some possible future extensions include:

- Combat system: Since RPGs commonly involve character battles, adding functionality to create and customize combat scenarios would be a valuable enhancement. Ideally, it would allow developers to define the combatants, set their statistics, determine their attack methods, and configure how they evolve—both for player characters and enemies. The second essential component would be the creation of combat scenes. Given that turn-based combat is a classic element of RPGs, it would fit naturally within this environment. Users could define which characters take part, when each turn’s actions are selected, and in what order the combatants act.
- Item system: Another staple of RPGs is the use of items—objects that the player can collect, such as weapons or potions. These would be linked to an inventory system that the player can manage. Items would apply effects that alter character states or game conditions. Each item would have one or more associated effects, which could be applied both during world exploration and in combat, depending on the context.

In summary, our editor allows users to easily create simple RPGs using the engine, enabling dynamic exploration of a game world. While already a powerful tool, additional features could further enhance the experience and broaden the possibilities for RPG creation. This would allow less experienced developers to build even more complete games.

Capítulo 9

Contribuciones Personales

Miguel Curros García

Mis tareas en el proyecto se centraron en el diseño y desarrollo del motor genérico de videojuegos así como dentro del apartado de *gameplay* todo lo relacionado con el sistema de eventos, tanto a nivel de motor como de editor. Adicionalmente, también creé la gestión de persistencia con lectura y escritura de los archivos de datos específicos del editor y parte del proceso de *build*.

En primer lugar, participé en conjunto con Alejandro González en la fase de diseño del motor donde dejamos bien claras cuáles serían las distintas capas de abstracción de la implementación. Desde el principio teníamos claro que la base del motor iba a estar basada en una arquitectura EC (*Entity-Component*), construyendo el resto de las funcionalidades alrededor. Comenzamos a definir cómo estructuraríamos esa arquitectura a través de un diagrama *UML* donde definimos tanto los distintos módulos que formarían el motor como la forma en la que se conectarían las partes más básicas de este con las funcionalidades específicas de *gameplay*.

A continuación, comencé con el desarrollo de este centrándome en los apartados de carga y gestión de recursos, audio y colisiones. Comenzando con la gestión de recursos, esta es una parte crucial del motor al estar este dirigido por datos; todo aquello que se quiere que ocurra en un juego estará definido en los datos a interpretar al motor. Esta parte tiene 3 partes clave: **Resource**, **ResourceHandler** y **ResourceMemoryManager**. Cada una de esas clases se encarga de una parte clave de la carga y descarga de recursos.

- **Resource** servirá de clase base para todos los tipos de recursos que se quieran cargar, cada uno de ellos implementará cómo se carga y descarga desde los archivos de datos.
- **ResourceHandler** será a quien recurran las distintas partes del motor que necesiten cualquier tipo de recurso. A través de la ruta a un recurso de un tipo especificado dará acceso a una instancia conteniendo ese mismo recurso cargado.
- Por último, **ResourceMemoryManager** se encargará de gestionar cuánta memoria está siendo ocupada en este momento por los recursos. A partir de un

límite especificado en un archivo de configuración esta clase se encargará de que no se supere ese umbral de memoria máxima ocupada por los recursos. Si se solicitara un recurso y no hubiera memoria suficiente para cargarlo a través de un algoritmo LRU se irá liberando memoria de otros recursos hasta que haya suficiente para cargar el nuevo. Con esto, conseguí estructurar la carga de recursos bajo demanda que evitaría posibles largos tiempos de espera los juegos.

Una vez esto estaba listo desarrollé el sistema de sonido que permitiría el control sobre la reproducción de archivos de audio dentro de los juegos. Para esto decidimos usar el nuevo sistema de audio de *SDL3* que cumplía todas nuestras necesidades. Nuestro objetivo con este sistema era la implementación de un componente **AudioSource** a través del que gestionar la reproducción de un archivo de audio. Además, queríamos que cada uno de estos **AudioSource** pudieran estar asociados a un conjunto de sonidos, pudiendo tener de esta forma un control más profundo del volumen; pues es típico y útil en los videojuegos permitir al jugador controlar el volumen general, de la música o de los efectos por separado.

Para poder reproducir un sonido creé una clase **AudioClip** que se encargaría de envolver las funcionalidades de **SDL_AudioStream**; la base de la reproducción de sonidos en *SDL3*; y ofrecer una interfaz acorde a nuestras necesidades. Para poder asignar estas pistas a distintos conjuntos de sonidos implementé la clase **AudioMixer**, que tendría un control de volumen asociado que se aplicaría a cada uno de sus **AudioClip**. Una vez con estas clases básicas pude implementar **AudioSource** que las utilizaría para ofrecer su funcionalidad esperada.

Por último en la base del motor, para la detección y gestión de colisiones buscábamos algo sencillo. Por el tipo de videojuegos que ofrecemos implementar basta con comprobaciones de colisiones entre formas rectangulares. Son comprobaciones sencillas y *SDL* ya ofrece funciones para comprobar si dos rectángulos tienen intersección. Con esto, la implementación de un componente **Collider** fue sencilla. A través de una clase **CollisionManager**, en cada actualización del juego se comprobarían las colisiones entre estos **Collider**. En estas comprobaciones se guardaría la información de qué **Collider** está colisionando, acaba de empezar a colisionar o acaba de dejar de colisionar con otro. De esta manera se ofrece un control sencillo para implementar comprobaciones dependientes de colisiones entre elementos de juego.

Una vez terminamos el apartado del motor genérico comenzamos con el desarrollo de partes específicas de *GamePlay* donde centré mis aportaciones en el sistema de eventos. De cara a ofrecer una experiencia personalizable de crear un videojuego sin necesidad de programar un punto clave era nuestro sistema de eventos. Este se centra alrededor del componente **EventHandler** que sirve como conjunto de eventos de una entidad. Cada **Event** está compuesto por una condición, **EventCondition**, y comportamientos, **EventBehaviour**.

- Comenzando por las condiciones, estas se encargan de manifestar si el estado del juego es el que esperan. Para ello, cada una de las condiciones debe implementar sus propias comprobaciones, esto lo hice a través de un sistema de

herencia. Además, igual que pasa con los componentes, cada una de estas condiciones deben poder crearse a partir de los datos proporcionados por el juego, por esta razón creé una clase `EventConditionFactory`, que se encargaría de instanciar el tipo correspondiente de `EventCondition` dado un identificador.

- Los comportamientos decidimos que estuvieran implementados en *Lua*. De esta manera se podrían ampliar las funcionalidades de un juego sin necesidad de recompilar el motor, ofreciendo una experiencia más flexible. Para implementarlo de la manera que diseñamos era necesario poder tener algún tipo de POO (Programación orientada a objetos) en *Lua*, pero no es algo que ofrezca el lenguaje por defecto. A través de asignar funciones a tablas y modificando sus *metatablas* pude obtener un comportamiento similar a las clases y la herencia. A partir de esa base implementé la clase `EventBehaviour` en *C++* envolviendo a cada instancia que hubiera en una escena de los distintos comportamientos implementados en *Lua*. Además de esto, para poder implementar cada uno de los comportamientos fue necesario definir desde *C++* qué clases y cómo se podían modificar dentro de las implementaciones de estos en *Lua*.

Cuando esto estuvo listo comencé con el desarrollo del editor donde creé la gestión de persistencia con lectura y escritura de los archivos de datos específicos del editor. Estos datos decidimos que íbamos a guardarlos también en *Lua* pues nos sería más sencillo de implementar al tener ya la infraestructura montada para ello. Gracias a la clase del editor `LuaManager` creada por Alejandro Massó, que permite acceder a tablas de *Lua* de un archivo, crear nuevas y guardar estas en nuevos archivos, la tarea consistió en escribir y leer estas tablas. Cada vez que se quisiera salvar un proyecto cada uno de sus recursos guardarían sus parámetros en nuevas tablas de *Lua* que se escribirían en archivos en subdirectorios específicos dentro de un directorio «`projectfiles`» dentro de la ruta del proyecto.

A continuación desarrollé todo el apartado de Eventos, desde la creación, edición y posterior traducción a archivos preparados para ser leídos por el motor. Para este apartado lo primero que hice fue crear una nueva pestaña en el editor, la pestaña de edición de eventos. En esta se podría ver un desplegable donde escoger un evento, una sección donde se mostraría la condición del evento y otra sección donde se mostrarían los comportamientos. La parte de creación y selección de eventos fue sencilla, aprovechando que Alejandro Massó ya había implementado la creación y selección de mapas, reutilicé el código que pude para esta parte. Para hacer los apartados de la condición y los comportamientos, igual que con la condición en el motor, tuve que crear factorías que permitieran crear los distintos tipos de instancias de cada uno de estos a partir de identificadores. Esto es porque cada condición y cada comportamiento debe definir en el editor su propia clase, ya que su persistencia e interfaz gráfica difieren entre sí, de modo que requieren implementaciones distintas; además por esa misma persistencia es por lo que se necesita la factoría, cada evento puede tener cualquier tipo de condición o comportamiento y se debe poder reconstruir al abrir un proyecto guardado.

Una vez completé todas las interfaces gráficas y la persistencia de todos los tipos de condiciones y comportamientos comencé con el proceso de *build*; convertir estos

datos a la sintaxis que el motor reconoce. En el caso de las condiciones fue sencillo, pues el formato en el que guardan su información en el editor es casi idéntico a lo que necesita el motor. Lo que refiere a los comportamientos no es así, hubo dos puntos clave en este apartado: dependencias de componentes y formato de escritura. Algunos de los comportamientos en el motor funcionan por su cuenta, es decir, con que existan dentro de su `EventHandler` cumplirán su función sin problema; en cambio hay otros que necesitan otros componentes para funcionar también de modo que, para que su proceso de *build* fuera correcto necesitaban indicarle a su entidad que escribiera los componentes faltantes con los parámetros correctos. Por último, a diferencia del resto de proceso de *build* este apartado no podía hacer uso del mecanismo que usamos para escribir el resto de tablas de *Lua*. Esto ocurre porque no estamos añadiendo otra tabla al uso, se necesita la llamada a la construcción de ese `EventBehaviour` porque el mecanismo de persistencia que se usa en el resto del editor no graba las *metatablas* asociadas a cada tabla, y esto es crucial de cara al funcionamiento de los comportamientos.

Alejandro González Sánchez

Mi contribución al Proyecto se ha centrado en el diseño e implementación del motor y parte del editor. En primer lugar, realicé una investigación sobre las posibles formas de adaptar el motor para que pudiera ser multiplataforma, realizando unas primeras pruebas técnicas y de concepto, y acabé decantándome por utilizar `SDL` como núcleo central del motor, principalmente alentado por todo el soporte multiplataforma que este aporta.

Una vez concluido esto, y con una versión básica de «juego» funcional tanto en *Android* como en *desktop*, desarrollamos un pequeño programa que mostraba un rectángulo de colores en pantalla, cuyos atributos (tamaño y color) se leían de un archivo `.lua`.

A continuación, empecé junto a Miguel Curros García la fase de diseño del motor. En este paso intentamos dejar bien definidas las diferentes partes que íbamos a necesitar, así como la forma en que se comunicarían entre sí. Para ello, generamos un diagrama UML de los diferentes componentes que íbamos a utilizar y lo separamos en diferentes niveles de abstracción, dejando clara la separación entre los componentes básicos del motor y los componentes de *gameplay* que íbamos a necesitar para poder implementar todas las funcionalidades específicas que queríamos para los juegos que ofrece nuestro editor.

Una vez cerrado este diseño, comenzamos con la implementación. Yo me centré en toda la parte de `Render`, `Core` (sistema de Entidades y Componentes) e `Input`. En primer lugar, desarrollé el *core* del motor: la estructura de Entidades y Componentes, todas ellas organizadas en escenas y gestionadas por un `SceneManager`. Creé cada una de las partes del ciclo de vida de estos elementos y monté toda la estructura para que pudieran ser cargados a partir de datos leídos de un archivo *Lua*. Para esto, nos apoyamos en `ComponentFactory` y en la clase `ComponentTemplate`, que, a través de una macro y una estructura de plantilla, aceleraba mucho el proceso de declarar nuevos componentes. A continuación, creé la clase `ComponentData`, que utilizariamos

para poder parametrizar los componentes que declarásemos en Lua. Finalmente, creé unos métodos en el `SceneManager` para poder instanciar entidades y escenas a partir de unos *blueprints* creados leyendo los archivos Lua correspondientes.

Una vez terminado esto, pasamos a la parte del *renderizado*. Aquí, generalicé los métodos de pintado de `SDL` y creé un bucle de *clear*, *present* y *render*, que se llama desde el bucle principal. Para acceder a las funciones de pintado, creé una clase virtual `RenderComponent` que implementa el método `render(RenderManager*)`. A continuación, creé todos los componentes básicos de *renderizado* que íbamos a necesitar (`Camera`, `Rectangle`, `Text`, `SpriteRenderer`, `Animator`), así como los recursos que estos iban a utilizar (`Sprite`, `Animation`, `Font`, `Color`). Todo esto quedó integrado con el ciclo de vida que creé en el apartado de `Core`, para permitir su inicialización parametrizada desde Lua.

Finalmente, hice una implementación sencilla de un sistema de *input* al que se pudiera acceder desde los componentes. Dado que, por diseño, solo íbamos a utilizar *clics/touchs* para mantener de una forma más simple el soporte multiplataforma, generalicé estos en un `struct`. Adicionalmente, creé el componente `Button`, al que se le podía asignar una función de Lua que se llamaría con unos parámetros preestablecidos al detectar un *input* en su área.

Con esto y las aportaciones de Miguel, dimos por terminado el motor genérico y pasamos a implementar los elementos específicos de *gameplay*. Aquí desarrollé un sistema de diálogo formado principalmente por dos componentes: un gestor de *textboxes*, que mostraba texto poco a poco y esperaba el *input* del usuario, y unas opciones compuestas por varios botones con posibles respuestas por parte del usuario. Lo siguiente fue crear un sistema de movimiento basado en A^* , que utilizarían tanto los NPC como el jugador a partir de un input de tipo *point and click*; también añadí la opción de aplicar animaciones como parámetro a este movimiento. Una vez cerrado esto, creé un gestor de mundo cuya función sería controlar qué mapas están activos en escena en cada momento, instanciando nuevos en caso necesario a partir de los *blueprints*. Todos estos sistemas serían la base del *gameplay* de *overworld* que ofrece nuestro editor, combinado con el sistema de eventos.

Con esto listo, pasamos al desarrollo del editor, que ya tenía una base implementada por Alejandro Massó. Aquí me centré, en primer lugar, en generar un sistema de traducción que convirtiera los datos generados por el editor en datos que siguieran la estructura del motor (entidades, componentes, escenas, *sprites* y animaciones), así como en la generación de otros archivos de configuración. A partir de esto, concreté un proceso de *build* que se encargaría de obtener los binarios precompilados del motor en formato ejecutable y los combinaría con los *assets* del usuario y los archivos de datos en formato motor, previamente traducidos, para obtener el producto final: el juego, contenido en un único directorio denominado `Build`.

Posteriormente, implementé otras funcionalidades del editor, como el inspector de objetos, la pestaña de conexiones entre mapas, la pestaña de configuración del jugador y la de configuración general. Para esta última, quise implementar una previsualización para el texto del jugador, por lo que tuve que gestionar una carga asíncrona de fuentes con `DearImGui`. Todas estas funcionalidades supusieron también una ampliación del sistema de traducción y *build* previamente comentado.

Por último, me dediqué a realizar pruebas con usuarios y corregir los problemas

que encontrábamos en estas. Finalmente, adapté la funcionalidad de *build* para que también fuese posible generar una APK lista para usar en dispositivos Android.

Alejandro Massó Martínez

Mi contribución al proyecto se ha basado en la investigación, diseño y desarrollo del editor, la creación de la *toolchain* de generación de dependencias y compilación de los distintos subproyectos, y la redacción de este documento.

En primer lugar, hice una exhaustiva investigación acerca de distintos editores de videojuegos, tanto específicos de RPG como generalistas, y, con la información obtenida, comencé a dibujar algunos pequeños bocetos de la interfaz de la aplicación, así como recopilar las características más comunes de ambos, para poder implementarlas finalmente en nuestro editor.

Posteriormente, implementé, en un repositorio aparte del del desarrollo del proyecto, una primera versión de la estructura del proyecto y de la *toolchain*, utilizando CMake. Pese a que conseguí que se abriese una ventana tanto para el editor como para el ejecutable del juego final, no fui capaz de conseguir la vinculación del proyecto de CMake con *Android Studio* para la generación del APK.

Cuando mi compañero Alejandro consiguió la vinculación de *Android Studio* con el CMake principal del proyecto, creé el repositorio final y modifiqué un poco su estructura, dividiéndola en los actuales módulos «proyectos» y «código fuente», lo cual separaba la implementación del código de la declaración de los subdirectorios de CMake, algo que habíamos visto en la asignatura de «Proyectos 3» durante el transcurso del grado.

Sin embargo, estas modificaciones volvieron a complicar el enlazado de *Android Studio*, pero tras unos arreglos y ajustes, finalmente la estructura final del repositorio estaba lista, y mis compañeros, que ya habían acabado con gran parte del diseño del motor, podían comenzar a programar.

Una vez que la estructura del repositorio estaba terminada, y, con la investigación previamente realizada, comencé a diseñar un esquema inicial de clases para el editor utilizando la sintaxis UML. En el esquema, se detallaban las clases **Project** y **Window**, así como todos los distintos gestores que se iban a encargar de la inicialización y gestión de distintos módulos del editor. En un principio, el editor iba a utilizar código del motor, por ejemplo, toda la parte de *renderizado*; pero finalmente, y debido a la inclusión de la librería **DearImGui**, decidí compartimentar e independizar el desarrollo del editor y del motor.

Una vez que la estructura de la base estaba diseñada, comencé a implementar las clases en C++. Las clases estaban divididas en tres módulos: *comunes*, *lectura/escripción* y *dibujado*. Comencé por la implementación de las clases comunes, es decir, **Editor**, que es la clase que se encarga de la inicialización entera del editor y de la inicialización de los gestores; y **Project**, que contiene la información más relevante de cada uno de los proyectos o juegos que se puedan hacer con el editor.

Después, implementé todos los distintos gestores del editor, como el `RenderManager`, que se encarga del dibujado de la interfaz en pantalla o el `InputManager`, que se encarga de la gestión de eventos de entrada del usuario mediante periféricos. Desde un primer momento, quise que el editor pudiese tener la opción de estar en múltiples idiomas, por lo que diseñé e implementé esta opción con el `LocalizationManager`, que además soporta otros idiomas que todavía no estén incorporados en el editor siempre y cuando se cumpla con la sintaxis esperada.

Implementé también el gestor de las preferencias de usuario, `PreferencesManager`; el gestor de los proyectos del editor, `ProjectsManager`, que permite la carga y guardado de los distintos proyectos creados por el usuario; y, el gestor de Lua, `LuaManager`, no sin antes haber hecho la descarga y compilación de Lua modificando el CMake, que, afortunadamente, debido al trabajo realizado anteriormente, fue una tarea sencilla.

Luego de haber implementado todos los gestores que conforman la base del editor, era el turno de la parte de dibujado. La estructura diseñada previamente se conformaba de ventanas y «subventanas», por lo que implementé la clase `Window` y `Subwindow`, que se aprovechaban de los conceptos de *ventana* y *ventana hija* de `DearImGui`. Como las ventanas se tienen que mostrar en una determinada jerarquía, implementé también la clase `WindowStack`. De esta manera, las ventanas que llegan posteriormente y se meten a la pila se renderizan una encima de la otra, como en todos los *softwares* que requieren del uso de múltiples ventanas. Al haber varios tipos de ventana distintos (por ejemplo, las modales o las pestañas) tuve que implementar una interfaz base, `RenderObject`, que sirviese para todas.

Cuando la base de dibujado estaba acabada, mi tarea fue la de implementar otras librerías necesarias para el motor, como `SDL_ttf`, `SDL_image` o `sdl2`. `SDL_ttf` dio algunos problemas con la configuración en CMake, pero al cabo de unos días fue solucionado y pude seguir adelante con el editor.

Primero, modifiqué parte de la implementación que había hecha con Lua para utilizar `sdl2`, que iba a facilitar el desarrollo utilizando esta herramienta. Después, comencé a implementar las primeras ventanas del editor, como `WelcomeWindow`, que es la ventana de bienvenida que se ve al iniciar el editor. Para ello, tuve que buscar unas fuentes `.ttf` (decantándome al final por *Raleway*), así como implementar un paquete de iconos para algunos botones. Elegí el paquete de iconos *FontAwesome 6* debido a su fácil inclusión con `DearImGui`. Una vez terminada la base de la ventana de bienvenida, me dediqué a añadir funcionalidad secundaria a todos los pequeños elementos que se encontraban en esa ventana, como los botones de cada uno de los proyectos o los botones de añadir y eliminar proyecto.

Compaginé el desarrollo de estos elementos con el inicio de la redacción del estado de la cuestión de esta memoria, por lo que, tuve que buscar y leer algunos textos y artículos sobre los juegos de rol y, especialmente, sobre los videojuegos de rol; y pude reutilizar lo que mis compañeros investigaron sobre motores de videojuegos, y lo que investigué sobre editores para la redacción de la parte del desarrollo de videojuegos.

Más tarde, tanto Miguel como Alejandro y yo decidimos en una reunión cuáles iban a ser los elementos imprescindibles que el editor tendría que tener en la *build*

final, y se decidió por tener un editor de mapas, un editor de objetos, un editor de eventos y un editor de *sprites* y animaciones. Después de esta decisión, *refactoricé* parte del código incorporado para hacerlo más flexible con los nuevos cambios, y comencé con el desarrollo de la ventana principal, `MainWindow` y del editor de mapas, `MapEditor`.

El grueso del desarrollo del editor se centró en el editor de mapas, añadiendo cada funcionalidad y arreglando cada error que me iba encontrando por el camino. A parte de la estructura principal del editor de mapas, desarrollé los asistentes de creación (o *wizards*) de los mapas y de los *tilesets*. Una vez que la base de los recursos, desarrollada por Miguel, estaba terminada, pude conectarlos con los asistentes ya desarrollados y con el propio editor de mapas, que los necesitaría para poder guardar y modificar.

Una vez acabado el editor de mapas, me centré en la redacción de la memoria, comenzando con los capítulos de diseño e implementación, y añadiendo más contenido al estado de la cuestión, que a estas alturas se dividiría en dos capítulos (*Juegos de rol* y *Videojuegos de rol*).

Después, volví a compaginar la redacción de la memoria con el desarrollo de una parte del editor, en este caso, el editor de *sprites* y animaciones, así como los asistentes de creación pertinentes de ambas herramientas. Esta, sin contar el arreglo de algunos errores que me encontré, sería mi última aportación al desarrollo de código en el proyecto.

Por último, me dediqué a realizar pruebas de usuarios, seguir y acabar con la redacción de la memoria, y arreglar errores que se habían encontrado en el código durante las mencionadas pruebas.

Bibliografía

*En un lugar de La Mancha
de cuyo nombre «sí quiero»
acordarme... un caballero
que antaño «enganchó y engancha»
... dejaba pasar los días
leyendo constantemente
libros de caballerías
sin dormir lo suficiente.*

Valeriano Belmonte

ADDICT, C. Revisiting: The Game of Dungeons (1975). <https://crpgaddict.blogspot.com/2019/01/revisiting-game-of-dungeons-1975.html>, 2019. Imagen de la interfaz de *dnd* del blog, accedido el 17-04-2025.

A.I. DESIGN. *Rogue: Exploring the Dungeons of Doom*. Commodore Amiga, Amstrad CPC, Atari 8-bit, ZX Spectrum, y PC. Epyx. 1980.

ANH, T. y NGUYÊN, L. T. H. Seeking Solace in the Fantastical Adventures of Dungeons and Dragons. <https://shorturl.at/kktyD>, 2021. Imagen del tablero de juego de *Dungeons & Dragons* accedida desde la web Saigoneer el día 18-05-2025.

ANSELL, B., HALLIWELL, R. y PRIESTLY, R. *Warhammer Fantasy Battle*. Games Workshop. 1983.

APPERLEY, T. Genre and game studies: Toward a critical approach to video game genres. *Simulation & Gaming - Simulat Gaming*, vol. 37, páginas 6–23, 2006.

ASCII. *RPG Maker*. 1992.

BARDER, O. «Breath Of The Wild» Is Not The Best «Zelda» Game Ever Made. <https://shorturl.at/QZUM0>, 2017. Imagen de *The Legend of Zelda: Breath of the Wild* del artículo publicado en la revista *Forbes*, accedido el 17-04-2025.

BARTON, M. *Dungeons and Desktops: The History of Computer Role-Playing Games*. EBL-Schweitzer. CRC Press, 2008. ISBN 9781439865248.

BETHESDA GAME STUDIOS. *The Elder Scrolls III: Morrowind*. PC y Xbox. Bethesda Softworks. 2002.

- BETHESDA GAME STUDIOS. *The Elder Scrolls V: Skyrim*. PC, PlayStation 3, Xbox 360. Bethesda Softworks. 2011.
- BIOWARE. *Baldur's Gate*. PC. Black Isle Studios. 1998.
- BLIZZARD NORTH. *Diablo*. PC, PlayStation. Blizzard Entertainment. 1997.
- JASON BULMAHN. *Pathfinder*. Paizo Publishing. 2009.
- CD PROJEKT RED. *The Witcher 3: Wild Hunt*. PlayStation 4, Xbox One, PC. CD Projekt. 2015.
- CENANCE, M. *RPG Maker MV* Event Editor. https://rpgmaker.fandom.com/wiki/Event?file=RPG_Maker_MV_Event_Editor.png, 2019. Imagen del editor de eventos de *RPG Maker MV* accedida desde la *wiki* de *RPG Maker* el día 29-04-2025.
- CHUNSOFT. *Dragon Quest*. Nintendo Entertainment System. Enix. 1986.
- CLARKE, R. I., LEE, J. H. y CLARK, N. Why Video Game Genres Fail: A Classificatory Analysis. 2015.
- CORE DESIGN. *Tomb Raider*. Sega Saturn, PC y PlayStation. Eidos Interactive. 1996.
- CRAWFORD, C. The Art of Computer Game Design. 1984.
- EDWARDS, R. GNS and Other Matters of Role-Playing Theory. <http://www.indie-rpgs.com/articles/1/>, 2001. Accedido el día 17 de mayo de 2025.
- ESPOSITO, N. A Short and Simple Definition of What a Videogame Is. 2005.
- FINE, G. A. *Shared Fantasy: Role-Playing Games as Social Worlds*. University of Chicago Press, 1983.
- TOBY FOX. *Undertale*. PC, PlayStation 4, PlayStation Vita, Xbox One y Nintendo Switch. Toby Fox. 2015.
- GAME FREAK. *Pokémon*. Nintendo Game Boy. Nintendo. 1996.
- GARCÍA, R. Amazing D&D Fantasy diorama tabletop terrain. <https://volomir.com/en/dungeonsanddragons-fantasy-diorama-tabletop-terrain/>, 2016. Imagen del diorama de Ryan Devoto extraída del blog de volomir el día 18-05-2025.
- GOTCHA GOTCHA GAMES. *RPG Maker MZ*. 2020. Imagen de la interfaz de *RPG Maker MZ* accedida desde https://store.steampowered.com/app/1096900/RPG_Maker_MZ/?l=spanish el día 29-04-2025.
- GREGORY, J. *Game Engine Architecture, Third Edition*. CRC Press, 2018. ISBN 9781351974288.

- GYGAX, E. G. y ARNESON, D. L. *Dungeons & Dragons*. Tactical Studies Rules. 1974.
- GYGAX, E. G. y PERREN, J. *Chainmail*. Tactical Studies Rules. 1971.
- HALLIWELL, R. F., PRIESTLY, R., DAVIS, G., BAMBRA, J. y GALLAGHER, P. *Warhammer Fantasy Roleplay*. Games Workshop. 1986.
- HEINEMANN, J. How I fell in love with *Kriegsspiel*. <https://kriegsspiel.org/how-i-fell-in-love-with-kriegsspiel/>, 2022. Imagen del tablero de juego de *Kriegsspiel* accedida desde la web oficial de la Sociedad Internacional de *Kriegsspiel* (*International Kriegsspiel Society*) el día 18-05-2025.
- HUNICKE, R., LEBLANC, M. y ZUBEK, R. MDA: A Formal Approach to Game Design and Game Research. *AAAI Workshop - Technical Report*, vol. 1, 2004.
- ID SOFTWARE. *Doom*. PC. id Software. 1993.
- ID SOFTWARE. *Quake*. PC. GT Interactive. 1996.
- INTERPLAY PRODUCTIONS. *Fallout: A Post Nuclear Role Playing Game*. PC. Interplay Productions. 1997.
- KONAMI. *Suikoden*. PlayStation, PC y Sega Saturn. Konami. 1995.
- LEMON64. Ultima I: The First Age of Darkness - Commodore 64 Game - Download Disk/Tape - Lemon64. <https://www.lemon64.com/game/ultima-1>, 2002. Imagen de *Ultima I*, accedida desde la página web de *Lemon64* el día 18-04-2025.
- LINIETSKY, J. y MANZUR, A. Getting started with Visual Scripting. https://docs.godotengine.org/en/3.5/tutorials/scripting/visual_script/getting_started.html, 2022. Imagen del editor visual de nodos de *Godot* 3.2, accedida desde la documentación oficial de *Godot* el día 29-04-2025.
- LORTZ, S. L. Role-Playing. *Different Worlds*, (1), páginas 36–41, 1979.
- LUCASARTS. *Grim Fandango*. LucasArts. 1998.
- LUCASFILM GAMES. *Maniac Mansion*. Lucasfilm Games. 1987.
- MONTOLA, M. y STENROS, J. *Nordic Larp*. Fëa Livia, Stockholm, Sweden, 2010. ISBN 9789163378560.
- NINTENDO. *Excitebike*. Nintendo Entertainment System. Nintendo. 1984.
- NINTENDO. *Super Mario Bros.*. Nintendo Entertainment System. Nintendo. 1985.
- NINTENDO. *The Legend of Zelda: Breath of the Wild*. Nintendo Wii U y Nintendo Switch. Nintendo. 2017.
- ORIGIN SYSTEMS. *Ultima I: The First Age of Darkness*. PC, Atari 8-bit y Commodore 64. California Pacific Computer. 1981.

- PADDINGTON, L. A Matter of Honor - a Warhammer Fantasy Battle Report. <https://frontlinegaming.org/2023/02/03/a-matter-of-honor-a-warhammer-fantasy-battle-report/>, 2023. Imagen de una partida de *Warhammer Fantasy Battle* extraída de la web *Frontline Gaming* el día 18-05-2025.
- PAIGE, J. y FISHER, B. *Official Hamster Republic Role Playing Game Construction Engine*. 1998.
- PERRIN, S. H., TURNER, R., HENDERSON, S. y JAMES, W. *RuneQuest*. Avalon Hill. 1983.
- PETERSEN, C. S. J. y WILLIS, L. *Call of Cthulhu*. Chaosium. 1981.
- PHILLIPS, A. y DAVIS, M. Tags for Identifying Languages. RFC 5646, 2009.
- POHJOLA, M. Dragonbane Memories. <https://nordiclarp.org/2021/08/18/dragonbane-memories/>, 2021. Imagen de una partida de LARP nórdico extraída de la web *Nordic LARP* el día 18-05-2025.
- REIN-HAGEN, M. y DAVIS, G. *Mind's Eye Theatre*. White Wolf. 1993.
- VON REISSWITZ, G. L. y VON REISSWITZ, G. H. R. J. *Kriegsspiel*. 1812.
- RONCE, S., SLOMOWSKI, A. y CHYLO, D. *RPG JS*. 2020.
- SCHELL, J. *The Art of Game Design: A Book of Lenses, Third Edition*. CRC Press, 2019. ISBN 9781351803632.
- SIR-TECH SOFTWARE. *Wizardry: Proving Grounds of the Mad Overlord*. PC. Sir-Tech Software. 1981.
- SOLOMON, C. *Dungeons & Dragons* (película). 2000. Silver Pictures y New Line Cinema.
- SQUARESOFT. *Chrono Trigger*. Super Nintendo Entertainment System. SquareSoft. 1995.
- SQUARESOFT. *Final Fantasy*. Nintendo Entertainment System. SquareSoft. 1987.
- SQUARESOFT. *Final Fantasy VII*. PlayStation. SquareSoft. 1997.
- SQUARESOFT. *Secret of Mana*. Super Nintendo Entertainment System. SquareSoft. 1993.
- TEKINBAS, K. y ZIMMERMAN, E. *Rules of Play: Game Design Fundamentals*. ITPro collection. MIT Press, 2003. ISBN 9780262240451.
- THEDARKB. Rogue Screenshot.png. <https://commons.wikimedia.org/w/index.php?curid=100192403>, 2021. Imagen de *Rogue* accedida desde la entrada en Wikipedia del juego ([https://en.wikipedia.org/wiki/Rogue_\(video_game\)](https://en.wikipedia.org/wiki/Rogue_(video_game))), accedido el 18-04-2025.

- TYCHSEN, A., HITCHENS, M., BROLUND, T. y KAVAKLI, M. Live Action Role-Playing Games: Control, Communication, Storytelling, and MMORPG Similarities. *Games and Culture*, vol. 1(3), páginas 252–275, 2006. ISSN 1555-4120.
- VALVE. *Half-Life*. PC y PlayStation 2. Sierra Studios. 1998.
- WADSTEIN, M. Tu primera hora con Unreal Engine 5.2. <https://dev.epicgames.com/community/learning/courses/Gee/tu-primera-hora-con-unreal-engine-5-2/8Bq8/introduccion-a-em-tu-primera-hora-con-unreal-engine-5-2-em>, 2023. Imagen de la interfaz de *Unreal Engine* 5, accedida desde el curso de *Unreal Engine* de Epic Games el día 29-04-2025.
- WANO. *RPG Paper Maker*. 2018.
- WELLS, H. G. *Little Wars*. Frank Palmer, 1913.
- WHISENHUNT, G. y WOOD, R. *dnd*. PLATO. 1975.
- WILLIAMS, A. *History of Digital Games: Developments in Art, Design and Interaction*. CRC Press, 2017. ISBN 9781317503811.

Apéndice A

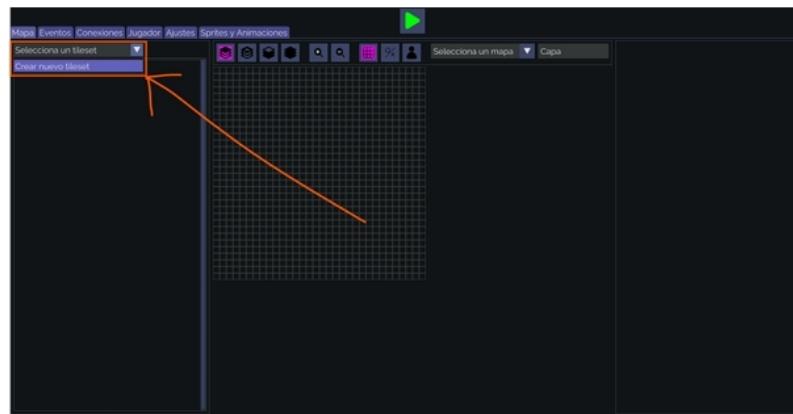
Guía de pruebas de usuario

Tutorial:

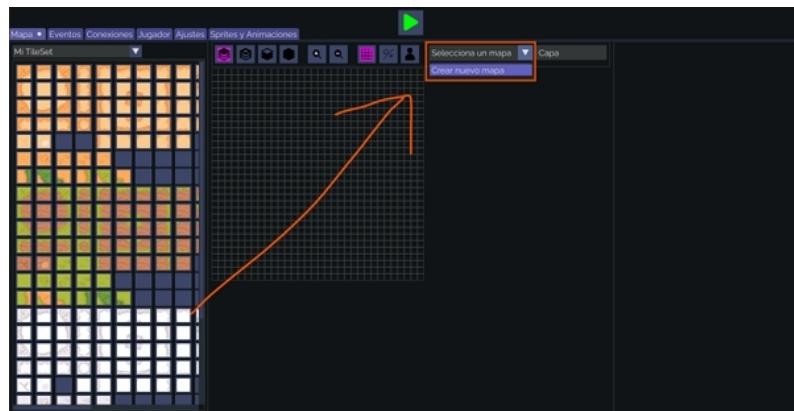
1. *Crear un proyecto*: lo primero que tendrás que hacer es crear un proyecto. Haz clic en «Nuevo proyecto». Dale un nombre, una localización, acepta y selecciónalo.



2. *Crear un mapa*: una vez abierto el proyecto estarás en la pestaña de edición de mapas. Aquí es donde establecerás el aspecto y comportamiento de cada zona. Lo primero que queremos es poder dibujar el escenario, por lo que necesitaremos un *tileset*. En la parte izquierda de la ventana tienes el selector de *tilesets*; abre el desplegable y selecciona «Crear un nuevo tileset». Se desplegará una nueva ventana en la que podrás escoger el nombre de tu *tileset*, qué imagen se usará y otras configuraciones. Una vez creado, lo podrás seleccionar.



¡Ya tienes tu paleta! Ahora, vas a intentar conseguir un lienzo: crea un nuevo mapa. En la zona central superior de la ventana está el desplegable de selección de mapa; igual que con el de *tileset*, despliégalos y crea un nuevo mapa. Podrás definir su nombre, su tamaño y su cantidad de capas.



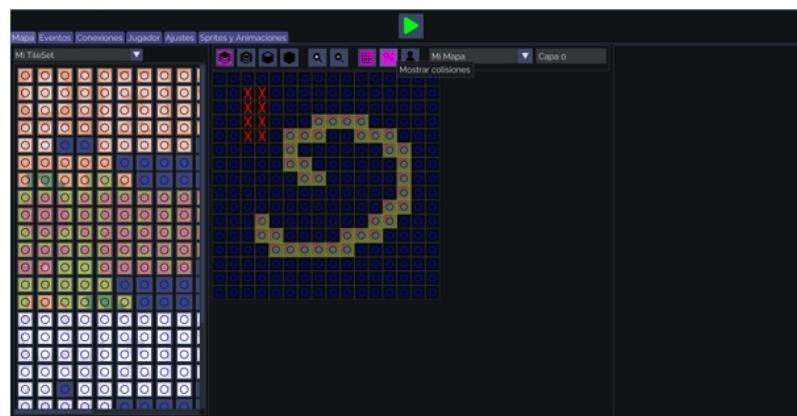
Una vez que tengas el *tileset* y el mapa creados, ¡es la hora de dibujar! Selecciona tu *tileset* y tu mapa, escoge qué casilla del *tileset* quieras usar y empieza a dibujar el mapa. Si ejecutas ahora el juego podrás ver una aplicación que muestra tu mapa.



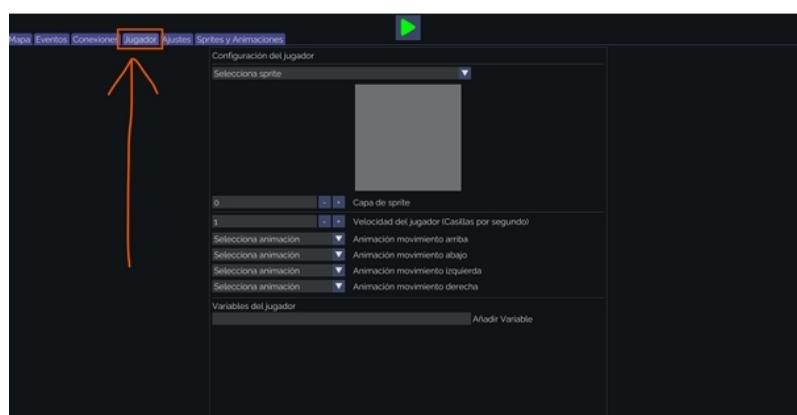
3. *Colisiones*: a continuación, vamos a agregar las colisiones a nuestro mapa. Para ello puedes utilizar el botón de «Mostrar colisiones».



Una vez dentro de este modo, podrás modificar las colisiones del *tileset* o del propio mapa. Cada vez que añadas un *tile*, este combinará sus colisiones con las del resto de *tiles* de esa casilla en el mapa, de forma que con que únicamente uno de ellos no permita el paso se añadirá un bloqueo en esa casilla. Si modificas directamente el valor de colisión de una casilla del mapa esto sobreescibirá esa combinación de las colisiones de los *tiles*. Con esto podrás generar entornos más dinámicos y controlar las zonas por las que quieras que pueda avanzar el jugador.



4. *Establece tu personaje principal:* arriba podrás ver una pestaña en la que pone «Jugador». Si la seleccionas cambiará la ventana. Aquí podrás escoger el aspecto del personaje. Una vez establecido volverás a probar la aplicación con tu personaje moviéndose en el mapa.



5. *¡Dale vida a tu mapa! Crea objetos:* si vuelves a la pestaña del mapa, podrás ver en la parte superior el siguiente ícono:



Si lo seleccionas, podrás ver que una casilla del mapa resalta en verde. Escoge la que quieras y clica el nuevo botón «Crear objeto» que hay ahora en la parte derecha de la ventana. Ahora, en esa misma zona, podrás editar tu objeto, cambiarle la imagen y la posición. Si vuelves a probar tu aplicación, verás que ahí está; pero no hace mucho, porque para eso tendremos que añadirle eventos.



6. *Eventos*: en la parte superior del editor puedes escoger la ventana «Eventos». Una vez entras por primera vez verás que está vacía, con un solo desplegable de selección. Ábrelo y crea un nuevo evento. Una vez creado verás dos partes diferenciadas en la ventana: «Condición» y «Comportamientos». La condición es el criterio bajo el que quieras que se active ese evento. Puedes escoger el que quieras. Los comportamientos son todo aquello que ocurrirá una vez se active el evento. Puedes añadir tantos como quieras del tipo que quieras, y se aplicarán uno detrás de otro en el orden que los establezcas. Puedes moverlos en la lista o eliminarlos cuando quieras.



Una vez hayas definido tu evento puedes volver a la pestaña del mapa y añadir el evento creado a tu objeto. Si pruebas la aplicación una vez más, verás cómo el objeto ejecuta el evento cuando se cumple la condición.

7. *Conexiones entre mapas*: a continuación, crea un segundo mapa. Puede ser más pequeño y sencillo. Una vez lo tengas listo, guárdalo (CTRL-S) y ve a la sección de conexiones entre mapas. Una vez allí podrás añadir varios mapas a un mundo. Puedes conectar los mapas que estén en un mismo mundo desplazándolos en la ventana; dos mapas que sean adyacentes se considerarán conectados y podrás pasar de uno a otro de forma natural en tu juego.



8. *Otras configuraciones*: finalmente, podrás configurar aspectos generales de tu juego, como el tipo de fuente, el color y el tamaño del texto; el color de fondo de las cajas de texto; la cantidad de casillas que aparecerán en cámara; volúmenes de sonidos... Establece todos estos parámetros como tú prefieras y dale a *play* para ver la versión definitiva de tu juego en acción.



